Research Report: AI Security is a LangSec Problem

Max von Hippel
Benchify, Inc.
San Francisco, USA
Email: max@benchify.com

Evan Miyazono
Atlas Computing
San Francisco, USA
Email: evan@atlascomputing.org

***Abstract*—The rapid development of Artificial Intelligence (AI) systems, and particularly Large Language Models (LLMs), has already started changing how software is written in industry. In this work, we categorize two important features of modern AI systems – structured outputs and tool-use – and explain how the security of each is, inherently, a LangSec problem. We provide anecdotal evidence from the San Francisco startup ecosystem to illustrate how companies are currently using, deploying, and securing AI systems with these features. Based on these observations and our analysis of current practices, we identify three concrete research directions where the LangSec community can contribute to securing both the parsing of LLM outputs and the safe deployment of LLM-powered tools. This work should be read as a call-to-action for the LangSec community to tackle outstanding, and growing, security problems catalyzed by AI.**

I. Introduction

While machine learning dates back to the 1950s [1], historically models were special-purpose, trained for specific tasks like facial recognition or anomaly detection. This changed dramatically with OpenAI's release of ChatGPT in late 2022 [2], which demonstrated unprecedented capabilities in generating natural text and code from arbitrary prompts.

Since then, progress in Large Language Model (LLM) based artificial intelligence has accelerated rapidly. The computational resources used in model training have grown by approximately half an order of magnitude annually [3], and novel architectural improvements like Chain-of-Thought [4] have enhanced model reasoning capabilities. Combined with larger, higher-quality training datasets, these advances produced AI systems with remarkable performance across diverse domains. For instance, OpenAI's o3 model outperformed 99.8% of programmers on Codeforces [5], while AlphaGeometry2 achieved gold-medal performance on International Math Olympiad geometry problems, solving 84% of problems from 2000-2024 and contributing to a silver-medal solution at IMO 2024 [6].[1]

Beyond so-called frontier models from OpenAI, Anthropic, Meta, or Google, we now have open-source LLMs that can run on consumer hardware - from desktop computers to laptops and even phones - with capabilities exceeding the original ChatGPT [7], [8]. These models are being rapidly integrated into software development workflows, both as interactive programming assistants, and as embedded components within larger systems.

[1]Note that it is difficult to know to what extent the problems in benchmark sets are out-of-distribution. For example, IMO problems might be based on classical theorems occurring in textbooks on which o3 was trained.

LLMs are transforming software development by making the raw production of code fast and cheap. Platforms like GitHub Copilot [9], Cursor [10], and Continue [11] allow developers to delegate programming tasks to AI assistants working in parallel. This has led to "vibes coding" —– a paradigm where developers primarily interact with their code through an AI intermediary, requesting modifications prosaically and then accepting them into the codebase with minimal review [12]. While largely acceptable for personal projects, this style of coding is increasingly becoming the norm in professional settings, where the pressure for engineering velocity exceeds the pressure to write good code. Notably, vibes coding represents a logical departure from traditional code synthesis, which began with Church's work in 1963 [13], because the synthesis problem starts with a formal specification for what the code is intended to do, and only ends when the code can be guaranteed to satisfy the spec, whereas vibes coding begins with an informal specification and ends when the code is saved to the file (regardless of correctness).

As AI seeps into consumer products, developer workflows, and more, the associated risks grow in both complexity and magnitude. Broadly speaking, there are three fields of research which try to address these risks: AI Ethics, which studies ethical implications of AI systems such as social harms caused by AI-enabled advertising; AI Safety, which studies the acceleration of AI capabilities and the risk those capabilities pose to long-term human survival; and AI Security, which studies the security of AI systems both intrinsically and in composition with other software. In this work we focus on AI Security, although the risks we discuss have implications for the two former camps as well.

We classify security risks as being accidental, malicious, or what we call architectural. Accidental risks are risks caused by AI systems that make mistakes. For example, Dou et. al. [14] find that both open and closed-source LLMs make certain categories of mistakes when asked to generate code, which, they hypothesize, relate to both the data the LLMs are trained on (including flawed code written by humans) and skews between the distribution of real-world code and the distribution of problems found in LLM benchmarks. Malicious issues can arise when LLMs are modified to favor deleterious outputs, for example, calls to violence or the proliferation of political propaganda, or in the case of code-generation, code with subtle vulnerabilities or backdoors. Such issues can arise through many vectors, including jail-breaking or data poison-

ing (see [15] for more). Finally, it is generally useful to think of an AI system as having "goals" (similar to how a heat-seeking missile has goals) that are implicit in its architecture, training data, and training methodology, and these goals may diverge significantly from security considerations [16]. We describe these goal mismatches as architectural issues. While some research, such as mechanistic interpretability work (e.g. [17]), attempts to shed light on the goals of AI systems, at present it is generally fair to characterize these goals as being both unknowable and unmodifiable.

The primary mitigation strategies for AI risks are alignment (get the system to understand what you want and make decisions as your proxy), review (make sure the human understands and approves all AI outputs), and what we call a "formalization-based" paradigm (specify objective properties you want the output to have, and then check them). For the most part, AI research labs developing LLMs expect the complexity of AI-generated solutions to grow exponentially, making review infeasible. Moreover, it is not clear they are implementing robust strategies to achieve alignment on the relevant timescales. Many users of AI systems expect to review AI outputs, but real-world pressures and the growth of AI capabilities make this seem unlikely in commercial environments. Meanwhile, AI systems are increasingly utilizing two critical features (structured outputs and tools) which happen to be highly amenable to formalization-based approaches. As a result, we argue that the security of contemporary LLM-based AI systems is, inherently, a language-theoretic problem, where the goals of the system should be formalized and the formalization should be checked over the generated text.

The rest of this paper is organized as follows. We explain structured outputs and tool-use, and the language-theoretic challenges of securing each capability, in Sec. II. Then in Sec. III we give some anecdotal reports of prevailing attitudes about structured outputs, tool-use, and AI security in the San Francisco startup community, leveraging a limited survey of startup founders. This is meant to help an academic audience understand what is happening inside of startups today, and where the industry may be headed as AI systems become more capable and prevalent. Unfortunately, our sample size is very small and should be viewed as anecdotal at best. We survey prior works on the language-theoretic security of AI systems in Sec. IV, and discuss future research directions in Sec. V. We conclude in Sec. VI.

II. Challenges

As developers incorporate AI into their applications, they naturally hit two roadblocks. First, LLMs produce inconsistent outputs, in contrast to standard RESTful services that can follow predefined APIs with structured, parsable responses. The natural solution is to induce the LLM to produce a structured output, or, massage its unstructured output(s) into some fixed shape post-generation. Second, LLMs lack the ability to *do* things in the real world, such as look up facts in a database, execute code, or receive and send emails, severely limiting their real-world use-cases. This is solved through tool use, where the LLM can invoke a known tool using a special code; the tool is then executed server-side and a stateful response is fed to the LLM before it takes another step. To get a sense of how popular structured outputs and tool-use are in the wild, we asked Maitai[2], a service that hosts, fine-tunes, routes, and corrects LLMs, to analyze their customer traffic for both. Their data shows that approximately 21.2% of requests involve structured outputs and 32.7% involve tool calls, with minimal overlap between these categories. Both structured outputs and tool use present difficult language-theoretic security challenges, as we explain below.

A. Structured Outputs

The first LLM products were text-to-text models that mapped a prose prompt to an equally unstructured response (e.g., "What is the meaning of life?" $\mapsto$ "The pursuit of happiness."). But developers building products with LLMs wanted structured outputs, i.e., JSON or XML payloads with specific shapes that could be reliably deserialized into predefined data structures [18] (see also §3.1 of [19]). Responding to this need, researchers built a variety of tools to both prompt LLMs to produce structured outputs (e.g. [20], [21], [22]) and verify that these outputs type-check (e.g. [23], [24], see also [25]).[3]

The proliferation of structured outputs goes hand-in-hand with the proliferation of parsers for these outputs. These parsers are typically developed incrementally and in response to organic developer demand, without particular attention to how new features impact the language-theoretic complexity of the parser or its security. Structured output parsing is conventionally done in the same place as the primary application logic, without any special sandboxing or isolation. Thus, a savvy attacker could potentially breach an application by feeding it inputs crafted to induce deleterious LLM outputs which, in turn, exploit parser vulnerabilities server-side. The LangSec challenge is, therefore, to build secure, reliable parsers for LLM-generated structured outputs that can be safely run in an unisolated, real-time environment. These parsers are fundamentally different from traditional ones because they cannot assume anything about the LLM outputs they are fed, and moreover, if an LLM output fails to parse, the parser should output a clear error message that can be used to re-prompt the LLM for a (repaired) output.

B. Tool Use

Tool use, or function calling, refers to frameworks by which an AI is made aware of, and given the capability to query, one or more pre-existing tools [27]. Examples include querying a database [28], [29] or table [30], hitting an API [31], executing LLM-generated code [32], [33], or even moving a robot [34]. The LLM invokes the tool by outputting a special (textual) command, as illustrated in Fig. 1.

The problem with tool use is that typically the tool in question is executed on the server that hosts the rest of the application logic. The tool use capabilities of the popular LLM

[2]https://trymaitai.ai/

[3]For a brief history of these efforts, see [26].

```
{
  "role": "assistant",
  "content": [{
    "type": "text",
    "text": "<thinking>Check weather and
        rain forecast for SF</thinking>"
  }, {
    "type": "tool_use",
    "id": "tool_wx_01",
    "name": "get_weather",
    "input": {
      "location": "San Francisco, CA",
      "include_forecast": true
    }
  }]
}
```

Fig. 1: Tool use invocation where an LLM requests weather information [35].

APIs are not built with any kind of default isolation, nor does the corresponding documentation suggest it. The LangSec challenge is, therefore, to build a safety system around tool use that can enforce policies like, "the `database_query` never deletes data from the database" or "the `get_weather` tool invokes no more than 100 queries per second."

C. Language-Theoretic Implications

The core idea of LangSec is that hackers will stitch together unintended bits of computation ("attacker-defined abstractions" [36]) – such as side-effects from bugs, over-privileged APIs, hardware defects, etc. – to form programming languages with which they can program malicious logic. In particular, this frequently occurs in parsers, because parsing is "hard" in the sense that a sufficiently advanced parser will likely become a "weird machine" programmable by the data it is meant to interpret. As the type systems of structured output frameworks become increasingly complex, so too will the (currently mostly ad-hoc) parsers that parse them, leading to the inevitable rediscovery of all the security pitfalls the LangSec community has spent over a decade analyzing in traditional parsers and compilers. Hackers will find ways, such as prompt injection or data poisoning, to induce LLMs into producing malformed objects that exploit vulnerabilities in the parsers they are fed to. For example, suppose an OS vendor allows users to submit textual descriptions of custom boot images, with the most popular submissions (upon a vote) being automatically fed to an image generation model and the final images being distributed to all users. An attacker might then attempt prompt injection techniques to exploit a LogoFail vulnerability [37] in the image parser. In an ideal world, inputs and outputs of the AI systems would be parsed rigorously enough to protect even an insecure UEFI in this scenario.

Meanwhile, tool use gives AI systems direct access to the same tools that hackers have spent decades exploiting, and all those pre-existing exploits will still work in this brave new context, with the caveat that now the attacker must take the additional step of figuring out how to coerce the LLM into outputting a sufficiently evil sequence of tool invocations.[4] We give examples of potential tool use exploits in Table I.

Tool	Example Exploit
Database Query	SQL injection
Code Sandbox	Sandbox escape to remote code execution
Email	Phishing Campaigns
Web Use	DDoS through excessive agent use

TABLE I: Examples of theoretical tool use exploits.

Clearly both problems – safely parsing structured outputs, and enforcing safety policies over tool invocations – fall squarely within the purview of LangSec research.

III. Prevailing Attitudes

To understand emerging industry attitudes about AI deployment, we conducted a limited anonymous survey of five founders from AI startups backed by the venture capital firm and incubator Y Combinator.[5] The survey was conducted through Bookface, Y Combinator's private social network.[6] While the small sample size makes these results anecdotal, they provide some insight into how venture-backed startups approach AI integration. We enumerate results in Table II.

Practice or Belief	Prevalence
Uses AI-enabled IDE	100% (5/5)
Employs human code review	80% (4/5)
Uses AI code review tools	40% (2/5)
Uses AI testing tools	20% (1/5)
Allows autonomous production changes	0% (0/5)
Uses structured outputs/DSLs	80% (4/5)
Developed custom parser	20% (1/5)
Allows AI resource allocation	20% (1/5)
Believes AI limitations persist by 2030	60% (3/5)
Expresses security concerns	80% (4/5)

TABLE II: Survey Results from YC Startup Founders (n=5)

Our survey shows widespread adoption of AI development tools, with all respondents using AI-enabled IDEs and 20% deploying code without human review. While 80% utilize AI with structured outputs, most rely on established parsing tools rather than custom solutions. No companies allow autonomous production changes, and only 20% permit AI resource allocation decisions with human oversight. Security concerns were common (80% of respondents), and most founders (60%) believe AI will still have significant limitations by 2030, particularly around fully autonomous problem-solving.

While our survey respondents generally demonstrated caution in AI deployment, we observed more concerning practices in broader industry discussions. One startup founder (outside our survey) described using an automated system in which LLMs modify code in response to runtime exceptions, with changes merged automatically into production upon a successful test against whatever input caused the exception.

[4]For a nice talk on red-teaming LLMs the reader is referred to [38].

[5]Y Combinator has funded companies in excess of 600B USD in value, such as Stripe, AirBnB, Instacart, DoorDash, Reddit, Coinbase, and Scale AI.

[6]The first author is the CTO of a Y Combinator backed startup and thus has access to the platform.

Anecdotal conversations with a CEO of a major AI laboratory further suggest that some organizations are significantly reducing hiring based on the belief that AI agents can now fully automate most software engineering. These observations underscore the urgency of improving AI security, as certain well-funded organizations are already progressing toward fully autonomous engineering.

IV. Prior Works

The only prior work we are aware of which analyzed LLMs from a LangSec perspective is by Lintilhac, Ackerman, and Cybenko, who built TEAIS, a framework for evaluating the performance of LLMs, based on property-based testing and mutation fuzzing [39]. They did not study tool-use or structured outputs.[7] Outside of academia, the LangSec project [40] implements a policy-based safety system enabling secure AI interactions with SQL databases. (To the best of our knowledge, the authors are unfamiliar with LangSec as a field and the name is coincidental.) On the other hand, some prior works used LLMs *as* fuzzers to find language-theoretic vulnerabilities in other systems [41], [42]. This is a nascent research topic and, although LangSec adjacent, not typically conducted or written from an explicitly LangSec perspective.

V. Discussion

So far, we have outlined what software engineers are doing today with AI, why these engineering practices carry clear security risks, and why those security risks fall squarely within the research camp of LangSec. These risks are not just theoretical; they are immediate. For example, Karliner's Rules File Backdoor [43] uses hidden unicode characters in Model Context Protocol configuration files to trick AI systems into generating malicious code. This is a classic example of a language-theoretic vulnerability, where the human's interpretation of the language is different from the machine's interpretation (see [44]). Another example is the Prompt-to-SQL attack category [45] in which an attacker uses prompt injection techniques to peform SQL injection against an AI system with tool use over a database. In this case, the tool-use invocation simply executes the raw SQL output by the AI system, rather than forcing it to conform to some specific, safe, predefined sublanguage first.

Next, we outline some existing research directions the LangSec community could build upon to address said risks.

Type Contracts. Many structured output frameworks, such as OpenAI's JSON mode [20], do not actually guarantee that the produced outputs adhere to the specified structure. For example, suppose we want to generate a JSON object with a specific schema, say, `{"name": "string", "age": "int"}`. A typical approach is to prompt an LLM to return an object matching the desired schema, and nothing else, and then to `try/catch` a `JSON.parse` call around the returned string and coerce the result to the desired type. Depending on the semantics, an invalid string may lead to the `catch` branch, to an `undefined` object, or to an object matching the schema but with `undefined` values (e.g., `"name": undefined`). Strict type-checking naturally reduces the surface of both parser bugs and attacks which exploit those bugs, improving software reliability and security. BAML [24] addresses this need with a strict type system built in Rust, and they are working on a system for dependent types. Approaching the same problem from the opposite direction, Willard and Louf pioneered a technique whereby the LLM drives a pushdown automata to produce words within its language [46]. These ideas could be extended; the type contracts for structured outputs could include arbitrary predicates (such as the `:output-contracts` in ACL2 [47]), and the idea of using an LLM in conjunction with a formal language could be extended up the Chomsky hierarchy.

Formal Verification and Self-Certification. An emerging body of research focuses on using LLMs to steer interactive theorem provers in order to automatically prove mathematical theorems (see e.g. [48], [49], [50], [51]). Possibly AI systems could be similarly utilized to "prove" their own safety/correctness. This would reunite LLM-based code-generation with classical program synthesis, where the task is only considered complete when the generated code is proven correct. Recently, Mehta and Dougherty built a benchmark to test the capability of AI systems to prove the correctness of generated code [52]. However, at the time of writing, whether AI systems can consistently prove nontrivial theorems about software remains an open question.

Sandboxing. If policy enforcement is too hard, another option is to isolate the tool-use in a sandbox. For example, a SQL Database tool could be isolated to a sandbox containing an ephemeral copy of the database being queried; or code execution could be isolated in an offline AWS Lambda Function.

VI. Conclusion

Large language models have a myriad of faults. They are nondeterministic, produce unexpected outputs, can be slow and expensive, and are typically hosted on platforms that experience fluctuating demand and, as a result, fluctuating network reliability. But the biggest problem with LLMs is how unreasonably effective they are at a wide array of problems for which other computational approaches utterly fail. Because LLM-based AI systems perform so well across a wide swath of problem types – from math Olympiad to competitive programming to customer service – developers are rapidly integrating them into pre-existing software systems. This integration outpaces the requisite, corresponding security work. In this paper, we argued that securing AI systems is a uniquely LangSec problem, particularly because of structured outputs and tool-use. We gave anecdotes from the "front lines" of venture-backed software development to motivate why this problem matters, and outlined three concrete research areas the LangSec community could investigate to improve the situation. We believe the next RowHammer-caliber vulnerability will be in an AI system; and we hope the LangSec community can prove us wrong.

[7] We believe that fuzzing LLMs to violate the constraints of their structured output type contracts is a low-hanging fruit for follow-on research.

Acknowledgment

We would like to thank Juan Castaño for useful feedback and Jacob Denbeaux for insightful conversations leading up to this manuscript.

References

[1] Arthur L Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3):210–229, 1959.

[2] Introducing ChatGPT. https://openai.com/index/chatgpt/, 2022. Accessed 2/10/25.

[3] Jaime Sevilla and Edu Roldán. Training compute of frontier ai models grows by 4–5x per year. *Epoch AI, May*, 28, 2024.

[4] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[5] Openai o3 and o3-mini—12 days of openai: Day 12. https://www.youtube.com/watch?v=SKBG1sqdyIU, 12 2024. Accessed 2/10/25.

[6] Yuri Chervonyi, Trieu H Trinh, Miroslav Olšák, Xiaomeng Yang, Hoang Nguyen, Marcelo Menegali, Junehyuk Jung, Vikas Verma, Quoc V Le, and Thang Luong. Gold-medalist performance in solving olympiad geometry with alphageometry2. *arXiv preprint arXiv:2502.03544*, 2025.

[7] Guan Wang, Sijie Cheng, Xianyuan Zhan, Xiangang Li, Sen Song, and Yang Liu. Openchat: Advancing open-source language models with mixed-quality data. *arXiv preprint arXiv:2309.11235*, 2023.

[8] AQ Jiang, A Sablayrolles, A Mensch, C Bamford, DS Chaplot, D de las Casas, F Bressand, G Lengyel, G Lample, L Saulnier, et al. Mistral 7b (2023). *arXiv preprint arXiv:2310.06825*, 2023.

[9] Copilot. https://copilot.microsoft.com/, 2025. Accessed 2/10/25.

[10] Cursor. https://www.cursor.com/, 2025. Accessed 2/10/25.

[11] Continue dev. https://www.continue.dev/, 2025. Accessed 2/10/25.

[12] Andrej Karpathy. Tweet about "vibes coding". https://x.com/karpathy/status/1886192184808149383, 2 2025. Posted 12:17 AM Feb 3, 2025. Accessed Feb 10, 2025.

[13] Joyce Friedman. Alonzo church. application of recursive arithmetic to the problem of circuit synthesissummaries of talks presented at the summer institute for symbolic logic cornell university, 1957, 2nd edn., communications research division, institute for defense analyses, princeton, nj, 1960, pp. 3–50. 3a-45a. *The Journal of Symbolic Logic*, 28(4):289–290, 1963.

[14] Shihan Dou, Haoxiang Jia, Shenxi Wu, Huiyuan Zheng, Weikang Zhou, Muling Wu, Mingxu Chai, Jessica Fan, Caishuang Huang, Yunbo Tao, et al. What's wrong with your code generated by large language models? an extensive study. *arXiv preprint arXiv:2407.06153*, 2024.

[15] ATLAS matrix. https://atlas.mitre.org/matrices/ATLAS, 2024. Accessed 2/10/25.

[16] Ryan Greenblatt, Carson Denison, Benjamin Wright, Fabien Roger, Monte MacDiarmid, Sam Marks, Johannes Treutlein, Tim Belonax, Jack Chen, David Duvenaud, et al. Alignment faking in large language models. *arXiv preprint arXiv:2412.14093*, 2024.

[17] Adly Templeton, Tom Conerly, Jonathan Marcus, Jack Lindsey, Trenton Bricken, Brian Chen, Adam Pearce, Craig Citro, Emmanuel Ameisen, Andy Jones, Hoagy Cunningham, Nicholas L Turner, Callum McDougall, Monte MacDiarmid, C. Daniel Freeman, Theodore R. Sumers, Edward Rees, Joshua Batson, Adam Jermyn, Shan Carter, Chris Olah, and Tom Henighan. Scaling monosemanticity: Extracting interpretable features from claude 3 sonnet. *Transformer Circuits Thread*, 2024.

[18] Michael Xieyang Liu, Frederick Liu, Alexander J Fiannaca, Terry Koo, Lucas Dixon, Michael Terry, and Carrie J Cai. "we need structured output": Towards user-centered constraints on large language model output. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, pages 1–9, 2024.

[19] Yu Liu, Duantengchuan Li, Kaili Wang, Zhuoran Xiong, Fobo Shi, Jian Wang, Bing Li, and Bo Hang. Are llms good at structured outputs? a benchmark for evaluating structured output capabilities in llms. *Information Processing & Management*, 61(5):103809, 2024.

[20] https://platform.openai.com/docs/guides/structured-outputs?api-mode=responses#json-mode, 2024. Accessed 2/10/25.

[21] Rahul Sengottuvelu. https://github.com/1rgs/jsonformer, 2023. Accessed 2/10/25.

[22] Ian Hoegen. https://docs.trymaitai.ai/sdk/structured_output, 2024. Accessed 2/10/25.

[23] https://python.langchain.com/docs/how_to/structured_output/, 2025. Accessed 2/10/25.

[24] https://docs.boundaryml.com/home/welcome, 2024. Accessed 2/10/25.

[25] structured-logprobs. https://arena-ai.github.io/structured-logprobs/, 2025. Accessed 2/10/25.

[26] Nguyen Dat. History of structured outputs for llms. https://memo.d.foundation/playground/01_literature/history-of-structured-output-for-llms/, 2024. Accessed 2/10/25; last updated 2025.

[27] Aaron Parisi, Yao Zhao, and Noah Fiedel. TALM: Tool augmented language models. *arXiv preprint arXiv:2205.12255*, 2022.

[28] Xuanhe Zhou, Guoliang Li, and Zhiyuan Liu. Llm as dba. *arXiv preprint arXiv:2308.05481*, 2023.

[29] Eduardo R Nascimento, Yenier T Izquierdo, Grettel M Garcıa, Gustavo MC Coelho, Lucas Feijó, Melissa Lemos, Luiz AP Paes Leme, and Marco A Casanova. My database user is a large language model. In *26th Int. Conf. on Enterprise Info. Sys*, 2024.

[30] Yunhu Ye, Binyuan Hui, Min Yang, Binhua Li, Fei Huang, and Yongbin Li. Large language models are versatile decomposers: Decompose evidence and questions for table-based reasoning. *arXiv preprint arXiv:2301.13808*, 2023.

[31] Yuchen Zhuang, Yue Yu, Kuan Wang, Haotian Sun, and Chao Zhang. Toolqa: A dataset for llm question answering with external tools. *Advances in Neural Information Processing Systems*, 36:50117–50143, 2023.

[32] Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*, 2022.

[33] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR, 2023.

[34] Shreyas Sundara Raman, Vanya Cohen, Ifrah Idrees, Eric Rosen, Raymond Mooney, Stefanie Tellex, and David Paulius. Cape: Corrective actions from precondition errors using large language models. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 14070–14077. IEEE, 2024.

[35] Tool use (function calling). 2025. Accessed 2/10/25.

[36] Erik Johannes Cornelius Bosman. Attacker-defined abstractions: Programming benign system functionality for malicious purposes. 2024.

[37] Fabio Pagani, Alex Matrosov, Yegor Vasilenko, Alex Ermolov, Sam Thomas, and Anton Ivanov. Logofail: Security implications of image parsing during system boot. https://i.blackhat.com/EU-23/Presentations/EU-23-Pagani-LogoFAIL-Security-Implications-of-Image_REV2.pdf, 2023. Accessed 4/18/25.

[38] Hacking genai with llm red teaming and beyond. https://www.youtube.com/watch?v=i2KJZ8J5kMA, 1 2025. Accessed 2/10/25.

[39] Paul Lintilhac, Joshua Ackerman, and George Cybenko. Testing and evaluating artificial intelligence applications. In *2024 IEEE Security and Privacy Workshops (SPW)*, pages 231–238. IEEE, 2024.

[40] Amit Schendel. LangSec: A security framework for text-to-SQL. https://github.com/langsec-ai/langsec.

[41] Joshua Ackerman and George Cybenko. Large language models for fuzzing parsers (registered report). In *Proceedings of the 2nd International Fuzzing Workshop*, pages 31–38, 2023.

[42] Yu Jiang, Jie Liang, Fuchen Ma, Yuanliang Chen, Chijin Zhou, Yuheng Shen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Shanshan Li, et al. When fuzzing meets llms: Challenges and opportunities. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 492–496, 2024.

[43] Ziv Karliner. New vulnerability in github copilot and cursor: How hackers can weaponize code agents. https://www.pillar.security/blog/new-vulnerability-in-github-copilot-and-cursor-how-hackers-can-weaponize-code-ager March 2025. Accessed 4/18/25.

[44] Falcon Momot, Sergey Bratus, Sven M Hallberg, and Meredith L Patterson. The seven turrets of babel: A taxonomy of langsec errors and how to expunge them. In *2016 IEEE Cybersecurity Development (SecDev)*, pages 45–52. IEEE, 2016.

[45] Rodrigo Pedro, Daniel Castro, Paulo Carreira, and Nuno Santos. From prompt injections to sql injection attacks: How protected is your llm-integrated web application? *arXiv preprint arXiv:2308.01990*, 2023.

[46] Brandon T Willard and Rémi Louf. Efficient guided generation for llms. *arXiv preprint arXiv:2307.09702*, 2023.

[47] https://www.cs.utexas.edu/~moore/acl2/v8-5/combined-manual/index.html?topic=ACL2S____DEFUNC. Accessed 2/10/25.

[48] Albert Q Jiang, Sean Welleck, Jin Peng Zhou, Wenda Li, Jiacheng Liu, Mateja Jamnik, Timothée Lacroix, Yuhuai Wu, and Guillaume Lample. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. *arXiv preprint arXiv:2210.12283*, 2022.

[49] Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J Prenger, and Animashree Anandkumar. Leandojo: Theorem proving with retrieval-augmented language models. *Advances in Neural Information Processing Systems*, 36, 2024.

[50] Minghai Lu, Benjamin Delaware, and Tianyi Zhang. Proof automation with large language models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1509–1520, 2024.

[51] Peiyang Song, Kaiyu Yang, and Anima Anandkumar. Towards large language models as copilots for theorem proving in lean. *arXiv preprint arXiv:2404.12534*, 2024.

[52] Q. Dougherty R. Mehta. Proving the coding interview: A benchmark for formally verified code generation. In *The Second International Workshop on Large Language Models for Code*, 2025.