TypePilot: Leveraging the Scala type system for secure LLM-generated code

Alexander Sternfeld
Institute of Entrepreneurship & Management, HES-SO
Le Foyer, Techno-Pôle 1
Sierre, Switzerland
alexander.sternfeld@hevs.ch

Andrei Kucharavy
Institute of Informatics, HES-SO
Techno-Pôle 3
Sierre, Switzerland
andrei.kucharavy@hevs.ch

Ljiljana Dolamic
Cyber-Defence Campus
armasuisse, Science and Technology
Thun, Switzerland
ljiljana.dolamic@armasuisse.ch

Abstract

Large language models (LLMs) have shown remarkable proficiency in code generation tasks across various programming languages. However, their outputs often contain subtle but critical vulnerabilities, posing significant risks when deployed in security-sensitive or mission-critical systems. This paper introduces *TypePilot*, an agentic AI framework designed to enhance the security and robustness of LLM-generated code by leveraging strongly typed and verifiable languages, using Scala as a representative example. We evaluate the effectiveness of our approach in two settings: formal verification with the Stainless framework and general-purpose secure code generation. Our experiments with leading open-source LLMs reveal that while direct code generation often fails to enforce safety constraints, just as naive prompting for more secure code, our type-focused agentic pipeline substantially mitigates input validation and injection vulnerabilities. The results demonstrate the potential of structured, type-guided LLM workflows to improve the SotA of the trustworthiness of automated code generation in high-assurance domains.

1 Introduction

In recent years, large language models (LLMs) have become powerful tools for assisting in software development, from generating boilerplate code to proposing non-trivial algorithmic implementations (Wang and Chen, 2023; Chen et al., 2021). Their fluency in natural and programming languages allows developers to interact with them without disrupting their workflow, accelerating the development lifecycle. However, as LLMs are increasingly used to write production code, concerns have emerged about the reliability and security of the generated output. Multiple studies and real-world analyses have shown that LLMs can introduce subtle yet serious vulnerabilities (Pearce et al., 2025).

This issue becomes particularly acute in the domain of mission-critical systems—software systems whose failure can lead to catastrophic outcomes, including physical harm, financial loss, major operational disruptions, or loss of life (Gabriel et al., 2022). Such systems are often implemented in strongly typed, safety-oriented programming languages like Coq, Scala, or more recently Rust, where the type system is a central mechanism for enforcing correctness and preventing classes of bugs at run time. Despite these safeguards, vulnerabilities still surface, often due to logical oversights, incorrect assumptions, or abstraction mismatches at boundaries. A well-known example is the 1999 NASA Mars Climate Orbiter failure, where one subsystem produced output in imperial units while another expected metric, leading to the spacecraft's loss due to an undetected discrepancy at the interface between components (Harish, 2025). A notable recent example of such a vulnerability in action occurred in January 2023, when a critical FAA system failure, later traced to a corrupted configuration file, led to the temporary grounding of all flights across the United States (reuters, 2023).

While LLMs are increasingly capable of detecting potential code vulnerabilities, they often fall short in generating robust corrections (Kulsum et al., 2024; Pearce et al., 2022). Our work addresses this gap. By focusing on Scala - a widely used language with extensive codebase on GitHub and documentation on StackOverflow (O'Grady, 2025), we propose TypePilot, an agentic AI approach that not only leverages the detection capabilities of LLMs but actively guides them to exploit the expressiveness of the Scala type system to add safety guarantees. By structuring interactions, TypePilot guides LLMs to generate and refine code that adheres to strict safety and correctness properties.

This paper is structured as follows: Section 2 describes the related literature, after which Section 3 outlines the methodology. Next, the results are presented in Section 4. Last, Section 6 concludes the paper and provides directions for future research. The code and results related to this paper are publicly available in this Github Repository.

2 Related work

2.1 LLMs for code generation

The use of large language models for code generation has grown rapidly, with coding specific models demonstrating impressive capabilities across a wide range of programming languages and tasks. However, several studies have pointed out that these models often produce code that is syntactically correct but semantically flawed or insecure. For instance, Pearce et al. (2025) shows that GitHub Copilot produces vulnerabilities in approximately 40% of test cases based on the top 25 Common Weakness Enumeration list from MITRE. Similarly, Khoury et al. (2023) show that ChatGPT generates vulnerable code in 16 out of 21 test cases, using a variety of programming languages targeting a diverse set of vulnerabilities.

There have been attempts to use separate LLMs in combination with sophisticated prompting strategies to patch such vulnerabilities. However, these approaches remain brittle, with models often misunderstanding the root cause or proposing fixes that break functionality. Kulsum et al. (2024) show that LLMs have difficulty in patching vulnerabilities that are either complex or linked to the project design. Similarly, Pearce et al. (2022) show that LLMs are not yet able to autonomously patch code vulnerabilities in real-world scenarios.

Our work builds upon these findings by exploring a different method for mitigating vulnerabilities - leverage the properties of strongly typed coding languages. We use *agentic AI*, where LLMs cooperatively operate as autonomous agents, which has been shown to result in better generations (Kumar et al., 2025; Wang et al., 2025).

3 Methodology

We will now describe the methodology that is used in this research. First, the models that are used in this research are specified. Then, we consider the ability of LLMs to generate code using the formal verification framework Stainless. Last, we consider the general case of type-system rooted vulnerabilities.

3.1 Model usage

Throughout this research, we use open-source models, with a focus on specialized coding models. Specifically, we used the coding models `Qwen/Qwen2.5-Coder-32B-Instruct`, `deepseek-ai/deepseek-coder-33b-instruct` and `codellama/CodeLlama-70b-Instruct-hf`. Additionally, we used the regular conversational models `meta-llama/Meta-Llama-3-70B`, `deepseek-ai/DeepSeek-R1-Distill-Llama-70B` and `Qwen/Qwen3-32B`.

3.2 Stainless

We first aim to leverage the formal verification framework Stainless (Lab for Automated Reasoning and Analysis, 2025) to improve the robustness of LLM-generated code. Formal verification refers to the use of mathematical methods to prove that a program satisfies certain correctness properties. Stainless is one of the most widely used verification frameworks in Scala, with extensive documentation. Stainless verifies whether Scala code meets user-specified safety properties by attempting to construct proofs over the code. To enable this, the code must explicitly state what is to be proven, and provide the necessary logical structure for the proof, using a subset of Scala tailored for verification.

To this end, we use both zero-shot and two-shot prompting to have a LLM both generate the code and the conditions. Figure 2 displays the prompt that is used in the two-shot prompting setting. The two examples that are given to the LLM are Stainless code for finding the maximum between two values and for returning the size of a list. The exact examples can be found in the Github Repository.

As displayed in Table 1, we use three simple tasks for evaluating the LLMs in the context of formal proofs: calculating Fibonacci number n, calculating the factorial of an input and assessing whether list *a* is a sublist of list *b*. The main vulnerability that the generated conditions should prevent

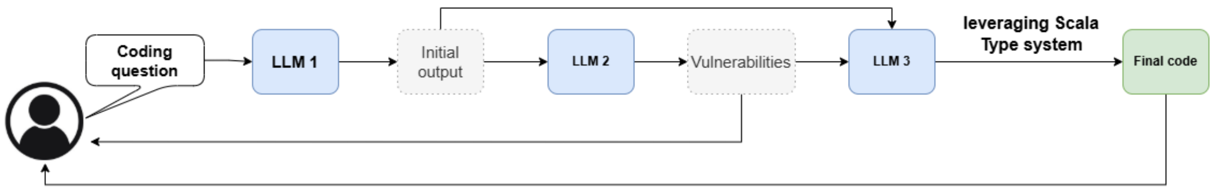

Figure 1: The full pipeline for the generation of code using TypePilot. After the initial generation of the code, the vulnerabilities are detected by a separate instance of the LLM. Then, a final LLM is prompted to leverage the Scala type system to improve the initial code, given the detected vulnerabilities.

are input variables that are invalid, such as a negative input for a factorial function. Additionally, the functions should also be robust to inputs that are too large and may cause an overflow error.

Generation of stainless code

<question> Use the stainless framework to write verifiable scala code for fewshot example 1 </question>

<answer> fewshot answer 1 </answer>

<question> Use the stainless framework to write verifiable scala code for fewshot example 2 </question>

<answer> fewshot answer 2 </answer>

<question> Use the stainless framework to write verifiable scala code for function description </question>

Figure 2: Prompt used to generate the Stainless code.

3.3 General case: type-system rooted vulnerabilities

As Stainless targets a niche subset of Scala applications, we also consider a more general setting. Specifically, we focus on two vulnerability categories: insufficient input constraints and injection

Code generation in *robust* setting

You are a scala code generator. You will be given a task description and you will generate the code for it. The code should start with "‘scala and end with "‘. Pay attention to the safety and robustness of the code, and leverage the Scala type system - for example ADTs, refined types, traits, sealed traits - where needed to make the code safer. The task is: user input

Figure 3: Prompt used to generate the code in the robust prompting setting.

attacks. In particular, we examine HTML, Bash, and URL injections—common security risks in back-end web development, especially when handling user inputs through web forms. The specific test cases are shown in Table 1. To assess the performance of LLMs on these tasks, we consider the following settings:

- **Baseline:** directly prompting a LLM to generate the code
- **Robust prompting:** directly prompting a LLM to generate the code, while emphasizing that the LLM should leverage the Scala type system to make the code robust to potential vulnerabilities.

Stainless	**General case: type-system rooted vulnerabilities**	
	Input constraints	*Code injection*
Calculating a fibonacci number	Calculating a fibonacci number	Greeting a user with HTML
Calculating the factorial of a number	Calculating the factorial of a number	Making a list of comments with HTML
Asserting if list a is a sublist of list b	Calculating a matrix multiplication	Searching a file using bash
	Calculating a matrix convolution	Pinging a host using bash
		Creating a redirect URL with HTML

Table 1: The test cases used to evaluate the LLMs in each of the settings. The most left column shows the test cases used to evaluate the performance of LLMs in generating code using the Stainless framework. The second and third column show the test cases for the general case looking at type-system rooted vulnerabilities.

Code generation using TypePilot

Initial code generation
You are a Scala code generator. You will be given a task description and you will generate the code for it. The code should start with "'scala and end with "'. The task is: user input

Vulnerability detection
You will be given a task description and generated code. Your task is to find potential vulnerabilities in the code that could lead to security issues or unexpected behavior. Solely describe the vulnerabilities, do not give me any code. Here is the task: user input Here is the previous code: initial output

Final code generation
You are a Scala code generator. You will be given a task description, generated code, and vulnerabilities that should be addressed. Your task is to improve the code by using the Scala type system - for example ADTs, refined types, traits, sealed traits - to address the vulnerabilities. The code should start with "'scala and end with "'. Here is the task: user input. Here is the previous code: initial output Here are the vulnerabilities: vulnerabilities

Figure 4: Prompts used to generate the initial code, the vulnerabilities and the final code in TypePilot. The final prompt guides the LLM to use the Scala type system to make the code more robust.

- **TypePilot:** use the agentic AI framework as displayed in Figure 1 to generate the code. After prompting a first LLM to generate the initial code, we ask a second LLM to detect the vulnerabilities in this code. We then ask a third instance of the LLM to improve the initial code using the Scala type system, to make it robust to the detected vulnerabilities.

The prompt that is used in the robust generation setting can be found in Figure 3. Similarly, Figure 4 shows the prompts that are used with TypePilot. In the baseline setting, we use the same prompt that is used for the initial code generation in TypePilot. For each of the models described in Section 3.1 we run each of the settings.

3.4 Comparison to existing work

Research on secure code generation using large language models (LLMs) remains limited, despite growing concerns about vulnerabilities in automatically generated code. A recent survey by Dai et al. (2025) highlights that most current approaches rely heavily on training data or static analysis tools, restricting their generalizability. Methods such as SafeCoder (He et al., 2024) and SVEN (He and Vechev, 2023) fine-tune LLMs with curated secure code datasets, and are thus inherently dependent on the availability and quality of specialized training corpora. Moreover, the fine-tuned LLMs do not generalize well to unseen vulnerabilities or programming languages. Similarly, PromSec (Nazzal et al., 2024) optimizes prompts through static

	Qwen-2.5-Coder (32B)			CodeLlama (70B)			Deepseek-coder (33B)		
	Baseline	Robust	TypePilot	Baseline	Robust	TypePilot	Baseline	Robust	TypePilot
Average age									
- Correct for regular input	✓	✓	✓	✓	✓	✓	✓	✓	✓
- Handle empty lists	✓	✓	✓	✓	✓	✓	✓	✓	✓
- Handle negative ages	✗	✗	✓	✗	✗	✓	✗	✗	✗
Fibonacci number N									
- Correct for regular input	✓	✓	✓	✓	✓	✓	✓	✓	✓
- Check for negative N	✗	✓	✓	✗	✗	✗	✗	✗	✓
- Handles large values of N	✗	✓	✓	✗	✗	✗	✗	✓	✓
Matrix multiplication									
- Correct for regular input	✓	✓	✓	✓	✓	✓	✓	✓	✓
- Check for empty matrices	✓	✗	✓	✗	✗	✗	✗	✗	✗
- Check for dimension matching	✓	✓	✓	✓	✗	✓	✗	✓	✓
Matrix convolution									
- Correct for square matrix input	✓	✓	✓	✗	✗	✗	✓	✓	✓
- Correct for regular matrix input	✓	✓	✓	✗	✗	✗	✓	✓	✓
- Handles rectangular kernels	✗	✗	✓	✗	✗	✓	✗	✗	✗
- Checks for empty kernel	✗	✓	✓	✗	✗	✓	✗	✗	✓
- Checks for empty matrix	✗	✓	✓	✗	✗	✓	✗	✗	✓
- Handles even sized kernels	✗	✗	✓	✗	✗	✗	✗	✗	✗

Table 2: Manual evaluation of the generated code regarding input constraints. For each case, ✓ indicates that the code is robust to the vulnerability, whereas ✗ indicates that the code is not robust to the vulnerability.

	Qwen-2.5-Coder (32B)			CodeLlama (70B)			Deepseek-coder (33B)		
	Baseline	Robust	TypePilot	Baseline	Robust	TypePilot	Baseline	Robust	TypePilot
HTML greeting									
Correctness and compilation	✓	✓	✓	✓	✓	✓	✓	✓	✗
Robust to injection	✗	∼	✓	✗	✓	✓	✗	✓	✓
HTML comments									
Correctness and compilation	✓	∼	✓	✓	✗	∼	✓	✓	✓
Robust to injection	✗	✓	✓	✗	✗	✓	✗	∼	✓
Bash file search									
Correctness and compilation	✓	✓	✓	✓	✓	✓	✓	✓	✓
Robust to injection	✗	✗	✓	✗	✗	✓	✗	✗	✓
Bash host ping									
Correctness and compilation	✓	✓	✓	✓	✗	✓	✓	✓	✓
Robust to injection	✓	✓	✓	✗	✗	✓	✗	✓	✓
URL redirect									
Correctness and compilation	✓	✓	✓	✓	✓	✓	✓	✓	✓
Robust to injection	✗	∼	∼	✗	✓	✓	✗	✗	✓

Table 3: Manual evaluation of the generated code regarding code injection. For each case, ✓ indicates that the code is robust to the vulnerability, whereas ✗ indicates that the code is not robust to the vulnerability.

analyzers, but also relies on labeled data and an external code-specific vulnerability scanner.

In contrast, our approach does not rely on task-specific training data or external static analyzers. Instead, it leverages the expressive power of strongly typed languages to enforce security constraints directly in the generated code. Because most existing methods depend on curated datasets or vulnerability scanners (as discussed above), there are few established baselines tailored to strongly typed languages like Scala or Rust. Given this gap, it is most appropriate to compare our method against prompting-based baselines in addition to the base model. Following Vero et al. (2025), we include a baseline where the model is given a general security reminder, which we call *robust prompting*. We also evaluate against *Self-Planning*, a coding-specific prompting strategy introduced by Jiang et al. (2024). Self-Planning is a two-stage prompting framework in which the LLM first generates a high-level plan for the coding task, after which it implements the plan in code.

4 Results

4.1 Stainless

In general, we see that none of the models is capable of consistently generating Stainless code that correctly compiles. Upon manual inspection, we found two main failure modes across all models. First, each of the LLMs regularly uses concepts that are present in Scala but not available in Stainless. As Stainless is a verification framework targeting a restricted subset of Scala, many features of full Scala—such as certain standard library functions—are unavailable. To illustrate, in the generated code from `Qwen/Qwen3-32B` for the verification of a sublist relation, the function `List.sliding` is used. However, the sliding operation is not defined for Stainless `List` objects. Similarly, in a generated code snippet the operation `println` was used, which is not available in Stainless. Second, the generated code often contains syntax errors. Whereas syntax errors could be resolved relatively easily by users, the usage of Scala components in Stainless is not trivially repaired. We hypothesize that the lack of performance is caused by a lack of training data related to Stainless, given that it is a niche framework. This observation is consistent with findings from other domains, for example, Fan et al. (2025) found that LLMs struggle to generate verifiable specifications using the VeriFast verification framework for C, despite preserving functional behavior. In appendix B, we provide a notable instance in which the generation avoids formal verification by using `@library` annotations.

4.2 General setting

Given that LLMs are not able to write compilable Stainless code, we shift our attention to a more general Scala setting, as described in Section 3.3. We consider two types of vulnerabilities: insufficient input constraints and code injection. The generated code is available in the anonymized repository.

4.2.1 Input constraints

Table 2 shows the results for each of the test cases for each of the models. For each of the models, ✗ indicates that the resulting code was not robust to the indicated vulnerability. The results show that in the baseline setting, the models are capable of generating functions that provide the correct output in a normal setting. However, the models are not capable of handling edge cases correctly. To illustrate, none of the models can correctly handle negative ages or a negative input to a Fibonacci function. We see that in the *robust* setting, models perform slightly better, and tend to be robust to some of the vulnerabilities. However, for none of the models the code is fully robust. With TypePilot, we obtain the best performance, with models generally being robust to most vulnerabilities related to input constraints.

When comparing the models, we observe that `Qwen-2.5-Coder (32B)` performs the best, passing all our checks when using TypePilot. In contrast, `CodeLlama (70B)` does not perform well, remaining vulnerable to a number of cases in each of the settings, highlighting the importance of study of specific code-generating LLM models.

4.2.2 Code injection

The second type of vulnerability we consider is code injection. Table 3 displays the results for each of the models, where ✗ indicates that the code is vulnerable to injection, ✓ indicates that code is robust to injection, and ∼ indicates that the code is partially robust to injection. The results show that in the baseline setting virtually all generated code is vulnerable to code injection. Robust prompting improves the performance, resulting in fewer vulnerabilities. As before, TypePilot achieves the best performance, with robust code generations in almost all settings.

4.2.3 Usage of the Scala type system

In the new framework, the Scala type system is used as a central tool to guide the generation of secure code. By prompting LLMs to leverage features such as sealed traits, smart constructors, and refined return types, we enable the generation of programs that encode correctness directly into their type signatures. This stands in contrast to baseline generations, which operate on unconstrained primitives and rely on ad hoc runtime logic to handle edge cases and errors.

Figures 5 shows an examples of code generated in the baseline and in the agentic AI framework, for the same test case and model. Figure 5 shows that in the baseline version, the `averageAge` function takes a `List[Int]` and performs a division after checking for emptiness. While this implementation is syntactically valid, it permits semantically invalid inputs—such as negative ages or values far outside a realistic human range—and silently defaults to returning 0.0 when the input list is empty. In contrast, the enhanced version defines a sealed trait `Age` and a case class `ValidAge`, with a smart constructor in the `Age` companion object that en-

(a) Baseline

```
object GeneratedFunctions {
    def averageAge(ages: List[Int]): Double = {
        if (ages.isEmpty) 0.0
        else ages.sum.toDouble / ages.length
    }
}
```

(b) TypePilot

```
object GeneratedFunctions {
  sealed trait Age {
    def value: Int
  }

  final case class ValidAge(value: Int) extends Age

  object Age {
    def apply(value: Int): Option[Age] = {
      if (value >= 0 && value <= 120) Some(ValidAge(value))
      else None
    }
  }

  def averageAge(ages: List[Age]): Option[Double] = {
    if (ages.isEmpty) None
    else {
        val (sum, count) = ages.foldLeft((0L, 0)) {
        case ((accSum, accCount), age) =>
        (accSum + age.value, accCount + 1)
    }
  }
}
```

Figure 5: Comparison of baseline and TypePilot average age function generations from `Qwen-2.5-Coder (32B)`

forces domain-specific constraints: only values between 0 and 120 are permitted. The `averageAge` function now accepts a list of validated `Age` values and returns an `Option[Double]`, making both the domain invariants and the possibility of undefined results (e.g., empty lists) explicit at the type level. This design ensures that all inputs have been prevalidated before the function executes, reducing the likelihood of subtle logic bugs and enabling safer composition in larger systems. A second example related to generating a function to search for files using bash is discussed in appendix C.

In TypePilot, the Scala type system is used not merely to enforce syntactic correctness but to encode domain abstractions rules, constrain behavior, and make failure modes explicit. By doing so, it transforms what would otherwise be runtime checks and ad hoc validations into statically enforced contracts. This shift leads to code that is more robust, more predictable, and better aligned with the principles of secure and maintainable software design. In the context of LLM-generated code, these benefits are particularly important, as they offer a principled way to guard against common pitfalls and encourage safer defaults during generation.

4.3 Vulnerability Analysis

We performed a post-hoc vulnerability analysis by categorizing the vulnerabilities observed in each test case. These categories include input constraint issues (shape violations, null dereferences and boundary violations) and code injection risks (HTML injection, bash injection and path traversal). For each method, we calculated the fraction of secure outputs and averaged the results across the three LLMs, which is displayed in Figure 6.

For input constraints, robust prompting offered limited improvements over the baseline, particularly for shape violations and null dereferences. It often inserted assertions but did not systematically enforce data structure correctness. TypePilot reduced these errors more effectively, as the presence of type specifications led the models to generate code structured around expected data formats rather than relying on runtime checks.

For code injection, TypePilot also lowered vulnerability rates, especially for bash injections where robust prompting typically altered command structure without validating input. Results varied between models: Qwen-2.5-Coder (32B) and Deepseek-Coder (33B) generally applied the type system consistently, while CodeLlama (70B) sometimes attempted to handle vulnerabilities outside the type framework. In some cases, type constraints were only partially used, such as defining a type for an output value but not for the input values.

Appendix D analyzes attention weights across the three methods, showing that TypePilot places greater emphasis on key safety terms during code generation than robust prompting.

4.4 Comparison to Self-Planning Code Generation

As an additional validation, we compared TypePilot to the Self-Planning prompting framework,

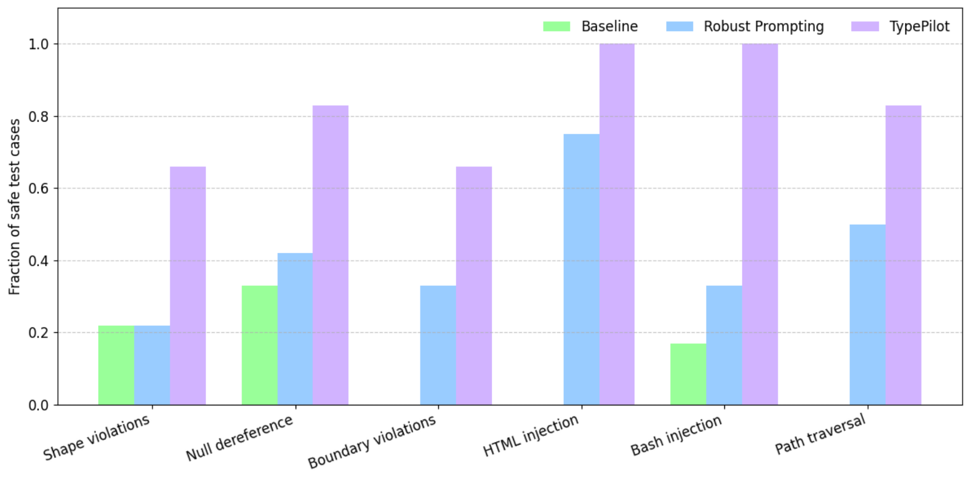

Figure 6: Fraction of secure code generations across vulnerability categories for each of the methods (baseline prompting, robust prompting, TypePilot). Results are averaged over all evaluated LLMs. Lower bars indicate a higher frequency of vulnerabilities; higher bars indicate safer generations.

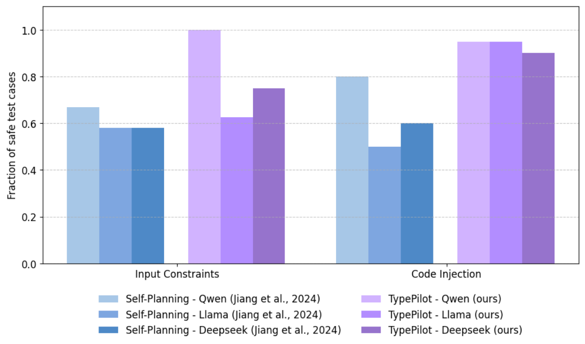

Figure 7: Comparison of the secure code generation methods TypePilot (ours) and Self-Planning, as introduced by Jiang et al. (2024).

as discussed in Section 3.4. In the Self-Planning framework, the model is first asked to outline a plan for solving the task. Afterwards, it is asked to write the code by executing the plan, and it is explicitly instructed to consider safety and security aspects before writing code. Overall, TypePilot outperforms self-planning for both the input constraint and code injection tasks. The difference is largest for Qwen-2.5-Coder (32B), which more reliably adheres to the type system instructions in TypePilot, resulting in fewer shape and null-handling issues compared to the Self-Planning setup.

Manual inspection of the Self-Planning outputs reveals that, despite explicit prompts to account for vulnerabilities during the planning and implementation stages, models frequently overlook or underaddress these concerns. The generated plans may mention security considerations in abstract terms but rarely translate them into concrete, protective measures in the final code. These findings suggest that simply instructing the model to "think about safety" is insufficient: introducing a structured intermediate step, such as TypePilot's type-enforced specification phase, is more effective in steering the model toward safer code generation.

5 Scaling

The primary goal of this work is to show that leveraging the type system in strongly typed languages can substantially mitigate vulnerabilities in LLM-generated code. While our experiments focus on relatively simple test cases, practical applications often involve larger, interconnected codebases with complex object hierarchies. Scaling our framework to such scenarios presents new challenges, primarily related to context management and dependency reasoning across multiple files and modules.

One promising direction is the development of a hybrid, object-aware prompting system. In this approach, metadata about each relevant object, including its types and invariants, is provided to the LLM prior to generation. This structured context could enable the model to reason more accurately about type interactions and enforce security constraints across function boundaries. Additionally, integrating lightweight symbolic reasoning or type inference engines could help LLMs maintain global consistency in larger projects, further reducing the risk of injection attacks and logical errors.

6 Conclusion

In this work, we aim to improve the security of LLM generated mission-critical code, focusing on the Scala strongly typed language. As Scala is routinely used in mission-critical software and engineers are increasingly often using LLMs to code, it is essential to ensure that the generated code is free of vulnerabilities. We first show that LLMs are not able to autonomously use the static verification tool Stainless. Therefore, we develop a more general agentic AI framework that structures multi-step interactions between LLMs for code generation. By leveraging the Scala type system, we significantly improve the quality and safety of generated code. Crucially, this approach transforms type systems from passive compile-time enforcers into active agents of code safety. We study two different classes of vulnerabilities, input constraints and code injection, and show that in both cases our framework improves code safety over a baseline and zero-shot robust prompting setting. We use the rigidity of the Scala type system to compensate for the inconsistencies picked up from the training code by LLMs, which in turn allow an easier interface to access the power of the Scala type system.

We conclude by suggesting two directions for future research. First, future work should test the framework's capabilities in more complex codebases. While this study provided a proof of concept using simple test cases, real-world software tends to be more complex, so validating our approach in these environments is important to assess its effectiveness. Second, deploying the framework in an active development setting would allow engineers to use it in their daily work and provide valuable feedback. This real-world input can guide further improvements and help tailor the framework to better meet the needs of software teams.

References

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code.

Shih-Chieh Dai, Jun Xu, and Guanhong Tao. 2025. A comprehensive study of llm secure code generation.

Wen Fan, Marilyn Rego, Xin Hu, Sanya Dod, Zhaorui Ni, Danning Xie, Jenna DiVincenzo, and Lin Tan. 2025. Evaluating the ability of large language models to generate verifiable specifications in verifast.

Ellie Gabriel, Xenophon Papademetris, Ayesha N. Quraishi, and Gregory P. Licholai. 2022. *Therac-25: Software that Killed*, page 263–267. Cambridge University Press.

Ajay Harish. 2025. When nasa lost a spacecraft due to a metric math mistake. Accessed: July 8, 2025.

Jingxuan He and Martin Vechev. 2023. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 1865–1879. ACM.

Jingxuan He, Mark Vero, Gabriela Krasnopolska, and Martin Vechev. 2024. Instruction tuning for secure code generation.

Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2024. Self-planning code generation with large language models. *ACM Trans. Softw. Eng. Methodol.*, 33(7).

Raphaël Khoury, Anderson R. Avila, Jacob Brunelle, and Baba Mamadou Camara. 2023. How secure is code generated by chatgpt? In *2023 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 2445–2451.

Ummay Kulsum, Haotian Zhu, Bowen Xu, and Marcelo d'Amorim. 2024. A case study of llm for automated vulnerability repair: Assessing impact of reasoning and patch validation feedback. In *Proceedings of the 1st ACM International Conference on AI-Powered Software*, AIware 2024, page 103–111, New York, NY, USA. Association for Computing Machinery.

Mayank Kumar, Jiaqi Xue, Mengxin Zheng, and Qian Lou. 2025. Tfhe-coder: Evaluating llm-agentic fully homomorphic encryption code generation.

Lab for Automated Reasoning and Analysis. 2025. Stainless: A verification framework for scala programs. `https://epfl-lara.github.io/stainless/index.html`.

Mahmoud Nazzal, Issa Khalil, Abdallah Khreishah, and NhatHai Phan. 2024. Promsec: Prompt optimization for secure generation of functional source code with large language models (llms). In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, page 2266–2280. ACM.

Stephen O'Grady. 2025. The redmonk programming language rankings: January 2025. Accessed: July 8, 2025.

Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2025. Asleep at the keyboard? assessing the security of github copilot's code contributions. *Commun. ACM*, 68(2):96–105.

Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2022. Examining zero-shot vulnerability repair with large language models.

reuters. 2023. Explainer: Why u.s. flights were grounded by a faa system outage. Accessed: July 8, 2025.

Mark Vero, Niels Mündler, Victor Chibotaru, Veselin Raychev, Maximilian Baader, Nikola Jovanović, Jingxuan He, and Martin Vechev. 2025. Baxbench: Can llms generate correct and secure backends?

Haoran Wang, Zhenyu Hou, Yao Wei, Jie Tang, and Yuxiao Dong. 2025. Swe-dev: Building software engineering agents with training and inference scaling.

Jianxun Wang and Yixiang Chen. 2023. A review on code generation with llms: Application and evaluation. In *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*, pages 284–289.