# From Natural Language to Verified Code: Toward AI Assisted Problem-to-Code Generation with Dafny-Based Formal Verification

Md Erfan , Md Kamal Hossain Chowdhury , Ahmed Ryan , Md Rayhanur Rahman *

Posted Date: 24 April 2026

doi: 10.20944/preprints202604.1772.v1

Keywords: formal verification; dafny; program synthesis; software correctness; uDebug

 doi:10.20944/preprints202604.1772.v1



*Article*

# From Natural Language to Verified Code: Toward AI Assisted Problem-to-Code Generation with Dafny-Based Formal Verification

**Md Erfan [1], Md Kamal Hossain Chowdhury [2], Ahmed Ryan [1] and Md Rayhanur Rahman [1,*]**

[1] Department of Computer Science, The University of Alabama, Tuscaloosa, USA
[2] Water Institute, The University of Alabama, Tuscaloosa, USA
* Correspondence: mrahman87@ua.edu

**Abstract**

Large Language Models (LLMs) show promise in automated software engineering, yet their guarantee of correctness is frequently undermined by erroneous or hallucinated code. To enforce model honesty, formal verification requires LLMs to synthesize implementation logic alongside formal specifications that are subsequently proven correct by a mathematical verifier. However, the transition from informal natural language to precise formal specification remains an arduous task. Our work addresses this by providing the NaturalLanguage2VerifiedCode (NL2VC)-60 dataset: a collection of 60 complex algorithmic problems. We evaluate 11 randomly selected problem sets across seven open-weight LLMs using a tiered prompting strategy: contextless prompts, signature prompts providing structural anchors, and self-healing prompts utilizing iterative feedback from the Dafny verifier. To address vacuous verification, where models satisfy verifiers with trivial specifications, we integrate the uDebug platform to ensure functional validation. Our results show that while contextless prompting leads to near-universal failure, structural signatures and iterative self-healing facilitate a dramatic performance turnaround. Specifically, Gemma 4-31B achieved a 90.91% verification success rate, while GPT-OSS 120B rose from zero to 81.82% success with signature-guided feedback. These findings indicate that formal verification is now attainable for open-weight LLMs, which serve as effective apprentices for synthesizing complex annotations and facilitating high-assurance software development.

**Keywords:** formal verification; dafny; program synthesis; software correctness; uDebug

## 1. Introduction

Formal specifications are indispensable for rigorously defining program logic and facilitating automated reasoning about software correctness. Formal specifications transform ambiguous behaviors into precise semantics, creating a framework for quality assurance through procedure contracts, loop invariants, and assertions. As a result of this mathematical clarity, these specifications have become essential across diverse tasks ranging from software testing [1] and model checking [2] to full-scale program verification.

Formal verification is increasingly adopted to develop high-assurance software by providing mathematical proof that programs strictly satisfy their specifications [3] than traditional dynamic analysis methods such as testing or fuzzing [4]. Consequently, high-stakes domains, including security-sensitive infrastructure, cryptographic libraries, and autonomous aerospace systems, rely on these rigorous correctness guarantees to prevent critical vulnerabilities and operational disruptions [5]. Despite significant breakthroughs in Satisfiability Modulo Theory (SMT) solvers [6], writing program properties and proofs remains a creative, manual process requiring immense expertise. Developers must manually generate complex annotations, such as loop invariants and ranking functions, to enable automated verification tools to complete a proof. This manual process is tedious and time-consuming, often exceeding the effort required to write the executable code itself. For example, the verification of

 doi:10.20944/preprints202604.1772.v1



the seL4 microkernel project [7] required an eleven-person-year effort [8], while the verified code for CompCert C [9] is more than three times the size of the compiler itself [10].

Concurrent with the evolution of formal methods, the rise of Large Language Model (LLM) based assistants has rapidly transformed modern software development workflows. AI-driven tools such as GitHub Copilot [**?** ], Cursor AI [11], and Amazon Q Developer [12] have accelerated programming tasks through natural language to code translation and intelligent autocompletion. By leveraging massive corpora of source code, these systems synthesize complex snippets from informal descriptions to automate traditional refactoring and implementation workflows. This shift has popularized vibe coding [**?** ], where developers rely on high-level intuition and conversational prompts to iterate on software rather than manual line-by-line implementation [13]. However, the use of LLMs in software synthesis introduces a new set of reliability concerns because, despite their fluency, these systems frequently produce code that is syntactically plausible but semantically incorrect, a phenomenon known as hallucinations. Furthermore, LLMs often exhibit insufficient reasoning capabilities when dealing with complex algorithmic logic and remain susceptible to generating insecure patterns due to adversarial poisoning or inherent biases in their training data [14]. Consequently, the need to ensure the correctness and logical integrity of LLM-generated code has emerged as a fundamental challenge in the software engineering community. The problems necessitate a bridge between the synthesis of AI and the rigorous mathematical certainty of formal verification.

Furthermore, software requirements are typically written in natural language, which is often ambiguous and imprecise. Capturing complete specifications from such requirements is difficult, and there is currently a lack of direct mapping from natural language to formal specifications. Several formal verification languages exist in the literature, such as F*, Coq, Lean, and the Java Modeling Language (JML); however, we choose Dafny [15] for our framework due to its unique balance of imperative programming and automated theorem proving. Dafny supports verification via *Design by Contract* [16] using assertions, preconditions, and postconditions. However, even in Dafny, authoring formal specifications and auxiliary verification assertions remains difficult for developers [17]. This challenge is exacerbated by a limited number of training data; while popular languages like Java and Python have over 5 million repositories on GitHub, the Dafny ecosystem has approximately 779 results [18]. This lack of large-scale data makes Dafny particularly challenging for LLM to synthesize correct code without producing hallucinations.

To bridge these gaps, we introduce **NL2VC-60**: **N**atural **L**anguage **2** **V**erified **C**ode dataset, a novel benchmark designed to evaluate the synthesis of formally verified code from complex, real-world requirements. We began by hand-authoring 60 high-quality Dafny programs to optimize our prompt generation strategies, specifically targeting the nuanced demands of competitive programming problems from the UVa Online Judge [19]. The UVa Online Judge is an online automated judging system for programming problems, hosted by the University of Valladolid. Using this foundation, we evaluate randomly selected 11 distinct problem sets across seven leading open-source LLMs using three specialized prompting techniques. Existing work primarily focuses on small-scale, textbook-style problems supported by limited Dafny datasets and natural language inputs that rarely exceed 50 words. In contrast, our problem set overcomes these constraints by incorporating complex algorithmic challenges that require significantly more detailed specifications and extensive descriptive contexts. Our analysis of the resulting code led to the creation of the first comprehensive dataset of Dafny-specific compilation and verification errors in the literature, providing a unique resource for understanding model failure modes in formal methods. We are the first to integrate uDebug [20] community test suites to ensure rigorous functional correctness. uDebug is a community-driven platform designed for competitive programmers to validate their solutions against high-quality test suites. By combining community-driven testing with SMT-based formal proof, NL2VC-60 offers a new standard for code generation that balances complex natural language requirements with mathematical certainty.

*The goal of this paper is to advance the frontier of AI-assisted NL problem-to-code generation by establishing a robust, Dafny-based formal verification framework that evaluates open-weight LLMs' ability to translate*

 doi:10.20944/preprints202604.1772.v1



*requirements into provably correct and functionally accurate software.*
The primary contributions of this work are as follows:

- We conduct the first comprehensive empirical study of open-source LLMs synthesizing verifiable Dafny code from real-world requirements, evaluating seven LLM models across three prompting strategies to establish baseline performance in formal code generation.
- We introduce NL2VC-60 dataset, a novel benchmark consisting of 60 hand-authored programs to bridges the gap between simple textbook tasks and the nuanced demands of competitive programming tasks.
- We provide the first categorization of Dafny-specific compilation and verification errors in the literature, creating a diagnostic dataset of model failure modes to guide future improvements in the synthesis of formal verification.
- We establish a rigorous evaluation pipeline for functional correctness by being the first to integrate extensive uDebug community test suites, ensuring synthesized programs are both formally verified and correct across thousands of real-world edge cases.

Ultimately, the synthesis of verified methods remains a vast problem space, and this paper serves as an initial exploration of its potential. While our first two contributions address the end-to-end task of synthesizing code from narrative prompts, our third contribution, the systematic categorization of compilation and verification errors, highlights that Large Language Models (LLMs) may be most effective when tackling specific sub-problems. These include generating formal specifications from natural language, synthesizing imperative code from existing contracts, or focusing exclusively on the annotation bottleneck by producing loop invariants and termination conditions.

Our study results suggest that efforts in LLM-assisted coding should concentrate on generating verifiable programs, and that the combination of open-source models with formal verification techniques provides a cost-effective path toward high-assurance software development. Modern software development requires handling real-world requirements, yet existing literature focuses on small, textbook-style programming and algorithmic problems. However, our study relies on different algorithmic patterns, which fully reflect real-world software requirements. We can reduce this gap; by curating a more representative set of problems from the UVa Online Judge.

The remainder of this paper is organized as follows. Section 2 provides a motivational example. Section 3 establishes the background on the Dafny language, open-weight LLMs, and the uDebug platform. Section 4 reviews relevant literature and existing benchmarks. Section 5 details our methodology, and Section 6 presents our experimental results and a detailed analysis of model failure modes. Section 7 discusses the implications of our findings and potential threats to validity in Section 8. Finally, Section 9 concludes the paper.

## 2. Motivational Example

We consider the *Magic Formula* problem (UVa 11934) [21] to illustrate the gap between traditional competitive programming and formal verification. The task requires counting how many values of a quadratic function $f(x) = ax^2 + bx + c$ are divisible by a divisor $d$ within the range $0 \leq x \leq L$. In a typical software engineering workflow, a developer might rely on sample test cases to verify their logic. However, such an approach is prone to off-by-one errors and boundary case failures that testing alone may not catch. By contrast, formal verification ensures that the counting logic remains correct across all possible integer inputs within the specified bounds, transforming a fragile test-based heuristic into a mathematically guaranteed solution.

```
function Power(x:int, n:int): int
  requires n >= 0
  decreases n
{
  if n == 0 then 1 else x * Power(x, n-1)
}
```

 doi:10.20944/preprints202604.1772.v1



```
method MagicFormula(a:int, b:int, c:int, d:int, l:int) returns (result:int)
  requires d > 0 && l >= 0
  // Formal Specification: The result must match the cardinality of the set
  ensures result == |set x:int | 0 <= x <= l && (a*Power(x,2) + b*x + c) % d
    == 0|
  // Boundary Case: Constant functions
  ensures (a == 0 && b == 0 && c % d == 0) ==> result == l + 1
{
    var x, count := 0, 0;
    while x <= l
      invariant 0 <= x <= l + 1
      decreases l - x
    {
        var value := (a*Power(x,2) + (b * x) + c);
        if value % d == 0 { count := count + 1; }
        x := x + 1;
    }
    result := count;
}
```

Listing 1: Dafny implementation (Magic Formula problem)

Our research proposes a shift from testing-based validation to formal synthesis. As shown in Listing 1, the Dafny implementation goes beyond the imperative logic of the loop by defining a formal *contract*. The `ensures` clause specifies the ground truth using mathematical set cardinality:

$$result = |\{x \in \mathbb{Z} \mid 0 \leq x \leq L \wedge (ax^2 + bx + c) \equiv 0 \pmod{d}\}| \tag{1}$$

This example motivates our work: by utilizing LLMs to generate both imperative code and associated formal specifications, we can leverage SMT solvers to provide a mathematical guarantee of correctness. This approach effectively eliminates common algorithmic bugs that persist even after extensive testing on platforms like UVa Online Judge [22].

## 3. Background

This section provides the theoretical and technical foundations for contextualizing our study of AI-assisted formal verification. We first discuss the unique architecture of the Dafny language and the inherent challenges of the specification burden. Then we discuss the current state of open-weight LLMs and the iterative prompt engineering techniques used to optimize their reasoning. Finally, we introduce uDebug as a validation layer to ensure that formally verified programs remain functionally robust under real-world test cases.

### *3.1. Dafny: A Verification-Aware Programming Language*

Dafny [15,23,24] is a verification-aware, statically typed programming language originally developed at Microsoft Research [25] and currently supported by the Amazon Automated Reasoning group [26]. Dafny bridges the gap between high-level programming paradigms, including imperative, functional, and object-oriented styles, and formal mathematical proof. A distinguishing feature of Dafny is its native support for *Design by Contract*, employing Floyd-Hoare-style [16] verification using preconditions (`requires`), postconditions (`ensures`), and loop invariants.

To develop a verified program, developers provide formal specifications along with executable code. The Dafny static program verifier then checks the functional correctness of the implementation against these specifications. This is achieved by transforming the code into an intermediate verification representation (Boogie) [27], encoding the conditions into predicate calculus, and invoking the Z3

 doi:10.20944/preprints202604.1772.v1



SMT solver [28] to prove their validity. In recent years, Dafny has been used by industry leaders like Amazon to verify AWS authorization logic [29] and by Intel for hardware encryption libraries [30].

While Dafny ensures that the code does what the developer specifies, the difficulty lies in the specification burden. As illustrated in Listing 1, a simple method to find the value of a quadratic function may require more lines of formal annotations (preconditions, postconditions, and invariants) than actual executable code. Researchers have observed that writing these auxiliary verification annotations remains the primary bottleneck in formal software development [17]. If an LLM can successfully synthesize both the implementation and the proofs required for verification, the code generation could lower the barrier for high-assurance software engineering.

### *3.2. Large Language Models and Open-Weight Models*

The landscape of Large Language Models (LLMs) has shifted from a dominance of proprietary APIs (like GPT-4 and PaLM-2) toward highly capable **Open-Weight Models** [31]. Unlike closed models, open-weight models [31] such as **Llama 3.3** [32], **Qwen 3** [33], **Gemma 3** [34], and **Gemma 4** [35] allow researchers to host the models locally, providing full control over parameters, token limits, and data privacy.

Recent advancements in these models have demonstrated that smaller, specialized architectures (e.g., **Qwen 3 Coder** [36]) can rival proprietary models in code generation and logical reasoning tasks. However, applying these open-weight models to verification-aware languages such as Dafny remains an underexplored frontier. Because Dafny code is scarce in public training datasets compared to Python or Java, our research explores whether the reasoning capabilities inherent in modern open-weight architectures can generalize to the syntactic and logical requirements of formal verification.

### *3.3. Prompt Engineering and Self-Healing*

Prompt engineering [37,38] is the systematic process of crafting inputs to align an LLMs output with a specific technical task [39]. In the context of Dafny, prompts must be engineered to be unambiguous and structured. We employ a tiered approach: starting from *Contextless Prompting* to establish a baseline, moving to *Signature Prompting* to provide structural anchors, and finally utilizing *Self-Healing Prompting*. Self-healing [40] mimics the human developer's workflow by feeding the Dafny verifier's error messages back into the LLM, allowing the model to iteratively repair its logic and specifications until verification is achieved.

### *3.4. uDebug: Beyond Vacuous Verification*

A critical challenge in LLM-driven formal synthesis is vacuous verification, where a model generates weak or trivial specifications that pass the Dafny verifier but do not actually solve the intended problem. We integrate **uDebug** [20] into our evaluation pipeline to mitigate the problem. uDebug is a community-driven platform designed for competitive programmers to validate their solutions against high-quality test suites. By providing an accepted output for given inputs, uDebug allows us to perform a dual-layer validation:

- **Formal Layer:** The Dafny verifier proves that the code is logically consistent with its formal specifications.
- **Functional Layer:** uDebug ensures the code is semantically correct by testing the generated code against extreme edge cases and boundary conditions contributed by the competitive programming community.

As noted by Professor Miguel Angel Revilla (Creator of UVa Online Judge), uDebug is a perfect complement for identifying critical inputs that break solutions [20]. In this research, we use uDebug to confirm that our synthesized, verified Dafny programs are also functionally robust in real-world scenarios type requirements.

 doi:10.20944/preprints202604.1772.v1



## 4. Summary of the Literature

This section contextualizes our research within the broader landscape of automated software engineering and formal verification. We review existing efforts in program synthesis, evaluate the evolving role of LLMs in formal methods, and compare current Dafny-centric benchmarks.

### *4.1. Program Synthesis and Verification with Dafny*

In the last two decades, formal methods for software synthesis [41] and verification [42] have transitioned from esoteric research topics to practical industrial tools [15,43]. Modern tools like Dafny, SAW, and SPIN are now mature enough to support critical applications in encryption algorithms [44], Ethereum Virtual Machine (EVM) bytecodes [45], scientific software, and quantum circuitry [46].

Despite this maturity, a barrier to adoption remains the scarcity of engineers trained in formal specification [47]. The gap has spurred research into automated support for Dafny, such as XDsmith for differential testing [48] and techniques for generating counterexamples when verifiers fail [49]. Our work builds on this momentum by exploring how LLMs can bridge the gap between Natural Language (NL) requirements and verified Dafny code.

### *4.2. LLMs for Formal Methods and Software Engineering*

LLMs have been increasingly applied to automated proof synthesis and theorem proving, with models like Baldur [50] and Thor [51] outperforming traditional heuristic tools like CoqHammer [52]. Beyond proofs, researchers have utilized LLMs to translate natural language into Isabelle/HOL [53] and event graphs [54]. While general-purpose models like GPT-4 often struggle with algorithmic reasoning [55], specialized models such as Minerva [56,57] demonstrate that domain-specific pre-training can mitigate these limitations.

In the broader software engineering context, LLMs now support code completion, repair, and test generation [58,59]. Our research follows the philosophy of letting LLMs generate plausible candidates while leveraging the Dafny verifier to guarantee correctness, effectively filtering out the hallucinations common in LLM-generated code.

### *4.3. Benchmarking Dafny Generation*

Existing literature on LLM-based Dafny generation remains relatively limited in both dataset scale, limited word length for problems, and problem diversity. Prior studies have primarily evaluated models ranging from GPT-3.5 to the recent Llama 3.3 [32], Qwen 3 [33], and Gemma 3 [60] on a narrow set of benchmarks. Table 1 provides a comparative overview with limitations of existing literature.

**Table 1.** Comparison of Dafny Verification Datasets and Benchmarks.

| Work / Dataset | Input Type | Problem Type | Dafny | Size | Avg Code | Limitations |
|---|---|---|---|---|---|---|
| Clover [61] | Short NL / Ann. | Textbook | Yes | 63–66 | ∼19 LoC | Simple problems, small scale |
| MBPP-Dafny [62] | NL (short) | Basic Python | Yes | 164 | ∼19 LoC | Entry-level tasks only |
| HumanEval-Dafny [63] | NL (short) | Algorithmic | Yes | 132 | ∼50 LoC | Still benchmark-style |
| DafnyBench [64] | Mixed | Real + Textbook | Yes | 782 | ∼53 LoC | Limited human-written verified programs |
| TacoDafny [65] | NL (Gen.) | Synthetic | Yes | Auto. | Varies | Synthetic, not real-world |
| ATLAS [66] | Alg. + Ref. | Algorithmic | Yes | Large | Varies | No direct NL to Dafny |
| SpecGen [67] | NL | LeetCode | No | N/A | N/A | Uses OpenJML, not Dafny |

 doi:10.20944/preprints202604.1772.v1



As shown in the table, benchmarks like Clover [61] and MBPP-Dafny [62] focus on textbook-level tasks with average lengths of only on average 19 lines of code. DafnyBench [64] represents a greater effort with 782 samples, yet the research relies heavily on converted rather than native requirements. Besides that, existing work primarily focuses on small-scale, textbook-style problems supported by limited Dafny datasets and natural language inputs that rarely exceed 50 words. In contrast, our problem set overcomes these constraints by incorporating complex algorithmic challenges that require significantly more detailed specifications and extensive descriptive contexts. While frameworks like ATLAS [66] synthesize verified code using algorithmic references and test cases, our approach targets the direct synthesis of Dafny programs from NL descriptions, addressing the complexity of requirements.

## 5. Approach

This study aims to bridge the gap between textbook-style benchmarks and real-world software requirements by evaluating LLM performance on complex algorithmic tasks curated from the UVa Online Judge. We focus on the transition from NL requirements to formally verified Dafny code through tiered prompting and iterative repair.

### *5.1. Research Questions*

We investigate the capability of LLMs to synthesize formally verified Dafny methods. We employ a tiered evaluation strategy to isolate the impact of structural hints and iterative feedback. We address the following research questions:

**RQ1 - (Contextless Prompting):** How effective are LLMs at synthesizing fully verified Dafny methods when provided only with a natural language description, without any formal structural hints?

**RQ2 - (Signature Prompting):** How does the provision of a formal method signature and accompanying functional tests affect the initial synthesis success rate compared to contextless prompting?

**RQ3 - (Self-Healing Capabilities):** To what extent can an iterative feedback loop recover failed synthesis attempts under varying initial conditions?

> **RQ3a (Self-Healing with Contextless Prompting):** Can LLMs repair verification failures when the initial attempt was generated from NL alone?
> **RQ3b (Self-Healing with Method Signature):** Does the presence of a pre-defined method signature provide a superior result for the self-healing process, leading to higher repair rates than contextless healing?

**RQ4 - (Error Analysis ):** To what extent can error descriptions help to overcome errors by using the signature prompt and the self-healing method?

### *5.2. Problem Curation and Abstraction*

This subsection details the systematic process of converting competitive programming tasks into standardized formal requirements to ensure high-fidelity evaluation of LLM.

#### 5.2.1. Test Dataset

We conducted our study using a collection of problems with rich natural-language descriptions and corresponding formally verified Dafny code. Existing literature often relies on small-scale, textbook-style datasets like MBPP or HumanEval; however, these focus on basic programming tasks with short specifications. To evaluate LLMs on complex, real-world requirements, we curated 60 set of problems from the **UVa Online Judge**, an automated judging system with thousands of competitive programming problems [22].

 doi:10.20944/preprints202604.1772.v1



#### 5.2.2. Problems Generalization

Unlike benchmarks that use one-line descriptions, UVa problems provide paragraph-level specifications involving different programming tasks, with average word counts exceeding 179. However, these problems are typically encumbered by what we term *presentation flavor* details designed for contest environments (e.g., input formatting, multiple test case counts like $|t| < 15$, and arbitrary constraints like $|a| < 10^9$) that do not contribute to the semantic understanding of the requirement.

To adapt these for formal verification, we performed a **Generalization Process** in Table 2 to make the problems generic. We manually transformed the competitive programming descriptions into a more generic, requirement-focused form by removing presentation-specific instructions while preserving the core computational logic. For example, a problem asking to identify relational operators ("$>$", "$<$", or "$=$") between two integers was stripped of its "process $T$ lines" loop instructions and reduced to its functional essence.

**Table 2.** Comparison Between Original UVa Problem Description and Generalized Generic Description.

| Component | Original UVa Problem Description (Competitive Flavor) | Generic Description (Requirement Focused) |
|---|---|---|
| **Description** | Some operators checks about the relationship between two values and these operators are called relational operators. Given two numerical values your job is just to find out the relationship between them that is (i) First one is greater than the second (ii) First one is less than the second or (iii) First and second one is equal. | Some operators checks about the relationship between two values and these operators are called relational operators. Given two numerical values your job is just to find out the relationship between them that is (i) First one is greater than the second (ii) First one is less than the second or (iii) First and second one is equal. |
| **Input** | First line of the input file is an integer $t$ ($t < 15$) which denotes how many sets of inputs are there. Each of the next $t$ lines contain two integers $a$ and $b$ ($\|a\|, \|b\| < 1000000001$). | The input contain two integers $a$ and $b$. |
| **Output** | For each line of input produce one line of output. This line contains any one of the relational operators '$>$', '$<$' or '$=$', which indicates the relation that is appropriate for the given two numbers. | The output contains any one of the relational operators '$>$', '$<$' or '$=$', which indicates the relation that is appropriate for the given two numbers. |
| **Sample Input** | 3<br>10 20<br>20 10<br>10 10 | 10 20<br>20 10<br>10 10 |
| **Sample Output** | $<$<br>$>$<br>$=$ | $<$<br>$>$<br>$=$ |

#### 5.2.3. Empirical Problem Selection

We selected 60 problems based on their acceptance rates and user submission statistics to act as a proxy for practical relevance. Across the selected problems, total submissions varied from approximately 27,000 to over 370,000 in the online Judge platform, ensuring the tasks were neither trivial nor excessively niche.

For each generalized problem, we developed ground-truth Dafny verification code by all the authors to ensure a unified and satisfactory implementation standard. We utilized the **uDebug** platform to access a diverse collection of test cases, including edge and corner cases, to validate the functional correctness of our verified solutions. This resulted in our main test dataset, NL2VC-60: a collection of 60 real-world algorithmic problems, each consisting of a generic requirement description, a formal method signature, and a suite of validation test cases. Since no public Dafny implementations of UVa problems existed prior to this study, our dataset minimizes the risk of data leakage during LLM evaluation.

 doi:10.20944/preprints202604.1772.v1



### 5.3. Human Written Dataset: NL2VC-60

To perform *Dynamic Few-Shot Prompting*, we required a high-quality, diverse collection of verified Dafny methods to serve as in-context exemplars. Given the absence of existing datasets that map complex natural language requirements to Dafny, we manually developed a reference set, **NL2VC-60**, consisting of 60 problems from our suite.

This process involved translating the core computational requirements of 60 UVa problems into complete Dafny implementations. Unlike standard coding tasks, this required the manual formulation of formal specifications, including method preconditions (`requires`), postconditions (`ensures`), and complex loop invariants until the Dafny verifier could formally prove the code's correctness. We performed all necessary annotations, hint insertions, and structural refinements until the verifier signaled a successful proof for each method.

In developing this dataset, we experienced first-hand the significant cognitive load associated with formal specification. Formulating precise postconditions that capture the semantic intent of paragraph-level requirements and providing sufficient invariants for algorithmic logic proved to be a rigorous undertaking. The creation took approximately 300 person-hours for the authors and 50 more hours to resolve conflicts among authors to create this dataset of 60 formally verified problems, even with access to official Dafny documentation [68].

### 5.4. Functional Validation via uDebug

A significant limitation in existing formal synthesis literature is the reliance on simplified functional validation. Current benchmarks typically employ either a small set of basic input-output pairs [69] or rely on the LLM itself to generate test cases [65]. Such methods often fail to identify subtle semantic bugs because they rarely cover the complex edge and corner cases inherent to algorithmic problems.

To address this gap, we incorporate *uDebug* [20] into our validation pipeline for the first time in the context of Dafny synthesis. *uDebug* is a community-driven platform that provides extensive, high-quality test suites for competitive programming problems, specifically designed to uncover logical flaws through extreme inputs and boundary conditions.

We ensure a dual-layer validation process by using uDebug: while the Dafny verifier proves that the code satisfies its formal specification, the *uDebug* integration confirms that the code remains functionally correct across a comprehensive range of real-world scenarios. This approach eliminates the circularity of using an LLM to test its own generated code and provides a much higher degree of confidence in the programs' robustness than standard textbook-style test sets.

### 5.5. LLM Selection and Evaluation Setup

Program synthesis in a verification-aware language like Dafny requires more than syntactic fluency. The synthesis demands an intricate understanding of formal semantics, proof obligations, and the underlying theorem-proving logic of the language [50]. To evaluate these reasoning capabilities, we selected a diverse suite of contemporary LLMs ranging from 4B to 176B parameters, including both general-purpose and code-specialized architectures.

To evaluate the performance of diverse generative architectures on formal Dafny synthesis, we utilize a selection of seven state-of-the-art Large Language Models (LLMs) ranging from specialized coding assistants to massive-scale general reasoners. Our general-purpose reasoning suite includes GPT-OSS-120B [70], GPT-OSS-20B [70], and Gemma 4-31b [71], which provide a baseline for high-level logic and instruction following. These are contrasted with a series of models specifically optimized for software engineering tasks: Qwen3.5-9B [72], Qwen3-Coder-30B [73], Codestral-22b-v0.1 [74], and the mixture-of-experts (MoE) based Qwen3.6-35b-a3b [73]. By evaluating models across this spectrum of parameter sizes and training objectives, we can analyze the correlation between general reasoning capacity and the precision required for formal specification generation.

 doi:10.20944/preprints202604.1772.v1



Table 3 provides a detailed overview of the model suite. Each model was evaluated across five temperature settings ($T = 0.0$ to $0.8$) to identify the optimal configuration for balancing creative exploration with logical precision. We consider the open-source weights architectures; our study provides a comprehensive look at the current state of automated formal verification across different scales of machine intelligence.

**Table 3.** Large Language Models Evaluated for Dafny Synthesis.

| Model | Params | Context | Type | Category |
|---|---|---|---|---|
| GPT-OSS-120B | 120B | 131k | OS | General |
| Qwen 3.6-35B-A3B | 35B | 256k | OS | Agentic |
| Gemma 4-31B | 31B | 256k | OS | Multimodal |
| Qwen 3 Coder 30B | 30B | 160k | OS | Coder |
| Codestral-22B-v0.1 | 22B | 32k | OS | Coder |
| GPT-OSS-20B | 20B | 128k | OS | General |
| Qwen 3.5-9B | 9B | 262k | OS | General |

OS = Open-Source Weights

### *5.6. Prompt Design*

Based on our research questions and the unique challenges posed by the NL2VC-60 dataset, we designed three distinct prompting strategies. Each level of prompting is intended to evaluate how increasing structural context and iterative feedback affect the synthesis of verified Dafny programs.

#### 5.6.1. RQ1 [Contextless Prompting]

To answer RQ1, we use *contextless prompting* by providing only the natural language problem description without any additional structural guidance. This setup establishes a baseline to evaluate the model's ability to infer program structure, formal specifications, and verification constraints (such as termination arguments) directly from the requirement.

**Contextless Prompt:**
You are an expert in Dafny. Output ONLY raw Dafny code.
Generate one Dafny source file for the following task.

Problem ID: <Problem_ID>
Task Description: <Generalized_Description>

#### 5.6.2. RQ2 [Method Signature Prompting]

For RQ2, we utilize *method signature prompting* by supplying an additional structured hint in the form of a formal method signature. This guidance constrains the solution space and helps the model align its implementation with the expected input-output behavior and type-system requirements. Code generation experiments for non-verified code often perform better when prompted with signatures; we hypothesize that this formal frame is even more critical for successful verification in Dafny.

**Method Signature Prompt:**
You are an expert in Dafny. Output ONLY raw Dafny code.
Generate one Dafny source file for the following task.

Problem ID: <Problem_ID>
Task Description: <Generalized_Description>

Method Signature Prompt: <Method_Signature_Prompt>

#### 5.6.3. RQ3 [Self-Healing Prompting]

To address RQ3, we employ *self-healing prompting* by iteratively refining generated programs based on direct feedback from the Dafny verifier. When a generated program fails verification, we

 doi:10.20944/preprints202604.1772.v1



feed the specific error messages (e.g., assertion violations or termination failures) back into the model along with the previous code. We apply this process to both contextless and method-signature settings (RQ3a and RQ3b) to evaluate the model's capacity to correct specification errors and invariant issues through autonomous repair.

**Self-Healing Prompt:**
The previous Dafny code failed verification with the following errors:
<Dafny_Verifier_Output>

Please repair the code to satisfy all specifications. Output ONLY the raw fixed Dafny code.

### *5.7. Evaluation Metrics*

We evaluate the quality of the LLM-synthesized methods using a multi-layered approach that combines formal verification, functional validation, and qualitative error analysis.

#### 5.7.1. Quantitative Metrics: verify@k and functional@k

The primary metric for our study is *verify@k* (adapted from *pass@k* [75,76]), which measures the model's ability to produce at least one formally verified solution within $k$ attempts. A problem is considered solved under this metric only if the Dafny verifier signals that the implementation satisfies all formal specifications, as follows in existing research [69].

#### 5.7.2. Qualitative Assessment: Specification Strength and Error Analysis

Automated metrics serve as a proxy for performance, but they do not capture the nuance of formal reasoning. To assess the semantic depth of the results, we manually reviewed all verified methods to ensure they contain strong formal specifications, specifically, postconditions that fully capture the problem's requirements rather than assertions.

For the failures observed in RQ3 (Self-Healing), we conducted a manual inspection of the verifier's error logs. We categorized these failures into distinct types, such as *termination failures* (missing or incorrect `decreases` clauses), *invariant violations*, and *index out-of-bounds* errors. This analysis allows to evaluate the extent to which the iterative feedback loop addresses the logical challenges of formal proof development.

### *5.8. Temperature Tuning*

The temperature is a hyperparameter in LLMs that controls the randomness and creativity of the decoding process [77–79]. Lower temperatures lead to more deterministic and focused outputs, while higher temperatures encourage diversity at the risk of logical incoherence. Since formal synthesis in Dafny requires high structural precision, identifying the optimal temperature for maximizing verification rates.

To determine the ideal configuration for our study, we conducted a temperature tuning experiment on a subset of the NL2VC-60 dataset. We selected a representative sample of problems and executed each across five distinct temperature settings: $T \in \{0.0, 0.2, 0.4, 0.6, 0.8\}$. This range allows us to observe the transition from greedy decoding ($T = 0.0$) to high-variance sampling ($T = 0.8$).

We evaluated the synthesized methods using the *verify@k* metric where $k \in \{1, 3, 5\}$. Our preliminary results indicated that lower temperatures (0.0 to 0.4) generally yielded higher success rates for initial synthesis (RQ1 and RQ2), as the models remained more faithful to Dafny's strict syntax. However, for the Self-Healing process (RQ3), slightly higher temperatures occasionally proved beneficial by allowing the model to explore alternative algorithmic implementations when the primary logic failed to verify. Based on these findings, we report our final results using the optimal temperature identified for each specific model and prompt type.

 doi:10.20944/preprints202604.1772.v1



### *5.9. Error Analysis*

We analyzed the error logs for different methods on identified three types of errors. We then examined the error types and classified them into several subgroups. Table 7 demonstrates the presence of different types of errors in open source LLMs.

#### 5.9.1. **Syntax Errors**

Dominant in Contextless and Signature Prompting for most models. This confirms that without iterative feedback, models frequently struggle with Dafny's specific grammar (e.g., missing semicolons, improper loop syntax).

#### 5.9.2. **Semantic / Type Errors**

High in models like GPT-OSS-120B and MistralAI. These occur when the code is structurally correct but violates Dafny's strict type system, or attempts to use unavailable modules (such as System).

#### 5.9.3. **Verification Errors**

Notably higher in Self-Healing categories for models like Gemma 4 and Qwen3-Code30B. This suggests that as models fix their syntax through iterations, they reach a stage where the code compiles but fails the deeper logical proof (e.g., an assertion might not hold).

### *5.10. Experimental Setup*

To conduct our large-scale synthesis and verification experiments, we utilized a distributed environment comprising a high-performance inference server and a local development machine. For the open-source Large Language Models (LLMs), we deployed **LM Studio version 0.4.8 (Build 1)** on a dedicated Ubuntu-based server. This server features a high-end hardware configuration equipped with four **NVIDIA RTX 6000 Ada Generation** GPUs, each providing 48 GB of VRAM (totaling 192 GB), supported by **NVIDIA Driver version 580.126.09** and **CUDA 13.0**. This infrastructure allowed us to host and query large-parameter models locally, ensuring consistent inference latency for our 60-problem dataset.

For the development of orchestration scripts and the formal verification of synthesized methods, we used a **MacBook Pro** (Model Mac16,8) running **macOS 15.3.1**. The software environment was managed using **Visual Studio Code version 1.115.0** (arm64) and **Python 3.14.4** within a dedicated virtual environment in our experiments.

To manage the iterative repair process, our self-healing orchestration script was configured with a **maximum of 10 repair attempts** per problem. If a model failed to produce a verified solution within these ten iterations, the result was recorded as a failure for that specific trial. Furthermore, to accommodate the processing time of large models (such as GPT-OSS-120B) and prevent connection timeouts, we implemented a **response wait time of 180 seconds** for all LM Studio API calls during our experiments.

For verifying the code, we employed **Dafny 4.11.0** [80], which represents a modern and stable iteration of the language. Since recent versions of Dafny introduced significant changes to the verification engine and syntax compared to the 3.x series, our experiments provide a rigorous test of the models' ability to adapt to contemporary formal verification standards. All functional validation against *uDebug* [20] test suites was executed within this same environment to ensure parity between the formal proof and the executable implementation.

## 6. Results

This section presents the empirical findings of our study. To establish a robust evaluation framework, we manually developed NL2VC-60, a benchmark dataset consisting of 60 formally verified Dafny programs used to optimize our tiered prompting strategies. For the primary comparative analysis, we evaluated seven state-of-the-art open-weight models against a subset of randomly selected 11

 doi:10.20944/preprints202604.1772.v1



algorithmic problems. Performance is measured using the *verify@k* metric, indicating the percentage of problems for which at least one successful verification was achieved within $k$ attempts.

### 6.1. RQ1: Performance of Contextless Prompting

The initial evaluation of LLMs using contextless prompting shows significant disparities in their ability to synthesize verifiable Dafny code from raw requirements. As shown in Table 4, the majority of the tested models, including the massive GPT OSS 120B and the Qwen series, failed to produce a single verified solution across all temperature settings. This widespread failure suggests that simply providing a natural language problem description is insufficient for most models to navigate the syntactic and semantics of the Dafny language.

Several observations emerge from the RQ1:

- Unlike the other general purpose models, Gemma 4-31B showed a surprising aptitude for generating verifiable programs even without external context. The model achieved a peak verify@5 success rate of 54.55% at a temperature of 0.2. This performance indicates that its pretraining likely involved a higher density of formal or algorithmic logic, allowing it to guess correct loop invariants and post-conditions that other models completely missed.
- Codestral was the only other model to consistently yield results, reaching a peak verify@5 of 27.27% at $T = 0.8$. The success of this model at higher temperatures suggests that while the model possesses the basic syntactic intuition for Dafny. The model often requires more stochastic exploration to arrive at the precise formal annotations needed to satisfy the Z3 SMT solver.
- The 0% success rate of the remaining five models highlights a fundamental challenge in the field. Even highly capable models struggle to infer complex formal specifications from scratch. The findings are reinforcing the need for more structured prompting techniques or retrieval augmentation to bridge the gap between informal requirements and mathematical proof.

 doi:10.20944/preprints202604.1772.v1



**Table 4.** Complete Verification Success Rates for Contextless Prompting (RQ1).

| Model | Temp (T) | verify@1 | | | verify@3 | | | verify@5 | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Succ. | Total | % | Succ. | Total | % | Succ. | Total | % |
| **GPT-OSS-120B** | 0.0 | 0 | 11 | 0.00% | 0 | 11 | 0.00% | 0 | 11 | 0.00% |
| | 0.2 | 0 | 11 | 0.00% | 0 | 11 | 0.00% | 0 | 11 | 0.00% |
| | 0.4 | 0 | 11 | 0.00% | 0 | 11 | 0.00% | 0 | 11 | 0.00% |
| | 0.6 | 0 | 11 | 0.00% | 0 | 11 | 0.00% | 0 | 11 | 0.00% |
| | 0.8 | 0 | 11 | 0.00% | 0 | 11 | 0.00% | 0 | 11 | 0.00% |
| **Qwen 3.5-9B** | 0.0 | 0 | 11 | 0.00% | 0 | 11 | 0.00% | 0 | 11 | 0.00% |
| | 0.2 | 0 | 11 | 0.00% | 0 | 11 | 0.00% | 0 | 11 | 0.00% |
| | 0.4 | 0 | 11 | 0.00% | 0 | 11 | 0.00% | 0 | 11 | 0.00% |
| | 0.6 | 0 | 11 | 0.00% | 0 | 11 | 0.00% | 0 | 11 | 0.00% |
| | 0.8 | 0 | 11 | 0.00% | 0 | 11 | 0.00% | 0 | 11 | 0.00% |
| **Qwen 3 Coder 30B** | 0.0 | 0 | 11 | 0.00% | 0 | 11 | 0.00% | 0 | 11 | 0.00% |
| | 0.2 | 0 | 11 | 0.00% | 0 | 11 | 0.00% | 0 | 11 | 0.00% |
| | 0.4 | 0 | 11 | 0.00% | 0 | 11 | 0.00% | 0 | 11 | 0.00% |
| | 0.6 | 0 | 11 | 0.00% | 0 | 11 | 0.00% | 0 | 11 | 0.00% |
| | 0.8 | 0 | 11 | 0.00% | 0 | 11 | 0.00% | 0 | 11 | 0.00% |
| **GPT-OSS 20B** | 0.0 | 0 | 11 | 0.00% | 0 | 11 | 0.00% | 0 | 11 | 0.00% |
| | 0.2 | 0 | 11 | 0.00% | 0 | 11 | 0.00% | 0 | 11 | 0.00% |
| | 0.4 | 0 | 11 | 0.00% | 0 | 11 | 0.00% | 0 | 11 | 0.00% |
| | 0.6 | 0 | 11 | 0.00% | 0 | 11 | 0.00% | 0 | 11 | 0.00% |
| | 0.8 | 0 | 11 | 0.00% | 0 | 11 | 0.00% | 0 | 11 | 0.00% |
| **Codestral-22B** | 0.0 | 1 | 11 | 9.09% | 1 | 11 | 9.09% | 1 | 11 | 9.09% |
| | 0.2 | 1 | 11 | 9.09% | 1 | 11 | 9.09% | 2 | 11 | 18.18% |
| | 0.4 | 1 | 11 | 9.09% | 2 | 11 | 18.18% | 2 | 11 | 18.18% |
| | 0.6 | 1 | 11 | 9.09% | 2 | 11 | 18.18% | 1 | 11 | 9.09% |
| | 0.8 | 1 | 11 | 9.09% | 1 | 11 | 9.09% | 3 | 11 | 27.27% |
| **Qwen 3.6-35B** | 0.0 | 0 | 11 | 0.00% | 0 | 11 | 0.00% | 0 | 11 | 0.00% |
| | 0.2 | 0 | 11 | 0.00% | 1 | 11 | 9.09% | 2 | 11 | 18.18% |
| | 0.4 | 0 | 11 | 0.00% | 1 | 11 | 9.09% | 2 | 11 | 18.18% |
| | 0.6 | 0 | 11 | 0.00% | 0 | 11 | 0.00% | 1 | 11 | 9.09% |
| | 0.8 | 0 | 11 | 0.00% | 0 | 11 | 0.00% | 1 | 11 | 9.09% |
| **Gemma 4-31B** | 0.0 | **3** | **11** | **27.27%** | 3 | 11 | 27.27% | 3 | 11 | 27.27% |
| | 0.2 | 0 | 11 | 0.00% | 4 | 11 | 36.36% | **6** | **11** | **54.55%** |
| | 0.4 | 0 | 11 | 0.00% | **5** | **11** | **45.45%** | 3 | 11 | 27.27% |
| | 0.6 | **2** | **11** | **18.18%** | **5** | **11** | **45.45%** | 4 | 11 | 36.36% |
| | 0.8 | **2** | **11** | **18.18%** | **3** | **11** | **27.27%** | **4** | **11** | **36.36%** |

### *6.2. RQ2: Performance of Signature Prompting*

The second phase of our evaluation explores the impact of providing the method signature as additional context. As illustrated in Table 5, the structural guidance resulted in a performance shift, effectively reversing the widespread failures observed in the contextless setting. By providing the skeleton of the Dafny method including the input parameters and return types, the models were freed from the burden of syntactic structure and could instead focus on synthesizing the internal logic and required verification annotations.

Several observations emerge from the RQ2:

- The most striking improvement was observed in **GPT-OSS-120B**. While this model recorded a 0% success rate in RQ1, the introduction of method signatures allowed it to achieve a peak *verify@5*

 doi:10.20944/preprints202604.1772.v1



rate of 63.64% at $T = 0.8$. This suggests that the model possesses a deep latent knowledge of formal verification and Dafny logic but lacks the ability to self-structure the initial code container from raw requirements.

- Surprisingly, **Qwen 3.5-9B** emerged as the top performer in this category, reaching a peak *verify@5* success rate of 72.73%. This indicates that signature prompting provides constraint to allow smaller models to focus their computational budget on the complex task, often outperforming much larger general-purpose models.
- Unlike the zero-shot results, peak performance under signature prompting was consistently achieved at higher temperatures (typically $T = 0.8$). This suggests that once the structural constraints are fixed via the signature, the models benefit from increased stochastic exploration to identify the precise mathematical formulations. For example, specific loop invariants or termination measures required to satisfy the SMT solver.
- All seven models demonstrated signs of life in this setting, with even the weakest models surpassing a 30% success rate at their peak. This confirms that the primary bottleneck in verifiable code generation is not necessarily the logic itself, but the difficulty of mapping informal natural language to the rigid formal signatures required by the Dafny compiler.

 doi:10.20944/preprints202604.1772.v1



**Table 5.** Verification Success Rates for Signature Prompting (RQ2).

| Model | Temp (T) | verify@1 | | | verify@3 | | | verify@5 | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Succ. | Total | % | Succ. | Total | % | Succ. | Total | % |
| **GPT-OSS-120B** | 0.0 | **6** | 11 | **54.55%** | 6 | 11 | 54.55% | 6 | 11 | 54.55% |
| | 0.2 | 5 | 11 | 45.45% | 6 | 11 | 54.55% | 6 | 11 | 54.55% |
| | 0.4 | **6** | 11 | **54.55%** | 6 | 11 | 54.55% | 6 | 11 | 54.55% |
| | 0.6 | **6** | 11 | **54.55%** | 5 | 11 | 45.45% | 6 | 11 | 54.55% |
| | 0.8 | **6** | 11 | **54.55%** | **7** | **11** | **63.64%** | **7** | **11** | **63.64%** |
| **Qwen 3.5-9B** | 0.0 | 3 | 11 | 27.27% | 2 | 11 | 18.18% | 2 | 11 | 18.18% |
| | 0.2 | **4** | 11 | **36.36%** | 3 | 11 | 27.27% | 5 | 11 | 45.45% |
| | 0.4 | 2 | 11 | 18.18% | **5** | 11 | **45.45%** | 4 | 11 | 36.36% |
| | 0.6 | 2 | 11 | 18.18% | 4 | 11 | 36.36% | 3 | 11 | 27.27% |
| | 0.8 | **4** | 11 | **36.36%** | **5** | 11 | **45.45%** | **8** | **11** | **72.73%** |
| **Qwen 3 Coder 30B** | 0.0 | 3 | 11 | 27.27% | 4 | 11 | 36.36% | 3 | 11 | 27.27% |
| | 0.2 | 3 | 11 | 27.27% | 4 | 11 | 36.36% | 3 | 11 | 27.27% |
| | 0.4 | **4** | 11 | **36.36%** | **5** | **11** | **45.45%** | **5** | **11** | **45.45%** |
| | 0.6 | **4** | 11 | **36.36%** | 4 | 11 | 36.36% | **5** | **11** | **45.45%** |
| | 0.8 | **4** | 11 | **36.36%** | 4 | 11 | 36.36% | 4 | 11 | 36.36% |
| **GPT-OSS 20B** | 0.0 | 3 | 11 | 27.27% | 3 | 11 | 27.27% | 3 | 11 | 27.27% |
| | 0.2 | 4 | 11 | 36.36% | 5 | 11 | 45.45% | **5** | **11** | **45.45%** |
| | 0.4 | 2 | 11 | 18.18% | 4 | 11 | 36.36% | **5** | **11** | **45.45%** |
| | 0.6 | **5** | **11** | **45.45%** | **6** | **11** | **54.55%** | 4 | 11 | 36.36% |
| | 0.8 | 2 | 11 | 18.18% | **6** | **11** | **54.55%** | **5** | **11** | **45.45%** |
| **Codestral-22B** | 0.0 | 3 | 11 | 27.27% | **3** | **11** | **27.27%** | 3 | 11 | 27.27% |
| | 0.2 | **4** | **11** | **36.36%** | **4** | **11** | **36.36%** | 5 | 11 | 45.45% |
| | 0.4 | 3 | 11 | 27.27% | 2 | 11 | 18.18% | 5 | 11 | 45.45% |
| | 0.6 | 2 | 11 | 18.18% | **3** | **11** | **27.27%** | **7** | **11** | **63.64%** |
| | 0.8 | 3 | 11 | 27.27% | 2 | 11 | 18.18% | 6 | 11 | 54.55% |
| **Qwen 3.6-35B** | 0.0 | **2** | **11** | **18.18%** | **3** | **11** | **27.27%** | **7** | **11** | **63.64%** |
| | 0.2 | **2** | **11** | **18.18%** | **3** | **11** | **27.27%** | 3 | 11 | 27.27% |
| | 0.4 | **2** | **11** | **18.18%** | **3** | **11** | **27.27%** | 4 | 11 | 36.36% |
| | 0.6 | 1 | 11 | 9.09% | **3** | **11** | **27.27%** | 4 | 11 | 36.36% |
| | 0.8 | 0 | 11 | 0.00% | **3** | **11** | **27.27%** | 3 | 11 | 27.27% |
| **Gemma 4-31B** | 0.0 | 3 | 11 | 27.27% | 3 | 11 | 27.27% | 5 | 11 | 45.45% |
| | 0.2 | 3 | 11 | 27.27% | 3 | 11 | 27.27% | 6 | 11 | 54.55% |
| | 0.4 | 3 | 11 | 27.27% | 3 | 11 | 27.27% | 4 | 11 | 36.36% |
| | 0.6 | 3 | 11 | 27.27% | 3 | 11 | 27.27% | 4 | 11 | 36.36% |
| | 0.8 | **5** | **11** | **45.45%** | **7** | **11** | **63.64%** | **7** | **11** | **63.64%** |

### *6.3. RQ3: Performance of Self-Healing Mechanisms*

The third research question evaluates the efficacy of iterative self-healing, where models are provided with error feedback from the Dafny compiler to repair their own code. The evaluation consists of two strategies: healing from a contextless baseline (RQ3a) and healing from a signature-guided baseline (RQ3b). As shown in Table 6, the ability to self-correct varies significantly across models, with the initial prompt quality serving as a critical predictor of repair success.

- **RQ3a: Contextless Healing:** When attempting to heal from the zero-shot failures of RQ1, most models remained stagnant. GPT-OSS-120B and several others continued to post 0% success rates. The findings suggests that without an initial structural foundation, compiler error messages are too abstract for the model to navigate toward a valid solution. However, **Gemma 4-31B** proved

 doi:10.20944/preprints202604.1772.v1



to be a notable exception, demonstrating a remarkable self-correction capability. By leveraging compiler feedback, it achieved a peak *verify@5* rate of 90.91% at $T = 0.2$ and $T = 0.6$, indicating that it can effectively use error logs to guess missing loop invariants and post-conditions.

- **RQ3b: Signature-Guided Healing:** Self-healing proved most potent when initiated from the signature-guided prompts of RQ2. In this setting, the structural skeleton provided enough stability for the compiler feedback to be actionable. **GPT-OSS-120B** demonstrated the most significant turnaround, rising to a peak of 81.82% success at $T = 0.2$. This suggests that when the method signature is fixed, larger models are efficient at using error feedback to refine mathematical proofs and satisfy the SMT solver.
- Unlike previous rounds, several models (such as Qwen 3 Coder 30B) reached a plateau where performance remained consistent across temperatures in the signature-guided setting. Conversely, **Gemma 4-31B** maintained high performance (over 80%) in both RQ3a and RQ3b, establishing itself as the most robust model for autonomous Dafny development, regardless of the initial prompt's context level.

 doi:10.20944/preprints202604.1772.v1



**Table 6.** Detailed Verification Success Rates for Self-Healing (RQ3).

| Model | Temp (T) | Contextless Healing (RQ3a) | | | Signature-Guided Healing (RQ3b) | | |
|---|---|---|---|---|---|---|---|
| | | Succ. | Total | % | Succ. | Total | % |
| **GPT-OSS-120B** | 0.0 | 0 | 11 | 0.00% | 7 | 11 | 63.64% |
| | 0.2 | 0 | 11 | 0.00% | **9** | **11** | **81.82%** |
| | 0.4 | 0 | 11 | 0.00% | 7 | 11 | 63.64% |
| | 0.6 | 0 | 11 | 0.00% | 8 | 11 | 72.73% |
| | 0.8 | 0 | 11 | 0.00% | 7 | 11 | 63.64% |
| **Qwen 3.5-9B** | 0.0 | **3** | **11** | **27.27%** | 2 | 11 | 18.18% |
| | 0.2 | 0 | 11 | 0.00% | 3 | 11 | 27.27% |
| | 0.4 | 2 | 11 | 18.18% | **5** | **11** | **45.45%** |
| | 0.6 | 0 | 11 | 0.00% | 4 | 11 | 36.36% |
| | 0.8 | 0 | 11 | 0.00% | 3 | 11 | 27.27% |
| **Qwen 3 Coder 30B** | 0.0 | 2 | 11 | 18.18% | **6** | **11** | **54.55%** |
| | 0.2 | 2 | 11 | 18.18% | **6** | **11** | **54.55%** |
| | 0.4 | 3 | 11 | 27.27% | **6** | **11** | **54.55%** |
| | 0.6 | 1 | 11 | 9.09% | **6** | **11** | **54.55%** |
| | 0.8 | **6** | **11** | **54.55%** | **6** | **11** | **54.55%** |
| **GPT-OSS 20B** | 0.0 | 0 | 11 | 0.00% | 4 | 11 | 36.36% |
| | 0.2 | **1** | **11** | **9.09%** | 4 | 11 | 36.36% |
| | 0.4 | 0 | 11 | 0.00% | **7** | **11** | **63.64%** |
| | 0.6 | 0 | 11 | 0.00% | 5 | 11 | 45.45% |
| | 0.8 | **1** | **11** | **9.09%** | 4 | 11 | 36.36% |
| **Codestral-22B** | 0.0 | 0 | 11 | 0.00% | 2 | 11 | 18.18% |
| | 0.2 | 0 | 11 | 0.00% | 3 | 11 | 27.27% |
| | 0.4 | 0 | 11 | 0.00% | 5 | 11 | 45.45% |
| | 0.6 | **1** | **11** | **9.09%** | **6** | **11** | **54.55%** |
| | 0.8 | 0 | 11 | 0.00% | 3 | 11 | 27.27% |
| **Qwen 3.6-35B** | 0.0 | 0 | 11 | 0.00% | 4 | 11 | 36.36% |
| | 0.2 | 0 | 11 | 0.00% | 4 | 11 | 36.36% |
| | 0.4 | 1 | 11 | 9.09% | 4 | 11 | 36.36% |
| | 0.6 | 0 | 11 | 0.00% | 4 | 11 | 36.36% |
| | 0.8 | **2** | **11** | **18.18%** | **6** | **11** | **54.55%** |
| **Gemma 4-31B** | 0.0 | 8 | 11 | 72.73% | 7 | 11 | 63.64% |
| | 0.2 | **10** | **11** | **90.91%** | 7 | 11 | 63.64% |
| | 0.4 | 9 | 11 | 81.82% | 8 | 11 | 72.73% |
| | 0.6 | **10** | **11** | **90.91%** | **9** | **11** | **81.82%** |
| | 0.8 | 8 | 11 | 72.73% | **9** | **11** | **81.82%** |

### *6.4. RQ4: Qualitative Error Analysis and Failure Taxonomy*

We conducted a systematic qualitative analysis of our total runs to understand the specific challenges in automated formal synthesis as depicted in Table 7. We categorized failures into a three-tiered taxonomy: **Syntax Errors** (malformed code structure), **Semantic and Type Errors** (type mismatches or API hallucinations), and **Verification Failures** (syntactically correct code that the SMT solver cannot prove).

#### 6.4.1. Syntactic Fragility and Contextual Dependence

Our analysis confirms that syntax errors are the primary bottleneck for models lacking structural anchors. In contextless settings, **GPT OSS 20B** and **GPT OSS 120B** produced syntax errors in the

 doi:10.20944/preprints202604.1772.v1



majority of attempts. These failures typically involve the misuse of Dafny-specific keywords or the generation of Pythonic indentation, which is incompatible with Dafny's curly-brace syntax. This suggests that while open-weight models possess general algorithmic logic, they lack the specific syntactic density required for niche-verification languages without external guidance.

#### 6.4.2. Semantic Drift and Invariant Generation

As we transitioned to Signature Prompting, syntax errors decreased significantly, but we observed a sharp rise in semantic and type errors. Models frequently hallucinate non existent predicates or attempt to perform arithmetic on incompatible types, such as treating a sequence as a set. A critical finding is the **Invariant Gap**; even when models generate correct imperative logic, they often fail to provide the inductive loop invariants required for the Z3 solver to complete the proof. Models like **Qwen 3 Coder 30B** demonstrated a tendency to repeat the same insufficient invariant across multiple self-healing iterations, indicating a logical plateau in the repair process.

#### 6.4.3. Functional Robustness and Vacuity

The most complex category involves code that satisfies the verifier but fails the functional test suite. By integrating **uDebug**, we identified several instances where models achieved verification by providing weak specifications. For instance, a model might satisfy a postcondition by returning a trivial constant that happens to meet a weak mathematical constraint. The uDebug community test cases acted as a vital truth oracle, identifying these as functional failures and ensuring that the verified code maintains real world utility against extreme edge cases.

**Table 7.** Dafny Compilation and Verification Errors.

| Model | Prompt Strategy | Total Runs | Syntax Errors | Semantic/Type Errors | Verification | Verified |
|---|---|---|---|---|---|---|
| **GPT-OSS-120B** | Contextless | 564 | 435 | 45 | 0 | 0 |
| | Signature Prompt | 816 | 397 | 53 | 111 | 150 |
| | Self-Healing | 1,134 | 620 | 471 | 5 | 20 |
| **GPT-OSS-20B** | Contextless | 672 | 597 | 75 | 0 | 0 |
| | Signature Prompt | 816 | 397 | 53 | 111 | 150 |
| | Self-Healing | 1,564 | 793 | 149 | 166 | 395 |
| **Codestral-22B** | Contextless | 1,285 | 732 | 518 | 0 | 33 |
| | Signature Prompt | 792 | 407 | 217 | 0 | 138 |
| | Self-Healing | 1,666 | 1,116 | 188 | 58 | 217 |
| **Qwen 3.6-35B** | Contextless | 470 | 9 | 23 | 0 | 438 |
| | Signature Prompt | 495 | 0 | 29 | 0 | 466 |
| | Self-Healing | 827 | 39 | 127 | 10 | 651 |
| **Qwen 3-Coder-30B** | Contextless | 1,510 | 579 | 657 | 15 | 46 |
| | Signature Prompt | 205 | 53 | 16 | 47 | 29 |
| | Self-Healing | 1,893 | 589 | 569 | 351 | 101 |
| **Qwen 3.5-9B** | Contextless | 910 | 430 | 350 | 23 | 56 |
| | Signature Prompt | 861 | 557 | 77 | 0 | 224 |
| | Self-Healing | 280 | 182 | 14 | 27 | 52 |
| **Gemma 4-31B** | Contextless | 1,016 | 489 | 209 | 25 | 251 |
| | Signature Prompt | 562 | 145 | 5 | 41 | 369 |
| | Self-Healing | 1,008 | 368 | 217 | 110 | 296 |

## 7. Findings and Discussion

This section presents a comprehensive analysis of our experimental results, detailing how different architectural scales and prompting methodologies influence the synthesis of provably correct

 doi:10.20944/preprints202604.1772.v1



code. By systematically decomposing the performance of seven state-of-the-art models across four Research Questions (RQs), we illustrate the critical transition from natural language requirements to formal mathematical proofs. The following findings highlight the interplay between model reasoning, structural guidance, and the iterative feedback loops required to overcome the data-scarcity bottleneck in the Dafny.

### *7.1. Summary of Findings*

Our evaluation of seven open-weight models across three prompting tiers shows critical insights into the automated synthesis of formally verified software.

- **RQ1 [Contextless Prompting]:** Our experiments show that while most LLMs fail to generate verifiable Dafny code from raw requirements, **Gemma 4-31B** and **Codestral-22B** demonstrate a surprising aptitude for the task. Specifically, Gemma 4-31B achieved a peak *verify@5* success rate of 54.55% at $T = 0.2$. However, the 0% success rate of the remaining five models suggests that without structural guidance or external context, most systems struggle to navigate the strict formal constraints of the Dafny language.
- **RQ2 [Signature Prompting]:** Providing the method signature as additional context drastically improved performance across the board, reversing the widespread failures observed in RQ1. Most notably, **GPT-OSS-120B** rose from a 0% success rate to 63.64%, while the smaller **Qwen 3.5-9B** achieved the highest overall *verify@5* score of 72.73% at $T = 0.8$. These results indicate that the primary bottleneck in verifiable synthesis is the structural mapping of requirements to formal signatures, rather than the generation of the underlying verification logic.
- **RQ3 [Self-Healing]:** Iterative self-healing significantly amplifies success rates, provided a structural foundation (method signature) is present. **Gemma 4-31B** emerged as the most resilient model, achieving a near-perfect 90.91% success rate in contextless healing. Meanwhile, **GPT-OSS-120B** achieved its performance ceiling (81.82%) only when signature-guided, suggesting that large-scale general purpose models require structural constraints to effectively interpret and act upon formal compiler feedback.
- **RQ4 [Error Distributions]:** Our analysis of compilation failures shows that **Syntax Errors** are the primary barrier in contextless settings, often exceeding 80% of total failures for models like GPT-OSS-20B. While **Signature Prompting** significantly reduces syntax issues, it shifts the bottleneck to **Verification Errors**, particularly for the largest models. Notably, **Self-Healing** effectively converts semantic and syntax errors into verified solutions for most models, though **Codestral-22B** and **Qwen 3-Coder-30B** show a tendency to regress into higher syntax error counts during iterative repair, suggesting a struggle to maintain syntactic integrity under compiler-driven feedback.

### *7.2. Discussion*

The results of this study suggest a shift in the paradigm of automated formal programming. While the scaling law often assumes that larger parameters equate to better reasoning, our findings indicate a more nuanced reality. The significant success of **Gemma 4-31B**, which achieved a near-perfect 90.91% success rate in self-healing, suggests that specific pretraining data density regarding formal and algorithmic logic is more critical than raw model size for the Dafny language.

Furthermore, the transition from total failure in RQ1 (0% for most models) to high success in RQ3b underscores the necessity of a Verification-in-the-Loop approach. By utilizing the Dafny verifier and the Z3 SMT solver as a ground-truth reward signal, we effectively mitigate the common LLM issue of logical hallucinations. The integration of **uDebug** was essential to this framework; it ensured that models did not achieve verification through vacuous or trivial specifications such as empty post-conditions but rather through functional correctness that holds up against rigorous edge cases.

Finally, the structural bottleneck identified in RQ2 suggests that the future of automated formal methods lies in hybrid prompting strategies. Even the most capable models, like **GPT-OSS-120B**, require a structural skeleton (the method signature) to bridge the gap between natural language intent

 doi:10.20944/preprints202604.1772.v1



and mathematical proof. This suggests that LLMs should be viewed not as autonomous agents, but as sophisticated co-processors that thrive when provided with high-level formal constraints to guide their stochastic exploration.

## 8. Threats to Validity

Following Siegmund et al. [81] and Feldt and Magazinius [82], we identified several threats to the validity of this study. The deterministic nature of our verification criteria, relying on the Dafny verifier's acceptance and the success of the **uDebug** test suite, is designed to ensure **construct and internal validity**. By using the formal verifier and the Z3 SMT solver as a ground-truth oracle, we eliminate human subjectivity in assessing whether the AI assisted synthesized code meets the formal requirements.

**External validity** concerns the generalizability of our results to other models and languages. We acknowledge that the open-weight landscape, featuring models like **Gemma 4-31B**, **Qwen 3.6-35B**, and **GPT-OSS-120B**, is evolving rapidly. Our findings are specific to the contemporary architectures and the Dafny 4.11.0 verification engine. While we expect the success rates to improve with future iterations, the performance disparities observed particularly the structural bottleneck in contextless prompting. The issue likely represents a fundamental challenge in mapping natural language to formal logic that persists across model generations.

A specific threat to external validity is vacuous verification, where a model satisfies the verifier with trivial specifications (e.g., `ensures true`). We mitigated this threat through our dual-layer validation pipeline. By requiring all verified methods to pass the **uDebug** functional test suites, we ensure that our results represent genuine functional correctness rather than mere logical consistency with a weak or empty specification. This approach strengthens the claim that the models are reasoning rather than satisfying the solver's constraints.

While our dataset introduces the first verified Dafny implementations for these UVa problems, there remains a risk that models may leverage cross-lingual knowledge of the underlying algorithms from more prevalent languages like C++ or Python. However, we mitigate this by focusing our evaluation on the synthesis of formal specifications and loop invariant constructs that are uniquely absent from standard competitive programming solutions.

Finally, to ensure reproducibility despite the rapid development of these tools, we have documented the precise hardware configurations (NVIDIA RTX 6000 Ada) and software versions (LM Studio 0.4.8, Python 3.14.4) used. The research artifacts, including the NL2VC-60 dataset and our synthesis pipeline, are provided to allow for verification of these results.

## 9. Conclusion

We investigated the potential of contemporary open-weight LLMs to synthesize formally verified methods and specifications in the Dafny programming language. Utilizing the NL2VC-60 dataset, we evaluated tiered prompting strategies across seven state-of-the-art models. Our findings confirm that while contextless natural language prompts generally lead to synthesis failure due to a structural bottleneck, tiered strategies incorporating formal method signatures and iterative self-healing allow models to overcome the scarcity of specialized training data. Notably, **Gemma 4-31B** emerged as a highly resilient verification assistant, achieving a peak success rate of **90.91%**, while the **GPT-OSS-120B** demonstrated the most significant performance leap when transitioned to a signature-guided healing pipeline.

Our results demonstrate that an iterative feedback loop utilizing direct SMT-solver output, combined with structural anchors, yielded the highest performance ceiling. The orchestrated self-healing pipeline achieved a verification success rate of **81.82%** for the 120B model and nearly 91% for the 31B model. By integrating the **uDebug** platform, we confirmed that these verified solutions are not only logically consistent but also functionally robust against extreme edge cases, effectively mitigating the risk of vacuous verification.

 doi:10.20944/preprints202604.1772.v1



These findings underscore the importance of Verification-in-the-Loop architectures and suggest that open-weight models can significantly lower the specification challenges. Our study suggests that the high success rates achieved by these models represent a several-thousand-fold cost reduction compared to human expert synthesis, making high-assurance software economically viable for general engineering tasks. Ultimately, the integration of formal oracles and generative models represents a vital path toward a future of trustworthy, AI-assisted software engineering where code is not plausible, but provably correct.

## References


1. Mesbah, A.; Van Deursen, A.; Roest, D. Invariant-based automatic testing of modern web applications. *IEEE Transactions on Software Engineering* **2011**, *38*, 35–53.
2. Rushby, J. Model checking and other ways of automating formal methods. *Position paper for panel on model checking for concurrent programs, Software Quality Week, San Francisco* **1995**.
3. ter Beek, M.; Broy, M.; Dongol, B. The role of formal methods in computer science education. *ACM Inroads* **2024**, *15*, 58–66.
4. Dipu, N.F.; Hossain, M.M.; Azar, K.Z.; Farahmandi, F.; Tehranipoor, M. Formalfuzzer: Formal verification assisted fuzz testing for soc vulnerability detection. In Proceedings of the 2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC). IEEE, 2024, pp. 355–361.
5. Paul, S.; Cruz, E.; Dutta, A.; Bhaumik, A.; Blasch, E.; Agha, G.; Patterson, S.; Kopsaftopoulos, F.; Varela, C. Formal verification of safety-critical aerospace systems. *IEEE Aerospace and Electronic Systems Magazine* **2023**, *38*, 72–88.
6. Barrett, C.; De Moura, L.; Stump, A. SMT-COMP: Satisfiability modulo theories competition. In Proceedings of the International Conference on Computer Aided Verification. Springer, 2005, pp. 20–23.
7. Klein, G.; Derrin, P.; Elphinstone, K. Experience report: sel4: formally verifying a high-performance microkernel. In Proceedings of the Proceedings of the 14th ACM SIGPLAN international conference on Functional programming, 2009, pp. 91–96.
8. Murray, T.; Matichuk, D.; Brassil, M.; Gammie, P.; Bourke, T.; Seefried, S.; Lewis, C.; Gao, X.; Klein, G. seL4: from general purpose to a proof of information flow enforcement. In Proceedings of the 2013 IEEE Symposium on Security and Privacy. IEEE, 2013, pp. 415–429.
9. Leroy, X. The CompCert C verified compiler: Documentation and user's manual. PhD thesis, Inria, 2025.
10. Leroy, X. Formal verification of a realistic compiler. *Communications of the ACM* **2009**, *52*, 107–115.
11. Anysphere, Inc.. Cursor: The AI Code Editor, 2024. Accessed: 2026-04-21.
12. Amazon Web Services, Inc.. Amazon Q Developer: AI coding companion, 2024. Accessed: 2026-04-21.
13. Ray, P.P. A review on vibe coding: Fundamentals, state-of-the-art, challenges and future directions. *Authorea Preprints* **2025**.
14. Ji, Z.; Lee, N.; Frieske, R.; Yu, T.; Su, D.; Xu, Y.; Ishii, E.; Bang, Y.J.; Madotto, A.; Fung, P. Survey of hallucination in natural language generation. *ACM computing surveys* **2023**, *55*, 1–38.
15. Leino, K.R.M. Dafny: An automatic program verifier for functional correctness. In Proceedings of the International conference on logic for programming artificial intelligence and reasoning. Springer, 2010, pp. 348–370.
16. Hoare, C.A.R. An axiomatic basis for computer programming. *Communications of the ACM* **1969**, *12*, 576–580.
17. Noble, J.; Streader, D.; Gariano, I.O.; Samarakoon, M. More programming than programming: Teaching formal methods in a software engineering programme. In Proceedings of the NASA Formal Methods Symposium. Springer, 2022, pp. 431–450.
18. GitHub Community. Dafny Repositories Search Results, 2026. Accessed: 2026-04-23.
19. UVa Online Judge. UVa Online Judge. https://onlinejudge.org/, n.d. Accessed: 2026-04-20.
20. uDebug Team. uDebug: Online Debugging Tool for Competitive Programming. https://www.udebug.com/, 2026. Accessed: 2026-04-19.
21. UVa Online Judge. Problem 11934: Magic Formula. https://onlinejudge.org/external/119/11934.pdf, 2010. Accessed: 2026-04-21.
22. UVa Online Judge. https://onlinejudge.org/, 2026. Accessed: 2026-04-19.
23. Leino, K.R.M. Developing verified programs with Dafny. In Proceedings of the Proceedings of the 2012 ACM conference on High integrity language technology, 2012, pp. 9–10.
24. Dafny Team. *Dafny Reference Manual*. Dafny Software Foundation, 2024. Accessed: 2026-04-20.

 doi:10.20944/preprints202604.1772.v1




25. Microsoft Research. Dafny: A Language and Program Verifier for Functional Correctness, 2024. Official Project Page.
26. Fedchin, A.; Dean, T.; Foster, J.S.; et al. A Toolkit for Automated Testing of Dafny. *Amazon Science* **2023**.
27. Le Goues, C.; Leino, K.R.M.; Moskal, M. The boogie verification debugger (tool paper). In Proceedings of the International Conference on Software Engineering and Formal Methods. Springer, 2011, pp. 407–414.
28. De Moura, L.; Bjørner, N. Z3: An efficient SMT solver. In Proceedings of the International conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2008, pp. 337–340.
29. Cook, B. Formal reasoning about the security of amazon web services. In Proceedings of the International Conference on Computer Aided Verification. Springer, 2018, pp. 38–47.
30. Wang, Y.; Le, H.; Gotmare, A.; Bui, N.; Li, J.; Hoi, S. Codet5+: Open code large language models for code understanding and generation. In Proceedings of the Proceedings of the 2023 conference on empirical methods in natural language processing, 2023, pp. 1069–1088.
31. Copet, J.; Carbonneaux, Q.; Cohen, G.; Gehring, J.; Kahn, J.; Kossen, J.; Kreuk, F.; McMilin, E.; Meyer, M.; Wei, Y.; et al. Cwm: An open-weights llm for research on code generation with world models. *arXiv preprint arXiv:2510.02387* **2025**.
32. Grattafiori, A.; Dubey, A.; Jauhri, A.; Pandey, A.; Kadian, A.; Al-Dahle, A.; Letman, A.; Mathur, A.; Schelten, A.; Vaughan, A.; et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* **2024**.
33. Yang, A.; Li, A.; Yang, B.; Zhang, B.; Hui, B.; Zheng, B.; Yu, B.; Gao, C.; Huang, C.; Lv, C.; et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388* **2025**.
34. Kamath, A.; Ferret, J.; Pathak, S.; Vieillard, N.; Merhej, R.; Perrin, S.; Matejovicova, T.; Ramé, A.; Rivière, M.; Rouillard, L.; et al. Gemma 3 technical report. *arXiv preprint arXiv:2503.19786* **2025**, *4*.
35. Manik, M.M.H.; Wang, G. Gemma 4, Phi-4, and Qwen3: Accuracy-Efficiency Tradeoffs in Dense and MoE Reasoning Language Models. *arXiv preprint arXiv:2604.07035* **2026**.
36. Cao, R.; Chen, M.; Chen, J.; Cui, Z.; Feng, Y.; Hui, B.; Jing, Y.; Li, K.; Li, M.; Lin, J.; et al. Qwen3-coder-next technical report. *arXiv preprint arXiv:2603.00729* **2026**.
37. White, J.; Fu, Q.; Hays, S.; Sandborn, M.; Olea, C.; Gilbert, H.; Elnashar, A.; Spencer-Smith, J.; Schmidt, D.C. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382* **2023**.
38. Giray, L. Prompt engineering with ChatGPT: a guide for academic writers. *Annals of biomedical engineering* **2023**, *51*, 2629–2633.
39. Reynolds, L.; McDonell, K. Prompt programming for large language models: Beyond the few-shot paradigm. In Proceedings of the Extended abstracts of the 2021 CHI conference on human factors in computing systems, 2021, pp. 1–7.
40. Tihanyi, N.; Charalambous, Y.; Jain, R.; Ferrag, M.A.; Cordeiro, L.C. A new era in software security: Towards self-healing software via large language models and formal verification. In Proceedings of the 2025 IEEE/ACM International Conference on Automation of Software Test (AST). IEEE, 2025, pp. 136–147.
41. Gulwani, S.; Polozov, O.; Singh, R. Program synthesis. *Foundations and Trends in Programming Languages* **2017**, *4*, 1–119.
42. Ringer, T.; Palmskog, K.; Sergey, I.; Milos, G.; Tatlock, Z. QED at large: A survey of engineering of formally verified software. *Foundations and Trends in Programming Languages* **2019**, *5*, 102–281.
43. Jones, C.B.; Misra, J. *Theories of programming: the life and works of Tony Hoare*; ACM, 2021.
44. Yang, Z.; Wang, W.; Casas, J.; Cocchini, P.; Yang, J. Towards a correct-by-construction FHE model. *Cryptology ePrint Archive* **2023**.
45. Cassez, F.; Fuller, J.; Ghale, M.K.; Pearce, D.J.; Quiles, H.M. Formal and executable semantics of the ethereum virtual machine in dafny. In Proceedings of the International Symposium on Formal Methods. Springer, 2023, pp. 571–583.
46. Li, L.; Zhu, M.; Cleaveland, R.; Nicolellis, A.; Lee, Y.; Chang, L.; Wu, X. Qafny: A quantum-program verifier. *arXiv preprint arXiv:2211.06411* **2022**.
47. Garavel, H.; Ter Beek, M.H.; Van De Pol, J. The 2020 expert survey on formal methods. In Proceedings of the International Conference on Formal Methods for Industrial Critical Systems. Springer, 2020, pp. 3–69.
48. Irfan, A.; Porncharoenwase, S.; Rakamarić, Z.; Rungta, N.; Torlak, E. Testing Dafny (experience paper). In Proceedings of the Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, 2022, pp. 556–567.
49. Chakarov, A.; Fedchin, A.; Rakamarić, Z.; Rungta, N. Better counterexamples for Dafny. In Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2022, pp. 404–411.

 doi:10.20944/preprints202604.1772.v1




50. First, E.; Rabe, M.N.; Ringer, T.; Brun, Y. Baldur: Whole-proof generation and repair with large language models. In Proceedings of the Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2023, pp. 1229–1241.
51. Jiang, A.Q.; Li, W.; Tworkowski, S.; Czechowski, K.; Odrzygóźdź, T.; Miłoś, P.; Wu, Y.; Jamnik, M. Thor: Wielding hammers to integrate language models and automated theorem provers. *Advances in Neural Information Processing Systems* **2022**, *35*, 8360–8373.
52. Czajka, Ł.; Ekici, B.; Kaliszyk, C. Concrete semantics with Coq and CoqHammer. In Proceedings of the International Conference on Intelligent Computer Mathematics. Springer, 2018, pp. 53–59.
53. Wu, Y.; Jiang, A.Q.; Li, W.; Rabe, M.; Staats, C.; Jamnik, M.; Szegedy, C. Autoformalization with large language models. *Advances in neural information processing systems* **2022**, *35*, 32353–32368.
54. Madaan, A.; Zhou, S.; Alon, U.; Yang, Y.; Neubig, G. Language models of code are few-shot commonsense learners. In Proceedings of the Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, 2022, pp. 1384–1403.
55. Frieder, S.; Pinchetti, L.; Chevalier, C.; Griffiths, R.R.; Salvatori, T.; Lukasiewicz, T.; Petersen, P.; Berner, J. Mathematical capabilities of chatgpt. *Advances in neural information processing systems* **2023**, *36*, 27699–27744.
56. Narkawicz, A.; Munoz, C.A.; Dutle, A.M. The MINERVA software development process. In Proceedings of the NASA Formal Methods Symposium (NFM) 2017, 2017, number NF1676L-26800.
57. Lewis, G.A.; Comella-Dorda, S.; Gluch, D.P.; Hudak, J.; Weinstock, C. Model-based verification: Analysis guidelines. Technical report, 2001.
58. Nashid, N.; Sintaha, M.; Mesbah, A. Retrieval-based prompt selection for code-related few-shot learning. In Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). IEEE, 2023, pp. 2450–2462.
59. Tufano, R.; Pascarella, L.; Bavota, G. Automating code-related tasks through transformers: The impact of pre-training. In Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). IEEE, 2023, pp. 2425–2437.
60. Sellergren, A.; Kazemzadeh, S.; Jaroensri, T.; Kiraly, A.; Traverse, M.; Kohlberger, T.; Xu, S.; Jamil, F.; Hughes, C.; Lau, C.; et al. Medgemma technical report. *arXiv preprint arXiv:2507.05201* **2025**.
61. Sun, C.; Sheng, Y.; Padon, O.; Barrett, C. Clover: Clo sed-loop ver ifiable code generation. In Proceedings of the International Symposium on AI Verification. Springer, 2024, pp. 134–155.
62. Austin, J.; Odena, A.; Nye, M.; Bosma, M.; Michalewski, H.; Dohan, D.; Jiang, E.; Cai, C.; Terry, M.; Le, Q.; et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* **2021**.
63. Banerjee, D.; Bouissou, O.; Zetzsche, S. DafnyPro: LLM-Assisted Automated Verification for Dafny Programs. *arXiv preprint arXiv:2601.05385* **2026**.
64. Loughridge, C.; Sun, Q.; Ahrenbach, S.; Cassano, F.; Sun, C.; Sheng, Y.; Mudide, A.; Misu, M.R.H.; Amin, N.; Tegmark, M. Dafnybench: A benchmark for formal software verification. *arXiv preprint arXiv:2406.08467* **2024**.
65. Wang, C.; Scazzariello, M.; Kostić, D.; Chiesa, M. Toward Automated, Contamination-free Dafny Benchmark Generation.
66. Baksys, M.; Zetzsche, S.; Bouissou, O.; Delmas, R.; Kong, S.; Holden, S.B. ATLAS: Automated Toolkit for Large-Scale Verified Code Synthesis. *arXiv preprint arXiv:2512.10173* **2025**.
67. Ma, L.; Liu, S.; Li, Y.; Xie, X.; Bu, L. Specgen: Automated generation of formal program specifications via large language models. In Proceedings of the 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE). IEEE, 2025, pp. 16–28.
68. The Dafny Project. *Dafny Reference Manual*. Amazon Web Services, 2024. Accessed: Apr. 22, 2026.
69. Misu, M.R.H.; Lopes, C.V.; Ma, I.; Noble, J. Towards ai-assisted synthesis of verified dafny methods. *Proceedings of the ACM on Software Engineering* **2024**, *1*, 812–835.
70. OpenAI. Introducing GPT-OSS: Open-Weight Reasoning Models, 2025. Accessed: Apr. 22, 2026.
71. Google DeepMind. Gemma 4: Next-Generation Open Multimodal Models, 2026. Accessed: Apr. 22, 2026.
72. Alibaba Qwen Team. Qwen3.5 and Qwen3.6-MoE: Advancing Open-Weight Foundation Models, 2026. Accessed: Apr. 22, 2026.
73. Alibaba Qwen Team. Qwen3-Coder-30B: Specialized Models for Agentic Code Intelligence, 2025. Accessed: Apr. 22, 2026.
74. Mistral AI. Codestral-22B-v0.1: An Open-Weight Model for Professional Coders, 2024. Accessed: Apr. 22, 2026.

 doi:10.20944/preprints202604.1772.v1




75. Kulal, S.; Pasupat, P.; Chandra, K.; Lee, M.; Padon, O.; Aiken, A.; Liang, P.S. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems* **2019**, *32*.
76. Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H.P.D.O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* **2021**.
77. Achiam, J.; Adler, S.; Agarwal, S.; Ahmad, L.; Akkaya, I.; Aleman, F.L.; Almeida, D.; Altenschmidt, J.; Altman, S.; Anadkat, S.; et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* **2023**.
78. Troshin, S.; Mohammed, W.; Meng, Y.; Monz, C.; Fokkens, A.; Niculae, V. Control the Temperature: Selective Sampling for Diverse and High-Quality LLM Outputs. *arXiv preprint arXiv:2510.01218* **2025**.
79. Ryan, A.; Khalil, I.; Jahid, A.A.; Erfan, M.; Park, S.; Rahman, A.A.U.; Rahman, M.R. Mind the Gap: Evaluating LLMs for High-Level Malicious Package Detection vs. Fine-Grained Indicator Identification. *arXiv preprint arXiv:2602.16304* **2026**.
80. The Dafny Project. Dafny NuGet Package: The Dafny Compiler and Verifier. NuGet Package Manager, 2024. Version 4.x.x, Accessed: Apr. 22, 2026.
81. Siegmund, J.; Siegmund, N.; Apel, S. Views on internal and external validity in empirical software engineering. In Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. IEEE, 2015, Vol. 1, pp. 9–19.
82. Feldt, R.; Magazinius, A. Validity threats in empirical software engineering research-an initial survey. In Proceedings of the Seke, 2010, pp. 374–379.