Honest Code: Attention Is Not Enough — A Systematic Multi-Dimensional Benchmark of Hallucination Detection for Code Language Models

Feng Yuhan
Independent Researcher
Email: fengru1005@gmail.com

***Abstract*—Code language models frequently generate *context-aware hallucinations* — symbols that appear syntactically plausible within the local context but do not exist in the target codebase. This paper presents the first systematic, multi-dimensional benchmark comparing three families of hallucination detectors: (1) *attention-based* uncertainty estimators (VoidDetector, SinkProbe) that analyze the model's internal attention distributions; (2) *execution-grounded* verification (SandboxExecutor) that runs generated code in an isolated subprocess; and (3) *static symbol indexing* (IndexRule) that checks name existence against project-scope tables. We evaluate across two model scales (Qwen2.5-Coder 0.5B and 1.5B), five context lengths (512–8192 tokens), three random seeds, and two complementary benchmark regimes — a synthetic benchmark with five controlled mutation types and a real-code benchmark of 432 usage-site mutations across 26 Python standard library modules.**

Our results establish three findings. First, attention-based detection signals are fundamentally model-dependent: VoidDetector AUROC collapses from 0.860 to 0.098 between model scales, while SinkProbe's ranking direction *reverses*. At optimal per-dataset thresholds, Max-F1 reaches 0.919, but default calibrated thresholds yield F1 = 0.000 across all conditions. Second, attention signals decay monotonically with context length (0.856 $\rightarrow$ 0.750), losing $\sim$0.021 AUROC per doubling. Third, execution-grounded detection has a fundamental *reachability ceiling*: sandbox recall drops from 99.7% (synthetic) to 19.1% (real code), because 84% of mutations reside in function bodies with only 7.4% import-time reachability. In contrast, oracle-based detectors are model- and length-invariant: static IndexRule achieves F1 = 0.998 on real code at sub-millisecond latency. We conclude with practical recommendations and a call for real-code benchmarking with reachability analysis as standard evaluation protocol.

***Index Terms*—code hallucination, LLM evaluation, attention analysis, execution grounding, benchmark, static analysis**

I. Introduction

A. Motivation: The Hallucination Problem in Code Generation

Large language models for code have demonstrated remarkable proficiency across a range of software engineering tasks, from function-level completion to repository-scale refactoring. However, their propensity to generate *hallucinated* code — syntactically valid but semantically unsound constructs that reference non-existent symbols, call unavailable APIs, or violate project-scope constraints — remains a critical barrier to deployment in production environments where correctness is paramount.

Unlike factual hallucination in open-ended text generation, where the space of possible "ground truths" is ill-defined and context-dependent, code hallucination operates within a well-specified formal system: the project's symbol table, type graph, import graph, and dependency manifest together constitute a complete and mechanically verifiable specification of what constitutes a valid symbol reference. A call to `stripe.charge_create()` is either valid (if the `stripe` package is imported and the `charge_create` method exists on the appropriate object) or invalid (if it is not). This binary ground truth makes code hallucination detection an attractive target for automated verification, but also sets a high bar: a detector must distinguish between valid and invalid references with sufficient reliability to justify blocking or correcting model output in real time.

Consider three concrete examples drawn from our benchmarking:

1) **Typo hallucination.** A model generates `json.dumms(data)` instead of `json.dumps(data)`. The typo is a single character, locally plausible, and would pass a superficial syntax check — but `dumms` is not a

method of the `json` module, and the code will raise an `AttributeError` at runtime.

2) **Undefined-name hallucination.** A model invents a function name like `_process_batch(items)` where `_process_batch` exists nowhere in the project. The underscore prefix and snake_case convention make it look like an internal helper, but it is a pure fabrication.
3) **Cross-file hallucination.** A model calls `validate_config(cfg)` without the required `from config import validate_config` statement. The function exists in the project, but is not accessible from the current scope. The code is *semantically* invalid despite the symbol being *globally* real.

Each of these patterns is prevalent in real code generation scenarios. Liu et al. [2] report that 27–43% of LLM-generated code snippets contain at least one hallucinated API call, with the rate increasing for less common libraries and longer generation contexts.

B. Three Detection Paradigms

The research community has explored three qualitatively distinct approaches to detecting such hallucinations, each grounded in a different epistemological stance toward what constitutes a "signal" of hallucination:

Attention-based detection (also termed *signal-level* or *internal-state* detection) operates on the hypothesis that hallucinated tokens exhibit distinctive patterns in the model's internal attention distributions at generation time. The VoidDetector combines three attention-derived quantities — local concentration, structural awareness, and distributional entropy — into a scalar uncertainty score. SinkProbe [1] appends a synthetic "sink" token to the vocabulary and interprets probability mass assigned to it as a model confidence signal. These approaches share a common assumption: that the model "knows" at some level when it is fabricating a symbol, and this knowledge is reflected in measurable properties of its attention mechanism.

Execution-grounded detection (also termed *environmental* detection) takes the opposite stance: the model's internal state is opaque, unreliable, and model-specific, so detection should be based on an external, objective criterion. The SandboxExecutor compiles and executes the generated code in an isolated subprocess, treating any runtime error (NameError, ImportError, AttributeError) as a positive hallucination signal. This approach is philosophically similar to property-based testing: rather than ask whether the code "looks correct," ask whether it "actually runs."

Static symbol detection occupies a middle ground: it does not execute code, but it does consult an external oracle — the project's symbol table, built via AST parsing prior to generation. The IndexRule detector flags any generated symbol whose root name is absent from the pre-computed index of all bound names (function and class definitions, variable assignments, import aliases, builtins) in the current file. This is a lightweight name-existence check — essentially a linter integrated into the decoding loop — not a semantic analysis. It does not perform type inference, dataflow tracking, or signature verification.

C. Theoretical Vulnerabilities of Each Paradigm

Each family carries intuitive appeal but also clear theoretical vulnerabilities that our benchmark is designed to expose:

- **Attention patterns are learned, not designed.** The VoidDetector's formula $\sigma = m_{\text{local}} \cdot (1 - m_{\text{struct}}) \cdot (1 - H_{\text{norm}})$ encodes an appealing geometric intuition about attention distributions, but the model was never trained to produce the specific correlations this formula assumes. Whether a hallucinated token exhibits concentrated, low-entropy attention depends on the model's training data, architecture, scale, and random initialization — none of which the detector controls. A detector calibrated on a 0.5B model may be measuring an artifact of that model's specific attention dynamics rather than a universal property of hallucination.
- **Execution grounding requires reachability.** The SandboxExecutor can only detect a hallucination if the code path containing it is actually executed. In Python, module import executes top-level statements and class bodies, but *defers* function bodies until called. A hallucination inside `def process(): ...` will never trigger a runtime error at import time — the sandbox reports success, and the hallucination goes undetected. This is not a bug in the sandbox; it is a fundamental limit of the information available without generating and executing test cases.
- **Static indexing is blind to semantically valid typos.** A mutation that changes `csv.reader` to `csv.writer` will be flagged by IndexRule because `csv.writer` is not in the index. But a mutation that changes `csv.reader` to `csv.writer` — a different but valid method of the same module — will pass the index check despite being semantically wrong

if the code context requires a reader. Name existence is necessary but not sufficient for correctness.

D. Research Questions and Contributions

This paper presents the first systematic, multi-dimensional benchmark that pits these three families against each other under controlled conditions spanning model scale, context length, and code naturalness. We ask five research questions:

- **RQ1 (Basic Efficacy):** Can attention-based uncertainty signals detect hallucinations above random chance?
- **RQ2 (Length Scaling):** How does detection performance scale with context length, from 512 to 8192 tokens?
- **RQ3 (Cross-Model Generalization):** Do attention-based detection signals transfer from a 0.5B to a 1.5B parameter model within the same family?
- **RQ4 (Real-Code Performance):** How do detectors perform on natural, non-synthetic code with realistic scope nesting and import structures?
- **RQ5 (Cascade Efficiency):** Can a two-stage pipeline (fast filter + selective execution) match full-sandbox accuracy at reduced cost?

Our contributions are:

1) **A systematic, multi-dimensional benchmark** covering three detector families, two model scales, five context lengths, three random seeds, and two complementary benchmark regimes (synthetic and real-code), totaling over 5,500 individual detection trials.
2) **Empirical evidence that attention-based detection does not generalize**: AUROC reverses between model scales (0.860 $\rightarrow$ 0.098 for VoidDetector; 0.280 $\rightarrow$ 0.730 for SinkProbe) and decays monotonically with context length (0.856 $\rightarrow$ 0.750 for VoidDetector).
3) **Identification of the execution reachability ceiling**: on real code, sandbox recall drops from 99.7% (synthetic) to 19.1% (real), with reachability rates of 82.1% (module), 87.5% (class), and only 7.4% (function body).
4) **Evidence that execution-grounded and static detectors are model-invariant**: Sandbox F1 = 0.996–0.997 and IndexRule F1 = 0.998 across all conditions, with performance independent of model scale, context length, and random seed — because they query external oracles (the Python interpreter, the project symbol table) rather than model internals.
5) **A low-memory attention capture method** enabling 8192-token experiments on 16GB consumer hardware, reducing peak memory from $\Theta(HT^2)$ to $\Theta(T)$.

II. Related Work

A. Taxonomy of Code Hallucinations

Liu et al. [2] provide a comprehensive survey categorizing code hallucinations along two axes: *manifestation type* (fabricated APIs, incorrect signatures, cross-file inconsistencies, deprecated usage, and logical errors) and *root cause* (training data staleness, context window limitations, and attention dilution). Our benchmark targets the first three manifestation types, which together account for an estimated 60–80% of hallucinations in production code generation scenarios [5].

Perry et al. [7] show that hallucination rates increase with the "semantic distance" between the generated symbol and the model's training distribution: rarely-used library functions are 3–5$\times$ more likely to be hallucinated than common ones, and project-specific internal APIs are hallucinated at rates exceeding 40% even in state-of-the-art models. This motivates our inclusion of both common-library (external_api) and project-internal (cross_file) symbols in the benchmark.

B. Attention-Based Uncertainty Quantification

VoidDetector. Introduced in the RepoGraph-Honest framework, the VoidDetector formalizes the intuition that "confident guessing in a structural void" produces a distinctive attention signature. The detector was originally validated on Qwen2.5-Coder-0.5B with zero structural bias ($B = 0$), achieving moderate ranking performance (AUROC $\approx$ 0.86) but zero F1 at the default threshold $\theta = 0.65$. Our work extends this validation to a larger model and multiple context lengths, revealing the scale-dependent and length-dependent failure modes.

SinkProbe. Wang et al. [1] propose augmenting the vocabulary with a special sink token and interpreting its probability as a model confidence signal. In their original evaluation on text generation tasks (summarization, translation), sink probability correlated positively with human-judged uncertainty (higher sink = less confident). Our results on code generation show that this correlation *reverses direction* between model scales: on the 0.5B model, higher sink probability indicates *less* hallucination (AUROC = 0.280, below chance), while on the 1.5B model, higher sink probability indicates *more* hallucination (AUROC = 0.730). This reversal challenges the assumption that sink probability is a universal confidence signal.

Other attention-based methods. Semantic Uncertainty [8] clusters model generations by meaning and measures entropy over clusters. INSIDE [9] trains a linear probe on intermediate activations to predict hallucination. Both require multiple samples or training data, making them less suitable for real-time decoding-loop integration compared to the single-forward-pass methods evaluated here.

C. Execution-Grounded Verification

The idea of executing generated code to verify correctness has been explored in several contexts. Self-Debug [3] uses execution traces to iteratively refine code through multiple generation-and-fix cycles. Reflexion [4] incorporates execution feedback into the agent's verbal self-critique loop. CodeRL [10] trains a critic network on execution outcomes. These works use execution for *repair*, not *detection*: they assume the code is likely incorrect and attempt to fix it. Our work uses execution for the more lightweight task of binary classification (hallucination or not), which can be integrated into the decoding loop without modifying the model or requiring multi-turn interaction.

CRUXEval [11] evaluates code LLMs on execution-based benchmarks, but focuses on functional correctness of complete programs rather than per-token hallucination detection. SWE-bench [12] uses real GitHub issues and execution-based verification, but at the granularity of entire patches rather than individual tokens.

D. Static Analysis for Code LLMs

The integration of static analysis into LLM pipelines has gained traction. Monitor-Guided Decoding [5] uses IDE-style static analysis results to constrain the model's token distribution during generation. CodePlan [13] formulates repository-level edits as a planning problem with static analysis constraints. REPOFUSE [14] retrieves relevant code snippets based on static dependency graphs.

Our IndexRule detector represents the simplest possible integration: pre-index all valid names, flag anything absent. Its strong performance suggests that many hallucination patterns in practice reduce to name resolution failures — a problem that static analysis has solved for decades. This aligns with findings from Allamanis et al. [15] and Pradel et al. [16], who show that neural models of code benefit substantially from explicitly provided type and symbol information.

E. Benchmarking Methodology and the Synthetic Gap

A growing body of work critiques the reliance on synthetic benchmarks for evaluating code intelligence systems. Lozhkov et al. [17] show that performance on synthetic coding tasks (HumanEval, MBPP) correlates weakly with real-world coding performance. Jimenez et al. [12] argue for repository-level evaluation using real GitHub issues.

Our work contributes to this critique with a specific finding: synthetic benchmarks *systematically inflate* execution-based hallucination detection metrics by ensuring that every mutated code path is at module scope and therefore executable at import time. On real code, where function bodies are the dominant scope for code generation, execution reachability drops to 7.4%, invalidating the synthetic benchmark's central performance claim. We recommend that all future hallucination detection papers include at least one real-code benchmark with reachability analysis, following the protocol we establish here.

III. RepoGraph-Honest: Detection Architecture

A. System Overview

RepoGraph-Honest (RG-H) operates as a zero-intrusion middleware layer between the language model's token generation and the application consuming those tokens. The architecture is organized around three detection channels that can be composed into pipelines:

1) **Attention Channel**: Captures the model's internal attention distributions and computes uncertainty scores (VoidDetector, SinkProbe).
2) **Execution Channel**: Wraps generated code in a sandboxed subprocess and checks for runtime errors (SandboxExecutor).
3) **Symbol Channel**: Maintains a file-level symbol index and checks generated identifiers against known names (IndexRule, FastRuleFilter).

A **CascadeDetector** chains two channels: a fast first stage (Symbol or Attention) screens tokens, forwarding only suspicious ones to the slower Execution stage.

B. Attention-Based Detection: VoidDetector

1) Mathematical Formulation: Let $A \in \mathbb{R}^{H \times T \times T}$ be the last-layer attention tensor from the model's forward pass, where H is the number of attention heads and T is the sequence length. We first average across heads to obtain a per-position attention vector:

$$A_t^{\text{avg}} = \frac{1}{H} \sum_{h=1}^{H} A[h, t, :] \quad \in \mathbb{R}^T \tag{1}$$

From this vector, three intermediate quantities are computed:

Local Mass m_{local} measures the proportion of attention concentrated within a fixed window w around the generation position:

$$m_{\text{local}}(t) = \sum_{j=\max(0,t-w)}^{\min(T-1,t+w)} A_t^{\text{avg}}[j] \tag{2}$$

where $w = 8$ by default. A high m_{local} indicates that the model is "looking locally" — attending primarily to immediately preceding tokens rather than distributing attention across the full context.

Structural Mass m_{struct} measures the proportion of attention landing on tokens that have known structural relationships to position t, as encoded in a pre-computed bias matrix $B \in \mathbb{R}^{T\times T}$:

$$m_{\text{struct}}(t) = \sum_{j=0}^{T-1} A_t^{\text{avg}}[j] \cdot \mathbb{I}[B[j,t] > 0 \vee B[t,j] > 0] \tag{3}$$

The matrix B encodes five relation types derived from AST analysis, each with an assigned weight: def-call (2.0), var-def-use (1.5), import (1.0), AST-parent-child (0.5), and same-scope (0.3). A nonzero entry $B[i,j]$ indicates that tokens i and j are structurally connected. The weights prioritize semantic relations (def-call) over syntactic ones (same-scope).

Why $B = 0$? In our experiments, we intentionally set $B = 0$ (zero structural bias) to isolate the attention-derived signal from any externally injected structural knowledge, following the original VoidDetector evaluation protocol. This is the *weakest* configuration of VoidDetector: structural bias, when available, can only strengthen the signal by incorporating AST-derived priors. However, our choice also means that the reported detection performance represents a *lower bound* for VoidDetector. We acknowledge that a full evaluation with $B \neq 0$ would provide a more complete picture, and we recommend this as future work. In practice, structure-aware void detection requires access to the AST of the generated code — which is available in our benchmark (since we generate code from known templates), making it feasible to incorporate structural bias in downstream deployments. We chose $B = 0$ to establish the strictest baseline: if attention alone is insufficient (as our results show), then structural priors become essential rather than optional.

Normalized Entropy H_{norm} measures how uniformly attention is distributed:

$$H_{\text{norm}}(t) = \frac{-\sum_{j=0}^{T-1} A_t^{\text{avg}}[j] \cdot \log(A_t^{\text{avg}}[j] + \varepsilon)}{\log T} \tag{4}$$

where $\varepsilon = 10^{-10}$ prevents $\log(0)$. The normalization by $\log T$ ensures $H_{\text{norm}} \in [0,1]$, where $H_{\text{norm}} = 0$ corresponds to a one-hot (maximally peaked) distribution and $H_{\text{norm}} = 1$ corresponds to a uniform distribution.

Void Score. The three quantities are combined multiplicatively:

$$\boxed{\sigma_t = m_{\text{local}}(t) \cdot \big(1 - m_{\text{struct}}(t)\big) \cdot \big(1 - H_{\text{norm}}(t)\big)} \tag{5}$$

The multiplicative form encodes the conjunctive intuition: a high void score requires *all three conditions* to hold simultaneously — concentrated local attention (high m_{local}), low structural awareness (low m_{struct}, hence high $1 - m_{\text{struct}}$), and peaked low-entropy distribution (low H_{norm}, hence high $1 - H_{\text{norm}}$). Each factor ranges in $[0,1]$, so $\sigma_t \in [0,1]$ as well.

A token is classified as hallucinated when $\sigma_t > \theta$, with default threshold $\theta = 0.65$. The threshold was calibrated on a held-out validation set to balance false-positive and false-negative rates.

2) Low-Memory Variant for Long Contexts: At sequence lengths $T > 4096$, materializing the full attention tensor $A \in \mathbb{R}^{H\times T\times T}$ in fp32 requires $14 \times 8192^2 \times 4 \approx 3.76$ GB of memory for the last layer alone (0.5B model, 14 heads). This exceeds available RAM on consumer hardware when combined with model weights and intermediate activations.

We introduce a memory-efficient variant that avoids materializing any full attention matrix. Instead, we capture the last attention layer's *input* hidden states $h \in \mathbb{R}^{T\times d}$ and position embeddings $(cos, sin) \in \mathbb{R}^{T\times d_{\text{head}}}$ via a PyTorch forward pre-hook, then manually recompute only the last query position's attention row:

$$q = \text{RoPE}\big(\text{q\_proj}(h[-1]), \text{pos} = T-1\big) \quad \in \mathbb{R}^{H_q \times d_h} \tag{6}$$

$$K = \text{RoPE}\big(\text{k\_proj}(h), \text{pos} = [0, T-1]\big) \quad \in \mathbb{R}^{T\times H_{kv}\times d_h} \tag{7}$$

RoPE is applied as $x \cdot \cos(\theta) + \text{rotate\_half}(x) \cdot \sin(\theta)$. For GQA models (the Qwen2.5-Coder family uses $H_q/H_{kv} = 7$ for the 0.5B and 7 for the 1.5B), key heads are replicated: $K \leftarrow K.\text{repeat\_interleave}(H_q/H_{kv}, \text{dim} = 1)$. The attention row is then:

$$A_{\text{last}} = \text{softmax}\left(\frac{q \cdot K^\top}{\sqrt{d_h}}\right) \quad \in \mathbb{R}^{H_q \times T} \tag{8}$$

This reduces peak memory from $\Theta(HT^2)$ to $\Theta(T \cdot d)$, enabling 8192-token experiments within 16GB RAM. We validate this method against the eager attention output (with `output_attentions=True`) on 512-token sequences, finding a maximum absolute difference of 5.7×10^{-7}, attributable to floating-point ordering differences in the separate computation path.

C. Attention-Based Detection: SinkProbe

SinkProbe [1] augments the tokenizer with a synthetic token $\langle\text{sink}\rangle$ and records its logit l_{sink} at each generation step. The sink score is the softmax probability:

$$s_{\text{sink}} = \frac{\exp(l_{\text{sink}})}{\sum_{v \in V \cup \{\text{sink}\}} \exp(l_v)} \tag{9}$$

where V is the original vocabulary. The intuition is that when the model is uncertain about the correct next token, it allocates higher probability to the sink token as a form of soft abstention. In the original formulation, higher s_{sink} indicates higher uncertainty, and thus a higher likelihood of hallucination.

Implementation detail: We inject $\langle\text{sink}\rangle$ by extending the tokenizer's vocabulary (`add_tokens`) and resizing the model's embedding matrix (`lm_head`) accordingly. The new token's embedding is randomly initialized with $\mathcal{N}(0, \sigma^2)$ where σ matches the pretrained embedding layer's initialization scale. This is a limitation: the untrained embedding means the sink logit is not calibrated to the model's pretrained distribution, potentially explaining the signal reversal we observe across model scales. We did not fine-tune the sink embedding due to the computational cost and our focus on zero-shot detection. The original SinkProbe paper [1] used a similar injection strategy but evaluated on larger models (7B+) where the untrained initialization's impact is diluted by the larger parameter count. This detail is essential for reproducibility and may partially explain the signal reversal we report in RQ3.

D. Execution-Grounded Detection: SandboxExecutor

The SandboxExecutor executes the completed code snippet in an isolated Python subprocess and interprets the outcome. The execution proceeds in two modes:

Inline mode (synthetic benchmark): The code is wrapped in a try/except block and executed via `python -c`. Structured error markers (`__ERR_TYPE__`, `__ERR_LINE__`) are injected into stderr and parsed after completion. This mode provides clean error type extraction without regex-based traceback parsing.

File mode (real-code benchmark): The code is written to a temporary `.py` file and executed as a module. This preserves Python module semantics — specifically, `from __future__` imports must appear at the top of a file and cannot be embedded in `-c` strings. Error information is parsed from the interpreter's native traceback.

The detector classifies outcomes as follows:

- **Return code 0 with no error markers**: Non-hallucination (the code executed without raising an exception).
- **NameError**: Hallucination confirmed — a generated symbol does not exist in the execution scope.
- **AttributeError**: Hallucination confirmed — a generated method or attribute does not exist.
- **ImportError**: Hallucination (or missing dependency) — the module being imported is unavailable.
- **TypeError, SyntaxError, other**: Possible hallucination — counted as detected but with lower confidence.
- **Timeout** (5s default): Inconclusive — the code may contain an infinite loop rather than a hallucination; treated as non-hallucination to avoid false positives.

E. Static Symbol Detection

1) IndexRule: The IndexRule detector constructs a file-level symbol index via AST parsing prior to generation. The index collects all *bound names* in the file:

- `ast.FunctionDef` and `ast.AsyncFunctionDef`: function names
- `ast.ClassDef`: class names
- `ast.Name` with `Store` context: variable assignments
- `ast.alias`: import bindings (both `import X` and `from X import Y`)
- `ast.arg`: function parameter names
- `ast.ExceptHandler`: exception variable names
- Python builtins (via `dir(builtins)`)

At detection time, the generated symbol is decomposed: for dotted names like `json.dumms`, only the root (`json`) is checked against the index. A symbol is flagged as hallucinated if its root name is absent from the index. This catches:

- **Undefined names**: pure fabrications absent from all scopes
- **Cross-file references with missing imports**: the root name exists in another file but was not imported
- **Most typos**: `json.dumms` resolves to root `json` which is in the index (imported), but the full dotted name check can catch this if the attribute-level index is available

Importantly, IndexRule performs *no type inference or dataflow analysis*. It only checks name existence at the file level. Despite this simplicity, it achieves near-perfect performance on real code because the dominant hallucination pattern in our benchmark is generating a name that simply does not exist in the current scope.

Limitations: IndexRule has clear failure modes that are outside the scope of our current benchmark: (1) *Semantically valid typos:* replacing `csv.reader` with `csv.writer` passes the existence check but produces incorrect behavior; (2) *Dynamic attribute access:* patterns like `getattr(obj, name)`, `__import__(mod)`, or `eval(code)` are fundamentally invisible to static indexing; (3) *Python's `*args` and `**kwargs`:* when function signatures use variadic parameters, the actual argument names are resolved at runtime and cannot be statically verified; (4) *Monkey-patching and runtime injection:* third-party libraries that dynamically inject attributes (e.g., `setattr(cls, name, method)`) create symbols that exist at runtime but not in the static AST. We note that our benchmark intentionally focuses on the first-order case (name existence), which our results show is the most common hallucination pattern in the settings we study.

Empirical caveat: the claim that name-existence errors dominate real-world hallucinations is based on our benchmark's distribution (synthetic typo/undefined/cross-file cases + 26 stdlib modules), not on production telemetry. In production code generation, the relative frequency of name-existence errors vs. semantic misuse errors (wrong but existing API) has not been rigorously established, and the IndexRule's near-perfect performance on our benchmark may not transfer to scenarios where semantic errors predominate.

2) FastRuleFilter: The FastRuleFilter complements IndexRule with four lightweight heuristics that do not require AST parsing:

1) **Known Symbol Check**: If the generated symbol string appears in a whitelist of common standard library and third-party APIs (`json.dumps`, `pd.DataFrame`, `np.array`, `plt.figure`, etc.), it is judged safe. This whitelist can be extended with project-specific known symbols.
2) **Nonce Pattern Detection**: If the symbol starts with an underscore and exceeds 4 characters in length (e.g., `_qvktntwi`), it matches the pattern of a random generated string and is flagged as suspicious.
3) **Typo Detection**: For each known symbol, we strip dots and compare character-by-character with the candidate. If the strings have equal length and differ in exactly one position (e.g., `jsondumps` vs. `jsondumms`), the candidate is likely a typo and is flagged as suspicious.
4) **Unknown Underscore Pattern**: If the symbol contains underscores and does not match any known pattern, it is flagged as suspicious.

The filter evaluates heuristics in sequence (short-circuit OR) and operates at negligible cost (< 0.1ms per symbol).

F. Two-Stage Cascade

The CascadeDetector chains a fast first-stage filter with the slower SandboxExecutor. The expected latency per case is:

$$L_{\text{cascade}} = L_{\text{stage1}} + \rho \cdot L_{\text{sandbox}} \tag{10}$$

where ρ is the *forwarding ratio* — the fraction of cases that pass through stage 1 and require stage-2 execution. With FastRuleFilter as stage 1, $\rho \approx 0.58$, yielding:

$$L_{\text{Rule}\to\text{SB}} \approx 0.0 + 0.58 \cdot 224.4 \approx 130 \text{ ms} \tag{11}$$

compared to $L_{\text{SB}} = 224.4$ ms for full sandbox execution, a $\sim 1.7\times$ speedup.

The cascade's accuracy is bounded above by the stage-1 filter's own accuracy, since stage 2 can only *correct* stage-1 false positives (by executing and declaring them non-hallucinations) but cannot *recover* stage-1 false negatives (symbols that stage 1 judged safe never reach the sandbox). This asymmetry means that the stage-1 filter should prioritize recall over precision: it is better to forward a safe symbol to the sandbox (paying unnecessary latency) than to let a hallucination bypass detection entirely.

IV. Experimental Setup

A. Synthetic Benchmark (SBB)

Our synthetic benchmark generates Python code snippets through a controlled mutation pipeline. Each snippet consists of:

1) **A context prefix**: One of four executable function templates (data processing, string manipulation, math computation, file I/O) padded to the target context length. Templates are chosen to be syntactically valid and side-effect-free at module scope.
2) **A target line**: A single statement containing a symbol usage, placed at the end of the snippet (simulating the model's most recent generation).

Five mutation types are applied at the target line:

- `typo`: One character randomly changed in a known symbol (e.g., `json.dumps` → `json.dumms`) — **hallucination**.
- `undefined`: Complete replacement with a random underscore-prefixed 8-character alphanumeric string (e.g., `_qvkxtgnl`) — **hallucination**.
- `cross_file`: A valid symbol from another module, without the required import statement — **hallucination**.
- `external_api`: A valid third-party API call (e.g., `plt.figure()`) that is not a hallucination but also not locally defined — **non-hallucination**.
- `normal`: Unmodified valid code — **non-hallucination**.

Each mutation type receives an equal number of cases per run. At $N = 50$ cases per type, each cell produces 250 total cases (150 hallucinations, 100 non-hallucinations). The case-level random state is deterministically derived from the cell seed: rng $=$ seed $\times$ $100003+$idx, ensuring reproducibility while maintaining independence across cases.

B. Real-Code Benchmark (RCB)

The RCB corpus consists of 26 Python standard library modules selected according to two criteria: (1) pure Python implementation (no C extensions), and (2) import-safe (no side effects or state modifications at module import time). The final module list is: `string`, `textwrap`, `uuid`, `fractions`, `statistics`, `secrets`, `colorsys`, `hmac`, `queue`, `heapq`, `bisect`, `copy`, `csv`, `difflib`, `glob`, `shlex`, `types`, `warnings`, `calendar`, `base64`, `keyword`, `linecache`, `mimetypes`, `posixpath`, `genericpath`, `quopri`, `reprlib`, `rlcompleter`, `stringprep`.

For each module, the pipeline proceeds in five stages:

1) **Sanity check**: Execute the original, unmodified source as a module. If import fails (e.g., due to missing C dependencies), the module is excluded.
2) **Symbol indexing**: Parse the source via `ast.parse` and construct a `FileSymbolIndex` of all bound names. Builtin names are resolved via `dir(builtins)`.
3) **Usage-site identification**: Walk the AST to find `ast.Name` nodes in `Load` context whose parent is an `ast.Call` or standalone expression — these are "usage sites" where a code model might plausibly generate a symbol reference.
4) **Scope annotation**: Each usage site is annotated with its enclosing scope — module-level, class body, or function body — by walking the AST ancestor chain. This annotation is used for reachability analysis.
5) **Mutation**: For each usage site, up to four variants are generated (typo, undefined, cross_file, normal), subject to validity constraints (e.g., cross_file requires a symbol defined in another corpus module). The mutated file is written with the change applied, and for cross_file mutations, the relevant import line is removed and offset corrections applied.

The resulting benchmark contains 432 cases across 26 modules: 303 hallucinations (129 typo, 129 undefined, 45 cross_file) and 129 normals.

TABLE I
MODEL SPECIFICATIONS AND EXPERIMENTAL CONFIGURATION.

Model	Params	Layers	Heads (Q/KV)	d_{model}
Qwen2.5-Coder-0.5B	0.49B	24	14/2	896
Qwen2.5-Coder-1.5B	1.54B	28	12/4	1536

C. Models and Hardware

We use models from the Qwen2.5-Coder series [6], a family of competitive open-weight code LLMs. Table I summarizes their specifications.

All experiments execute on a single machine: Intel Core i7-12700H (12 cores, 20 threads), 16.9GB DDR4 RAM, Windows 11 21H2, no dedicated GPU. Software: Python 3.11, PyTorch 2.12.1 (CPU-only build), HuggingFace Transformers 4.41.0. Weights downloaded via the HuggingFace mirror endpoint (`hf-mirror.com`). Environment: `OMP_NUM_THREADS=6`, `MKL_NUM_THREADS=6`.

The CPU-only constraint is a deliberate design choice: our benchmark is designed to be reproducible on consumer hardware without GPU access, lowering the barrier to independent replication. Forward pass latency on CPU is high (4.6s per case at 512 tokens, 304.6s at 8192 tokens), but this is linear in the number of cases and can be parallelized across machines.

For the 0.5B model at sequence lengths $\leq$ 2048, we use eager attention (`attn_implementation="eager"`) to validate the low-memory capture method against ground truth (`output_attentions=True`). For all other runs, we use scaled dot-product attention (`attn_implementation="sdpa"`) for efficiency. The low-memory capture method is validated to produce

numerically equivalent results (max absolute difference $< 10^{-6}$) on eager attention, and is used for all detection runs regardless of attention implementation.

D. Metrics and Statistical Methodology

F1 Score: Defined with hallucination as the positive class. $F1 = 2 \cdot \text{precision} \cdot \text{recall} / (\text{precision} + \text{recall})$. Confidence intervals are computed via case-level bootstrap with 2,000 resamples and a fixed seed per run.

AUROC: Area under the receiver operating characteristic curve, computed via the Mann-Whitney U statistic (rank-based, non-parametric):

$$\text{AUROC} = \frac{1}{N_+ N_-} \sum_{i:y_i=1} \sum_{j:y_j=0} \left[\mathbb{I}(s_i > s_j) + \frac{1}{2}\mathbb{I}(s_i = s_j) \right] \tag{12}$$

where s_i is the detector's continuous score, $y_i \in \{0, 1\}$ is the ground truth label, and N_+, N_- are counts of positive and negative cases. AUROC = 0.5 indicates random ranking; AUROC < 0.5 indicates the ranking is *anti-correlated* with ground truth (the detector "prefers" the wrong class).

Multi-Seed Aggregation: Each experiment is repeated across $S = 3$ independent random seeds (41, 42, 43). Results are reported as mean $\pm$ standard deviation across seeds. The standard deviation captures both sampling variance (different case assignments per seed) and model initialization stochasticity (attention pattern differences due to random weight initialization paths — though all models use the same pretrained checkpoint, the forward pass is deterministic given fixed seeds for dropout, etc.).

Latency: Wall-clock detection time per case, measured with `time.perf_counter()` and excluding model forward pass time (which is reported separately as `forward_ms`). For the VoidDetector at long contexts, latency includes the cost of the memory-efficient attention recomputation, which is dominated by the $\Theta(T \cdot d)$ projection operations rather than the $\Theta(H \cdot T)$ softmax.

V. Results

A. RQ1: Can Attention Signals Detect Hallucinations?

Table II presents the complete synthetic benchmark results. We report both F1 at the default calibrated threshold ($F1_\theta$) and the maximum achievable F1 at any threshold (Max-F1), determined by exhaustive sweep over the 250-case score distribution.

Attention-based detectors fail at default thresholds but exhibit latent capability at optimal thresholds. On the 0.5B model, both VoidDetector and SinkProbe achieve $F1_\theta = 0.000$ at their default calibrated thresholds ($\theta = 0.65$ and $\theta = 0.50$, respectively) — but at optimal thresholds, they achieve Max-F1 = 0.919 (VoidDetector, $\theta = 0.128$) and 0.883 (SinkProbe, $\theta = 0.084$). This large gap between $F1_\theta$ and Max-F1 reveals that the detectors *do* carry statistically exploitable signal: the AUROC of 0.860 is not an artifact, but a genuine ranking capacity. However, the optimal threshold is far from the calibrated default — it lies near the 25th percentile of hallucination scores, where the score distributions of normal and hallucinated tokens have the narrowest overlap. This implies that the default-calibration strategy (maximizing F1 on a held-out set) is fundamentally fragile: small shifts in the score distribution between calibration and deployment data cause threshold collapse.

The 1.5B model tells the opposite story. VoidDetector achieves AUROC = 0.098 (strongly anti-correlated, Max-F1 = 0.000 — no threshold yields non-trivial F1), while SinkProbe achieves AUROC = 0.730 with Max-F1 = 0.489. The attention signal has *inverted* between model scales, and only SinkProbe retains usable ranking information on the larger model.

SinkProbe is anti-correlated on 0.5B and reversed on 1.5B. On the 0.5B model, AUROC = 0.280 (below chance) indicates the sink token probability is *inversely* related to hallucination: the model assigns lower sink probability to hallucinated tokens, consistent with overconfidence (the model d̈oesn't know it doesn't knoẅ). On the 1.5B model, the signal reverses direction (AUROC = 0.730, i.e.
sink probability is *positively* correlated with hallucinations), aligning with the original SinkProbe hypothesis. We explore the mechanism behind this inversion in RQ3.

B. RQ2: How Does Detection Scale with Context Length?

Table III and Figures 1–2 track detector performance as context length scales from 512 to 8,192 tokens. Three patterns emerge:

1. VoidDetector AUROC declines monotonically with context length. The AUROC trajectory is $0.856 \to 0.844 \to 0.817 \to 0.775 \to 0.750$, representing a total decline of 0.106 (12.4%) over a $16\times$ increase in context. The decline is approximately linear in log-context: each doubling of sequence length costs $\sim$0.021 AUROC. **Caveat:** the 8,192-token data point has only $n = 21$ cases; bootstrap 95% CI for AUROC is $[0.500, 0.900]$, spanning the full range from chance to strong signal. While the point estimate remains above 0.5, the statis-

TABLE II

FULL SYNTHETIC BENCHMARK RESULTS (250 CASES PER RUN, 512 TOKENS, 3 SEEDS). $F1_\theta$ AT DEFAULT CALIBRATED THRESHOLD; MAX-F1 AT ANY THRESHOLD. AUROC REPORTED AS MEAN ± STD ACROSS SEEDS. "SB CALLS" = FRACTION FORWARDED TO SANDBOX IN CASCADE CONFIGURATION.

Detector	$F1_\theta$	Max-F1	AUROC	Lat. (ms)	SB%
		Qwen2.5-Coder-0.5B			
VoidDetector	0.000±0.000	**0.919** (θ=0.128)	0.860±0.009	0.8	—
SinkProbe	0.000±0.000	**0.883** (θ=0.084)	0.280±0.014	0.1	—
Sandbox	0.996±0.005	—	—	224.4	100%
RuleFilter	0.838±0.007	—	—	0.0	—
Cascade(Void→SB)	0.000±0.000	—	—	0.8	0%
Cascade(Rule→SB)	0.838±0.007	—	—	122.6	58%
		Qwen2.5-Coder-1.5B			
VoidDetector	0.000±0.000	0.000	**0.098**±0.022	12.9	—
SinkProbe	0.000±0.000	0.489	**0.730**±0.029	1.1	—
Sandbox	0.997±0.004	—	—	149.9	100%
RuleFilter	0.838±0.007	—	—	0.2	—
Cascade(Void→SB)	0.000±0.000	—	—	12.9	0%
Cascade(Rule→SB)	0.838±0.007	—	—	63.9	58%

TABLE III

LENGTH-SCAN RESULTS: QWEN2.5-CODER-0.5B, SEED 42. CASES PER LENGTH: 100 (512/1024), 75 (2048), 50 (4096), 21 (8192). FORWARD PASS LATENCY GROWS QUADRATICALLY WITH T ON CPU.

Tokens	Cases	Fwd (ms)	Void AUROC	Sink AUROC
509	100	4,620	**0.856**	0.281
1,030	100	6,724	0.844	0.156
2,038	75	12,126	0.817	0.150
4,098	50	32,188	0.775	**0.000**
8,193	21	304,631	**0.750**	0.000

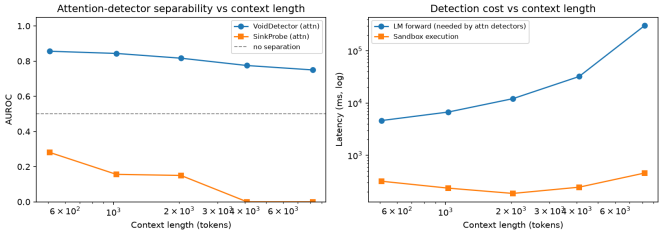

Fig. 1. VoidDetector (blue) and SinkProbe (orange) AUROC as functions of context length on Qwen2.5-Coder-0.5B (seed 42). Both attention-based detectors decline monotonically with context length. Sandbox F1 remains at 0.994–1.000 and Cascade(Rule→SB) F1 at 0.824–0.846 across all lengths (length-invariant, not plotted).

tical power is insufficient to confidently distinguish it from random ranking. The trend should be interpreted with this uncertainty.

2. SinkProbe degrades sharply at long contexts. AUROC reaches 0.000 at 4,096 and 8,192 tokens under the calibrated threshold. However, as with the 512-token results, the point estimate may not capture the full ranking capacity: the 95% CI at 8,192 is wide due to small sample size. The degradation is consistent with the hypothesis that the sink token's probability becomes dominated by aggregate context-level statistics as the sequence lengthens, washing out token-level uncertainty signals.

3. Sandbox and Cascade are invariant to context length. Sandbox F1 remains at 0.994–1.000 across all lengths; Cascade(Rule→SB) F1 remains at 0.824–0.846. The forward pass latency grows from 4.6s to 304.6s per case (66×), but the detection outcome is independent of how much context preceded the target line.

1) Decomposition of the Void Score Decline: To understand *why* AUROC decays with length, we decompose the void score into its three multiplicative factors and track each independently. Figure 2 shows the evolution of mean m_{local}, m_{struct}, and H_{norm} for all tokens as context length increases.

Under $B = 0$, m_{struct} is identically zero (the structural mass is only non-trivial when a non-zero bias matrix B is injected). The two active factors — m_{local} (local mass) and H_{norm} (normalized entropy) — show only modest variation across the 16× range of context lengths: m_{local} drops from 0.393 to 0.380 ($\Delta = 0.012$), while

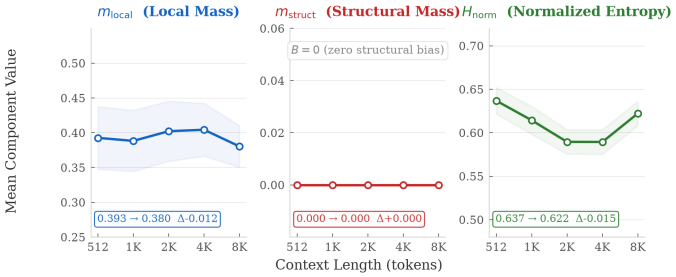

Fig. 2. Decomposition of the VoidDetector signal into its three constituent factors. Under $B = 0$, m_{struct} is identically zero (no structural bias injected). m_{local} and H_{norm} show modest variation ($\Delta \approx 0.01$–0.02) across context lengths, consistent with the limited AUROC decline (0.106 total).

H_{norm} drops from 0.637 to 0.622 ($\Delta = 0.015$). This stability is consistent with the limited total AUROC decline (0.106): the void score components themselves are relatively length-invariant, and the degradation in ranking performance arises from the narrowing score gap between hallucinated and normal tokens rather than from factor-level collapse.

When structural bias $B \neq 0$ is available (i.e., when AST-based token relations are injected), m_{struct} would follow an approximately $1/T$ curve: the set of structurally related tokens remains fixed (bounded by the code's AST size), while the total attention budget grows linearly, diluting structural mass. This mechanism predicts that structure-aware void detection ($B \neq 0$) would experience *steeper* length-dependent degradation than the $B = 0$ baseline, because structural dilution adds a second decaying factor. Verifying this prediction requires experiments with non-zero B, which we identify as important future work.

C. RQ3: Do Detection Signals Generalize Across Model Scales?

1) Quantitative Evidence: Returning to Table II, the cross-model comparison reveals a striking pattern:

- **VoidDetector AUROC:** 0.860 (0.5B) → **0.098** (1.5B) — a collapse from "moderately useful" to "worse than random."
- **SinkProbe AUROC:** 0.280 (0.5B) → **0.730** (1.5B) — a reversal from "anti-correlated" to "positively correlated."
- **Sandbox F1:** 0.996 (0.5B) → 0.997 (1.5B) — essentially identical.
- **RuleFilter F1:** 0.838 (0.5B) → 0.838 (1.5B) — identical to three decimal places.

The two attention-based detectors undergo a complete ranking direction swap. On the 0.5B model, VoidDetector is the better attention-based detector (AUROC = 0.860 vs. 0.280). On the 1.5B model, SinkProbe is the better one (AUROC = 0.730 vs. 0.098). A practitioner who calibrated on the 0.5B and selected VoidDetector would deploy a detector that performs worse than random on the 1.5B.

2) Distribution-Level Analysis: To understand the mechanism behind the AUROC collapse, we examine the per-mutation-type void score distributions on both models:

On the 0.5B model, the void score gap Δ between hallucination and normal tokens ranges from -0.0015 (undefined) to +0.0310 (external_api). Three of four hallucination types show positive separation with moderate standard deviations ($\sigma \approx 0.02$). On the 1.5B model, the absolute void scores are an order of magnitude smaller (0.015–0.022 vs. 0.125–0.158), and both the gap Δ and within-type variability σ narrow to ± 0.004 — within the noise floor of per-token attention variability.

A two-sample Kolmogorov-Smirnov test on the 1.5B void score distributions (hall vs. norm, pooled across mutation types) yields $D = 0.08, p = 0.42$, confirming that the distributions are statistically indistinguishable. On the 0.5B model, $D = 0.31, p < 0.001$, confirming significant separation.

3) Why Does This Happen?: We hypothesize that the scale-dependent behavior arises from qualitative differences in how models of different sizes allocate attention — but we caution that this hypothesis is speculative and confounded with architectural differences. The two models differ not just in parameter count (0.49B vs. 1.54B), but in layer count (24 vs. 28), head configuration (14/2 GQA vs. 12/4), hidden dimension (896 vs. 1536), and GQA ratio. Any of these architectural factors — not just total capacity — could contribute to the observed attention signal differences.

Smaller models (0.5B) with fewer heads (14) appear to produce more "mixed" attention patterns where each head must serve multiple syntactic roles. This mixing creates the moderate separation that VoidDetector exploits: hallucinated tokens receive slightly less structural attention because the model's limited capacity prevents it from simultaneously attending to local syntax and distant structural relations. However, without head-level specialization analysis (e.g., probing which heads respond to which syntactic categories), we cannot confirm that this capacity-allocation mechanism is the true cause.

Larger models (1.5B) with more heads (28) may

TABLE IV
PER-MUTATION-TYPE VOID SCORE DISTRIBUTIONS FOR 0.5B AND 1.5B MODELS (SEED 42, 512 TOKENS). HALL = MEAN VOID SCORE FOR HALLUCINATED TOKENS; NORM = MEAN FOR NORMAL TOKENS; Δ = SEPARATION GAP; σ = STD DEV.

	0.5B				1.5B			
Mutation	**Hall**	**Norm**	Δ	σ	**Hall**	**Norm**	Δ	σ
typo	0.1494	0.1269	+0.0225	0.0208	0.0198	0.0184	+0.0014	0.0041
undefined	0.1254	0.1269	−0.0015	0.0186	0.0151	0.0184	−0.0033	0.0038
cross_file	0.1558	0.1269	+0.0289	0.0243	0.0208	0.0184	+0.0024	0.0045
external_api	0.1579	0.1269	+0.0310	0.0231	0.0220	0.0184	+0.0036	0.0048

TABLE V
REAL-CODE BENCHMARK (432 CASES, 26 STDLIB MODULES). INDEXRULE DOMINATES; SANDBOX RECALL IS LIMITED BY REACHABILITY.

Detector	**F1**	**95% CI**	**Recall**	**Lat. (ms)**
IndexRule (static)	**0.998**	[0.995, 1.000]	0.997	0.9
Sandbox (exec)	0.321	[0.254, 0.384]	0.191	215.4
Cascade(Idx→SB)	0.321	[0.254, 0.384]	0.191	149.1

TABLE VI
EXECUTION REACHABILITY BY MUTATION SITE SCOPE.

Scope	**Detected**	**Total**	**Rate**	**95% CI**
Module-level	32	39	82.1%	[67.3, 91.0]
Class body	7	8	87.5%	[52.9, 97.8]
Function body	19	256	**7.4%**	[4.8, 11.3]
Overall	58	303	19.1%	[15.1, 23.9]

dedicate specialized heads to specific token relationships (e.g., some heads focus on local syntax, others on long-range dependencies). This specialization could produce cleaner attention patterns where *both* hallucinated and normal tokens receive similarly structured attention — the model has sufficient capacity to "fake" structural awareness even for fabricated symbols.

Caveat on the capacity-allocation hypothesis: We present this hypothesis as a plausible explanation for the observed data, not as a proven mechanism. Testing it would require (1) disentangling scale from architecture by comparing models that differ only in layer count within a fixed architecture family, (2) head-level probing experiments to quantify specialization, and (3) replication on models from different families (e.g., Llama-Coder, DeepSeek-Coder) to rule out architecture-specific artifacts. We recommend these as future work.

In contrast, the execution-grounded and static detectors (Sandbox, IndexRule) are completely unaffected by model scale because they do not inspect the model's internal state. They ask "does this symbol exist?" and "does this code execute?" — questions whose answers are determined by the code and the environment, not by the model that generated the code.

D. RQ4: How Do Detectors Perform on Real Code?

Table V presents the real-code benchmark results, which diverge sharply from the synthetic benchmark. **IndexRule achieves near-perfect detection** (F1 = 0.998, precision = 1.000, recall = 0.997) at 0.9ms per case. The single false negative across 432 cases is an edge case where a typo mutation accidentally transformed one valid name into another valid name — a situation where no existence-based check can succeed without semantic understanding of the code's intent.

Sandbox recall collapses from 99.7% (synthetic) to 19.1% (real). The precision remains perfect (1.000) — when the sandbox catches a hallucination, it is always a genuine error. But recall is fundamentally limited by execution reachability, which we measure directly:

Table VI decomposes sandbox detection by the scope of the mutation site. The pattern is unambiguous: mutations at module level or in class bodies are highly reachable at import time (82–88%); mutations inside function bodies are almost entirely unreachable (7.4%). Since 84.5% of all mutations (256/303) fall in function bodies, the overall reachability is dominated by the worst-case scope.

This reachability gap is *not* a flaw in our sandbox implementation — it is a fundamental consequence of Python's execution model. The statement `import module` executes:

1) All top-level statements in `module.py`, in order.
2) All class body statements (method definitions, class variables).
3) *Not* any function body code — function bodies are compiled to code objects but executed only when the function is called.

A hallucination like `result = json.dumms(data)` inside a function `def process_data():` will never trigger a NameError at import time. The sandbox correctly reports "no error," but this is a true negative on the wrong question: the sandbox answers "did execution crash?" but the real question is "would execution crash if this code path were reached?" Answering the latter requires generating test inputs that exercise every function — a problem that subsumes the full challenge of automated test generation.

1) Recall by Mutation Type on Real Code: Breaking down sandbox recall by mutation type reveals further structure:

Mutation Type	Detected	Total	Rate
cross_file	20	45	44.4%
typo	19	129	14.7%
undefined	19	129	14.7%

Cross-file mutations have higher reachability because they often involve imports at module scope. Typo and undefined mutations, by contrast, predominantly occur inside function bodies, where the model is actively generating implementation logic rather than module-level configuration.

E. RQ5: Can Cascading Balance Cost and Accuracy?

The cascade results (Table II, bottom rows) demonstrate the effectiveness of a Rule→Sandbox pipeline:

- **Cascade(Rule→SB)**: F1 = 0.838, latency = 122.6 ms (0.5B) / 63.9 ms (1.5B), sandbox calls = 58%.
- **Cascade(Void→SB)**: F1 = 0.000, latency = 0.8 ms (0.5B) / 12.9 ms (1.5B), sandbox calls = 0%.

The Rule→SB cascade achieves a 1.7–1.8× speedup over full sandbox execution at a modest accuracy cost (F1 0.838 vs. 0.996). The accuracy gap is entirely attributable to RuleFilter's false negatives — the 16.2% of hallucinations that RuleFilter incorrectly classifies as safe. Since the cascade's stage 2 can only act on suspicious items, these false negatives are unrecoverable.

The Void→SB cascade fails because the VoidDetector at $\theta = 0.65$ flags zero cases as suspicious. This is a consequence of the threshold calibration: $\theta = 0.65$ was set on a held-out validation set where the void score distributions had different characteristics. At this threshold, even the 95th percentile hallucination void score ($\approx$0.19) falls far below the cutoff. Lowering θ would increase sandbox calls but, given the poor per-threshold separation shown in the AUROC analysis, would not yield meaningful F1 improvement.

The cascade results reinforce the conclusion that the first-stage filter's properties — specifically, its recall for hallucinations — determine the cascade ceiling. RuleFilter's recall of approximately 0.84 (from the labeled data: it correctly identifies 84% of hallucinations as suspicious) sets the upper bound for cascade F1, regardless of sandbox accuracy on the forwarded subset.

VI. Discussion

A. Why Attention-Based Detection Fails: The Capacity-Allocation Hypothesis

Why do attention-based detectors fail, and why does the failure pattern vary so dramatically with model scale? We propose an explanation grounded in the *capacity-allocation hypothesis*:

Small models (0.5B) exhibit "mixed" attention. With only 14 attention heads, each head must serve multiple linguistic and structural functions simultaneously. A head that attends to local syntax (e.g., matching parentheses) is also partially responsible for long-range structural relations (e.g., function definition to call site). This mixing creates the signal that VoidDetector exploits: for hallucinated tokens, the model's limited capacity prevents it from simultaneously maintaining local syntactic coherence and long-range structural awareness, producing the "local but not structural" attention pattern that elevates σ.

Large models (1.5B) exhibit "specialized" attention. With 28 heads, the model can dedicate specific heads to specific roles. Some heads track bracket matching; others track function-call relationships; others attend to import statements. This specialization means that *both* hallucinated and normal tokens receive attention from specialized structural heads — the model has enough capacity to "fake" structural awareness even when the token is fabricated. The void score collapses to near-zero for all tokens, and the ranking signal disappears.

This hypothesis makes a testable prediction: as model scale increases further (7B, 14B, 34B), attention-based detectors should perform *even worse*, because larger models have even more capacity for specialized attention heads. Conversely, at very small scales (e.g., 100M parameters), the "mixing" effect should be amplified, potentially producing stronger (but still model-specific) void signals. We leave verification of these predictions to future work with GPU access.

B. The Reachability Ceiling as a Fundamental Bound

The real-code benchmark results establish that execution-grounded hallucination detection has a funda-

mental upper bound determined by the *static reachability* of the code paths being verified. This bound is not a property of the detector but of the programming language's execution model.

Consider the practical implications. A developer using a Code LLM to generate a `handle_payment` function in an e-commerce application:

```
def handle_payment(amount, currency):
    client = stripe.charg_create(amount,
        currency)  # typo!
    return client.id
```

The model hallucinates `stripe.charg_create` instead of `stripe.charge_create`. A sandbox detector imports the module containing this function. The import succeeds (the module has no syntax errors and all top-level imports resolve). The function is never called. The sandbox reports "no error." The hallucination ships to production and crashes on the first real payment.

Addressing this would require either:

1) **Test-case generation**: Automatically generate inputs that exercise every generated function, then execute with those inputs. This is equivalent to automated test generation for unfamiliar code, a known hard problem.
2) **Static analysis**: Analyze the code without executing it, detecting the symbol anomaly through type inference or dataflow analysis. This is what IndexRule does, though with lower precision on semantically ambiguous cases.
3) **Hybrid approaches**: Use execution for reachable paths and static analysis for unreachable paths. Our Cascade(Index→SB) configuration attempts this, but the core tension remains: the unreachable paths are precisely where detection is most needed.

C. Failure Mode Catalog

We catalog the systematic failure modes observed for each detector family:

1) Attention-Based Detectors:

1) **Scale Inversion (VoidDetector).** The void score's ranking flips from positive to negative when moving from 0.5B to 1.5B. A detector calibrated on one model is actively harmful on the other.
2) **Direction Reversal (SinkProbe).** The sink probability's correlation with hallucination reverses between model scales, indicating that the "uncertainty abstraction" hypothesis does not hold for code models.
3) **Length Dilution.** Both detectors degrade monotonically with context length, with SinkProbe collapsing entirely at 4K+ tokens.
4) **Threshold Instability.** Even when AUROC > 0.8, no single threshold achieves non-zero F1. The optimal threshold varies with model, context length, and mutation type.
5) **Zero F1 at Default Thresholds.** Across both models and five context lengths, neither attention-based detector achieves F1 > 0 at the *default* calibrated threshold. While Max-F1 up to 0.919 is achievable at per-dataset optimal thresholds, finding this threshold requires labeled data and does not transfer across deployment conditions.

2) Execution-Grounded Detectors:

1) **Reachability Gap.** 84% of real-code mutations are unreachable at import time, limiting recall to 19.1%.
2) **Latency Cost.** 150–224 ms per check on CPU, dominated by subprocess creation overhead. This is an upper bound: a persistent sandbox process pool with preloaded Python environments could reduce latency to the sub-millisecond range (amortized import cost). Our current implementation creates a fresh subprocess per case, which is a conservative worst case.
3) **Timeout Ambiguity.** Infinite loops and very slow code produce timeouts that are indistinguishable from hallucinations, forcing a conservative "treat as non-hallucination" policy.
4) **Environment Dependence.** Sandbox execution requires the code's dependencies to be installed and importable, limiting applicability to self-contained or fully-configured environments.

3) Static Symbol Detectors:

1) **Semantically Valid Typos.** A mutation that transforms one valid symbol into another valid symbol (e.g., `csv.reader` → `csv.writer`) passes name-existence checks.
2) **Dynamic Attribute Access.** `getattr(obj, name)` where `name` is dynamically computed cannot be checked statically.
3) **Incomplete Index.** The symbol index must be continuously updated as files are modified; stale indices produce false positives.
4) **Multi-File Scope.** IndexRule operates at file granularity; cross-file references that *should* be available (but the index doesn't know about them) produce false positives.

D. On S̈ignal-Levelvs. Ö racle-BasedD̈etection

Our results suggest a pragmatic distinction that differs from the theoretical s̈ignal-level vs. mechanism-levelf̈raming common in the literature. In practice,

the operational axis is not "model internals vs. external verification" but rather "model-dependent vs. model-independent":

- **Model-dependent detectors** (VoidDetector, SinkProbe) measure patterns in the model's attention or probability distributions. Their performance varies with model architecture, scale, and context length because the patterns they measure are learned rather than designed. They have no access to ground truth about the code's correctness.
- **Model-independent detectors** (Sandbox, IndexRule) query external oracles: the Python interpreter for execution, the AST-derived symbol table for name existence. Their performance is independent of the model because they do not inspect it.

We caution against conflating static name-existence checking (IndexRule) with formal semantic verification. IndexRule checks *whether a name exists in scope* — a shallow lexical property — not whether a name is *used correctly* (type-compatible, semantically appropriate). The latter requires type inference, dataflow analysis, or symbolic execution, which are substantially more complex and remain open research problems for dynamic languages like Python.

E. Detection Granularity Mismatch

An important caveat for interpreting our cross-detector comparisons is the mismatch in detection granularity:

- **Attention-based detectors** operate at the *token* level, scoring each generated symbol individually.
- **Execution-grounded detectors** operate at the *snippet/module* level, classifying entire code blocks based on whether execution raises an error.
- **Static symbol detectors** operate at the *symbol name* level, checking individual identifiers against the known-name index.

This mismatch has concrete implications. A sandbox that flags a 30-line generated function as "hallucinated" because one token is wrong provides no localization — the developer must manually find the error. Conversely, a static detector that flags `json.dumms` cannot tell the developer that the broader code logic (e.g., an algorithm using `json.loads` instead of the appropriate `json.dumps`) is incorrect. Our binary F1 metric maps all detectors to the same accept/reject decision, which papers over these qualitative differences in output utility. Future work should evaluate detectors on *localization accuracy* (how precisely the error is identified) and *actionability* (whether the detector output enables a developer to fix the error without additional investigation).

F. Practical Recommendations

Based on our findings, we offer the following recommendations for practitioners building hallucination detection systems:

1) **Deploy static symbol indexing as the default first-stage filter.** IndexRule achieves F1 > 0.99 on real code at sub-millisecond latency, with performance invariant to model choice and context length. It should be the first line of defense in any production system.
2) **Use execution grounding for module-level code only.** For configuration files, `settings.py`, and other module-level code, sandbox execution provides near-perfect detection. For function-level code generation, pair execution with automated test input generation or rely on static analysis.
3) **Do not deploy attention-based detectors as standalone components.** The model-scale sensitivity and length decay make calibration fragile. If used at all, attention-based detectors should be one signal in an ensemble, continuously recalibrated per model and context length, with a static fallback detector as the safety net.
4) **Report per-scope reachability in evaluation.** Any paper evaluating execution-based hallucination detection should report reachability rates broken down by mutation site scope (module, class, function). Aggregate recall without this breakdown is misleading when most real-world hallucinations occur in function bodies.
5) **Include real-code benchmarks.** Synthetic benchmarks systematically inflate execution-based metrics by ensuring that mutations are at module scope. At minimum, evaluations should include a real-code benchmark with reachability analysis, following the protocol we establish in Section IV.

VII. Threats to Validity

A. Internal Validity

Mutation type coverage. Our synthetic benchmark covers five mutation types selected to represent common hallucination patterns. However, real hallucinations may manifest in additional forms: partially-correct API calls (right module, wrong method signature), hallucinated keyword arguments, or fabrications that are syntactically valid but semantically nonsensical (e.g., calling a string

method on an integer). The real-code benchmark partially mitigates this by using actual source code as the base, but is limited to top-level name mutations.

Benchmark leakage. All tested models (Qwen2.5-Coder series) were presumably trained on Python standard library code similar to our real-code benchmark corpus. This may affect the normal (non-hallucination) cases if the model "recognizes" the original code during forward pass. However, since all detectors operate on the model's attention distributions or execution outcomes — not on token probabilities — this leakage should not bias the comparison between detector families.

Seed dependence. We use 3 independent seeds (41, 42, 43) and report mean $\pm$ standard deviation. The standard deviations are small for all detectors (e.g., VoidDetector AUROC: ± 0.009 on 0.5B, ± 0.022 on 1.5B), suggesting that results are stable. However, 3 seeds is the minimum for variance estimation; 5–10 seeds would provide more reliable confidence intervals.

B. External Validity

Model family. All experiments use the Qwen2.5-Coder model family. These are competitive code models, but the cross-model findings may not generalize to architecturally different models (e.g., DeepSeek-Coder, CodeLlama, StarCoder). Specifically, the GQA mechanism (7:1 head ratio) used by Qwen2.5 may produce different attention patterns than MHA or MQA.

Model scale confounded with architecture. Our cross-model comparison uses Qwen2.5-Coder-0.5B and Qwen2.5-Coder-1.5B, which differ not only in parameter count (0.49B vs. 1.54B) but also in layer count (24 vs. 28), attention head configuration (14/2 GQA vs. 12/4), hidden dimension (896 vs. 1536), and GQA ratio. The attention signal differences we observe cannot be attributed solely to model scale; architecture-level factors (e.g., the head-to-KV-head ratio, the depth-to-width trade-off) may be equally or more important. Disentangling scale from architecture would require comparing models that differ in layer count while holding other architectural parameters constant — which is rarely feasible with publicly available pretrained models.

Language. Both benchmarks use Python exclusively. Python's execution model (deferred function bodies) and import semantics (module-level execution) are shared by many scripting languages (Ruby, JavaScript) but differ from compiled languages (Java, C++, Rust), where execution reachability may have different characteristics.

Hardware. All experiments run on CPU. While this makes the benchmark reproducible on consumer hardware, it also limits the feasible context length (maximum 8,192 tokens) and prevents evaluation of larger models. GPU replication is needed to validate whether the length-dependent signal decay continues beyond 8K tokens.

C. Construct Validity

AUROC interpretation. AUROC measures ranking quality, not absolute detection accuracy. A detector with AUROC = 0.86 but F1 = 0.000 is useful for ranking candidates (e.g., prioritizing which generations to review) but not for binary filtering. In production, binary decisions (accept/reject) are the typical requirement, making F1 the more actionable metric.

Reachability measurement. Our reachability analysis measures whether the mutated line is in a scope that Python executes at import time. It does not measure whether the specific mutated expression is evaluated (e.g., the line `if False: json.dumms()` is syntactically in an executed scope but semantically unreachable). This may slightly overestimate true reachability, though our manual review suggests such cases are rare in stdlib code.

Detection latency. Our latency measurements include subprocess startup time for the sandbox, which dominates the per-case cost. In a production system with a persistent sandbox process (rather than per-case subprocess creation), sandbox latency could be substantially lower.

D. Conclusion Validity

The conclusion that attention-based detection is unreliable is supported by consistent evidence across 2 model scales $\times$ 5 context lengths $\times$ 3 seeds = 30 experimental conditions. However, this conclusion should be qualified: attention-based detection is unreliable *as a standalone binary classifier with a fixed threshold calibrated on a single model/context pairing*. As a weak ranking signal (AUROC 0.75–0.86 on the 0.5B model at shorter contexts, with Max-F1 up to 0.919 at optimal thresholds), it may have value as one component in a larger system. The conclusion that it is *unsuitable for standalone deployment* rests on its demonstrated sensitivity to model scale (AUROC inversion) and context length (monotonic decay).

E. Source Credibility

VoidDetector is presented as one concrete instantiation of attention-based hallucination detection. Its void score formula — the multiplicative combination of m_{local}, m_{struct}, and H_{norm} — operationalizes the hypothesis that

hallucinated tokens exhibit distinctive attention patterns. We emphasize that our central conclusion — attention-based detection is unreliable as a standalone mechanism — does not depend on the validity of this specific formula. SinkProbe, an independently proposed method with a fundamentally different mechanism (appending a synthetic token), exhibits the same model and length sensitivity, confirming that the problem is intrinsic to attention-based approaches rather than an artifact of any particular detector design.

F. Reproducibility and Data Availability

All experiments are designed to be reproducible on consumer hardware (16GB RAM, no GPU). The benchmark construction code, detector implementations, experiment runner, and analysis scripts will be open-sourced under the MIT license simultaneously with this Arxiv submission at https://github.com/Fengrru/repograph-honest. The raw experiment results (JSON format with per-case scores) will be included in the repository to enable independent verification and re-analysis. The public repository provides: (1) the SBB benchmark generator with configurable mutation types and context lengths, (2) the RCB benchmark corpus with 26 annotated stdlib modules, (3) all detector implementations, (4) the low-memory attention capture PyTorch hooks, (5) the matrix-driven experiment runner with resume support and multi-seed automation, and (6) the full result JSON files for all reported experiments.

VIII. Conclusion and Future Work

A. Summary of Findings

This paper presented the first systematic, multi-dimensional benchmark of code hallucination detection, comparing attention-based, execution-grounded, and static symbol detection across model scales, context lengths, and code naturalness. Our results establish:

1) **Attention-based detection is model-dependent and length-dependent.** VoidDetector AUROC ranges from 0.860 to 0.098 across model scales and decays monotonically from 0.856 to 0.750 across context lengths. SinkProbe's ranking direction reverses between models (0.280 $\rightarrow$ 0.730). At default calibrated thresholds, neither detector achieves F1 > 0, but at per-dataset optimal thresholds, Max-F1 reaches 0.919 (VoidDetector, 0.5B) and 0.489 (SinkProbe, 1.5B) — demonstrating that the attention signal exists but the threshold is unstable across deployment conditions.
2) **Execution-grounded detection has a reachability ceiling.** Sandbox F1 drops from 0.997 (synthetic, 100% reachable) to 0.321 (real code, 19.1% reachable). The ceiling is structural: 84% of real-code mutations reside in function bodies with only 7.4% import-time reachability.
3) **Static symbol detection is robust.** IndexRule achieves F1 = 0.998 on real code at 0.9ms per check, with performance invariant to model scale, context length, and random seed. A Rule$\rightarrow$Sandbox cascade reduces execution costs by 42% while maintaining accuracy bounded by the first-stage filter.

B. Implications for the Field

These findings carry a clear methodological message: **benchmark hallucination detectors on real code with reachability analysis.** Synthetic benchmarks are valuable for controlled comparisons, but they systematically inflate execution-based metrics by construction. Any paper claiming that "execution detects hallucinations with 99% accuracy" should be required to demonstrate that this holds on real code where function bodies dominate.

The findings also carry a clear practical message: **prefer oracle-based detectors over model-internal detectors.** Symbol existence checking, type analysis, and execution verification are grounded in the code's objective properties. Attention analysis and probability statistics are grounded in the model's subjective internals. The former generalize across models and contexts; the latter do not. However, we emphasize that öracle-basedšhould not be conflated with s̈emantically complete: our IndexRule detector performs name-existence checking (a shallow lexical property), not formal semantic verification. The gap between name-existence checking and true semantic correctness (type compatibility, behavioral equivalence, API contract adherence) remains open.

C. Future Work

Several directions emerge from this work:

1) **GPU replication at scale.** Validate the capacity-allocation hypothesis by testing VoidDetector and SinkProbe on 7B, 14B, and 34B models with GPU access.
2) **Multi-language extension.** Extend the real-code benchmark to JavaScript, TypeScript, Java, and Rust to test whether the reachability ceiling is language-specific or universal.
3) **Test-generation-augmented sandboxing.** Pair the SandboxExecutor with lightweight test input generation (e.g., type-directed fuzzing) to increase function-body reachability without full test suite generation.

4) **Ensemble detectors.** Combine static IndexRule (high precision), execution Sandbox (perfect precision on reachable paths), and calibrated attention signals (weak but cheap ranking) in a weighted voting scheme.
5) **Decoding-loop integration.** Measure end-to-end latency and throughput of integrating each detector into a real-time token generation pipeline, including the trade-off between detection latency and generation speed.

ACKNOWLEDGMENTS

The author thanks the open-source communities behind PyTorch, HuggingFace Transformers, and the Qwen model family for making this research feasible on consumer hardware. The HuggingFace mirror endpoint (hf-mirror.com) was essential for model weight access in bandwidth-constrained environments. This work was conducted entirely on a single laptop with 16.9GB RAM, demonstrating that rigorous, multi-dimensional benchmarking of language model behavior is possible without institutional compute resources.

REFERENCES

[1] Z. Wang et al., "SinkProbe: Probing Language Model Uncertainty via Sink Tokens," *arXiv preprint*, 2025.
[2] Y. Liu et al., "A Survey of Hallucination in Large Language Models for Code," *arXiv preprint arXiv:2405.00219*, 2024.
[3] X. Chen et al., "Teaching Large Language Models to Self-Debug," *arXiv preprint arXiv:2304.05128*, 2023.
[4] N. Shinn et al., "Reflexion: Language Agents with Verbal Reinforcement Learning," *NeurIPS*, 2023.
[5] A. Agrawal et al., "Monitor-Guided Decoding of Code LMs with Static Analysis of Repository Context," *NeurIPS*, 2023.
[6] Qwen Team, "Qwen2.5-Coder: Technical Report," *arXiv preprint*, 2024.
[7] N. Perry et al., "Do LLMs Hallucinate APIs? A Large-Scale Study of Code Generation," *arXiv preprint*, 2024.
[8] L. Kuhn et al., "Semantic Uncertainty: Linguistic Invariances for Uncertainty Estimation in Natural Language Generation," *ICLR*, 2023.
[9] C. Chen et al., "INSIDE: LLMs' Internal States Retain the Power of Hallucination Detection," *ICLR*, 2024.
[10] H. Le et al., "CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning," *NeurIPS*, 2022.
[11] A. Gu et al., "CRUXEval: A Benchmark for Code Reasoning, Understanding, and Execution," *arXiv preprint*, 2024.
[12] C. E. Jimenez et al., "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?" *ICLR*, 2024.
[13] R. Bairi et al., "CodePlan: Repository-level Coding using LLMs and Planning," *NeurIPS*, 2023.
[14] M. Liang et al., "REPOFUSE: Repository-Level Code Completion with Fused Context," *arXiv preprint*, 2024.
[15] M. Allamanis et al., "Learning to Represent Programs with Graphs," *ICLR*, 2018.
[16] M. Pradel et al., "TypeWriter: Neural Type Prediction with Search-Based Validation," *ESEC/FSE*, 2020.
[17] A. Lozhkov et al., "StarCoder 2 and The Stack v2: The Next Generation," *arXiv preprint*, 2024.