The Language of Security: How Prompt Syntax Shapes Secure Code Generation in Open LLMs

Matteo Cicalese*, Antonio Della Porta*, Stefano Lambiase†,
Emanuele Iannone‡, Torge Hinrichs‡, Riccardo Scandariato‡, Fabio Palomba*
*Software Engineering (SeSa) Lab, University of Salerno, Italy
†Human Augmentation and Collaboration (HAC) Group, Aalborg University, Denmark
‡Hamburg University of Technology, Institute of Software Security, Germany

***Abstract*—Large Language Models (LLMs) are increasingly used for source code generation despite their outputs often exhibiting security vulnerabilities. Prior work shows that prompt engineering can mitigate such risks, yet (1) they focused on high-level prompting strategies, neglecting recent evidence that fine-grained syntactic variations can substantially alter model behavior; and (2) predominantly evaluate proprietary LLMs, limiting the applicability of their findings in industrial settings where self-hosted, open models are preferred for privacy, compliance, and deployment control. In this paper, we study how *fine-grained syntactic constituents* of prompts influence the security of open LLM-generated code. Using a parser-driven approach, we systematically generate syntactic variants of security-relevant code generation prompts and evaluate their impact on code security across multiple open LLMs and programming languages. Our results show that specific syntactic elements, such as constraints, guards, conditions, and concept bindings, and their position within the prompt consistently affect the likelihood of generating insecure code. These findings identify prompt syntax as a concrete security control surface and provide actionable guidance for reducing vulnerability risk in LLM-assisted development.**

***Index Terms*—Secure Code Generation; Large Language Models for Software Engineering; Empirical Software Engineering.**

I. Introduction

Large Language Models (LLMs) are increasingly integrated into modern software development, enabling code generation from natural-language descriptions through tools such as GitHub Copilot and ChatGPT [1]. While primarily optimized for functional correctness, developer productivity [2], non-functional requirements [3]–[6] empirical evidence shows that LLM-generated code often overlooks security concerns and is frequently *vulnerable by default* [7]–[9]. Large-scale repository analyses report thousands of CWE-mapped vulnerabilities in AI-generated code [8], controlled experiments show that iterative AI-driven refinement can amplify critical flaws without human oversight [9], and comparative studies find higher rates of high-risk vulnerabilities than in human-written code [7], [10]. At the developer level, AI assistance has been shown to increase the likelihood of introducing security vulnerabilities [11], and nearly 40% of GitHub Copilot–generated code contains security weaknesses [12], and technical debt [13]–[16]. At scale, even modest systematic weaknesses can significantly expand the software attack surface.

A promising line of defense against these risks is *prompt engineering*, which steers model behavior through carefully crafted natural-language instructions [17]–[19]. Recent work in secure code generation has demonstrated that prompting strategies matter: explicitly embedding security requirements in prompts [18], enriching task descriptions with security constraints and operational conditions [19], secure coding guidelines [20], or iteratively asking models to review and refine their outputs [17] can measurably reduce the incidence of insecure code. As such, these studies have provided essential evidence that *'how we ask'* an LLM to generate code can be as important as *'which model we use'*.

Despite recent notable advances in the field, existing research has predominantly examined prompts at a *coarse level of abstraction*, focusing on high-level techniques (e.g., prompt patterns and iterative prompting), overall structure, or the presence of explicit security instructions. This leaves open a fundamental and practically relevant question: *What happens below the level of prompting strategies?* In other words, beyond what is requested, does *how* the request is linguistically constructed matter for security outcomes?

These questions are motivated by prior work showing that small linguistic variations, such as syntactic rearrangements, paraphrases, or clause removal, can substantially alter LLM behavior even when semantic intent is preserved [21]–[24]. In code generation, such perturbations significantly degrade correctness and robustness [21], [22], and similar syntactic sensitivities have been observed in question answering and reasoning tasks [24], [25]. As such, these findings indicate that prompts are not interpreted as abstract semantic specifications, but as structured linguistic artifacts whose internal composition directly shapes the LLM's reasoning process. Our working hypothesis builds directly on this evidence:

> **Working Hypothesis.** If small linguistic and syntactic variations can alter correctness and robustness, then specific syntactic constituents of prompts may also systematically influence the security of the code generated by LLMs.

From this perspective, studying prompt language at a finer granularity is a necessary step to understand why certain prompting practices succeed or fail. High-level strategies implicitly rely on low-level linguistic mechanisms, such as clauses that encode constraints, modifiers that qualify actions, or phrases that bind conditions. As a consequence, without

isolating these constituents, it is unclear which elements truly act as security-relevant control surfaces and which are incidental. This gap is particularly consequential given the growing adoption of *open-source LLMs*. While much existing work on secure code generation evaluates proprietary models [2], open-weight LLMs are increasingly deployed in industrial and organizational contexts where privacy, deployment control, cost, and reproducibility are critical [26], [27]. In compliance- or confidentiality-sensitive settings, organizations often favor self-hosted models over cloud-based alternatives, limiting the transferability of findings derived solely from closed systems [26]. Yet, despite their practical relevance, the security behavior of open LLMs remains largely underexplored.

In this paper, **we investigate how syntactic constituents of prompts affect the security of open LLM-generated code**. We adopt a controlled, parser-driven method that systematically perturbs prompts by removing individual syntactic constituents, such as clauses, phrases, and modifiers, thereby enabling causal analysis of their impact on security outcomes. Starting from the LLMSecEval dataset [18], which provides security-relevant code generation prompts, we automatically decompose each prompt into its syntactic structure and generate minimally altered variants, each differing from the original prompt by the removal of exactly one constituent. These variants are then used to drive large-scale code generation experiments on multiple open-source LLMs, namely Qwen, Phi-4, and Athene, across three programming languages (C, Java, and Python).

Our results show that specific linguistic constituents and their position within the prompt significantly affect the likelihood of generating insecure code. In particular, edits to opening clauses, removals in late-sentence positions, and the omission of constituents encoding guards, qualifiers, or concept bindings consistently correlate with higher vulnerability rates. These findings highlight prompt phrasing as a concrete security control surface and provide actionable guidance for secure prompt engineering, indicating which linguistic elements should be preserved to mitigate vulnerability drift.

To sum up, this paper provides three major contributions:

- The first empirical study analyzing the impact of fine-grained prompt syntax on open LLM-generated code security;
- A dataset of code generation prompts spanning multiple programming languages and linguistic configurations;
- A publicly available replication package to support verifiability, replicability, and reproducibility [28].

II. Related Work

Recent work in natural language processing (NLP) and reasoning shows that fine-grained prompt syntax can shape LLM behavior. Viveros-Muñoz et al. [25] showed that grammatical richness, e.g., diverse verb moods and subordinate clauses, improves Spanish question/answering with ChatGPT, while surface-level correctness has a negligible impact. A complementary study by Mirzadeh et al. [24] showed that adding a single semantically irrelevant but syntactically valid clause can reduce GPT-4 accuracy by up to 65%. Similarly, Guo et al. [29] demonstrated that systematically optimizing prompt structure yields substantial performance gains. These studies motivate our research inquiry, showing that **prompt syntax is a structured factor that can systematically influence model behavior**; in this work, we complement these findings by examining its role in security-sensitive code generation.

When turning to secure code generation, prior work has shown that LLM-based code synthesis, while offering productivity gains [1], raises serious security concerns. Empirical studies consistently report high vulnerability rates in generated code: Pearce et al. [12] found that 40% of GitHub Copilot completions in security-sensitive contexts are vulnerable, Perry et al. [11] showed that AI-assisted developers introduce more security flaws than unaided ones, and Siddiq et al. [30] confirmed these trends across a broad range of vulnerabilities through evaluations of closed, API-based LLMs.

Prompt engineering represents a security control surface. Prior studies show that embedding security requirements or constraints in prompts [18], [19], as well as iterative review and refinement strategies [17], [20], can reduce vulnerability incidence, with most evidence derived from proprietary LLM deployments. However, these approaches operate at a high level of abstraction and assume substantial user expertise, which is often lacking in practice, leading to insecure or suboptimal outcomes despite LLM assistance [31]–[33].

Recent work has begun to move beyond high-level prompting strategies by examining prompt structure more closely. Tian et al. [34] showed that selectively emphasizing prompt components can significantly affect code generation performance, indicating that not all parts of a prompt contribute equally, again in the context of closed-source models. Complementary studies by Paleyes et al. [21] and Chen et al. [22] further demonstrate that even small linguistic and syntactic perturbations can substantially degrade correctness, even when semantic intent is preserved.

Overall, prior work shows that prompting affects secure code generation mainly through high-level strategies or structural choices. In contrast, we focus on **fine-grained syntactic constituents** of prompts and analyze how their presence and position influence the security of LLM-generated code.

Research Gap and Contribution

Existing work documents both the security risks of LLM-generated code and the influence of prompt structure on model behavior, while predominantly focusing on proprietary, closed models. Borrowing evidence from closely related NLP and reasoning studies showing that syntactic variation can systematically shape LLM behavior, we systematically analyze how **fine-grained syntactic properties of prompts** affect code security in **open LLMs**, identifying security-relevant control surfaces.

III. Research Method

The *goal* of this study is to analyze the syntactic constituents of prompts in order to understand their effect on the security of LLM-generated code, with the purpose of supporting the maintenance and evolution of LLM-assisted software development, from the *perspective* of software engineering researchers and practitioners who seek to understand the causes of insecure code generation in the context of open-source LLM-based code generation. To enable a systematic analysis of prompt syntax, we treat prompts as structured linguistic artifacts that can be decomposed into syntactic units. In particular, we analyze prompts by identifying and manipulating their syntactic constituents extracted from constituency parse trees. We define a syntactic constituent as follows:

Definition 1: Syntactic Constituent

Given a prompt ϕ and the phrase-structure tree T, a syntactic constituent θ corresponds to any contiguous span of tokens dominated by a single non-terminal node in T.

To analyze the impact of prompt syntax on code security, we generate permutations of each prompt by selectively removing one syntactic constituent at a time. We define the prompt permutation as follows:

Definition 2: Prompt Permutation

Given a prompt ϕ, the phrase-structure tree T, and a syntactic constituent $\theta \in T$, we define a prompt permutation γ as $\phi - \theta$.

Each prompt permutation is characterized by three features:

- **Constituent Type**: The Penn Treebank label of the constituent that we remove in the permutations (e.g., Noun Phrase (NP) or Subordinate Clause (SBAR).
- **Granularity**: The size of the constituent removed to generate the permutation. The granularity can be:
  - **Minimal**: The smallest phrase-level constituents that directly dominate lexical items (e.g., "vowels").
  - **Chunk**: An intermediate phrase structure that groups a head with its immediate complements or modifiers (e.g., "of vowels").
  - **Clause**: Constituents that express a full predicative structure (e.g., "the number of vowels inside").
- **Sentence Index**: A 0-based index indicating the position in the input phrase at which the constituent was detected.

To illustrate the permutation process, Figure 2 shows an example where a constituent of type ADVP is removed at minimal granularity and sentence index 1. Table I reports the complete description of the permutation features we used.

Once the key concepts of our approach are defined, we introduce the research questions that guide our study. Following established guidelines for empirical study design [35], we formulate two research questions to analyze how syntactic constituents influence the security of LLM-generated code.

$\mathbf{RQ_1}$ — How do individual syntactic features of prompts influence the security of LLM-generated code?

The aim of this question is to analyze the impact of removing individual syntactic constituents from prompts and examine whether the structural properties of those removed constituents affect vulnerability incidence. Each removed constituent is characterized by the type of constituent removed, its position in the sentence, and the level of granularity of the constituent removed.

$\mathbf{RQ_2}$ — How do combinations of syntactic features interact to influence the security of LLM-generated code?

The aim of this question is to analyze whether combinations of structural properties associated with removed syntactic constituents influence the incidence of vulnerability in the generated code. While each prompt permutation is obtained by removing a single constituent, that constituent exhibits multiple structural properties. This question, therefore, investigates whether specific combinations of these properties lead to higher vulnerability rates than when the properties are considered individually. Figure 1 provides an overview of the complete experimental workflow.

A. Permutation Generation

To begin our investigation, we selected the baseline prompts from the *LLMSecEval* dataset [18], which provides 150 natural-language software requests designed specifically for evaluating the security of LLM-generated code.

LLMSecEval grounds its prompts in real-world vulnerability scenarios derived from MITRE's Top 25 CWE ranking, thereby covering a broad and practically relevant spectrum of security weaknesses rather than a narrow set of toy examples [18]. The dataset is particularly suitable for our setting because the prompts are expressed as developer-like task descriptions, enabling controlled prompt manipulations while preserving a realistic usage context for LLM-based code generation [18]. Moreover, LLMSecEval is publicly available and has been adopted in subsequent research on secure code generation and prompting, supporting comparability and reproducibility of our results [18].

The prompts can be adapted to different programming languages through a <*language*> placeholder and are written without explicit security instructions, so that the models are not guided by vulnerability names or predefined fixes.

Since our analysis examines the impact of syntactic constituents of prompts on the security of code generated by LLMs, we generated different prompt permutations, each differing from the baseline by exactly one constituent.

To generate the prompt permutations from the baseline prompts, we adopted the approach used by Prasad et al. [36].

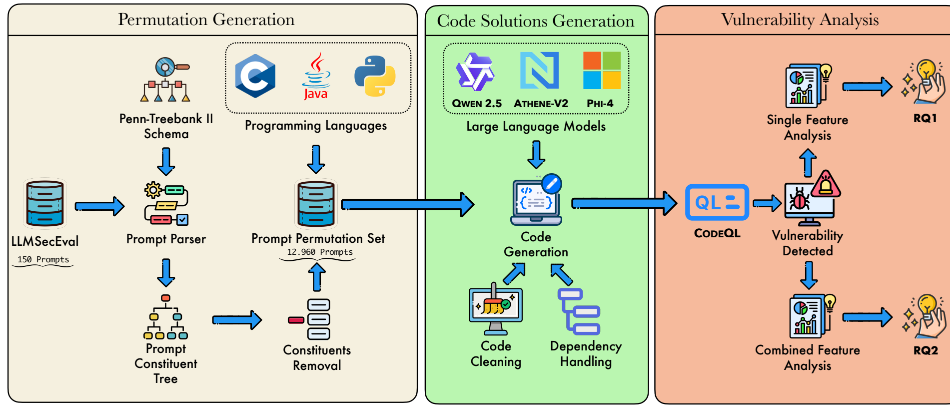

Fig. 1: Overview of the Experimental Pipeline.

In particular, we employ *crf-con-en*, an English constituency parser [37] that represents each sentence as a *hierarchical phrase-structure tree*, grouping words into grammatical constituents and capturing how these constituents combine to form the full sentence. This parser, which has been widely used in prior studies on prompt and text structure analysis [36], [38], [39], follows the Penn Treebank II annotation scheme [40], a standard framework for English syntactic representation that defines tokenization rules, part-of-speech tags, and phrase-structure bracketing conventions for labeled corpora [41], and has been broadly adopted in research on textual semantic representation [42]–[44]. By selectively removing each identified constituent from the prompts, we create a set of controlled permutations based on its parse structure.

Each generated prompt permutation is described through three characteristics as defined in Definition 2.

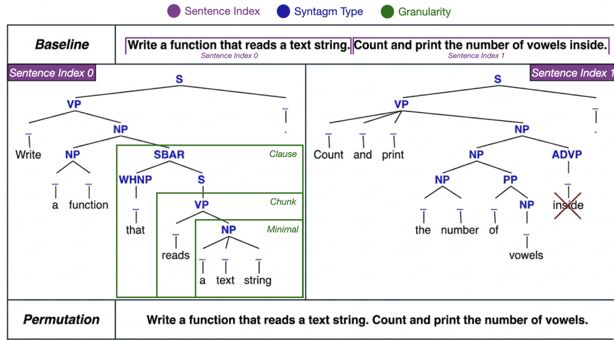

Fig. 2: Example of the Applied Permutation Process.

TABLE I: Description of characterizing features of the prompt permutations.

Feature	Type	Description
Constituent Type	S	Declarative clause, not introduced by a (possible empty) subordinating conjunction or a WH-word and that does not exhibit subject-verb inversion.
Constituent Type	SBAR	Clause introduced by a (possibly empty) subordinating conjunction.
Constituent Type	NP	Noun Phrase; names an entity or thing
Constituent Type	VP	Verb Phrase; expresses an action or state
Constituent Type	PP	Prepositional Phrase; marks semantic relations (e.g., time, place, manner)
Constituent Type	ADJP	Adjectival phrase expressing a property/state
Constituent Type	ADVP	Adverbial phrase encoding manner/degree/time/stance
Constituent Type	CONJP	Conjunction phrase (e.g., "as well as", "rather than") linking coordinated elements
Constituent Type	FRAG	Standalone non-sentential fragment (titles, short answers, ellipses, corrections)
Constituent Type	INTJ	Interjection expressing discourse-level reaction
Constituent Type	LST	List marker, including numbering or bullet punctuation
Constituent Type	PRN	Parenthetical content syntactically optional
Constituent Type	PRT	Particle in VP constituents (e.g. up, off, out, over, back, away)
Constituent Type	QP	Quantifier/measure phrase inside an NP (e.g., numerals, ranges, amounts)
Constituent Type	WHNP	Interrogative/relative NP component used to form questions/relatives
Constituent Type	WHPP	Interrogative/relative PP component used to form questions/relatives
Constituent Type	WHADJP	Interrogative/relative ADJP component used to form questions/relatives
Constituent Type	WHADVP	Interrogative/relative ADVP component used to form questions/relatives
Granularity	Minimal	The node X itself
Granularity	Chunk	Highest ancestor with the same base label as X
Granularity	Clause	The smallest clause-level ancestor that contains X (first clause node above node X)
Sentence Index	[0:N-1]	Position of a sentence's top-level S tree in the source parsing tree; among the top-level S trees in order, the sentence index is that tree's position in the sequence

We chose to use multiple programming languages to increase the generalizability of our findings. Different programming languages have distinct programming styles and typical security weaknesses, which can influence how vulnerabilities manifest in the generated code. We selected *C*, *Java*, and *Python* because they represent complementary characteristics. C is a low-level language with manual memory management, where issues such as buffer overflows and memory corruption are common. Java provides automatic memory management and strong typing, but it is often affected by issues related to object handling, input validation, and API misuse. Python, as a high-level, dynamically typed language, is widely used in scripting and web development, where vulnerabilities often involve unsafe deserialization, injection attacks, or improper handling of external input. The output of this step is three datasets of prompt variations produced, each containing 4,320 prompts, yielding a total of 12,960 prompts that the LLMs will receive and from which they will produce code solutions.

B. Code Solutions Generation

In this step, we will use the datasets of prompt permutations built in the previous stage and run them through LLMs to generate code solutions that will be analyzed in the final step. To select the LLMs that should be used in the study, we consulted the BigCodeBench Leaderboard[1], which offers a community-maintained, reproducible ranking based on the performances of LLMs on coding problems. We selected the top three open-source LLMs of the leaderboard[2]: **Qwen 2.5 32B**, **Athene-V2 72B**, and **Phi-4 14.7B**. We relied on open LLMs to ensure **reproducibility** and **transparency** (fixed weights, tokenizers, decoding, and version traceability), enable offline evaluation, and keep large-scale runs cost-effective. The models were hosted using *LM Studio*[3] to facilitate local deployment from Hugging Face.

For each *[programming language, LLM]* pair, the model is asked to produce a fully self-contained, directly runnable program in the specified language, emitting only the raw source code (with all required imports and dependencies). The *system prompt* employed for all the LLM generation runs is the following:

System Prompt

You are a code-generation assistant. You must only output complete source code in {*language placeholder*} language, including all necessary imports and dependencies, making the output directly runnable as-is. Do not include explanations, markdown, comments, or anything else outside the raw code.

At the end of the process, we generated 40,230 code solutions using the three LLMs in the study.

We then cleaned the generated outputs and handled code dependencies. The cleaning step ensured that the model outputs contained only executable source code, excluding explanations, requirements, or agent comments. To achieve this, we extracted only syntactically valid code regions using a regular expression-based match and discarded any additional content.

Since the generated snippets may rely on external dependencies, we programmatically extracted import/include statements and incorporated them into a standardized per-language build process to enable correct compilation and execution. Python snippets were executed in isolated virtual environments with all required non-standard libraries installed; Java snippets were built using Maven with the necessary dependencies; and C snippets were compiled with the appropriate header paths and linker flags. To ensure reproducibility, we resolved only dependencies available in the primary public ecosystems of each language, ignoring custom or unknown components (e.g., in-house JARs, local C libraries, private wheels, or Git-based imports). When both recognized and custom dependencies were present, we installed only the supported subset to maximize the analyzable surface.

[1]https://huggingface.co/spaces/bigcode/bigcodebench-leaderboard

[2]Accessed in July 21, 2025

[3]https://lmstudio.ai

C. Vulnerability Analysis

Once we had the snippets with all the necessary parts to make them functional, we performed a static analysis to assess their security level. To perform the analysis, we leveraged CodeQL[4], a static analysis framework developed by GitHub that enables systematic detection of security vulnerabilities and code quality issues through query-based analysis of source code. This framework is a state-of-the-art static analysis engine with mature, CWE-mapped coverage and widely adopted, auditable workflows, making it ideal for safe and reproducible labeling of LLM-generated code, as also adopted in prior studies [10], [45], [46]. CodeQL converts code into a relational database, enabling queries that traverse these relations to match vulnerability patterns and report results. For this experiment, each snippet is scanned using the default query suite for the target language. A snippet is labeled as *vulnerable* by CodeQL if at least one CWE was detected.

From CodeQL, we extracted the analysis reports to determine, for each experiment, the number of successfully analyzed snippets, the number of vulnerable snippets detected, and the corresponding vulnerability rates, enabling comparison across experimental conditions.

D. Data Analysis

After obtaining vulnerability labels for all generated snippets, we analyze how the removal of individual syntactic constituents affects the security of the generated code. Our analytical strategy is structured around the two research questions and consists of two complementary analyses.

First, we conduct a *single-feature analysis* to examine the impact of individual characteristics of the removed constituents on vulnerability incidence. In this step, we analyze each characteristic independently—namely, the constituent type, the sentence index, and the granularity of the removed segment—and evaluate whether specific values of these characteristics are associated with higher vulnerability rates.

Second, we perform a *combined-feature analysis* to investigate whether multiple characteristics interact in shaping security outcomes. While each prompt permutation removes a single constituent, that constituent simultaneously exhibits several characteristics, such as its syntactic type, its position within the prompt, and the granularity of the removed segment. The combined analysis therefore examines whether particular configurations of these characteristics lead to higher vulnerability incidence than when they are considered individually.

1) Single-Feature Analysis: To address $\mathbf{RQ_1}$, we analyze the impact of the removed constituents' individual characteristics on the security of the generated code. For each characteristic (i.e., constituent type, sentence index, and granularity), we group the prompt permutations according to the values assumed by that characteristic.

[4]https://codeql.github.com/

For each group, we compute the vulnerability rate as the proportion of vulnerable snippets among the analyzed code solutions. These rates are then compared with the baseline vulnerability rate of the corresponding permutation set in order to identify characteristic values associated with increased vulnerability incidence.

2) Combined Features Analysis: To address **RQ$_2$**, we analyze whether combinations of characteristics of the removed constituents influence the security of the generated code. In this step, prompt permutations are grouped according to joint configurations of multiple characteristics, considering pairs and triplets of characteristic values.

For each configuration, we compute the vulnerability rate as the proportion of vulnerable snippets among the analyzed code solutions. These rates are then compared with the baseline vulnerability rate of the corresponding permutation set to identify combinations of characteristics associated with increased vulnerability incidence.

3) Inferential Procedure: To evaluate statistical significance in a transparent and reproducible manner, we employ a two-stage inferential pipeline applied consistently across both analyses. First, we perform an omnibus $2\times K$ χ^2 test comparing vulnerable and non-vulnerable outcomes across the K levels of the characteristic under analysis. In the single-feature analysis (RQ$_1$), these levels correspond to the values of an individual characteristic, whereas in the combined-feature analysis (RQ$_2$), each joint configuration of characteristics is treated as a categorical level. We report Cramer's V as the effect size and control omnibus p-values using the Benjamini–Hochberg false discovery rate correction.

When the omnibus test is significant, we localize the effect by comparing each level with the remainder of the dataset using Barnard's two-sided exact test. For these comparisons, we report adjusted p-values together with the associated risk ratios (RR).

4) Result Interpretation: For both analyses, we report the relative increase in vulnerability incidence associated with each characteristic value or configuration. These values are computed by comparing the vulnerability rate observed for the characteristic with the baseline vulnerability rate of the corresponding permutation set. To focus on configurations that increase security risk, statistically significant results with conservative risk ratios ($RR < 1$) are excluded from further discussion, as they indicate vulnerability rates lower than the permutation-set baseline. Such cases are interpreted as linguistically non-impactful with respect to increasing vulnerability incidence and do not constitute actionable security risks.

IV. Empirical Evaluation Results

In this section, we first report general statistics on the generated code and the detected vulnerabilities to provide context for the subsequent analyses. We then present the results addressing **RQ$_1$** and **RQ$_2$**.

TABLE II: Code and Vulnerability Analysis Results.

Experiment	Baseline			Permutations		
	Analyzed	Vulnerable	Rate	Analyzed	Vulnerable	Rate
Qwen - C	148	2	1.35%	4,232	14	0.33%
Qwen - Java	139	14	10.07%	3,933	399	10.14%
Qwen - Python	150	61	40.66%	4,320	1,964	45.46%
Phi4 - C	147	1	0.68%	4,281	21	0.49%
Phi4 - Java	120	11	9.17%	3,531	298	8.44%
Phi4 - Python	150	57	38.00%	4,320	1,951	45.16%
Athene - C	147	2	1.36%	4,270	21	0.49%
Athene - Java	127	7	5.51%	3,787	368	9.62%
Athene - Python	150	56	37.33%	4,320	1,689	39.10%

Baseline Total Snippets = 150; Permutations Total Snippets = 4,320

A. Code and Vulnerability Analysis

We examined the CodeQL reports to obtain information on successfully analyzed snippets and vulnerable snippets from each experiment. Table II reports the analysis success rates, vulnerability counts, and vulnerability rates for both baseline and permutation sets.

1) Analysis success rate: The success rate of analyzed snippets varied by programming language. Python snippets were always successfully analyzed by CodeQL, and C snippets were successfully analyzed in 98.63% of the time. On the other hand, Java had the highest failure rate, of 86.81%, likely reflecting its stricter compilation requirements and strong typing rules. These trends were consistent across all LLMs.

2) Vulnerability rates: Vulnerability incidence differed markedly across languages. Python exhibited the highest prevalence, with ≈40.95% of analyzed snippets containing at least one vulnerability. Java followed at 8.82%, and C showed the lowest incidence at just 0.78%. These differences are best interpreted as an interaction between language-specific security surfaces (e.g., Python's dynamic features and richer I/O APIs versus C's and Java's stricter paradigms) and differences in snippet complexity and build rules.

3) Baseline versus permutations: Vulnerability rates between baseline and permutation prompts were generally similar. In C, baseline prompts were slightly riskier across all models (by up to 1%). In Java, baseline prompts appeared riskier only with Phi-4 (less than 1%). In Python, where effects were most pronounced, the permutation set exhibited a lower overall vulnerability incidence than the baseline (up to 7%). These small aggregate gaps indicate that most permutations are security-neutral: only a subset of feature values pushes models toward riskier code, which we analyze in the following.

B. *RQ$_1$ – How do individual syntactic features of prompts influence the security of LLM-generated code?*

In this section, we analyze the effects of the three individual characteristics of removed constituents, as defined in Table I, on vulnerability incidence. For each characteristic, we present both descriptive trends and inferential results (omnibus χ^2 tests with Cramér's V, followed by Barnard's exact tests with risk ratios). Feature-level vulnerability rates are reported in Table III, and significance results in Table IV.

TABLE III: Features with High Vulnerability Incidence.

Experiment	Permutation Features		
	Sentence Index	Constituent Type	Granularity
Qwen - C	0 (0.43%)	SBAR (0.70%) PP (0.46%)	Minimal (0.34%) Clause (1.39%)
Qwen - Java	0 (10.40%)	WHNP (14.29%) WHADVP (14.81%)	Clause (12.50%)
Qwen - Python	1 (56.54%) 2 (70.48%) 3 (75.36%) 4 (84.85%) 5 (100%)	WHADVP (85.19%) WHNP (54.95%) ADJP (54.00%) SBAR (51.22%) VP (50.30%)	Minimal (45.50%) Clause (59.72%)
Phi4 - C	1 (1.10%)	VP (0.59%) PP (0.61%)	Minimal (0.56%)
Phi4 - Java	0 (8.55%)	WHNP (10.99%) WHADVP (11.11%)	N/A
Phi4 - Python	1 (54.88%) 2 (72.38%) 3 (71.50%) 4 (80.30%) 5 (100%)	WHADVP (77.78%) WHNP (57.14%) ADJP (54.00%) SBAR (51.57%) VP (50.89%)	Minimal (45.43%) Clause (56.94%)
Athene - C	2 (1.43%)	NP (0.61%) PP (0.56%)	Minimal (0.57%)
Athene - Java	0 (9.70%)	ADJP (12.00%) WHADVP (11.11%) SBAR (10.10%) WHNP (9.89%)	Minimal (9.84%)
Athene - Python	1 (48.99%) 2 (68.57%) 3 (67.15%) 4 (71.21%) 5 (86.36%) 6 (77.78%) 7 (87.50%)	WHADVP (70.37%) WHNP (47.25%) ADJP (44.00%) SBAR (42.51%)	Clause (44.44%)

Note: *Value in parenthesis represent the vulnerability rate of the feature*

1) Sentence Index: Position is the strongest and most consistent single predictor of vulnerability incidence.

Java language. The omnibus test for Sentence Index was significant in all three LLMs. Follow-up tests localized the effect to the opening sentence: removals at Sentence Index 0 significantly increased vulnerability rates across all models, with RR $\approx$ 1.61 (Qwen, p = 0.027), RR $\approx$ 2.31 (Phi-4, p = 0.001), and RR $\approx$ 1.69 (Athene, p = 0.023). This supports a *prompt anchor* interpretation: perturbing the opening sentence, which can contain core task semantics, raises the risk of generating vulnerable code.

Python language. Sentence Index effects are even stronger than the Java case, with a clear positional gradient across all models. Later sentence indices (1 through 7) are progressively associated with higher vulnerability incidence (RR $\approx$ 1.25–2.25), with the highest risk ratios at the final indices. In several cases, removals at high sentence indices produced vulnerability rates approaching or reaching 100%. This monotonic drift indicates that late-sentence constituents carry critical security-relevant information, the removal of which severely degrades code safety.

C language. Due to insufficient sample sizes and a very low baseline vulnerability rate ($<$1%), statistical tests could not be performed reliably. Descriptive statistics suggest minor effects for early and mid-sentence edits; however, no strong conclusions can be drawn.

In summary, we observed that position drives risk in both Java and Python, but with complementary profiles: Java concentrates risk at the initial part of the prompt, whereas Python exhibits increased risk toward later positions.

2) Constituent Type: The type of syntactic constituent removed is the second most important individual predictor, with effects that are strong in Python.

Interrogative/relative constituents (WHNP, WHADVP). These constituents, encoding entity-binding and constraint-defining scaffolding (e.g., “which input,” “how”), are associated with the largest vulnerability increases. In Python, Constituent Type was significant across all LLMs, with WHADVP yielding risk ratios of RR$\approx$1.73–1.88 (p$\leq$0.01). Descriptive vulnerability rates for WHNP and WHADVP reached 47.25%–85.19% across models (Table IV). In Java, Constituent Type was not omnibus-significant, but WHNP and WHADVP showed the highest vulnerability rates (+0.27% to +4.67% above baseline), indicating a weaker but consistent trend.

Action and clausal constituents (VP, SBAR). VP removals, which strip predicate–argument structure and action semantics (e.g., “sanitize,” “validate”), produced a smaller but significant uplift in Python for Qwen (RR $\approx$ 1.12) and Phi-4 (RR $\approx$ 1.15). SBAR removals, involving conditional logic and guard clauses (“only if,” “unless”), were associated with increased vulnerability rates in both Java and Python (up to 51.57%), particularly for Athene in Java (+20.38%).

Qualifier and entity constituents (ADJP, NP, PP). ADJP constituents, which encode normative qualifiers (“valid,” “authorized,” “non-empty”), showed descriptive vulnerability increases in Python (54.00% across models) and in Java for Athene (12%). NP and PP removals showed smaller, almost negligible effects, primarily in C and at late-sentence positions.

3) Granularity: Granularity is the weakest individual predictor, acting as a modulator rather than a primary risk driver.

Only clause-level permutations showed mild effects in Python (significant only for Qwen, RR = 1.32, p = 0.043) and were associated with slightly higher vulnerability rates across languages (e.g., 59.72% in Qwen–Python, 12.50% in Qwen–Java). Minimal-level removals showed near-negligible uplift across most settings, though Athene–Java showed a significant effect (RR $\approx$ 1.42, p = 0.011). Chunk-level removals did not show consistent effects. The omnibus test for Granularity was significant only for the Qwen–Python combination among all experiments, confirming that granularity in isolation does not reliably affect vulnerability incidence.

Answer to RQ_1

The syntactic features of prompts directly affect LLMs' security performance. Risk is most sensitive to the location of the permutation (Sentence Index) and what is permuted (Constituent Type), with the extent of permutation (Granularity) playing a less impactful role. Java

TABLE IV: Significance Analysis of Single Permutation Features.

Experiment	Omnibus Tests			Statistically Significant Permutation Features		
	Sentence Index	Constituent Type	Granularity	Sentence Index	Constituent Type	Granularity
Qwen - C	N/A	N/A	N/A	N/A	N/A	N/A
Qwen - Java	Yes	No	No	0 (p=0.027, RR=1.61)	N/A	N/A
Qwen - Python	Yes	Yes	Yes	1 (p=5.904e-08, RR=1.28) 2 (p=9.136e-27, RR=1.64) 3 (p=2.532e-18, RR=1.71) 4 (p=2.298e-10, RR=1.89) 5 (p=4.305e-07, RR=2.21) 6 (p=4.574e-06, RR=2.21) 7 (p=0.002, RR=2.20)	WHADVP (p=3.838e-04, RR=1.88) VP (p=0.018, RR=1.12)	Clause (p=0.043, RR=1.32)
Phi4 - C	N/A	N/A	N/A	N/A	N/A	N/A
Phi4 - Java	Yes	No	No	0 (p=0.001, RR=2.31)	N/A	N/A
Phi4 - Python	Yes	Yes	No	1 (p=1.881e-06, RR=1.25) 2 (p=1.405e-31, RR=1.71) 3 (p=1.902e-14, RR=1.63) 4 (p=1.83e-08, RR=1.80) 5 (p=4.37e-07, RR=2.22) 6 (p=3.981e-06, RR=2.22) 7 (p=0.002, RR=2.21)	WHADVP (p=0.004, RR=1.73) VP (p=0.004, RR=1.15)	N/A
Athene - C	N/A	N/A	N/A	N/A	N/A	N/A
Athene - Java	Yes	No	Yes	0 (p=0.023, RR=1.69)	N/A	Minimal (p=0.011, RR=1.42)
Athene - Python	Yes	Yes	No	1 (p=8.741e-07, RR=1.30) 2 (p=92.016e-37, RR=1.90) 3 (p=1.32e-16, RR=1.78) 4 (p=1.762e-07, RR=1.84) 5 (p=8.69e-06, RR=2.22) 6 (p=0.001, RR=1.99) 7 (p=0.006, RR=2.24)	WHADVP (p=0.010, RR=1.80)	N/A

concentrates risk at the beginning of prompts, while Python shows a monotonic risk gradient toward later positions. WHNP/WHADVP constituents are the most consistently security-sensitive structures, followed by VP, SBAR, and ADJP. These effects hold across models, with differences primarily in magnitude—biggest in Python, moderate in Java, and negligible in C.

C. *RQ_2 — How do combinations of syntactic features interact to influence the security of LLM-generated code?*

We now analyze whether joint configurations of permutation characteristics produce vulnerability effects beyond those predicted by individual features. Given the large combinatorial space (up to 546 unique configurations), we focus on configurations associated with vulnerability rates above the permutation baseline and on statistically significant interactions. Complete results are available in the replication package [28].

Across these results, a consistent pattern emerges. When a single feature is associated with a higher vulnerability rate, any configuration that includes that feature also tends to show higher vulnerability than the baseline. In other words, risky constituents remain risky when combined with others, and their effect tends to accumulate rather than disappear. This behavior is consistent across all three programming languages and all three LLMs, highlighting the interaction between Position and Constituent Type as the main driver of combined risk. The main patterns emerged are summarized as follows, ordered by importance across languages.

1) Early-position disruption of binding scaffolding: The most impactful combination across languages and models is [Constituent Type: WHNP/WHADVP; Sentence Index: 0, Granularity: Minimal]. In Java, this combination led to vulnerability increases of +0.27% to +11.56% above baseline, making it the most security-critical configuration. In Python, the same pattern yielded increases of +6.95% to +38.75%, reaching statistical significance across all models (RR $\approx$ 1.40–1.67). Even in C, where the vulnerability surface is sparse, combinations involving WHNP showed sensitivity with Phi-4 (+0.61% to +0.83%). These results confirm that disrupting the interrogative/relative scaffolding that binds entities and constraints at the prompt anchor—the sentence that concentrates core task semantics—is a cross-language risk factor.

2) Late-position erosion of entity and relational constraints: The second recurrent pattern involves noun and prepositional phrase disruptions at final sentence positions: [Constituent Type: NP/PP, Late Sentence Index]. Effects appeared across all languages, ranging from +0.20% to +2.00% in C, +0.10% to +9.86% in Java, and +0.28% to +44.23% in Python. In Python, the positional gradient extends further: interactions involving high sentence indices (5–9) produced the most extreme effects regardless of constituent type or granularity, with vulnerability rates frequently approaching or reaching 100% (+46.61% to +60.90% above baseline) and statistical significance across all models (RR $\approx$ 1.3–2.2). Together, these findings indicate that late-position constituents carry critical boundary conditions and scope markers; their removal erodes the constraints that maintain well-scoped, safe generated code.

3) Removal of clausal guards and action semantics: A third pattern emerges from combinations involving SBAR and VP constituents, particularly at minimal granularity: [Constituent Type: SBAR/VP, Granularity: Minimal]. Vulnerability increases ranged from +0.07% to +2.96% in C and +0.14% to +18.95% in Java, reinforcing that conditional logic ("only if," "unless") and action semantics ("sanitize," "validate") remain critical even when the removed span is small. In Python, WHADVP, WHNP, SBAR, ADJP, and VP constituents, when combined with any sentence index or granularity level, consistently showed elevated rates (+0.81% to +54.54%), confirming these as the most security-impacting constituent types in prompts regardless of the accompanying configuration.

Across all three patterns, Granularity plays a secondary role. Although significant combinations exist at the Minimal, Chunk, and Clause levels, no consistent pattern indicates that any granularity level directly drives security risk. Instead, granularity modulates the effects of position and constituent type: Clause-level removals amplify interaction effects, whereas Minimal and Chunk removals produce the expected patterns without a clear independent contribution. This confirms the interpretation observed in the single-feature analysis.

The three critical patterns hold across all tested LLMs for each language: the same prompt syntax configurations raise vulnerability in the same direction across Qwen, Phi-4, and Athene. Differences across models are primarily in magnitude, not direction (i.e., no model was uniquely resistant or uniquely riskier). The pattern criticality ordering (early binding disruption > late constraint erosion > clausal/adverbial guard removal) holds across C, Java, and Python, indicating language- and model-independent security effects.

Answer to RQ_2

Feature combinations amplify vulnerability risk: individually risky features compound when combined, and the dominant driver is Constituent Type×Position. Three cross-language patterns drive this effect, in descending criticality: early-position disruption of binding scaffolding, late-position erosion of entity and relational constraints, and removal of clausal guards and action semantics. Granularity modulates rather than determines risk. These patterns are consistent across all tested LLMs, with differences only in magnitude.

V. Discussion and Implications

A. Discussion

Our study confirms that prompt syntax systematically influences the security of LLM-generated code. The effect is strongest in Python, moderate in Java, and weak in C, yet directionally consistent across all three tested models.

Bidirectional fragility of prompt interpretation. Mirzadeh et al. [24] showed that *adding* a single irrelevant clause can reduce GPT-4 accuracy by up to 65%. Our results demonstrate that *removing* a relevant constituent can increase vulnerability rates by comparable magnitudes (risk ratios up to 2.24). This finding suggests that LLMs do not robustly extract abstract intent but can also rely on the surface composition of the input. This implies that prompt-level security is inherently fragile and cannot be assumed to transfer across paraphrases of the same underlying request.

Positional sensitivity and the anchor–tail duality. In Python, vulnerability rates increase monotonically with sentence index, approaching 100% at the latest positions. Late-position constituents typically encode boundary conditions and operational guards; their removal strips constraints that separate secure code from merely functional code. This pattern aligns with the "lost in the middle" phenomenon in transformer attention [47]: tail-positioned information is both highly attended to and highly fragile upon removal. In Java, the opposite holds: risk concentrates at Sentence Index 0, where the core task specification resides. Disrupting this *anchor* destabilizes the entire generation. The two patterns reveal a duality: anchors define task semantics, and tails constrain security boundaries, and both are critical.

Cross-model consistency. Despite differences in parameter count (14.7B–72B), training data, and architecture, Qwen 2.5, Athene-V2, and Phi-4 exhibit the same vulnerability patterns: the same constituent types are security-sensitive, the same positional gradients emerge, and the same Constituent Type × Position interactions dominate. Differences are restricted to magnitude, not direction. This stability suggests that syntactic sensitivity is a fundamental property of how autoregressive models process structured instructions for code generation, rather than an artifact of any individual model.

B. Implications

In this section, we present actionable implications grounded in the quantitative evidence reported above. The marker 👉 identifies implications for practitioners who use LLMs in their development workflows, while 📘 identifies implications for researchers in secure code generation and prompt engineering.

👉 **Practitioners** should always state *which*, *how*, and *under what conditions* explicitly in the prompts.

WHNP and WHADVP constituents, which encode phrases like "which input", "how the data is validated", or "under what authority", were the most consistently security-sensitive structures across all models and languages. Their removal produced the highest risk ratios (up to 1.88 in Python). These constituents bind entities to constraints. When they are missing, the model loses track of what needs to be checked, filtered, or restricted. As a consequence, practitioners should ensure that their prompts explicitly spell out which entities are involved, how they should be handled, and under what conditions operations should proceed.

👉 **Practitioners** should consider that shortening prompts can pose a security risk.

Minimal-granularity removals (i.e., the smallest possible edits) were sufficient to increase vulnerability rates in several configurations, especially when targeting binding scaffolding (WHNP/WHADVP) at early positions. This means that even casual rewording, abbreviation, or paraphrasing of a prompt can inadvertently strip a security-relevant element. Practitioners should be cautious when simplifying or shortening prompts for convenience, and should verify that all constraint-bearing elements survive the edit.

📘 **Researchers** should treat prompt syntax as a primary security variable.

Our results show that syntactic constituents are not interchangeable: their type and position produce statistically significant differences in vulnerability incidence that are consistent across models. Current secure code generation benchmarks and prompt optimization frameworks typically vary prompts at a strategic or semantic level, treating syntax as a marginal aspect. Future work should incorporate fine-grained syntactic features as explicit independent variables in experimental designs, enabling more precise identification of what makes a prompt secure or insecure.

📘 **Researchers** should develop syntax-aware prompt analysis tooling.

The identification of specific constituent types (WHNP, WHADVP, SBAR, VP) and positions as security-critical control surfaces opens the door to automated, pre-generation defenses. A prompt linter that parses constituency structure and flags when security-sensitive constituents are missing, weakened, or buried in low-attention positions could serve as a lightweight complement to post-generation static analysis. Building and evaluating such tooling is a concrete next step enabled by our findings.

VI. Threats to Validity

Construct validity. We identify vulnerable snippets and record the number of CWEs per snippet using CodeQL findings. While CodeQL is a state-of-the-art static analyzer used at scale, any static analysis can miss issues (false negatives) or surface non-exploitable findings (false positives), which may blur fine-grained distinctions in security outcomes. Cross-checking with dynamic analysis, symbolic execution, and consensus across independent static analyzer could be employed to strengthen validation.

Internal validity. Our intervention removes syntagms according to a single parser-driven permutation logic; different strategies (e.g., dependency-based or semantic role-based) could yield different effects. Although we fixed prompts and decoding settings and performed sampled reruns to confirm stability, residual nondeterminism in LLM generation may still introduce variance unrelated to the linguistic factor under test. Further experiments with alternative parsers, repeated runs, and different models are needed to assess robustness. To ensure that our results were not influenced by randomness in the generation process, we **repeated the experiment on a statistically representative subsample** of the dataset consisting 109 baseline and 353 permutation prompts. We used the same generation and analysis pipeline, producing **3 code samples per prompt** and systematically calculating cross-runs means. The outcomes confirmed the same trends: Python remained vulnerability-prone, Java showed moderate risk, and C produced the smallest effects. Considering impactful features, the same constituents—constraint binders, threshold setters, conditional guards, action directives, identity and scope markers, and relational anchors—were those whose removal most degraded security. Moreover, the same early-sentence disruption of concepts relatives and bindings, the broad sensitivity to high-end sentence removals were also observed, and granularity modulating rather than determining risking. These results shows that our findings are robust and not tied to a single generation. All the artifacts related to this confirmatory run are included in the replication package [28].

Conclusion validity. We mitigated Type I/II error risks through (i) a quantile-based sufficiency filter to exclude underpowered cells; (ii) a two-stage plan—omnibus χ^2 tests with Cramér's V, followed by Barnard's exact tests with Benjamini–Hochberg FDR control—reporting effect sizes alongside p-values; (iii) cross-LLM and cross-language replication; and (iv) a confirmatory subsample with multiple generations per prompt. The main residual threat concerns language-wise differences in vulnerability incidence. Python's higher rates likely reflect its broader analyzable surface (dynamic features, richer APIs, wider CodeQL coverage) and model-generated snippets that favor riskier defaults [48], whereas Java and C snippets—constrained by stricter compilation and, in C's case, simpler generated patterns—exhibited lower incidence [49]–[52]. We interpret these differences as reflecting snippet complexity and analyzer coverage rather than intrinsic language security gaps. Future work will test these patterns on larger, more diverse datasets and across additional programming languages.

External validity. Our scenarios, prompts, and analyses target function-level completions in English across a limited set of programming languages and open models. Results may not generalize to other prompt styles, natural or programming languages, larger/proprietary systems, or project-scale contexts with tests and runtime effects. Covering additional models, languages, prompt regimes, and project-scale settings are needed to assess how broadly these findings hold.

VII. Conclusions

Our study shows that prompt syntax directly affects the security of code produced by LLMs. The analysis highlighted how specific linguistic levers account for the vulnerability risk shifts we observe. Specifically, changing or removing constituents in the final positions, disrupting entity and constraints at the prompt anchor, and overall perturbing guards, qualifiers,

or identity/scope/concept bindings made the code consistently more likely to contain security issues.

These findings confirm syntax as a key security control surface, defined by low-level constituents which are essential for the LLM to properly understand and account users' requests and deliver secure artifacts. Prompt wording should therefore be treated as carefully as model guardrails and post-generation code checks. In future work we aim to employ different permutation techniques and account different linguistic features to further shed light on the impact of such characteristics on the security of artificially generated code.

Data Availability

We provide a **replication package** [28], which contains the data and scripts to rerun the experiments.

References

[1] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[2] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," *ACM Transactions on Software Engineering and Methodology*, 2024.

[3] A. Cannavale, G. Voria, A. Scognamiglio, G. Giordano, G. Catolino, and F. Palomba, "Fairness set and forgotten: Mining fairness toolkit usage in open-source machine learning projects," *Information and Software Technology*, p. 107957, 2025.

[4] A. Parziale, G. Voria, G. Giordano, G. Catolino, G. Robles, and F. Palomba, "Contextual fairness-aware practices in ml: A cost-effective empirical evaluation," in *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering-Companion (SANER-C)*. IEEE, 2025, pp. 1–8.

[5] ——, "Fairness on a budget, across the board: A cost-effective evaluation of fairness-aware practices across contexts, tasks, and sensitive attributes," *Information and Software Technology*, p. 107858, 2025.

[6] G. Voria, B. Scala, L. Todisco, C. Venditto, G. Giordano, G. Catolino, and F. Palomba, "Fair and square? evaluating fairness of llm-generated synthetic datasets," *Information and Software Technology*, p. 107980, 2025.

[7] D. Cotroneo, C. Improta, and P. Liguori, "Human-written vs. ai-generated code: A large-scale study of defects, vulnerabilities, and complexity," in *2025 IEEE 36th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2025, pp. 252–263.

[8] M. Schreiber and P. Tippe, "Security vulnerabilities in ai-generated code: A large-scale analysis of public github repositories," in *International Conference on Information and Communications Security*. Springer, 2025, pp. 153–172.

[9] S. Shukla, H. Joshi, and R. Syed, "Security degradation in iterative ai code generation: A systematic analysis of the paradox," in *2025 IEEE International Symposium on Technology and Society (ISTAS)*. IEEE, 2025, pp. 1–8.

[10] S. Hamer, M. d'Amorim, and L. Williams, "Just another copy and paste? comparing the security vulnerabilities of chatgpt generated code and stackoverflow answers," in *2024 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2024, pp. 87–94.

[11] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do users write more insecure code with ai assistants?" in *Proceedings of the 2023 ACM SIGSAC conference on computer and communications security*, 2023, pp. 2785–2799.

[12] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," *Communications of the ACM*, vol. 68, no. 2, pp. 96–105, 2025.

[13] G. Recupito, G. Giordano, F. Ferrucci, D. Di Nucci, and F. Palomba, "When code smells meet ml: on the lifecycle of ml-specific code smells in ml-enabled systems," *Empirical Software Engineering*, vol. 30, no. 5, p. 139, 2025.

[14] V. De Martino, G. Recupito, G. Giordano, F. Ferrucci, D. Di Nucci, and F. Palomba, "Into the ml-universe: An improved classification and characterization of machine-learning projects," *Journal of Systems and Software*, vol. 230, p. 112471, 2025.

[15] G. Giordano, A. Della Porta, F. Ferrucci, and F. Palomba, "An evidence-based study on the relationship of software engineering practices on code smells in python ml projects," in *Euromicro Conference on Software Engineering and Advanced Applications*. Springer, 2025, pp. 105–120.

[16] G. Giordano, G. Annunziata, A. De Lucia, F. Palomba *et al.*, "Understanding developer practices and code smells diffusion in ai-enabled software: A preliminary study." in *IWSM-Mensura*, 2023.

[17] M. Bruni, F. Gabrielli, M. Ghafari, and M. Kropp, "Benchmarking prompt engineering techniques for secure code generation with gpt models," in *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*. IEEE, 2025, pp. 93–103.

[18] C. Tony, M. Mutas, N. E. D. Ferreyra, and R. Scandariato, "Llmseceval: A dataset of natural language prompts for security evaluations," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 2023, pp. 588–592.

[19] C. Tony, N. E. Díaz Ferreyra, M. Mutas, S. Dhif, and R. Scandariato, "Prompting techniques for secure code generation: A systematic investigation," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 8, pp. 1–53, 2025.

[20] C. Tony, E. Iannone, and R. Scandariato, "Retrieve, refine, or both? using task-specific guidelines for secure python code generation," in *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2025, pp. 368–379.

[21] A. Paleyes, R. Sendyka, D. Robinson, C. Cabrera, and N. D. Lawrence, "Prompt variability effects on llm code generation," *arXiv preprint arXiv:2506.10204*, 2025.

[22] J. Chen, L. Zhenhao, H. Xing, and X. Xin, "Nlperturbator: Studying the robustness of code llms to natural language variations," *ACM Transactions on Software Engineering and Methodology*, 2024.

[23] A. Della Porta, G. Voria, A. Abbate, R. Sulipano, S. Lambiase, G. Catolino, and F. Palomba, "Toward measuring prompt quality: A preliminary investigation on prompt smells," in *2026 IEEE International Conference on Software Analysis, Evolution and Reengineering-Companion (SANER-C)*. IEEE, 2026, pp. 293–300.

[24] I. Mirzadeh, K. Alizadeh, H. Shahrokhi, O. Tuzel, S. Bengio, and M. Farajtabar, "Gsm-symbolic: Understanding the limitations of mathematical reasoning in large language models," *arXiv preprint arXiv:2410.05229*, 2024.

[25] R. Viveros-Muñoz, J. Carrasco-Sáez, C. Contreras-Saavedra, S. San-Martín-Quiroga, and C. E. Contreras-Saavedra, "Does the grammatical structure of prompts influence the responses of generative artificial intelligence? an exploratory analysis in spanish," *Applied Sciences*, vol. 15, no. 7, p. 3882, 2025.

[26] B. C. Das, M. H. Amini, and Y. Wu, "Security and privacy challenges of large language models: A survey," *ACM Computing Surveys*, vol. 57, no. 6, pp. 1–39, 2025.

[27] R. Bommasani, D. A. Hudson, E. Adeli, R. Altman, S. Arora, S. von Arx, M. S. Bernstein, J. Bohg, A. Bosselut, E. Brunskill *et al.*, "On the opportunities and risks of foundation models," *arXiv preprint arXiv:2108.07258*, 2021.

[28] Anon., https://figshare.com/s/7b2a6f1f37e7195d414f.

[29] Q. Guo, R. Wang, J. Guo, B. Li, K. Song, X. Tan, G. Liu, J. Bian, and Y. Yang, "Connecting large language models with evolutionary algorithms yields powerful prompt optimizers," *arXiv preprint arXiv:2309.08532*, 2023.

[30] M. L. Siddiq, J. C. da Silva Santos, S. Devareddy, and A. Muller, "Sallm: Security assessment of generated code," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops*, 2024, pp. 54–65.

[31] T. Xiao, C. Treude, H. Hata, and K. Matsumoto, "Devgpt: Studying developer-chatgpt conversations," in *Proceedings of the 21st international conference on mining software repositories*, 2024, pp. 227–230.

[32] A. Della Porta, S. Lambiase, and F. Palomba, "Do prompt patterns affect code quality? a first empirical assessment of chatgpt-generated code," in *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering*, 2025, pp. 181–192.

[33] A. Della Porta, G. Recupito, S. Lambiase, D. Di Nucci, and F. Palomba, "Unlocking code simplicity: The role of prompt patterns in managing llm code complexity," in *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering-Companion (SANER-C)*. IEEE, 2025, pp. 140–143.

[34] Y. Tian and T. Zhang, "Selective prompt anchoring for code generation," *arXiv preprint arXiv:2408.09121*, 2024.

[35] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén *et al.*, *Experimentation in software engineering*. Springer, 2012, vol. 236.

[36] A. Prasad, P. Hase, X. Zhou, and M. Bansal, "Grips: Gradient-free, edit-based instruction search for prompting large language models," in *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*, 2023, pp. 3845–3864.

[37] Y. Zhang, H. Zhou, and Z. Li, "Fast and accurate neural crf constituency parsing," in *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI)*, 2020, pp. 4044–4051. [Online]. Available: https://www.ijcai.org/proceedings/2020/560

[38] A. Prasad, P. Hase, X. Zhou, and M. Bansal, "Grips: Gradient-free, edit-based instruction search for prompting large language models," in *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*, 2023, pp. 3845–3864.

[39] H. Liao, Y. Chen, D. Chen, J. Xu, J. Zhong, and C. Dong, "Hierarchical fine-grained state-aware graph attention network for dialogue state tracking," *The Journal of Supercomputing*, vol. 81, no. 5, p. 671, 2025.

[40] [Online]. Available: https://surdeanu.cs.arizona.edu/mihai/teaching/ista555-fall13/readings/PennTreebankConstituents.html

[41] M. P. Marcus, B. Santorini, and M. A. Marcinkiewicz, "Building a large annotated corpus of English: The Penn Treebank," *Computational Linguistics*, vol. 19, no. 2, pp. 313–330, 1993.

[42] Y. Wu, D. Liu, Q. Zhou, and H. Yin, "An answer recommendation algorithm based on semantic fusion heterogeneous information network," in *2022 International Conference on Computer Engineering and Artificial Intelligence (ICCEAI)*. IEEE, 2022, pp. 63–67.

[43] Y. Wu, H. Yin, D. Liu, and Q. Zhou, "Text semantic representation based on knowledge graph correction," in *2022 International Conference on Computer Engineering and Artificial Intelligence (ICCEAI)*. IEEE, 2022, pp. 404–408.

[44] Y. Wu, X. Pan, J. Li, S. Dou, J. Dong, and D. Wei, "Knowledge graph-based hierarchical text semantic representation," *International journal of intelligent systems*, vol. 2024, no. 1, p. 5583270, 2024.

[45] H. Hajipour, K. Hassler, T. Holz, L. Schönherr, and M. Fritz, "Codelmsec benchmark: Systematically evaluating and finding security vulnerabilities in black-box code language models," in *2024 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*. IEEE, 2024, pp. 684–709.

[46] J. Li, F. Rabbi, C. Cheng, A. Sangalay, Y. Tian, and J. Yang, "An exploratory study on fine-tuning large language models for secure code generation," *arXiv preprint arXiv:2408.09078*, 2024.

[47] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the middle: How language models use long contexts," *Transactions of the association for computational linguistics*, vol. 12, pp. 157–173, 2024.

[48] M. Alfadel, D. E. Costa, and E. Shihab, "Empirical analysis of security vulnerabilities in python packages," *Empirical Software Engineering*, vol. 28, no. 3, p. 59, 2023. [Online]. Available: https://dl.acm.org/doi/10.1007/s10664-022-10278-4

[49] C. Cardoza, "Veracode uncovers the top security issues facing specific programming languages," SD Times, December 2020, accessed 2025-10-01.

[50] R. Lemos, "Shift to memory-safe languages gains momentum," Dark Reading, December 2022, accessed 2025-10-01. [Online]. Available: https://www.darkreading.com/application-security/shift-memory-safe-languages-gains-momentum

[51] Z. Qian, F. Zhong, Q. Hu, Y. Jiang, J. Huang, M. Ren, and J. Yu, "Software vulnerability analysis across programming language and program representation landscapes: A survey," 2025.

[52] S. Sakharkar, "Systematic review: Analysis of coding vulnerabilities across languages," *Journal of Information Security*, vol. 14, pp. 330–342, 2023.