We Have a Package for You! A Comprehensive Analysis of Package Hallucinations by Code Generating LLMs

Joseph Spracklen*, Raveen Wijewickrama*, A H M Nazmus Sakib*, Anindya Maiti†, Murtuza Jadliwala*
*University of Texas at San Antonio
{joseph.spracklen, raveen.wijewickrama, ahmnazmus.sakib, murtuza.jadliwala}@utsa.edu
†University of Oklahoma
am@ou.edu

Abstract—**The reliance of popular programming languages such as Python and JavaScript on centralized package repositories and open-source software, combined with the emergence of code-generating Large Language Models (LLMs), has created a new type of threat to the software supply chain: *package hallucinations*. These hallucinations, which arise from fact-conflicting errors when generating code using LLMs, represent a novel form of package confusion attack that poses a critical threat to the integrity of the software supply chain. This paper conducts a rigorous and comprehensive evaluation of package hallucinations across different programming languages, settings, and parameters, exploring how different configurations of LLMs affect the likelihood of generating erroneous package recommendations and identifying the root causes of this phenomena. Using 16 different popular code generation models, across two programming languages and two unique prompt datasets, we collect 576,000 code samples which we analyze for package hallucinations. Our findings reveal that 19.7% of generated packages across all the tested LLMs are hallucinated, including a staggering 205,474 unique examples of hallucinated package names, further underscoring the severity and pervasiveness of this threat. We also implemented and evaluated mitigation strategies based on Retrieval Augmented Generation (RAG), self-detected feedback, and supervised fine-tuning. These techniques demonstrably reduced package hallucinations, with hallucination rates for one model dropping below 3%. While the mitigation efforts were effective in reducing hallucination rates, our study reveals that package hallucinations are a systemic and persistent phenomenon that pose a significant challenge for code generating LLMs.**

1. Introduction

Modern generative AI models are large deep learning systems, pre-trained on extensive datasets, enabling them to learn the underlying distribution and produce novel outputs. Today, a variety of commercial and open-source generative models are available, capable of producing synthetic images (e.g., Stable Diffusion [70], DALL-E [68], MidJourney [54]), videos (e.g., OpenAI Sora [63], Microsoft VASA-1 [87]), audio (e.g., AudioGen [33]) and text/conversations (e.g., ChatGPT [5], BlenderBot [72]) through easy-to-use interfaces and textual commands. These powerful models have enabled a range of new applications such as creative content generation, personalized virtual assistants, sophisticated chatbots, and coding assistants.

A particular class of generative models, known as Large Language Models (or LLMs), are foundational text generation systems capable of understanding and creating natural language and other types of textual content, and can be trained to perform a wide range of specialized tasks such as answering questions, summarizing documents, translating languages, and completing sentences. Some popular examples of such foundational LLMs include GPT-4 [1], Gemini [76], LlaMA [80], and Mistral [29]. One such emergent use of LLMs is the ability to generate computer code, accomplished by training or fine-tuning on vast programming-related datasets including code repositories, technical forums, coding platforms, documentation, and web data relevant to programming. Both commercial/black-box (e.g., GPT-4 [1], Claude [3]) and open-source (e.g., CodeLlama [71], DeepSeek Coder [19]) varieties of such code generation LLMs are readily available, and are being extensively used by both novice and expert programmers in their coding workflows to increase productivity. Studies indicate that up to 97% of developers are using generative AI to some degree and that approximately 30% of code written today is AI-generated, reflecting significant perceived gains in efficiency and convenience [73], [43].

Despite their tremendous success in solving complex language-related tasks, LLMs have many shortcomings, with a phenomenon called *hallucinations* being a particularly serious issue. Hallucinations are outputs produced by LLMs that may appear to be plausible or truthful, but in reality are either factually incorrect, overestimate and distort the true facts, are nonsensical, or completely unrelated to the input task. Hallucinations present a critical challenge to the effective and safe deployment of LLMs in public facing applications due to their potential to generate inaccurate or misleading information. As a result, there has been increased attention in the research literature on the topics of detection and mitigation of hallucinations in LLMs [22], [28]. However, most of the existing research efforts have focused only on hallucinations in classical natural language generation and prediction tasks such as machine translation, summarization, and conversational AI [24], [53], [40], [10]. The occurrence and impact of hallucinations during code generation, particularly regarding the type of hallucinated content and its implications for code security, remain relatively unexplored in the research literature. Hallucinations occurring during LLM-assisted code generation could result in code snippets that conflict with the user's requirements, the contextual information provided at input time, or the code knowledge base itself. Such hallucinations and their effects, if not appropriately mitigated during generation or time-of-use, could seriously undermine the correctness, security, and performance of the developed software. Recently, Liu et al. [46] has shown that popular LLMs (e.g., ChatGPT, CodeRL and CodeGen) significantly hallucinate during code generation and have established a comprehensive taxonomy of hallucinations in LLM-generated code.

In this work, we are concerned with a very specific type of hallucination during code generation called *package hallucination*. Package hallucination occurs when an LLM generates code that either recommends or contains a reference to a package/library which does not actually exist. Package hallucinations, especially if they are repeated, can have significant implications on the security of the generated code, with one severe and emergent threat being a *package confusion attack*. In a package confusion attack, an adversary takes advantage of package hallucinations in the LLM-generated code by actually publishing a package with the same name as the hallucinated or fictitious package and containing some malicious code/functionality. Then, as other unsuspecting and trusting LLM users are subsequently recommended the same fictitious package in their generated code, they end up downloading the adversary-created malicious package, resulting in a successful compromise. From that point on, any other software or code that utilizes or is dependent on this code containing the hallucinated package, is automatically also dependent on the adversary-created malicious package, thus becoming part of an entire codebase or software dependency chain.

Package confusion attacks, through techniques such as typosquatting (i.e., creating packages with names similar to popular ones to deceive users) and name similarity, have been a longstanding issue in the open-source software community. Package hallucinations by code-generating LLMs threaten to exacerbate the problem by exposing an additional threat surface for such attacks [58], [74], [34]. Trivial cross-referencing based hallucinated package detection techniques (i.e., comparing the package name with a list of known legitimate packages) are practically infeasible due to the large, dynamic, and open-source nature of popular package repositories. Additionally, an adversary may have already created a package corresponding to a hallucinated package making cross-referencing ineffective. Despite recent research [46] showing that LLMs are prone to package hallucinations, the extent to which this phenomenon occurs in current state-of-the-art (SOTA) publicly-available commercial (black-box) and open-source LLMs, the nature of these hallucinations, their impact on the security of the generated code, and the effectiveness of potential mitigation measures have not been comprehensively investigated.

In this paper we conduct the first systematic study of the frequency and nature of package hallucinations across a variety of code generation LLMs operating under a diverse set of model settings and parameters. We rigorously analyze and quantify the impact of these hallucinations on the security of the generated code (vis-à-vis package confusion attacks) and evaluate several mitigation strategies. We specifically make the following novel contributions:

- **Quantifying the incidence and origins of package hallucinations in code generating LLMs**: We comprehensively analyze the prevalence of package hallucinations in Python and JavaScript code generated by popular publicly-available commercial and open-source LLMs. We also examine the common behaviors in LLMs that lead to package hallucinations, including hallucination repetition, output verbosity, and the ability of models to detect their own hallucinations.
- **Analyze the effect of model settings and training data on package hallucinations:** We further study how specific model settings, such as training data recency, model temperature, and decoding strategies affect the occurrence and nature of package hallucinations.
- **Characterizing the generated hallucinated packages**: We carefully study several key properties of the hallucinated packages, such as their semantic similarity to popular packages, the occurrence of package recommendations from other programming languages, package persistence, and the significance of packages that have been recently removed from source repositories.
- **Mitigation Strategies**: We propose and comprehensively evaluate several techniques to effectively mitigate package hallucinations in LLM-generated code.

2. Background and Related Work

In this section we provide a brief background on code generation LLMs, the issue of hallucinations in LLMs, and its impact on open-source software security.

2.1. Large Language Models (LLMs)

LLMs and Automated Code Generation using LLMs. The advent of extremely versatile and high-performing attention-based transformer models [81] have catalyzed the emergence of *foundational language models*, which are large-scale models described by billions of parameters and pre-trained on vast datasets. These foundational LLMs can be tailored to perform a wide range of downstream tasks without the need for task-specific architecture modifications [4]. Models such as BERT (Bidirectional Encoder Representations from Transformers) [11], CLIP (Contrastive Language-Image Pretraining) [67], and GPT (Generative Pre-Trained Transformer) [5] are examples of foundational language models that have emerged as new benchmarks for a wide range of tasks.

The primary method to prepare these foundational models for a specific downstream application is *fine-tuning*, which involves providing the base model with a smaller, task-specific dataset. The model trains on the new data, making slight adjustments to its internal parameter weights in order to optimize its performance for the new application. Nearly all SOTA models in natural language processing (NLP) and computer vision are derivatives of such large foundational language models, excelling in tasks such as sentiment analysis, object recognition, image captioning, and information extraction [4]. Foundational models have also been effectively adapted to domain-specific applications across various fields, most notably healthcare, law, and education [56], [50], [7], [25].

One such emergent application of fine-tuned foundational language models has been *code generation*. Commercial products such as Codex [9] and GPT-4 [1] from Open AI and open-source models such as CodeLlama [71] and DeepSeek [19] are able to produce functional code snippets, debug existing code, and translate code between programming languages. These tools have rapidly gained popularity and earned trust amongst developers, with 97% of DevOps and SecOps programmers having integrated such generative AI models into their workflows [73].

The quality of code generated by these tools has quickly increased, as success rates for correctly answering coding prompts have skyrocketed from a mere 25% in June 2021 to 96% by April 2024 [9]. As code generation models become more ubiquitous in software development, there is a growing concern about the potential for generating insecure or flawed code that could lead to vulnerabilities in deployed applications. Early versions of code generation tools were shown to write code that contained a vulnerability in the MITRE Top-25 Common Weakness Enumeration (CWE) list 40% of the time [64]. More recent research has shown that AI-assisted programming not only results in less secure code but also instills a false sense of security [65].

Hallucinations by LLMs. It has been well-documented that LLMs can unintentionally produce harmful information [48], [85], be manipulated for malicious purposes [20], [31], expose private information [41], and often carry inherent biases in their training data [15]. A related phenomenon is the notion of hallucinations, which are instances where LLMs generate misleading or entirely fictitious information. These errors arise in various forms: the model might misinterpret the intended input (*input-conflicting hallucination*), produce inconsistencies with prior outputs (*context-conflicting hallucination*), or contradict established facts (*fact-conflicting hallucination*) [89]. Hallucinations can stem from three main root causes: (i) *data*, (ii) *training*, and (iii) *inference* [23]. Hallucinations from data occur when the data source itself is flawed with misinformation [44], bias [15], or incomplete records [61]. Architecture flaws [45] or sub-optimal training objectives [83] during training could also result in downstream hallucinations, while inference time issues such as defective coding strategies [21] and imperfect decoding representations [8], [49] also contribute to hallucinations.

Hallucinations can also be categorized based on whether they can be directly verified from the source content; if so, it is termed an *intrinsic hallucination*, otherwise, it is considered an *extrinsic hallucination* [26]. The structured nature of code leads to intrinsic hallucinations that are directly traceable to syntactic errors, while extrinsic hallucinations arise from complex interactions or gaps in the model's training data [46]. Code generation hallucinations manifest in several ways, including functional bugs that impair the intended operation, code that performs the wrong task, dead code that never gets executed, and, perhaps most critically, security vulnerabilities that can be exploited.

The persistent issue of hallucinations in LLMs has spurred extensive research into various mitigation strategies, broadly categorized into *prompt engineering* and *model architecture enhancements* [79]. Prompt engineering techniques such as Retrieval Augmented Generation (RAG) [39] and self-refinement methods [52] aim to refine the input provided to the model to produce more accurate outputs. Alternatively, developing more robust models involves approaches such as supervised fine-tuning [77], inference time intervention [42], and incorporating knowledge graphs [27] to improve model understanding and reduce errors/hallucinations in the model output. The manipulation of the *temperature* parameter within each model has also been shown to significantly influence the prevalence of hallucinations in LLM output [2]. The temperature parameter modulates the probability distribution over potential output tokens [55]. Lower temperatures produce less random, more predictable outputs, while higher temperatures increase the likelihood of sampling low-frequency tokens, raising the risk of hallucinations [12].

The conflict between the non-deterministic nature of LLMs and mitigating hallucinations is inherently challenging. This non-determinism fosters creativity and generates diverse, innovative content, making it a desired feature despite its role in producing hallucinations. Balancing creativity with accuracy remains a central challenge in the deployment of LLMs, underscoring the complexity of developing effective mitigation strategies.

2.2. Software Supply Chain Security

Open-Source Software (OSS) and Software Supply Chain. OSS is any software or application with source code that anyone can inspect, modify, and enhance. Reliance on OSS, in particular publicly-available packages and libraries for software development, has been steadily increasing. The Linux foundation estimates that 98% of the current codebases include some form of OSS [57], and this dependence is only expected to increase during the next year [69]. Many programming languages, including four of the top-5 most popular on GitHub, rely on centralized package repositories to manage package distribution and dependency management. Services such as npm [59] and PyPI [13] are the two most popular examples of package repositories, serving as the primary package ecosystem for Python and JavaScript, respectively.

OSS Security. The convenience and simplicity afforded by the public nature of open-source software repositories also makes them an ideal platform for malware distribution by malicious actors. A total of 245,000 malicious packages were found on open-source repositories in 2023, a 300% increase from 2022 and 2× the amount of all previous years combined [73], another trend that is predicted to continue for the foreseeable future [16]. Existing OSS security vulnerabilities can be broadly categorized as either *insecure* code or *malicious* code. Insecure code is the result of poor coding practices or unmaintained legacy code that allows malicious actors to exploit an open vulnerability. Repository statistics show that one in eight open-source packages have security risks and 18% of available packages are unmaintained [73]. On the other hand, malicious packages or libraries contain intentionally harmful functionality, which research has shown allows attackers to execute arbitrary code on victim machines once these packages are downloaded [60].

Package Confusion Attacks. Many packages/libraries rely on other packages to function, thus creating huge *dependency trees* which further compounds the problem of OSS security [90]. For an adversary to infect a particular software product or ecosystem, they only need to infect a single package in the dependency chain [58], [32]. Code generating LLMs only further exacerbate this problem by generating insecure code, which is typically not scrutinized by users due to the high level of trust in the underlying model [65]. Further compounding the issue is the potential inclusion of this insecure code in the dependency chain of other packages, leading to a cascading effect where vulnerabilities are propagated across numerous codebases. There have been many well-documented OSS supply chain attacks on both public and private package repositories and dependency chains, including the notable SolarWinds compromise [6]. Public OSS repositories such as PyPI and npm have implemented several measures to raise the barrier for malicious packages to be distributed on their platforms, such as two-factor authentication, namespace protection, and software signing [82], [88]. It is currently not known if the repositories utilize scanning methods for malicious code, but they often do not release the full list of packages that have been removed.

Once a malicious package has been created and uploaded to a software repository, adversaries have devised various techniques to trick users into downloading the package, which is subsequently incorporated into their codebase and dependency chains. At a high level, these attacks work by deliberately naming malicious packages to mimic or appear as legitimate packages, and are collectively referred to as *package confusion attacks* [58]. Package confusion attacks can be broadly categorized into *typosquatting*, *combosquatting*, *brandjacking*, and *similarity* attacks [35], and are distinct from other types of software supply chain attacks such as corrupting legitimate packages or developing unique malicious packages from scratch as part of a long-term campaign. There have been over 1,200 documented package confusion attacks in the last six years [58], including an incident involving the popular PyTorch package [74] and a campaign by the North Korean Advanced Persistent Threat

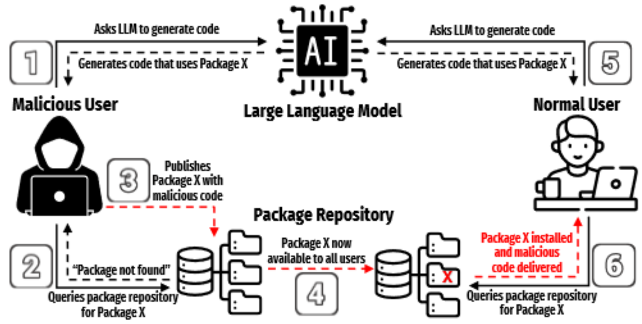


Figure 1: Exploiting Package Hallucination.

Lazarus Group [34]. At one point the PyPI repository had to temporarily suspend package uploads and new account creation due to the number of malicious packages being uploaded [13]. Code-generating LLMs and the hallucinated packages recommended by them significantly amplify these existing challenges and risks associated with package confusion attacks.

2.3. Related Work

The idea of code generation models recommending malicious or typosquatted packages was first suggested in 2021 as tools such as GPT-3 and Codex were released as viable code generation platforms [5]. At the time, the risk of code generation tools suggesting vulnerable, malicious, or typosquatted packages was assessed to be low [9]. Notably, the related but distinct concept of package hallucinations was not explicitly considered in this initial risk assessment; either because such an attack had not been considered or the threat was thought to be negligible. The capabilities of generative AI agents have advanced significantly since that introductory evaluation. Although a thorough study of package hallucinations has not been previously conducted, a blog post by Lanyado in June 2023 [37], and subsequent follow-up post in March 2024 [36], present the first preliminary findings on the threat of package hallucination attacks. Their first study reports that package hallucinations occur in 20% of JavaScript responses and 35% of Python responses using ChatGPT 3.5. In the follow-up work, they found that 24.2% of package recommendations were hallucinated using GPT-4 and 22.2% using GPT-3.5. Two other commercial models were tested, Google Gemini and Cohere, which reported 64.5% and 29.1% hallucination rates, respectively. No open-source models were tested. This study also contained a proof of concept for executing a package hallucination attack by actually publishing a package (with dummy code) using the name of a hallucinated package that was generated during their testing. In just three months, the dummy package received over 30,000 downloads on PyPI, establishing that the attack vector exposed by LLM-generated package hallucinations is absolutely viable [36].

Compared to these previous works which primarily focused on identifying the prevalence of hallucinations and providing high-level recommendations, our work aims to significantly advance the understanding of package hallucinations in LLM-generated code. We conduct a rigorous and comprehensive evaluation across a broader range of models (including the first analysis of package hallucination in open-source models of any kind) at a scale and level of detail that has not been previously documented. To this end, we provide thorough testing with a larger custom dataset covering two programming languages (Python and JavaScript) followed by analysis into the root causes and defining characteristics of this phenomenon, including the first study on the effects of temperature, decoding strategies, deleted packages, stale training data, semantic similarity, model verbosity, and language confusion on package hallucinations. We then propose potential mitigation strategies to curb hallucination attacks, the first such work with recommendations to limit hallucinations in code generation LLMs. Lastly, we contribute a novel dataset for code generation that can be used to test LLMs across a broad range of coding tasks.

3. Research Questions

Adversary Model and Assumptions. In this work, we assume an adversary that wants to accomplish a package confusion attack by leveraging package hallucinations in the code generated by publicly-available commercial/closed-source (e.g., ChatGPT [5]) and open-source code generation LLMs (e.g., CodeLlama [71]). The target of the adversary are users who employ such LLMs for generating code snippets or entire programs, which they directly use in their codebase without any significant verification or modification. In other words, the target users fully trust the LLMs to include only valid package names in the generated code. We assume that the adversary has access to the same set of LLMs for code generation as the target users, and is unable to modify or manipulate the model and model parameters of these LLMs (e.g., via re-training or fine-tuning) before they are used by the victims. The adversary is able to determine a list of hallucinated packages in the code generated by these LLMs (for example, by cross-referencing on the package repository), and then to realize a package confusion attack the adversary is also able to create packages with the same names on the corresponding package repositories. These newly created packages corresponding to the fictitious packages generated by the LLM could contain malicious code or functionality to compromise the target users. Figure 1 provides a high-level overview of package hallucination exploitation process.

Research Questions. We now organize our investigation of package hallucinations by popular publicly-available LLMs into five broad Research Questions (RQ), as outlined below.

RQ1: ***How prevalent are package hallucinations while generating Python and JavaScript code using LLMs?*** By means of this RQ, our goal is to gain a comprehensive understanding of how often package hallucinations occur (i.e., rate of hallucinations) when using popular commercially available and open-source LLMs to generate Python and JavaScript code for a variety of different programming tasks.

RQ2: ***How are package hallucinations impacted by certain model settings?*** Here our goal is to comprehensively analyze how select model properties and settings, namely training data recency, model temperature, and decoding strategies impact the package hallucinations produced by the LLMs.

RQ3: ***What are the commonly observed model behaviors related to package hallucinations?*** This RQ will exhaustively study model behaviors such as hallucination repetition by a single LLM (hallucination persistence) and across multiple LLMs (cross-model hallucinations), output verbosity, and the ability of LLMs to detect their own hallucinations.

RQ4: ***What are some of the defining properties/attributes of the observed package hallucinations?*** The goal of this RQ is to analyze the properties of the hallucinated packages such as semantic similarity between hallucinated and popular packages, number of cross-language hallucinations (i.e. non-existent packages from the language requested but valid packages in another programming language), and the number of generated packages that were recently removed from the source repositories.

RQ5: ***Is it possible to effectively mitigate package hallucinations using best practices in the literature and knowledge gained from earlier results?*** Through this RQ, we will investigate if code generating LLMs can be designed to reduce hallucinations. In this direction, we will comprehensively study if techniques such as Retrieval Augmented Generation (RAG), self-detected feedback, decoding strategies, and supervised fine-tuning are effective hallucination reduction strategies.

4. Experiment Design

To answer the RQs outlined above, we design several experiments to first iteratively prompt LLMs to generate code and then analyze the generated code. Our experimentation pipeline comprises of three distinct phases: (i) *prompt dataset generation*, (ii) *code generation*, and (iii) *hallucination detection*, each of which is described below:

4.1. Prompt Dataset

The experiments are designed to exhaustively test each LLM through a complete range of potential coding tasks. As existing datasets of coding prompts are used as benchmarks for correctness [9], [47] and contain only a limited number of prompts that lack diversity, we decided to create a new code prompt dataset for our experiments. Our goal was to create a dataset that was both realistic and representative of the coding task requests observed in practice by everyday users. To accomplish this, we employed two distinct approaches, as described below.

Stack Overflow Dataset. In order to model the input prompts (to LLMs) in our experiments around real programmer questions, our first prompt dataset was created using

Stack Overflow [75] questions across relevant programming topics and subject areas. Stack Overflow is a popular online question-and-answer service for software programmers and developers. To capture a wide range of topics, we utilized the "tag" feature of Stack Overflow, which allows users to label posts according to a subject matter. We included any tag that had over 5,000 questions and was also relevant to Python or JavaScript (the two programming languages we focus in this work, as detailed in Section 4.2). For each of the 240 tags that met this criteria, we extracted the 20 most up-voted questions, resulting in 4,800 prompts (i.e., 4,800 prompts for Python and 4,800 prompts for JavaScript).

As more recent data is less likely to be included in the pre-training data of LLMs, we are also interested in investigating the correlation between data recency (i.e., how recently the question was asked on Stack Overflow) and model hallucination rate. To enable such an analysis, we ran two queries on Stack Overflow; one that only captured the most popular questions in the selected tags from 2023 and another that captured the most popular questions of all-time. By including the two different ranges of time, we effectively doubled the original number of prompts, for a total of 9,600 for each of the two languages. There is some overlap between the datasets for each language, as some of the tags pertain to general coding questions while others are language-specific. Also, not all questions asked on Stack Overflow may involve coding or require code to answer the question. For example, some questions might be about conceptual topics, such as software design principles or debugging strategies, which do not necessitate code snippets. Rather than attempting to filter out such prompts during the code generation phase, which is non-trivial and error-prone, the LLM is asked to answer the question and only provide code if necessary. In the end, this may result in a slightly lesser number of usable LLM-generated code samples, but is more realistic as LLMs are expected to accommodate imperfect user inputs.

LLM-generated Dataset. As a majority of the programming tasks require some library/package, our next idea was to use the package repositories themselves as a good representation of the full spectrum of coding topics. For creating our second prompt dataset, we take the 5,000 most popular (according to number of downloads) Python and JavaScript packages and scrape the official package description as listed on pypi.org and npm.org, respectively. These descriptions are then input individually to the Llama-2 70B model with the following instruction: *"Generate a prompt to produce Python/JavaScript code that would accomplish the same functionality as the following package description: [package description]"*. This process worked remarkably well and generated roughly 4,800 prompts for Python and JavaScript each, resulting in two datasets of approximately the same size. Similar to the Stack Overflow dataset, we doubled the LLM-generated dataset for recency analysis by dividing it into two segments: packages most downloaded in the past year and packages most downloaded in the period before the past year (ensuring no overlap). When a package ranks among the most downloaded for both the past year and all-time, we give preference to the latter. During this process some packages with no description or descriptions in a non-English language were discarded. Both the Stack Overflow and LLM-generated datasets will be made publicly available through the project's GitHub page. A truncated list from the LLM-generated dataset can be found in Appendix A.

4.2. Code Generation

Model Selection. For our experiments, we chose the top-ranked models from the EvalPlus leaderboard (as of January 2024) [47]. While creating our list, we disregarded the fine-tuned versions that were ranked below their corresponding foundational models, and only selected one fine-tuned version of the same foundational model of the same parameter size [47]. EvalPlus maintains a ranking of the top performing LLMs for code correctness according to a rigorous code synthesis evaluation framework. Our goal was to include a mix of top performing base models and a few of the best performing fine-tuned variants. We also included the GPT series of models (GPT-3.5, GPT-4, and GPT-4 Turbo) in our experiments, which currently hold the top rankings on the leaderboard. GPT models are widely considered as SOTA in terms of code generation models at the time of writing and add value to our experiments as representative commercial models. Table 1 provides a complete list of the models that we tested in our experiments.

TABLE 1: Details of the models that were evaluated.

Model	Parameters	License	Open Source
ChatGPT 4.0 [1]	Unknown	Commercial	✗
ChatGPT 4.0 Turbo [1]	Unknown	Commercial	✗
ChatGPT 3.5 Turbo [5]	Unknown	Commercial	✗
CodeLlama [71]	7B, 13B, 34B	Free	✗
DeepSeek [19]	1.3B, 6.7B, 33B	Free	✓
Magicoder [86]	6.7B	Free	✓
WizardCoder [51]	34B	Free	✓
Mistral [29]	7B	Free	✓
Mixtral [30]	8x7B	Free	✓
OpenChat [84]	7B	Free	✓
WizardCoder-Python [51]	7B	Free	✓
CodeLlama-Python [71]	33B	Free	✓

Language Selection. In our experiments, we focus on two of the most popular programming languages, JavaScript and Python, according to GitHub's 2023 Octoverse report [17]. The open-source package repositories for these languages, npm and PyPI, represent ecosystems of 4.5 million and 535 thousand packages, respectively [78]. Given the popularity of these two programming languages and the scale of these software ecosystems, they are more likely to be targets of adversaries and impacted by package hallucinations and related package confusion attacks. It is safe to assume that many open-source code generation models have been fine-tuned to program in these two specific languages, particularly Python due to its high popularity. Online services such as HuggingFace actively track the best coding models, specifically using Python and JavaScript as the two benchmark languages [47]. In total, we tested 16 different models (see Table 1), fourteen of which were tested for both Python and JavaScript, while two fine-tuned Python-specific models, WizardCoder-Python and CodeLlama-Python, were only tested for Python.

Testing Environment. To create a uniform and fair testing environment, we utilize the *text-generation-webui* tool for LLMs, which supports a majority of open-source LLMs [62]. This tool provides both a GUI and an API that allows a user to use LLMs locally and offline. It also provides fine-grained control of parameters during testing which ensures that each model is evaluated using identical environments. We opted to use quantized versions of the open-source LLM models, using the GPTQ quantization method. Quantization is a process that reduces the precision of model parameters, improving inference speed and reducing memory requirements without significantly impacting performance. The GPTQ quantization method, specifically, uses a one-shot weight quantization method based on approximate second-order information that has a negligible effect on the accuracy of models, making it an ideal choice [14]. Additionally, quantized models better simulate the performance that a typical user can expect when running models on commercial grade hardware, making them more accessible and practical for everyday use. For testing uniformity, we use the same parameters and quantization precision for all open-source models, which are summarized in Appendix C. Model testing was conducted in two distinct computing environments - a Debian environment with 40 nodes, each equipped with 40 CPU cores, 1TB of RAM, and NVIDIA A100 or V100 GPUs and a Ubuntu system with 80 CPU cores, 750 GB of RAM, and 3 NVIDIA RTX 6000 GPUs. To generate code for our analysis, we query each of the LLMs (Table 1) with prompts from the two datasets using the Python `requests` packages to interact with the text-generation-webui API. The message to the API contains a system message with specific instructions for the model and the prompt itself. An overview of the process, including system messages used during each step, are detailed in Appendix B. This results in a total of 19,200 code samples per model (16 Python tests + 14 JavaScript tests * 19,200 = 576,000 total code samples), which are further analyzed to determine which packages are required to execute the generated code.

4.3. Detection Methodology and Heuristics

Detection of hallucinated packages from LLM outputs or code samples is non-trivial. Before we outline our heuristics to detect packages in the generated Python and JavaScript code samples, let's first discuss the notion of packages and modules in modern interpreted languages. To enable code modularity and reusability, interpreted programming languages such as Python and JavaScript allow for entities called *modules*. A module is a chunk of code, often in an external file, that performs a specific task or function. By encapsulating related code into modules, developers can organize their programs more efficiently and make them easier to maintain and share. A *package* is a collection of related modules that work together to provide certain functionality. To use a particular module in their source code, a developer must install the appropriate package into its development environment by first downloading the package from an online package manager or repository such as PyPI or npm, if it is not locally available, and then import the desired module into the code using appropriate import functions, e.g., `import` in Python or `require` in JavaScript.

These module names do not necessarily need to match the package names and the namespace for modules is not protected, i.e., different packages may include modules of the same name. This discrepancy poses a significant challenge for detecting package dependencies from the raw Python and JavaScript code, as the `import`/`require` statements that are typically included in code samples for importing modules do not have a unique mapping to package names. Thus, it would not be accurate to simply parse the generated code for `import` or `require` statements, as those statements refer to module names and not package names. To solve this problem, we employ the following three heuristics to determine/identify package names:

Heuristic 1. As part of our first heuristic, we parse the generated Python and JavaScript code for "`pip install`" and "`npm install`" commands, respectively. These commands look for the specified package in the PyPI/npm repository, resolve its dependencies, and install everything in the current Python/JavaScript environment to ensure that future module requests will work. This is the most straightforward heuristic for detecting package names (and thus, hallucinations), as it involves explicit commands from the code generation model for package download/installation. This is significant because if the referenced hallucinated package was indeed used by an adversary to execute a package confusion attack, it could immediately trigger download/install of the malicious code in the package. Note that we did not directly ask the model to provide these commands, but allowed them to occur naturally during the generation process. As such, we observed that these instances ("`pip install`" and "`npm install`") occur for 7% of the total outputs.

Heuristic 2. For the second heuristic, each generated code sample is used as an input into the same model that generated it. The model is then asked (prompted) for a list of packages that would be required to run the given code. Our intuition is to mimic an actual user/developer who is using LLMs for code generation. If the user gets an error due to an uninstalled package when attempting to execute the generated code, they could query the model for the correct package to install. We wanted to replicate this intuitive process to identify package names required by the generated code.

Heuristic 3. As the third heuristic, we re-use the original prompt used to generate the code sample as an input to the model and ask the model to output package names that would be required to accomplish this coding task. Similar to the previous heuristic, this process of extracting package names simulates another approach users would take to obtain package names from the model that generated the code, if the required packages were not mentioned in explicit "`pip install`" and "`npm install`" commands. Once each model provides specific package names (through the three heuristics outlined above), we simply compare each package name to a master list of package names acquired from PyPI and npm, respectively (each list is as of 10 January, 2024). If a package name is not on the master list it is considered a hallucination. To measure LLMs' propensity to produce hallucinated packages during code generation we use the *package hallucination rate* metric, which can be expressed as a simple ratio of the number of hallucinated packages to the total number of recommended packages.

5. Evaluation Results

In this section, we present the results of our experimental analysis related to **RQ1** - **RQ4**. For RQs 2 through 4, we focus our analysis only on the Python programming language and a subset of the original models tested. Given the consistent results that we were able to obtain across both languages, this narrowed scope of discussion does not compromise the generalizability of the conclusions and allows for a deeper analysis of package hallucinations in a controlled setting. We selected GPT-4 Turbo, GPT-3.5, CodeLlama, and DeepSeek for in-depth analysis, representing the best-performing and most popular open-source models. The full set of hallucination results for all models, covering both Python and JavaScript, are presented in Appendix G.

5.1. Prevalence of Package Hallucinations (RQ1)

In our first experiment, our goal was to quantify the prevalence of package hallucinations across different models by generating and analyzing a large number of code samples. We conducted 30 tests (using 16 models for Python and 14 models for JavaScript, as outlined in Table 1) producing a combined 576,000 code samples using both the Stack Overflow and LLM-generated datasets (Section 4.1). Each code sample was evaluated for hallucinations according to the heuristics defined in Section 4.3, which include parsing the generated code and prompting the model for packages twice per code sample, for a total of 1,152,000 package prompts across all tests.

These 30 tests generated a total of 2.23 million packages in response to our prompts, of which **440,445 (19.7%) were determined to be hallucinations, including 205,474 unique non-existent packages** (i.e. packages that do not exist in PyPI or npm repositories and were distinct entries in the hallucination count, irrespective of their multiple occurrences). GPT-4 Turbo resulted in the lowest overall hallucination rate at 3.59%, while DeepSeek 1B had the best hallucination rate among the open-source models at 13.63%. Python code resulted in fewer hallucinations than JavaScript (15.8% on average compared to 21.3% for JavaScript), a result that is expected, given that the JavaScript ecosystem contains nearly 10x the amount of packages compared to

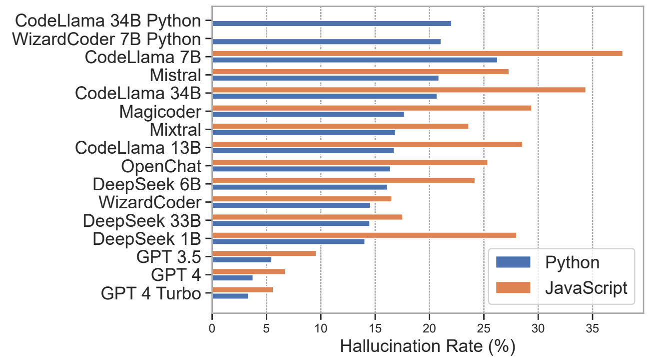


Figure 2: Observed hallucination rates of the tested models.

Python (npm contains 4.75 million while PyPI contains 548,000). Despite the difference in the hallucination rate between the two languages, there is a linear relationship between the results (as shown in Figure 11 in the appendix), which demonstrates that the propensity of a model to hallucinate is positively correlated between programming languages. The total hallucination rates for each evaluated model are presented in Figure 2. For a comprehensive breakdown of the results for each language, please refer to Appendix G. These results provide strong evidence that package hallucinations are a significant issue across all code generating LLMs.

5.2. Impact of Model Settings (RQ2)

Recency of Source Material in Prompts. The models we tested were shown to be more sensitive to hallucinations when coding prompts that deal with more recent topics were used. The best performing model for this experiment was WizardCoder 33B, with a difference between the rate of recent prompts and all-time prompts of only 0.58%, with the more recent dataset resulting in a hallucination rate of 14.59% compared to 14.01% for the all-time dataset. Representing the commercial models, GPT-4 Turbo was a close second with a difference of 0.63% between recent and non-time constrained prompts. Note that a lower difference between the rates of recent and all-time prompts indicates better performance in handling questions that fall outside the model's pre-training data. Notably, WizardCoder, a fine-tuned version of the DeepSeek models, achieved a 50% lower difference in hallucination rates during the recency experiment than the DeepSeek models after recording similar results in the primary experiment. WizardCoder uses a novel fine-tuning approach, named Evol-Instruct, that uses existing open-source data to create additional datasets that are more complex, diverse, and realistic [51]. While the primary goal of these fine-tuning methods is to increase overall coding proficiency, the results, especially achieving better performance than the GPT models in recency testing, indicate this process has a secondary benefit in reducing hallucinations that is worthy of future work. Overall, all 16 Python models we evaluated **demonstrated a higher hallucination rate when being prompted about questions or packages that**

were popular within the past year (Figure 3). These higher rates are partly due to inherent limitations of LLMs. As noted in the OpenAI GPT-4 technical report [1], LLMs cannot update themselves with new information post-release and struggle with prompts beyond their training data cut-off date. This cut-off date is critical because models lack knowledge of events that occur after it. Although fine-tuning can enhance specific tasks, it does not generally improve the model's overall knowledge of the world. The extensive testing and fine-tuning required for modern foundational models create a significant lag between the training data cut-off and model release, widening the gap in knowledge of recent information. The massive cost of training LLMs from scratch, indicated by the 1,400,000 GPU hours (220 years) required for the 12 CodeLlama models, makes continuously updating pre-training data impractical [71]. This cost, along with increasing model sizes and training times, poses a significant barrier to reducing package hallucinations for advanced coding prompts and packages.

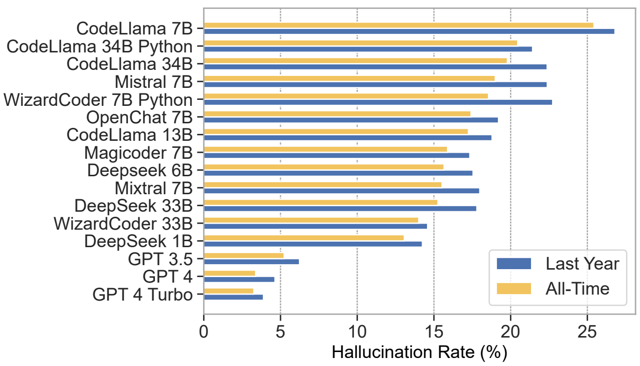


Figure 3: Hallucination rates of recent vs. all-time data sets.

Effect of Temperature Settings. We varied the temperature (Section 2.1) setting for each model between the minimum and maximum allowed values and observed the change in hallucination rate (the maximum temperature for GPT series models is limited to 2, while the open-source models can be set up to 5). All models exhibited a **clear increase in hallucination rate as temperature value increases**, with the effect becoming severe at maximum values. The OpenAI models, as shown in Figure 4, displayed only a slight increase in hallucination rate between temperatures 0 and 1, which then increased sharply between 1 and 2. Most notably, GPT-4 resulted in a hallucination rate of (8.9%), nearly 4x lower than GPT 3.5 (31.8%) at its maximum temperature. At the highest temperature values, open-source models start to generate more hallucinated packages than valid packages. Further, open-source models such as CodeLlama, also demonstrated a tendency to reduce hallucination rate from temperature values 0 to 1.5 before beginning a rapid ascent from 1.5 to 5 (see Figure 4). Most LLMs operate at a default temperature in the range of 0.7 to 1, which by our results is slightly sub-optimal from a package hallucination perspective. Users of commercial LLMs do not have control over this parameter unless they are using the API, which incurs an additional cost beyond a monthly subscription. Consequently, open-source users have more fine-grained control of temperature values, although the proper use of the temperature setting depends on user knowledge, as well as the specific model and environment being employed.

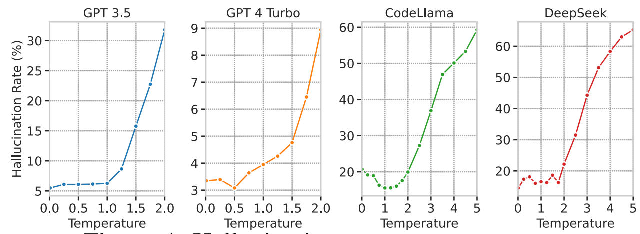


Figure 4: Hallucination rate vs. temperature.

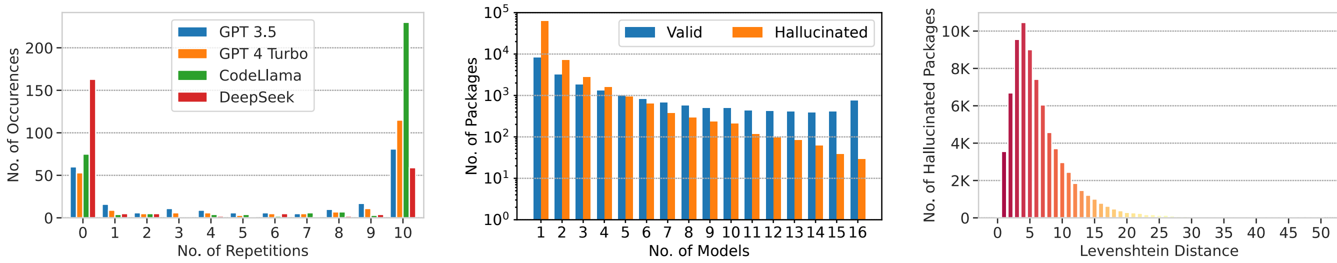
Effect of Decoding Strategies. Next, we adjusted several decoding parameters (top-p, top-k, and min-p values) to reduce the chances of a low probability token being selected as a potential package, with the intuition that lower probability tokens correspond to higher probabilities of hallucination in this context. Below is a summary of the parameters and the values we modified. We evaluated each listed value in isolation, followed by a combined evaluation of the values highlighted in bold, resulting in a total of 10 tests. Note that top-k and min-p were only tested for DeepSeek and CodeLlama, as those values are not able to be modified through the OpenAI API.

- Top-p (0.4, **0.6**, 0.8): Tokens with probabilities adding up to less than this number are discarded.
- Top-k (5, **10**, 15): Select only the top-k most likely tokens.
- Min-p (0.1, **0.2**, 0.3): Tokens with probability smaller than (min-p * probability of most likely token) are discarded.

Varying the decoding values induced a slight *increase* (1.16% on average) in hallucination rate for all four models across all values tested. As we will expand on in RQ3, package hallucinations are often persistently repeated across many iterations. This suggests that even greedy decoding strategies, which prioritize the most probable tokens (i.e., the most probable token is always selected), can generate fictitious packages. The persistent nature of package hallucinations highlights the inherent complexity of the problem and underscores that this phenomenon is a deeply embedded in the fabric of these models.

5.3. Model Behaviors (RQ3)

Frequency of Repeated Hallucinations. To determine whether hallucinations are random error or repeatable phenomena, this analysis will focus on the persistence of hallucinations within a model. We randomly sampled 500 prompts that generated package hallucinations during our initial testing and repeated those queries 10 times per prompt, then recorded how many times the original hallucination was regenerated. Our analysis reveals an unexpected



(a) (b) (c)

Figure 5: (a) Frequency of the occurrence of the same hallucinated package when querying a model with the same prompt 10 times, (b) No. of models in which each unique package (valid and hallucinated) appeared, with the y-axis on a log scale, (c) No. of hallucinated packages vs. Levenshtein to the nearest valid package.

dichotomy when repeatedly querying a model with the same prompt that generated a hallucination: a certain set of hallucinated packages were repeated in all 10 iterations (43%), while another set did not repeat at all across the 10 iterations (39%). This is indicated in Figure 5a, which shows notable spikes at zero repetitions and at 10 repetitions respectively for all the models. Further, 58% of the time, a hallucinated package is repeated more than once in 10 iterations, which shows that a majority of **hallucinations are not simply random errors, but a repeatable phenomenon that persists across multiple iterations**. This is significant because a persistent hallucination is more valuable for malicious actors looking to exploit this vulnerability and makes the hallucination attack vector a more viable threat.

Verbose Models vs. Conservative Models. LLMs operate with inherent randomness and uncertainty, yet this uncertainty is generally not communicated by the LLM to users. Rather, the results are usually presented confidently without mentioning of potential fact-conflicting statements. This non-deterministic behavior of LLMs enables novel and creative output, a desired feature for many NLP tasks but less welcome for code generation, which requires a high degree of accuracy and must adhere to rigid syntax. We define a verbose model as one that operates with higher degree of uncertainty and randomness by generating a greater number of distinct package names while a conservative model generates a lesser number of distinct packages, generally using only the most popular and well-known packages. To this end, we investigated whether verbose models correspond to higher rate of package hallucinations. Our results show a **correlation between the hallucination rate and number of unique packages** that were recommended during this experiment (i.e. a more verbose model was associated with a higher hallucination rate). In light of these findings, as shown in Figure 6, it is reasonable to suggest that coding models, or models performing coding tasks, should operate in a more conservative manner when generating packages to answer a coding prompt. The most successful models in our study (i.e., the lowest hallucination rates) adhered to a smaller subset of well-known packages when generating code, and these models (e.g., the GPT series) also scored the highest on the EvalPlus [47] code quality benchmarks.

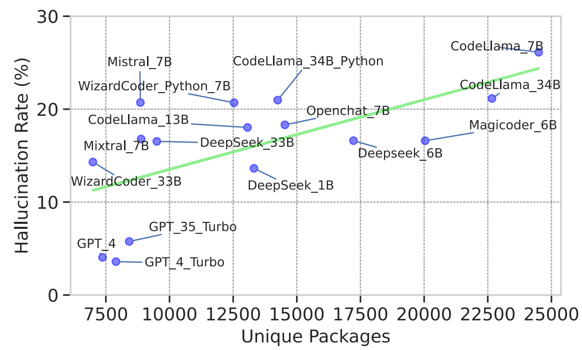


Figure 6: Unique packages vs. total hallucination rate.

This suggests that improving code quality and reducing hallucinations can potentially be achieved simultaneously without a trade-off.

LLMs' Ability to Detect Hallucinations. We then evaluated each model's ability to identify hallucinations vs. valid packages, both from its own code generation outputs and those generated by other models. To test this, we conducted two binary classification tests: (i) each model's ability to detect hallucinations vs. valid packages from its own generated code, and (ii) each model's ability to detect the same from code generated by other models. Names of valid and hallucinated packages produced by each model were randomly sampled and each model was asked if the provided name was a valid Python package. The identification accuracy was calculated as the ratio of correct identifications to the total number of provided packages.

Curiously, 3 of the 4 models (GPT 4 Turbo, GPT 3.5, and DeepSeek) proved to be **highly adept at detecting their own hallucinations** (see Figure 7) with detection accuracies above 75%. The precision and recall values for this test, which also averaged above 75%, are presented in Appendix F. This phenomenon implies that each model's specific error patterns are detectable by the same mechanisms that generate them, suggesting an inherent self-regulatory capability. The indication that these models have an implicit understanding of their own generative patterns which could be leveraged for self-improvement is an im-

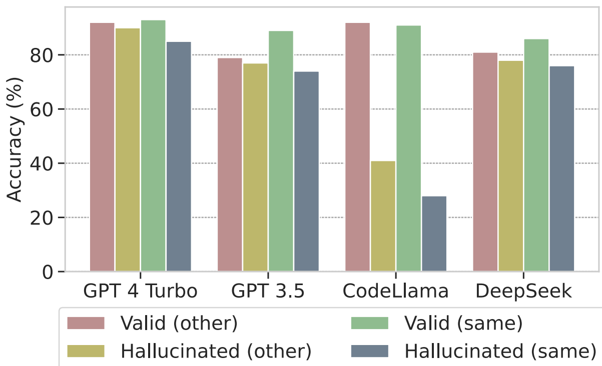


Figure 7: The ability of models to correctly identify valid vs. hallucinated packages.

portant finding for developing mitigation strategies. While capable of detection, none of the models displayed a high enough detection accuracy (all less than 90% cumulative accuracy) to be relied upon as a stand-alone classification mechanism to distinguish between hallucinated vs. valid packages.

Across the board, each model displayed a better ability to identify valid vs. hallucinated packages generated by other models rather than its own, suggesting each model has certain "blind spots" inherent to their architecture. CodeLlama displays unique and interesting behavior during both the tests, as it has an overwhelming propensity to label most packages as valid, resulting in a lower accuracy for hallucinated packages (see Figure 7).

5.4. Characteristics of Hallucinations (RQ4)

Occurrence of the Same Hallucination Across Different Models. To analyze the possibility of the same hallucinated packages being generated across different LLM models, we measured how many other models also generated the same package name given a confirmed package hallucination. Figure 5b shows a clear pattern where **a vast majority of distinctly generated package names were generated by only one model**. In other words, specific package names were usually unique to a single model, where only the most common packages were generated by more than one model. The two populations diverge as the number of models increases, with the number of hallucinated packages decreasing nearly exponentially and the distribution of valid packages becoming more uniform. The finding that valid packages are less dependent on the specific model used for generation can be accounted to the widespread use of the most popular packages. These packages frequently appear in training data and are applicable to the most universal coding problems, causing them to be used in a significant portion of prompts.

Combining the insights gained during the previously discussed persistence analysis (Figure 5a) leads to a key observation. Hallucinations are often persistent (61% are repeated within 10 iterations) within the same model but are not often repeated between models, as 81% of hallucinated packages are generated by only one model. This further reinforces the evidence that while hallucinations are a common phenomenon across various models, the exact nature of these hallucinations is generally model-specific. This suggests a **ubiquitous susceptibility to hallucinations in the architecture or training process of these models, but the manifestation of such errors is influenced by each model's unique configuration and dataset**.

Semantic Similarity Between Hallucinated and popular packages. In order to analyze the semantic similarity between hallucinated and popular real/valid packages, we measured the average *Levenshtein distance* of a package to its nearest neighbor (i.e., the closest valid package). Levenshtein distance is a measure of how many insertions, deletions, and substitutions are required for two strings to match [38]. If the distribution of Levenshtein distances is skewed heavily right, with a peak at or near 0, this would indicate that most hallucinations are very similar to valid package names. In that case, attackers could infer a hallucination target based on more traditional package confusion methods (e.g., typosquatting) rather than analyzing a large volume of model output over time to detect persistent hallucinations that could be used as vessels for malicious code. A higher distance reflects that package hallucinations are more random in nature and difficult to predict, rather than the result of minor grammatical errors. Due to the computational complexity of calculating Levenshtein distance for each hallucinated package compared to all valid Python packages, we limited our search to the most downloaded 20k Python packages according to the PyPI BigQuery dataset [66]. This approach mimics traditional package confusion methods, since typosquatting a rarely used package is not lucrative for attackers.

The results of the Levenshtein distance test, as seen in Figure 5c, suggests that most package hallucinations are not simple *off-by-one errors*. An off-by-one error in our case is defined as a difference between the hallucinated package and its closest match of 1-2 numbers, letters, or punctuation marks, with only 13.4% (10,263 of 76,489) belonging to this category. Another 37.9% (29,025 of 76,489) of packages registered a score between 3-5, which would indicate two words with a common root word or concept that still differ significantly. Notably, 48.6% (37,207 of 76,489) of hallucinations scored 6 or greater, with 20.2% (15,457 of 76,489) of those being 10+, indicating two strings that are very different and likely do not share any common theme.

The presence of such a large proportion of high Levenshtein values suggests that **the majority of hallucinations are not merely trivial typographical errors but are instead substantively different from existing package names**. The observed results provide further evidence that the root cause of hallucinations is likely to be more complex than minor string manipulation, pointing to deeper issues in the model's generative processes that govern the creation of package names. The long right tail of the distribution in Figure 5c indicates a wide variety of hallucinations spread across a broad range, revealing a diversity in types of errors and reinforcing that the generation of hallucinations is a complex issue not limited to simple character substitutions, additions, and deletions.

Effect of Deleted Packages. To determine whether packages that existed before the pre-training data cut-off date (i.e., final day of data included in the model's training set) but were subsequently removed, contribute significantly to package hallucinations, we conducted an analysis using package download counts obtained via Google BigQuery [18]. We searched PyPI download counts from 2022 and earlier to compile a list of packages that existed before 2023 but are no longer available on PyPI. This list was then compared against the master list of packages (obtained from PyPI as of January 10, 2024) of hallucinated packages across all models. We detected 12,871 packages that were available between 2020 and 2022 that have since been removed from PyPI. Out of these deleted packages, only 133 (0.17%) were generated during our analysis, indicating that **deleted packages are a negligible source of package hallucinations**. This finding contradicts our hypothesis, as we expected a sizable percentage of hallucinated packages due to the presence of deleted packages in the training data.

Effect of Language Confusion. Another interesting behavior observed during the main experiment is that many models have a tendency to confuse programming languages while generating package output. In other words, the model is asked to provide Python packages, but instead provides JavaScript packages. To validate the potential existence of this behavior in code generating LLMs, we obtained master lists of packages (using libraries.io [78]) from the nine most popular open-source repositories and compared our list of hallucinated package names generated during Python testing to the respective master lists of valid packages from other programming languages. Any intersection between the two lists would indicate that the LLM, while being asked for Python code, generated a package name that is not published to PyPI but is a valid package name in another programming language (i.e. the LLM generated a package for the wrong language). Overall, **only JavaScript is a significant source of cross-language hallucinations, as 8.7% (6,705 of 76,489) of hallucinated Python packages are valid JavaScript packages**. All other languages contributed negligible hallucinations, combining for only 0.8% (663 of 76,489) across eight other open-source repositories which includes R, Rust, Ruby, PHP, Swift and .NET (see Appendix E for complete results).

6. Mitigation

Motivated by the finding of systemic package hallucination across all the models we evaluated, we then investigated methods to mitigate the occurrence of hallucinations (addressing RQ5) by incorporating several of the most popular techniques from the literature.

6.1. Mitigation Strategies

A seemingly straightforward mitigation strategy when detecting package hallucinations might involve comparing a master list of valid packages to the model's code output. It is imperative to understand why this approach is merely a superficial solution rather than addressing the core issue at hand. The fundamental vulnerability inherent in code generation by LLMs regarding package hallucinations is not solely that they produce non-existent packages, but that they generate package names that were not present in their training data. Our study emphasizes the detection of hallucinated packages as indicators of an underlying vulnerability. The genuine threat posed by package hallucinations emerges when such a hallucinated package is officially published and contains malicious code. The mere act of publishing does not render a previously hallucinated package benign; on the contrary, it becomes an active threat seeking to exploit a security vulnerability. Consequently, simply cross-referencing generated packages with a master list would erroneously validate a malicious package post-publication. Dedicated mitigation techniques are essential to prevent the generation of hallucinated packages altogether, thereby addressing the root cause of this latent vulnerability.

Implementing mitigation strategies specific to code generation LLMs is an unexplored topic, therefore we rely on general hallucination mitigation strategies proposed for standard NLP tasks that can be applied to code generation. These strategies can generally be grouped into two broad categories: prompt engineering and developing models [79]. Prompt engineering includes methods such as Retrieval Augmented Generation (RAG), self-refinement, and prompt tuning. RAG approaches involve enriching the original prompt with additional information gathered from an external source, such as the web or a pre-determined database [39]. This augmentation can occur at any stage—before, during, or after response generation—and can be iterative, improving over multiple cycles until the response is verified as accurate. Self-refinement strategies, on the other hand, utilize the model itself to detect and refine potential hallucinations.

The second main mitigation strategy involves improving the underlying LLM model itself through improved decoding strategies or supervised fine-tuning. Supervised fine-tuning alters model parameters to improve performance on tasks prone to hallucinations, utilizing a labeled dataset for more precise training. Decoding strategies are also considered a viable mitigation strategy from the literature, but based on our findings from RQ2 we know that altering decoding parameters, such as top-k, top-p, and min-p, do not result in a decreased hallucination rate. We evaluate each of the three remaining categories to determine their applicability to code generation and our specific use case of reducing hallucinations, ultimately developing a viable mitigation strategy.

6.2. Mitigation Implementation and Results

Retrieval Augmented Generation (RAG). We utilized a method of before generation RAG to supplement the prompt with valid package names to assist the model in generating a response. We made an additional dataset to serve as the supplementary information by taking the top 20,000 most popular packages from PyPI and asking LlaMA-2 to generate a list of 5 questions that each package could help answer. After removing duplicate responses, this resulted in 65,000 statements in the form “Package [x] could answer questions about [y]”. These 65,000 statements were stored in a vector database, which enables efficient retrieval of semantically similar statements. When a model was asked to recommend packages given a code generation prompt or Stack Overflow question, the vector database would first return the top 5 most semantically similar statements. These statements are prepended to the prompt to give the model additional information containing established valid packages to generate a non-hallucinated response.

Self-Refinement. Drawing on insights from our findings in RQ3 (see Section 5.3), which revealed that LLMs often exhibit proficiency in identifying their own package hallucinations, we implemented a self-refinement method. Following the generation of package names, the model is queried regarding the validity of these packages. If the model indicates that the packages are invalid, the response is regenerated with a specific instruction to not use the invalid package. This regeneration process is allowed to iterate up to five times, acknowledging that many package hallucinations are persistent, as demonstrated in RQ3, and may be repeatedly generated.

Fine-tuning. For our next method, we fine-tuned the models using the code/package list (Heuristic 1) and prompt/package list (Heuristic 2) pairs that were generated during our initial experiments. (Section 5.1) All hallucinations were filtered out and the models were re-trained using the remaining valid responses (560,000 samples).

We implemented these mitigation techniques using the DeepSeek Coder 6B and CodeLlama 7B models. These models were selected because they represent two distinct classes of foundational models, with these specific parameter sizes reflecting diverse performance levels: DeepSeek being among the best-performing, and CodeLlama among the worst-performing during our initial experiments (see Section 5.1). Both models were tested using each of the above methods individually and then using all three methods in an ensemble configuration.

TABLE 2: Performance of the mitigation techniques.

	DeepSeek	**CodeLlama**
Baseline (No Mitigations)	16.14%	26.28%
Retrieval Augmented Generation (RAG)	12.24%	13.40%
Self-Detected Feedback	13.04%	25.51%
Fine-tuning	**2.66%**	**10.27%**
Ensemble	**2.40%**	**9.32%**

Results. Overall, all the mitigation strategies we implemented resulted in a reduced rate of package hallucination, with **RAG and Supervised Fine-Tuning proving particularly effective** (see Table 2). Fine-tuning proved to significantly improve results, especially for the DeepSeek model, where hallucinations were reduced by 83%, achieving a total rate of just 2.66%, which is a lower rate than any of the ChatGPT models (observed during RQ1). Self-detected feedback was also much more effective for the DeepSeek model (19% reduction) compared to the CodeLlama model (3% reduction). This aligns with our results in RQ3, where the DeepSeek model was proficient at detecting hallucinations while the CodeLlama model had a strong bias towards labeling packages as valid, that limited its ability to reliably detect errors. The ensemble method of combining all mitigation strategies further improved results, reducing hallucination rates by 85% and 64% from their baseline levels for DeepSeek and CodeLlama, respectively.

7. Discussion and Conclusion

One limitation of our study is the emergence of more advanced models since our evaluations. These newer models may offer improved performance and different hallucination characteristics, which were not captured in our study. The study also includes fewer commercial models due to funding constraints, meaning the findings may not fully represent the performance and hallucination tendencies of the latest commercial LLMs.

In terms of future work, the precise underlying causes of package hallucinations is still an open question. This includes exploring the architecture and components of LLMs that may contribute to these errors, examining the adequacy of tokenizers, and assessing how training data composition and preprocessing impact hallucination tendencies. Identifying and mitigating these underlying issues could lead to more robust and reliable code generation models. Future work could also focus on developing and testing more sophisticated mitigation strategies tailored specifically for code generation tasks. This could involve advanced techniques in prompt engineering, leveraging complex knowledge graphs, refining loss functions, and exploring new fine-tuning methods. Integrating real-time feedback mechanisms to dynamically adjust model output could further reduce hallucination rates. Understanding how package hallucinations are systemic and persistent at the token level remains crucial.

In conclusion, we systematically studied package hallucinations in code generation LLMs, including both commercial and open-source models. Our comprehensive analysis revealed that 19.7% of generated packages are fictitious, posing a critical threat to software security through package confusion attacks. We identified key behavioral patterns and characterized hallucinated packages, proposing effective mitigation strategies. Our findings underscore the importance of addressing package hallucinations to enhance the reliability and security of AI-assisted software development.

References

[1] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv:2303.08774*, 2023.

[2] R. Aksitov, C.-C. Chang, D. Reitter, S. Shakeri, and Y. Sung, "Characterizing attribution and fluency tradeoffs for retrieval-augmented large language models," *arXiv:2302.05578*, 2023.

[3] Anthropic, "The Claude 3 Model Family: Opus, Sonnet, Haiku," https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf, 2023, [Online; accessed 15-May-2024].

[4] 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:2108.07258*, 2021.

[5] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *NIPS*, vol. 33, pp. 1877–1901, 2020.

[6] CBSNews, "SolarWinds: How Russian spies hacked the Justice, State, Treasury, Energy, and Commerce Departments," https://www.cbsnews.com/news/solarwinds-hack-russia-cyberattack-60-minutes-2021-02-14, 2021, [Online; accessed 15-May-2024].

[7] I. Chalkidis, M. Fergadiotis, P. Malakasiotis, N. Aletras, and I. Androutsopoulos, "LEGAL-BERT: The muppets straight out of law school," in *EMNLP*, T. Cohn, Y. He, and Y. Liu, Eds., 2020.

[8] H.-S. Chang and A. McCallum, "Softmax bottleneck makes language models unable to represent multi-mode word distributions," in *ACL*, 2022.

[9] 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:2107.03374*, 2021.

[10] X. Chen, M. Li, X. Gao, and X. Zhang, "Towards improving faithfulness in abstractive summarization," *NIPS*, 2022.

[11] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *NAACL*, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:52967399

[12] N. Dziri, A. Madotto, O. Zaïane, and A. J. Bose, "Neural path hunter: Reducing hallucination in dialogue systems via path grounding," in *EMNLP*, 2021.

[13] P. S. Foundation, "Python Package Index - PyPI," https://pypi.org, 2024, [Online; accessed 15-May-2024].

[14] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "OPTQ: Accurate quantization for generative pre-trained transformers," in *ICLR*, 2023.

[15] I. O. Gallegos, R. A. Rossi, J. Barrow, M. M. Tanjim, S. Kim, F. Dernoncourt, T. Yu, R. Zhang, and N. K. Ahmed, "Bias and fairness in large language models: A survey," *arXiv:2309.00770*, 2023.

[16] Gartner, "Gartner Identifies Top Security and Risk Management Trends for 2022," https://www.gartner.com/en/newsroom/press-releases/2022-03-07-gartner-identifies-top-security-and-risk-management-trends-for-2022, 2022, [Online; accessed 15-May-2024].

[17] GitHub, "Octoverse: The state of open source and rise of AI in 2023," https://github.blog/2023-11-08-the-state-of-open-source-and-ai, 2023, [Online; accessed 15-May-2024].

[18] Google, "BigQuery," https://cloud.google.com/bigquery, 2024, [Online; accessed 15-May-2024].

[19] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv:2401.14196*, 2024.

[20] J. Hazell, "Spear phishing with large language models," *arXiv:2305.06972*, 2023.

[21] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, "The curious case of neural text degeneration," in *ICLR*, 2020.

[22] L. Huang, W. Yu, W. Ma, W. Zhong, Z. Feng, H. Wang, Q. Chen, W. Peng, X. Feng, B. Qin *et al.*, "A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions," *arXiv preprint:2311.05232*, 2023.

[23] ——, "A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions," *arXiv:2311.05232*, 2023.

[24] M. Huang, X. Zhu, and J. Gao, "Challenges in building intelligent open-domain dialog systems," *TOIS*, 2020.

[25] S. E. Huber, K. Kiili, S. Nebel, R. M. Ryan, M. Sailer, and M. Ninaus, "Leveraging the potential of large language models in education through playful and game-based learning," *Educational Psychology Review*, 2024.

[26] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, Y. Xu, E. Ishii, Y. J. Bang, A. Madotto, and P. Fung, "Survey of hallucination in natural language generation," *ACM Computing Surveys*, 2023.

[27] Z. Ji, Z. Liu, N. Lee, T. Yu, B. Wilie, M. Zeng, and P. Fung, "RHO: Reducing hallucination in open-domain dialogues with knowledge grounding," in *ACL*, 2023.

[28] Z. Ji, T. Yu, Y. Xu, N. Lee, E. Ishii, and P. Fung, "Towards mitigating LLM hallucination via self reflection," in *EMNLP*, H. Bouamor, J. Pino, and K. Bali, Eds., 2023.

[29] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. d. l. Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier *et al.*, "Mistral 7b," *arXiv:2310.06825*, 2023.

[30] A. Q. Jiang, A. Sablayrolles, A. Roux, A. Mensch, B. Savary, C. Bamford, D. S. Chaplot, D. d. l. Casas, E. B. Hanna, F. Bressand *et al.*, "Mixtral of experts," *arXiv:2401.04088*, 2024.

[31] D. Kang, X. Li, I. Stoica, C. Guestrin, M. Zaharia, and T. Hashimoto, "Exploiting programmatic behavior of llms: Dual-use through standard security attacks," *arXiv preprint:2302.05733*, 2023.

[32] B. Kaplan and J. Qian, "A survey on common threats in npm and pypi registries," in *MLHat*, 2021.

[33] F. Kreuk, G. Synnaeve, A. Polyak, U. Singer, A. Défossez, J. Copet, D. Parikh, Y. Taigman, and Y. Adi, "Audiogen: Textually guided audio generation," in *ICLR*, 2023.

[34] R. Labs, "VMConnect supply chain attack continues, evidence points to North Korea," https://www.reversinglabs.com/blog/vmconnect-supply-chain-campaign-continues, 2023, [Online; accessed 15-May-2024].

[35] P. Ladisa, H. Plate, M. Martinez, and O. Barais, "Sok: Taxonomy of attacks on open-source software supply chains," in *SP*, 2023.

[36] B. Lanyado, "Can you trust chatgpt's package recommendations?" https://lasso-security.webflow.io/blog/ai-package-hallucinations, 2023, [Online; accessed 15-May-2024].

[37] ——, "Diving Deeper into AI Package Hallucinations," https://vulcan.io/blog/ai-hallucinations-package-risk, 2023, [Online; accessed 15-May-2024].

[38] V. I. Levenshtein *et al.*, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, 1966.

[39] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *NIPS*, 2020.

[40] C. Li, B. Bi, M. Yan, W. Wang, and S. Huang, "Addressing semantic drift in generative question answering with auxiliary extraction," in *ACL-IJCNLP*, 2021.

[41] H. Li, D. Guo, W. Fan, M. Xu, J. Huang, F. Meng, and Y. Song, "Multi-step jailbreaking privacy attacks on chatGPT," in *EMNLP*, 2023.

[42] K. Li, O. Patel, F. Viégas, H. Pfister, and M. Wattenberg, "Inference-time intervention: Eliciting truthful answers from a language model," *NIPS*, 2024.

[43] J. T. Liang, C. Yang, and B. A. Myers, "A large-scale survey on the usability of ai programming assistants: Successes and challenges," in *ICSE*, 2024.

[44] S. Lin, J. Hilton, and O. Evans, "TruthfulQA: Measuring how models mimic human falsehoods," in *ACL*, 2022.

[45] B. Liu, J. Ash, S. Goel, A. Krishnamurthy, and C. Zhang, "Exposing attention glitches with flip-flop language modeling," *NIPS*, 2024.

[46] F. Liu, Y. Liu, L. Shi, H. Huang, R. Wang, Z. Yang, and L. Zhang, "Exploring and evaluating hallucinations in llm-powered code generation," *arXiv:2404.00971*, 2024.

[47] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *NIPS*, 2024.

[48] Y. Liu, G. Deng, Z. Xu, Y. Li, Y. Zheng, Y. Zhang, L. Zhao, T. Zhang, K. Wang, and Y. Liu, "Jailbreaking chatgpt via prompt engineering: An empirical study," *arXiv:2305.13860*, 2023.

[49] Y. Liu, X. Zeng, F. Meng, and J. Zhou, "Instruction position matters in sequence generation with large language models," *arXiv:2308.12097*, 2023.

[50] M. Y. Lu, B. Chen, D. F. Williamson, R. J. Chen, I. Liang, T. Ding, G. Jaume, I. Odintsov, L. P. Le, G. Gerber *et al.*, "A visual-language foundation model for computational pathology," *Nature Medicine*, 2024.

[51] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, and D. Jiang, "Wizardcoder: Empowering code large language models with evol-instruct," in *ICLR*, 2024.

[52] A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegreffe, U. Alon, N. Dziri, S. Prabhumoye, Y. Yang *et al.*, "Self-refine: Iterative refinement with self-feedback," *NIPS*, 2024.

[53] J. Maynez, S. Narayan, B. Bohnet, and R. T. Mcdonald, "On faithfulness and factuality in abstractive summarization," in *ACL*, 2020.

[54] Midjourney, "Midjourney Model Versions," https://docs.midjourney.com/docs/model-versions, 2023, [Online; accessed 15-May-2024].

[55] I. Momennejad, H. Hasanbeig, F. Vieira Frujeri, H. Sharma, N. Jojic, H. Palangi, R. Ness, and J. Larson, "Evaluating cognitive maps and planning in large language models with cogeval," *NIPS*, 2024.

[56] M. Moor, O. Banerjee, Z. S. H. Abad, H. M. Krumholz, J. Leskovec, E. J. Topol, and P. Rajpurkar, "Foundation models for generalist medical artificial intelligence," *Nature*, 2023.

[57] F. Nagle, J. Dana, J. Hoffman, S. Randazzo, and Y. Zhou, "Census ii of free and open source software—application libraries," *LF, LISH and OpenSSF*, 2022.

[58] S. Neupane, G. Holmes, E. Wyss, D. Davidson, and L. De Carli, "Beyond typosquatting: an in-depth look at package confusion," in *USENIX*, 2023.

[59] npm, https://npmjs.com, 2024, [Online; accessed 15-May-2024].

[60] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's knife collection: A review of open source software supply chain attacks," in *DIMVA*, 2020.

[61] Y. Onoe, M. Zhang, E. Choi, and G. Durrett, "Entity cloze by date: What LMs know about unseen entities," in *NAACL*, 2022.

[62] Oobabooga, "A Gradio web UI for Large Language Models." https://github.com/oobabooga/text-generation-webui, 2023, [Online; accessed 15-May-2024].

[63] OpenAI, "Video generation models as world simulators," https://openai.com/index/video-generation-models-as-world-simulators, 2024, [Online; accessed 15-May-2024].

[64] 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," in *SP*, 2022.

[65] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do users write more insecure code with ai assistants?" in *CCS*, 2023.

[66] PyPA, "Analyzing PyPI package downloads," https://packaging.python.org/en/latest/guides/analyzing-pypi-package-downloads, 2024, [Online; accessed 15-May-2024].

[67] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark *et al.*, "Learning transferable visual models from natural language supervision," in *ICML*, 2021.

[68] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever, "Zero-shot text-to-image generation," in *ICML*, 2021.

[69] RedHat, "The State of Enterprise Open Source 2022," www.redhat.com/en/enterprise-open-source-report/2022, 2022.

[70] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer, "High-resolution image synthesis with latent diffusion models," in *CVPR*, 2022.

[71] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, "Code llama: Open foundation models for code," *arXiv preprint:2308.12950*, 2023.

[72] K. Shuster, J. Xu, M. Komeili, D. Ju, E. M. Smith, S. Roller, M. Ung, M. Chen, K. Arora, J. Lane *et al.*, "Blenderbot 3: a deployed conversational agent that continually learns to responsibly engage," *arXiv:2208.03188*, 2022.

[73] Sonatype, "9th Annual State of the Software Supply Chain," https://www.sonatype.com/state-of-the-software-supply-chain/introduction, 2023, [Online; accessed 15-May-2024].

[74] ——, "Top 8 malicious attacks recently found on PyPI," https://blog.sonatype.com/top-8-malicious-attacks-recently-found-on-pypi, 2023, [Online; accessed 15-May-2024].

[75] stackoverflow, "Stack Overflow," https://stackoverflow.com/, 2024, [Online; accessed 15-May-2024].

[76] G. Team, R. Anil, S. Borgeaud, Y. Wu, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth *et al.*, "Gemini: a family of highly capable multimodal models," *arXiv:2312.11805*, 2023.

[77] K. Tian, E. Mitchell, H. Yao, C. D. Manning, and C. Finn, "Fine-tuning language models for factuality," in *ICLR*, 2024.

[78] TIDELIFT, "Libraries.io," https://libraries.io, 2024, [Online; accessed 15-May-2024].

[79] S. Tonmoy, S. Zaman, V. Jain, A. Rani, V. Rawte, A. Chadha, and A. Das, "A comprehensive survey of hallucination mitigation techniques in large language models," *arXiv:2401.01313*, 2024.

[80] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv:2307.09288*, 2023.

[81] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *NIPS*, 2017.

[82] D.-L. Vu, Z. Newman, and J. S. Meyers, "Bad snakes: Understanding and improving python package index malware scanning," in *ICSE*, 2023.

[83] C. Wang and R. Sennrich, "On exposure bias, hallucination and domain shift in neural machine translation," in *ACL*, 2020.

[84] G. Wang, S. Cheng, X. Zhan, X. Li, S. Song, and Y. Liu, "Openchat: Advancing open-source language models with mixed-quality data," *arXiv:2309.11235*, 2023.

[85] A. Wei, N. Haghtalab, and J. Steinhardt, "Jailbroken: How does llm safety training fail?" *NIPS*, 2024.

[86] Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang, "Magicoder: Source code is all you need," *arXiv:2312.02120*, 2023.

[87] S. Xu, G. Chen, Y.-X. Guo, J. Yang, C. Li, Z. Zang, Y. Zhang, X. Tong, and B. Guo, "Vasa-1: Lifelike audio-driven talking faces generated in real time," *arXiv:2404.10667*, 2024.

[88] J. Zhang, K. Huang, B. Chen, C. Wang, Z. Tian, and X. Peng, "Malicious package detection in npm and pypi using a single model of malicious behavior sequence," *arXiv:2309.02637*, 2023.

[89] Y. Zhang, Y. Li, L. Cui, D. Cai, L. Liu, T. Fu, X. Huang, E. Zhao, Y. Zhang, Y. Chen, L. Wang, A. T. Luu, W. Bi, F. Shi, and S. Shi, "Siren's song in the ai ocean: A survey on hallucination in large language models," *arXiv:2309.01219*, 2023.

[90] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *USENIX*, 2019.

Appendix

Appendix A. Truncated List of LLM Generated Coding Prompts

1) Generate Python code that implements a simple web server that can handle GET and POST requests using the http.server module.
2) Generate Python code that imports the AWS SDK for Python and creates an Amazon S3 bucket, an Amazon EC2 instance, and an Amazon RDS database, and sets up user authentication using IAM roles.
3) Generate Python code that implements a simple flock-based file locking mechanism using the 'with' syntax, allowing multiple threads to safely access a shared resource while ensuring exclusive access for writing operations.
4) Generate Python code that implements a backport of f-strings to Python versions prior to 3.6 using the 'str.format()' method.
5) Generate Python code that imports the Selenium library and uses it to automate interactions with a web application, such as navigating to pages, filling out forms, and verifying expected elements are present on the page.
6) Generate Python code that implements a rate limiter for Flask applications using the 'limiter' library, which provides a simple way to add rate limiting to any Flask endpoint.

⋮

9810) Generate Python code that imports the PyGlove library and uses it to manipulate various Python objects, such as lists, dictionaries, and strings, by applying operations like reversal, sorting, indexing, slicing, concatenation, and membership testing.
9811) Generate Python code that imports the necessary libraries and uses the Fuzzy Self-Tuning PSO algorithm to perform global optimization for a given function.
9812) Generate Python code that imports the necessary libraries and sets up a configurable middleware pipeline for making HTTP requests to the Microsoft Graph API using the Core component of the Microsoft Graph Python SDK.
9813) Generate Python code that imports the necessary CUDA libraries and creates a simple kernel that performs a matrix multiplication using CUDA's GPU acceleration.
9814) Generate Python code that imports the threading module and uses it to create threads for monitoring and tracing in an application, using the OpenCensus API to collect metrics and trace data.

Appendix B. System Messages and Prompts

Figures 8 to 10 below show the system messages and prompts that are sent to each model to generate the code samples and package lists.

System Message

You are a coding assistant that generates [Python/JavaScript] code. Provide only the code and add additional explanatory text only when absolutely necessary. If no code is required to answer the question, simply reply 'None'.

Prompt

[LLM Generated prompt from Appendix A or Stack Overflow question.

Figure 8: Code generation phase.

System Message

You are a coding assistant that determines [Python/JavaScript] packages necessary to execute code. Respond with only a list of [Python/JavaScript] packages, separated by commas and no additional text or formatting. If there is no code provided, respond 'None', otherwise the response must begin with the name of a [Python/JavaScript] package.

Prompt

Which [Python/JavaScript] packages are required to run this code: [Code sample generated from code generation phase]

Figure 9: Package generation phase - Heuristic 1.

System Message

You are a coding assistant that recommends [Python/JavaScript] packages that would be helpful to solve given problems. Respond with only a list of [Python/JavaScript] packages, separated by commas and no additional text or formatting. The response must begin with the name of a [Python/JavaScript] package.

Prompt

Which [Python/JavaScript] packages would be useful in solving the following coding problem: [Original LLM generated prompt or Stack Overflow question]

Figure 10: Package Generation phase - Heuristic 2.

Appendix C. Model Parameters

Table 3 shows the model parameters used during our RQ1 tests.

TABLE 3: An overview of the model parameters.

Parameter	Value
Temperature (Code Generation)	0.7
Temperature (Package Prompts)	0.01
Top-p	0.9
Top-k	20
Repetition Penalty	1
Max tokens (Code Generation)	2048
Max tokens (Package Prompts)	64
Typical-p	1
Epsilon Cutoff	0
Eta Cutoff	0
Diversity Penalty	0

Appendix D. Python vs. JavaScript Hallucination

The linear relationship demonstrating a model's propensity to hallucinate across both Python and JavaScript (as described in Section 5.1) is shown in Figure 11.

Appendix E. Cross Language Hallucinations

Table 4 displays the number of packages that were generated for Python specific code and were detected as hallucinations, but are actually valid packages in other programming languages.

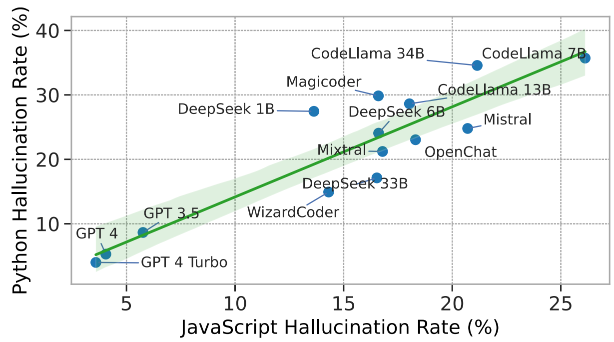


Figure 11: Python vs. JavaScript hallucination rates.

TABLE 4: Confusion by programming language repository.

Programming Language	No. of Cross-Language Hallucinations
JavaScript (npm)	6,705
R (CRAN)	293
Rust (Cargo)	181
Ruby (Rubygems)	123
PHP (Packagist)	47
Swift/Objective C (Cocoapods)	10
.NET (Nuget)	9
Go (Go)	0
Java (Maven)	0

Appendix F. Hallucination Detection Performance

Table 5 shows average precision and recall values for our test on LLMs' hallucination detection performance (as described in Section 5.3).

TABLE 5: Performance of hallucination detection tests.

Model Name	Other		Same	
	Precision	Recall	Precision	Recall
GPT 4 Turbo	0.91	0.91	0.89	0.89
GPT 3.5	0.78	0.78	0.82	0.82
CodeLlama	0.72	0.66	0.66	0.60
DeepSeek	0.80	0.80	0.81	0.78

Appendix G. Complete Results for Python and JavaScript

Table 6 and Table 7 shows complete results for package hallucination experiments observed across all tested models for Python and JavaScript.

TABLE 6: Hallucination Percentages for all models tested using Python code

Model	Total Hallucination	LLM Generated Prompts	Stack Overflow Prompts	"pip install"
GPT-4 Turbo	**3.59% (2,739/76,313)**	**3.29% (1,518/46,204)**	**4.07% (1,169/28,728)**	**3.77% (52/1,381)**
GPT-4	4.05% (2,969/73,396)	3.83% (1,741/45,403)	4.45% (1,046/23,487)	4.04% 182/4,506
GPT-3.5 Turbo	5.76% (4,387/76,123)	5.98% (2,495/41,7)	5.50% (1,868/33,948)	5.63% (24/426)
DeepSeek 1B	**13.63% (12,481/91,543)**	11.07% (5,847/52,806)	**16.39% (5,901/36,007)**	26.85% (733/2,730)
DeepSeek 33B	16.53% (7,071/42,788)	13.85% (3,623/26,167)	25.47% (3,033/11,906)	8.80% (415/4,715)
WizardCoder 33B	14.31% (4,909/34,300)	**9.79% (1,579/16,125)**	21.40% (2,8523/13,329)	9.84% (477/4,846)
DeepSeek 6B	16.61% (16,526/99,505)	14.01% (9,240/65,957)	23.56% (6,792/28,828)	10.47% (494/4,720)
OpenChat 7B	18.31% (16,932/92,452)	17.39% (9,582/55,092)	19.98% (6,454/32,307)	17.73% (896/5,053)
CodeLlama 13B	18.03% (12,404/68,809)	15.21% (6,450/42,410)	22.76% (5,752/25,273)	17.94% (202/1,126)
Mixtral 8x7B	16.79% (7,753/46,166)	13.12% (2,749/20,951)	20.92% (4,068/19,949)	16.23% (936/5,766)
MagiCoder 7B	16.60% (20,258/122,057)	15.76% (11,994/76,096)	18.48% (7,621/41,248)	13.64% (643/4,713)
CodeLlama 34B	21.15% (24,905/117,777)	15.22% (9,495/62,366)	28.56% (14,891/52,135)	15.84% (519/3,276)
Mistral 7B	20.71% (7,959/38,437)	14.47% (2,808/19,412)	30.69% (3,922/12,778)	19.67% (1,229/6,247)
WizardCoder 7B - Python	20.69% (11,408/55,131)	16.80% (4,698/27,962)	26.73% (6,112/22,867)	13.90% (598/4,302)
CodeLlama 34B - Python	20.97% (12,128/57,833)	19.01% (5,913/31,112)	23.39% (6,208/26,540)	**3.87% (7/181)**
CodeLlama 7B	26.12% (27,814/106,487)	21.51% (12,961/60,261)	32.53% (14,671/45,099)	16.15% (182/1,127)

TABLE 7: Hallucination percentages for all models tested using JavaScript code

Model	Total Hallucination	LLM Generated Prompts	Stack Overflow Prompts	"npm install"
GPT-4 Turbo	**4.00% (2,101/52,484)**	**2.57% (735/28,545)**	**5.58% (1283/23,009)**	**8.92% (83/930)**
GPT-4	5.29% (2,911/55,021)	3.78% (1,116/29,534)	3.86% (1,672/23,416)	5.94% 123/2,071
GPT-3.5 Turbo	8.65% (4,576/52,890)	6.92% (1,930/27,909)	10.46% (2,579/24,662)	21.00% (67/319)
DeepSeek 1B	27.45% (29,305/106,755)	23.96% (14,300/59,681)	31.87% (14,975/46,988)	34.88% (30/86)
DeepSeek 33B	17.12% (10,505/61,373)	13.28% (5,472/41,209)	25.65% (4,940/19,260)	10.29% (93/904)
WizardCoder 33B	**14.93% (3,876/25,969)**	**7.83% (1,038/13,256)**	**23.31% (2,772/11,894)**	**8.06% (66/819)**
DeepSeek 6B	24.06% (25,178/104,628)	17.36% (9,595/55,255)	31.82% (15,493/48,693)	13.24% (90/680)
OpenChat 7B	23.04% (24,863/107,903)	18.34% (10,275/56,039)	28.18% (14,557/51,657)	14.98% (31/207)
CodeLlama 13B	28.62% (11,984/41,866)	19.10% (3,774/19,757)	37.15% (8,200/22,071)	26.32% (10/38)
Mixtral 8x7B	21.22% (9,429/44,435)	14.83% (2,882/19,436)	27.98% (6,257/22,362)	11.00% (290/2,637)
MagiCoder 7B	29.85% (40,085/134,276)	26.27% (20,703/78817)	35.10% (19,301/54,982)	16.98% (81/477)
CodeLlama 34B	34.57% (38,607/111,668)	25.18% (13,090/51,995)	42.77% (25,489/59,590)	33.73% (28/83)
Mistral 7B	24.79% (10,505/42,381)	20.60% (4,961/24,083)	34.59% (5,252/15,183)	9.37% (292/3,115)
CodeLlama 7B	35.71% (33,877/94,876)	27.32% (12,103/44,298)	43.07% (21,751/50,507)	32.39% (23/71)