Formally Specifying the Intended Behavior of the Program: LLM-Driven Neuro-Symbolic Program Specification Synthesis

Cheng Wen[1], **Junjie Hu**[1], **Yikun Hu**[2], **Jie Su**[1], **Bin Yu**[2], **Dugang Liu**[3], **Zhiwu Xu**[3], **Weidi Sun**[4], **Shengchao Qin**[1,2*], **Cong Tian**[2]

[1] Guangzhou Institute of Technology, Xidian University

[2] School of Computer Science and Technology, Xidian University

[3] College of Computer Science and Software Engineering, Shenzhen University

[4] School of Mathematical Sciences, Peking University

wencheng@xidian.edu.cn, {24181214428,23079100008}@stu.xidian.edu.cn

Abstract

Formal verification can provide strong mathematical guarantees about software correctness, but it typically requires developers to write detailed formal specifications (*e.g.*, contracts and loop invariants), which is costly and error-prone. We introduce AUTOSPEC$^+$, an LLM-driven neuro-symbolic demonstration system that reframes specification writing as constrained structured synthesis: large language models generate candidate specifications at the granularity of proof-relevant program components, while a symbolic verifier acts as a deterministic critic that checks legality, satisfiability, and proof adequacy, rejecting or repairing candidates in an iterative loop. This design turns unconstrained text generation into constrained structured synthesis, substantially reducing hallucinations and producing proof-ready annotations. We evaluate AUTOSPEC$^+$ on seven benchmark suites, showing strong effectiveness. We release an open-source, Dockerized system with ensemble LLM backends and inter-modular verification support for reproducible demonstration and deployment[1].

1 Introduction

Software failures remain a persistent source of security vulnerabilities, financial loss, and safety risks. Formal verification addresses this problem by proving, with mathematical rigor, that a program satisfies a target property (*e.g.*, memory safety assertions or functional correctness) (Sieber, 2013; Hähnle and Huisman, 2019). In practice, however, deductive formal verification is bottlenecked by the need for *formal specifications*, such as precise contracts and loop invariants that describe intended behavior in a machine-checkable specification language (Hatcliff et al., 2012) (*e.g.*, ACSL (Baudin et al., 2021) for C programs, JML (Leavens et al., 2006) for Java programs, Verus (Lattuada et al., 2023) for Rust programs). Even for moderately sized C code, writing such specifications typically requires expertise and substantial manual effort, which limits the adoption of formal verification in mainstream development.

Recent advances in large language models (LLMs) suggest a compelling alternative: ask an LLM to produce the formal specifications from the source code (Wen et al., 2024; Chen et al., 2025; Yang et al., 2025; Zhang et al., 2026; Hu et al., 2026). Yet, specification synthesis is a high-precision, structured generation task with hard semantic constraints. Unlike natural language summaries, incorrect specifications are not "approximately right": they are either rejected by the verifier (due to syntax errors or inconsistencies) or, more subtly, they may be well-formed but too weak to establish the desired proof. Naïvely prompting an LLM with an entire program often yields plausible-looking but unverifiable outputs, and repeated interactions can accumulate errors.

This paper presents AUTOSPEC$^+$, a tool that makes specification writing a *neuro-symbolic* agent loop. The core idea is simple: use LLMs as proposal generators for specifications (the neural part), and use deductive verification as a deterministic critic that enforces correctness constraints (the symbolic part). AUTOSPEC$^+$ targets ACSL (ANSI/ISO C Specification Language (Baudin et al., 2021)), a formal annotation language for C programs, and uses FRAMA-C (Kirchner et al., 2015) and and its WP-plugin (Blanchard, 2020; Blanchard et al., 2024) as the formal verification backend. Given unannotated C code and a property to verify, AUTOSPEC$^+$ iteratively produces ACSL annotations, checks them with the verifier, and refines

*Corresponding authors: Shengchao Qin.

[1]The source code and an out-of-the-box Docker image of AUTOSPEC$^+$ are available at: https://github.com/Xidian-ICTT-GZ/AutoSpec.

The full experiment result can be accessed at: https://huggingface.co/spaces/saki222/experient_result_full.

The demonstration video is available at: https://youtu.be/RZwsio2x6YU.

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations), pages 672–682

them using verifier feedback until the property is proved or a budget is reached.

A key challenge for LLM-driven specification generation is that the proofs (*e.g.*, loop invariants and contracts) must align across multiple functions and nested control flow. AUTOSPEC$^+$ therefore performs *proof-aware decomposition* using static analysis. It constructs an extended call graph in which both functions and loops are first-class nodes and synthesizes specifications bottom-up. At each step, the system prompts the LLM with only the proof-relevant code slice and already-validated specifications of callees, reducing context length and focusing generation on the current obligation. AUTOSPEC$^+$ can be viewed as an instance of *constrained structured generation*: the model generates candidate formal meaning representations of code behavior (ACSL), while a symbolic engine enforces hard constraints (provability under a formal semantics). This is a general reliability pattern for tool-augmented LLM agents: rather than trusting a single model output, we search over candidates and keep only those that pass a deterministic checker.

AUTOSPEC$^+$ is a tool designed to alleviate this bottleneck. Building on our earlier work (Wen et al., 2024) on program-specification synthesis using LLMs, AUTOSPEC$^+$ provides a practical, fully automated pipeline that transforms unannotated C code into a provable program annotated with machine-generated specifications. The tool integrates static analysis, program decomposition, structured prompting of LLMs, and automatic validation through FRAMA-C (Kirchner et al., 2015) and its WP-plugin (Blanchard, 2020).

This paper makes the following contributions:

- We introduce AUTOSPEC$^+$, a neuro-symbolic LLM agent that synthesizes formal specifications for C programs by integrating neural generation with symbolic verification. It utilizes a proof-aware decomposition and iterative refinement workflow that turns unconstrained text generation into verifier-approved structured generation.
- We demonstrate strong empirical performance and robustness across multiple tasks and benchmarks, including state-of-the-art verification success rates for partial correctness and high success in termination synthesis for total correctness.
- We deliver a demonstration tool that differs from prior research prototypes by emphasizing modularity engineering: the LLM backend is replaceable via a unified model ensemble interface,

```
1  /*@
2  requires \valid(buf);                                  1. Function Contract
3  requires \valid(buf+(0..num-1));
4  ensures \result == \sum(0, num-1, \lambda int k; buf[k])%255;
5  assigns \nothing; */
6  unint08 CheckSumAdd08(const unint08 *buf, unint32 num) {
7      unint32 sum = 0;
8      /*@
9      loop invariant 0 <= i <= num;                      2. Loop Invariant
10     loop invariant i == 0 ||
11         sum == \sum(0, i-1, \lambda int k; buf[k]);
12     loop assigns i, sum;
13     loop variant num - i; */                           3. Ranking Function
14   for (unint32 i = 0; i < num; i++) {
15       sum = sum + buf[i];
16     }
17     return (unint08)(sum%255);
18 }
19
20 /*@
21 requires pIp->rdLen == 19;                             4. Function Contract
22 requires pIp->pbuff[17] != pIp->frm;
23 requires pIp->pbuff[0] != 0xEB || pIp->pbuff[1] != 0x90;
24 ensures pIp->bComSuc == FALSE32;
25 ensures pIp->cntLenRdS == 0;
26 ensures pIp->cntUpdataS == 0;
27 ensures pIp->cntUpdataO == 0;
28 ensures pIp->cntHeadS == \old(pIp->cntHeadS) + 1;
29 ensures pIp->totalHeadS == \old(pIp->totalHeadS) + 1;
30 ensures pIp->cntHeadO == \old(pIp->cntHeadO) + 1;
31 ensures pIp->totalHeadO == \old(pIp->totalHeadO) + 1;
32 ensures pIp->pbuff[17] == pIp->frm;
33 ensures pIp->pbuff[0] != 0xEB || pIp->pbuff[1] != 0x90;
34 assigns pIp->bComSuc, pIp->cntLenRdS, pIp->cntUpdataS,
35 pIp->cntUpdataO, pIp->cntHeadS, pIp->totalHeadS,
36 pIp->cntHeadO, pIp->totalHeadO, pIp->frm; */
37 void Fg333saCheckFun(Fg333saCheck *pIp) {
38   unint08 chksum;
39   Fg333saCheck *p = (Fg333saCheck *)pIp;
40   p->bComSuc = FALSE32;
41   if (p->rdLen == 19) {
42     p->cntLenRdS = 0;
43     if (p->pbuff[17] != (p->frm)) {
44       p->cntUpdataS = 0;
45       p->cntUpdataO = 0;
46       p->frm = p->pbuff[17];
47       if ((p->pbuff[0] == 0xEB) && (p->pbuff[1] == 0x90)) {
48         p->cntHeadS = 0;
49         p->cntHeadO = 0;
50         chksum = CheckSumAdd08(&p->pbuff[0], 18);
51         ...
```

Figure 1: A simplified C program from an aerospace control system, annotated with ACSL specifications

AUTOSPEC$^+$ is open-sourced and Dockerized, and it supports multi-file inter-modular verification, and optionally generates ranking functions (loop variants) to lift proofs from partial to total correctness.

2 From Code to Verifiable Specification

This section answers the key questions for demonstrations: what problem the system solves, why it matters, what is novel, how it compares to existing systems, and who the target audience is.

2.1 Illustrative Industrial Example

We ground the discussion with an illustrative industrial software IP component `Fg333saCheckFun`, obtained from China Academy of Space Technology (CAST) (Ma et al., 2026b,a). This IP implements safety-relevant communication checking in an aerospace control setting: it validates an input frame (length, update/frame count, header, checksum) and updates multiple *continuous* and *cumu-*

01_Validate data length. If the length is not 19, increment both the continuous length error counter and the cumulative length error counter by 1, then return a validation failure.
02_Check if the frame count is updated. If the frame count does not update, increment both the continuous data update error counter and the cumulative data update error counter by 1, then return a validation failure.
03_Validate the frame header. If the frame header matches the expected value, reset the continuous frame header error counter.
04_Validate whether the checksum is correct. Calculate the checksum of the data. If the checksum is correct, reset the continuous checksum error counter and mark the communication as successful; otherwise, increment both the continuous and cumulative checksum error counters.
05_Update continuous length read counter. If the frame count matches the expected value, increment both the continuous and cumulative length read counters.

Figure 2: The Requirement `Fg333saCheckFun`

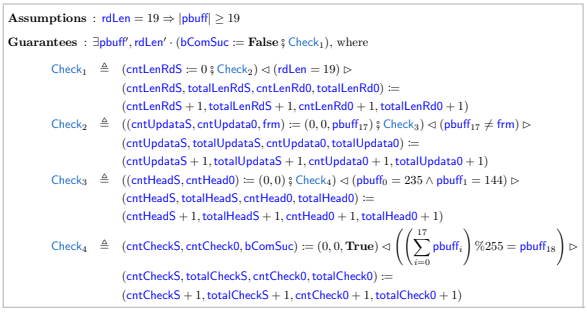

Figure 3: The Formal Model of `Fg333saCheckFun`

lative counters that track communication health. The intended behavior is specified in a structured natural-language requirement document (Figure 2). Figure 1 shows a simplified implementation and the kinds of ACSL specifications needed for formal verification, including function contracts, loop invariants, and ranking functions for termination. Finally, Figure 3 illustrates a corresponding formal behavioral model with assumptions and contract-style guarantees, which actually exists in the aerospace embedded software IP database.

2.2 What Problem Does the System Solve?

AUTOSPEC$^+$ addresses the formal specification bottleneck in deductive program verification. Although modern verifiers can prove that a C program satisfies a target property, they typically require developers to manually write *machine-checkable specifications* (*e.g.*, function contracts and loop invariants) that precisely describe the program's intended behavior. In practice, this manual step is often the dominant cost and the main barrier to adoption. Given C code, AUTOSPEC$^+$ automatically synthesizes such specifications and outputs *proof-ready annotated code* that an off-the-shelf toolchain (Frama-C/WP) can check.

Considering the example, the verifier cannot establish that the implementation matches the natural-language requirements in Figure 2 unless

Table 1: C Program Specification Generation Tools

Method/Tool	Loop Invariant	Ranking Function	Function Contract	Complex Struct.	Inter-modular	Model Ensemble
Learning-based methods						
Code2Inv (Si et al., 2018)	✔	✘	✘	✘	✘	–
CLN2INV (Ryan et al., 2020)	✔	✘	✘	✘	✘	–
LIPuS (Yu et al., 2023)	✔	✘	✘	✘	✘	–
Clause2Inv (Cao et al., 2025)	✔	✘	✘	✘	✘	–
LLM agent-based methods						
AutoSpec (Wen et al., 2024)	✔	✘	✔	✔	✘	✘
ACInv (Liu et al., 2025)	✔	✘	✘	✔	✘	✘
SLD-Spec (Chen et al., 2025)	✔	✘	✔	✔	✘	✘
AUTOSPEC$^+$	✔	✔	✔	✔	✔	✔

the code is augmented with formal specifications that connect: 1. a precise `Function Contract` for `CheckSumAdd08` that links the return value to the buffer sum. 2. a `Loop Invariant` summarizing partial checksum accumulation. 3. a `Ranking Function` (loop variant) ensuring progress and enabling total correctness proofs. 4. a `Function Contract` for `Fg333saCheckFun` that characterizes how counters and state variables change under requirement-relevant conditions.

Without such specifications, the verifier cannot modularly reason about the function's effects. Prompt-only LLM outputs are also unreliable in this setting: they may be syntactically invalid for ACSL, inconsistent with the implementation, or too weak to discharge proof obligations.

2.3 Why It Matters?

Trustworthy system software relies heavily on verifiable C code, where failures can be safety-critical. Formal verification can provide stronger guarantees than testing, but its adoption is limited because specification writing requires specialized expertise and time. By synthesizing verifier-approved specifications, AUTOSPEC$^+$ reduces the human cost of proof construction and makes it more practical to specify the actual behavior of the program.

Once a program's behavior is expressed in ACSL formal language, a symbolic verifier can check whether the specification is consistent with the code (*i.e.*, the implementation satisfies the contract and invariants). After consistency with the code is established, the resulting specification becomes a concrete artifact that can be easily audited against natural language requirements (Figure 2) by developers. Moreover, once verified, a contract serves as a reusable formal model of the function's behavior (Figure 3): callers can rely on the verified interface without re-verifying internal details, enabling compositional reuse in larger systems.

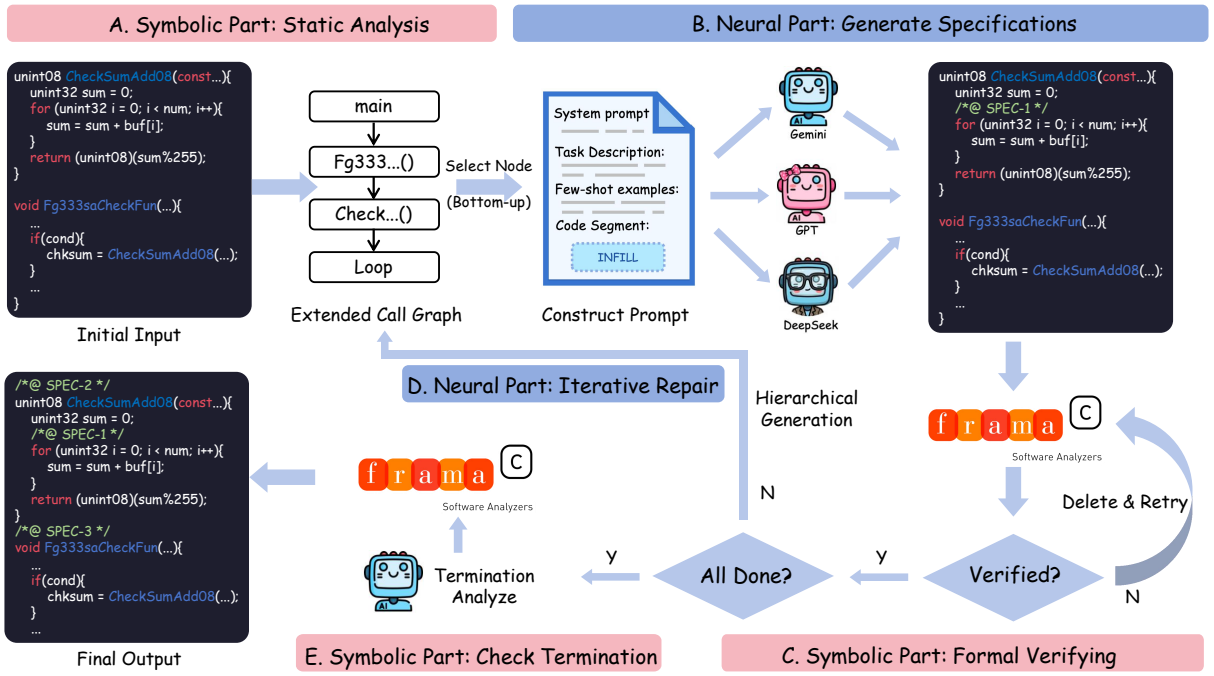

Figure 4: Overall Workflow of AUTOSPEC$^+$'s Neuro-symbolic Loop

2.4 What is Novel?

AUTOSPEC$^+$ operationalizes a neuro-symbolic agent loop for C program specification synthesis: LLMs propose ACSL candidates (*neural*), and Frama-C/WP serves as a deterministic critic (*symbolic*) that checks legality/satisfiability/adequacy and drives iterative repair. Beyond the algorithmic loop, AUTOSPEC$^+$ is engineered as a deployable tool: it provides a replaceable LLM backend (with an ensemble interface), supports inter-modular verification, and can synthesize ranking functions to upgrade from partial to total correctness.

2.5 How it Compares to Existing Systems?

Table 1 summarizes representative C specification-generation systems. Prior work largely falls into two categories. Learning-based approaches largely focus on loop invariants and typically do not generate full function contracts, termination arguments, or inter-modular specifications needed for end-to-end verification of realistic C components. Recent LLM agent-based systems can generate richer annotations, but commonly lack robust support for termination, multi-file inter-modular verification, and replaceable/ensemble model backends. AUTOSPEC$^+$ provides these capabilities in one end-to-end, verifier-in-the-loop pipeline, moving from *plausible-looking* specifications to *verifiable* specifications that can be reused as certified interfaces.

2.6 Who is the Target Audience?

This paper targets two communities: For NLP/AI researchers, AUTOSPEC$^+$ demonstrates a concrete pattern for reliable LLM agents: enforce hard semantic constraints by integrating a symbolic critic into generation. For SE researchers, AUTOSPEC$^+$ provides a practical workflow to formalize intended behavior, validate specifications against code, and reuse verified contracts in larger systems.

3 System Overview

Figure 4 shows the overview of AUTOSPEC$^+$ as a neuro-symbolic loop that turns source code into *verifier-approved* specifications. The system alternates between (i) neural parts, where an LLM proposes candidate ACSL specifications, and (ii) symbolic parts, where a deductive verifier validates those candidates and provides feedback. The loop continues until the target proof succeeds or a budget is reached. Below we describe the five main parts (A–E) highlighted in Figure 4.

3.1 A. Static Analysis (Symbolic)

AUTOSPEC$^+$ first performs static analysis to decide *where* specifications are needed and *in what order* to synthesize them. It builds an extended call graph where both functions and loop bodies are nodes, reflecting proof dependencies: a caller can only be verified once its callees are summarized by contracts, and a loop can only be verified once it

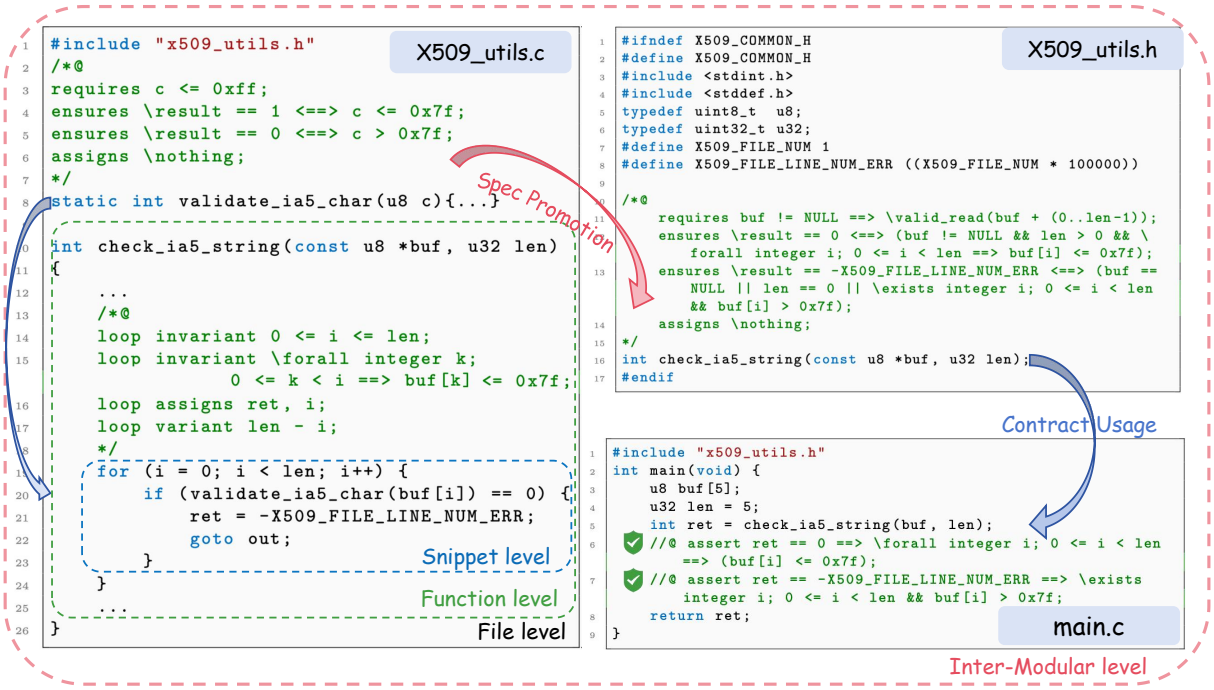

Figure 5: An X509 Parser Example with Multiple Files

has invariants (and optionally a variant). This graph yields a *bottom-up* schedule, so leaf components (*e.g.*, the loop in `CheckSumAdd08` in Figure 1) are handled before higher-level logic. For multi-file projects, the analysis also resolves cross-file calls and identifies header interfaces that must expose verified contracts for modular verification.

3.2 B. Generate Specifications with LLMs (Neural)

Following the bottom-up schedule, AUTOSPEC$^+$ synthesizes ACSL specifications component-by-component. For a target node (function or loop), the system builds a structured prompt containing: (i) the relevant code slice, (ii) already-validated specifications of callees (if any), and (iii) few-shot examples for verified ACSL contracts/invariants/-variants (refer to Appendix B). An LLM then proposes a set of candidate specifications. Optionally, a model-ensemble controller can query multiple LLMs and pool candidates to improve robustness.

3.3 C. Formal Verifying (Symbolic)

Candidate specifications are not trusted by default. After the LLM proposes ACSL specification, AUTOSPEC$^+$ invokes FRAMA-C/WP (Blanchard, 2020) as a symbolic checker that decides which candidates are satisfiable. The checker validates each candidate along three dimensions: (1) *legality*: ACSL syntax and type-checks, (2) *satisfiability*: the code can satisfy the specification, and (3) *adequacy*: the specification is strong enough to discharge the current proof obligations. Only verifier-approved clauses are retained and composed into the annotated program, making the verifier a deterministic *critic* that filters hallucinations and prevents incorrect specifications from propagating.

3.4 D. Iterative Repair (Neural)

AUTOSPEC$^+$ synthesizes and verifies specifications node-by-node, following the bottom-up order of the extended call graph. When verification fails, AUTOSPEC$^+$ does not restart from scratch. Instead, it begins a new iteration that reuses all verifier-approved specifications from previous rounds and focuses only on the remaining failing components (typically moving from leaf nodes upward to the root as needed). This iterative reuse-and-repair strategy reduces error accumulation and avoids regenerating already-correct specifications.

3.5 E. Check Termination (Symbolic)

AUTOSPEC$^+$ can go beyond proving *partial correctness* ("if the program terminates, it is correct") and prove *total correctness* by also showing termination. To do this, AUTOSPEC$^+$ synthesizes variants that should strictly decrease on every loop iteration and never go below zero. The LLM proposes candidate variants, and the verifier proves whether the program terminates. This stage is optional and

Table 2: Effectiveness of AUTOSPEC$^+$ (including model ensemble and termination analysis)

Benchmark	Deepseek -v3.1	Qwen3 -235B	Kimi-K2	Gemini 2.5 Pro	GPT-4o	Model Ensemble	Success Rate	Termination Analysis	Verified Term. Rate
SyGuS (Alur et al., 2019)	93/133	101/133	104/133	131/133	105/133	132/133	99.30%	65/73/132	89.04%
OOPSLA-13 (Dillig et al., 2013)	27/46	22/46	26/46	41/46	23/46	41/46	89.13%	20/22/41	90.90%
Frama-C-Problems (Frama-C, 2026)	23/51	23/51	33/51	31/51	23/51	39/51	76.40%	37/37/39	100.00%
SV-COMP (Beyer, 2022)	6/21	2/21	4/21	18/21	4/21	18/21	85.71%	16/16/18	100.00%
X509 Parser (ANSSI-FR, 2026)	2/6	2/6	1/6	3/6	2/6	4/6	66.67%	4/4/4	100.00%

* Each cell reports the number of programs verified out of the total benchmark size. In Termination Analysis column, the format is Proven Termination / Intended Termination / Verified Partial Correctness. The difference between intended termination and verified partial correctness accounts for programs with inherent non-termination, as confirmed by manual inspection.

is enabled when users require termination guarantees in addition to functional correctness.

Outputs and artifacts. At the end of the workflow, AUTOSPEC$^+$ outputs: (i) the annotated C code with verifier-approved ACSL specifications, (ii) verification logs and proof summaries for transparency, and (iii) (when enabled) termination certificates via validated loop variants.

4 Additional Engineering Support

To make AUTOSPEC$^+$ more practical, we add three features that improve scalability and usefulness.

❶ **Inter-modular verification (multi-files).** AUTOSPEC$^+$ analyzes cross-file dependencies and performs *specification promotion*: once a callee contract is verified in a `.c` file, it is lifted into the corresponding `.h` interface so callers can be verified modularly (the red arrow in Figure 5).

❷ **Model ensemble.** The LLM is a pluggable component. AUTOSPEC$^+$ supports a unified model interface and an optional ensemble controller that queries multiple LLMs and accepts any verifier-approved candidate, reducing brittleness.

❸ **Reproducible deployment.** We release AUTOSPEC$^+$ as an open-source, Dockerized package that bundles the static analyzer, LLM agent, Frama-C/WP, and scripts for one-command reproduction.

Case Study: X.509 Parser (Inter-modular). We demonstrate inter-modular verification on a simplified X.509 parser example (Ebalard et al., 2019) with two modules: a utility module `x509_utils.c` implementing IA5-string validation and a caller `main.c` containing safety assertions about `buf` after calling `check_ia5_string`. The proof requires a precise behavioral contract for the utility function, which must be exposed via `x509_utils.h`, as shown in Figure 5. AUTOSPEC$^+$ builds the proof bottom-up across four levels:

- *Snippet level:* synthesize loop invariants (and optional variants) for loop bodies in `x509_utils.c`.

Table 3: Comparison with SOTA Approaches (Pass@5)

Method	SyGuS	OOPSLA-13	Frama-C	SV-COMP	X509	Overall
Learning-based methods						
Code2Inv	54.88%	23.91%	-	-	-	32.68%
Clause2Inv	99.30%	-	-	-	-	51.36%
CLN2Inv	93.23%	0.00%	-	-	-	48.25%
LiPuS	93.23%	-	-	-	-	48.25%
Single LLM with Prompt (Appendex B)						
Qwen2.5-Coder-7B	2.30%	2.20%	0.00%	5.00%	0.00%	1.94%
GPT-3.5-turbo	3.70%	4.40%	17.60%	5.00%	0.00%	6.61%
GPT-4o	12.00%	13.10%	27.50%	22.00%	0.00%	15.95%
LLM agent-based methods						
AutoSpec	85.71%	82.60%	60.78%	76.20%	100%	79.76%
ACINV	79.70%	76.10%	48.30%	76.20%	-	70.42%
SLD-Spec	-	-	62.74%	-	-	12.45%
AUTOSPEC$^+$	99.30%	89.13%	76.40%	85.71%	66.67%	91.05%

* '-' indicates that the method was unable to tested on this benchmark, due to configuration issues (these learning based methods only support loop invariant generation; SLD-Spec failed to run on all benchmarks except Frama-C-Problems).

- *Function level:* synthesize function contract for helper functions (*i.e.*, `validate_ia5_char`).
- *File level:* verify `x509_utils.c` using the synthesized helper contracts.
- *Inter-modular level:* promote the verified contract of `check_ia5_string` into `x509_utils.h`, then verify `main.c` using the promoted interface contract, successfully verifying two safety assertions.

5 Evaluation

We evaluated AUTOSPEC$^+$ against existing SOTA systems on seven benchmarks: SyGuS (Alur et al., 2019), OOPSLA-13 (Dillig et al., 2013), Frama-C-problem (Frama-C, 2026), SV-COMP (Beyer, 2022), the real-world X509 project (ANSSI-FR, 2026), FM-bench-verified (fm universe, 2026a), and Live-FM-Bench (fm universe, 2026b).

5.1 Overall Effectiveness

Table 2 reports effectiveness for individual LLM backends and the model ensemble, together with termination analysis results. Gemini 2.5 Pro demonstrates strong empirical performance among the five LLMs we evaluated. With the model-ensemble controller, AUTOSPEC$^+$ verifies 91.05%

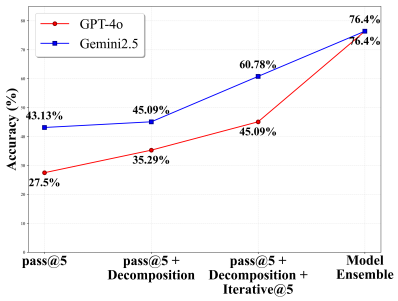

Figure 6: Ablation Study on *Frama-C-Problems* benchmark

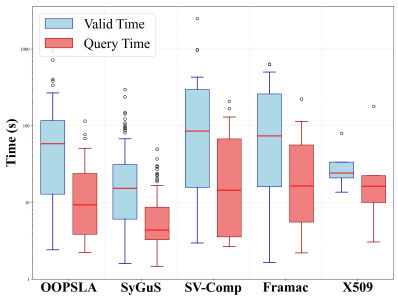

Figure 7: Time Cost Analysis (using *GPT-4o* as model backend)

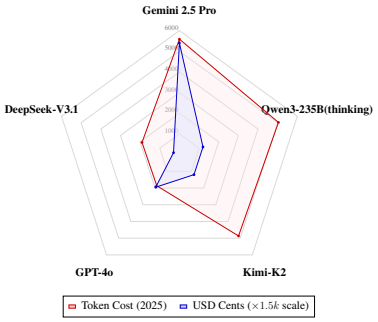

Figure 8: Token Cost and Expenditure Analysis

of programs overall (pass@5), achieving strong success rates across datasets. Across all suites, the ensemble consistently improves over any single model, indicating robustness to model choice and LLM stochasticity. On the dynamically evolving benchmark (FM-bench-verified + Live-FM-Bench, 630 verified programs), Figure 9 shows that the AUTOSPEC$^+$ yields large gains over standalone prompting across all tested LLMs (Appendix C).

5.2 Comparison to SOTA Systems

Table 3 compares AUTOSPEC$^+$ against representative SOTA systems and prompt-only baselines. AUTOSPEC$^+$ achieves the best overall verification rate (91.05%) among compared methods, outperforming learning-based methods (which typically target invariants only) and prompt-only LLM baselines (which rarely produce verifier-acceptable specifications). These results support AUTOSPEC$^+$'s key design choice: using symbolic verification as a deterministic critic to turn plausible LLM outputs into proof-ready specifications.

5.3 Termination Analysis

Termination analysis is AUTOSPEC$^+$'s new feature that extends verification from *partial correctness* to *total correctness*, which other systems do not support. In our experiments, AUTOSPEC$^+$ proves termination for 93.42% of the *intended-terminating* programs among those already verified for partial correctness (Table 2), showing that the same neuro-symbolic loop supports both functional and termination proof obligations.

5.4 Ablation Study

Figure 6 ablates AUTOSPEC$^+$'s key components on the Frama-C-Problems benchmark. Plain prompt-only LLM inference (Pass@5) verifies only 27.50% (GPT-4o) / 43.13% (Gemini 2.5 Pro). Adding proof-aware decomposition yields a small gain, while adding iterative refinement (Iter@5) provides a large improvement. Finally, enabling model ensemble attains 76.40%, showing that the ensemble further boosts performance.

5.5 Time Cost, Token Cost, and Expenditure

Figure 7 breaks down runtime into LLM querying and symbolic validation. Across benchmarks, symbolic validation dominates (typically < 100 s), while LLM querying is typically < 10 s per program, reflecting that AUTOSPEC$^+$ relies on generating *few* candidates and letting the verifier filter them. Figure 8 summarizes per-sample inference overhead: token usage is typically from 2 k to 6 k, with cost ranging from 0.2 to 3.6 cents depending on the model, motivating our replaceable/ensemble backend for balancing performance and expense.

6 Conclusions

AUTOSPEC$^+$ demonstrates a practical neuro-symbolic workflow for formally specifying the intended behavior of the program: LLMs propose ACSL specifications, and a symbolic verifier filters and repairs them until they become proof-ready. This bridges software engineering and AI: it lowers the barrier to using formal methods in real codebases while providing a general verifier-in-the-loop recipe for making LLM agents trustworthy under hard semantic constraints.

Acknowledgements

This work was supported in part by the National Natural Science Foundation of China (Nos. 62472339, 62302375, 62192734, 6240073166), the Basic Research Foundation of Shenzhen City (No. JCYJ20250604184202003), the China Postdoctoral Science Foundation funded project (No. 2023M723736), and the Fundamental Research Funds for the Central Universities.

References

Rajeev Alur, Dana Fisman, Saswat Padhi, Rishabh Singh, and Abhishek Udupa. 2019. Sygus-comp 2018: Results and analysis. *CoRR*, abs/1904.07146.

ANSSI-FR. 2026. a rte-free x.509 parser: https://github.com/ANSSI-FR/x509-parser, accessed: 2026-02-27.

Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. 2021. Acsl: Ansi/iso c specification.

Dirk Beyer. 2022. Progress on software verification: Sv-comp 2022. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 375–402, Cham. Springer International Publishing.

Allan Blanchard. 2020. Introduction to c program proof with frama-c and its wp plugin.

Allan Blanchard, Claude Marché, and Virgile Prevosto. 2024. Formally expressing what a program should do: the acsl language. In *Guide to Software Verification with Frama-C: Core Components, Usages, and Applications*, pages 3–80. Springer.

Weining Cao, Guangyuan Wu, Tangzhi Xu, Yuan Yao, Hengfeng Wei, Taolue Chen, and Xiaoxing Ma. 2025. Clause2inv: A generate-combine-check framework for loop invariant inference. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):1009–1030.

Zehan Chen, Long Zhang, Zhiwei Zhang, JingJing Zhang, Ruoyu Zhou, Yulong Shen, JianFeng Ma, and Lin Yang. 2025. Sld-spec: Enhancement llm-assisted specification generation for complex loop functions via program slicing and logical deletion. *arXiv preprint arXiv:2509.09917*.

Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. 2013. Inductive invariant generation via abductive inference. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, page 443–456, New York, NY, USA. Association for Computing Machinery.

Arnaud Ebalard, Patricia Mouy, and Ryad Benadjila. 2019. Journey to a rte-free x. 509 parser. In *Symposium sur la sécurité des technologies de l'information et des communications (SSTIC 2019)*, volume 186.

fm universe. 2026a. Fm-bench-verified: a manual-cleaned and verified benchmark which contains 280 c programs, properties under verification, together with ground-truth acsl specifications. https://huggingface.co/datasets/fm-universe/FM-bench-verified. Accessed: 2026-02-27.

fm universe. 2026b. Live-fm-bench: a dataset continuously updated and contamination-free evaluation benchmark of llms for program verification (a.k.a., formal specification generation). currently, it contains 360 c programs under verification together with the properties to be verified. https://huggingface.co/datasets/fm-universe/Live-FM-Bench. Accessed: 2026-02-27.

Frama-C. 2026. A repository dedicated to problems related to the verification of programs using the tool frama-c: https://github.com/manavpatnaik/frama-c-problems, accessed: 2026-02-27.

Reiner Hähnle and Marieke Huisman. 2019. Deductive software verification: from pen-and-paper proofs to industrial tools. *Computing and Software Science: State of the Art and Perspectives*, pages 345–373.

John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew Parkinson. 2012. Behavioral interface specification languages. *ACM Comput. Surv.*, 44(3).

Junjie Hu, Cheng Wen, Jialun Cao, Yikun Hu, Dugang Liu, Zhi Ma, Zhiwu Xu, and Shengchao Qin. 2026. When large language models meet formal theorem proving: A survey. In *International Conference on Knowledge Science, Engineering and Management*. Springer.

Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-c: A software analysis perspective. *Formal aspects of computing*, 27:573–609.

Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying rust programs using linear ghost types. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):286–315.

Gary T Leavens, Albert L Baker, and Clyde Ruby. 2006. Preliminary design of jml: A behavioral interface specification language for java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38.

Ruibang Liu, Minyu Chen, Ling-I Wu, Jingyu Ke, and Guoqiang Li. 2025. Enhancing automated loop invariant generation for complex programs with large language models. *Science of Computer Programming*, page 103387.

Zhi Ma, Xiao Liang, Cheng Wen, Chen Rui, Bin Gu, Shengchao Qin, Cong Tian, and Mengfei Yang. 2026a. Automated ltl specification generation from industrial aerospace requirements. In *International Symposium on Formal Methods*, pages 1–20. Springer.

Zhi Ma, Cheng Wen, Bin Yu, and Jie Su. 2026b. Integrating ensemble learning and large language models for efficient formal verification of ip-based aerospace systems. *Information Fusion*, 125:103466.

Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Jana. 2020. Cln2inv: Learning loop invariants with continuous logic network. In *International Conference on Learning Representations*.

Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018. Learning loop invariants for program verification. *Advances in Neural Information Processing Systems*, 31.

Kurt Sieber. 2013. *The foundations of program verification*. Springer-Verlag.

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

Fanpeng Yang, Xu Ma, Shuling Wang, Xiong Xu, Qinxiang Cao, Naijun Zhan, Xiaofeng Li, and Bin Gu. 2025. Automated synthesis of formally verified multi-abstraction function summaries. *arXiv preprint arXiv:2506.09550*.

Shiwen Yu, Ting Wang, and Ji Wang. 2023. Loop invariant inference through smt solving enhanced reinforcement learning. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 175–187.

Yuchen Zhang, Cheng Wen, Zhiwu Xu, Dugang Liu, Jialun Cao, Yuwei Liu, Shengchao Qin, and Cong Tian. 2026. Enhancing llm-based proof synthesis for rust programs via semantic chunking and hierarchical context expansion. In *International Symposium on Theoretical Aspects of Software Engineering*. Springer.

A Data Availability

AUTOSPEC$^+$ is licensed under the Apache 2.0 open-source license.

- The source code and out-of-the-box Docker image of AUTOSPEC$^+$ are available at: https://github.com/Xidian-ICTT-GZ/AutoSpec.
- The full experiment result can be accessed at: https://huggingface.co/spaces/saki222/experient_result_full.
- The demonstration Video is available at: https://youtu.be/RZwsio2x6YU.

A.1 Implementation Details

AUTOSPEC$^+$ is implemented as a command-line tool composed of modular backend components written in both C and Python. Its static analysis component is implemented as a Clang compiler plugin in C, leveraging the LLVM/Clang infrastructure to extract loop structures, build extended call graphs, and support multi-file, inter-modular level analysis. The verification backend relies on the open-source FRAMA-C framework and its WP plugin to validate the legality, satisfiability, and adequacy of synthesized specifications. The LLM agent is implemented in Python and communicates with one or more LLMs through a unified interface.

To facilitate deployment and reproducibility, we additionally provide a Docker container bundling all dependencies: Clang, FRAMA-C, Python scripts, and an LLM agent. Users may run AUTOSPEC$^+$ entirely inside the container, eliminating environment-configuration issues and enabling seamless artifact evaluation.

A.2 Usage

AUTOSPEC$^+$ provides a flexible execution mode for inter-modular level verification. Users interact with the tool through a driver script: `main.py`.

The script `main.py` executes AUTOSPEC$^+$ on a user-specified set of C source files. The parameter `-f` accepts one or more `.c` files, separated by commas. This is useful when verifying a project where multiple C files depend on each other. Header files (`.h`) do not need to be specified; they are automatically discovered and indexed using FRAMA-C's multi-file support. Users can select the backend models using the `-m` parameter.

Partial Correctness Proof for Single/Multi-File Verification

```
python3 main.py -f file1.c,file2.c,... -o output-dir
-m model1,model2,...
```

After partial correctness has been established using either of the above modes, `generate_variant.py` performs termination analysis. It synthesizes ranking functions and invokes FRAMA-C's WP plugin to validate termination, thereby upgrading verification results from partial correctness to total correctness. The script accepts either a list of C files (`-f`) or a directory (`-i`) matching the earlier modes.

Total Correctnerss Proof by Termination Analysis

```
python3 generate_variant.py -f file1.c,file2.c,... -o
output-dir -m model1,model2,...
```

Note that the `-m` (Models) and `-o` (Output) parameters are consistent across all three driver scripts, enabling the user to specify the LLM backend(s) and the destination directory where annotated code and verification logs are saved.

B Prompt Used in AUTOSPEC$^+$

Prompt for Loop Invariant

```
ACSL is a specification language for C programs that
conforms to the design by contract paradigm, utilizing
Hoare-style pre- and postconditions and invariants. You are
an expert in program verification, and please generate loop
invariant as C annotation comments at the infill location
(annotated by "/* @ »> INFILL «< */") using ACSL language.
Note that the ‘loop invariant‘ clause is a condition that
is true at the beginning and end of every loop iteration.
The ‘loop assign‘ is a condition that describes the set of
variables modified by the loop. You cannot use functions
defined in existing programs as part of the specification;
focus on describing the properties. I will provide you with
some few-shot examples:
{few-shot examples}
Now you need to generate proper loop invariant for the
following code:
{code}
Just show me the loop invariant and don't explain it.
```

Prompt for Function Contract (Pre/Post Condition)

```
ACSL is a specification language for C programs that
conforms to the design by contract paradigm, utilizing
Hoare-style pre- and postconditions and invariants. You
are an expert in program verification, and please generate
function contract as C annotation comments at the infill
location (annotated by "/* @ »> INFILL «< */") using ACSL
language. The pre- and post-conditions of functions are
written as ‘requires‘ and ‘ensures‘ clauses, respectively.
The memory locations that can be modified within a function
call can be specified with an ‘assigns‘ annotation (usually
known as frame condition) The return value of a function can
be accessed (in particular in the function's post-condition)
with the ‘
result‘ clause. Just show me the code and don't explain it.
I will provide you with a few-shot examples:
{few-shot examples}
Now you need to generate a proper loop invariant for the
following code:
{code}
Just show me the function contract and don't explain it.
```

Prompt for Termination

```
You are an expert assistant specializing in generating ACSL
specifications for the Frama-C framework. Your task is
to generate a single 'loop variant' clause based on the
provided context (loop annotations, loop condition, and
```

```
loop body).
Your response MUST adhere to the following strict rules: 1.
You MUST return ONLY the single-line 'loop variant' clause.
2. Do NOT include any explanations, preambles, or Markdown
formatting (like "'c ... "'). 3. The 'loop variant' must
satisfy ACSL requirements: it must be a non-negative integer
expression that strictly decreases on each loop iteration.
4. If the context is a 'for' loop like 'for (i = 0; i <
n; i++)', a good variant is 'n - i'. 5. If the context
is a 'while' loop like 'while (j < n)' with 'j++' in the
body, a good variant is 'n - j'. 6. If the context is a
decrementing 'while' loop like 'while (i > 0)' with 'i--' in
the body, a good variant is 'i'.
Example of a valid response:
loop variant n - j;
======= Start Context =======
Existing Loop Annotation
{annotation_block}

Loop Condition
{while_line}

Loop Body
{loop_body}
======= End Context =======

[Task]
Please generate a 'loop variant' clause for the above C
language while loop. The 'loop variant' must meet ACSL
requirements: it is a non-negative integer expression and
strictly decreases after each loop iteration. Please return
only a single-line 'loop variant' clause, e.g., loop variant
n - j;
```

C Supplementary Experimental Data

LiveFMBench is a dynamically evolving benchmark specifically designed for the automated synthesis of ACSL specifications. The dataset currently comprises two primary subsets: FM-bench-verified (fm universe, 2026a) and Live-FM-Bench (fm universe, 2026b), featuring a total of 630 verified, high-quality C programs. Figure 9 illustrates a significant performance leap when utilizing the AutoSpec$^+$ workflow compared to standalone model inference across all tested LLMs.

- First, AutoSpec$^+$ serves as a powerful performance multiplier for both Pass@1 and Pass@5 metrics. In the FM-bench-verified, for instance, the Pass@1 rate of Llama-3.3 surges from a mere 6.2% (green bar) to 51.5% (red bar) when integrated with the agent—an eight-fold increase. Similar trends are observed for the Pass@5 metric (purple vs. blue bars).
- Second, the data highlights the "performance equalization" effect of the agentic workflow. In standalone mode, there is a clear hierarchy among models. However, under the AutoSpec$^+$ framework, these disparities are drastically reduced. On the Live-FM-Bench subset, the agent enables even relatively "weaker" models like Qwen3-32B to achieve success rates comparable to more advanced reasoning models, effectively mitigating limitations in native zero-shot reasoning through structured engineering modules.

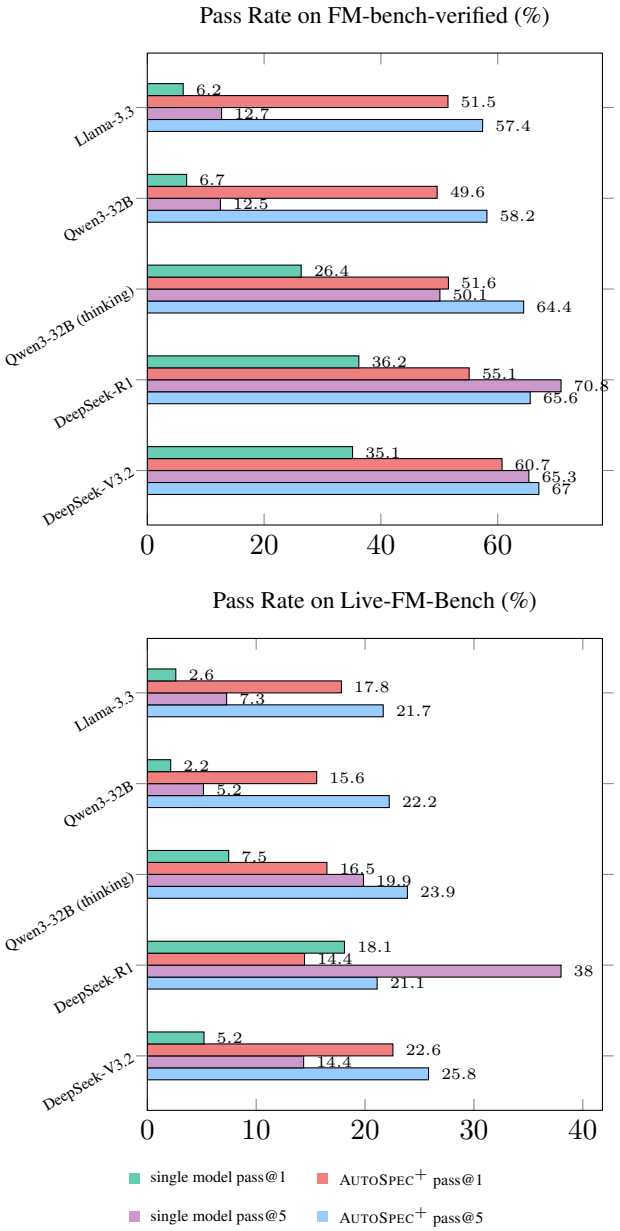

Figure 9: Performance on LiveFMbench

- Finally, the results on the Live-FM-Bench indicate increased task complexity, as evidenced by the overall lower success rates compared to FM-bench-verified. Despite this difficulty gradient, AutoSpec$^+$ maintains high stability, consistently outperforming the Pass@5 performance of standalone models with just its Pass@1 results in several cases.