Phoenix: International Multidisciplinary Research Journal
Vol 4, No. 1.1, March, 2026
ISSN: 2583-6897

HALLUCINATIONS IN LARGE LANGUAGE MODELS: A COMPREHENSIVE STUDY

Prof. More Santosh.S
Department of Computer Science, Mamasaheb Mohol College
Bhagwan Misal
Department of Computer Science, Mamasaheb Mohol College

Abstract

Large Language Models (LLMs) have achieved remarkable success across various domains including natural language processing, code generation, content creation, and decision support systems. However, one of the most critical challenges affecting their reliability is hallucination — the generation of factually incorrect, fabricated, or misleading information.

This research study examines the phenomenon of hallucinations in Large Language Models, focusing on their types, causes, detection mechanisms, and mitigation strategies. The study explores intrinsic, extrinsic, and domain-specific hallucinations, particularly in software code generation. Furthermore, it analyzes the role of training data, probabilistic token prediction, reasoning limitations, and context-awareness constraints in contributing to hallucinations.

Keywords: Large Language Models, Hallucination, AI Reliability, Code Generation, Retrieval-Augmented Generation, AI Ethics, Model Detection.

Table of Contents:

1. Introduction - Objectives of the Research

Large Language Models (LLMs) have achieved state-of-the-art performance in diverse tasks, but their reliability is compromised by hallucination the generation of content that is factually incorrect, inconsistent, or fabricated. This phenomenon poses significant risks, particularly in high-stakes domains like software engineering, where generated code can introduce bugs, vulnerabilities, and maintenance issues.

Understanding the nature of these hallucinations is paramount to building more reliable models. Research has focused on creating detailed taxonomies of *how* models hallucinate, analysing the underlying mechanisms, and developing robust detection and mitigation strategies.

Objectives of this Research:

1. To synthesize a comprehensive taxonomy of LLM hallucinations by reviewing general and domain-specific (code generation) literature.
2. To analyse the underlying causes and mechanisms that lead to hallucinations in LLMs.
3. To conduct a comparative review of current detection and mitigation strategies, with a focus on their computational efficiency and practical applicability in complex scenarios.
4. To identify the limitations of current approaches and outline a clear scope for future research.

2 Literature Review of previous research in the area and justification/ Importance/ Value of further research.

The review of the literature reveals a deep and growing concern over LLM hallucinations, categorized both generally and within the critical domain of code generation

A. General Taxonomy of Hallucinations:

Hallucinations are categorized based on their relationship to the model's knowledge and the provided context:

- Intrinsic Hallucinations: The generated content is inconsistent with the model's own internal, parametric knowledge.
- Extrinsic Hallucinations: The model misinterprets or fails to integrate provided external information (e.g., prompt context).
- Amalgamated Hallucinations: The model incorrectly combines multiple, disparate facts into a single, erroneous output.
- Non-Factual Hallucinations: The generated content directly contradicts established, verifiable world knowledge.

B. Domain-Specific Taxonomy: Code Generation:

In software development, hallucinations are classified based on the complexity of the task:

1. Function-Level Hallucinations (Standalone code snippets):
   - Intent Conflicting: Code deviates from the user's core purpose.
   - Knowledge Conflicting: Code misuses an API, invents parameters, or uses undefined identifiers (a common type).
   - Context Inconsistency/Repetition: Internal logical issues or unnecessary code duplication.
   - Dead Code: Unused code segments are generated.

Repository-Level Hallucinations (Code that must integrate with a large project):

- Task Requirement Conflicts (43.53%): Violating the stated functional or non-functional requirements (e.g., using an unsafe function like yaml.load() instead of yaml.safe_load()).
- Factual Knowledge Conflicts (31.91%): Conflicts with established facts about libraries or domain knowledge (e.g., calling a non-existent parameter for a common function).
- Project Context Conflicts (24.56%): Inconsistency with the specific, private context of the user's repository (e.g., importing a package that doesn't exist in the project's environment).

C. Underlying Causes of Hallucinations:

The root causes are multifaceted:

- Next-Token Prediction: LLMs are trained to predict the next word (token) in a sequence based on the preceding words, maximizing the statistical probability of the entire sequence. They are, at their core, pattern-matching engines, not knowledge retrieval systems.
- Training Data Quality: Models learn and reproduce bad practices from low-quality, public code corpora (e.g., widespread insecure code).
- Knowledge Gaps: Outdated Knowledge (model knowledge is static) and Overshadowing (specific instructions are overlooked for dominant patterns).
- Failed Reasoning: The model struggles to capture subtle user intentions or accurately extract relevant details from context.
- Context Awareness Limits: The inability to feed an entire project's codebase into the context window forces the model to guess at available dependencies, leading to Project Context Conflicts.

D. Justification for Further Research:

The persistence of hallucinations, particularly the highly practical Project Context Conflicts, shows a critical gap between LLM capability and real-world reliability. This research is important because it synthesizes new work focusing on these complex repository-level scenarios, justifying the urgent need for computationally efficient and context-aware detection and mitigation techniques to ensure the security, maintainability, and correctness of AI-assisted software.

2. Figures

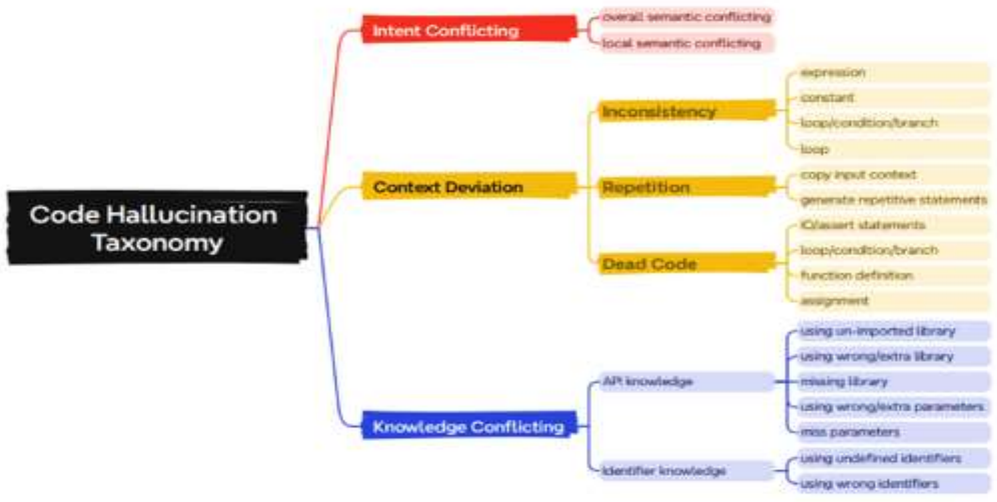


Fig. 1: Taxonomy of Hallucinations in LLM-generated code.

It categorizes different types of hallucinations that can occur in LLM-generated code. The taxonomy branches into four major types:

- **Intent Conflicting** — when the generated code conflicts with the intended logic.
- **Context Deviation** — including issues like inconsistency, repetition, and dead code.
- **Knowledge Conflicting** — mistakes due to wrong API knowledge or identifier usage

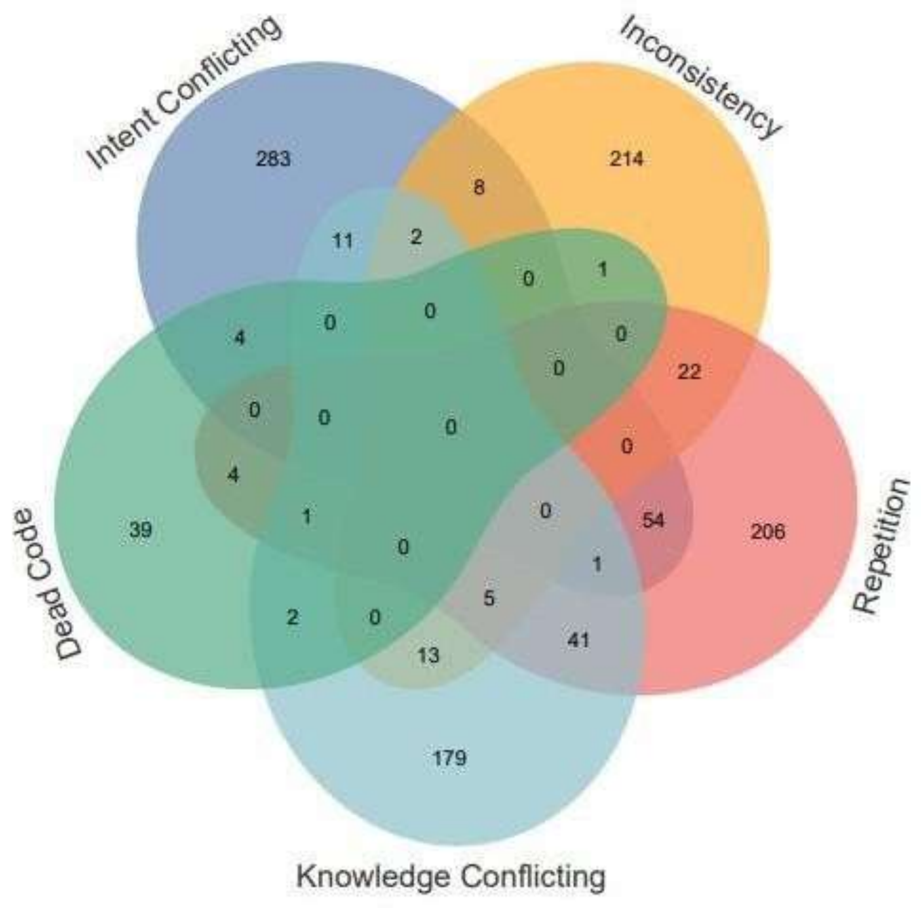


Fig. 2: Distribution of the co-occurrence of various hallucinations within a single program.

Source: Figure from the research article "Exploring and Evaluating Hallucinations in LLM-Powered Code Generation."
It visualizes the categories and characteristics of hallucinations encountered during LLM code-generation experiments.

Task Requirement Conflicts

- **Functional:** Wrong/missing functionality.
- **Non-functional:** Breaks security/performance/style.

Factual Knowledge Conflicts

- **Background:** Wrong concepts.
- **Library:** Wrong library use.
- **API:** Wrong/outdated API calls.
- **Environment:** Wrong system assumptions.
- **Dependency:** Missing/incorrect dependencies.

Project Context Conflicts

- **Resource:** Missing/non-existent files or configs

Source: Figure from the research article "Exploring and Evaluating Hallucinations in LLM-Powered Code Generation."

ISSTA022:6 Z. Zhang, Y. Wang, C. Wang, E. Shi, Y. Ma, W. Zhong, J. Chen, M. Mao, Z. Zheng

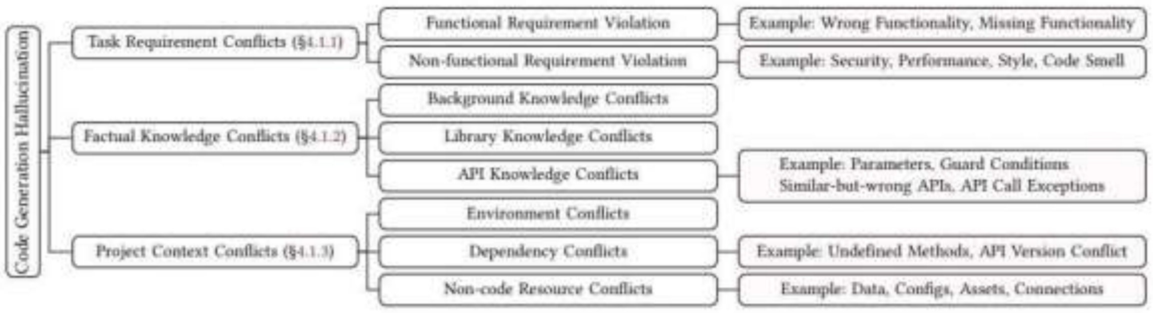

Code Generation Hallucination
Task Requirement Conflicts (§4.1.1)
Factual Knowledge Conflicts (§4.1.2)
Project Context Conflicts (§4.1.3)
Functional Requirement Violation
Non-functional Requirement Violation
Background Knowledge Conflicts
Library Knowledge Conflicts
API Knowledge Conflicts
Environment Conflicts
Dependency Conflicts
Non-code Resource Conflicts
Example: Wrong Functionality, Missing Functionality
Example: Security, Performance, Style, Code Smell
Example: Parameters, Guard Conditions
Similar-but-wrong APIs, API Call Exceptions
Example: Undefined Methods, API Version Conflict
Example: Data, Configs, Assets, Connections

Fig. 1. Taxonomy of Hallucinations in LLM-based Code Generation.

5. Methodology (Data Collection)

This research project is a synthetic review and analysis of existing literature; it does not involve the collection of new, primary experimental data.

Methodology:

1. Systematic Review: Thorough review of all provided documents to identify core themes, synthesized taxonomies, and experimental findings related to LLM hallucinations, specifically focusing on code generation.
2. Taxonomy Synthesis: Consolidation of various general and code-specific hallucination classification systems into a single, comprehensive structure.
3. Method Analysis: Extraction and comparison of proposed detection and mitigation strategies, including "LLM-Check" (Internal State Analysis), "Uncertainty-Based Ensembles", and "Retrieval-Augmented Generation (RAG)".
4. Comparative Reporting: Synthesis of the analysis into this structured report to address the research objectives

6. Actual Work Done

The actual work performed was the detailed comparative analysis and synthesis of detection and mitigation strategies identified in the literature.

A. Hallucination Detection Strategies

Recent research favours efficient, single-response detection methods over slow, expensive consistency checks.

1. Method 1: Internal State Analysis (LLM-Check)

- Concept: Analyses the internal hidden states and attention maps of an auxiliary LLM when it processes a generated response.
- Mechanism: Computes a "Hidden Score" and an "Attention Score" (via eigen-analysis).
- Finding: These scores effectively distinguish truthful from hallucinated content. The method is highly compute-efficient, offering speedups of up to 45x-450x over multi-response baselines, requiring only a single forward pass.

2. Method 2: Uncertainty-Based Ensembles

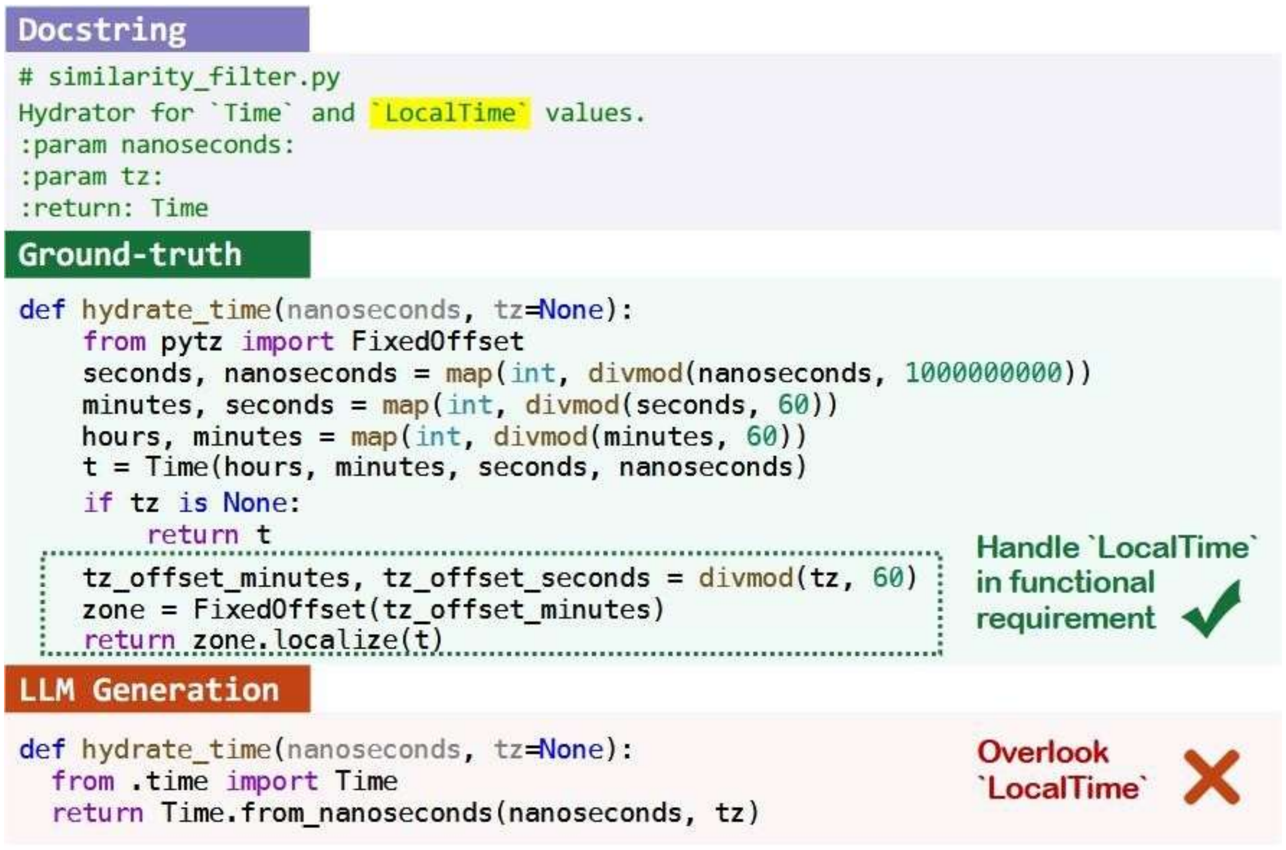
- Concept: Uses the model's own uncertainty (predictive entropy) as a feature for binary classification.
- Mechanism: Employs a fast and memory-efficient ensemble method (adapting BatchEnsemble with LoRA) to measure predictive entropy, which is then fed to a simple classifier.

- Finding: Highly effective for detecting faithfulness hallucinations, achieving 97.8% accuracy in one study, demonstrating that a model's internal states can effectively signal its own confidence. Examples:

Source:

LLM Hallucinations in Practical Code Generation: Phenomena, Mechanism, and Mitigation.

Model Used: GPT-3.5-Tubo

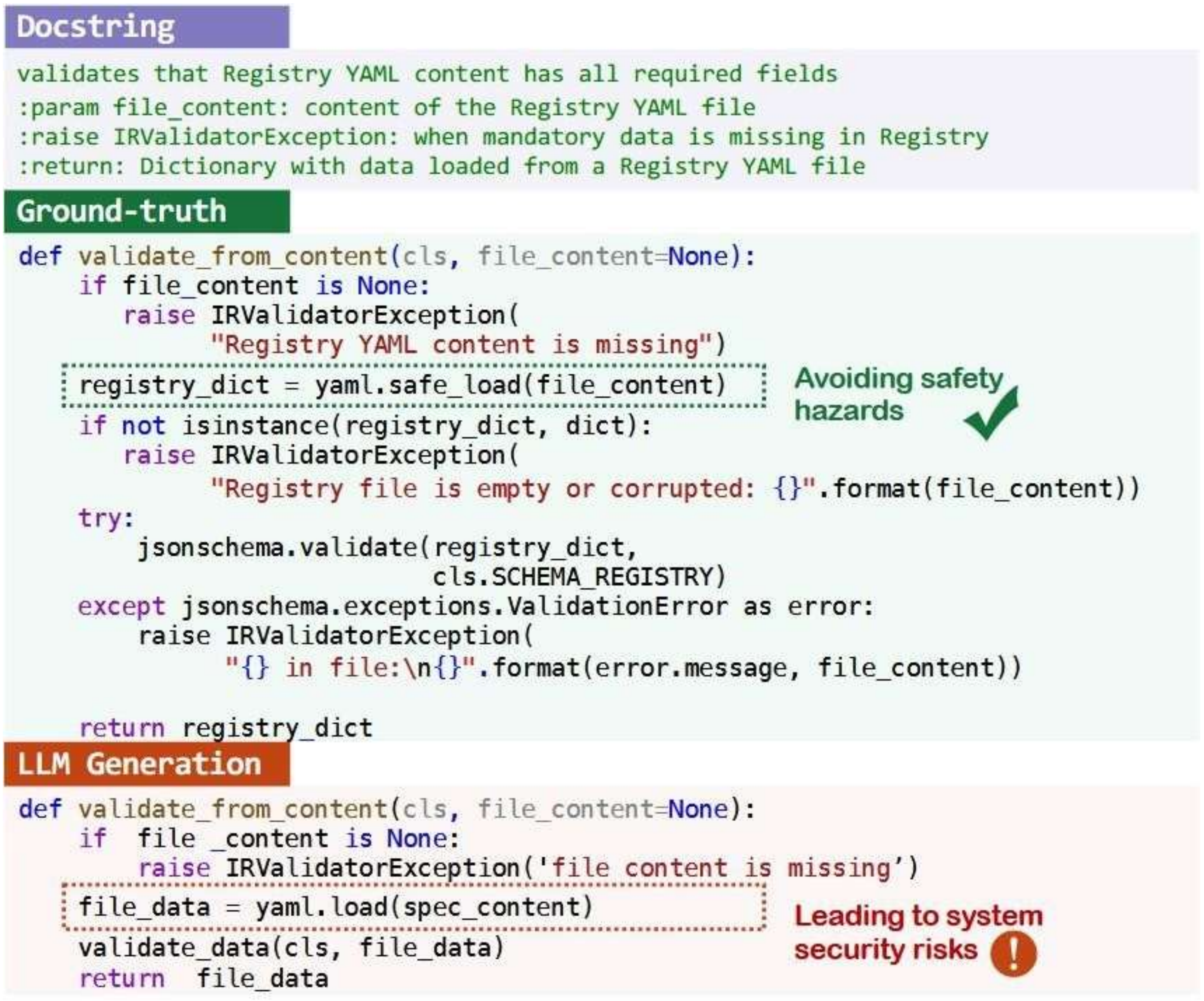


```
# similarity_filter.py
Hydrator for `Time` and `LocalTime` values.
:param nanoseconds:
:param tz:
:return: Time
```

```
def hydrate_time(nanoseconds, tz=None):
    from pytz import FixedOffset
    seconds, nanoseconds = map(int, divmod(nanoseconds, 1000000000))
    minutes, seconds = map(int, divmod(seconds, 60))
    hours, minutes = map(int, divmod(minutes, 60))
    t = Time(hours, minutes, seconds, nanoseconds)
    if tz is None:
        return t
    tz_offset_minutes, tz_offset_seconds = divmod(tz, 60)
    zone = FixedOffset(tz_offset_minutes)
    return zone.localize(t)
```

```
def hydrate_time(nanoseconds, tz=None):
  from .time import Time
  return Time.from_nanoseconds(nanoseconds, tz)
```

Fig. 3. Example: Functional Requirement Violation.


Docstring

```
validates that Registry YAML content has all required fields
:param file_content: content of the Registry YAML file
:raise IRValidatorException: when mandatory data is missing in Registry
:return: Dictionary with data loaded from a Registry YAML file
```

Ground-truth

```
def validate_from_content(cls, file_content=None):
    if file_content is None:
       raise IRValidatorException(
             "Registry YAML content is missing")
    registry_dict = yaml.safe_load(file_content)
    if not isinstance(registry_dict, dict):
       raise IRValidatorException(
             "Registry file is empty or corrupted: {}".format(file_content))
    try:
        jsonschema.validate(registry_dict,
                            cls.SCHEMA_REGISTRY)
    except jsonschema.exceptions.ValidationError as error:
        raise IRValidatorException(
              "{} in file:\n{}".format(error.message, file_content))

    return registry_dict
```

LLM Generation

```
def validate_from_content(cls, file_content=None):
    if  file _content is None:
        raise IRValidatorException('file content is missing')
    file_data = yaml.load(spec_content)
    validate_data(cls, file_data)
    return  file_data
```

Fig. 4. Example: Non-functional Requirement Violation.

Source:
LLM Hallucinations in Practical Code Generation: Phenomena, Mechanism, and Mitigation.

Model Used: GPT-3.5-Tubo

B. Hallucination Mitigation Strategies

1. Self-Correction Challenges

- Finding: Even when prompted to recognize their own errors, LLMs (e.g., ChatGPT) show limited ability to correct them. One study found that ChatGPT was only able to successfully mitigate 16% of the code hallucinations it identified.

2. Retrieval-Augmented Generation (RAG)

- Concept: Used primarily for complex, repository-level generation.

- Mechanism: The system retrieves relevant code snippets from the project's *own repository* and adds them to the prompt as context before generation.
- Finding: This lightweight RAG approach consistently improved the Pass@1 (correctness) scores across all six LLMs studied. It was particularly effective at mitigating Task Requirement Conflicts and Project Context Conflicts by supplying necessary project-specific examples.

3. LLM-as-a-Judge (The Pragmatic Approach)

- This is the most popular, scalable, and versatile method in production and evaluation.
- Mechanism: A high-performing LLM (often a proprietary model like GPT-4, referred to as the "Judge LLM") is provided with the Input Prompt, the Generated Output from the model being tested, and often the Source Context.
- Process: The Judge LLM is given a detailed, Chain-of-Thought (CoT) prompt instructing it to:
    - Break the generated output into individual factual claims.
    - Check each claim against the source context (or general knowledge).
    - Assign a numerical score for Hallucination (or Faithfulness).

7. Results (Key Findings and Synthesis)

The synthesis of the literature yields three principal conclusions regarding LLM hallucinations:

1. **Taxonomies are Maturing:** The focus has shifted from general intrinsic/extrinsic categories to highly practical, domain-specific taxonomies for code generation. The prominence of Project Context Conflicts confirms that the main barrier to deployment is the LLM's struggle with real-world, private code repositories.
2. **Detection is Moving Toward Efficiency:** Research demonstrates that reliance on slow, multi-response methods is unnecessary. Efficient, single-response techniques (analysing internal states and uncertainty) are proving capable of real-time hallucination detection.
3. **Mitigation Requires External Context:** Correction is harder than detection. The success of RAG in the repository setting confirms that the most effective mitigation strategy is to overcome the model's reliance on outdated or generalized parametric knowledge by supplying timely, correct, and highly relevant project-specific context.

8. Future scope of research and limitation

Limitations:

- This project is a synthetic review and does not provide new experimental data. Its findings are contingent on the methodologies and results of the source papers.

- The comparison of advanced detection methods (LLM-Check vs. Uncertainty Ensembles) and mitigation methods (RAG) is based on findings from distinct experimental setups and lacks a single, unified cross-comparison.

Future Scope of Research:

1. Robust, Unified Evaluation: Developing unified benchmarks that test for both function-level and repository-level hallucinations across different LLM architectures and programming languages.
2. Advanced Mitigation Agents: Creating sophisticated, multi-agent systems where one agent generates code, another validates it against project context and security rules, and a third performs advanced, context-aware corrections.
3. Explainable AI for Hallucinations: Further exploring the relationship between a model's internal representations (e.g., attention weights) and the *mechanism* of hallucination to develop models that are "correct-by-construction."
4. Security and Non-functional Hallucinations: Focusing dedicated research on developing automated tools to detect subtle, yet high-impact, hallucinations related to security vulnerabilities and non-functional requirement violations, which are often missed by simple functional tests.

9. Bibliography

- Arteaga, Gabriel Y., Thomas B. Schön, and Nicolas Pielawski. "Hallucination Detection in LLMs: Fast and Memory-Efficient Fine-Tuned Models." *Proceedings of the 6th Northern Lights Deep Learning Conference (NLDL), PMLR 265* (2025).
- Cleti, Meade, and Pete Jano. "Hallucinations in LLMs: Types, Causes, and Approaches for Enhanced Reliability." *Preprint* (2024).
- Liu, Fang, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, Li Zhang, Zhongqi Li, and Yuchi Ma. "Exploring and Evaluating Hallucinations in LLM-Powered Code Generation." *arXiv preprint arXiv:2404.00971v2* (2024).
- Sriramanan, Gaurang, Siddhant Bharti, Shoumik Saha, Priyatham Kattakinda, Vinu Sankar Sadasivan, and Soheil Feizi. "LLM-Check: Investigating Detection of Hallucinations in Large Language Models." *38th Conference on Neural Information Processing Systems (NeurIPS 2024)* (2024).
- Zhang, Ziyao, Chong Wang, Yanlin Wang, Ensheng Shi, Yuchi Ma, Wanjun Zhong, Jiachi Chen, Mingzhi Mao, and Zibin Zheng. "LLM Hallucinations in Practical Code Generation: Phenomena, Mechanism, and Mitigation." *Proceedings of the ACM on Software Engineering, Vol. 2, No. ISSTA, Article ISSTA022* (2025).