ETF: An Entity Tracing Framework for Hallucination Detection in Code Summaries

Anonymous ACL submission

Abstract

Recent advancements in large language models (LLMs) have significantly enhanced their ability to understand both natural language and code, driving their use in tasks like natural language-to-code (NL2Code) and code summarization. However, LLMs are prone to hallucination—outputs that stray from intended meanings. Detecting hallucinations in code summarization is especially difficult due to the complex interplay between programming and natural languages. We introduce a first-of-its-kind dataset with ∼10K samples, curated specifically for hallucination detection in code summarization. We further propose a novel Entity Tracing Framework (ETF) that a) utilizes static program analysis to identify code entities from the program and b) uses LLMs to map and verify these entities and their intents within generated code summaries. Our experimental analysis demonstrates the framework's effectiveness, leading to a 73% F1 score. This approach provides an interpretable method for detecting hallucinations by grounding entities, allowing us to evaluate summary accuracy.

```java
public RowBuilder int16(String name){
 ColumnInt16 column =
 new ColumnInt16(_columns.size(), name
    , _offset);
 _offset += column.length();
 _columns.add(column);
 return this;}
```

Summary: .... This method is used to add a new column of data type int16 (16-bit integer) to the existing data structure. It creates a new ColumnInt16 object with the given name and size (16 bits), updates the offset value to accommodate the new column...........

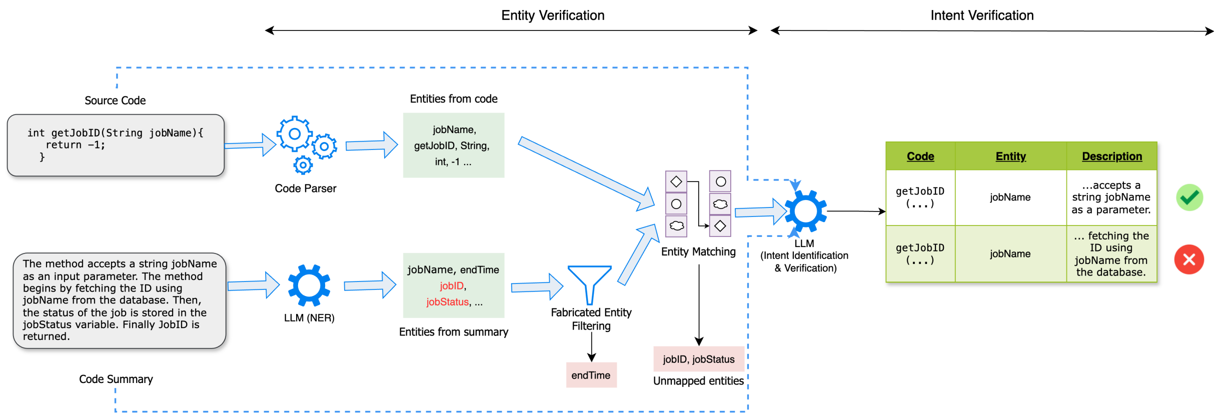
Example 1 (LLama3-70B): Confused Data Type

1 Introduction

Hallucination in natural language processing is defined as a condition in which a language model produces a text that is either incoherent or does not faithfully represent the provided source input (Ji et al., 2023). Similarly, in the context of code summarization, *hallucination can be defined as a condition in which the generated summary does not accurately capture the intent and implementation details of the given input code.*

Hallucination can originate from a combination of factors, where one common reason could be the misinterpretation of code entities. This misunderstanding can impact the model's ability to interpret the intended functionality of the code, resulting in an inaccurate portrayal of its purpose. For instance, consider the Example 1, where the intention of the *int16* java method is to create a new 16-bit integer column (ColumnInt16) with a specified name, update the position for the next column, add it to the list of columns, and then return the RowBuilder object. However, the generated explanation introduces a non-existent int16 datatype and proceeds to discuss the rest of the logic as if it were valid. This could mislead a novice Java developer into believing that an int16 datatype exists in Java. Furthermore, several large language models (LLMs) like LLaMA and Granite failed to detect this hallucination. One reason could be that int16 is a valid datatype in other programming languages such as C, C++, C#, and Go, causing both humans and LLMs to confuse it with learning from those languages. Similarly, in the example shown in Figure 1, the java method *getJobID()* takes "jobName" as an argument and simply returns -1. The summary generated by the model provides a detailed explanation, including how the method getJobID() connects to the database and attempts to retrieve the jobID using the given jobName. Additionally, the summary mentions it stores the "jobStatus" in a variable. Clearly, the generated summary has no supporting entities in the code for db access and the model is relying on the method name to hallucinate a plausible summary of the method.

In this work, we study different factors that can



```
int getJobID(String jobName){
   return -1;
  }
```

The method accepts a string jobName as an input parameter. The method begins by fetching the ID using jobName from the database. Then, the status of the job is stored in the jobStatus variable. Finally JobID is returned.

Code	Entity	Description	
getJobID (...)	jobName	...accepts a string jobName as a parameter.	✓
getJobID (...)	jobName	... fetching the ID using jobName from the database.	✗

Figure 1: Proposed Methodology: This diagram illustrates our end-to-end Entity Tracing Framework (ETF), which takes source code and a corresponding summary as input and returns if the summary is hallucinated or not. First, we use code parsers to extract entities from the source code and employ large language models (LLMs) to identify entities from the summary. Next, we apply string-based heuristics to match entities from the summary to the code. Following this, an LLM verifies the accuracy of each entity's description by cross-referencing the source code with relevant sentences in the summary. This process enables the localization of hallucinated content in the summary, ultimately enhancing its interpretability.

lead to hallucination and list down a taxonomy to map the common causes easily. Noting a lack of datasets to reliably research this topic. Therefore, we create a first-of-its-kind dataset for studying hallucination in code summarization with **411 summaries** generated by seven different large language models, broken into **9933 entity-level samples**. This dataset consists of code and a corresponding summary describing the code. The annotation consists of a) NER, b) Entity Description Verification, and (c) Overall Summary Quality (not focusing on completeness or conciseness). We then introduce a framework that evaluates the correctness of the generated summary. For this, we verify if the entities discussed in the summary are present in the code and correctly described in the summary. The framework leverages code parsers like javalang[1] to list the different entities in the code snippet and prompt-based approaches to detect entities in summary. We note that detecting entities in the generated summary is more difficult due to the high degree of polysemy (Tabassum et al., 2020). For example, entities like "list", "while", "if", etc, can be a code entity or natural language entities. This necessitates our reliance on large language models with high reasoning capabilities for detecting entities on the summary side. We then map the detected entities from the summary to code by using string-matching heuristics. The sentences with unmapped entities can be considered as ungrounded (source of extrinsic hallucination). For each mapped entity, we then have a tuple <*code, entity, intent-related sentence*>, where the intent-related sentence can be considered as the sentence in summary mentioning the entity. The final step is to verify each tuple from the summary for intrinsic hallucination to assess the correctness of the code summary. Our experiments demonstrate the importance of localizing entities in the summary for effective hallucination detection. Our contributions are :

- A taxonomy covering diverse reasons that might lead to hallucination in the code summarization (Figure 2).
- A novel dataset[2] for studying hallucination detection in code summarization, featuring 411 summaries from 7 LLMs and 10K entity-level samples (Table 1) with an explanation of causes for hallucination as per taxonomy.
- A first-of-its-kind approach for entity-level hallucination detection in code summarization inspired by the insights from human behaviour during code reviews leading to a performance of 73% F1 score (Table 2).

2 Related Work

Recent advances in the NLP community have witnessed significant improvements in hallucination detection pipelines. In this section, we discuss some of the works that are relevant to ours.

[1]https://github.com/c2nes/javalang

[2]We plan to open source the Data and Code

Hallucination in Natural Langauge: Rawte et al. (2024); Sahoo et al. (2024) review recent advances in hallucination detection in natural language, emphasizing its practical significance. Recently, prompt-based methods (Arora et al. (2022), Manakul et al. (2023), Agrawal et al. (2023), Dhuliawala et al. (2023)) are being used to detect hallucinations in the text produced by LLMs. Xiao and Carenini (2022) and Zhang et al. (2022) attempt to address entity-level verification in natural language inputs. Both of these works involve improving the correctness of natural language summaries and do not discuss anything in the context of code. We note that most of the hallucination detection frameworks (Manakul et al., 2023; Arora et al., 2022; Dhuliawala et al., 2023; Valentin et al., 2024; Rebedea et al., 2023) in natural language do not enforce reference text for grounding. In our setup of code summarisation, the generated summary has to be evaluated with respect to a reference text (the code snippet). Therefore, neccessiating an approach which could compare the code summary to the code snippet. Maynez et al., 2020; Ji et al., 2023 discuss further fine-graining of hallucination in natural language as intrinsic and extrinsic hallucination. More specifically, **Intrinsic hallucination** occurs when the given text contradicts the reference, while **Extrinsic hallucination** happens when the text cannot be verified against the reference. We use a similar convention in our paper.

Hallucination in Code Generation: The code generation space has captured significant attention due to its practical significance in software development. (Jiang et al., 2024b) discusses recent developments in code generation and suggests the importance of addressing hallucination for improving the reliability of LLMs. Liu et al. (2024) studies hallucination in code generation and proposes a categorization that encompasses five categories of hallucinations based on the conflicting objectives and varying degrees of deviation observed in code generation. Tian et al., 2024; Agarwal et al., 2024; Spracklen et al., 2024 advanced the field with datasets and frameworks addressing hallucination in code generation. These studies highlight that while LLM-generated code may be syntactically correct and semantically plausible, it often fails to execute as intended or meet requirements.

Despite progress in hallucination detection, code summarization remains underexplored. Kang et al., 2024 and Zhang, 2024 focused on inconsistencies in comment generation, addressing specific aspects like design constraints and parameter types, but their methods face challenges due to reliance on execution environments. In contrast, our approach validates the full functionality of generated outputs, independent of external dependencies, offering a more reliable solution by grounding entities in the input code and verifying their intent.

3 Datasets

To create the hallucination dataset for code summarization, we consider code snippets from Java programming language and CodeXGLUE (Lu et al., 2021) – Code-To-Text dataset. We focused on Java programming language due to its widespread relevance in the industry. It offers a rich set of entities (such as classes, methods, and variables) due to its structured design and strict typing system. The dataset was annotated by 8 annotators who are experts in Java and held at least a Master's degree in Computer Science, with some having a PhD in the field. On average, the annotators had 4+ years of experience in Java programming. We report the statistics in Table 1 and describe the data curation process below:

Summary Generation: We generate summaries from CodeXGLUE by prompting seven different LLMs (Appendix A) with 600 code snippets. By producing multiple summary variants, we can assess hallucination generation by different LLMs and evaluate hallucination detection techniques under varied conditions. We present quantitative results (Table 3) and qualitative analysis in Section 6. During initial annotation, we found that annotators spent considerable time verifying summaries, often requiring online documentation searches, leading to an average annotation time of 30 minutes or more per summary. To ensure feasibility, we randomly prune samples and use $\sim 10\%$ of the data for the final hallucination annotation task (Table 1).

Named Entity Recognition Since our framework involves tracing entities from summary to code, we perform NER of the summaries based on the tagset suggested in Tabassum et al. (2020) (prompt in Appendix 5).

Hallucination Labeling: For each detected code entity in a summary, all sentences describing that entity are considered relevant. To account for the scenario where the relevant sentence can be noisy, we introduced a third label, "IRRELEVANT", which can be used to evaluate the performance of the intent-detection module and removed

Category	**Count**	**Percentage (%)**
Summary Level Classification		
Hallucinated	130	31.63%
Not Hallucinated	281	68.36%
Total Summaries	**411**	**100%**
Entity Level Classification		
CORRECT	9024	90.84%
INCORRECT	303	3.05%
IRRELEVANT	606	6.11%
Total Entities	**9933**	**100%**

Table 1: Overall Data Statistics

during the preprocessing. Thus, we obtain tuples of (code, entity, relevant sentences) for each entity. A total of 9933 such tuples, sampled from 441 summaries, were selected for human annotation (hired based on volunteering) to detect hallucinations. Out of these, 4354 tuples (from 222 summaries) were independently reviewed by two different sets of annotators, leading to a Cohen Kappa score of 0.72, implying high agreement. The conflicts were resolved by two independent meta-annotators. The annotators were asked to evaluate the overall summary by assigning a label of ‘GOOD’, ‘FAIR’, and ‘POOR" *We observe that, on average, 1.33 entities were marked as hallucinated for the summaries rated as ’FAIR’ or ’POOR’*. Therefore, we consider a summary as hallucinated if at least one of the entities is hallucinated. After pre-processing, we consider the instance with labels "CORRECT" and "INCORRECT" in human data and treat the "IRRELEVANT" label predicted by the model as "INCORRECT". We provide the complete annotation guideline in Appendix E.2.

4 Categorization of Factors for Hallucination in Code Summarization

In this section, we describe the various factors that could lead to hallucination in code summaries (Figure 2) based on what we learned from the annotation process. This classification, based on the underlying factors of hallucination, offers insights into the generative behaviors of language and code models. We describe these categories of hallucination factors below and discuss their statistical analysis in Figure 5.2.

HC1: Based on Identifier Name Bias: Name Bias refers to the tendency of language models to rely on identifier names when interpreting code. We classify this bias into three subcategories based on its source: 1)variables, 2)functions, 3)libraries. The model can misinterpret code due to the linguistic characteristics of these entity names. As the semantics of the code is defined by the underlying logic rather than their lexical meaning of entities, this may lead to hallucination. In the example shown in Figure 1, the model (Granite-20B) incorrectly assumes that `getJobID` is about retrieving a job ID, based purely on their names, even though the actual code logic suggests otherwise.

HC2: Insufficient knowledge: This involves scenarios where the model generates incorrect summaries due to lack of knowledge. This may include an incorrect explanation of the imported libraries that the model did not see in its training data, incorrect information about the keyword, etc. We further divide this category into two parts:

1) **Contextual code:**This occurs when the model fails to correctly explain the code, often because it has not encountered the functionality of code during training or is working with a low-resource language like COBOL, where fundamental rules may be misrepresented in the summary.

2) **Non-contextual code** involves the scenario when the input does not contain the complete code and mentions an unseen library or an unknown construct whose functionalities are not understood by the model. For example, in the code sample shown in the HC2 Example, the model incorrectly describes the purpose of SQLException.

```
public String getString (int
    columnPosition) throws
SQLException {
    return (String) resultSet.
        getObject(columnPosition);}
```

Summary: ...The method first checks if the ResultSet object is null. If it is, a SQLException is thrown. . . .

HC2: Granite-20B insufficient knowledge

HC3: Code Complexity: This pertains to the model’s tendency to produce incorrect code summaries due to high code complexity. This may stem from the model’s insufficient reasoning capabilities to understand the code or the misinterpretation of user instructions. Key factors contributing to complexity include: 1) **Length**: Longer code is more complex and involves more interdependencies with more potential points of failure. 2) **Lexical Complexity**: Complex vocabulary, including diverse operands and operations, increases the number of elements to track and understand. 3) **Logical Complexity**: Code with high cyclomatic complexity,

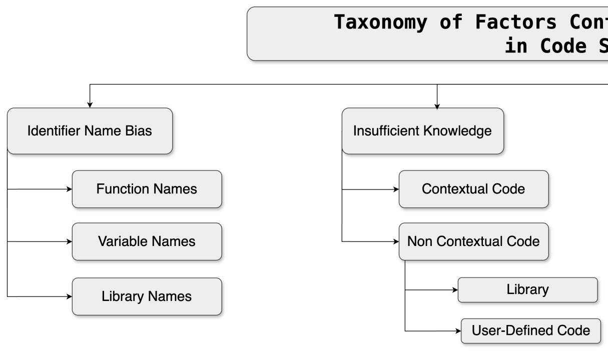


Figure 2: Taxonomy of hallucination in code summarization based on the causes of hallucination. We start with four broad categories and then present a fine-grained classification of each category.

multiple paths, or distant method invocations increases the challenge of comprehension.

```
public <V>V execute(RedisCallback<V>
    cb) { Jedis jedis = jedisPool.
    getResource();
boolean success = true;
try {return cb.execute(jedis);}
catch (JedisException e) {
     success = false;
     if (jedis != null){
jedisPool.returnBrokenResource(jedis)
    ;} throw e;
} finally { if (success) { jedisPool.
    returnResource(jedis);}}}
```

Summary: ...it then returns the ‘jedis" object to the pool using the ‘returnBrokenResource" method if the ‘success" variable is ‘false"...

HC3: Codellama-7B confusion with conditions

In the code snippet shown in HC3 Example, the model Codellama-7b (Roziere et al., 2023) produces an incorrect interpretation of the condition. This may be due to increased complexity due to nesting leading to more complicated logic that are challenging to understand by model.

HC4: Natural language Context: This refers to cases where natural language in code snippets, such as outdated **comments** or **log statements**, causes hallucinations in code summaries. In the code snippet shown in HC4 Example, the LLama3-70B model incorrectly infers that the property variable contains a list of key-value pairs inferred from a commented line. However, the ‘property" variable contains an alphanumeric string followed by one or more semi-colons.

5 Methodology

A code summary typically has a global and local view similar to texts (Maharaj et al., 2023). While the global view includes purpose, functionality, control flow, data flow, etc., the local view includes the details of key entities (variables, functions, etc.) from the source code and their purpose (hereby referred to as the intent of the entity). Our approach is based on the intuition that software developers, while verifying the documentation for a given code repository, first understand the local aspects of the code and then build a bottom-up concept for understanding the global aspects of the code. This involves reading the code line by line and tracing the specific code entities from the documentation to the original code.

This behaviour aligns with working memory theory in cognitive science (Baddeley and Hitch, 1994); working memory is a brain system that temporarily stores and manipulates the information necessary for complex cognitive tasks like learning and reasoning. The capacity of working memory is bounded by 7 ± 2 object at any point in time, which further reduces to 2-3 objects if the objects have relational dependencies with each other. Since code summaries often involve interdependent objects, developers must focus on local aspects to build a global understanding, suggesting a bottom-up heuristic for code summary comprehension.

We leverage these behavioural insights to de-

```
public static HashSet<String>
    createSetFromProperty(String
    property){...
if (property != null && !property.
    equals("null")) { // "([\\w]*)
    =([\\w]*);"
     Pattern params =     Pattern.
        compile("([\\w]+)[;]*"); ...}
```

Summary: ... The input string is expected to contain a list of properties in a specific format, where each property consists of a name-value pair (e.g., "name=value;")...

HC4: LLama3-70B mislead by the comment

sign an LLM-powered framework for detecting hallucinations in code summaries, which involves tracing the entities from the summary to the code. This aspect of mapping the entities from the summary to the code aims to simulate the bottom-up behavioural model of verifying the description of coding entities at a time. With these insights, we aim to measure the correctness of a code summary as a two-step process: (1) Entity Verification and (2) Entity-Intent Verification. The detailed flow of this framework can be found in Figure 1.

5.1 Entity Verification

In entity verification, we check if the entities in the summary are present in the source code to detect extrinsic hallucination. This involves extracting entities from both the code and summary, then mapping entities from summary to the code. We elaborate on this process below:

Entity Extraction from code: We leverage program analysis to extract entities from code (*Javalang* Python package [3]). The code is tokenized (lexer) and parsed into an abstract syntax tree (AST). This tree structure represents the hierarchical organization of code elements, making it easier to analyze. This yields a fine-grained classification of all the tokens present in the code such as variable names, class names, function names, etc.
Entity Extraction from summary: Tabassum et al. (2020) propose the task of entity detection in code summaries and introduce a relevant NER tagset. We adopt this tagset for extracting entities from code summaries (Prompt: Appendix A Figure 5).

Leveraging LLMs to recognize entities introduces the risk of hallucinations, where the model may fabricate entities not present in the code summary. To address this, we implement a filtration step to remove such fabricated entities. We evaluate Gemini and GPT-4-Omni for Code NER using human-collected data, with results in Appendix C. Additionally, we assess an open-source model as part of our contribution. Our findings show a strong correlation between GPT-4-Omni predictions and human data, confirming its effectiveness for entity detection in our framework.

Entity Matching: Once the entities from the code and summary are extracted, we compare them to identify the subset of entities present in the summary but not in the code. These entities are termed ungrounded, and all the sentences in the summary containing these entities can be labelled as extrinsic hallucination. The subset of entities in both the summary and the code goes through an additional verification round for intrinsic hallucination. This is to validate if the intent of the entity in the summary is correctly described as per the code.

[3]https://github.com/c2nes/javalang

5.2 Entity-Intent Verification

The presence of an entity in both the summary and code indicates that the entity is valid but does not warrant the correctness of the context in which it is discussed. For example, in Figure 1 *jobId* is a correct entity, but the context of *retrieving jobID from the database* is incorrect. To address this problem, we propose verifying whether the intent of each mapped entity is accurately described in the summary. We extract all sentences containing the entity of interest from the summary to form its intent context. To identify these relevant sentences that describe an entity's intent, we explored two approaches: (1) prompt-based and (2) string-matching heuristics. Our qualitative assessment, detailed in Appendix D, demonstrates that rule-based heuristics were both more effective and efficient than prompt-based methods, which were prone to hallucinations. Therefore, we relied on string-matching-heuristics for our framework. After identifying the entity and intent, we use LLMs with zero-shot prompting to verify their correctness with the code (Prompt: Appendix 5). We also experimented with few-shot prompts by including examples of various hallucination types in code summaries along with the representative code. However, performance degraded due to the increased prompt length, consistent with findings in recent works like Mirzadeh et al. (2024).

To identify the quality of the whole summary with respect to the code, we aggregate the individual entity-intent hallucination and set the threshold for labelling as 1, as discussed in (Section 3) following human annotation where a summary was rated as 'FAIR' or 'POOR' when an average of 1.33 entities were wrongly described.

6 Experiments and Results

For summary generation, we consider instruction-tuned versions of the SOTA models from the IBM-Granite family (20B and 34B) (Mishra et al., 2024), Llama3 family (8B and 70B) (Touvron et al., 2023), CodeLlama family (7B and 34B) (Roziere et al., 2023) and Mistral-7B (Jiang et al., 2023).

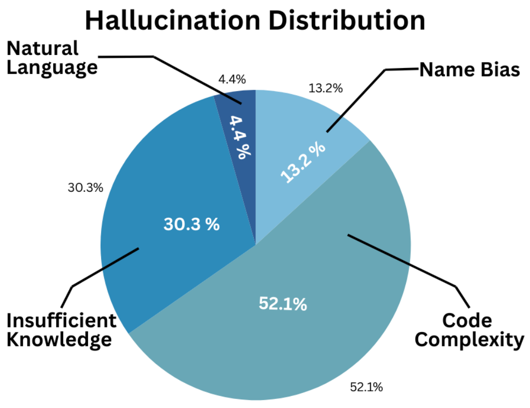


Figure 3: Distribution of different hallucination categories proposed in the taxonomy. We observe that the models tend to hallucinate most frequently due to the high complexity of the code, while significant instances of insufficient knowledge were also identified.

For intent verification, we consider Llama3.1-70B, Llama3-8B (Touvron et al., 2023), Mixtral-8x22B (Jiang et al., 2024a), Mistral7B-v3 (Jiang et al., 2023), GPT4-Omni (Achiam et al., 2023) and Gemini2-Flash (Team et al., 2023). All experimental details can be found in Appendix (B).

Model	P	R	F1
Instance Level			
Gemini-2.0-Direct	0.51	0.50	0.42
Gemini-2.0-ETF*	0.64	0.65	0.64
GPT4-Omni-Direct	0.48	0.50	0.28
GPT4-Omni-ETF*	**0.72**	**0.74**	**0.73**
Mixtral-8x22B-Direct	0.48	0.48	0.45
Mixtral-8x22B-ETF*	0.62	0.61	0.61
Llama-3.1-70B-Direct	0.57	0.54	0.38
Llama-3.1-70B-ETF*	0.62	0.62	0.54
Llama3-8B-Direct	0.60	0.59	0.48
Llama3-8B-ETF*	0.51	0.55	0.50
Mistral-7Bv3-Direct	0.16	0.50	0.24
Mistral-7Bv3-ETF*	0.51	0.50	0.41
Entity Level			
Gemini-2.0	0.58	0.62	0.60
GPT4-Omni	**0.59**	**0.69**	**0.61**
Mixtral-8x22B	0.48	0.38	0.39
Llama-3.1-70B	0.55	0.62	0.56
Llama3-8B	0.59	0.51	0.26
Mistral-7Bv3	0.52	0.59	0.49

Table 2: We report macro Precision (P), Recall (R) and F1 for two evaluation aspects: 1) Instance Level- label the entire summary, and 2) Entity level-label individual entities in summaries.

Entity-Intent Verification: In this aspect of evaluation, we aim to verify the intent of an individual entity. We report the results of entity-intent verification in the Table 2. In general, we observe consistent improvements across all the models compared to direct approach. It can be observed that the GPT4-Omni F1-Score is 0.61 while the Gemini F1 Score is 0.64. Upon analysis, we found that these models often classify INCORRECT tuples as CORRECT when the code references a function or library that is not defined in the input. In such cases, the model infers the functionality based on the library name (Identifier Name Bias 4), which can be difficult to verify.

Instance Level Hallucination Verification: In this aspect of evaluation, we aim to verify the overall summary instance. To compare our approach, we consider a direct setup which involves providing a <code, summary> tuple to identify if the summary is hallucinated or not. We provide these results in Table 2, and it can be observed that our approach provides significant improvement in F1-Score when compared to the Direct approach. In general, the direct evaluation method suffers from hallucinations, such as when identified entities for hallucination are absent from the summary or when natural language entities are mistakenly considered code entities, overall resulting in poor performance. This conveys that our finer-grained evaluation approach provides a more reliable method to identify hallucinated summaries. It also helps with interpretability as it identifies the hallucinated sections of the summary.

7 Analysis

In this section, we discuss various quantitative and qualitative insights of our framework. We first discuss summaries generated by individual models and then elaborate on the general predictive behaviour of our framework and cases of errors.

7.1 Quantitative Analysis

As shown in Table 3, Granite-20B produced shorter summaries, while Llama3-70B generated longer ones. Other models had similar average lengths, reflecting varying elaboration due to differences in training methodologies. For entity mapping, we observe that Llama3-70B has the most mapped entities, indicating the tendency of the model to stay grounded. Granite-20B has the most unmapped entities, which indicates its tendency to produce

Models	Summary Length	CE count	Fabricated (↓)	Mapped(↑)	Unmapped (↓)
Codellama-7B	236.10	8.638	*35.68%*	80.17%	17.65%
Mistral-7B	227.91	6.961	8.59%	79.92%	17.22%
Llama3-8B	257.21	9.45	6.44%	84.50%	12.77%
Granite-20B	*148.95*	7.10	4.25%	*79.54%*	*19.64%*
Granite-34B	214.50	*6.55*	2.45%	85.69%	12.41%
Codellama-34B	278.67	8.22	4.87%	79.78%	19.05%
Llama3-70B	313.58	10.01	**0.15%**	**88.44%**	**8.69%**

Table 3: Quantitative Analysis: This table presents summary statistics for all seven models, where (↑) means a higher score is better and (↓) means lower is better. "CE count" is the number of code entities in the summary. "Fabricated" is the percentage of fabricated entities during named entity recognition, normalized by the total entities. "Mapped" is the percentage of entities correctly mapped to the code, and "Unmapped" refers to those unmapped, both normalized by the total entity count.

content which may not be directly related to the code, leading to extrinsic hallucination.

7.2 Predictive Analysis

```
private List<Transaction>
    retrieveTransactions(String
    rowStatusCd) throws HubException
    {................}
```

Summary: This method retrieves transactions using the "Hub" API based on the input parameters by the "Hub database". It interacts with several key entities, including "squid:s1166" and "squid:s1172", to gather the data......

Analysis: Hallucination and Unmapped Entities

Our framework captures ungrounded entities in summaries. For example, when the model refers to a non-existent Hub API based on the keyword 'HubException", the unmapped entities such as "Hub" are identified. This enables fine-grained and interpretable detection of hallucinations.

7.3 Error Analysis

This section discusses the two major error cases in our framework:

Error Case 1-Creative Summary:

```
public static int writeShort(
    ArrayView target, int offset,
    short value) {...}
```

Summary: Here's a more detailed implementation of the above method: ....{JAVA CODE}.....

Creative Summary by Mistral 7B

During the generation, the model may mention certain aspects of the code in a creative way which may not be incorrect. In the given example, the code summary discusses a more elaborate version of the input code by restating an elongated version. Here, the entities present in the summary are predicted to be ungrounded by our framework.

Error Case 2- Changed Entity Form: Language models may not use the exact names of code entities in summaries. For example, the entity "PreparedStatement" may be referred to as "prepared statement," which could be missed during named entity recognition due to the change in form. The verification of these kinds of summaries may not reflect the inaccuracies due to such sentences.

```
protected PreparedStatement
    setObjects(Connection connection,
    String query, Object... objects)
    throws SQLException {....}
```

Summary: ......set the parameters of the **prepared statement** using the provided objects. This method is likely responsible for iterating over the objects and setting them as parameters in the **prepared statement**. Finally, the method returns the **prepared statement** object.....

Changed Entity Form by Llama3-70B

8 Conclusion and Future Work

Our work addresses the critical challenge of detecting hallucinations in code summarization, a task that demands a deep understanding of both programming and natural languages. By introducing a novel dataset and the Entity Tracing Framework (ETF) with 73% F1 score, we establish a systematic approach to grounding code entities within summaries, enabling a more interpretable and accurate evaluation of explanations. In the future, this framework can be enhanced by incorporating a multi-agent system and leveraging multiple LLMs in tandem to improve prediction accuracy. The current framework can be further developed to better mitigate the occurrence of hallucinations.

9 Limitations

While the framework is designed to be generic, certain components—such as code parsers for entity detection—may be unavailable for low-resource programming languages like COBOL or Perl. Moreover, the performance of large language models (LLMs) heavily depends on their parameter size and the volume of training data. Given the complexity of this task, smaller open-source models may struggle to perform effectively, reinforcing the need for larger LLMs. However, these larger models may not be scalable and often demand significantly greater computational resources.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Vibhor Agarwal, Yulong Pei, Salwa Alamir, and Xiaomo Liu. 2024. Codemirage: Hallucinations in code generated by large language models. *arXiv preprint arXiv:2408.08333*.

Ayush Agrawal, Lester Mackey, and Adam Tauman Kalai. 2023. Do language models know when they're hallucinating references? *arXiv preprint arXiv:2305.18248*.

Simran Arora, Avanika Narayan, Mayee F Chen, Laurel Orr, Neel Guha, Kush Bhatia, Ines Chami, Frederic Sala, and Christopher Ré. 2022. Ask me anything: A simple strategy for prompting language models. *arXiv preprint arXiv:2210.02441*.

Alan D Baddeley and Graham J Hitch. 1994. Developments in the concept of working memory. *Neuropsychology*, 8(4):485.

Shehzaad Dhuliawala, Mojtaba Komeili, Jing Xu, Roberta Raileanu, Xian Li, Asli Celikyilmaz, and Jason Weston. 2023. Chain-of-verification reduces hallucination in large language models. *arXiv preprint arXiv:2309.11495*.

Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12):1–38.

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7b. *arXiv preprint arXiv:2310.06825*.

Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. 2024a. Mixtral of experts. *arXiv preprint arXiv:2401.04088*.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024b. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*.

Sungmin Kang, Louis Milliken, and Shin Yoo. 2024. Identifying inaccurate descriptions in llm-generated code comments via test execution. *arXiv preprint arXiv:2406.14836*.

Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, and Li Zhang. 2024. Exploring and evaluating hallucinations in llm-powered code generation. *arXiv preprint arXiv:2404.00971*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

Kishan Maharaj, Ashita Saxena, Raja Kumar, Abhijit Mishra, and Pushpak Bhattacharyya. 2023. Eyes show the way: Modelling gaze behaviour for hallucination detection. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 11424–11438.

Potsawee Manakul, Adian Liusie, and Mark JF Gales. 2023. Selfcheckgpt: Zero-resource black-box hallucination detection for generative large language models. *arXiv preprint arXiv:2303.08896*.

Joshua Maynez, Shashi Narayan, Bernd Bohnet, and Ryan McDonald. 2020. On faithfulness and factuality in abstractive summarization. *arXiv preprint arXiv:2005.00661*.

Iman Mirzadeh, Keivan Alizadeh, Hooman Shahrokhi, Oncel Tuzel, Samy Bengio, and Mehrdad Farajtabar. 2024. Gsm-symbolic: Understanding the limitations of mathematical reasoning in large language models. *arXiv preprint arXiv:2410.05229*.

Mayank Mishra, Matt Stallone, Gaoyuan Zhang, Yikang Shen, Aditya Prasad, Adriana Meza Soria, Michele Merler, Parameswaran Selvam, Saptha Surendran, Shivdeep Singh, et al. 2024. Granite code models: A family of open foundation models for code intelligence. *arXiv preprint arXiv:2405.04324*.

Vipula Rawte, Aman Chadha, Amit Sheth, and Amitava Das. 2024. Tutorial proposal: Hallucination in large language models. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024): Tutorial Summaries*, pages 68–72.

Traian Rebedea, Razvan Dinu, Makesh Sreedhar, Christopher Parisien, and Jonathan Cohen. 2023. Nemo guardrails: A toolkit for controllable and safe llm applications with programmable rails. *arXiv preprint arXiv:2310.10501*.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Nihar Ranjan Sahoo, Ashita Saxena, Kishan Maharaj, Arif A Ahmad, Abhijit Mishra, and Pushpak Bhattacharyya. 2024. Addressing bias and hallucination in large language models. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024): Tutorial Summaries*, pages 73–79.

Joseph Spracklen, Raveen Wijewickrama, AHM Sakib, Anindya Maiti, and Murtuza Jadliwala. 2024. We have a package for you! a comprehensive analysis of package hallucinations by code generating llms. *arXiv preprint arXiv:2406.10279*.

Jeniya Tabassum, Mounica Maddela, Wei Xu, and Alan Ritter. 2020. Code and named entity recognition in stackoverflow. *arXiv preprint arXiv:2005.01634*.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*.

Yuchen Tian, Weixiang Yan, Qian Yang, Qian Chen, Wen Wang, Ziyang Luo, and Lei Ma. 2024. Codehalu: Code hallucinations in llms driven by execution-based verification. *arXiv preprint arXiv:2405.00253*.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Simon Valentin, Jinmiao Fu, Gianluca Detommaso, Shaoyuan Xu, Giovanni Zappella, and Bryan Wang. 2024. Cost-effective hallucination detection for llms. *arXiv preprint arXiv:2407.21424*.

Wen Xiao and Giuseppe Carenini. 2022. Entity-based spancopy for abstractive summarization to improve the factual consistency. *arXiv preprint arXiv:2209.03479*.

Haopeng Zhang, Semih Yavuz, Wojciech Kryscinski, Kazuma Hashimoto, and Yingbo Zhou. 2022. Improving the faithfulness of abstractive summarization via entity coverage control. *arXiv preprint arXiv:2207.02263*.

Yichi Zhang. 2024. Detecting code comment inconsistencies using llm and program analysis. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 683–685.

A Prompts

Summary Generation Prompt

Assume you are an expert in understanding JAVA code.

Question: As a Java Expert, please provide a detailed summary of the following Java code with the following sections:
1. Inputs and outputs of the method
2. Business purpose
3. Detailed functional summary of the method.

“‘
{CODE}
“‘

Figure 4: Summary Generation Prompt- This prompt was used for generating the summaries from different language models

Intent Verification Prompt

Assume you are an expert in understanding JAVA code. Your task is to verify whether the description of ’mapped_entity’ in the given text is correct, incorrect, or irrelevant with respect to the code. Only output one of the following labels: [“CORRECT", “INCORRECT", “IRRELEVANT"].

Description:
$\{relevant\_sent\}$

$[CODE]$
{CODE}
$[/CODE]$

Figure 5: Intent Verification Prompt- This prompt was used for verifying the description of a given entity based on the sentences that mention the entity

Named Entity Recognition Prompt

Assume you are an expert in understanding Java and performing named entity recognition related to Java code. You have to label the entities by considering the following labels:

Code Entities: CLASS, VARIABLE, FUNCTION, LIBRARY, VALUE, DATA TYPE, and HTML or XML TAG
Natural Language Entities: APPLICATION, UI ELEMENT, LANGUAGE, DATA STRUCTURE, ALGORITHM, FILE TYPE, FILE NAME, VERSION, DEVICE, OS, WEBSITE, and USER NAME.

For every entity in the input mention the entity_type in the given format only. Strictly follow this template and only print the output without any other word. You can follow the example below:
‘‘‘
{Incontext Example}
‘‘‘
Now consider the summary describing a code below:
{generated_summary}

Figure 6: Named Entity Recognition Prompt

Direct Evaluation Prompt

Assume you are an expert in understanding JAVA code. Your task is to verify if the description of the code entities present in the given summary is correctly described or NOT as per the code logic. Output all the ‘entity_name’ and a relevant_sentence’ corresponding to the ‘entity_name’, which are incorrectly described. Do not provide any other details. Strictly follow this format:
[$entity\_name$: “”, $relevant\_sentence$: “”]
Summary:
{SUMMARY}

Code:
{CODE}

Figure 7: Direct Evaluation Prompt- This prompt was used to detect the hallucinated entities and sentences from the summary without breaking into entities

B Experimental Setup

In our setup, we conducted all the experiments using NVIDIA A100-SXM4-80GB GPU in a single or multi-GPU environment. For our experiments, we consider instruction-tuned versions of the SOTA code and language models, from IBM-Granite family (20B-instruct; 34B-instruct) (Mishra et al., 2024), Llama3 family (8B-instruct and 70B-instruct) (Touvron et al., 2023), CodeLlama family (7B and 34B) (Roziere et al., 2023) and Mistral family (7B-instruct) (Jiang et al., 2023). We use the GPT4-Omni version for our framework and keep the temperature at 0.3 and set max_new_tokens to 4000.

C NER Evaluation

This section discusses the NER performance of various models considered in this work. To perform NER using LLMs, we provide the code summary and NER tagset in the prompt (Appendix 5) using a one-shot in-context example to extract all the entities discussed in the summary accompanied by their types. To evaluate the entity extraction, we assess two key aspects: entity coverage and entity type correctness. 1) **Entity Coverage**: This measures whether all valid entities in the summary are detected. We quantify this using the Jaccard Similarity between the entities in the generated output and those in the ground truth. 2) **Entity Type Correctness**: This evaluates whether the detected entities have been assigned the correct types. For this, we use the F1 score as the metric.

Models	Jaccard Similarity	F1
GPT-4-Omni	0.81	0.92
Gemini-1.5-Flash	0.64	0.92

Table 4: **NER Results on Human Data**

We observed a good correlation between GPT4-Omni and human data and, therefore, used it for NER in our pipeline. As an additional contribution, we also evaluate the open-source models considered in this work for the task of Named Entity Recognition on summaries generated from 600 code snippets initially sampled from CodeXGlue data using GPT predictions as ground truth.

Models	Jaccard Similarity	F1
Llama3-8B	0.5298	0.78
Llama3-70B	**0.5981**	**0.90**
Mistral-7B	0.4458	0.65
Granite-20B	0.4897	0.85
Granite-34B	0.48181	0.84
Codellama-7B	0.4586	0.84
Codellama-34B	0.5079	0.83

Table 5: **NER Results on GPT Data**

D Intent Detection

In this section, we describe the two distinct approaches for intent detection.

D.1 String Matching Heuristics

By string matching heuristics, we mean character-level matching with the following regex expressions:

- The word is either preceded by or succeeded by a space char
- ignore the “`” characters since some of the entities are enclosed using these quoted marks by models.
- Account for brackets: some of the function names in the summary include “()” and some of the variables include “[]”

The above regexes are designed to capture all the cases of entity forms in summary. It can be noted that these regex rules are evaluated in a single module.

D.2 Prompt based Approaches

Here, we discuss the general prompt-based approach we tried for Intent detection. We give the complete prompt in Figure 8. Qualitatively, we observed the following drawbacks of this approach:

- We observe high Hallucination in the generated output, which leads to the introduction of fabricated sentences not present in the summaries.
- We observed inaccurate extraction of the sentences, where the extracted sentence has slight variations from the original sentence present in the summary.
- We observe missing sentences, i.e., not all the sentences discussed in the summary are captured in the generated output.

Intent Detection Prompt

Assume you are a Java expert. You have to identify all the relevant sentences about the given entity. Here, a sentence is relevant to the mapped entity if the sentence discusses the given entity. You have to generate the output strictly in the JSON format: $\{entity\_name : \text{“”}, relevant\_sentence : \text{“”}\}$
Given Entity:
{mapped entity}
Given Summary:
{SUMMARY}

Figure 8: Intent Detection Prompt- This prompt was used to retrieve all the relevant sentences from the summary

These observations led us to prefer simpler string-matching heuristics, which are significantly cheaper in computational aspects.

E Annotation Details

We discuss our annotation process and the annotator's guidelines here:

E.1 Background

The dataset was annotated by eight annotators who are experts in Java and held at least a Master's degree in Computer Science, with some having a PhD in the field. On average, the annotators had 4+ years of experience in Java programming.

E.2 Guidelines

The annotation process was conducted in two stages. In the first stage, we implemented a three-step procedure to annotate hallucinations in code summaries independent of specific hallucination categories.
The first part of the annotation process involved validating the Named Entity Recognition Output. This involved annotating missed or incorrectly identified entities in the summary itself by selecting the appropriate label from the drop-down.
The second part of the annotation involved evaluating if each sentence accurately describes the entity in the code snippet, marking the entity-sentence pair as:

- **CORRECT:** If the relevant sentence correctly describes the code

- **INCORRECT:** If the relevant sentence incorrectly describes the code
- **IRRELEVANT:** If the relevant sentence does not talk about the mapped entity itself

The third part involved rating the summary based on hallucination severity as

- **POOR** (Most part is hallucinated): The generated code summary shows below-average correctness.
- **FAIR** (Only some part is hallucinated): The generated code summary meets expectations.
- **GOOD** (Almost no hallucination): The generated code summary is completely correct.

The second stage involved defining hallucination categories based on annotator feedback and organizing them into a structured taxonomy (Figure 2). This finalized taxonomy was then provided to the annotators, who were asked to assign a specific hallucination category from the predefined options. Annotators were also encouraged to include comments explaining their annotations, as these explanations can be useful for researchers utilizing our dataset.