Specification and Detection of LLM Code Smells

Brahim MAHMOUDI[*]
École de technologie supérieure
Montréal, Québec, Canada

Zacharie CHENAIL-LARCHER[*]
École de technologie supérieure
Montréal, Québec, Canada

Naouel MOHA
École de technologie supérieure
Montréal, Québec, Canada

Quentin STIÉVENART
Université du Québec à Montréal
Montréal, Québec, Canada

Florent AVELLANEDA
Université du Québec à Montréal
Montréal, Québec, Canada

Abstract

Large Language Models (LLMs) have gained massive popularity in recent years and are increasingly integrated into software systems for diverse purposes. However, poorly integrating them in source code may undermine software system quality. Yet, to our knowledge, there is no formal catalog of code smells specific to coding practices for LLM inference. In this paper, we introduce the concept of LLM code smells and formalize five recurrent problematic coding practices related to LLM inference in software systems, based on relevant literature. We extend the detection tool *SpecDetect4AI* to cover the newly defined LLM code smells and use it to validate their prevalence in a dataset of 200 open-source LLM systems. Our results show that LLM code smells affect 60.50% of the analyzed systems, with a detection precision of 86.06%.

CCS Concepts

• **Software and its engineering** → **Software reliability**; *Software maintenance and evolution*; *Software design engineering*; • **Computing methodologies** → Natural language processing.

Keywords

Large Language Models, LLMs, Code Smells, LLM Integration

ACM Reference Format:
Brahim MAHMOUDI, Zacharie CHENAIL-LARCHER, Naouel MOHA, Quentin STIÉVENART, and Florent AVELLANEDA. 2025. Specification and Detection of LLM Code Smells. In . ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

In recent years, Large Language Models (LLMs) have revolutionized the way information is processed and have gained increasing importance in everyday life, with the number of LLM-related publications multiplying year after year [50]. They are also integrated into a growing number of software systems [43].

However, LLMs are not always reliable [22]. Their performance and behavior can vary significantly depending on how they are used [51]. To ensure reliability and maintainability of LLM-Integrating systems (systems that use LLMs as components, ranging from simple inference to extensive agentic logic), it is essential to properly integrate them, both at the architectural level and within the source code itself [6]. Therefore, as previously done for general [14] and machine learning-specific practices [52], establishing coding guidelines, including the formalization of code smells, is essential for LLM-Integrating systems.

While prior studies have defined taxonomies of general defects in LLM-Integrating systems [43] and prompt-related defects [44], to our knowledge, there is no formal concept addressing code-specific poor practices for the integration of LLM inference in software systems.

In this paper, we introduce the concept of LLM code smells and specify five concrete cases identified from relevant literature. We define them as recurring source code patterns tied to LLM inference that, while not directly causing bugs or failures, undermine maintainability, reliability, performance, or robustness.

Our key contributions are: (i) a catalog of five concrete LLM code smells; (ii) *SpecDetect4LLM*, a dedicated static detection tool; and (iii) an empirical study on 200 Python open-source systems to assess their prevalence and validate their relevance.

2 Related Work

The concept of **code smells** is well established in software engineering. It can be defined as a recurring, low-level coding practice that degrades software quality, while not representing an explicit bug [14]. Over time, researchers have extended the concept by developing catalogs and taxonomies for specific application domains, such as for Android applications [7] and Machine Learning (ML) [52]. We essentially follow the same approach, but for LLM-Integrating systems.

The closest existing work to our study is the taxonomy of code defects for LLM-based autonomous agents introduced by **Ning et al.** [35]. While some of their defects, such as *LLM API-related Defect*, touch similar aspects to our code smells, both concepts differ significantly in scope, granularity, and effects. We focus on LLM inference calls in system source code, which occur both in simple integrations and in their agent systems [20]. Their defects are also broader and more general, adopting a higher-level view, while we precisely pinpoint poor coding practices. Finally, their defects directly cause failures and malfunctions, whereas LLM code smells harm the maintainability, reliability, performance and robustness. **Shao et al.** [43] introduce a taxonomy of defects for LLM-enabled and RAG systems.

[*]The first two authors contributed equally to this work and share first authorship.

Their defects are fundamentally different from our code smells, as they describe higher-level symptoms not explicitly tied to recurrent coding practices.

Other related work can be organized into four categories: (i) **taxonomies of failures** in LLM-Integrating systems that emphasize high-level symptoms rather than code-level practices [8, 24, 48]; (ii) **prompt-smell taxonomies** that are complementary to our study, but operate on prompts, not code [42, 44]; (iii) **anti-patterns** for LLM inference **benchmarking** rather than integration [1]; and (iiii) **analyses of defects in LLM-generated code**, whereas we study human-written source code that integrate LLMs [13, 53].

Unlike prior works that catalog agent defects, prompt smells, or runtime failures, we **define LLM code smells** at the source-code level, **create a detailed catalog**, **develop a static analysis tool (*SpecDetect4LLM*) to detect** them, and **conduct a prevalence assessment** across diverse open-source systems.

3 Catalog of LLM Code Smells

This section covers our catalog of five LLM code smells. Table 1 summarizes their sources, effects on software quality, and prevalence. While we focus on Python code, the concepts also apply across languages and use cases. Due to space constraints, we include extended explanations/definitions in our replication package [3].

3.1 LLM Temperature Not Explicitly Set (TNES)

Context: In LLM APIs, SDKs, and runtimes (e.g., OpenAI, Hugging Face, Ollama), temperature controls sampling randomness [45].
Problem: Relying on an implicit temperature reduces maintainability and reliability. Defaults temperature differ across providers/models [15, 36, 38] and may change over time [32], which harms reproducibility, portability and can silently alter behavior.
Solution: Always specify explicitly the temperature and document it. Tune by task: low (0–0.3) for precise, repeatable automation; higher (0.7–1.0) for creative generation; avoid extremes [30].
Example: In Listing 1, the red version omits temperature, while the green version makes it explicit.

```
response = openai.chat.completions.create(
- model = "gpt-4o-2024-11-20", messages = messages)
+ model = "gpt-4o-2024-11-20", messages = messages,temperature = 1.0)
```

Listing 1: LLM Temperature Not Explicitly Set (TNES)

3.2 No System Message (NSM)

Context: In role-based chat APIs and runtimes, the system message sets global behavior, constraints, and tone for the assistant.
Problem: Without a system message, the model lacks high-level guidance, which reduces consistency and adherence to constraints. Outputs become more generic and harder to control. This degrades reliability, as additional iterations or longer prompts are often required to achieve adequate results [10].
Solution: Always include a clear system message that defines roles, goals, and constraints. Keep task specifics in the user message [19].
Example: In Listing 2, the red version omits a *system* message, whereas the green version adds a concise system instruction to anchor behavior, improve consistency and response quality.

```
response = openai.chat.completions.create(
  model="gpt-4o-2024-11-20",
- messages=[{"role": "user", "content": "Explain
recursion with an example"}])
+ messages=[{"role": "system", "content": "You are a
Computer Science tutor. Answer clearly."},{"role":"user",
"content": "Explain recursion with an example"}])
```

Listing 2: No System Message (NSM)

3.3 No Model Version Pinning (NMVP)

Context: In LLM APIs, runtimes, and hubs (e.g. OpenAI, Ollama, Hugging Face), models can be called via moving aliases (e.g., *gpt-4o*) or immutable versions/snapshots (e.g., *gpt-4o-2024-11-20*). Aliases may advance as providers update models [29].
Problem: Using only a provider alias removes explicit versioning, so weights, prompts, and safety filters can change without notice and shift behavior [33]. It reduces maintainability by eroding traceability and reproducibility. Runs cannot be tied to a stable model build and portability across environments degrades as the same alias may yields different behavior.
Solution: Always specify an immutable identifier and record it with other run metadata. Update versions via change control [18, 38, 47].
Example: In Listing 3, the red version uses a moving alias, whereas the green version pins an immutable version/snapshot to ensure reproducibility and traceability.

```
response = openai.chat.completions.create(
  - model="gpt-4o", messages=messages )
  + model="gpt-4o-2024-11-20", messages=messages)
```

Listing 3: No Model Version Pinning (NMVP)

3.4 No Structured Output (NSO)

Context: LLM-Integrating systems often expect typed fields (e.g., JSON) but rely on LLM inference outputs which may not respect the format and produce raw free-form text. This smell applies when the output is later parsed, indexed, or executed assuming structure.
Problem: Without an enforced output schema, the system may receive free-form text where structured fields are expected [25]. This increases error-proneness via schema drift, missing or renamed fields, type mismatches, and silent truncation that passes as success, breaking parsers and downstream steps [25]. It degrades reliability as runs become inconsistent, data stores accumulate corrupted or hallucinated values, and execution/storage/display paths face injection [12].
Solution: Enforce structured output at the API boundary. With OpenAI, declare a JSON Schema via *response_format* (chat completions) or *text.format* (responses). With the Python SDK, you may instead bind the format directly to classes [37]. Always validate results to handle refusals or other errors [21, 37, 49].
Example: In Listing 4, the red version consumes free-form text, whereas the green version enables strict parsing and safer downstream use by enforcing a JSON schema via `response_format`.

```
# Define a JSON schema e.g. result_schema = ...
- response = openai.chat.completions.create(
-     model="gpt-4o-2024-11-20", messages=messages)
+ response = openai.chat.completions.create(
+     model="gpt-4o-2024-11-20", response_format={
+         "type": "json_schema",
+         "json_schema": {"name": "Result", "schema":
```

Table 1: Overview of LLM code smells: sources, effects, prevalence, and detection precision.

Code Smell	Source			Effect				Prevalence		Precision		
	Literature	Grey	Empirical	RO	P	M	R	%	##	TP	FP	P(%)
NSO	[25]	[12, 21, 28, 31, 37, 38, 49]	[5, 27]	✓			✓	40.50%	81/200	16	4	80.00
UMM	[9, 17]	[2, 4, 15, 16, 28, 38, 39]		✓	✓	✓		38.00%	76/200	19	3	86.36
TNES	[30, 32]	[4, 15, 18, 36, 38, 45]				✓	✓	36.50%	73/200	19	4	82.61
NMVP	[33, 41, 46, 47]	[4, 15, 18, 29, 40]				✓	✓	36.00%	72/200	22	1	95.65
NSM	[19, 34]	[10, 11, 18, 38]				✓	✓	34.50%	69/200	18	3	85.71

* *Sources* - Literature: peer-reviewed literature; Grey: grey literature (official docs, tech reports, blogs); and Empirical: primary empirical data (e.g., commits, repos).
* *Effects* - RO: Robustness (error-prone); P: Performance (cost, latency, memory); M: Maintainability (reproducibility, portability); R: Reliability (response quality, consistency, stability over time).
* *Prevalence* - % : affected systems; ## = number of affected system by the code smell/total number of systems .
* *Precision* - TP: True Positive; FP: False Positive; P: Precision.

```
+          result_schema}}, messages=messages)
```

Listing 4: No Structured Output (NSO)

3.5 Unbounded Max Metrics (UMM)

Context: Hosted LLM APIs (e.g., OpenAI, Anthropic) expose finite token windows, per-request output caps, and rate limits. Pipelines that ignore these constraints, or omit their own limits on concurrency and response size, are prone to throttling and partial outputs.
Problem: Leaving token budgets, timeouts, and retries unbounded undermines system robustness and performance. Unspecified token budgets may yield outputs that exceed context limits (truncating mid-structure or return 400), overload downstream parsers, or inflate token costs and memory usage [9]. Not specifying timeouts and retry limits can cause requests to hang indefinitely, leading to unpredictable latency, rising costs, and reduced throughput as resources are tied up by long-running calls [2]. Not specifying these values also hinders maintainability, as defaults may change over time and tracking settings is essential for reproducibility.
Solution: Always bound and adjust the *max_output_tokens*, *timeout*, and *max_retries* parameters [39]. Monitoring the number of input tokens is also recommended.
Example: In Listing 5, the red version leaves metrics unbounded, whereas the green enforces bounded metrics.

```
- client = OpenAI()
- response = client.responses.create(model=
      "gpt-4o-2024-11-20", input=prompt)
+ client = OpenAI(timeout=20, max_retries=3)
+ response = client.responses.create(
+     model="gpt-4o-2024-11-20",
+     input=prompt, max_output_tokens=256)
```

Listing 5: Unbounded Max Metrics (UMM)

4 Methodology

Figure 1 presents our three-step methodology. Step 1 explains the creation of the catalog, step 2 covers the development of the detection tool, and step 3 presents the prevalence study.

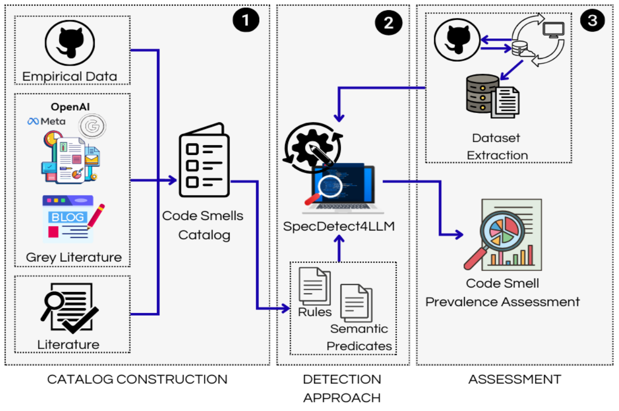

Figure 1: Workflow of the methodology

4.1 Step 1: Catalog Construction

Goal: Consolidate evidence from multiple sources to define a catalog of *LLM code smells* with precise, implementation-oriented specifications.
Inputs: i) Academic literature on LLM engineering and efficiency/reliability; ii) Grey literature (engineering blogs, provider documentations); iii) Empirical artifacts (GitHub commits, Stack Overflow (SO) posts). We follow the multi-source triangulation [23].
Outputs: A *LLM code smells Catalog* where we described each code smell using a common pattern: *Name*, *Context*, *Problem*, *Solution*, *Example*, *Sources*, *Effect*.
Procedure: We consolidate the final catalog in five steps, following established guidelines for systematic mapping studies in software engineering [23]. (1) *Search*: compose query families that pair LLM integration terms (e.g., "temperature", "token budget") with quality terms (e.g., "technical debt", "defect", "best practice"). (2) *Screen*: title/abstract screening, then full-text reads; record evidence type and strength. (3) *Synthesize*: normalize synonyms, merge duplicates, and draft code smell cards using a fixed schema. (4) *Triangulate*: map each code smell to providers/APIs and cross-validate with their documentation and code examples. (5) *Validate*: dual-review of each code smell, resolve disagreements by example-driven discussion.

A candidate code smell is admitted if it (i) is observable at the code level in LLM-Integrating systems; (ii) has cross-source support

(>5 of academic / grey / community / code change); and (iii) has an actionable remediation applicable across providers.

4.2 Step 2: Detection Approach

Goal: Develop a detection tool for LLM code smells.
Inputs: The LLM code smell catalog from Step 4.1.
Outputs: *SpecDetect4LLM*, an LLM integration static detection tool.
Procedure: We extend *SpecDetect4AI* [26], a static analyzer that detects AI-specific code smells through Domain Specific Language-defined rules. These rules allow practitioners to use semantic predicates to detect coding patterns at a high level, avoiding direct abstract syntax tree manipulation. Therefore, we: (1) add LLM-specific detection rules derived from our catalog; (2) introduce new reusable semantic predicates for common LLM integration idioms. *SpecDetect4LLM* is released as a versioned module of *SpecDetect4AI* with comprehensive tests and documentation. We select *SpecDetect4AI* because it is an open-source tool that (i) demonstrated high accuracy against prior baselines, (ii) offers an extensible specification-driven design, and (iii) has strong reproducibility properties.

4.3 Step 3: Prevalence Assessment

Goal: Run *SpecDetect4LLM* on a dataset of 200 LLM-Integrating systems to deliver (i) per-smell prevalence and co-occurrence statistics and (ii) a precision-oriented manual audit demonstrating the practical relevance of the catalog.
Inputs: LLM-Integrating Systems Dataset: a curated, versioned systems list with metadata.
Outputs: Prevalence Assessment: per-smell prevalence, density, and co-occurrence statistics, with breakdowns by framework, provider and application type.
Procedure: Following the three-stage setup in Figure 1, we (i) extract a dataset of open-source LLM-Integrating systems, and (ii) assess the prevalence in the extracted dataset.

i) *Dataset Extraction*: We assemble a dataset of 200 LLM-Integrating systems. First, we collect 100 open-source systems via the GitHub API, filtering by LLM keywords (e.g. *openai*, *llama*), programming language (Python), popularity (*stars:>20*), and recency (sorted by last update). Then, we automatically verify LLM integration by parsing dependency files (e.g., *requirements.txt*).

To strengthen external validity, we combine this collection with the 100 systems from Shao et al. [43]. This increases sample diversity as their set captures widely used and previously studied systems, while our crawl emphasizes more recent and actively maintained projects. We unify both sources, remove overlaps, and harmonize metadata (e.g. stars, last update). This construction yields a balanced dataset spanning established and emergent LLM-Integrating systems, suitable for robust empirical validation.

ii) *Code Smell Prevalence Assessment*: We execute *SpecDetect4LLM* over the dataset under a standardized configuration, recording findings per file and per system. We then conduct a dual-review of a stratified sample of detections (reporting Cohen's κ) and audit false positives with targeted negative fixtures. For each code smell, we draw random samples until we reach at least 20 detections spanning at least 5 distinct systems. Each instance is labeled as *true positive* (TP) or *false positive* (FP) using a pre-defined decision sheet, enabling us to report per-smell precision $P = \frac{\text{TP}}{\text{TP+FP}}$. We did not estimate recall as it would require exhaustive file-level ground truthing and is therefore out-of-scope for this study. Likewise, the small precision sample size constrains the accuracy of our effectiveness assessment. Instead, we position our results as a precision-oriented prevalence study. All artifacts of this assessment are available in our replication package [3].

5 Empirical Validation

We executed *SpecDetect4LLM* [3] on the full dataset of 200 systems and obtained 6,337 alerts across 60.50% (121) of them.
Observation 1-*depth versus breadth*: NMVP produces the most alerts (2,472) yet appears in fewer projects (72/200) than other smells. This pattern is consistent with alias propagation: once a moving tag (e.g., *gpt-4o*) is adopted, it is repeated across many call sites inside the same projects, inflating counts.
Observation 2-*friction shapes prevalence*: NSO and UMM have fewer raw alerts than NMVP, but affect the largest share of projects (NSO: 81/200, 40.50%; UMM: 76/200, 38.00%). Both require additional engineering work, schema design and strict parsing/validation for NSO and explicit limits on retries, timeouts, response size for UMM. Therefore, they surface early and persist across stacks. In the same vein, TNES is both frequent (1,374) and widespread (73/200), which matches the common practice of relying on framework defaults during prototyping and never revisiting them.
Observation 3-*prompt hygiene lags integration*: NSM is the least frequent (480) yet still present in more than half affected projects (69/121). Many teams wire up API calls before standardizing prompt roles, which weakens reproducibility and makes behavior sensitive to minor prompt edits.
Reliability of findings: A stratified manual audit shows high precision across the detection of all LLM code smells, for an average of 86.06% (Table 1). The imperfections in the detection are mainly due to the limits *SpecDetect4AI* [26], which lacks support for dynamic detection and context-based analysis. Recall was not estimated, but the precision levels indicate that the observed prevalence patterns reflect real defects rather than noise.
Implications: Configuration and governance issues (NMVP, TNES) concentrate within projects as habits propagate, while engineering-effort code smells (NSO, UMM) diffuse broadly because removing them requires cross-cutting refactoring. Improving governance and reducing integration friction are likely to shift the overall quality profile of LLM-Integrating systems.

6 Future Plans

Our future research directions are clear. First, we plan to expand the catalog and further formalize our taxonomy by documenting additional LLM code smells and defining categories. This will be done through further research in the scientific literature and by analyzing common practices in open-source projects. Second, we will extend *SpecDetect4LLM* to include the newly documented smells and validate its detection effectiveness (precision and recall) through controlled experimentation. We will explore dynamic detection to capture execution-dependent effects beyond static detection. Finally, we will measure the impact of the identified code smells on the performance, robustness, reliability and maintainability of software systems through empirical studies.

7 Conclusion

In this work, we introduced the concept of LLM code smells and presented a catalog of five, derived from relevant literature. We extended *SpecDetect4AI* [26] to statically detect these smells and used it to study their prevalence in 200 open-source systems. We found that all LLM code smells were present in 60.50% of the studied projects, with a precision of 86.06%. These findings confirm their relevance and the necessity of such a catalog and a dedicated detection tool. Overall, this work lays the foundation for providing clear and practical coding guidelines to improve code quality in LLM-Integrating systems, and the necessary resources to support their adoption by practitioners.

References

[1] Amey Agrawal, Nitin Kedia, Anmol Agarwal, Jayashree Mohan, Nipun Kwatra, Souvik Kundu, Ramachandran Ramjee, and Alexey Tumanov. 2025. On Evaluating Performance of LLM Inference Serving Systems. arXiv:2507.09019 [cs.LG]
[2] Amazon Web Services. 2025. Timeouts, retries and backoff. https://aws.amazon.com/fr/builders-library/timeouts-retries-and-backoff-with-jitter. Accessed 2025-09-25.
[3] anonymous. 2025. Replication_Package_LLM-Integrating_code_smells. https://anonymous.4open.science/r/Code_Smell_LLM-E933.
[4] Anthropic. 2025. Claude Documentation (API, Models). https://docs.claude.com/
[5] arena-ai. 2025. *structured-logprobs*. https://github.com/arena-ai/structured-logprobs OS library: enhances OpenAI Structured Outputs with token logprobs.
[6] Alessio Bucaioni, Martin Weyssow, Junda He, Yunbo Lyu, and David Lo. 2025. A Functional Software Reference Architecture for LLM-Integrated Systems. arXiv:2501.12904 [cs.SE] https://arxiv.org/abs/2501.12904
[7] Silvio G. Carvalho, Maurício Aniche, João Veríssimo, Alessandro Garcia, Vitor Alves, and Rohit Gheyi. 2019. An empirical catalog of code smells for the presentation layer of Android apps. *Empirical Software Engineering* 24, 6 (Dec. 2019), 3546–3586. doi:10.1007/s10664-019-09768-9
[8] Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. 2025. Why Do Multi-Agent LLM Systems Fail? arXiv:2503.13657 [cs.AI]
[9] Zixi Chen, Yinyu Ye, and Zijie Zhou. 2025. Adaptively Robust LLM Inference Optimization under Prediction Uncertainty. arXiv:2508.14544 [cs.LG] https://arxiv.org/abs/2508.14544
[10] Dan Cleary. 2025. *System Messages: Best Practices, Real-world Experiments & Prompt Injection Protectors.* https://www.prompthub.us/blog/everything-system-messages-how-to-use-them-real-world-experiments-prompt-injection-protectors PromptHub Blog.
[11] cyz3a5c0v1. 2023. What is the use case of System role. https://stackoverflow.com/questions/76272624/what-is-the-use-case-of-system-role
[12] Developer Service. 2025. *A Practical Guide on Structuring LLM Outputs with Pydantic.* https://dev.to/devasservice/a-practical-guide-on-structuring-llm-outputs-with-pydantic-50b4
[13] Ali Mohammadi Esfahani, Nafiseh Kahani, and Samuel A. Ajila. 2024. Understanding Defects in Generated Codes by Language Models. arXiv:2408.13372 [cs.SE] https://arxiv.org/abs/2408.13372
[14] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: improving the design of existing code.* Addison-Wesley Longman Publishing Co., Inc., USA.
[15] Google. 2025. Gemini API — Google AI for Developers. https://ai.google.dev/
[16] Google Cloud. 2025. Quotas and limits — BigQuery. https://cloud.google.com/bigquery/quotas
[17] Tingxu Han, Zhenting Wang, Chunrong Fang, Shuyuan Zhao, Shiqing Ma, and Ziang Chen. 2025. Token-Budget-Aware LLM Reasoning. In *ACL 2025*. Association for Computational Linguistics, 24842–24855. doi:10.18653/v1/2025.findings-acl.1274
[18] Hugging Face. 2025. Transformers Documentation (Generation, Chat Templates, Model Revisions). https://huggingface.co/docs/transformers
[19] Minbyul Jeong, Jungho Cho, Minsoo Khang, Dawoon Jung, and Teakgyu Hong. 2025. System Message Generation for User Preferences using Open-Source Models. arXiv:2502.11330 [cs.CL] https://arxiv.org/abs/2502.11330
[20] Zixuan Ke, Fangkai Jiao, Yifei Ming, Xuan-Phi Nguyen, Austin Xu, Do Xuan Long, Minzhi Li, Chengwei Qin, Peifeng Wang, Silvio Savarese, Caiming Xiong, and Shafiq Joty. 2025. A Survey of Frontiers in LLM Reasoning: Inference Scaling, Learning to Reason, and Agentic Systems. arXiv:2504.09037 [cs.AI] https://arxiv.org/abs/2504.09037
[21] Daniel Kharitonov. 2024. *Enforcing JSON Outputs in Commercial LLMs.* https://medium.com/data-science/enforcing-json-outputs-in-commercial-llms-3db590b9b3c8 Towards Data Science.
[22] Aisha Khatun. 2024. *Uncovering the Reliability and Consistency of AI Language Models: A Systematic Study.* Ph. D. Dissertation. University of Waterloo. https://uwspace.uwaterloo.ca/items/e01e11a6-e033-4f6a-85c6-849fba74e039
[23] Barbara Kitchenham and Stuart Charters. 2007. *Guidelines for performing Systematic Literature Reviews in Software Engineering.* Technical Report EBSE-2007-01. EBSE 2007. https://www.elsevier.com/__data/promis_misc/525444systematicreviewsguide.pdf
[24] Pierre Le Jeune, Jiaen Liu, Luca Rossi, and Matteo Dora. 2025. RealHarm: A Collection of Real-World Language Model Application Failures. In *LLMSEC 2025*, Leon Derczynski, Jekaterina Novikova, and Muhao Chen (Eds.). 87–100.
[25] Michael Xieyang Liu, Frederick Liu, Alexander J. Fiannaca, Terry Koo, Lucas Dixon, Michael Terry, and Carrie J. Cai. 2024. "We Need Structured Output": Towards User-centered Constraints on Large Language Model Output. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI EA '24)*. Association for Computing Machinery, New York, NY, USA, Article 10, 9 pages. doi:10.1145/3613905.3650756
[26] Brahim Mahmoudi, Naouel Moha, Quentin Stievenert, and Florent Avellaneda. 2025. AI-Specific Code Smells: From Specification to Detection. arXiv:2509.20491 [cs.SE] doi:10.48550/arXiv.2509.20491
[27] mariafilippa. 2023. PydanticOutputParser has high chance failing when completion contains new line #3709. https://github.com/hwchase17/langchain/issues/3709 GitHub issue.
[28] Microsoft Learn. 2025. Azure OpenAI — Documentation (Quotas, Structured Outputs, How-To). https://learn.microsoft.com/en-us/azure/ai-foundry/openai/
[29] Microsoft Learn. 2025. Design to Support Foundation Model Life Cycles. https://learn.microsoft.com/en-us/azure/architecture/ai-ml/guide/manage-foundation-models-lifecycle
[30] Nguyen Nhat Minh, Andrew Baker, Clement Neo, Allen G. Roush, Andreas Kirsch, and Ravid Shwartz-Ziv. 2025. Turning Up the Heat: Min-p Sampling for Creative and Coherent LLM Outputs. In *ICLR 2025*.
[31] Modelmetry. 2024. *How To Ensure LLM Output Adheres to a JSON Schema.* https://modelmetry.com/blog/how-to-ensure-llm-output-adheres-to-a-json-schema
[32] João Eduardo Montandon, Luciana Lourdes Silva, Cristiano Politowski, Daniel Prates, Arthur de Brito Bonifácio, and Ghizlane El Boussaidi. 2025. Unboxing Default Argument Breaking Changes in Data Science Libraries. (2025). arXiv:2408.05129 [cs.SE] doi:10.48550/arXiv.2408.05129 JSS.
[33] Masumi Morishige and Ryo Koshihara. 2025. Ensuring Reproducibility in Generative AI Systems for General Use Cases: A Framework for Regression Testing and Open Datasets. doi:10.48550/arXiv.2505.02854
[34] Anna Neumann, Elisabeth Kirsten, Muhammad Bilal Zafar, and Jatinder Singh. 2025. Position is Power: System Prompts as a Mechanism of Bias in Large Language Models (LLMs). In *Proceedings of the 2025 ACM Conference on Fairness, Accountability, and Transparency (FAccT '25)*. ACM, 573–598. doi:10.1145/3715275.3732038
[35] Kaiwen Ning, Jiachi Chen, Jingwen Zhang, Wei Li, Zexu Wang, Yuming Feng, Weizhe Zhang, and Zibin Zheng. 2024. Defining and Detecting the Defects of the Large Language Model-based Autonomous Agents. arXiv:2412.18371 [cs.SE] https://arxiv.org/abs/2412.18371
[36] Ollama. 2025. Modelfile: Valid parameters and values. https://github.com/ollama/ollama/blob/main/docs/modelfile.md#valid-parameters-and-values
[37] OpenAI. 2025. API Reference — Structured model outputs. https://platform.openai.com/docs/guides/structured-outputs
[38] OpenAI. 2025. OpenAI Platform Documentation. https://platform.openai.com/docs
[39] OpenAI. 2025. openai-python. https://github.com/openai/openai-python. Accessed 2025-09-25.
[40] OpenRouter. 2025. *OpenRouter.ai — One API for Any Model.* https://openrouter.ai/
[41] Frank Reyes, Yogya Gamage, Gabriel Skoglund, Benoit Baudry, and Martin Monperrus. 2024. BUMP: A Benchmark of Reproducible Breaking Dependency Updates. In *SANER 2024*. arXiv:2401.09906 doi:10.48550/arXiv.2401.09906
[42] Krishna Ronanki, Beatriz Cabrero-Daniel, and Christian Berger. 2024. Prompt Smells: An Omen for Undesirable Generative AI Outputs. arXiv:2401.12611 [cs.LG] https://arxiv.org/abs/2401.12611
[43] Yuchen Shao, Yuheng Huang, Jiawei Shen, Lei Ma, Ting Su, and Chengcheng Wan. 2025. Are LLMs Correctly Integrated into Software Systems? arXiv:2407.05138 [cs.SE] https://arxiv.org/abs/2407.05138
[44] Haoye Tian, Chong Wang, BoYang Yang, Lyuye Zhang, and Yang Liu. 2025. A Taxonomy of Prompt Defects in LLM Systems. arXiv:2509.14404 [cs.SE] https://arxiv.org/abs/2509.14404
[45] Vellum AI. 2025. LLM Temperature: How It Works and When You Should Use It. https://www.vellum.ai/llm-parameters/temperature

[46] Daniel Venturini, Filipe Roseiro Cogo, Ivanilton Polato, Marco A. Gerosa, and Igor Scaliante Wiese. 2023. I Depended on You and You Broke Me: An Empirical Study of Manifesting Breaking Changes in Client Packages. (2023). arXiv:2301.04563 doi:10.48550/arXiv.2301.04563 TOSEM, 2023.

[47] Greg Wilson, D. A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H. D. Haddock, Kathryn D. Huff, Ian M. Mitchell, Mark D. Plumbley, Ben Waugh, Ethan P. White, and Paul Wilson. 2014. Best Practices for Scientific Computing. *PLOS Biology* 12, 1 (2014), e1001745.

[48] Cailin Winston and René Just. 2025. A Taxonomy of Failures in Tool-Augmented LLMs. In *AST 2025*. 125–135. doi:10.1109/AST66626.2025.00019

[49] Matt Wyman and Sarah Barber. 2024. *How to Validate the Output of LLM-Based Products*. https://okareo.com/blog/posts/validate-llm-output

[50] Zhiqiu Xia, Lang Zhu, Bingzhe Li, Feng Chen, Qiannan Li, Chunhua Liao, Feiyi Wang, and Hang Liu. 2025. Analyzing 16,193 LLM Papers for Fun and Profits. arXiv:2504.08619 [cs.DL] https://arxiv.org/abs/2504.08619

[51] Wenli Yang, Lilian Some, Michael Bain, and Byeong Kang. 2025. A comprehensive survey on integrating large language models with knowledge-based methods. *Knowledge-Based Systems* 318 (2025), 113503. doi:10.1016/j.knosys.2025.113503

[52] Haiyin Zhang, Luís Cruz, and Arie van Deursen. 2022. Code Smells for Machine Learning Applications. arXiv:2203.13746 [cs.SE] https://arxiv.org/abs/2203.13746

[53] Terry Yue Zhuo, Junda He, Jiamou Sun, Zhenchang Xing, David Lo, John Grundy, and Xiaoning Du. 2025. Identifying and Mitigating API Misuse in Large Language Models. arXiv:2503.22821 [cs.SE]