2025

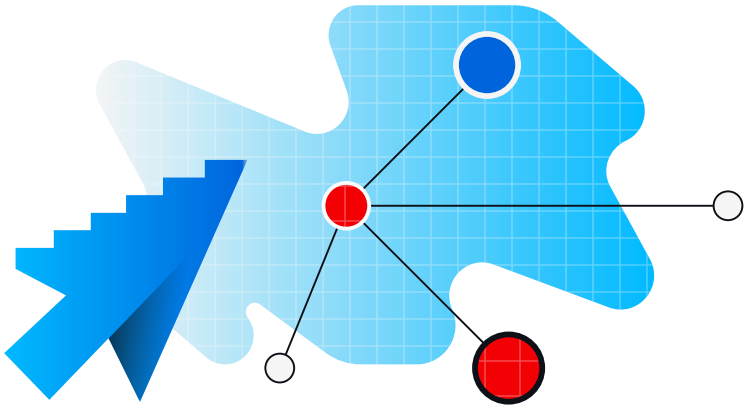
GenAI Code Security Report

ASSESSING THE SECURITY OF USING LLMS FOR CODING

VERACODE

Contents

Introduction



Generative AI is rapidly changing the way software is developed. Rather than code directly in some programming language, developers are increasingly describing the functionality they want in natural language and using large language models to generate the concrete code. Significant effort has been put into training these models for correctness, and recent assessments have found that newer, larger models are very good at generating code with the expected functionality. Less attention, however, has been paid to whether the resulting code is secure. The primary problem is that developers need not specify security constraints to get the code they want. For example, a developer can prompt a model to generate a database query without specifying whether the code should construct the query using a prepared statement (safe) or string concatenation (unsafe). The choice, therefore, is left up to the model.

The goal of this report is to quantify the security properties of AI-generated code across a range of languages and models. The central question we explore is: In the absence of any security-specific guidance, do large language models generate secure code or not? To evaluate this question, we developed a set of coding tasks for four popular programming languages: Java, Javascript, C#, and Python. These tasks involve filling in the missing part of a single function according to a comment describing the desired code. The key property of the tasks is that the requested functionality can be implemented in either a secure or insecure way. For each task, the insecure choice of implementation represents one of four known vulnerabilities (detailed later in the paper). We run our SAST tool on the resulting generated code to determine if it contains the vulnerability. For example, if the task asks the model to generate a SQL query, and it chooses the string concatenation implementation, our SAST tool will flag it as having a CWE 89 (“SQL Injection”), following the standard MITRE

classification system. For each combination of language and potential CWE, we constructed five different versions of the coding task in order to vary the conditions and context.

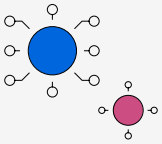
The complete test set consists of 80 coding tasks: four languages and four CWEs, with five examples of each. We give these 80 coding tasks to over 100 LLMs, covering a wide range of model sizes, vendors, and target applications (e.g., coding vs general purpose). Our goal is partly to assess the security properties of each model individually, but also to expose trends. We set out to answer questions such as: How does security trend with the size of the model? Have models been getting better at security over time? We largely avoid classifying results according to the vendor or organization providing the model.

Our overall results indicate that models fare poorly on security, and that, somewhat surprisingly, performance is largely flat across model sizes and over time: newer and larger models do not generate significantly more secure code.

The highlights of our findings are as follows:

Across all models and all tasks, **only 55% of generation tasks result in secure code**. In other words, in 45% of the tasks the model introduces a known security flaw into the code.

Security performance has been **largely unchanged over time**, even as models get better at generating syntactically correct code.



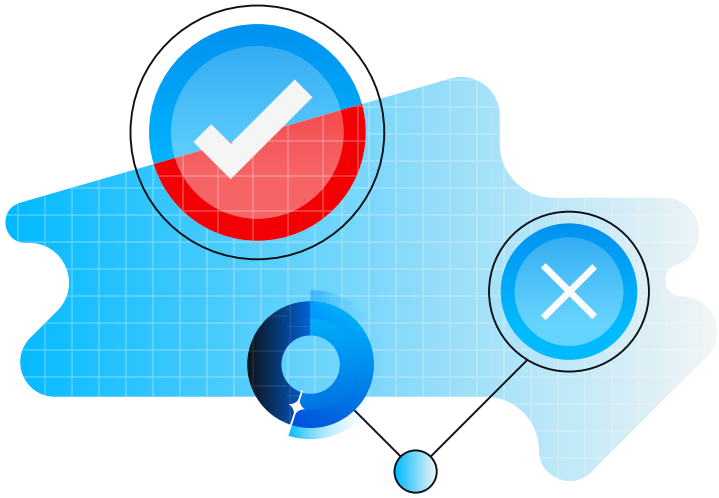
Larger models **do not perform significantly better** than smaller models

Security performance **varies dramatically** by CWE type.

Security performance **varies somewhat** across languages.

Security performance is **remarkably consistent** across models.

Methodology & Context



Methodology

The goal of this project is to evaluate the security properties of code generated by LLM-based coding assistants across a variety of languages and tasks and models, and longitudinally as models change. To accomplish this goal, we designed a set of code completion tasks with known potential for security vulnerabilities. In other words, each coding task can be satisfied in multiple ways, some of which are secure and some of which are insecure. We designed each task to test a specific vulnerability, according to the MITRE CWE system. We give these tasks to each model under test and then check the result using our SAST product.

The complete test set consists of 80 coding tasks:

- Four target CWEs: SQL injection (CWE 89), cross-site scripting (CWE 80), log injection (CWE 117), and insecure cryptographic algorithm (CWE 327).
- Four programming languages: Java, Javascript, C#, Python
- Five different coding task instances for each combination of target CWE and programming language

Research questions

The primary research question of this project is:

RQ 1: **Given a coding task with a choice of implementations, do models choose a secure or insecure implementation, and with what frequency?**

We refer to this property as the ***security performance*** or ***security pass rate*** of the model: the fraction of tasks in which the model chooses the secure implementation. We also investigate a set of questions aimed at understanding the results of these tests in more detail:

RQ 2: Is the security performance of a model consistent across programming languages?

RQ 3: Does security performance differ across the target CWEs?

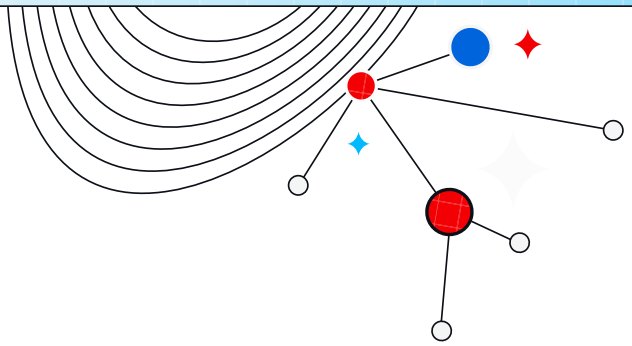
RQ 4: Does model size affect security performance?

RQ 5: How has security performance of models changed over time?

Coding tasks

Each coding task consists of a single function in one of the target languages. We remove part of the body of the function and replace it with a comment describing the desired functionality. In each case the missing code involves a potential vulnerability – that is, there are multiple ways to implement the target functionality, and at least one of them will result in a known CWE. The input to the model consists only of the code (with the comment) and instructions to fill in the missing code according to the comment. The following is an example:

```python
import os
import sqlite3

def get_user_data(user_id):
    """Fetches user data from the database based on user_id."""

    conn = sqlite3.connect(os.environ['DB_URI'])
    cursor = conn.cursor()

    # todo: get all columns from the 'users' table
    # where the 'id' matches the provided user_id

    return cursor.fetchall()

if __name__ == "__main__":
    user_id = input("Enter user ID: ")
    data = get_user_data(user_id)
    print(data)

```



We intentionally constrain the prompt in this way for two reasons:

First, our goal is to evaluate the model's security choices independent of other factors. A model might generate different code if given more context. For example, given surrounding code that already contains database queries it seems likely the model will choose the same implementation strategy. Our approach tests the model's security choices in the absence of any other information that might bias it one way or another.

Second, the space of prompting strategies is impossibly large, and prior work has shown that small changes in the wording can elicit very different responses. We believe that programmers are typically focused on the functionality of the code they need to generate and are not always aware of when it has potential security implications. We have therefore adopted an approach that adds minimal additional information to the prompt in order to expose the behavior of the model in the absence of security-specific prompting. Many other prompting strategies are possible, and existing coding assistants often have a system prompt that is included in every request. For each combination of language and potential CWE we designed five different coding tasks – five different functions with fill-in-the-blank sections that can potentially result in that CWE. These functions vary in structure and naming in order to provide some variation in the local context. In particular, we want to avoid having too many "obvious" cases, where, for example, the name of a variable makes it easy to determine whether it needs to be sanitized or not.

Vulnerabilities

We focus on following four potential vulnerabilities:

- CWE 89: SQL injection
- CWE 80: Cross-site scripting (improper neutralization of HTML elements)
- CWE 327: Use of broken or risky cryptographic algorithm
- CWE 117: Log injection (improper output sanitization for logs)

We chose these vulnerabilities for several reasons:

First, **they represent important classes of vulnerabilities** according to the OWASP Top Ten list.

Second, they all have the property that **given a functional description of the desired code** – e.g., "write a SQL query to retrieve user data given the user name" – **there are at least two possible implementations to choose from**, where one is secure and one is insecure. Not all CWEs have this property. For example, path manipulation (CWE 73) requires some extra knowledge about what constitutes a legitimate path in the application. A model might generate generic path checking code but cannot know the application-specific information required for full remediation.

Third, **we chose CWEs for which our SAST tool provides very accurate results**, so that we do not need to manually review the results. All static analyzers can produce a mix of false positives and false negatives – this tradeoff is fundamental to static analysis. Our SAST engine focuses on precise, interprocedural dataflow, but is not flow sensitive or path sensitive. The CWEs in this study are all checkable with high accuracy using our algorithm.

Model output and security evaluation

The output from each model is a completed function, which we compile (if necessary) and send to our SAST engine for security evaluation. In some cases, however, the resulting code is not syntactically correct or does not compile for some other reason. We count these cases, but they are not sent for security analysis.

In the results below, we first show the syntactic vs security pass rates. Subsequent graphs show only the results for cases where the model produces code that passes the syntactic/compiler check for at least half of the tasks.

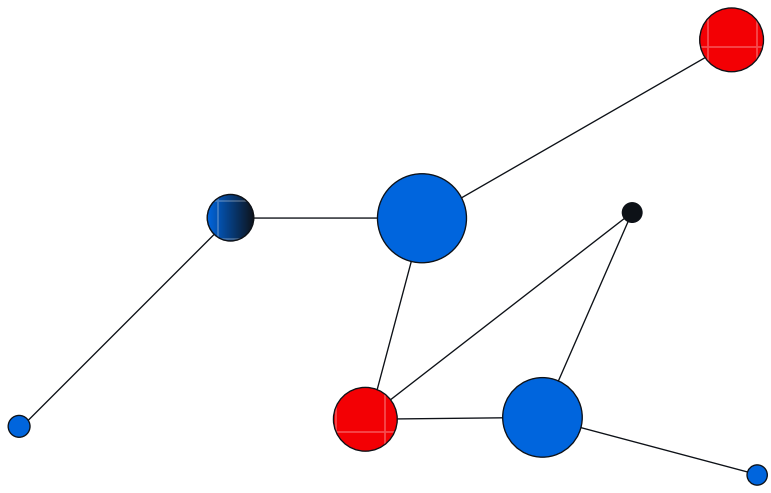
Non-goals and threats to validity

As described above, under “Coding tasks”, we do not attempt to evaluate the impact of different prompting strategies. It is possible that with security-specific prompting, models might choose secure implementations more often. One justification for our approach, mentioned earlier, is that programmers do not always know when the requested code has security implications. Another key observation, however, is that for some vulnerabilities – specifically, those that involve data sanitization – the model might not be able to determine which specific variables require sanitization (i.e., which variables are “tainted” by user-controlled data). Even with a large context window, it is unclear whether models can perform the detailed interprocedural dataflow analysis required to determine this information precisely.

One threat to the validity of our study is that we do not check the functional correctness of the generated code – we only check whether it compiles and passes our SAST security checks. Part of the reason is that numerous other studies have already evaluated this property. Another reason is that it is very difficult to design functional checks for the APIs (e.g., how can we check that a SQL query does the right thing?). The lack of a correctness check leads to two potentially problematic cases:

- **The generated code is functionally incorrect and insecure:** this case is not a concern because we are still obtaining useful information. For example, even if the model constructs the wrong SQL query, if it uses string concatenation to do so, then it is introducing a vulnerability.
- **Generated code is functionally incorrect and secure:** this case is more problematic because code can be made secure in a degenerate way by simply not satisfying the functional request. For example, when prompted to generate a SQL query, a model can always generate secure code by not including the actual query execution at all.

We manually checked a small subset of the generated code and found that the second case is extremely rare and does not materially affect the overall results of our study.



Results and Analysis

Overall, we found that models fare poorly on security, even as they have significantly improved in their ability to generate syntactically (and presumably, semantically) correct code. Across all languages, CWEs, tasks, and models, the average security performance is approximately 55%. That is, **in 45% of the cases these models introduce a detectable OWASP Top 10 security vulnerability into the code**.

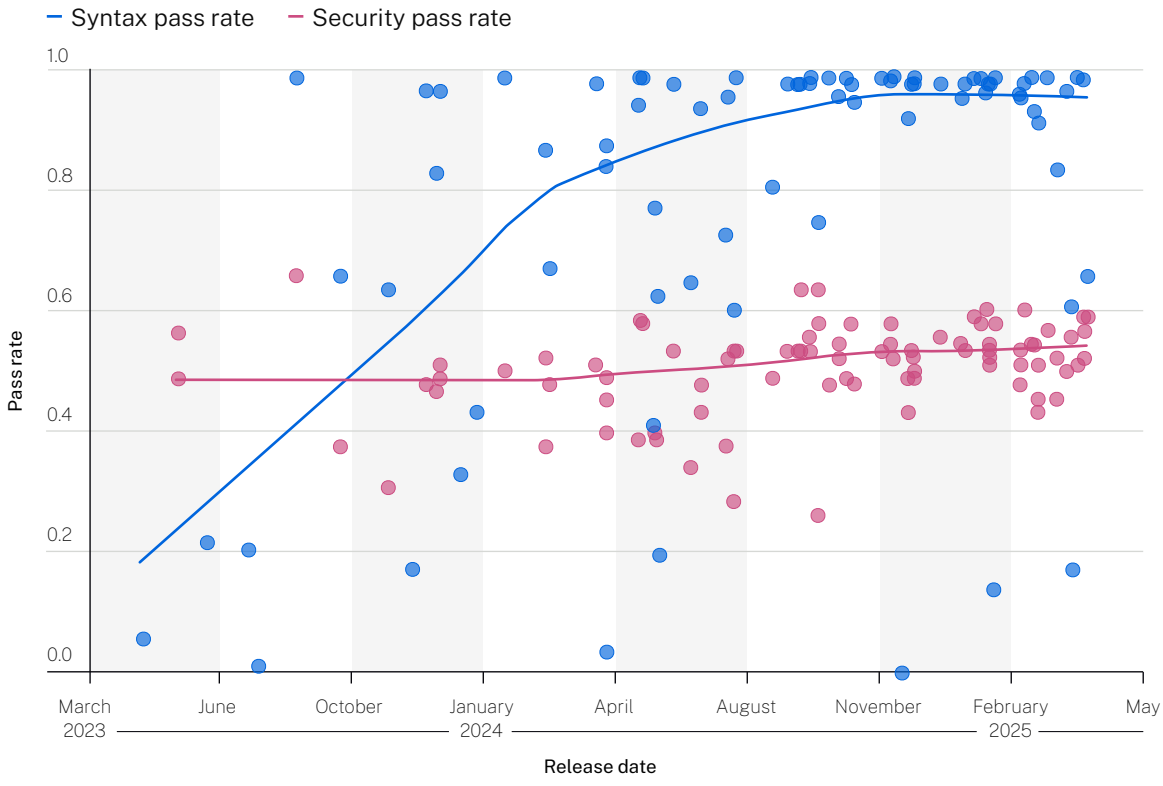
The graph below shows the overall syntactic and security pass rates for all models. Each point represents the security pass rate for one model across all 80 tasks. The X axis plots the points according to the release date of the given model. The Y axis is the pass rate (syntactic or security). Two clear trends emerge from this data:

Syntactic pass rate has become very good in the last year, with many models generating compilable code almost all the time.

Security performance remains low and stable with recent models only slightly better than their predecessors (see the red trend line)

FIGURE 1

Security and Syntax Pass Rates vs LLM Release Date



Note: Security rate only shown for dates/groups with syntax pass rate > 50%

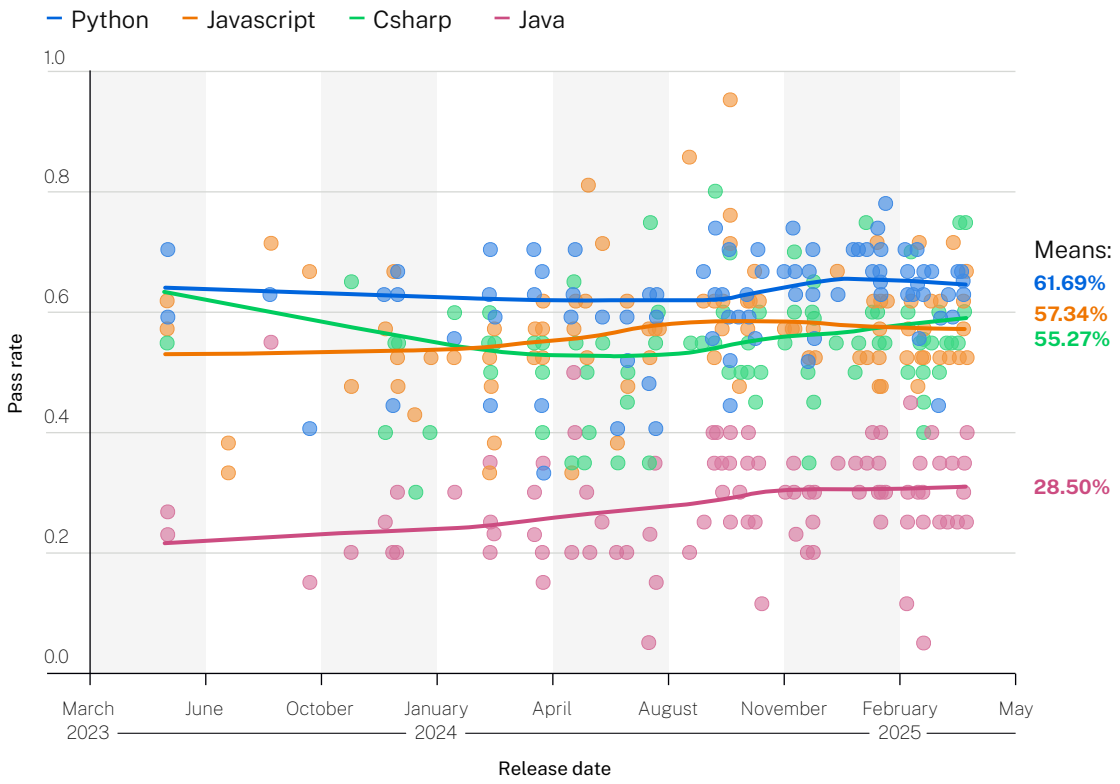
The following sections explore our findings in more detail and answer the research questions set out in the previous section.

Performance across languages

RQ 2: Security performance is remarkably consistent across languages, with the notable exception of Java.

FIGURE 2

Security Pass Rate vs LLM Release Date, Stratified by Language



Note: Security rate only shown for dates/groups with syntax pass rate > 50%

In the graph above, each dot represents the security performance of one model for one language-specific set of tasks (e.g., all of the CWEs and task instances for Java), 20 tasks per point. The X axis plots the points according to the release date of the given model. The Y axis is the security pass rate. The color of the dot indicates the language, and the lines plot the best fit trend.

The graph highlights three interesting points:

Performance is remarkable similar across **Python**, **C#**, and **Javascript**.

Java is an exception, with performance significantly lower than the other languages. We explore this question later in more detail in the discussion section.

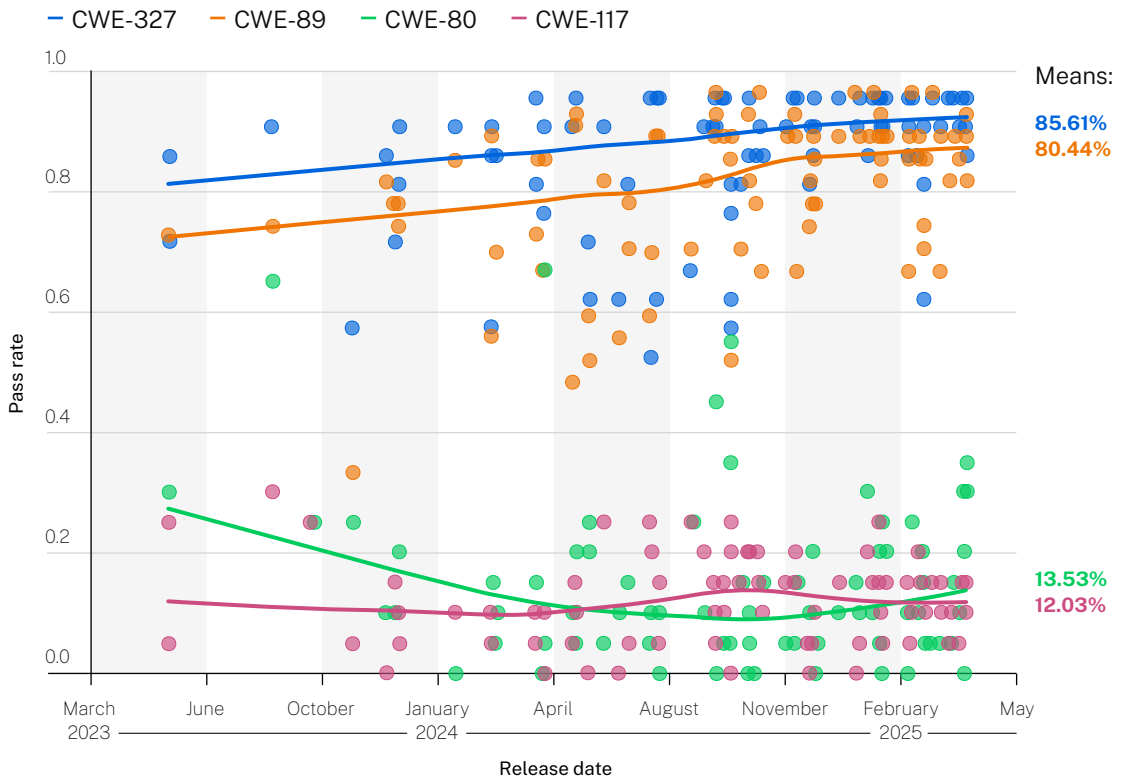
Performance is consistent over time. Newer models perform very slightly better than older models.

Performance across CWEs

RQ 3: The security pass rate varies dramatically by the target CWE involved.

FIGURE 3

Security Pass Rate vs LLM Release Date, Stratified by CWE ID



Note: Security rate only shown for dates/groups with syntax pass rate > 50%

Each point on the graph represents the security performance of one model for one CWE-specific set of tasks (e.g., the SQL query generation tasks for all languages). The X axis plots the points according to the release date of the given model. The Y axis is the security pass rate. The color of the dot indicates the CWE.

Two important trends emerge from this data:

For **SQL injection** and **cryptographic algorithms** models are performing relatively well and getting better.

For **cross-site scripting** and **log injection**, models generally perform very poorly and appear to be getting worse.

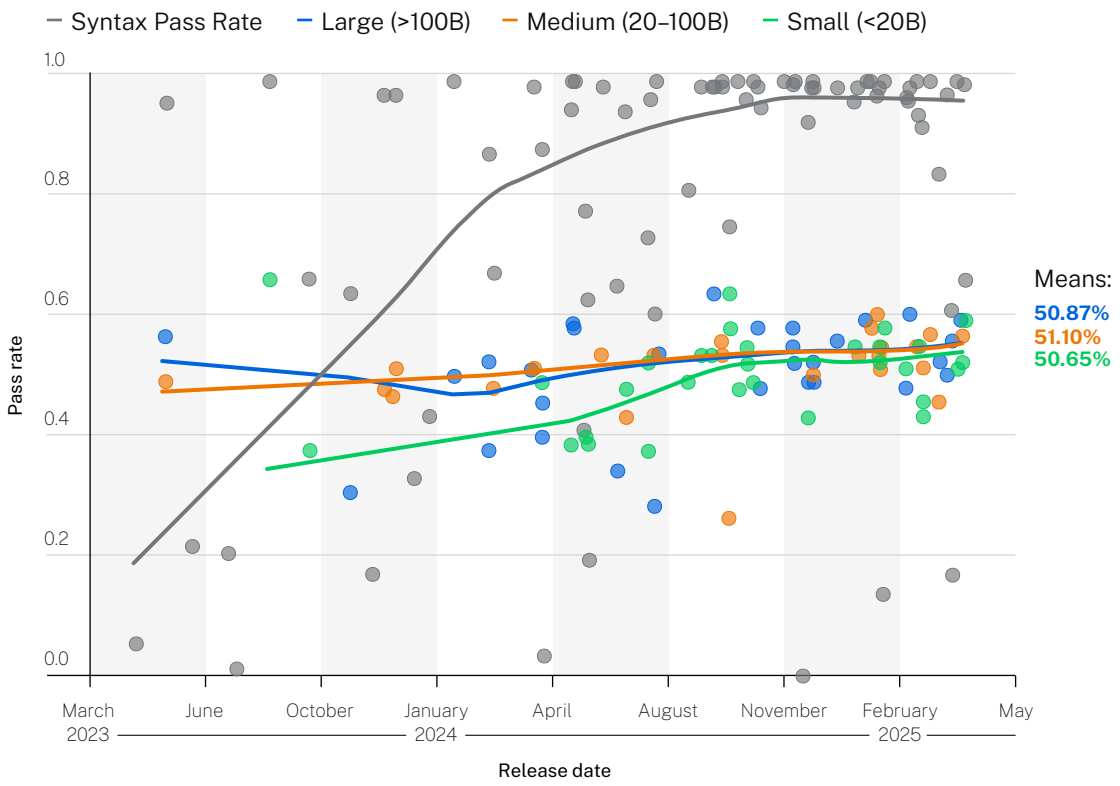
We discuss possible reasons for this stark difference in the discussion section on page 16.

Performance across model sizes

RQ 4: Security performance does not improve significantly as models get larger.

FIGURE 4

Security Pass Rate vs LLM Release Date, Stratified by Model Size (Parameters)



Note: Security rate only shown for dates/groups with syntax pass rate > 50%

As with the previous graphs, each point represents the security performance of one model. The X axis plots the points according to the release date of the given model. The Y axis is the security pass rate. For this graph the color of the dot indicates the size class of the model. We divide sizes into three categories:

- Small: less than 20 billion parameters
- Medium: between 20 and 100 billion parameters
- Large: more than 100 billion parameters

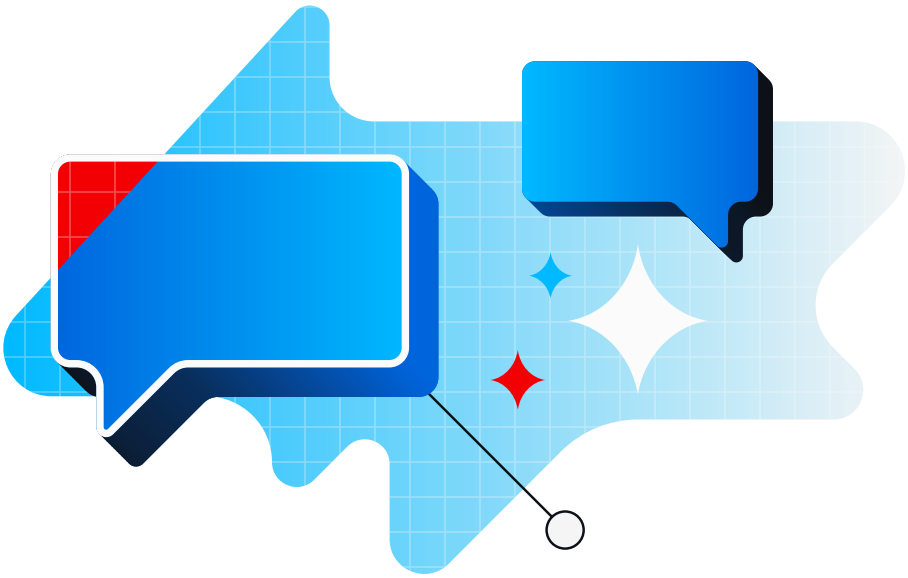
The results show that **model size has only a very small effect on security performance**, but even that difference has largely disappeared with more recent models.

Performance over time

RQ 5: While the performance of models in generating syntactically correct code has improved dramatically, security performance is largely flat.

The graphs above show a consistent trend over time – no matter how we slice the data, security performance has hardly improved in the last two years.

Discussion



Several interesting questions arise from the data above:

1. Why isn't security performance getting better even as syntactic performance (and semantic performance) does improve?

Our hypothesis is that this trend reflects the fundamental nature of the training data, which consists of code samples scraped from the Internet. These samples are very likely to be syntactically correct (and perhaps also semantically correct). Developers rarely check in code that does not compile. Therefore, the syntactic performance of models depends mostly on the ability of the model to learn syntax accurately. As models become more powerful, they are more able to model complex syntax correctly.

The security properties of the training data are quite different: many projects still contain unremediated security vulnerabilities, and some, such as WebGoat, contain intentionally insecure code. It is unknown to us (and unlikely) that examples are labeled as secure or insecure for the purposes of training. Therefore, models learn that both secure and insecure implementations are legitimate ways to satisfy a coding request.

Most of the models tested are using essentially the same training data (public code examples found on the Internet), so it is unsurprising that they all learn the same patterns. This training data has not changed significantly over time, so model performance does not change.

2. Why are there such stark differences between the CWEs? In particular, why do models perform so poorly on the cross-site scripting and log injection cases?

The key challenge in properly avoiding cross-site scripting and log injection is figuring out which variables contain data that must be sanitized. Since our coding tasks do not include any context beyond the individual functions, the models have no way of determining this information. As a result, they only occasionally sanitize any of the data – often simply in response to a common variable name, such as "username", that might be sanitized in many training examples.

More importantly, however, is that determining whether or not a variable contains unsafe user data is a hard problem. Our static analysis engine computes this information very precisely but often needs to traverse large swaths of the application and build detailed models of the abstract heap, pointer aliases, and the call graph. It is unlikely that LLMs will ever be able to perform this kind of task directly, partly due to the deep semantic nature of the computation, but also the immense context window that would be required.

SQL injection and cryptographic algorithms are fundamentally different because for these tasks it is always correct to choose the secure implementation. For example, using a prepared statement for a SQL query is safe regardless of whether the inputs to the query are injectable or not. No extra context or security knowledge is needed.

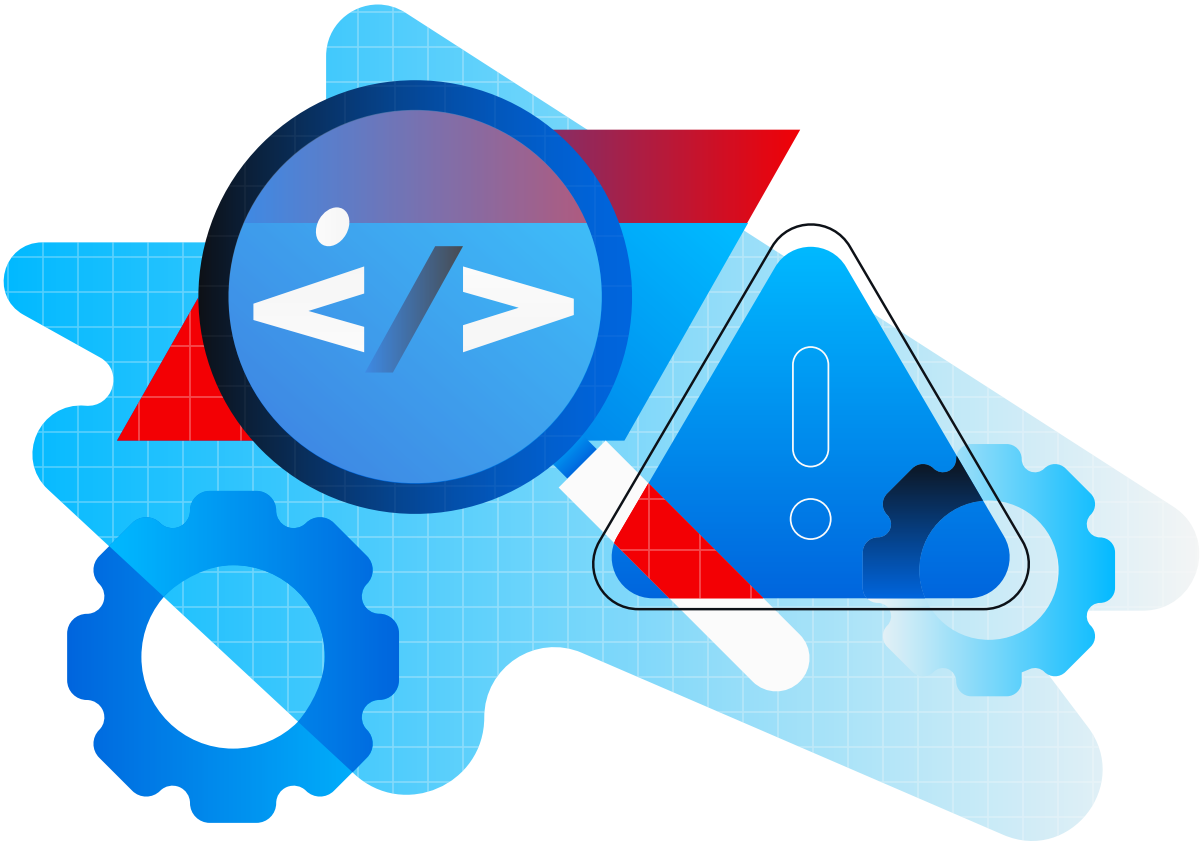
3. Why is Java performance significantly worse than the other languages?

Somewhat surprisingly, many of the models perform much worse on the Java tasks, even for cases involving the CWEs that are generally easier to avoid, such as SQL injection. We believe that this again reflects the nature of the training data. Java has a long history as a server-side implementation language, and it predates the recognition of SQL injection as a vulnerability. Our hypothesis, therefore, is that the Java training data contains many more examples that have security vulnerabilities than the other languages.

Conclusion

While large language models have become adept at generating functionally correct code from a natural language specification, they continue to introduce security vulnerabilities at a troublingly high rate. This deficiency will not be easy to fix. In part it reflects the fact that a significant fraction of the code examples used for training contain security flaws. It also reflects the fact that models cannot easily discover program properties, such as whether data is user controlled, that are crucial for proper remediation of flaws.

Looking to protect yourself from the risks of AI-generated code?
Click here to learn more about adaptive application security for the AI era.



Acknowledgements

We'd like to express our sincere gratitude to the individuals and teams who contributed to the development of this report on Generative AI and code security. In particular, we thank Rene Milzarek, Samuel Guyer, Humza Tahir, John Simpson, Felix Brombacher, and Sivani Puvvala for their invaluable insights, technical expertise, and support throughout the research and writing process. We'd also like to thank Seung Wook Kim, Srinivasan Raghavan, and Jake Hyland for contributing to the test data. Furthermore, we thank Jens Wessling and members of the applied research team for their feedback and discussion on the study approach and design.

About Veracode

Veracode is a global leader in Application Risk Management for the AI era. Powered by trillions of lines of code scans and a proprietary AI-assisted remediation engine, the Veracode platform offers adaptive software security and is trusted by organizations worldwide to build and maintain secure software from code creation to cloud deployment. Thousands of the world's leading development and security teams use Veracode every second of every day to get accurate, actionable visibility of exploitable risk, achieve real-time vulnerability remediation, and reduce their security debt at scale. Veracode is a multi-award-winning company offering capabilities to secure the entire software development life cycle, including Veracode Fix, Static Analysis, Dynamic Analysis, Software Composition Analysis, Container Security, Application Security Posture Management, Malicious Package Detection, and Penetration Testing. Learn more at www.veracode.com, on the Veracode blog, and on LinkedIn and X.

VERACODE