Prompt Drift as a Reliability Risk in Software Engineering

An empirical investigation into the stability of LLMs in code generation and testing pipelines.

Ethan Chen
Khoi Nguyen
Pei-Yu Lin
Pablo Rodriguez
Shuang Ma

01-26-26

The Problem

Software Engineering relies on deterministic inputs and outputs. However, LLMs exhibit "Prompt Drift" - output instability even under rigid constraints. Performance gaps between best and worst runs can reach 70%.

Project Goat: Empirically measure drift across Code Gent Summarization, and Bug Detection.

Research Questions

1 Consistency

How consistent are code LLMs across repeated identical prompts?

2 Sensitivity

How sensitive are models to semantically equivalent prompt paraphrasing?

3 Model Comparison

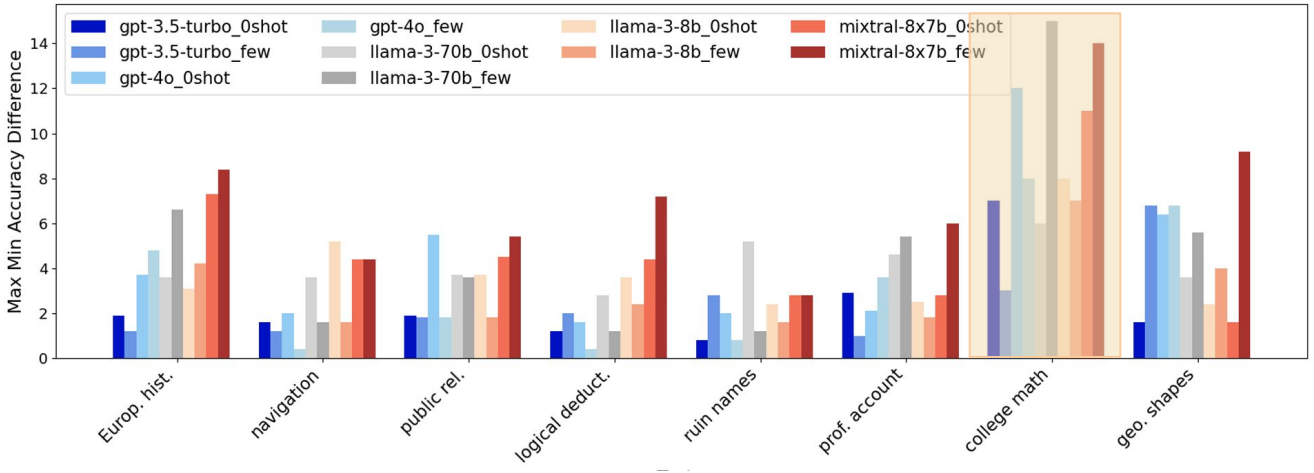
Which models (GPT-4 vs. Llama-3/CodeLlama) exhibit the highest stability?

4 Drivers

What task characteristics contribute most to instability?

Paper 1 - Overview

Source: Atil et al., Non-Determinism of 'Deterministic' LLM Settings



Even with determinism is an engineering illusion.
Instability is a system feature.

Key Findings

1. Investigation: 5 Models x 8 Tasks x 10 Runs.

2. The Gap: Up to 15% accuracy variance across identical runs.

3. The Extremes: Performance gap between 'Best' and 'Worst' run reached 70%.

4. Conclusion: Non-demintized: Non-determinism is likely inherent to efficient compute infrastructure (batching, hardware).

Paper 1 - Metrics

TARr (Raw Agreement)

Output 1: “The value is 5”
Output 1: “The value is 5”

MATCH

TARa (Answer Agreement)

Output 1: “Answer: 5”
Output 1: “The result is 5”

SEMANTIC MATCH

We separate ‘string stability’ from ‘semantic stability’.
A model can be accurate but unstable

Paper 1 - Relevance

System Factors

- Sparse Operations
- Continuous Batching

Inference Noise

Methodology Control

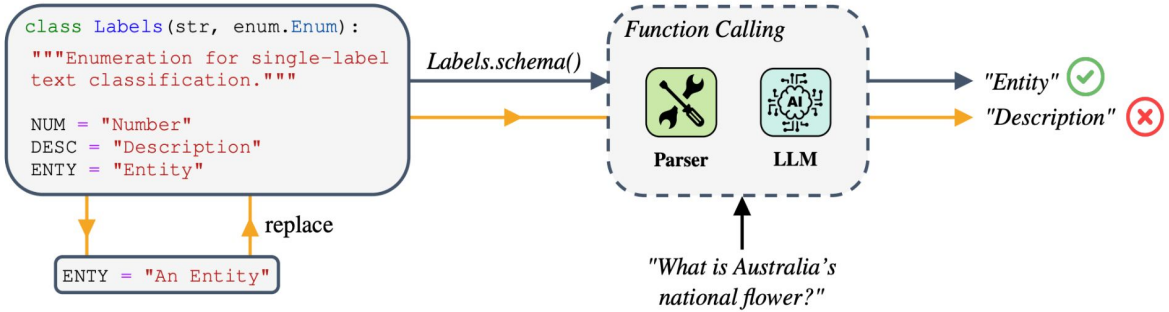
- N=10 repeated runs per prompt
- Measure drift, not just accuracy
- Use TAR metrics for summarization

Atil et al. (2024)

Confirms ‘Drift’ is an infrastructure reality
Establishes TAR baselines

Paper 2 Problem Definition

Errica et al., What Did I Do Wrong? Quantifying LLMs' Sensitivity and Consistency to Prompt Engineering, NEC.



A slight change in the definition of the class "ENTY" causes a minor prompt variation that disrupts the LLAMA's prediction. This happens under the hood, making it very hard for a developer to debug the program.

Insight:

A tiny change can dramatically flip the model's prediction.
Accuracy alone does not tell developers how unstable a model is or why it fails.

How can we quantify the sensitivity of an LLM to variations of the prompt?

Sensitivity, measuring how much predictions change under semantically equivalent prompt variations
Consistency, measuring how stable predictions are across samples of the same class.

Paper 2: Methodology

Metric 1: Sensitivity

How much predictions vary under prompt rephrasings

- Defined as **normalized entropy** of distribution:

$$S_\tau(\boldsymbol{x}) = -\mathbb{E}_{y\sim p_\tau(\cdot|\boldsymbol{x})}[\ln p_\tau(y|\boldsymbol{x})]/\ln(C),$$

- Expected sensitivity:

$$S_\tau = \mathbb{E}_{\boldsymbol{x}}[S_\tau(\boldsymbol{x})] \approx \frac{1}{N}\sum_{i=1}^{N} S_\tau(\boldsymbol{x}_i)$$

Metric 2: Consistency

How similar predictions are across samples of the same class

- Total Variation Distance (TVD):

$$\mathrm{TVD}(p,q) = \frac{1}{2}\sum_{c=1}^{C}|p(c)-q(c)|,$$

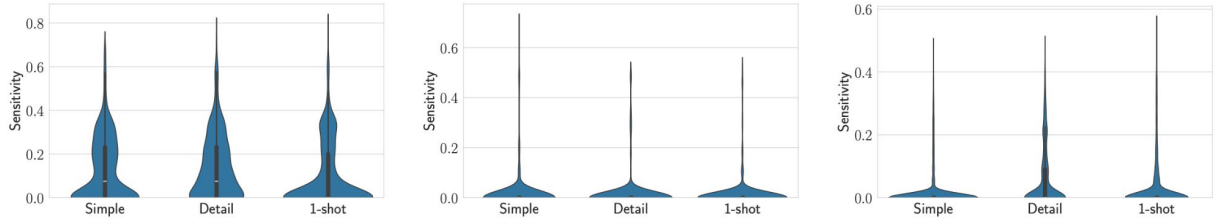
- Pair-wise consistency

$$C_y(\boldsymbol{x},\boldsymbol{x}') = 1 - \mathrm{TVD}(p_\tau(\cdot|\boldsymbol{x}), p_\tau(\cdot|\boldsymbol{x}')]$$

- Expected consistency:

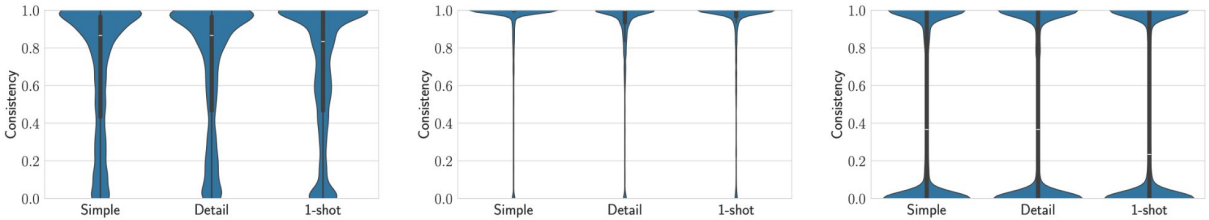
$$C_y = \mathbb{E}[C_y(\boldsymbol{x},\boldsymbol{x}')] \approx \sum_{\boldsymbol{x},\boldsymbol{x}'\in\mathcal{D}_y} \frac{C_y(\boldsymbol{x},\boldsymbol{x}')}{|\mathcal{D}_y|^2}.$$

Paper 2: Results



(a) TREC (b) CB (c) DBPedia

Sensitivity for each sample of the dataset according to different prompting strategies.



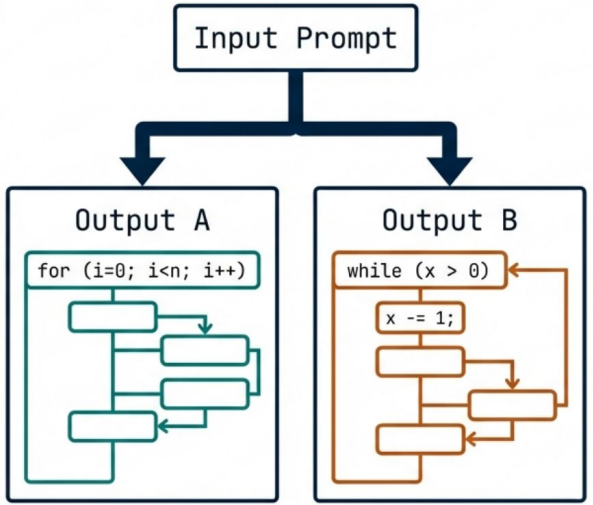
(a) TREC (b) DBPedia (c) WoS

Violin plot of the Llama3 consistency over samples of the same classes, arranged by prompting technique, on different datasets.

Paper 3 Motivation

Measuring LLM Code Generation Stability via Structural Entropy

- LLMs often generate different code for the same prompt.
- This variability reduces reliability and reproducibility.
- Existing metrics focus on correctness, not consistency.
- They study how to directly measure code generation stability



Paper 3 Methodology

Phase1: Subtree Extraction

Input: Generation Code(Ta,Tb)

Action: Parse to AST -> Extract depth-bounded subtrees

Output: canonical encoding

Phase2: Empirical Distributions

Actions: Count how often each subtree pattern appears.

Calculation: Count frequencies -> Normalize to vectors(P,Q)

Output: subtree frequency vector

Phase3: Similarity Matrix

Action:Compare subtree distributions between code samples.

Metrics: Structural Cross-Entropy (SCE) and Jensen Shannon Divergence (JSD)

Result: Stability Score[0,1]

Paper 3 Conclusion

Model	Language(Task)	Avg. BLEU	Code BLEU	Pass@1	Pass@5	TSED	SCE(structural)	SCE	JSD(structural)	JSD
LLaMA-3.1 8B it	Python BigCodeBench	0.428	0.669	0.289	0.481	0.785	0.798	0.656	0.940	0.898
Qwen-2.5 7B it		0.596	0.716	0.339	0.482	0.765	0.823	0.726	0.947	0.913
Qwen-2.5-Coder 7B		0.614	0.715	0.373	0.517	0.764	0.823	0.722	0.951	0.918
LLaMA-3.1 8B it	SQL Spider	0.495	N/A	0.673	0.824	0.832	0.729	0.669	0.934	0.905
Qwen-2.5 7B it		0.770	N/A	0.716	0.790	0.921	0.886	0.850	0.971	0.958
Qwen-2.5-Coder 7B		0.664	N/A	0.787	0.860	0.927	0.822	0.781	0.962	0.946

- Results show that even when pass@k is high, stability can still be low.
- JSD scores are generally high, meaning overall structure is similar.
- But SCE scores are lower, showing that token-level and fine-grained differences are common.
- This reveals instability that pass@k and BLEU miss.

Paper 4: Motivation and Problem

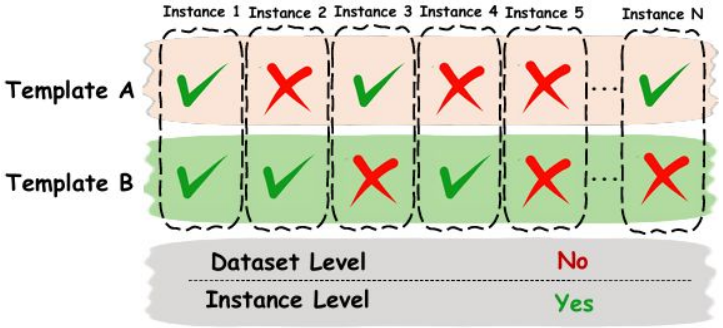
ProSA: Assessing and Understanding the Prompt Sensitivity of LLMs

Main idea

- LLM answers change a lot when prompts are worded differently
- Current evaluations hide this problem

What the paper does

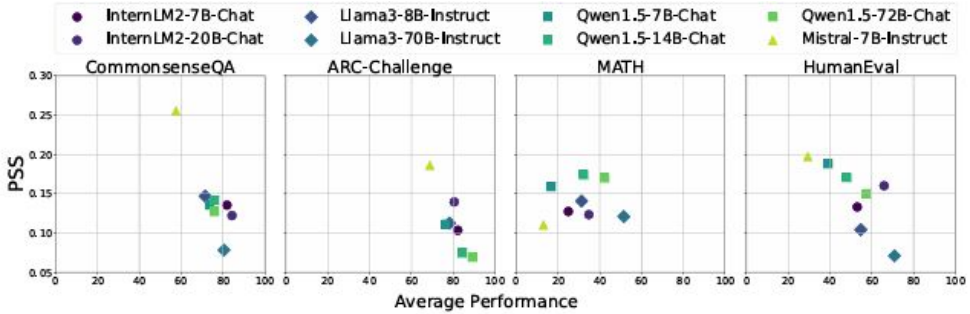
- Studies prompt sensitivity
- Looks at same question, different prompts
- Proposes a new metric: PromptSensiScore (PSS)



Paper 4 Metrics and results

Key idea: Instance-level evaluation

- Not dataset averages
- Look at each question individually



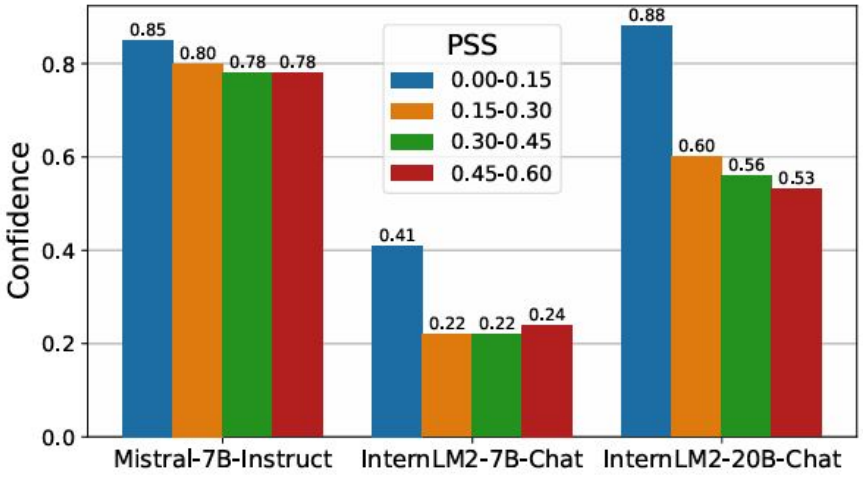
PromptSensiScore (PSS)

- Compare model answers across multiple prompt versions
- High PSS = unstable model
- Low PSS = robust model

Main findings

- Larger models are usually more stable
- Math and reasoning tasks are very sensitive
- Adding few-shot examples helps a lot

Paper 4: Explanation



Why prompt sensitivity happens

- It's linked to model confidence
- Low confidence → answers flip easily

Why this matters

- Benchmarks may overestimate model quality
- Users may get unreliable answers
- Evaluation should include prompt robustness

Takeaway

- Prompt sensitivity is real
- It's measurable
- It should be part of LLM evaluation

Paper 5: Motivation and Problem

Benchmarks and Metrics for Evaluations of Code Generation: A Critical Review

TABLE I
PERFORMANCE COMPARISON OF LANGUAGE MODELS

Model	Base Model	Size	Year	Benchmark	Score
GPT-NEO [18]	GPT-2	125M 1.3B 2.7B	2021	HumanEval	02.97 16.30 21.37
GPT-J [19]	GPT-2	6B	2021	HumanEval	27.74
Codex [20]	GPT-3	300M 679M 2.5B 12B	2021	HumanEval	36.27 40.95 59.50 72.31
TabNine	?	?	2021	HumanEval	7.59
ChatGPT	GPT-3.5	?	2022	HumanEval	94.00
Gemini-Ultra [17]	Transformer	?	2023	HumanEval Natural2Code	74.40 74.90
Gemini-Pro [17]	Transformer	?	2023	HumanEval Natural2Code	67.70 69.60
CodeGen [21]	Auto-regressive Transformer	350M 2.7B 6.1B 16.1B	2023	HumanEval	35.19 57.01 65.82 75.00
SantaCoder [22]	Decoder	1.1B	2023	MultiPL-E	45.90
InCoder [23]	Transformer	1.1B 6.7B	2023	HumanEval	25.20 45.00
StarCoder [12]	StarCoder-Base	15.5B	2023	HumanEval MBPP	33.60 52.70

LLMs in Code Generations:

- Programming tasks categorized into D2C (Description to Code), C2D (Code to Description), and C2C (Code to Code). Code generation falls into D2C
- General purpose LLMs (ChatGPT) performs “better” in the benchmark, which conflicts with our assumptions
- HumanEval scores increased dramatically (2.97% to 94%) in one year. This signal potential problem in the accuracy of these benchmarks
- Model size and benchmark scores vary wildly. This is questionable comparability
- Benchmarks may not reflect *real* code usability or significance of differences

Paper 5 - Methodology

TABLE IV
STRUCTURE OF DATA FOR EACH BENCHMARK

Benchmark	Context Code	#Test Cases	Solution
APPS	-	+	+
HumanEval	function signature	7.7	-
MBPP	-	3	-
MathQA-Python	-	3	-
ClassEval	class skeleton	33.1	+
CoderEval	function signature	+	-
Multipl-E	function signature	3 to 7	-
DS-1000	signature	1.6	+
HumanEval+	function signature	774.8	-
CONCODE	function signature	-	-
R-benchmark	-	-	+
JuICe	-	-	+
Exec-CSN	function signature	+	+
EvoCodeBench	function signature	+	+

How current benchmark work:

- Data is sourced from code repositories, online forums, coding challenge sites, and textbooks.
- Tasks include a natural-language description and optional context code
- Context code define hint for the model and #test cases define evaluation rigor
- pass@k metric to estimate the probability of generating at least one correct solution in k trials

Paper 5 - Conclusion

TABLE V
EVALUATION PER BENCHMARK

Benchmark	Correctness	Metric	Model	Result
APPS	pass all tests	%pass@100	GPT-2 1.5B	0.68
			GPT-Neo 2.7B	1.12
			GPT-3 175B	0.06
	test pass rate	AvgTPR	GPT-2 1.5B	7.96
			GPT-Neo 2.7B	10.15
			GPT-3 175B	0.55
HumanEval	pass all tests	%pass@100	GPT-Neo-2.7B	21.37
			GPT-J-6B	27.74
			Tabnine	7.59
			Codex-12B	72.31
MBPP	pass all tests	%pass@1	Decoder only-8B	79.0
			Transformer-68B	82.8
			Lang. Model-137B	83.8
MathQA -Python	pass all tests	%pass@1	Decoder only-8B	74.7
			Transformer-68B	79.5
			Lang. Model-137B	81.2
ClassEval	pass all tests	%pass@5	GPT-4	42.0
			GPT-3.5	36.0
			WizardCoder	23.0
			StarCoder	14.0
			SantaCoder	10.0
			CodeGen	13.0
			CodeGeeX	10.0
			InCoder	8.0
			Vicuna	4.0
			ChatGLM	3.0
			PolyCoder	3.0
CoderEval (Python)	pass all tests	%pass@10	CodeGen	23.48
			PanGu-Coder	27.39
			ChatGPT	30.00
CoderEval (Java)	pass all tests	%pass@10	CodeGen	33.48
			PanGu-Coder	43.04
			ChatGPT	46.09
Multipl-E (HumanEval)	pass all tests	%pass@1	Codex	≈ 36
			CodeGen	≈ 9
			InCoder	≈ 6
Multipl-E (MBPP)	pass all tests	%pass@1	Codex	≈40
			CodeGen	≈14
			InCoder	≈ 15
DS-1000	pass all tests	%pass@1	Codex-002	41.25
			CodeGen-6B	8.4
			InCoder-6B	7.45
HumanEval+	pass all tests	%pass@100	CodeGen	≈64.0
			SantaCoder	40.6
			InCoder	≈29.8
			PolyCoder	13.6
			ChatGPT	89.8
			Vicuna	40.25
			StableLM-α	11.9
			GPT-J	25.9
			GPT-Neo	16.8
ConCode	BLEU	Avg BLEU	Retrieval	20.27
			Seq2Seq	23.51
			Seq2Prod	21.29
			User-designed	22.11
R-benchmark	Satisfactory	Avg #attempt$_k$	ChatGPT	1.6

Current benchmark inadequacy:

- Model rankings change when testing rigor increases
- Benchmark results are highly sensitive to dataset design and sourcing
- The standard pass@k metric fails to reflect real-world usability, most don't re-run multiple times to find one solution
- The #attemptk metric better reflects productivity and whether code is a "useful starting point

Takeaway

Drift Exists (Atil et al.)

Prompts Drive It (Errica et al.)

We Can Measure It (Song et al.)

Phrasing can change result

New metrics is required (Paul et al.)

To transition LLms from experiments to production components, we must treat **‘Stability’** as a first-class metric alongside **‘Accuracy’**.

End of slides.

Our Experimental Design

	Models (tentative)	Tasks	Datasets
1 2 3	GPT–4 (Baseline) Llama-3 CodeLlama	Code Generation Summarization Bug Detection Test Generation	HumanEval CodeSearchNet

Timeline

Jan-Feb: Infrastructure & Prompt Engineering

Feb: Data Collection (800+ queries)

Feb-Mar: Analysis & Final Report