# Evaluating Inference-Time Defenses against Package Hallucination in LLM-Generated Code


Alberick Euraste Djire
University of Luxembourg
Luxembourg, Luxembourg
AI4D (CITADEL)
Ouagadougou, Burkina Faso
euraste.djire@uni.lu

Iyiola E. Olatunji
University of Luxembourg
Luxembourg, Luxembourg
emmanuel.olatunji@uni.lu

Melissa Tessa
University of Luxembourg
Luxembourg, Luxembourg
melissa.tessa@uni.lu

Earl T. Barr
University College London
London, United Kingdom
e.barr@ucl.ac.uk

Jacques Klein
University of Luxembourg
Luxembourg, Luxembourg
jacques.klein@uni.lu

Tegawendé F. Bissyandé
University of Luxembourg
Luxembourg, Luxembourg
tegawende.bissyande@uni.lu





## Abstract

LLMs are increasingly used for code generation, yet they frequently hallucinate non-existent software packages, creating exploitable entry points into the software supply chain. We make four contributions to this problem. First, we show that prior evaluation methodologies systematically inflate hallucination rates by misclassifying standard-library modules as hallucinations in some languages. For Python, the overestimation reaches 9.4 percentage points. Second, we evaluate seven inference-time defenses for mitigating package hallucinations, including five guided decoding strategies (Greedy, Contrastive, DoLa, Nudging, and Active Layer-Contrastive Decoding), an iterative self-refinement approach (Self-Refine), and a Retrieval-Augmented Generation (RAG)-based defense.. Across eight models spanning five families and four programming languages (Python, JavaScript, Ruby, Rust), RAG reduces the package hallucination rate (PHR) in 18 of 32 model–language configurations. Third, we introduce Package Utility (PU) to assess whether defenses preserve valid and task-relevant recommendations. Among strategies evaluated, Greedy decoding provides the strongest average mitigation–utility trade-off. Fourth, we stress-test all strategies under adversarial prompts seeded with fabricated package names and find that PHR surges by up to 45 percentage points relative to standard prompts, with Ruby consistently the most vulnerable language (80.9–95.2%). Under adversarial conditions, RAG and Self-Refine outperform all decoding-only strategies, indicating that robust defense requires either external grounding or iterative self-verification when prompts are actively hostile.

Our results recast package hallucination as both a measurement problem and a decoding-time control problem, and they demonstrate that the choice of defense must be matched to the threat model and recommendation utility.




*ASE '26, Munich, Germany*



ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3837555



## CCS Concepts

• **Software and its engineering** → **Software usability**; **Software libraries and repositories**; • **Security and privacy** → *Software security engineering*.

## Keywords

LLM, Package Hallucination, Guided Decoding, Supply Chain Security, Adversarial Prompts

**ACM Reference Format:**
Alberick Euraste Djire, Iyiola E. Olatunji, Melissa Tessa, Earl T. Barr, Jacques Klein, and Tegawendé F. Bissyandé. 2026. Evaluating Inference-Time Defenses against Package Hallucination in LLM-Generated Code. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3832783.3837555


## 1 Introduction

Code-generating large language models (LLMs) have become a part of software development, powering IDE assistants, automated pipelines, and interactive chat-based programming workflows [9, 11, 28, 31, 37, 38]. When a model produces code, it implicitly selects which external dependencies a developer will import, install, and ultimately trust. This selection is driven by the model's parametric memory rather than by verified package metadata, and it can go wrong in a way that is difficult to detect and easy to exploit. When an LLM recommends a package that does not exist in any legitimate registry, the result is a *package hallucination* [34]. If an attacker registers that hallucinated name on a public registry and uploads a malicious payload, any developer who blindly installs the LLM's recommendation executes attacker-controlled code. This attack pattern, variously called *slopsquatting* or AI-induced package squatting [1, 2], is a growing threat to the software supply chain [18, 40].

Empirical evidence shows that package hallucination is neither rare nor confined to weak models. Spracklen et al. [34] found that at least 5.2% of packages recommended by commercial LLMs and 21.7% of those from open-source models were hallucinated, across more than 576,000 code samples. Krishna et al. [17] showed that hallucination rates vary with programming language, model size, and prompt specificity. Haque et al. [13] demonstrated that the

phenomenon extends to shell-command generation for Go, where fabricated module paths take realistic URL-style forms. The problem is compounded by the fact that hallucinated names are often persistent across repeated generations [34], making them predictable targets for attackers. Recent work has also shown that package hallucination is more prevalent in smaller language models than in larger ones, making them a particularly challenging setting for mitigation [17, 34]. Therefore, we focus our evaluation on smaller open-source models. Beyond providing a stringent testbed for inference-time defenses, these models remain attractive in practice because of their lower computational requirements and widespread deployment in resource-constrained environments.

Several defenses have been proposed. Retrieval-augmented generation (RAG) grounds the model in verified registry data and can sharply reduce hallucination rates [20, 34]. Self-refinement asks the model to critique and revise its own output [24, 34]. Supervised fine-tuning updates model weights with package-aware supervision [34]. Each of these approaches has clear merits, but each also has structural limitations. RAG introduces an external retrieval pipeline that may not be available in all deployment settings and that can itself be poisoned. Self-refinement operates post-hoc and is unreliable for small models that cannot detect their own errors [19]. Fine-tuning is expensive and couples the defense to a specific model checkpoint.

A largely unexplored alternative is to intervene during generation itself, at the point where the model commits to dependency tokens. Guided decoding strategies modify the token selection policy at inference time without retraining the model and without relying on external knowledge. Contrastive Decoding [21] re-ranks candidate tokens by contrasting a larger "expert" model against a smaller "amateur" model, favoring tokens that reflect the expert's superior factual grounding. DoLa [3] contrasts logits from late and early transformer layers within a single model, amplifying factual knowledge encoded in deeper layers while suppressing surface patterns from shallower ones. Zhang et al. [41] extended this idea with Active Layer-Contrastive Decoding (ALCD), which uses reinforcement learning to decide when to apply layer contrasts. Nudging [10] uses a small aligned surrogate model to replace tokens whenever the base model's confidence falls below a threshold. All four methods were originally developed for natural-language factuality tasks; none has been evaluated for package hallucination. Beyond mitigation, we identify a fundamental measurement flaw. Existing evaluation frameworks classify recommended packages absent from the target registry (e.g., PyPI or npm) as hallucinated. This misclassifies standard-library modules (e.g. `os`, `math`, and `json` in Python) which appear in no registry. Consequently, reported rates are inflated by up to 7.6 percentage points under default decoding, distorting baselines and apparent defense effectiveness. This paper makes four contributions:

- **A corrected evaluation framework for package hallucination.** We identify a systematic source of false positives in registry-based evaluation, namely the misclassification of standard-library imports as hallucinated packages, and reduce this bias by augmenting registry-based validation with language-specific standard-library manifests.
- **Inference-time defenses based on guided decoding for smaller LLMs.** We adapt and evaluate five inference-time decoding strategies (Greedy, Contrastive Decoding, DoLa, ALCD and Nudging) that intervene directly in token selection to reduce package hallucinations without model retraining or dependence on external retrieval. We compare them against standard baselines, including Vanilla decoding, RAG, and Self-Refine.
- **A utility-aware evaluation of package-hallucination defenses.** We introduce package utility (PU), a precision–recall measure of valid and task-relevant recommendations, and show that reducing PHR does not necessarily preserve recommendation usefulness.
- **A multilingual evaluation under both standard and adversarial prompts.** We assess all strategies across four programming languages with distinct package ecosystems and under adversarial prompts seeded with fabricated package names, providing a comparative view of when guided decoding is effective and where it fails.

## 2 Related Work

**Package Hallucination.** Package hallucination occurs when an LLM generates imports or dependency recommendations for packages that do not exist in the target software ecosystem [17, 34]. It represents a specialized form of code hallucination, alongside broader syntactic, semantic, and requirement-level failures in LLM-generated code [23, 35, 42]. Unlike syntactic errors, which can often be detected through static analysis and iterative feedback [4, 6], package hallucinations are typically syntactically valid and only become apparent during installation or execution [13, 17, 34]. This makes them a software supply-chain security concern, enabling attacks such as slopsquatting [1, 2, 36] and related forms of package squatting including typosquatting, impersonation squatting, and compound squatting [15, 25]. Empirical studies have shown that package hallucination is prevalent across models, programming languages with over 19.7% of recommended packages for Python and JavaScript prompting strategies and more severe for smaller LLMs [13, 17, 34, 39]. Prior work further attributes hallucinations to both generation-time uncertainty [8, 16] and incorrect factual associations stored in parametric memory [8, 34]. Our work builds on this literature by correcting evaluation artifacts that inflate hallucination rates and by studying inference-time mitigation strategies.

**Defenses Against Package Hallucination.** Existing defenses against package hallucination fall into three broad families. First, knowledge-grounding methods supplement the model with verified package information. Spracklen et al. [34] showed that Retrieval-Augmented Generation (RAG) [20] substantially reduces hallucinations while largely preserving code quality. Second, post-hoc detection and correction methods validate or revise generated dependencies. Krishna et al. [17] proposed checking dependencies against time-aware package registries, while Self-Refine [24] iteratively critiques and revises outputs without additional training. However, smaller models often fail to detect their own hallucinations and may enter repetitive refinement loops [19]. Third, parameter-based methods modify the model itself. Supervised fine-tuning achieves strong reductions in package hallucination but requires training

data, computational resources, and model-specific checkpoint maintenance [34]. Lower-temperature decoding can also reduce hallucination frequency, although sampling-parameter changes alone are insufficient [34]. In contrast, decoding-time mitigation remains comparatively underexplored. Our work addresses this gap by evaluating guided decoding as an inference-time defense that requires no model retraining or external retrieval.

**Guided Decoding.** Guided decoding modifies token selection at inference time without retraining the model. Standard approaches include deterministic methods such as greedy and beam search, and stochastic methods such as top-$k$ sampling, nucleus sampling, and temperature scaling; their effectiveness depends on the task, model size, alignment, and quantization [33]. More recent methods use model outputs or internal representations to improve factuality. Contrastive Decoding contrasts an expert model with a smaller amateur model [21] and has also been shown to improve reasoning [26]. DoLa contrasts early and late transformer layers within a single model [3], while Active Layer-Contrastive Decoding (ALCD) learns when such contrasts should be applied [41]. Nudging instead uses a small aligned model to replace tokens when the base model is uncertain [10]. In code generation, guided and constrained decoding has been used to enforce syntax [12, 29], improve program structure through tree search [30], generate secure code through supervised co-decoding [14], and adjust line-level logits to promote essential control structures [22]. However, prior work has not systematically evaluated these techniques as defenses against package hallucination. We address this gap by studying Contrastive Decoding, DoLa, ALCD, and Nudging in this setting.

## 3 Research Questions

To the best of our knowledge, this is the first study to systematically evaluate guided decoding strategies as defenses against package hallucination and to compare them against both pre-generation and post-generation baselines. Our evaluation spans four programming languages (JavaScript, Python, Ruby, and Rust) across five model families, enabling an analysis of how ecosystem characteristics such as registry size, naming conventions, and package distribution interact with model behavior. We further assess the robustness of all mitigation strategies under adversarial prompts, where the user explicitly steers the model toward non-existent packages. Our study is organized around four research questions:

**RQ1 (Baseline Characterization).** *How does the package hallucination rate vary across model families and programming languages when evaluation accounts for standard-library imports?*

**RQ2 (Defense Effectiveness).** *How effective are guided decoding strategies in reducing package hallucination compared with pre-generation (RAG) and post-generation (Self-Refine) baselines?*

**RQ3 (Mitigation–Utility Trade-off).** *What trade-offs arise between the effectiveness of package-hallucination mitigation strategies and the utility of their generated package recommendations?*

**RQ4 (Adversarial Robustness).** *How robust are existing and proposed mitigation strategies when prompts are deliberately seeded with fabricated package names?*

## 4 Methodology

We evaluated open-weight models across five families, seven strategies, and four programming languages: Python, JavaScript, Ruby, and Rust. Hallucination detection uses enhanced ground-truth registry snapshots to mitigate false-positive biases, with each strategy tested on the complete dataset $\mathcal{D}_{\text{eval}}$ over three independent runs.

### 4.1 Dataset Construction

Following Spracklen et al. [34], we construct an evaluation dataset spanning four programming languages. **Python** and **JavaScript** are chosen for their popularity and the size of their package ecosystems (PyPI and npm, respectively). **Ruby** and **Rust** are chosen as languages with smaller, more structured, and easily verifiable package namespaces (RubyGems and Crates.io, respectively). This selection provides diversity in registry size, naming conventions, and ecosystem maturity. Our dataset construction pipeline consists of three steps.

**Step 1: Package Sampling.** For each language, we retrieved the top 1,000 most popular packages from the corresponding registries (PyPI, npm, RubyGems, and Crates.io) using the `Libraries.io` API, which provides unified access to package metadata across more than 36 package managers. Each retrieved record consists of a `(package_name, description)` pair, which serves as the seed for prompt generation in Step 3.

**Step 2: Ground-Truth Registry Snapshot.** To enable reliable hallucination detection, we collected the complete list of package names available in each registry as of March 4, 2026. We further augment each registry snapshot with the standard-library module list for the corresponding language. This augmentation is critical to our corrected evaluation methodology: without it, legitimate standard-library imports such as `os`, `math`, and `json` in Python would be erroneously flagged as hallucinated. The resulting ground-truth package set $\mathcal{P}_\ell$ for each language $\ell \in$ {Python, JavaScript, Ruby, Rust} is the union of registry packages and standard-library modules.

**Step 3: Prompt Generation.** Starting from each `(package_name, description)` pair, we used `GPT-4o-mini` to synthetically generate a natural-language coding instruction that reflects the package's functionality. The system prompt used for this step was:

> **System Prompt: Package Recommendation**
>
> You are a coding assistant that recommends packages useful to solve given problems. Respond with only a list of <language> packages, separated by commas, enclosed in square brackets, no additional text.
> **Example output:** `[package1, package2]`

This procedure yields a final evaluation dataset $\mathcal{D}_{\text{eval}}$ of 4,000 instructions, with $|\mathcal{D}_\ell| = 1{,}000$ for each language $\ell$.

### 4.2 Models

We evaluate eight *instruction-tuned*, open-weight language models from five families, spanning a range of parameter scales:

- **Gemma 3** (Google): `gemma-1b`, `gemma-4b`
- **DeepSeek-Coder** (DeepSeek): `deepseek-1.3b`, `deepseek-6.7b`
- **Qwen 2.5** (Alibaba): `qwen-1.5b`, `qwen-3b`
- **Mistral** (Mistral) : `mistral-7b` we used Instruct-v0.3.
- **Llama 3.1** (Meta) : `llama-8b`

We focus on lightweight models (1b–8b parameters) for two reasons. First, prior work has shown that package hallucination rates decrease with model scale, with larger models generally exhibiting greater resistance to fabricated recommendations [17, 34]. Smaller models are therefore a harder and more practically important test bed for mitigation strategies, since they are widely deployed on resource-constrained hardware. Second, several of our guided decoding strategies (Contrastive Decoding, Nudging) require running two models simultaneously, and limiting parameter counts keeps inference within the memory budget of a single GPU.

## 4.3 Hallucination Detection

For each `(model, language, strategy)` triple, the model is queried on the full dataset $\mathcal{D}_{\text{eval}}$ and asked to recommend packages. The raw outputs are parsed to extract the set of recommended package names. Each name is then checked against the ground-truth set $\mathcal{P}_\ell$, which includes both registry packages and standard-library modules. A recommended name is classified as hallucinated if and only if it is absent from $\mathcal{P}_\ell$.

We report the following metrics for each configuration

- **Micro PHR** (benchmark level). Computed over the full output of a given (model, language, strategy) configuration:

$$\text{micro-PHR} = \frac{N_{\text{hall}}}{N_{\text{gen}}}$$

- **Macro PHR** (sample level). Computed per prompt $i$ and then averaged:

$$\text{PHR}_i = \frac{n_{\text{hall},i}}{n_{\text{gen},i}}, \qquad \text{macro-PHR} = \frac{1}{|\mathcal{D}_{\text{eval}}|} \sum_i \text{PHR}_i$$

where $n_{\text{gen},i}$ and $n_{\text{hall},i}$ are the number of generated and hallucinated packages for prompt $i$, respectively. If $n_{\text{gen},i} = 0$, the prompt contributes 0 to the sum.

## 4.4 Package Utility Score (PU)

PHR measures the fraction of generated package names that are invalid but not their usefulness. A defense can lower PHR by producing fewer packages or suppressing recommendations. We introduce the Package Utility Score (PU), measuring whether generated packages are valid and task-relevant. For prompt $i$, let $G_i$ be the generated package set and $\mathcal{P}_\ell$ the language-specific universe of registry packages and standard-library modules. Since each prompt is derived from seed package $s_i$ and its description $d(s_i)$ [34], we construct a task-specific reference set $R_i \subseteq \mathcal{P}_\ell$ by computing cosine similarity between $d(s_i)$ and every verified same-language package description $d(p)$ using `text-embedding-3-small` embedder, then selecting $s_i$ and the $k-1$ most similar packages. Therefore, $R_i$ contains only valid packages and approximates those functionally related to the seed, crediting valid alternatives while excluding hallucinations. The useful generated set is $U_i = G_i \cap R_i \cap \mathcal{P}_\ell = G_i \cap R_i$, where the explicit intersection with $\mathcal{P}_\ell$ emphasizes validity because $R_i \subseteq \mathcal{P}_\ell$. Package precision and recall are defined as

$$\text{PU}_{\text{P}i} = \frac{|U_i|}{|G_i|}, \qquad \text{PU}_{\text{R}i} = \frac{|U_i|}{|R_i|}.$$

Precision is the fraction of generated dependencies in the reference set, whereas recall is the fraction of that set recovered. We combine them using an F1-style score:

$$\text{PU}_i = 2 \times \frac{\text{PU}_{\text{P}i} \times \text{PU}_{\text{R}i}}{\text{PU}_{\text{P}i} + \text{PU}_{\text{R}i}}.$$

We set $\text{PU}_i = 0$ when $G_i = \varnothing$ or both precision and recall are zero. That is, an empty output avoids hallucination but provides no package utility. Precision penalizes hallucinated or unrelated packages; recall penalizes output collapse. Because $R_i$ uses vetted registry metadata, PU is reproducible and independent of LLM-generated reference labels, but remains a semantic proxy rather than complete ground truth. To complement this, we also evaluate recommendation utility through generated code. Specifically, after removing hallucinated packages, we ask the same model to solve the original task using the recommended packages and count how many are imported or otherwise used. This operationally complements description similarity.

## 4.5 Baselines and Decoding Strategies

We compare five guided decoding strategies against three baselines.
**Baselines.** *Vanilla Decoding* uses the model's standard generation configuration, with temperature set to 1.0, top-$k$ set to 50, and top-$p$ set to 1.0, and serves as the non-deterministic reference setting. *Self-Refine* [24, 34] is a post-generation iterative correction approach where the model critiques and revises its own output over multiple passes, included as a representative post-hoc mitigation technique. *RAG* [20, 34] supplements the prompt with retrieved package information from the registry, included as a representative pre-generation grounding technique.
**Guided Decoding Strategies.** *Greedy Decoding* selects the highest-probability token at each step, yielding a more deterministic alternative. *Contrastive Decoding (CD)* [21, 26] generates tokens by subtracting the log-probability distribution of a smaller *amateur* model from that of the *expert* (target) model, penalizing tokens that are probable under both models and rewarding tokens specific to the expert. *DoLa* [3] computes a contrastive distribution between an early and a late transformer layer within the same model, amplifying factual knowledge encoded in deeper layers while suppressing surface patterns from shallower ones. *Active Layer-Contrastive Decoding (ALCD)* [41] is built upon DoLa and consisted of a selection of tokens where to apply the contrastive distribution decoding based on predefined policy. *Nudging* [10] uses a small aligned surrogate model to generate replacement tokens whenever the base model's confidence falls below a calibrated threshold, effectively steering generation toward more grounded outputs at points of high uncertainty.
**Experimental Configuration.** To ensure a fair comparison across strategies, we fix a single hyperparameter configuration per strategy and apply it uniformly across all eight models and four languages considered. This choice trades a small amount of per-configuration optimality for comparability: a strategy that wins under a shared configuration is evidence of genuine robustness rather than an artifact of grid search. Table 1 summarizes the configuration used for each strategy. Our choice of decoding hyperparameters is based on the previous study made by Shi et al. [33]. For RAG, we follow the retrieval configuration of Spracklen et al. [34], adopting their top-$k$ retrieval setup.

**Table 1: Hyperparameter configuration per strategy.**

| Strategy | Hyperparameters |
|---|---|
| Baseline (Vanilla) | $T$=1.0, top-$k$=50, top-$p$=1.0 |
| Greedy | $T$=0 |
| Contrastive Decoding (CD) | $\alpha$=0.5; amateur/expert layer pairing (model-specific) |
| DoLa | mature_layer = final layer of target model; early_exit_layers= {4, 8, 12, 16}; $\delta$=0.1; repetition_penalty= 1.2 |
| ALCD | mature_layer = final layer of target model; early_exit_layers= {4, 8, 12, 16}; $\delta$=0.1; repetition_penalty= 1.2; active decision policy => $entropy \geq 1$ |
| Nudging | surrogate_model = largest in family; top_prob_threshold= 0.4 |
| RAG | retrieval_$k$=5; corpus = per-language package registry |
| Self-Refine | max_refinement_iterations= 5 |

## 5 Experiments and Results

This section presents experimental results aligned with the four research questions from Section 3. We repeat each experiment three times and report the mean and standard deviation across runs. For large tables where space is limited, we report the mean in the main paper and provide the full per-run values and standard deviations in the artifact. We first quantify the impact of standard-library correction on package hallucination measurement. We then characterize the baseline performance across multiple models and languages (RQ1), compare inference-time defenses for package recommendation and code generation (RQ2), evaluate package utility across different reference sets (RQ3), and assess the robustness of defenses to adversarial prompts involving fabricated package names (RQ4).

### 5.1 The Impact of Standard-Library Correction

Prior evaluations can overestimate package hallucination by treating standard-library imports as invalid because they are absent from centralized registries. For example, Python modules such as `os` and `math` do not appear on PyPI despite being valid dependencies. To quantify this bias, we collected standard-library manifests for the two applicable languages and computed PHR with and without correction. Table 2 reports the overestimation $\Delta$ by language and strategy. For Python, excluding standard-library modules inflates PHR by up to 9.4 pp, compared with 2.5 pp for Ruby, reflecting Python's larger and more frequently imported standard library. RAG shows the smallest overestimation in both languages, consistent with its reliance on verified package information. Standard-library misclassification can also alter the relative ranking of mitigation strategies, affecting both measurement accuracy and comparative evaluation. We therefore apply standard-library correction throughout our evaluation.

### 5.2 Baseline Package Hallucination

We characterize the hallucination behavior of eight models under vanilla conditions by querying them on the full evaluation dataset $\mathcal{D}_{\text{eval}}$ and extracting recommended package names, validated against ground-truth registry snapshots $\mathcal{P}_\ell$. The result is shown in Table 3.

**Table 2: Micro PHR with and without standard library (stdlib) correction, averaged over all models. $\Delta > 0$ means overestimation without the correction.**

| Lang. | Strategy | $N_{\text{gen}}$ | $\text{PHR}_{\text{w/ stdlib}}$ (%) | $\text{PHR}_{\text{w/o stdlib}}$ (%) | $\Delta$ (pp) |
|---|---|---|---|---|---|
| Python | Vanilla | 79,965 | $24.7_{\pm 0.67}$ | $32.3_{\pm 0.53}$ | +7.6 |
| | Greedy | 47,796 | $19.3_{\pm 0.00}$ | $27.4_{\pm 0.00}$ | +8.1 |
| | S-Ref | 39,465 | $19.0_{\pm 1.14}$ | $28.4_{\pm 1.26}$ | **+9.4** |
| | CD | 24,489 | $26.7_{\pm 0.55}$ | $32.6_{\pm 0.50}$ | +5.9 |
| | DoLa | 51,540 | $32.3_{\pm 0.60}$ | $38.9_{\pm 0.58}$ | +6.6 |
| | Nudge | 17,628 | $21.3_{\pm 1.40}$ | $30.4_{\pm 0.91}$ | +9.1 |
| | RAG | 59,910 | $11.5_{\pm 0.04}$ | $14.3_{\pm 0.02}$ | +2.8 |
| | ALCD | 49,257 | $27.6_{\pm 0.71}$ | $34.8_{\pm 0.79}$ | +7.1 |
| Ruby | Vanilla | 69,072 | $38.2_{\pm 0.89}$ | $40.4_{\pm 0.96}$ | +2.2 |
| | Greedy | 39,771 | $30.3_{\pm 0.00}$ | $32.8_{\pm 0.00}$ | **+2.5** |
| | S-Ref | 34,596 | $31.9_{\pm 0.29}$ | $34.1_{\pm 0.32}$ | +2.2 |
| | CD | 16,596 | $40.9_{\pm 0.53}$ | $42.6_{\pm 0.58}$ | +1.7 |
| | DoLa | 47,223 | $46.6_{\pm 0.39}$ | $49.0_{\pm 0.38}$ | +2.4 |
| | Nudge | 19,446 | $38.0_{\pm 0.87}$ | $40.3_{\pm 1.09}$ | +2.2 |
| | RAG | 61,260 | $11.7_{\pm 0.03}$ | $13.3_{\pm 0.01}$ | +1.6 |
| | ALCD | 43,110 | $39.3_{\pm 0.21}$ | $41.8_{\pm 0.20}$ | **+2.5** |

**PHR varies across models and languages.** Mistral-7b exhibits the lowest Overall micro PHR (16.1%), while Qwen-1.5b achieves the highest (47.9%). Scaling from smaller to larger models correlates with a reduction in average micro PHR: DeepSeek shows a decrease of 18.3 pp (39.2% to 20.9%), Gemma 19.8 pp (38.2% to 18.4%), and Qwen 20.3 pp (47.9% to 27.6%). Among programming languages, Rust has the highest micro PHR of 63.1% on Qwen-1.5b, followed by Ruby on Gemma-1b (53.9%), indicating that more than half of the recommended packages are hallucinated. Javascript has the least micro PHR of 7.6%.

**Package-generation volume varies across models.** The mean number of packages generated per prompt (macro) ranges from approximately 1 for DeepSeek-1.3b to 12 for Llama-8b. Overall, Qwen-1.5b produces the most packages (100,896), including 48,330 hallucinations, while DeepSeek-1.3b produces the fewest. Micro- and macro-PHR capture complementary perspectives. Micro-PHR measures hallucination across all generated packages at the benchmark level and therefore gives greater weight to prompts producing longer recommendation lists. Macro-PHR instead averages per-prompt rates, reflecting hallucination for a typical prompt regardless of output length. For most configurations, micro-PHR exceeds macro-PHR, indicating that high-volume prompts are also more hallucination-prone. The largest gaps occur for Qwen-1.5b on Rust (+23.3 pp), DeepSeek-1.3b on Ruby (+23.2 pp), and Gemma-1b on Ruby (+22.7 pp). Reporting both metrics therefore distinguishes aggregate package-level exposure from typical prompt-level behavior.

**RQ1 Summary**

Package hallucination is pervasive across all eight models and four languages Rust and Ruby are challenging for the different studied models and scaling from smaller to larger variants reduces hallucination by 16–21 pp within each family. Yet no model is hallucination-free.

**Table 3: Vanilla decoding PHR per model and language. $\bar{n}$ = mean per prompt. *Overall* is column aggregates of all languages and for micro and macro PHR, it is the mean. Higher PHR per model is in bold.**

| Model | Lang. | $N_{\text{gen}}$ | $N_{\text{hall}}$ | micro PHR (%) | $n_{\text{gen}}$ | $n_{\text{hall}}$ | macro PHR (%) |
|---|---|---|---|---|---|---|---|
| deepseek-1.3b | Python | 3,024 | 1,077 | $35.6_{\pm 1.16}$ | 2.02 | 0.72 | $20.7_{\pm 1.64}$ |
| | JavaScript | 1,950 | 585 | $30.0_{\pm 5.49}$ | 1.30 | 0.39 | $13.9_{\pm 1.77}$ |
| | Ruby | 1,470 | 600 | $40.8_{\pm 3.35}$ | 0.98 | 0.40 | $17.6_{\pm 1.20}$ |
| | Rust | 2,658 | 1,305 | $\mathbf{49.1}_{\pm 0.55}$ | 1.78 | 0.87 | $\mathbf{35.4}_{\pm 0.23}$ |
| | *Overall* | 9,102 | 3,567 | $39.2_{\pm 0.55}$ | 4.55 | 1.78 | $21.9_{\pm 0.23}$ |
| deepseek-6.7b | Python | 5,013 | 630 | $12.6_{\pm 1.77}$ | 3.34 | 0.42 | $10.6_{\pm 0.77}$ |
| | JavaScript | 4,647 | 474 | $10.2_{\pm 2.68}$ | 3.10 | 0.32 | $9.5_{\pm 1.03}$ |
| | Ruby | 5,850 | 1,428 | $24.4_{\pm 1.66}$ | 3.90 | 0.95 | $15.2_{\pm 1.11}$ |
| | Rust | 2,970 | 1,329 | $\mathbf{44.8}_{\pm 0.88}$ | 1.98 | 0.89 | $\mathbf{45.7}_{\pm 1.42}$ |
| | *Overall* | 18,480 | 3,861 | $20.9_{\pm 0.18}$ | 9.24 | 1.93 | $20.2_{\pm 0.62}$ |
| gemma-1b | Python | 9,780 | 2,358 | $24.1_{\pm 1.62}$ | 3.26 | 0.79 | $18.4_{\pm 0.38}$ |
| | JavaScript | 12,948 | 4,362 | $33.7_{\pm 0.82}$ | 4.32 | 1.45 | $18.9_{\pm 1.10}$ |
| | Ruby | 11,256 | 6,063 | $\mathbf{53.9}_{\pm 0.84}$ | 3.75 | 2.02 | $\mathbf{31.2}_{\pm 0.36}$ |
| | Rust | 11,130 | 4,443 | $39.9_{\pm 0.99}$ | 3.71 | 1.48 | $28.5_{\pm 1.49}$ |
| | *Overall* | 45,114 | 17,226 | $38.2_{\pm 0.42}$ | 11.28 | 4.31 | $24.2_{\pm 0.71}$ |
| gemma-4b | Python | 9,894 | 1,200 | $12.1_{\pm 0.87}$ | 3.30 | 0.40 | $12.3_{\pm 0.68}$ |
| | JavaScript | 10,866 | 1,590 | $14.6_{\pm 0.81}$ | 3.62 | 0.53 | $16.3_{\pm 1.08}$ |
| | Ruby | 10,290 | 2,904 | $\mathbf{28.2}_{\pm 2.29}$ | 3.43 | 0.97 | $\mathbf{26.9}_{\pm 0.70}$ |
| | Rust | 10,908 | 2,022 | $18.5_{\pm 0.29}$ | 3.64 | 0.67 | $17.0_{\pm 0.85}$ |
| | *Overall* | 41,958 | 7,716 | $18.4_{\pm 0.56}$ | 10.49 | 1.93 | $18.1_{\pm 0.17}$ |
| llama-8b | Python | 17,016 | 4,176 | $24.5_{\pm 0.95}$ | 11.34 | 2.78 | $20.4_{\pm 0.26}$ |
| | JavaScript | 17,796 | 4,362 | $24.5_{\pm 0.40}$ | 11.86 | 2.91 | $23.0_{\pm 0.62}$ |
| | Ruby | 13,344 | 4,737 | $35.5_{\pm 1.16}$ | 8.90 | 3.16 | $\mathbf{31.7}_{\pm 1.24}$ |
| | Rust | 12,378 | 4,674 | $\mathbf{37.8}_{\pm 0.19}$ | 8.25 | 3.12 | $30.8_{\pm 1.14}$ |
| | *Overall* | 60,534 | 17,949 | $29.7_{\pm 0.40}$ | 30.27 | 8.97 | $26.5_{\pm 0.36}$ |
| mistral-7b | Python | 4,821 | 597 | $12.4_{\pm 0.38}$ | 3.21 | 0.40 | $9.1_{\pm 0.14}$ |
| | JavaScript | 4,713 | 360 | $7.6_{\pm 0.42}$ | 3.14 | 0.24 | $7.4_{\pm 0.37}$ |
| | Ruby | 4,863 | 1,212 | $\mathbf{24.9}_{\pm 1.00}$ | 3.24 | 0.81 | $\mathbf{21.0}_{\pm 0.08}$ |
| | Rust | 5,742 | 1,071 | $18.6_{\pm 3.70}$ | 3.83 | 0.71 | $16.3_{\pm 0.75}$ |
| | *Overall* | 20,139 | 3,240 | $16.1_{\pm 0.06}$ | 10.07 | 1.62 | $13.5_{\pm 0.27}$ |
| qwen-1.5b | Python | 23,553 | 8,280 | $35.1_{\pm 1.60}$ | 7.85 | 2.76 | $21.1_{\pm 0.84}$ |
| | JavaScript | 28,320 | 11,454 | $40.4_{\pm 3.33}$ | 9.44 | 3.82 | $23.8_{\pm 0.98}$ |
| | Ruby | 16,686 | 8,181 | $49.0_{\pm 3.25}$ | 5.56 | 2.73 | $35.3_{\pm 0.94}$ |
| | Rust | 32,337 | 20,415 | $\mathbf{63.1}_{\pm 1.15}$ | 10.78 | 6.80 | $\mathbf{39.8}_{\pm 1.76}$ |
| | *Overall* | 100,896 | 48,330 | $47.9_{\pm 1.65}$ | 25.22 | 12.08 | $30.0_{\pm 0.97}$ |
| qwen-3b | Python | 6,864 | 1,458 | $21.2_{\pm 2.36}$ | 2.29 | 0.49 | $20.6_{\pm 0.23}$ |
| | JavaScript | 6,348 | 1,662 | $26.2_{\pm 2.28}$ | 2.12 | 0.55 | $25.4_{\pm 0.52}$ |
| | Ruby | 5,313 | 1,284 | $24.2_{\pm 2.15}$ | 1.77 | 0.43 | $23.6_{\pm 0.63}$ |
| | Rust | 11,676 | 3,936 | $\mathbf{33.7}_{\pm 1.96}$ | 3.89 | 1.31 | $\mathbf{25.6}_{\pm 0.63}$ |
| | *Overall* | 30,201 | 8,340 | $27.6_{\pm 0.57}$ | 7.55 | 2.08 | $23.9_{\pm 0.80}$ |

## 5.3 Impact of Guided Decoding Defenses

Here, we evaluate the effect of five guided decoding strategies (Greedy, Contrastive Decoding, DoLa, Nudging, and ALCD) and three baselines Vanilla decoding, RAG, and Self-Refine across all eight models and the four programming languages considered. Table 4 shows the results. We observe that no single strategy dominates uniformly, indicating that hallucination mitigation is multifactorial and strongly conditioned on both model family and language ecosystem.

**RAG performs well for Python, Ruby, and Rust but degrades performance for JavaScript.** RAG achieves the lowest PHR in 19 of 32 configurations and reduces the cross-model average from 29.7% under Vanilla decoding to 18.1%, the largest overall improvement among the evaluated strategies. It lowers PHR by an average of 10.7 pp for Python, 23.4 pp for Ruby, and 24.9 pp for Rust, without increasing PHR for any model in these languages. In contrast, RAG increases PHR for 7 of 8 models on JavaScript, with an average degradation of 12.7 pp. This suggests that the npm retrieval index or retrieval configuration requires careful validation before deployment.

**DoLa performs poorly on DeepSeek but remains competitive for some model families.** DoLa has the highest average PHR among the guided decoding strategies, reaching 38.0% overall. Its performance deteriorates substantially on DeepSeek models, with DeepSeek-1.3b producing PHR values between 83.5% and 91.2%. However, DoLa remains moderately effective for models such as Mistral-7b and Qwen-3b, demonstrating that its effectiveness is architecture-dependent.

**Self-Refine is most effective for Llama-8B.** Self-Refine achieves the lowest PHR across all four languages for Llama-8B, reducing PHR by 8–18 pp relative to Vanilla decoding. Its performance is less consistent for other models, where it frequently ranks among the weaker strategies. This suggests that successful self-correction depends on the capabilities of the underlying model rather than on architecture alone.

**Nudging serves as the most dependable strategy for JavaScript.** Where RAG performs poorly on JavaScript, Nudging consistently reduces PHR for Gemma-1b, Qwen-1.5b, Qwen-3b, and Llama-8b, making it the strongest strategy for this language overall. However, it degrades performance on DeepSeek models, further demonstrating that guided decoding strategies are not uniformly transferable across model families.

**CD and ALCD provide limited gains, while Greedy decoding is a strong fallback.** CD fails to achieve the lowest PHR in any configuration, while ALCD generally underperforms against Vanilla. In contrast, Greedy decoding matches Self-Refine's average PHR (25.6%) and provides substantial reductions over Vanilla, making it a recommended fallback strategy.

## 5.4 Extension to Full Code Generation, Syntax Validity and Code Smells

In our previous experiments in Sections 5.1, 5.2, and 5.3, we prompted each LLM to recommend a set of useful packages for every instruction in $\mathcal{D}_{\text{eval}}$. We extend this analysis to full code generation by prompting each model to produce complete, runnable code for the same instructions. All imported packages are validated against the same registry snapshots. The system prompt for this task was:

System Prompt: Code Generation

> You are a helpful assistant that generates <language> code for a given task. Respond with code inside a "'<language> ... "' block when appropriate.

We exclude RAG from this experiment because, in our setup, it affects only package recommendation and does not modify the subsequent code-generation process. Figures 1a and 1b show the results of valid and hallucinated packages in generated code by language and model. At the model level (Figure 1b), Self-Refine produces the most package references but increases both valid and hallucinated counts, indicating that its gains in package recommendation do

**Table 4: Micro PHR (%) per model, language, and strategy on the package recommendation task. Bold = lowest (best) per row.**

| Model | Lang. | Vanilla | Greedy | Self-Refine | CD | DoLa | Nudge | RAG | ALCD |
|---|---|---|---|---|---|---|---|---|---|
| deepseek-1.3b | Python | 35.6 | 44.3 | 36.2 | 44.1 | 87.3 | 42.7 | **18.3** | 66.3 |
| | JavaScript | 30.0 | 38.8 | **20.0** | 38.5 | 83.5 | 42.8 | 41.5 | 70.0 |
| | Ruby | 40.8 | 52.9 | 50.9 | 56.8 | 91.2 | 69.4 | **22.3** | 73.7 |
| | Rust | 49.1 | 70.7 | 40.4 | 58.3 | 88.9 | 81.3 | **24.6** | 77.4 |
| deepseek-6.7b | Python | 12.6 | 12.6 | 26.4 | 46.6 | 79.2 | 33.1 | **9.4** | 41.4 |
| | JavaScript | **10.2** | **10.2** | 15.6 | 40.6 | 88.8 | 32.0 | 36.0 | 46.6 |
| | Ruby | 24.4 | 24.4 | 29.2 | 49.6 | 87.7 | 63.7 | **6.9** | 52.6 |
| | Rust | 44.8 | 44.8 | 39.2 | 49.7 | 54.1 | 74.3 | **7.3** | 49.6 |
| gemma-1b | Python | 24.1 | 16.2 | 18.4 | 18.9 | 17.2 | 11.9 | **7.9** | 21.1 |
| | JavaScript | 33.7 | **13.2** | 32.0 | 23.9 | 17.1 | 14.4 | 60.9 | 22.2 |
| | Ruby | 53.9 | 33.4 | 30.0 | 37.1 | 39.3 | 30.2 | **3.5** | 40.3 |
| | Rust | 39.9 | 31.1 | 36.7 | 38.9 | 35.3 | 24.8 | **2.9** | 36.7 |
| gemma-4b | Python | 12.1 | 12.5 | 10.8 | 11.6 | 16.9 | 11.9 | **6.7** | 16.7 |
| | JavaScript | 14.6 | 12.5 | 14.4 | 15.0 | 20.3 | **11.7** | 12.0 | 14.6 |
| | Ruby | 28.2 | 24.4 | 23.6 | 20.1 | 42.4 | 23.9 | **8.7** | 29.2 |
| | Rust | 18.5 | 17.8 | 18.6 | 19.2 | 22.7 | 14.5 | **8.6** | 17.0 |
| llama-8b | Python | 24.5 | 24.5 | **10.8** | 21.6 | 20.9 | 11.4 | 18.9 | 21.9 |
| | JavaScript | 24.5 | 24.5 | **9.9** | 18.2 | 22.3 | 12.7 | 28.6 | 22.4 |
| | Ruby | 35.5 | 35.5 | **17.6** | 24.6 | 36.2 | 19.0 | 22.7 | 35.6 |
| | Rust | 37.8 | 37.8 | **27.3** | 33.5 | 33.4 | 28.9 | 28.7 | 34.1 |
| mistral-7b | Python | 12.4 | 14.6 | 10.3 | 13.4 | 10.1 | 11.7 | 11.3 | **9.9** |
| | JavaScript | 7.6 | 7.6 | 9.4 | 12.3 | **7.0** | 10.4 | 19.9 | 7.6 |
| | Ruby | 24.9 | 24.6 | 25.4 | 18.4 | 20.2 | 17.2 | **16.5** | 20.3 |
| | Rust | 18.6 | 23.4 | 20.0 | 17.2 | 15.9 | 19.2 | 20.8 | **14.4** |
| qwen-1.5b | Python | 35.1 | 14.9 | 16.9 | 24.2 | 13.2 | 15.7 | **7.7** | 28.8 |
| | JavaScript | 40.4 | 18.0 | 22.6 | 28.7 | 16.8 | **12.3** | 47.5 | 28.6 |
| | Ruby | 49.0 | 28.5 | 46.5 | 35.6 | 32.1 | 20.5 | **7.4** | 40.1 |
| | Rust | 63.1 | 31.7 | 48.2 | 46.3 | 34.3 | 23.8 | **5.1** | 55.4 |
| qwen-3b | Python | 21.2 | 15.3 | 22.3 | 14.9 | 13.5 | 12.5 | **11.5** | 15.0 |
| | JavaScript | 26.2 | 21.1 | 20.9 | 20.8 | 24.4 | **12.0** | 42.5 | 22.7 |
| | Ruby | 24.2 | 18.8 | 32.1 | 25.4 | 23.2 | 20.4 | **5.7** | 22.4 |
| | Rust | 33.7 | 19.9 | 36.8 | 28.9 | 20.3 | 18.9 | **8.1** | 27.3 |

not transfer reliably to code generation. Mistral-7b produces the fewest references while maintaining the most favorable valid-to-hallucinated ratio, reflecting more conservative dependency use. Nudging preserves more valid references than Vanilla across models, whereas CD reduces both valid and hallucinated references, suggesting output suppression. At the language level (Figure 1a), Rust exhibits the highest hallucination volume, nearly matching its valid-reference count. For JavaScript, Self-Refine substantially increases hallucinated references, while DoLa produces very few references overall. Python is the most controlled setting, with Nudging maintaining a clearer separation between valid and hallucinated references.

**Syntax Validity and Code Smells.** Beyond package hallucination, we assess the structural quality of generated code using Semgrep, measuring two complementary metrics: ❶ *syntax error rate* (fraction of programs that could not be parsed due to a syntax error) and ❷ *code smell rate* (fraction exhibiting Semgrep-detected anti-patterns). Results are reported in Figure 2a and 2b. We observe that Vanilla and Greedy decoding ensure nearly perfect syntactic validity across six models, while Self-Refine maintains clean syntax despite being the least effective for code smells. Guided strategies, however, consistently introduce syntax errors, particularly in Rust and JavaScript. Notably, *Nudging* exhibits the most heterogeneous syntax degradation, with DeepSeek-6.7b showing a 29% syntax error rate for Rust. In contrast, Gemma-4b stands out by producing 0% syntax errors across all languages analyzed. For code smells, DoLa, Nudging, and CD yield nearly zero smell counts across most model-language combinations, whereas Vanilla and Self-Refine are significant sources of anti-patterns. Vanilla decoding sees the highest smells in Rust models, while Self-Refine exacerbates these issues, particularly in Gemma-4b for Ruby. Overall, these findings suggest that certain strategies can lead to both syntax errors and code smells, highlighting the importance of the modeling approach used.

**RQ2 Summary**

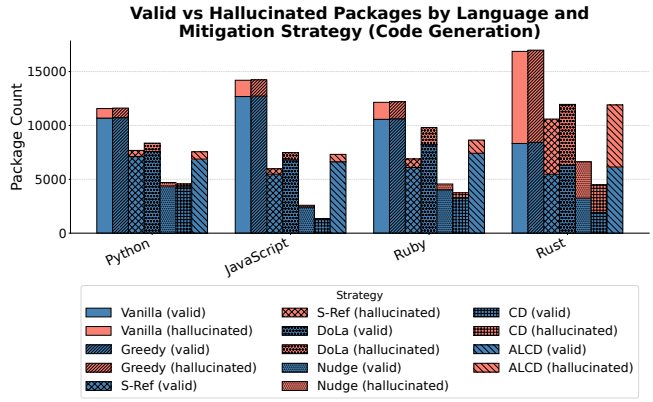
No single strategy dominates across models, languages, and tasks. RAG achieves the lowest PHR in 19/32 package-recommendation configurations for Python, Ruby, and Rust but degrades JavaScript by 12.7 pp on average; Self-Refine is strongest for Llama-8B. In code generation, Vanilla and Greedy preserve syntax, whereas guided strategies can introduce parsing errors but often reduce code smells. Strategy selection must therefore consider the model family, language, task, and broader code quality.

## 5.5 Package Utility of Guided Decoding

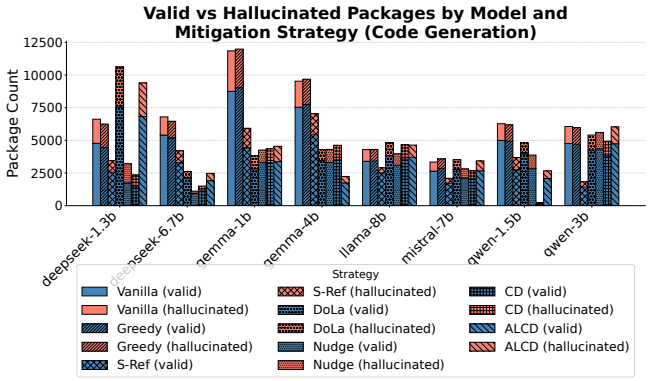
We assess whether model-recommended packages are useful for the target task, rather than merely valid. We measure this using Package Utility (PU) (see Section 4.4), a precision–recall metric computed against a task-specific reference set $R$. We consider two reference-set constructions: ❶ the top-$k$ packages whose description is most similar to the seed package in terms of cosine similarity, where $k \in \{1, 2, 5, 10\}$, and ❷ a code-based reference set containing packages obtained in the code generation experiment in Section 5.4. Figures 3a–3c report the micro-averaged precision ($PU_P$), recall ($PU_R$), and overall (PU) scores.

**Precision is insensitive to $k$ while recall is not.** Across strategies and models, micro $PU_P$ increases by marginally 1–2 percentage points from Top-1 to Top-10 (mean values from 13.1 to 14.8), despite the tenfold growth of the reference set. This suggests that models recommend a limited set of packages based on training distribution rather than by valid reference coverage, indicating that precision reflects generation selectivity. In contrast, micro $PU_R$ decreases from a mean of 23.4 at Top-1 to only 2.7 at Top-10, an 8.6-fold reduction. This trend indicates that models capture a narrow scope of semantically proximate packages, with no strategy achieving a mean recall above 8 at Top-10.

**Reference based on generated code packages set enhances utility recovery.** The generated code package reference results in significantly higher metric values, with mean PU rising to 21.0 from 4.3 at Top-10 and 15.5 at Top-1. This improvement reflects the alignment between the packages obtained from the generated code given a prompt sets and the recommended packages list provided by the model given the same prompt, while establishing that this approach provide informative operational definition of PU.

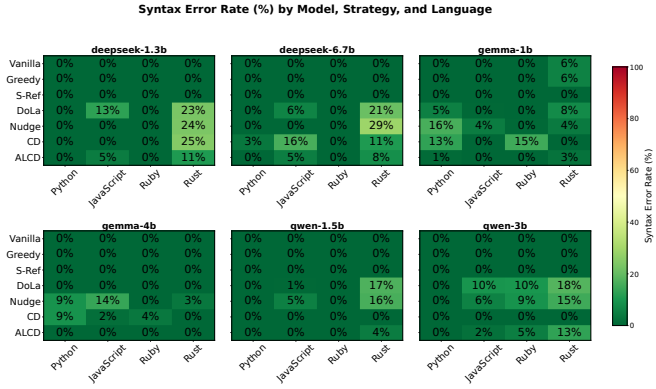



(a) Distribution of valid vs. hallucinated packages by language and strategy for full code generation.

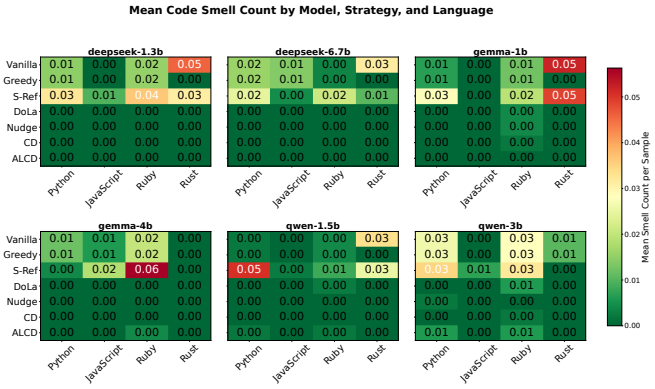



(b) Distribution of valid vs. hallucinated packages by model and strategy for full code generation.

**Figure 1: Valid and hallucinated package references extracted from full code generation outputs, aggregated by language (left) and by model (right) across all seven strategies. Blue segments denote registry-validated packages; red segments denote hallucinated references.**




(a) Syntax error rate (%) per model, strategy, and language.




(b) Code smell rate (%) per model, strategy, and language.

**Figure 2: Structural quality of generated code per model, strategy, and language. Left: syntax error rate (%); green cells denote error-free output, warmer colors indicate increasing parse failure rates. Right: mean Semgrep-detected code smell count per sample.**

**DeepSeek utility collapse confirmed across all definitions.** Both DeepSeek variants show near-zero PU under all tested conditions, reinforcing previous findings of output suppression. Conversely, Baseline and Greedy on DeepSeek-6.7b achieve a collective PU of 30.2, indicating retained latent utility.

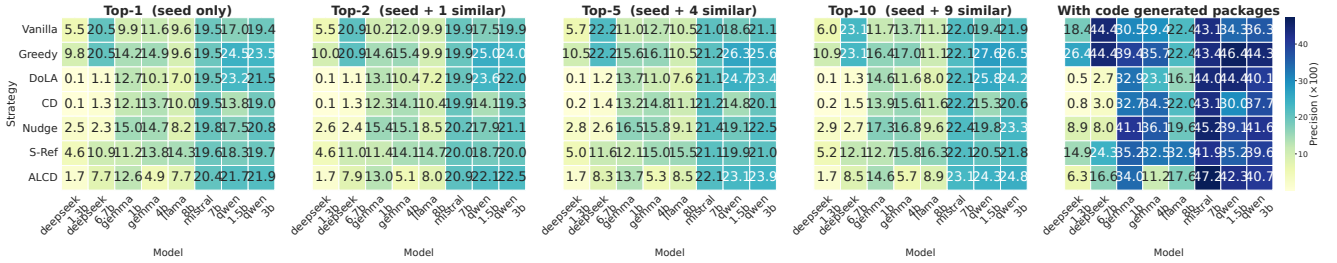
**Mitigation–utility trade-off.** Among strategies evaluated on both PHR and PU, Greedy provides the strongest average trade-off, reducing mean PHR from 29.7% under Vanilla to 25.6% while attaining the highest PU under all five reference-set definitions. Self-Refine ranks second followed by Nudging. CD, DoLa, and ALCD do not reduce average PHR relative to Vanilla and therefore provide an unfavorable overall trade-off.

**RQ3 Summary**

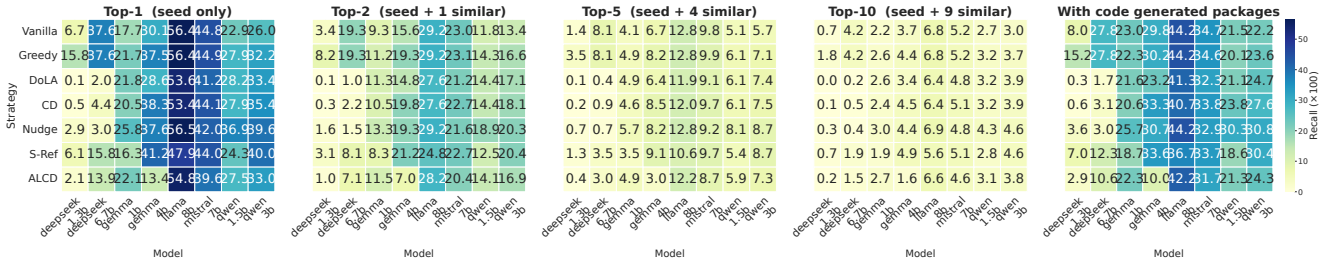
Package utility depends strongly on reference-set construction. Expanding the similarity-based set from Top-1 to Top-10 leaves precision nearly unchanged but reduces recall drastically from 23.4 to 2.7 and mean PU from 15.5 to 4.3. Code-derived references yield the highest mean PU, providing a more operational measure of utility.

## 5.6 Adversarial Robustness

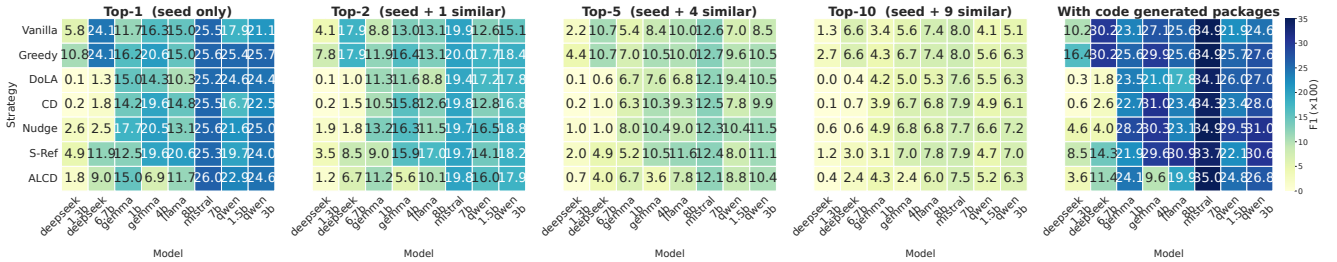
A defense that works only under benign instructions offers limited practical assurance [5, 7, 27, 32]. In this section we evaluate all mitigation strategies under adversarial prompts where the user explicitly directs the model to install and use fabricated packages.




**(a) Package Utility Precision** $\mathrm{PU_P}$ **(×100).**




**(b) Package Utility Recall** $\mathrm{PU_R}$ **(×100).**




**(c) Package Utility** PU **(×100).**

**Figure 3: Package Utility (PU) across five reference-set definitions for all seven strategies and eight models on the package recommendation task. Values are multiplied by 100 for readability. Rows: strategies; columns: models.**

**Adversarial Prompt Construction.** We collected 1,000 non-existent package names for each Python and JavaScript language produced by both open-source (Meta-Llama, Alibaba families) and closed-source (OpenAI GPT family) LLMs when prompted on coding tasks. Each candidate was verified as absent from the npm and PyPI registries via direct lookup. These names cover a broad spectrum of hallucination patterns including completely invented identifiers, cross-ecosystem confusions, fuzzy variants of real names, standard-library misattributions, and case or naming-convention mismatches. To extend coverage to Ruby and Rust, we reused these npm- and PyPI-origin names as adversarial seeds. This choice is deliberate: cross-ecosystem confusion is itself one of the most prevalent hallucination patterns, and a model asked to use a JavaScript-origin name in a Ruby context may hallucinate a plausible gem rather than refuse. Each name was inserted into a language-specific instruction template directing a coding assistant to install and use the corresponding package, yielding 1,000 adversarial prompts per language (4,000 total).

**Impact of Adversarial Prompting.** Table 5 shows that adversarial prompts substantially increase PHR relative to the vanilla baselines. Averaged across the four languages, we observe an increment ranging from 20.6 pp The effect is particularly severe for models that perform well under benign prompts. Gemma-4b, whose vanilla PHR ranges from 12.1% to 28.2%, reaches 49.7%–90.2% under attack. Similarly, Mistral-7b, the strongest model under vanilla decoding (7.6%–24.9%), rises to 46.6%–76.6%, an average increase of 47.8 pp. Thus, strong performance under benign prompts provides limited protection once the prompt itself is adversarially framed. Qwen-1.5b shows the smallest increase. This can be attributed to its vanilla PHR already being the highest in the study (35.1–63.1%); on Rust, its adversarial PHR (51.4%) is in fact lower than its vanilla rate (63.1%), the single case where adversarial framing does not increase hallucination. Ruby is the most vulnerable language and Rust the most resistant in each of the eight models we test, with no exception. Adversarial PHR under vanilla decoding spans 46.6% (Mistral-7b, Rust) to 92.5% (DeepSeek-1.3b, Ruby). This universality suggests that Ruby's smaller package ecosystem provides weaker negative-evidence anchors in pre-training corpora regardless of model family or scale, while Rust's strict compiler-enforced naming conventions consistently constrain fabrication.

**Effectiveness of Mitigation Strategies.** Table 5 shows that no single strategy dominates across all eight models, although a scale-dependent pattern emerges. RAG performs best overall, achieving the lowest PHR in 16 of 32 model-language combinations (50%). It is particularly effective for lower-capacity models, producing the lowest PHR on 3 of 4 languages for DeepSeek-1.3b (Python: 48.6%, JavaScript: 71.8%, Ruby: 87.2%) and Qwen-1.5b (Python: 24.5%, JavaScript: 46.9%, Ruby: 64.2%), on all 4 languages for DeepSeek-6.7b, and on 2 of 4 for Gemma-1B (Python: 39.0%, JavaScript: 49.2%). Retrieved registry evidence therefore appears especially useful for models with limited self-correction capacity. The JavaScript degradation observed under benign prompts (Section 5.3) is absent under attack, where RAG achieves the lowest JavaScript PHR for 5 of 8 models. Self-Refine performs more consistently on models with stronger self-correction capabilities, yielding the lowest PHR in 13 of 32 combinations. It ranks first across all four languages for Llama-8B, three of four for Mistral-7b, and for Gemma-4b on JavaScript (31.0%), Ruby (69.8%), and Rust (38.6%). It also performs best for Qwen-3b on Ruby (57.9%) and Rust (44.3%). The lowest PHR observed in the study is 11.4%, achieved by Gemma-4b on Python with RAG. This result indicates that retrieval grounding can provide strong resistance to adversarial package injection even for a 4b model. Together, RAG and Self-Refine produce the lowest PHR in 29 of 32 combinations. Nudging performs best for DeepSeek-1.3b and Gemma-1b on Rust. For Gemma-1B on Ruby, no mitigation strategy improves upon the baseline PHR of 83.7%. This result shows that, for small models operating in under-represented package ecosystems, the evaluated interventions may fail to reduce hallucination and can instead increase it.

**Decoding-only strategies provide inconsistent protection under adversarial prompting.** DoLa increases PHR relative to Greedy in 24 of 32 model–language configurations (75.0%). The largest degradation occurs for DeepSeek-1.3B on Rust, where PHR rises from 38.6% under Greedy to 94.1% under DoLa, an increase of 55.5 pp. Similar failures occur for Gemma-4b on Rust, with PHR increasing from 49.5% to 87.1% (+37.6 pp), and for Mistral-7b on Rust, where it increases from 46.1% to 81.3% (+35.2 pp). DoLa underperforms Greedy across all four languages for Gemma-1b, Qwen-3b, Llama-8b, and Mistral-7b. Most improvements are concentrated in the DeepSeek family: both DeepSeek-1.3b and DeepSeek-6.7b outperform Greedy in three of four languages, suggesting that DoLa's layer selection may be better aligned with DeepSeek's internal representations. ALCD mitigates some of these failures by applying layer contrast selectively. It outperforms DoLa in 23 of 32 configurations, including reductions from 81.6% to 51.0% for Llama-8b on Rust and from 49.8% to 36.5% for Qwen-1.5b on Rust. ALCD also improves upon DoLa across all four languages for DeepSeek-6.7b, Qwen-1.5b, and Qwen-3b. However, it does not achieve the lowest PHR in any configuration and improves upon the unmitigated Vanilla baseline in only 6 of 32 cases, all involving DeepSeek-1.3b, Gemma-4B, or Qwen-1.5b on Rust. Thus, ALCD reduces DoLa's degradation but remains an unreliable adversarial defense. Nudging and CD also produce mixed results. Nudging improves upon Greedy across all four languages for DeepSeek-1.3b, including a reduction from 38.6% to 26.6% on Rust (−12.0 pp). It also reduces PHR for Gemma-1b on Rust from 75.0% to 47.6% (−27.4 pp). In contrast, PHR increases from 63.1% to 72.5% for Qwen-1.5b on JavaScript (+9.4 pp) and from 76.8% to 84.8% for Gemma-1b on JavaScript (+8.0 pp). These results confirm that Nudging remains model- and language-dependent. CD does not achieve the lowest PHR in any configuration and consistently trails RAG and Self-Refine. Greedy remains close to the Vanilla baseline in most cases, indicating that deterministic decoding alone provides limited adversarial robustness.

**Why decoding-only defenses fail under adversarial prompts.** Decoding-only methods operate on local token probabilities or internal representations, but they do not verify whether a package exists. When the prompt contains a fabricated package, the fake name becomes a strong contextual anchor. Without external registry evidence or explicit self-verification, guided decoding may preserve or even amplify plausible-looking package names rather than reject them. This explains why RAG and Self-Refine are more robust under adversarial prompts. Specifically, RAG introduces external registry grounding, while Self-Refine gives the model an explicit opportunity to reconsider generated dependencies.

**RQ4 Summary**

No defense is universal under adversarial prompting. Adversarial prompts substantially increase PHR. Ruby is consistently the most vulnerable language, while Rust is the most resistant. RAG and Self-Refine achieve the lowest PHR in 29 of 32 configurations, with RAG favoring lower-capacity models and Self-Refine stronger models. Decoding-only defenses remain unreliable under adversarial prompting.

## 6 Limitations and Threats to Validity

**Benchmark realism and generalizability.** We extended the evaluation dataset from [34] to Ruby and Rust synthetically, with coding instructions generated by GPT-4o-mini from registry (package_name, description) pairs. However, no human validation or comparison against authentic developer queries such as those found on Stack Overflow, GitHub Issues, or coding-assistant logs has been conducted. Nonetheless, we believe this controlled construction provides consistent coverage across languages and enables reproducible comparisons.

**Model and ecosystem coverage.** Our evaluation covers eight small open-weight models from five families and four programming-language ecosystems. This focus provides a challenging and practically relevant setting, but the findings may not generalize to larger models, other programming languages, or private registries.

**Configuration sensitivity.** We apply one fixed hyperparameter configuration per strategy across all models and languages to support controlled comparison and do not perform optimization. Consequently, the reported rankings should be interpreted as results under the evaluated configurations rather than as upper bounds on each strategy's performance.

## 7 Conclusion

This paper revisited package hallucination in LLM-generated code from four angles: measurement, mitigation, utility, and adversarial robustness. Our corrected evaluation framework, which accounts

**Table 5: Micro PHR (%) and raw counts ($N_{\text{hall}}/N_{\text{gen}}$) in parentheses per model, language, and strategy under adversarial prompts. Lowest PHR per row in bold.**

| Model | Lang. | Vanilla | Greedy | Self-Refine | DoLa | RAG | Nudge | CD | ALCD |
|---|---|---|---|---|---|---|---|---|---|
| deepseek-1.3b | JavaScript | 84.4 (320/379) | 87.4 (340/389) | 75.2 (194/258) | 84.9 (253/298) | **71.8** (492/685) | 76.9 (230/299) | 87.7 (193/220) | 76.1 (242/318) |
| | Python | 80.3 (392/488) | 90.4 (339/375) | 81.2 (234/288) | 68.5 (204/298) | **48.6** (470/968) | 80.0 (419/524) | 93.9 (845/900) | 73.6 (245/333) |
| | Ruby | 92.5 (368/398) | 93.2 (449/482) | 91.8 (180/196) | 93.1 (244/262) | **87.2** (429/492) | 91.2 (466/511) | 96.8 (987/1,020) | 87.3 (207/237) |
| | Rust | 72.4 (422/583) | 38.6 (248/643) | 91.2 (312/342) | 94.1 (636/676) | 79.3 (466/588) | **26.6** (238/895) | 91.9 (570/620) | 86.0 (301/350) |
| deepseek-6.7b | JavaScript | 73.1 (425/581) | 81.3 (231/284) | 86.0 (505/587) | 91.6 (164/179) | **61.8** (777/1,257) | 96.7 (205/212) | 76.4 (292/382) | 86.1 (167/194) |
| | Python | 73.2 (423/578) | 83.6 (316/378) | 59.5 (601/1,010) | 81.0 (111/137) | **41.0** (1,156/2,819) | 88.1 (244/277) | 60.1 (196/326) | 75.0 (165/220) |
| | Ruby | 86.1 (353/410) | 95.2 (479/503) | 87.8 (360/410) | 90.0 (199/221) | **77.5** (1,418/1,829) | 95.7 (485/507) | 83.4 (151/181) | 89.9 (249/277) |
| | Rust | 66.9 (521/779) | 85.2 (574/674) | 81.1 (439/541) | 74.3 (182/245) | **58.9** (1,328/2,253) | 94.3 (528/560) | 96.3 (1,423/1,478) | 67.0 (185/276) |
| gemma-1b | JavaScript | 77.0 (1,151/1,494) | 76.8 (994/1,295) | 78.5 (707/901) | 88.5 (571/645) | **49.2** (883/1,793) | 84.8 (1,180/1,391) | 77.6 (1,303/1,680) | 88.6 (658/743) |
| | Python | 72.6 (1,224/1,685) | 74.8 (1,265/1,691) | 66.7 (873/1,309) | 84.6 (737/871) | **39.0** (640/1,640) | 73.4 (1,144/1,558) | 75.1 (1,383/1,841) | 79.6 (930/1,168) |
| | Ruby | **83.7** (1,113/1,329) | 86.3 (1,073/1,244) | 88.4 (729/825) | 86.4 (654/757) | 86.1 (802/931) | 87.4 (587/672) | 86.3 (1,116/1,293) | 86.6 (775/895) |
| | Rust | 69.2 (1,162/1,679) | 75.0 (1,134/1,511) | 75.0 (496/661) | 88.2 (464/526) | 69.0 (459/665) | **47.6** (1,068/2,243) | 48.0 (828/1,726) | 90.9 (680/748) |
| gemma-4b | JavaScript | 88.6 (1,692/1,909) | 89.4 (1,774/1,984) | **31.0** (567/1,828) | 82.6 (804/973) | 71.1 (927/1,303) | 88.1 (1,667/1,892) | 86.2 (1,692/1,963) | 84.1 (630/749) |
| | Python | 77.2 (1,232/1,596) | 79.1 (1,234/1,560) | 37.7 (590/1,563) | 89.1 (895/1,004) | **11.4** (277/2,434) | 75.3 (1,287/1,709) | 73.5 (1,387/1,886) | 86.9 (806/927) |
| | Ruby | 90.2 (1,161/1,287) | 90.7 (1,180/1,301) | **69.8** (981/1,405) | 93.0 (745/801) | 82.5 (1,053/1,276) | 91.5 (1,355/1,481) | 90.3 (1,058/1,172) | 88.5 (711/803) |
| | Rust | 49.7 (1,016/2,046) | 49.5 (1,014/2,050) | **38.6** (924/2,393) | 87.1 (681/782) | 55.9 (1,103/1,973) | 47.9 (962/2,009) | 51.3 (1,088/2,120) | 88.5 (637/720) |
| llama-8b | JavaScript | 80.1 (1,328/1,657) | 80.2 (1,235/1,539) | **43.4** (836/1,925) | 89.4 (680/761) | 49.0 (95/194) | 77.4 (1,125/1,454) | 72.2 (1,324/1,833) | 89.6 (600/670) |
| | Python | 71.4 (1,275/1,785) | 71.2 (1,300/1,826) | **32.5** (1,069/3,288) | 89.2 (815/914) | 60.8 (818/1,345) | 72.2 (1,212/1,678) | 66.4 (1,515/2,283) | 86.6 (720/831) |
| | Ruby | 90.4 (1,746/1,931) | 87.2 (1,487/1,705) | **62.4** (1,163/1,864) | 94.6 (955/1,009) | 83.9 (1,108/1,321) | 90.1 (1,755/1,947) | 85.8 (1,855/2,163) | 94.1 (604/642) |
| | Rust | 49.1 (1,205/2,456) | 48.5 (1,189/2,452) | **32.2** (1,537/4,771) | 81.6 (703/861) | 71.4 (769/1,077) | 48.4 (1,120/2,313) | 49.7 (1,422/2,859) | 51.0 (616/1,209) |
| mistral-7b | JavaScript | 65.7 (992/1,509) | 66.1 (979/1,482) | **34.2** (465/1,359) | 79.2 (449/567) | 62.1 (958/1,542) | 64.8 (991/1,529) | 68.1 (1,173/1,723) | 82.7 (335/405) |
| | Python | 65.6 (1,553/2,367) | 66.3 (1,573/2,374) | 48.7 (696/1,429) | 90.2 (826/916) | **41.5** (832/2,006) | 64.7 (1,505/2,327) | 65.6 (1,575/2,402) | 87.3 (738/845) |
| | Ruby | 76.6 (1,220/1,593) | 80.9 (1,157/1,430) | **55.0** (834/1,517) | 86.8 (191/220) | 79.2 (1,127/1,423) | 82.3 (827/1,005) | 85.0 (1,447/1,702) | 93.7 (164/175) |
| | Rust | 46.6 (1,361/2,920) | 46.1 (1,363/2,957) | **42.0** (939/2,238) | 81.3 (655/806) | 56.9 (2,157/3,792) | 46.7 (1,364/2,918) | 50.4 (1,303/2,586) | 77.3 (559/723) |
| qwen-1.5b | JavaScript | 65.4 (981/1,500) | 63.1 (1,040/1,647) | 55.6 (789/1,418) | 79.6 (565/710) | **46.9** (927/1,975) | 72.5 (1,173/1,619) | 75.9 (1,438/1,894) | 75.9 (183/241) |
| | Python | 71.6 (1,437/2,007) | 74.9 (1,398/1,866) | 62.6 (696/1,112) | 85.1 (730/858) | **24.5** (624/2,551) | 81.0 (1,662/2,051) | 76.3 (1,527/2,000) | 77.1 (592/768) |
| | Ruby | 81.4 (1,265/1,554) | 84.4 (1,488/1,763) | 83.5 (988/1,183) | 89.3 (507/568) | **64.2** (1,688/2,629) | 86.1 (1,112/1,291) | 92.2 (1,921/2,084) | 81.7 (384/470) |
| | Rust | 51.4 (1,639/3,186) | 51.1 (1,624/3,175) | **29.8** (941/3,159) | 49.8 (696/1,398) | 53.7 (1,380/2,569) | 45.6 (1,182/2,592) | 48.9 (1,192/2,437) | 36.5 (519/1,423) |
| qwen-3b | JavaScript | 75.0 (1,227/1,636) | 74.9 (1,199/1,600) | 54.2 (623/1,150) | 90.6 (903/997) | **50.5** (963/1,908) | 72.9 (1,193/1,636) | 74.0 (1,321/1,785) | 86.9 (637/733) |
| | Python | 74.2 (1,309/1,765) | 76.6 (1,197/1,562) | 38.3 (362/944) | 86.3 (816/945) | **34.5** (688/1,994) | 72.5 (1,182/1,630) | 68.1 (1,519/2,232) | 80.0 (427/534) |
| | Ruby | 89.2 (1,736/1,947) | 91.8 (1,813/1,974) | **57.9** (987/1,704) | 94.7 (1,323/1,397) | 65.9 (1,328/2,016) | 89.8 (1,880/2,093) | 87.3 (2,096/2,400) | 94.0 (864/919) |
| | Rust | 51.6 (1,424/2,758) | 52.1 (1,461/2,803) | **44.3** (947/2,139) | 71.3 (536/752) | 61.0 (904/1,483) | 49.5 (1,188/2,401) | 53.6 (1,546/2,886) | 64.7 (389/601) |

for standard-library modules that prior pipelines misclassified as hallucinated, reveals that Python hallucination rates have been overstated by up to 9.4 percentage points in previous work. We recommend per-language standard-library exclusion as a baseline requirement for any future evaluation of package hallucination.

On the mitigation side, our systematic evaluation of five guided decoding strategies across eight models and four programming languages demonstrates that inference-time intervention is a viable and practical defense. Contrastive Decoding reduced hallucination in 19 of 32 configurations, while Nudging provided the best trade-off between hallucination reduction and output completeness, making it a promising lightweight strategy under standard, non-adversarial prompts among the guided decoding strategies. These results show that package hallucination can be substantially reduced without model retraining and without external retrieval infrastructure. Our utility analysis further shows that hallucination reduction and recommendation usefulness are not equivalent. Among all evaluated strategies, Greedy decoding provides the strongest average trade-off, followed by Self-Refine and Nudging.

At the same time, our adversarial evaluation exposes clear limits. Prompts seeded with fabricated package names amplify hallucination rates significantly by up to 58.1 percentage points, and all five decoding-only strategies fail to provide consistent protection under these conditions. RAG and Self-Refine performed best under hostile prompts, indicating protection requires external grounding or iterative self-verification. Practitioners should match defenses to the threat model: guided decoding for standard (non-adversarial) usage and stronger interventions for adversarial settings. Future work should evaluate more sophisticated attack vectors, such as indirect prompt injection or multi-turn manipulation, and combining guided decoding with lightweight registry verification at inference time.

## 8 Acknowledgment

This work was supported by the Luxembourg Ministry of Foreign and European Affairs through their Digital4Development (D4D) portfolio under the LuxWAyS project and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (Project NATURAL - Grant agreement N° 949014).

## Data Availability

The data used in this research are derived from publicly available sources. To ensure transparency and reproducibility, the source code, datasets, and usage instructions are publicly accessible in our anonymized repository at https://zenodo.org/records/21786704.

## References


[1] Wadhah Al-Zofi. 2025. AI-Induced Supply-Chain Compromise: A Systematic Review of Package Hallucinations and Slopsquatting Attacks. doi:10.21203/rs.3.rs-8007192/v1

[2] Thomas E. Armstrong. 2025. Slopsquatting assurance: Auditing AI's New Supply Chain Threat. *EDPACS* 70, 12 (Dec. 2025), 1–9. doi:10.1080/07366981.2025.2510097 _eprint: 10.1080/07366981.2025.2510097.

[3] Yung-Sung Chuang, Yujia Xie, Hongyin Luo, Yoon Kim, James Glass, and Pengcheng He. 2024. DoLa: Decoding by Contrasting Layers Improves Factuality in Large Language Models. doi:10.48550/arXiv.2309.03883 arXiv:2309.03883 [cs].

[4] Hantian Ding, Varun Kumar, Yuchen Tian, Zijian Wang, Rob Kwiatkowski, Xiaopeng Li, Murali Krishna Ramanathan, Baishakhi Ray, Parminder Bhatia, and Sudipta Sengupta. 2023. A Static Evaluation of Code Completion by Large Language Models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 5: Industry Track)*. Association for Computational Linguistics, 347–360. doi:10.18653/v1/2023.acl-industry.34

[5] Albérick Euraste Djiré, Abdoul Kader Kaboré, Iyiola E Olatunji, Earl T Barr, Jacques Klein, and Tegawendé F Bissyandé. 2025. Memorization or interpolation? detecting llm memorization through input perturbation analysis. *arXiv preprint arXiv:2505.03019* (2025). doi:10.48550/arXiv.2505.03019

[6] Greta Dolcetti, Vincenzo Arceri, Eleonora Iotti, Sergio Maffeis, Agostino Cortesi, and Enea Zaffanella. 2026. Helping LLMs improve code generation using feedback from testing and static analysis. *Discover Artificial Intelligence* 6, 1 (March 2026). doi:10.1007/s44163-026-01009-5

[7] Djiré Albérick Euraste, Kaboré Abdoul Kader, Jordan Samhi, Earl T Barr, Jacques Klein, and Tegawendé F Bissyandé. 2026. Learned or Memorized? Quantifying Memorization Advantage in Code LLMs. *arXiv preprint arXiv:2604.13997* (2026). doi:10.48550/arXiv.2604.13997

[8] Sebastian Farquhar, Jannik Kossen, Lorenz Kuhn, and Yarin Gal. 2024. Detecting hallucinations in large language models using semantic entropy. *Nature* 630, 8017 (2024), 625–630. doi:10.1038/s41586-024-07421-0

[9] Dren Fazlija, Iyiola E Olatunji, Daniel Kudenko, and Sandipan Sikdar. 2026. Towards Sensitivity-Aware Language Models. *arXiv preprint arXiv:2601.20901* (2026). doi:10.48550/arXiv.2601.20901

[10] Yu Fei, Yasaman Razeghi, and Sameer Singh. 2025. Nudging: Inference-time Alignment of LLMs via Guided Decoding. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (Eds.). Association for Computational Linguistics, Vienna, Austria, 12702–12739. doi:10.18653/v1/2025.acl-long.623

[11] Yuyao Ge, Lingrui Mei, Zenghao Duan, Tianhao Li, Yujia Zheng, Yiwei Wang, Lexin Wang, Jiayu Yao, Tianyu Liu, Yujun Cai, Baolong Bi, Fangda Guo, Jiafeng Guo, Shenghua Liu, and Xueqi Cheng. 2025. A Survey of Vibe Coding with Large Language Models. doi:10.48550/arXiv.2510.12399 arXiv:2510.12399 [cs].

[12] Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. 2023. Grammar-Constrained Decoding for Structured NLP Tasks without Finetuning. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 10932–10952. doi:10.18653/v1/2023.emnlp-main.674

[13] Md Nazmul Haque, Elizabeth Lin, Lawrence Arkoh, Biruk Tadesse, and Bowen Xu. 2025. Secure or Suspect? Investigating Package Hallucinations of Shell Command in Original and Quantized LLMs. doi:10.48550/arXiv.2512.08213 arXiv:2512.08213 [cs].

[14] Xuan He, Dong Li, Hao Wen, Yueheng Zhu, Chao Liu, Meng Yan, and Hongyu Zhang. 2025. CoSEFA: An LLM-Based Programming Assistant for Secure Code Generation via Supervised Co-Decoding. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*. ACM, Clarion Hotel Trondheim Trondheim Norway, 1198–1202. doi:10.1145/3696630.3728609

[15] Wenxin Jiang, Berk Çakar, Mikola Lysenko, and James C. Davis. 2025. ConfuGuard: Using Metadata to Detect Active and Stealthy Package Confusion Attacks Accurately and at Scale. doi:10.48550/arXiv.2502.20528 arXiv:2502.20528 [cs].

[16] Jannik Kossen, Jiatong Han, Muhammed Razzak, Lisa Schut, Shreshth Malik, and Yarin Gal. 2024. Semantic entropy probes: Robust and cheap hallucination detection in llms. *arXiv preprint arXiv:2406.15927* (2024). doi:10.48550/arXiv.2406.15927

[17] Arjun Krishna, Erick Galinkin, Leon Derczynski, and Jeffrey Martin. 2025. Importing Phantoms: Measuring LLM Package Hallucination Vulnerabilities. doi:10.48550/arXiv.2501.19012 arXiv:2501.19012 [cs].

[18] Piergiorgio Ladisa, Serena Elisa Ponta, Antonino Sabetta, Matias Martinez, and Olivier Barais. 2023. Journey to the Center of Software Supply Chain Attacks. *IEEE Security & Privacy* 21, 6 (2023), 34–49. doi:10.1109/MSEC.2023.3302066

[19] Yunseo Lee, John Youngeun Song, Dongsun Kim, Jindae Kim, Mijung Kim, and Jaechang Nam. 2025. Hallucination by Code Generation LLMs: Taxonomy, Benchmarks, Mitigation, and Challenges. doi:10.48550/arXiv.2504.20799 arXiv:2504.20799 [cs].

[20] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems* 33 (2020), 9459–9474.

[21] Xiang Lisa Li, Ari Holtzman, Daniel Fried, Percy Liang, Jason Eisner, Tatsunori Hashimoto, Luke Zettlemoyer, and Mike Lewis. 2023. Contrastive Decoding: Open-ended Text Generation as Optimization. doi:10.48550/arXiv.2210.15097 arXiv:2210.15097 [cs].

[22] Zike Li, Mingwei Liu, Anji Li, Kaifeng He, Yanlin Wang, Xin Peng, and Zibin Zheng. 2025. A Preliminary Study on the Robustness of Code Generation by Large Language Models. doi:10.48550/arXiv.2503.20197 arXiv:2503.20197 [cs].

[23] Fang Liu, Yang Liu, Lin Shi, Zhen Yang, Li Zhang, Xiaoli Lian, Zhongqi Li, and Yuchi Ma. 2026. Beyond Functional Correctness: Exploring Hallucinations in LLM-Generated Code. doi:10.48550/arXiv.2404.00971 arXiv:2404.00971 [cs].

[24] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-Refine: Iterative Refinement with Self-Feedback. In *Advances in Neural Information Processing Systems 36 (NeurIPS 2023)*. Neural Information Processing Systems Foundation, Inc. (NeurIPS), 46534–46594. doi:10.52202/075280-2019

[25] Shradha Neupane, Grant Holmes, Elizabeth Wyss, Drew Davidson, and Lorenzo De Carli. 2023. Beyond Typosquatting: An In-depth Look at Package Confusion. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 3439–3456. https://www.usenix.org/conference/usenixsecurity23/presentation/neupane

[26] Sean O'Brien and Mike Lewis. 2023. Contrastive decoding improves reasoning in large language models. *arXiv preprint arXiv:2309.09117* (2023). doi:10.48550/arXiv.2309.09117

[27] Iyiola E Olatunji, Franziska Boenisch, Jing Xu, and Adam Dziedzic. 2025. Adversarial attacks and defenses on graph-aware large language models (llms). *arXiv preprint arXiv:2508.04894* (2025). doi:10.48550/arXiv.2508.04894

[28] Iyiola E Olatunji, Alberick Euraste Djire, Jacques Klein, and Tegawendé F Bissyandé. 2026. Do Not Copy/Paste: Soft Barriers for Copying in AI-Assisted Programming. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. doi:10.1145/3832783.3834554

[29] Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. doi:10.48550/arXiv.2201.11227 arXiv:2201.11227 [cs].

[30] Henrijs Princis, Arindam Sharma, and Cristina David. 2025. TreeCoder: Systematic Exploration and Optimisation of Decoding and Constraints for LLM Code Generation. doi:10.48550/arXiv.2511.22277 arXiv:2511.22277 [cs].

[31] Prateek Kumar Rajput, Yewei Song, Abdoul Aziz Bonkoungou, Iyiola E. Olatunji, Abdoul Kader Kabore, Jacques Klein, and Tegawendé F. Bissyandé. 2026. Correctness isn't Efficiency: Runtime Memory Divergence in LLM-Generated Code. In *Proceedings of the IEEE/ACM 48th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '26)*. ACM, 693–703. doi:10.1145/3786583.3786908

[32] Jens Rauch, Iyiola E Olatunji, and Megha Khosla. 2021. Achieving differential privacy for $k$-nearest neighbors based outlier detection by data partitioning. *arXiv preprint arXiv:2104.07938* (2021). doi:10.48550/arXiv.2104.07938

[33] Chufan Shi, Haoran Yang, Deng Cai, Zhisong Zhang, Yifan Wang, Yujiu Yang, and Wai Lam. 2024. A Thorough Examination of Decoding Methods in the Era of LLMs. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, Miami, Florida, USA, 8601–8629. doi:10.18653/v1/2024.emnlp-main.489

[34] Joseph Spracklen, Raveen Wijewickrama, A H M Nazmus Sakib, Anindya Maiti, and Bimal Viswanath. 2025. We Have a Package for You! A Comprehensive Analysis of Package Hallucinations by Code Generating LLMs. In *34th USENIX Security Symposium (USENIX Security 25)*. USENIX Association, Seattle, WA, 3687–3706. https://www.usenix.org/conference/usenixsecurity25/presentation/spracklen

[35] Florian Tambon, Arghavan Moradi-Dakhel, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Giuliano Antoniol. 2025. Bugs in large language models generated code: an empirical study. *Empirical Software Engineering* 30, 3 (Feb. 2025). doi:10.1007/s10664-025-10614-4

[36] Melissa Tessa, Iyiola E. Olatunji, Jacques Klein, and Tegawendé F. Bissyande. 2026. Position: The Iceberg of Pitfalls in LLM-Based Secure Code Generation. *Proceedings of the AAAI Symposium Series* 9, 1 (2026), 162–165. doi:10.1609/aaaiss.v9i1.42920

[37] Melissa Tessa, Iyiola E Olatunji, Aicha War, Jacques Klein, and Tegawendé F Bissyandé. 2026. How Secure is Secure Code Generation? Adversarial Prompts Put LLM Defenses to the Test. *arXiv preprint arXiv:2601.07084* (2026). doi:10.48550/arXiv.2601.07084

[38] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F Bissyandé. 2023. Is ChatGPT the ultimate programming assistant–how far is it? *arXiv preprint arXiv:2304.11938* (2023). doi:10.48550/arXiv.2304.11938

[39] Lukas Twist, Jie M. Zhang, Mark Harman, and Helen Yannakoudakis. 2026. Library Hallucinations in LLMs: Risk Analysis Grounded in Developer Queries. doi:10.48550/arXiv.2509.22202 arXiv:2509.22202 [cs].

[40] Laurie Williams, Giacomo Benedetti, Sivana Hamer, Ranindya Paramitha, Imranur Rahman, Mahzabin Tamanna, Greg Tystahl, Nusrat Zahan, Patrick Morrison, Yasemin Acar, Michel Cukier, Christian Kästner, Alexandros Kapravelos, Dominik Wermke, and William Enck. 2025. Research Directions in Software Supply Chain Security. *ACM Trans. Softw. Eng. Methodol.* 34, 5 (May 2025), 146:1–146:38. doi:10.1145/3714464

[41] Hongxiang Zhang, Hao Chen, Muhao Chen, and Tianyi Zhang. 2025. Active Layer-Contrastive Decoding Reduces Hallucination in Large Language Model Generation. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (Eds.). Association for Computational Linguistics, Suzhou, China, 3028–3046. doi:10.18653/v1/2025.emnlp-main.150

[42] Ziyao Zhang, Chong Wang, Yanlin Wang, Ensheng Shi, Yuchi Ma, Wanjun Zhong, Jiachi Chen, Mingzhi Mao, and Zibin Zheng. 2025. LLM Hallucinations in Practical Code Generation: Phenomena, Mechanism, and Mitigation. *Proceedings of the ACM on Software Engineering* 2, ISSTA (June 2025), 481–503. doi:10.1145/3728894

Received 2026-03-26; accepted 2026-06-18