Quantifying Uncertainty in LLM Descriptions of Binary Code Functions

Anonymous authors
Paper under double-blind review

Abstract

Large language models have demonstrated effectiveness in summarizing source code, suggesting potential in using them to summarize compiled software binaries. However, compilation removes linguistic cues used for comprehension, leading to a lower information environment with greater reliability challenges than source code. Uncertainty quantification (UQ) methods offer a way to identify low-quality summaries but have not yet been explored in this domain. We create a framework generating descriptions of functions from decompiled software, score them against ground truth summaries created from source code, and evaluate the ability of UQ metrics to separate correct and incorrect descriptions. Our results show several metrics capable of identifying functions with unreliable description quality, with the best performing metric using cosine similarities between embeddings to measure the consistency of descriptions generated by leveraging input variations of related clusters in the call graph, demonstrating UQ's potential to validate results to users in this domain.

1 Introduction

Understanding how compiled software behaves, without access to source code, is fundamental to both finding security flaws and analyzing malicious programs (Votipka et al., 2020). As part of the incident response to discovered malware, analysts examine compiled executables to determine functionality, extract indicators of compromises, and create detection signatures: rules and patterns that help quickly find related infections and detect future variants (Botacin, 2024). This task is both cognitively demanding and time-constrained. The problem is compounded by the sheer volume of malware in circulation—analysts routinely face backlogs that far exceed their capacity for manual review (Botacin, 2024).

Analysts often depend on decompilers: tools that translate machine instructions into higher-level, more readable, source code-like representations (Al-Kaswan et al., 2023). This speeds up reverse engineers (REs) by producing more readable, abstract representations of the program instead of raw assembly language (Yakdan et al., 2016). However, even state-of-the-art decompilers produce code that is difficult to interpret; function and variable names are absent or non-descriptive and the code's purpose remains unclear.

Recent advances in large language models (LLMs) have led researchers to examine how LLMs can assist reverse engineers interpreting binary code. LLMs have been shown useful in a number of reverse engineering tasks including generating natural-language summaries of decompiled functions, aiming to provide analysts with a quick description of what each function does (Al-Kaswan et al., 2023; Shang et al., 2025). Promising early results suggest that these models can generate summaries that reduce analysts' workload and time.

Despite these promising results, deploying LLM-generated summaries in operational reverse engineering workflows raises a central concern: reliability. A summary that is fluent but incorrect, or simply incomplete, can mislead analysts, resulting in misclassification, missed detection of malicious behaviors, or wasted effort. Reliability challenges can stem from both limitations in LLMs (hallucinations, overconfidence, etc.), but also limitations in the inputs. While decompiled code resembles the source code many LLMs are trained on more closely than assembly, it can be imperfect or misrepresent the true program behavior (Cao et al., 2024).

These observations point to multiple sources of *uncertainty*. One source is uncertainty over inputs, which arises when the LLM must infer a function's behavior from incomplete evidence, especially when types, names, and higher-level structure are missing or when behavior depends on indirect calls, data flow, or external libraries. The model and the outputs themselves also introduce uncertainty, from limitations in training or the specificity of the descriptions. In security-critical settings, these all can lead to operational risk. This motivates a practical question: *when should an analyst trust an LLM-generated summary?* If uncertainty can be quantified and calibrated, analysts or frameworks can review low certainty samples or leverage high certainty LLM-generated summaries to refine summaries of more challenging functions.

Sampling-based uncertainty quantification (UQ) methods offer a potential solution. Techniques such as Semantic Entropy (Farquhar et al., 2024a) and SelfCheckGPT (Manakul et al., 2023), estimate model confidence by generating multiple outputs for the same input and measuring their consistency. Responses that vary widely across samples suggest the model is uncertain; responses that converge suggest confidence. These methods have shown promises for detecting hallucinations in open-domain question answering and biographical generation, achieving area under the receiver operating characteristic curve (AUROC) improvements of 10–15% over naive baselines (Farquhar et al., 2024a).

Yet it remains unclear whether these methods transfer to the domain of binary code summarization. Decompiled code differs substantially from the natural-language corpora on which UQ methods were developed and evaluated. Function summaries are technical, domain-specific, and grounded in program semantics rather than world knowledge. The failure modes may differ; an LLM might hallucinate a function's purpose while remaining internally consistent across samples, or it might produce semantically equivalent descriptions that appear superficially inconsistent. Moreover, functions in real binaries have strong dependencies with other functions. One function may call other functions, and in order to understand its behavior, one needs to also understand the behaviors of the called functions. This raises two questions: which UQ signals reliably predict summary quality, and can we utilize function dependencies in the call graph to augment existing concepts for quantifying uncertainty in large language models?

In this paper, we introduce an uncertainty-aware experimental framework to generate LLM-based descriptions of binary functions using graph clustering contexts from C++ GitHub binaries aligned with source code. We develop a ground truth comparison against source code generated descriptions using LLM-as-a-Judge (Zheng et al., 2023) with manual human judge inter-rater agreement study. We then perform what is to our knowledge the first comparison of UQ metrics in this domain, evaluating several novel consistency metrics leveraging the program call graph against several existing metrics using the Prediction Rejection Ratio (Malinin et al., 2017). Our results demonstrate that such metrics can identify functions where the quality of LLM descriptions are more sensitive to model context, and separate correct from incorrect description even when source code is not present.

2 Background

This work sits at the intersection of LLM-assisted binary analysis and uncertainty quantification for large language models. We first review prior work on binary code summarization, then summarize the uncertainty quantification methods most relevant to our setting.

LLMs for Binary Code Summarization Binary reverse engineering analyzes compiled executable binaries to recover their structure, behavior, and high-level intent when source code or symbols are unavailable. Typical tasks include summarizing function behaviors, recovering function and variable names, inferring data types, and detecting vulnerabilities. Together, these methods support capabilities such as vulnerability discovery and mitigation, malware analysis and attribution, and software provenance and migration.

Within reverse engineering, binary code summarization involves describing a function's behavior and intent into a short, human-readable summary from machine-level representations of a binary. Recent work has adopted large language models to generate binary summaries from various representation inputs, including decompiled code, assembly instruction sequences, or multi-modal program representations. Al-Kaswan et al. (2023) introduce BinT5, the first pre-trained generative model for binary code summarization, by fine-tuning the source-code model CodeT5 (Wang et al., 2021) on pairs of decompiled code and code documentation.

Xiong et al. (2023) introduce HexT5 which further expands this approach to multi-task learning and supports various downstream tasks including code summarization, variable name recovery, function name recovery, and similarity detection. Using LLMs to generate better decompiled code has also received a fair amount of attention, with a recent survey by Wang et al. (2026) highlighting trends such as recompiling code to verify correctness of decompilation.

Other work incorporates structural information from the binary in addition to pseudo code to summarize binaries. CP-BCS (Ye et al., 2023) encodes the bidirectional instruction-level control flow graph (CFG) and pseudo code from decompilation to summarize stripped binary functions. Zhu et al. (2025) utilize a multi-modal heterogeneous code graph that aligns assembly code graphs with pseudo-code graphs for intent-aware summarization.

Evaluation of binary code summarization typically relies on metrics adapted from machine translation, such as BLEU (Papineni et al., 2002), METEOR (Banerjee & Lavie, 2005), and ROUGE-L (Lin, 2004). However, these metrics focus on lexical similarity and often fail to capture the semantic equivalence between the binary function and the generated summary. To address this, recent work has suggested new metrics. Jin et al. (2023) quantify the semantic similarity between the ground-truth and reference summaries by computing the cosine similarity of their embeddings. Lu et al. (2025) introduce BLEURT-sum that is based on BLEURT (Sellam et al., 2020) but is further fine-tuned on malware code, decompiled pseudocode, and functional summaries to evaluate summarization of malware code. However, we have not seen other work incorporating UQ for evaluating the reliability of generated binary code summaries.

Uncertainty Quantification for LLMs Uncertainty quantification is an important component of modern machine learning as it helps characterize when the input data or model might be *insufficient* for a task (Stracuzzi et al., 2018), providing evidence that outputs may not be correct or robust. In large language models, this issue is especially important because fluent outputs can still be factually incorrect, incomplete, or misleading. A survey by ShorinwaOla et al. (2025) and benchmarking effort from Vashurin et al. (2025) provide overviews and comparison of many strategies; we will highlight a few common high level approaches.

Due to their low cost, one baseline type of approaches for LLM UQ evaluations are self-confidence or self-verbalization methods (Tian et al., 2023; Xiong et al., 2024). Here, the LLM is asked to assign a confidence score to its answer when generating a response. While this method has a low cost, it can be prone to overconfidence (Xiong et al., 2024; Groot & Valdenegro Toro, 2024), struggle to identify a lack of knowledge in reasoning models (Zeng et al., 2025), or perform better only for larger models (Vashurin et al., 2025).

Another common class of approaches leverage the model's output probability distribution and is sometimes referred to as information-based methods (Vashurin et al., 2025). LLMs generate tokens by assigning probabilities over a vocabulary of possible next tokens conditioned on the input prompt and the previously generated context. Metrics in this category are motivated by the idea that the shape of this distribution can provide information about model uncertainty. For example, when several candidate continuations receive similar probability mass, the model may be more uncertain; a sharply concentrated distribution implies great certainty (Fomicheva et al., 2020).

A challenge with these metrics is that they may contain noise from irrelevant sources of uncertainty (Vashurin et al., 2025). One metric that attempts to address some of these issues is Semantic Entropy (Farquhar et al., 2024b), which first groups model generations into semantically similar sets and then measures uncertainty across those groups rather than across individual generations.

Evaluating the *consistency* of outputs is another common category of UQ metric, in which multiple generations are produced for the same input and then compared. If the responses vary substantially, the model may be uncertain; if they converge, the model may be more reliable. Although Semantic Entropy requires access to output probabilities, consistency-based methods can also be applied in black-box settings by measuring similarity directly between generated texts. One example constructs a similarity matrix over sampled outputs and uses the spectrum of the corresponding graph Laplacian as a measure of consistency (Lin et al., 2024). Our work builds on this intuition, extending consistency-based ideas to sets of similar inputs while targeting low query cost in order to identify functions for which the LLM exhibits higher uncertainty.

3 Datasets and Models

Datasets Practically, the uncertainty quantification of function summaries from a binary matters most when source code is unavailable. However, evaluation requires decompiled functions to be aligned with their corresponding source code counterparts. Therefore, we selected C++ GitHub projects that satisfied the following criteria: we could compile a binary from the project, the project possessed a permissive license to permit sharing the data, the binaries were neither trivial nor extremely large (between 100 and 10000 functions), the binaries did not contain keywords "test", "debug", or "sample" in their build path, and initial filtering suggested at least 50% of functions could be linked to their source code. These binaries were then stripped of function identifier names to be realistic and challenging to summarize. We also evaluate results only on functions without descriptive names (*e.g.*, names that are memory address references such as FUN_00102040). After initial filtering and extracting our features, we had a set of 393 featurized binaries.

Additionally, linguistic cues such as function names or meaningful strings in the binary could quickly indicate what the purpose of the function is to a language model, and reduce the uncertainty in creating descriptions. Therefore, we divided the decompiled functions into 8 categories based on how much linguistic information was present in function identifiers and strings. There are 4 levels of function labels: no calls are made in the function, all function calls have descriptive names, some functions calls have descriptive names, and no function calls have descriptive names. For strings, we track whether there are string literals present as a yes/no categorical variable.

To keep experiments manageable, we downsampled from the larger dataset into two sets of 20 binaries. First, we selected 20 binaries that gave high coverage across these information categories (*Initial Sample*). However, also we sampled an additional 20 binaries randomly after completing metric selection to verify replicability (*Replication Set)*. We evaluated around 6900 functions in the first set and 2600 in the second (after filtering for unnamed functions with linked source code that were successfully processed through the framework), with additional details on the dataset in Appendix 8.1.

Models For this study, we selected three recent open access LLMs: `LLama3.3 70B Instruct`[1], `GPT OSS 120B`[2] (OpenAI, 2025), and `Devstral2 123B Instruct`[3]. They represent a range of capabilities of LLMs that were high performers for open models at time of their release. We do not include frontier commercial models like Claude as they do not provide access to test all metrics, and for practical relevance, reverse engineers may not be able to share sensitive samples to external cloud hosted models. All Models were hosted on vLLM servers (Kwon et al., 2023) and queried in parallel for efficiency.

4 Description Generation Pipeline and Sources of Uncertainty

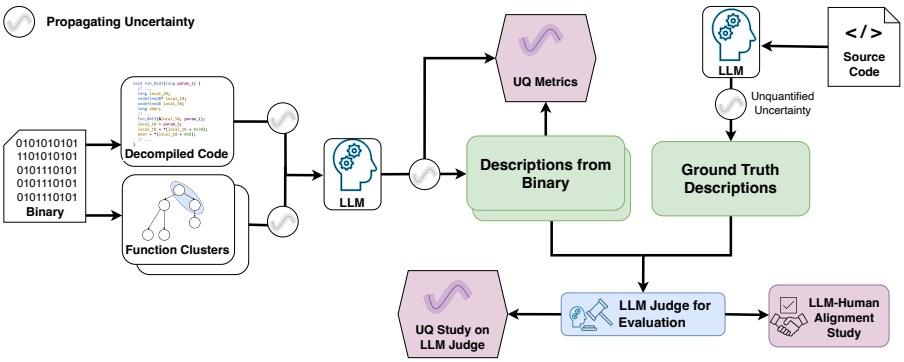


Figure 1: High level diagram of the description generation and evaluation framework. The diagram shows where uncertainty may arise, how it propagates through the process, and where we try to quantify it.

[1]https://huggingface.co/meta-llama/Llama-3.3-70B-Instruct
[2]https://huggingface.co/openai/gpt-oss-120b
[3]https://huggingface.co/mistralai/Devstral-2-123B-Instruct-2512

```
name: mkdir_recursive_at_cwd,
domain: filesystem,
input: [{name: path, type: char *}, {name: mode, type: mode_t}],
output_type: int64 (0 on success, -1 on error),
description: Creates the directory hierarchy specified by *path* relative to the current
 working directory. It is a thin wrapper around the internal recursive mkdir
implementation (FUN_0012faf4) that passes AT_FDCWD (-100) as the directory file
descriptor, then forwards the path and mode arguments. Returns 0 on success or
propagates the error code on failure.,
confidence: 9
```

Figure 2: Example Function Description Generated by an LLM

Figure 1 gives an overview of the process of generating descriptions which specifies where potential sources of uncertainty arise and which uncertainties this study attempts to quantify. The framework extracts decompiled code from the binary using the Ghidra Reverse Engineering Framework[4], as well as taking in clusters of related functions in the call graph to provide context for the LLM generated descriptions. Ghidra is one of the more popular reverse engineering frameworks that supports a wide variety of languages and reverse engineering tasks, including decompilation and building call graphs, which we leveraged to obtain our data.

Both data extraction stages have the potential to introduce uncertainty. Decompilers like the one Ghidra uses can fail to produce good decompiled code. Additionally, function clusters may contain unrelated functions or lack sufficient context to generate the description. While uncertainties from these two points are not directly quantified independently, their impact propagates to later in the pipeline and are quantified by metrics after generation of decompiled descriptions (*UQ Metrics* in Figure 1). Metrics quantifying uncertainty in this region are the key focus of this study and can indicate either inherent uncertainty in the LLM, or insufficiency in the inputs (either from poor decompiled code or lack of necessary related functions).

The other uncertainty we quantify in this study (*UQ Study on LLM Judge* in Figure 1) comes from the ground truth evaluation. Objectively assessing the quality of generated text is difficult, and so we also assess uncertainty present here, as described in Section 5.

Finally, we do not measure the uncertainty of the generated descriptions from source code. Source code summarization does not suffer the same information loss binary summarization, so one of assumptions of this work is that these source code summarization are "good enough", based on recent AI model performance in understanding source code as well as some manual inspection.

4.1 Generating Function Descriptions

When identifiers are removed from code, a single function's decompilation will not always include sufficient context for the LLM to unambiguously describe the code. Therefore, we provide clusters of related functions to the LLM as input context. To create descriptions, we iterate over the functions in the cluster, producing one description per prompt, which is given the decompiled code of the target function along the decompiled code of the other functions in the cluster. If this context is too long, we truncate parts of the cluster until the prompt is under 32K tokens. The prompt instructs the LLM to create a *structured* description in JSON format, where the model creates several fields - a suggested name, the domain of the function, input and output names and types, a short description of a few sentences, and a self confidence score, as seen in Figure 2. We tested other strategies, including asking models to simultaneously describe all functions in the cluster, generating descriptions starting at the cluster leaves and propagating them up the call graph, but among the methods tested, the whole cluster performed best in preliminary experiments, so we selected it for this UQ study.

To select clusters of related functions, we wanted an algorithm that would guarantee that functions could appear in multiple clusters, so we chose an algorithm based on simulated annealing (Kirkpatrick et al., 1983) that maximizes a modularity (Newman & Girvan, 2004) metric (details in Appendix 8.2). This algorithm

[4] https://github.com/NationalSecurityAgency/ghidra

has some inherent randomness, so by running it multiple times, we could generate different clusterings of the nodes, creating input variability that our developed UQ metrics will leverage (Section 6.1) While we considered other clustering algorithms, including Louvain based clusters (Blondel et al., 2008), ModX (Yang et al., 2022), and a proprietary algorithm, these algorithms did not guarantee that a particular function would appear in a set number of clusters.

5 Ground Truth Evaluation

To evaluate the quality of our decompiled function descriptions, we need metrics to evaluate how accurately they reflect descriptions of the corresponding source code. Traditional metrics in this space, such as BLEU score (Papineni et al., 2002), ROUGE (Lin, 2004), *etc.*, are insufficient to capture the semantic similarities between the two descriptions and large scale human manual scoring is too expensive. Two alternatives we considered are using the *cosine embedding distance* between the generated and ground truth description, and *LLM Judge methods*, as they are targeted more towards the semantic meaning of the descriptions. Ultimately, for this study we settled on LLM Judge methods with categorical numerical scores. The Judge methods were easier to threshold to create binary correct/incorrect labels when comparing our UQ metrics in Section 6 and the categories could be defined to have interpretable meanings.

5.1 LLM Judge as a Proxy for Human Evaluators

Methodology LLM Judge or LLM-as-a-judge methods are becoming more popular in LLM evaluations (see Gu et al. (2024) for a survey of applications), as these methods can stand in for more expensive human evaluations. As LLMs have improved instruction following and alignment with human preferences, the argument for using LLM Judge techniques is that they can act as a reasonable proxy for a human rater. When they can act as a proxy for humans, they allow a far more scalable method to judge the quality over large datasets where asking human judges would be infeasible.

The usefulness and applicability of LLM Judges come with several important caveats. First, if humans with subject matter expertise cannot reliably make a judgment of quality, we cannot expect LLMs to perform well either. As a result, the Judge needs to have a clear rubric that defines what the scores mean. If the definitions in the rubric are poor or too ambiguous, any rater will give scores of limited reliability. To minimize ambiguity in judgments, we use a rubric that we refined to classify descriptions into a range of quality from 1 to 5, and also include few shot examples of judgments to further clarify. The text of our judge prompts can be found in Appendix 8.7. Secondly, regardless of whether a judge is a human or LLM, the inputs provided must be sufficient to make the judgment. As our focus is on descriptions of functions, the text of the description is both sufficient and not too large to make the scope of evaluation challenging. Finally, LLM Judges can be susceptible to adversarial manipulations (*e.g.,* (Raina et al., 2024)), though exploring the impact of these is outside the scope of this study.

Since accurately measuring the quality of generated descriptions when source code is available is necessary to correctly assess the usefulness of the UQ metrics we test later, we evaluate LLM Judge scoring against ground truth in a couple ways. In Section 5.2 we describe a manual study to see how well LLM Judge scores align with human rater judgments, and in Section 5.3 we employ uncertainty quantification on the LLMs themselves to see how consistent their scores are under variations of the prompts.

5.2 Evaluation of LLM Judge and Human Judge Alignment

To evaluate the alignment of the LLM Judge to human rating, we take random stratified samples of 30, covering both a range of models and specific scores on the 1-5 scale. Here, a sample is a pair of descriptions (one generated from a decompiled function and another from the ground truth source code) along with the score from the Judge model evaluating their similarity. Specifically, we took 10 samples from each of our three main description-generating models, along with using `GPT OSS 120B` as the judge model, and evenly sampled from each of the 5 score categories. This gives two examples for each combination of score category and generating model within each stratified sample of 30. As we have structured JSON responses (see the example in Figure 2), we wanted to evaluate judging on the entire description (referred to as *structured*

from now on), and on just the "description" field (referred to as *base*), providing two sets of samples for 60 evaluations in total. We test on both these variants of the output to see if results are consistent when considering just a focused text summary and when considering the full structured output that gives names to the function, it's parameters, and it's domain.

Each sample of 30 is evaluated by two human raters (from the paper's authors) according to the following procedure. First, the raters look at the LLM Judge prompt and familiarize themselves with the rubric of scoring for each category. The raters then calibrate by rating one example together, before separating and grading the rest of the examples separately. Afterwards, the raters compare their ratings on each example to determine where there are disagreements and adjust the scores if there are misunderstandings to create a score before and after discussion.

The ratings between the human raters are graded with a weighted Cohen's Kappa (Cohen, 1968) to assess agreement, reporting both the linear and quadratic weighted Cohen's Kappa. These weights apply different penalties for how much the raters disagree, so we report both for completeness. As the judge ratings are ordinal data on a 1-5 scale, the unweighted Cohen's Kappa is not appropriate, so we do not report it. For comparing the two judges with one or more LLM Judges, we use Kendall's W (Kendall & Smith, 1939). For assessing the agreement between raters, we use common interpretation categories inspired from Landis and Koch Landis & Koch (1977) (*e.g.*, where above 0.6 is substantial agreement and above 0.8 is strong agreement). If humans agree well on the scoring and LLM Judges fall within that range, they could be considered a reasonable proxy for a human judge.

After this initial selection of a `GPT OSS 120B` Judge, we also evaluate the agreement with our other models (`LLama3.3-70B-Instruct, Devstral2-123B-Instruct`), selecting the same set of samples and recording the model scores. This does not involve additional manual alignment studies as the selected set of pairs and human ratings are reused.

Results First, the top of Table 1 shows agreement between the two human raters on each study using Cohen's weighted κ with both weight penalties. Before the discussion between the raters, agreement was moderate at worst, turning into strong agreement (above 0.8) afterwards. Post discussion, human disagreement was usually degree of rating off at most, and reflected some of the limitations of a 5 category system (e.g., raters were uncertain if something was a 2 or 3, and one rated it a 2, another a 3). The bottom of Table 1 then shows the Kendall's W between human raters and each of the models individually, and then with all models together. All of the agreements are quite high (over 0.8), suggesting that these model and the chosen prompt are a reasonable stand in for a human judge's evaluation of the description's quality.

Table 1: Inter-rater agreement between the two human raters (top) and between human raters and specific models (bottom). The top results show both types of weighted Cohen's κ before and after the judges discussed and resolved disagreements. The bottom results show Kendall's W agreements of each model with both human raters before and after disagreement resolution.

LLM Judge Prompt	**Weighted Cohen's κ (Linear/Quadratic)**	**Weighted Cohen's κ (Revised) (Linear/Quadratic)**
Base Judge	0.693/0.798	0.885/0.949
Structured Judge	0.585/0.789	0.833/0.925
LLM Judge Model	**Kendall's W (Base Judge) (Original/Revised)**	**Kendall's W (Structured Judge) (Original/Revised)**
`LLama3.3-70B-Instruct`	0.863/0.916	0.857/0.865
`GPT OSS 120B`	0.872/0.923	0.904/0.928
`Devstral2-123B-Instruct`	0.880/0.919	0.878/0.895
All	0.866/0.892	0.856/0.858

5.3 LLM Judge UQ Validation

Methodology To check the consistency of the LLM Judge models, we incorporate a method based on the work by Xiong et al. (2024). In this method, we modify the LLM Judge prompts to remove the few shot examples and the output instructions with a template. We create five variations of model instructions using a template, instructing the model to apply variations using confidence scores, different chain of thought (Wei et al., 2022) style prompting, and top-K methods (in our case, K is 5, and the model is asked to order the 1-5 scale by order of preference). By using this method, we can then compute Kendall's W again to evaluate the inter-rater agreement among the results. If the judge is consistent (above 0.8), we have greater confidence that the scores are resistant to specifics in the prompts.

Table 2: Kendall's W inter-rater agreement within each model when varying prompt output format.

Model	Kendall's W (Description)	Kendall's W (Structured Description)
`LLama3.3 70B Instruct`	0.802	0.863
`GPT OSS 120B`	0.884	0.909
`Devstral2 123B Instruct`	0.423	0.657

Results Table 2 shows the agreement of each model with itself, under 5 different variations of the prompt, requesting different styles of output formatting. While `Llama3.3 70B Instruct` and `GPT OSS 120B` show high agreement, there is much lower self consistency from the `Devstral2 123B Instruct` model. Diving a little deeper, we can see that when prompted to rank all 5 scores without using a Chain of Thought, the `Devstral2 123B` model becomes wildly different from all other variations[5]. As a result, while we include results of Devstral2 Judge rankings in other experiments (which rely on the original prompt aligned with humans), the greater unreliability and susceptibility to prompt variation puts more skepticism on the model's judgments, particularly if they disagree with other models.

6 Uncertainty Quantification Metrics for Decompiled Code Function Descriptions

While ground truth comparisons are useful for measuring model capabilities on benchmarks, in practice, reverse engineers often try to understand binary behavior when they do not have source code available. To be practically useful, the metrics must only leverage information present in the binary. As a system may process many functions and binaries, time and compute efficiency are also preferable - a metric identifying problems in fewer queries is preferable. Finally, metrics that require less access to model information have broader applicability. Therefore, we limit our focus to black box and gray box metrics, where we define as requiring only the output text or the output text and log probabilities from the description generating model, respectively. White box metrics requiring information about model training or weights are excluded.

6.1 Selected Gray and Black Box Uncertainty Quantification Metrics

Existing Metrics The baseline UQ metric we use is self confidence, which is produced as part of the LLM description at minimal extra cost. The metric has the model rate its confidence in the description on a scale of 1 to 10. We also leveraged the open source LM-Polygraph framework (Vashurin et al., 2025) to implement the Semantic Entropy (Farquhar et al., 2024b) and the Eigenvalue Laplacian (Lin et al., 2024) metrics, with the default query count (10 generations). While Semantic Entropy is categorized as a white box metric in their framework, it only requires access to log probabilities (gray box under our definitions), and so we modified the code slightly to run more efficiently with our vLLM server (Kwon et al., 2023). We also tried to run the similarly classified Claim Conditioned Probability metric (Fadeeva et al., 2024), but were unable to get it to run efficiently with our framework.

Developed metrics Drawing from evidence from some other domains (Savage et al., 2024) on the effectiveness of consistency based metrics, we focused heavily on developing consistency-based metrics for our

[5]Appendix 8.3 shows the correlation plots between LLM Judge variants.

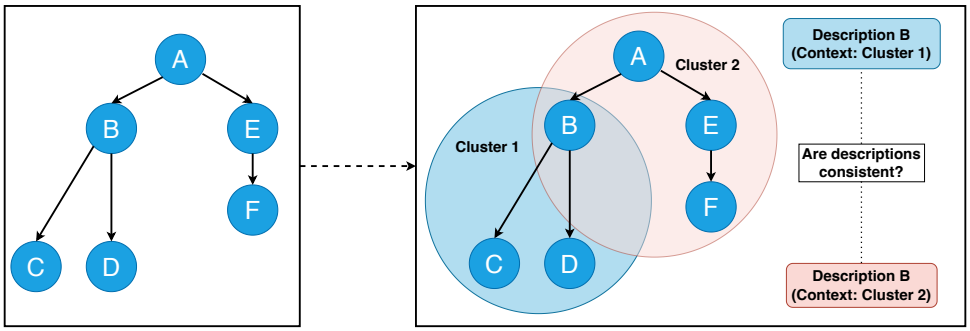


Figure 3: Overlapping functions in a call graph provide variable context to check function consistency.

domain. Our novel developed metrics use this high level idea of consistency and combine it with knowledge of how functions in a program relate to each other through the *function call graph* - *i.e.,* a graph that links together how the functions call each other. The left side of Figure 3 gives an example of a simplified call graph, where a function A invokes two functions B and E, and so on.

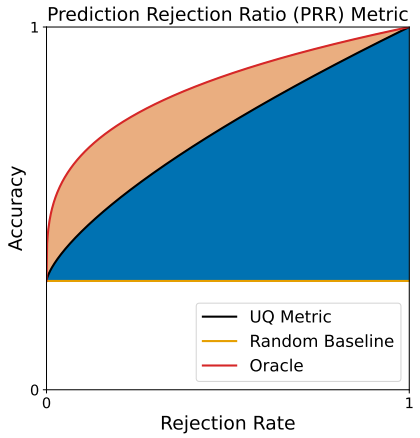
A method to leverage this cluster information is as a source of input variety, altering the input context to determine if the LLM changes its description. By using a clustering algorithm that guarantees all functions will appear in N clusters (refer to Appendix 8.2 for details), we can guarantee multiple contexts for every function with a description generated by the LLM. To test the effectiveness of the metrics under conditions with the least performance overhead, we use $N = 2$ clusters. Then, using function B as an example from Figure 3, we prompt the LLM with different input contexts. If there are inconsistent output descriptions, this indicates greater uncertainty for the LLM on this function.

Under this idea we used two metrics to determine if the description was consistent. First, we used another LLM Judge prompt, asking the model to score on a scale of 1 to 10 if the two descriptions were the same (a larger scale was used to allow more granular thresholding). To avoid potential influence from the order, the prompt permutes the order of the descriptions being evaluated randomly (for N=2, this is just both orders), and we select the minimum score as a worst case estimate. Secondly, we also embedded the descriptions using the `intfloat/e5-mistral-7b-instruct` model and took the cosine similarity of the embeddings as a consistency measure. These metrics provide a lower cost black box method to assess the consistency of the descriptions, leveraging the graph information to inform the context of the generations.

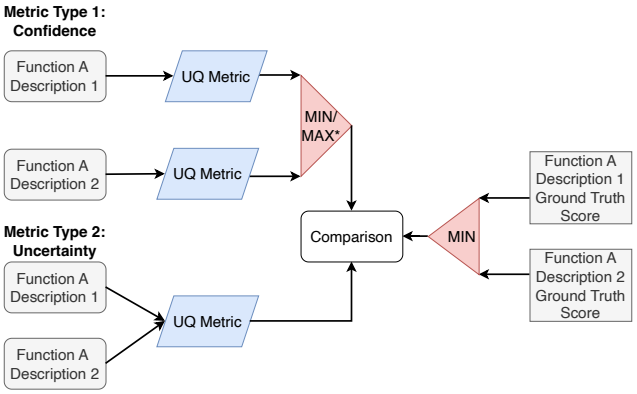
We also considered measuring the consistency of descriptions with their neighbors and other functions within the cluster. These metrics proved to be not very effective, but for completeness, details of how these metrics were calculated along with results are included in Appendix 8.5.

6.2 Evaluating Uncertainty Quantification Metrics

Methodology To compare between UQ metrics, we adopt the Prediction Rejection Ratio (PRR) metric (Malinin et al., 2017). The PRR metric compares the performance of a proposed UQ metric against both an oracle and random baselines, but requires the ability to label samples as either a binary correct/incorrect. To binarize the 1-5 judge rating, we select 4s and 5s on the rating scale as "correct" and 3 or below as "incorrect". This determination was made by both the rubric definition of 4s and 5s and through our manual reviews of the ratings. For additional robustness, we also used each of 3 models to judge the correctness of all the models, with both the *base* and *structured* judges, and report the mean performance across all 6 judges. In the context of PRR, an oracle UQ metric is one that optimally filters all incorrect samples from the data (which is the inverse of the accuracy column). PRR is the ratio between the area under the curve between UQ metric and the random baseline and the oracle metric and the random baseline, as seen in Figure 4a: $\frac{\text{AUC}_{\text{UQ}}}{\text{AUC}_{\text{Oracle}}}$. We implemented this metric using some functionality from the LM-Polygraph framework (Vashurin et al., 2025) though we used the median of 1000 random runs as our random baseline instead of the mean.



(a) Visual demonstration of PRR metric.



(b) Demonstration of two types of metrics for uncertainty quantification and how we use the worst case (either min or max) to evaluate them against the worst case ground truth scores.

Figure 4: Evaluation and aggregation procedure for comparing metrics side-by-side.

Table 3: Mean PRR scores across all 6 LLM Judges for the top 20 binaries sampled for varied function information and The model in each column name corresponds to the model that generated the descriptions.

Top 20 binaries for function information variety	Mean LLM Judge PRR for each generating model		
Metric	GPT OSS 120B	Llama3.3-70B	Devstral2 123B
Self Confidence (Min)	0.449	0.299	0.436
Consistency Embedding Similarity	**0.523**	**0.454**	**0.561**
Consistency LLM Judge (Min)	0.286	0.388	0.474
Eigenvalue Laplacian (Max)	0.371	-0.128	0.331
Semantic Entropy (Max)	0.376	-0.086	0.412
20 randomly sampled binaries	**Mean LLM Judge PRR for each generating model**		
Metric	GPT OSS 120B	Llama3.3-70B	Devstral2 123B
Self Confidence (Min)	0.450	0.277	0.384
Consistency Embedding Similarity	**0.489**	0.447	**0.515**
Consistency LLM Judge (Min)	0.323	**0.453**	0.443
Eigenvalue Laplacian (Max)	0.253	-0.192	0.121
Semantic Entropy (Max)	0.244	-0.163	0.162

Additionally, the metrics we are comparing can be classified into different groups, confidence metrics (type 1) and uncertainty metrics (type 2). Confidence metrics are associated with *individual* generations of model, and uncertainty metrics represent a value over *multiple* iterations of the model. Figure 4b showcases how we decided to put these metrics on a similar footing. For each input variation of the cluster, we take the minimum confidence score and compare against the minimum ground truth score. The rationale for using the min is to capture the worse case performance of the model. If the self confidence is a useful indicator of uncertainty, then a reverse engineer seeing a low self confidence score among the generations could detect uncertainty, while a more stable high level confidence under varying inputs suggests greater model stability and confidence. Additionally, although the Semantic Entropy and Eigenvalue Laplacian metrics are generated over 10 variations, they are associated with a single input cluster context. We aggregate them as well, taking the worse case uncertainty value (max value in this case), so a final single aggregated value exists for decompiled function, enabling a direct comparison of PRR across all metrics.

Overall Results The top half of Table 3 shows overall metric performance using the PRR score for the original set of 20 binaries we sampled that prioritized functions with different levels of information present

Table 4: PRR scores using human judges for accuracy (i.e., average human rater score is greater than 4 after discussion). The *base* and *structured* columns represent two different sets of 30 samples each.

Metric	Human Judge PRR (Base)	Human Judge PRR (Structured)
Self Confidence	**0.692**	0.355
Consistency Embedding Similarity	0.580	**0.699**
LLM Judge Consistency	0.201	0.246
Eigenvalue Laplacian	0.375	0.414
Semantic Entropy	0.057	0.420

Table 5: Performance in PRR of the metrics on the category of functions with the least available linguistic information (all function calls have no meaningful names, and no string literals are present). The model in each column name corresponds to the model that generated the descriptions.

	Mean PRR		
Metric	GPT OSS 120B	Llama3.3-70B	Devstral2 123B
Self Confidence (Min)	**0.436**	0.089	0.375
Consistency Embedding Similarity	0.418	**0.276**	**0.407**
Consistency LLM Judge (Min)	0.303	0.239	0.392
Eigenvalue Laplacian (Max)	0.229	-0.080	0.176
Semantic Entropy (Max)	0.290	-0.027	0.303

in the decompiled code, and the bottom half of the table shows the results when applied to the 20 additionally randomly sampled binaries. Given the varying usefulness of self confidence metrics reported in the literature (Xiong et al., 2024; Savage et al., 2024; Vashurin et al., 2025), one potentially surprising result is that the self confidence baseline competitively filtered poor descriptions. The two existing LLM UQ metrics Semantic Entropy and Eigenvalue Laplacian are sometimes able to discriminate good from bad descriptions, but they also demonstrate significant variation in performance depending on the model. In particular, they are worse than random on `LLama3.3-70B-Instruct`, suggesting that their effectiveness might depend on the model. Self confidence also decreases on this model, aligning with observations from Vashurin et al. (2025) that it might align with model size. Between the two metrics we created measuring the consistency of output with varied graph input, the cosine similarity between embeddings is more effective, and indeed seems to be the best metric overall, with 1 exception for `LLama3.3-70B-Instruct` on the replication sample dataset. Even there, the metric seems less prone to variability than other metrics across the models.

We also used the small set of human judge scores from our LLM Judge reliability to calculate PRR on the functions, with results shown in Table 4. We marked descriptions as accurate if the average human rater score is above 4, and the columns for *base* and *structured* judgments are included separately as they were different sets of functions. As these only represent a total of 30 samples each, with even fewer examples rated 4 or higher, they are prone to potentially large swings in the effectiveness of the metrics, but can sanity check the main results. We see that for the judgments on the base descriptions, the self confidence is marked as the best, and for the structured samples, the embedding consistency metric is the best. However, the self confidence shows more variation in the two sets with a lower minimum, suggesting the embedding consistency metric still remains the best overall.

Results by Function Information Category In Section 3 we described our initial sample of 20 binaries selected to have high coverage over the eight information categories we defined. We want to understand how the available information in the function impacts the effectiveness of UQ metrics. To answer this, we investigate the question in two ways. First, what is the performance of the top metrics on the category with the least amount of linguistic information (all function calls lack meaningful names and no string literals are present)? Second, across all categories, how do the functions perform?

Table 5 shows the mean PRR scores for the category of functions with the least available information - they have no string literals present and all calls to other functions have uninformative names. The embedding con-

Table 6: Performance in PRR of the top metrics across difference categories. The model in each column name corresponds to the model that generated the descriptions. The cutoff for displayed metrics is that the are above 0.3 PRR across all generating models.

Function Information Level	Metric	Mean PRR GPT OSS 120B	Llama3.3-70B	Devstral2 123B
All Calls Named & No Strings	Embedding Consistency	**0.458**	**0.374**	**0.452**
All Calls Unnamed & Strings	Embedding Consistency	**0.459**	**0.336**	**0.468**
All Calls Unnamed & Strings	Consistency LLM Judge (Min)	0.386	0.312	0.452
No Calls & Strings	Self Confidence (Min)	**0.803**	**0.471**	0.643
No Calls & Strings	Embedding Consistency	0.635	0.432	**0.655**
No Calls & Strings	Consistency LLM Judge (Min)	0.473	0.414	0.584
Some Calls Named & No Strings	Embedding Consistency	**0.561**	0.417	0.529
Some Calls Named & No Strings	Consistency LLM Judge (Min)	0.350	**0.425**	**0.557**
Some Calls Named & Strings	Embedding Consistency	**0.513**	**0.428**	**0.556**

sistency metric has the highest scores overall, though self confidence is slightly better on the `GPT OSS 120B` model. The consistency metric is close, and the self confidence's severe drop on `Llama3.3-70B-Instruct` again highlights the impact of the model.

Next, Table 6 shows metric performance across the different information categories, limited to metrics and categories with stronger performances (combinations with < 0.3 PRR are omitted). Once again, the embedding similarity metric most often has the best performance, with some nuances. For example, the category of function that make no function calls to other functions and have string literals represents the most "self-contained" functions, where context from the cluster is less needed for the LLM to have sufficient context. In this case, the self confidence often performs the best, with a very strong showing compared to the oracle for `GPT OSS 120B`. We also see the consistency LLM Judge is sometimes close in performance to the embedding metric, even beating it in the category of functions with a mixture of meaningful function call names, but no strings. However, its performance is much more variable; with a more rigorous judge process as we did for the baseline judge (rubric design, refinement with human studies) it is possible it might better separate good and poor descriptions.

7 Discussion

To summarize, we created a UQ-aware pipeline to automatically generate descriptions of compiled software binaries, which, as far as we are aware is novel to this domain. The partial information environment of compiled software is well suited to the application of UQ, where missing or misleading information may be common. Our results showed that by varying input context of related functions, that low cost black UQ metrics could separate higher and lower quality descriptions, with a cosine similarity over embedded outputs performing best across a variety of models and function types. While this study constitutes only initial experimentation in this area, we argue that the broader successes seen with language models in supporting program understanding in source code make LLMs for reverse engineering a growing area. Even as models improve, the ability to identify where they are more uncertain will remain important, especially in adversarial environments such as reverse engineering malware. More broadly, for applications outside the reverse engineering domain, our work also showcases how uncertainty for LLMs can be applied in graph settings where generated data can be associated with each other, and not just evaluated in isolation.

Future Work There are several directions to expand upon this initial UQ study for binary descriptions. One direction would be to consider how description quality could be improved for less certain examples by iteratively using descriptions on more certain samples. Essentially, a system could use UQ metrics to describe the "easier" examples first and then use that to improve descriptions of "harder" functions. Likewise, if UQ metrics capture different uncertainties, the combining them may provide stronger indicators for discriminating between correct and incorrect generations. Finally, adapting UQ metrics to LLM agentic workflows will become more important as such tooling systems are applied both to reverse engineering tasks and in general for systems where reliability and credibility are important.

References

Ali Al-Kaswan, Toufique Ahmed, Maliheh Izadi, Anand Ashok Sawant, Premkumar Devanbu, and Arie van Deursen. Extending source code pre-trained language models to summarise decompiled binaries. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 260–271. IEEE, 2023.

Satanjeev Banerjee and Alon Lavie. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pp. 65–72, 2005.

Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

Marcus Botacin. What do malware analysts want from academia? a survey on the state-of-the-practice to guide research developments. In *Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses*, pp. 77–96, 2024.

Ying Cao, Runze Zhang, Ruigang Liang, and Kai Chen. Evaluating the effectiveness of decompilers. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 491–502, 2024.

Jacob Cohen. Weighted kappa: Nominal scale agreement provision for scaled disagreement or partial credit. *Psychological Bulletin*, 70(4):213–220, 1968. doi: 10.1037/h0026256.

Ekaterina Fadeeva, Aleksandr Rubashevskii, Artem Shelmanov, Sergey Petrakov, Haonan Li, Hamdy Mubarak, Evgenii Tsymbalov, Gleb Kuzmin, Alexander Panchenko, Timothy Baldwin, et al. Fact-checking the output of large language models via token-level uncertainty quantification. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 9367–9385, 2024.

Sebastian Farquhar, Jannik Kossen, Lorenz Kuhn, and Yarin Gal. Detecting hallucinations in large language models using semantic entropy. *Nature*, 630(8017):625–630, 2024a.

Sebastian Farquhar, Jannik Kossen, Lorenz Kuhn, and Yarin Gal. Detecting hallucinations in large language models using semantic entropy. *Nature*, 630(8017):625–630, June 2024b. ISSN 1476-4687. doi: 10.1038/s41586-024-07421-0. URL https://www.nature.com/articles/s41586-024-07421-0. Publisher: Nature Publishing Group.

Marina Fomicheva, Shuo Sun, Lisa Yankovskaya, Frédéric Blain, Francisco Guzmán, Mark Fishel, Nikolaos Aletras, Vishrav Chaudhary, and Lucia Specia. Unsupervised Quality Estimation for Neural Machine Translation. *Transactions of the Association for Computational Linguistics*, 8:539–555, September 2020. ISSN 2307-387X. doi: 10.1162/tacl_a_00330. URL https://doi.org/10.1162/tacl_a_00330. _eprint: https://direct.mit.edu/tacl/article-pdf/doi/10.1162/tacl_a_00330/1923296/tacl_a_00330.pdf.

Tobias Groot and Matias Valdenegro Toro. Overconfidence is key: Verbalized uncertainty evaluation in large language and vision-language models. In Anaelia Ovalle, Kai-Wei Chang, Yang Trista Cao, Ninareh Mehrabi, Jieyu Zhao, Aram Galstyan, Jwala Dhamala, Anoop Kumar, and Rahul Gupta (eds.), *Proceedings of the 4th Workshop on Trustworthy Natural Language Processing (TrustNLP 2024)*, pp. 145–171, Mexico City, Mexico, June 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.trustnlp-1.13. URL https://aclanthology.org/2024.trustnlp-1.13/.

Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, et al. A survey on llm-as-a-judge. *The Innovation*, 2024.

Xin Jin, Jonathan Larson, Weiwei Yang, and Zhiqiang Lin. Binary code summarization: Benchmarking chatgpt/gpt-4 and other large language models. *arXiv preprint arXiv:2312.09601*, 2023.

M. G. Kendall and B. Babington Smith. The Problem of m Rankings. *The Annals of Mathematical Statistics*, 10(3):275 – 287, 1939. doi: 10.1214/aoms/1177732186. URL https://doi.org/10.1214/aoms/1177732186.

Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

J Richard Landis and Gary G Koch. The measurement of observer agreement for categorical data. *biometrics*, pp. 159–174, 1977.

Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pp. 74–81, 2004.

Zhen Lin, Shubhendu Trivedi, and Jimeng Sun. Generating with confidence: Uncertainty quantification for black-box large language models. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. URL https://openreview.net/forum?id=DWkJCSxKU5.

Haolang Lu, Hongrui Peng, Guoshun Nan, Jiaoyang Cui, Cheng Wang, Weifei Jin, Songtao Wang, Shengli Pan, and Xiaofeng Tao. Malsight: Exploring malicious source code and benign pseudocode for iterative binary malware summarization. *IEEE Transactions on Information Forensics and Security*, 20:6733–6747, 2025. doi: 10.1109/TIFS.2025.3583552.

Andrey Malinin, Anton Ragni, Kate Knill, and Mark Gales. Incorporating uncertainty into deep learning for spoken language assessment. In Regina Barzilay and Min-Yen Kan (eds.), *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 45–50, Vancouver, Canada, July 2017. Association for Computational Linguistics. doi: 10.18653/v1/P17-2008. URL https://aclanthology.org/P17-2008/.

Potsawee Manakul, Adian Liusie, and Mark Gales. Selfcheckgpt: Zero-resource black-box hallucination detection for generative large language models. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 9004–9017, 2023.

Mark EJ Newman and Michelle Girvan. Finding and evaluating community structure in networks. *Physical review E*, 69(2):026113, 2004.

OpenAI. gpt-oss-120b & gpt-oss-20b model card, 2025. URL https://arxiv.org/abs/2508.10925.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pp. 311–318, 2002.

Vyas Raina, Adian Liusie, and Mark Gales. Is llm-as-a-judge robust? investigating universal adversarial attacks on zero-shot llm assessment. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 7499–7517, 2024.

Thomas Savage, John Wang, Robert Gallo, Abdessalem Boukil, Vishwesh Patel, Seyed Amir Ahmad Safavi-Naini, Ali Soroush, and Jonathan H. Chen. Large Language Model Uncertainty Measurement and Calibration for Medical Diagnosis and Treatment, June 2024. URL https://www.medrxiv.org/content/10.1101/2024.06.06.24308399v2. Pages: 2024.06.06.24308399.

Thibault Sellam, Dipanjan Das, and Ankur Parikh. BLEURT: Learning robust metrics for text generation. In Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel Tetreault (eds.), *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 7881–7892, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.704. URL https://aclanthology.org/2020.acl-main.704/.

Xiuwei Shang, Guoqiang Chen, Shaoyin Cheng, Benlong Wu, Li Hu, Gangyang Li, Weiming Zhang, and Nenghai Yu. Binmetric: A comprehensive binary code analysis benchmark for large language models. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence, IJCAI-25*, pp. 7715–7723, 2025.

ShorinwaOla, MeiZhiting, LidardJustin, RenAllen Z, and MajumdarAnirudha. A Survey on Uncertainty Quantification of Large Language Models: Taxonomy, Open Research Challenges, and Future Directions. *ACM Computing Surveys*, September 2025. doi: 10.1145/3744238. URL https://dl.acm.org/doi/10.1145/3744238. Publisher: ACMPUB27New York, NY.

David J Stracuzzi, Michael C Darling, Maximillian G Chen, and Matthew G Peterson. Data-driven uncertainty quantification for multisensor analytics. In *Ground/Air Multisensor Interoperability, Integration, and Networking for Persistent ISR IX*, volume 10635, pp. 155–167. SPIE, 2018.

Katherine Tian, Eric Mitchell, Allan Zhou, Archit Sharma, Rafael Rafailov, Huaxiu Yao, Chelsea Finn, and Christopher D Manning. Just ask for calibration: Strategies for eliciting calibrated confidence scores from language models fine-tuned with human feedback. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 5433–5442, 2023.

Roman Vashurin, Ekaterina Fadeeva, Artem Vazhentsev, Lyudmila Rvanova, Daniil Vasilev, Akim Tsvigun, Sergey Petrakov, Rui Xing, Abdelrahman Sadallah, Kirill Grishchenkov, Alexander Panchenko, Timothy Baldwin, Preslav Nakov, Maxim Panov, and Artem Shelmanov. Benchmarking uncertainty quantification methods for large language models with lm-polygraph. *Transactions of the Association for Computational Linguistics*, 13:220–248, 03 2025. ISSN 2307-387X. doi: 10.1162/tacl_a_00737. URL https://doi.org/10.1162/tacl_a_00737.

Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S Foster, and Michelle L Mazurek. An observational investigation of reverse {Engineers'} processes. In *29th USENIX Security Symposium (USENIX Security 20)*, pp. 1875–1892, 2020.

Yuanchao Wang, Ye Tian, Hongmei Wei, and Fei Wang. Large language models for binary decompilation: Paradigms, evaluation, and challenges. In *Proceedings of the 2026 6th International Conference on Computer Network Security and Software Engineering*, pp. 212–218, 2026.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (eds.), *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 8696–8708, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.685. URL https://aclanthology.org/2021.emnlp-main.685/.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *Advances in Neural Information Processing Systems*, 35:24824–24837, December 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html.

Jiaqi Xiong, Guoqiang Chen, Kejiang Chen, Han Gao, Shaoyin Cheng, and Weiming Zhang. Hext5: Unified pre-training for stripped binary code information inference. In *Proceedings of the 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 774–786. IEEE, 2023.

Miao Xiong, Zhiyuan Hu, Xinyang Lu, Yifei Li, Jie Fu, Junxian He, and Bryan Hooi. Can llms express their uncertainty? an empirical evaluation of confidence elicitation in llms. In *International Conference on Learning Representations*, volume 2024, pp. 23650–23678, 2024.

Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 158–177. IEEE, 2016.

Can Yang, Zhengzi Xu, Hongxu Chen, Yang Liu, Xiaorui Gong, and Baoxu Liu. ModX: binary level partially imported third-party library detection via program modularization and semantic matching. In *Proceedings of the 44th International Conference on Software Engineering*, pp. 1393–1405, Pittsburgh Pennsylvania, May 2022. ACM. ISBN 978-1-4503-9221-1. doi: 10.1145/3510003.3510627. URL https://dl.acm.org/doi/10.1145/3510003.3510627.

Tong Ye, Lingfei Wu, Tengfei Ma, Xuhong Zhang, Yangkai Du, Peiyu Liu, Shouling Ji, and Wenhai Wang. Cp-bcs: Binary code summarization guided by control flow graph and pseudo code. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 14740–14752, 2023.

Qingcheng Zeng, Weihao Xuan, Leyang Cui, and Rob Voigt. Thinking out loud: Do reasoning models know when they're right? In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 1394–1407, Suzhou, China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-332-6. doi: 10.18653/v1/2025.emnlp-main.73. URL https://aclanthology.org/2025.emnlp-main.73/.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

Kangchen Zhu, Zhiliang Tian, Shangwen Wang, Weiguo Chen, Zixuan Dong, Mingyue Leng, and Xiaoguang Mao. Misum: Multi-modality heterogeneous code graph learning for multi-intent binary code summarization. *Proc. ACM Softw. Eng.*, 2(FSE), June 2025. doi: 10.1145/3715780. URL https://doi.org/10.1145/3715780.

8 Appendix

8.1 Additional Dataset Details

8.1.1 Dataset Filtering Details

Our initial set of binaries consisted of 373 Linux binaries and 403 Portable Executable (PE) or Windows binaries, selected for projects with the following licenses: MIT, Apache, GPL, AGPL, LGPL, or variants of the BSD license. After running extraction scripts to collect decompiled code, call graphs, and link the decompiled code and source code together in call graphs, we had a total of 393 annotated call graphs.

8.1.2 Function Category Statistics

Table 7 shows the counts of how many functions fall into each of the information categories describing whether or not the function had string literals or meaningful function identifies in them. We include statistics for the full dataset of binaries, and the two sets of 20 binaries selected for this study. The initial set has more samples in each category as they were sampled to cover them well compared to the replication set, which was randomly sampled. Note that like the results in the main paper these statistics are only for functions that have are missing names themselves (*e.g.*, have only automatically assigned names like `FUN_00102040`), filtering out library functions and internal functions retaining meaningful names. The difference between "Total" and "Total Evaluated" represents the number of functions that were successfully processed through all stages of the pipeline.

8.2 Graph Clustering Algorithm Details

We use simulated annealing (Kirkpatrick et al., 1983) to cluster with respect to modularity, a common metric for community detection comparing the density of connections within the community compared to outside of it (Newman & Girvan, 2004). Starting from an initial partition, the algorithm repeatedly proposes a local

Table 7: Break down of counts of function categories in broader dataset of licensed binaries and in the two sets of 20 binaries chosen for evaluation for unnamed functions.

Has String Literals	Named Functions	Full Dataset	Initial Set	Replication Set
No	All Calls Unnamed	11421	1665	732
Yes	All Calls Unnamed	5456	741	457
No	Some Calls Unnamed	6810	1029	594
Yes	Some Calls Unnamed	13808	1620	318
No	All Calls Named	4979	697	169
Yes	All Calls Named	4234	498	323
No	No Function Calls	8559	877	306
Yes	No Function Calls	1111	109	55
Total	—	56378	7236	2954
Total Evaluated	—	—	6884	2617

modification to the current clustering and evaluates the resulting change in modularity, denoted ΔQ. If $\Delta Q \geq 0$, the move is accepted. If $\Delta Q < 0$, the move is accepted with probability

$$P(\text{accept}) = \exp\left(\frac{\Delta Q}{T_t}\right),$$

where T is a temperature parameter controlling how likely the algorithm is to accept worse solutions. Since $\Delta Q < 0$ for a worsening move, this probability lies between $(0, 1)$ and decreases as either the loss in modularity increases or the temperature decreases. The temperature is updated with a geometric cooling schedule

$$T_{t+1} = \alpha T_t,$$

with initial temperature $T_0 = 0.1$ and cooling rate

$$\alpha = 0.5^{1/(2n)},$$

where n is the number of vertices. This choice halves the temperature every $2n$ iterations.

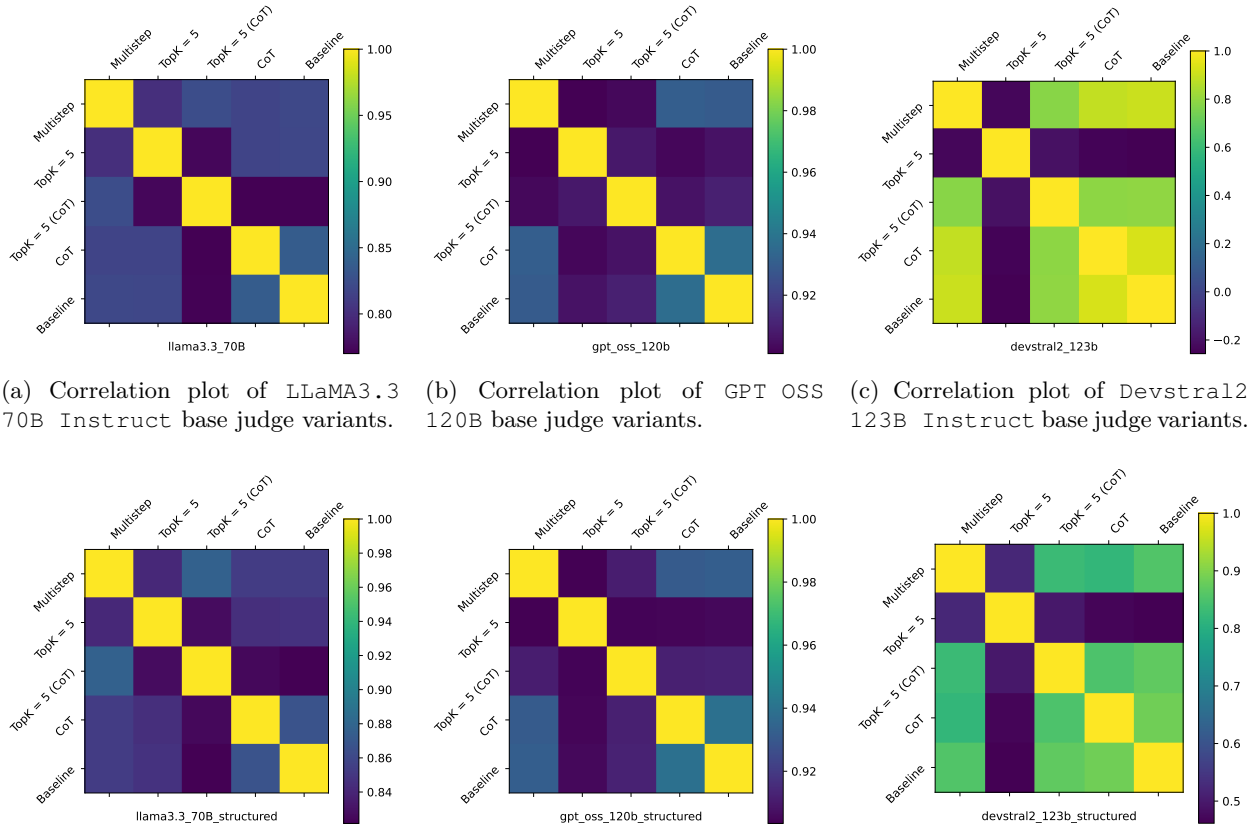
At each iteration, one of the following moves is proposed: 1) Move a randomly selected vertex to another community, possibly creating a new singleton community, 2) merge two communities, or 3) split a community randomly into two parts.

These moves are selected with probabilities 0.8, 0.1, and 0.1, respectively. In addition, for the single-vertex move, we use a greedy variant with probability 0.2: instead of choosing the destination community at random, we move the vertex to the community that gives the largest modularity improvement among feasible choices. This substantially improved runtime in practice. The parameters were set so that we could achieve good results on networks of various sizes while balancing run times. We did so by looking at how modularity changed over the number of iterations to balance exploration with convergence.

To enforce application-specific constraints, we discard proposed moves that violate the community size bound (at most 32,000 characters of decompiled code, except for singleton communities). This limitation was used as earlier models we tested with had more limited context sizes. The algorithm terminates if the modularity changes by less than $\text{tol} = 10^{-6}$ for $2n$ consecutive iterations.

To allow vertices to appear in multiple communities, we run the algorithm independently k times and treat the resulting k clusterings as multiple community assignments. In practice, the different runs often converge to distinct local optima, yielding limited overlap between repeated solutions.

8.3 LLM Judge UQ Study Correlation Plots



(a) Correlation plot of `LLaMA3.3 70B Instruct` base judge variants.

(b) Correlation plot of `GPT OSS 120B` base judge variants.

(c) Correlation plot of `Devstral2 123B Instruct` base judge variants.

(d) Correlation plot of `LLaMA3.3 70B Instruct` structured judge variants.

(e) Correlation plot of `GPT OSS 120B` structured judge variants.

(f) Correlation plot of `Devstral2 123B Instruct` structured judge variants.

Figure 5: Correlation plots for the LLM Judge ground truth UQ Study, showcasing consistency over variations in the requested output format.

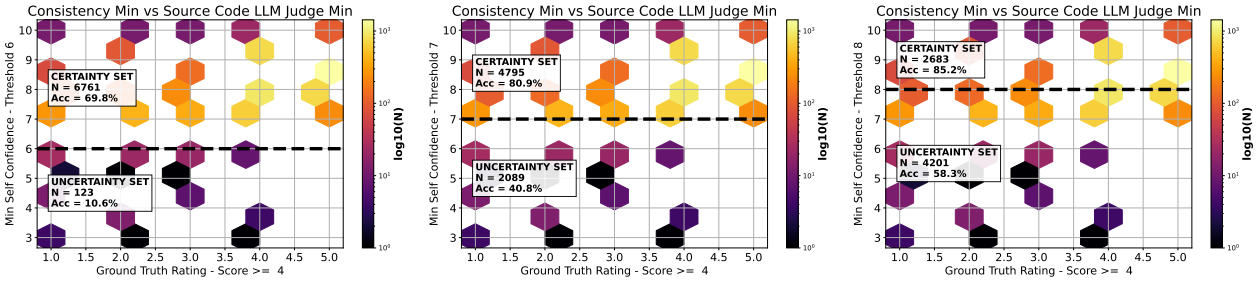
Figure 5 shows 6 correlation heatmap plots for our different LLM Judges. These correspond to the Kendall's W values presented in Table 2. The models other than *Devstral2 123B* show strong consistency in the rating under varying the output format and instructions, and that model specifically has the TopK variant without chain of thought becoming very different for both judge variants.

8.4 Example UQ Metric Visualization and Functions

8.4.1 Example Visualizations for UQ Metrics

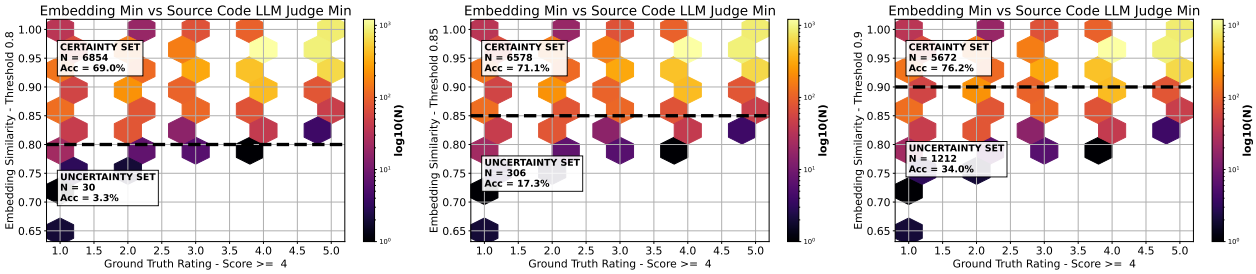
While we use PRR compare UQ metrics consistently across thresholds, visualizing the accuracy gain when removing high uncertainty samples is also useful. We can pick a threshold for a particular metric to divide the results into a *certainty set* and an *uncertainty set*. The certainty set contains samples with higher certainty, and the uncertainty set contains samples with low certainty. Metrics that increase accuracy in the certainty set efficiently while placing fewer samples into the uncertainty set are better (which is what PRR measures).

Figure 6 and Figure 7 show plots for self confidence and the embedding consistency metric respectively, varying the threshold to show accuracy and the sizes of the certainty and uncertainty sets. These examples are specifically for descriptions generated and judged with `GPT OSS 120B`, and use our initial set of top 20 binaries that were selected to cover different information categories. The hexbin plots measure the density of points, and an ideal UQ metric would have a diagonal line on this plot, aligning perfectly with the ground



(a) Self confidence with threshold 6. (b) Self confidence with threshold 7. (c) Self confidence with threshold 8.

Figure 6: Example hexbin plot with GPT OSS 120B judge using minimum self confidence as a threshold to divide functions into a certainty and uncertainty set, increasing the threshold left to right.



(a) Embedding Consistency with threshold 0.8. (b) Embedding Consistency with threshold 0.85. (c) Embedding Consistency with threshold 0.9.

Figure 7: Example hexbin plot with GPT OSS 120B judge using the description embedding cosine similarity thresholds to divide functions into a certainty and uncertainty set, increasing the threshold left to right.

truth LLM Judge scores. Comparing between the two thresholds, while both the self confidence scores and embedding consistencies for these models are able to separate and improve the accuracy in the consistency set, the embedding consistency score removes more of the worst samples effectively.

8.4.2 UQ on Example Functions

Here, we provide examples of UQ metrics being successful and unsuccessful in identifying sets of descriptions with poor generations. First, for a successful example, look at Figure 8. In this example, we see two generated descriptions of the same decompiled function with different input function cluster contexts. The embedding consistency metric in this case is low relative to the distribution (< 0.75), and we also see self-confidence scoring much worse on the description that is not close to the ground truth (describing the function as a socket connection instead of related to Lua parsing).

Figure 9 gives an example where the generated descriptions do not generate variance over their selected clusters in a way that identifies potential problems. While both decompiled descriptions provide a similar high level view of the algorithm, they both do not capture the higher level domain of bioinformatics that is clear from the source description. When samples are consistently inaccurate and models more confident in their generations, the UQ metrics are not able to identify problems in the generation.

8.5 Additional Consistency UQ Metrics

In addition to using clusters from the call graph to create variability in the prompts for the same function, we also explored using the relationships between functions to assess consistency. At a high level, the intuition is that functions that call each other are related, and that inconsistencies between their descriptions could be a sign of an incorrect description. We implemented two types of metrics, again one based on an LLM Judge and another on cosine embedding similarity.

```
---------------------------------- Decompiled Description 1 ----------------------------------
name: socket_send_request,
domain: networking,
...
description: Prepares and sends a network request using the socket object. It sets the
socket state, optionally allocates or reuses a request buffer, fills the request header,
writes the payload, updates internal counters, and triggers any pending event processing.
confidence: 5
---------------------------------- Decompiled Description 2 ----------------------------------
name: luaK_emit_numeric_for_loop,
domain: scripting / Lua bytecode generation,
...
description: Generates the byte-code for a numeric for-loop. It updates the parser state,
reserves registers, emits the loop-control instructions (initialization, step, limit, and
jump), and patches the jump offsets. The function handles two cases: a simple for-loop (
isLocal == 0) where it creates the loop variable and limit/step registers, and a more
complex case (isLocal != 0) where it uses pre-computed registers.
confidence: 8
---------------------------------- Ground Truth Description ----------------------------------
name: forbody,
domain: Lua compiler code generation / parsing,
...
description: Generates the bytecode for the body of a Lua for loop. It sets up the loop
block, adjusts local variables, emits the appropriate FORPREP/FORLOOP or TFORCALL/TFORLOOP
instructions depending on whether the loop is numeric or generic, patches jump targets, and
updates line information.
```

Figure 8: Example of 2 decompiled descriptions and their associated ground truth description where the minimum self confidence score and the embedding metric are low relative to their distributions, identifying problems in the set of generated descriptions. Input/output fields are omitted.

```
---------------------------------- Decompiled Description 1 ----------------------------------
name: lookup_linear_range_entry,
domain: data lookup / mapping,
...
description: Searches a variable-length table stored at offset 0x60 of the object pointed to
 by table_ptr. Each entry (16 bytes) contains a start index, an offset, a step and a length.
 The function finds the first entry where value_a lies within the start-plus-length interval
 and where value_b matches the linear expression offset + (value_a - start) * step. If such
an entry is found, the function returns the step value from that entry; otherwise it returns
 0.,
confidence: 8
---------------------------------- Decompiled Description 2 ----------------------------------
name: lookup_linear_mapping_step,
domain: data lookup / mapping,
...
description: 'Searches the array of mapping entries stored at offset 0x60 of the given
structure. Each entry defines a range (start, length) and a linear transformation (step and
offset). The function finds the first entry where the supplied value lies within the range
and where the target equals (value-start) * step + offset. If such an entry is found, it
returns the step value; otherwise it returns 0.
confidence: 9
---------------------------------- Ground Truth Description ----------------------------------
name: AreAlignedPairs,
domain: bioinformatics / protein structure analysis,
...
description: Determines whether two residue indices (res1, res2) belong to an aligned beta-
strand pair in the given PDBChain. If they are aligned, returns the pair's direction (
typically 1 or -1); otherwise returns 0.
```

Figure 9: Example of 2 decompiled descriptions and their associated ground truth description where self confidence scores and the embedding metric are higher, but the decompiled descriptions miss some key elements about the domain the function iso operating in. Input/output fields are omitted.

8.5.1 LLM Judge Neighbor Consistency Metric

For the LLM Judge metric, we asked an LLM to consider pairs of functions, where one was the calling and the other the callee function. The LLM was asked to assess if the relationship between the descriptions was plausible (i.e., does it make sense for function A to call function B), rating it as 1 for inconsistent, 2 for consistent, and 3 for unknown/ambiguous. We used two prompts, one where the Judge was provided just the generated descriptions, and another where they were provided the descriptions and the original decompiled code for the two functions. The text for these prompts can be found in Appendix 8.7.2.

An inconsistency score was generated for each node by summing the count of 1s for all the neighbors and normalizing by the total number of edges (*ambiguity allowed* in the tables) or by the total number of edges not rated as ambiguous (*no ambiguity*). To formalize this slightly more, if we treat the graph as undirected and define the following:

- v_i - a node in the graph
- $E_N(v_i)$ - the set of edges that contain v_i

With the LLM Judge being written more formally as:

$$\text{Score}(e_{ij}) = \begin{cases} 1, & \text{if LLM Judge rates neighboring nodes as consistent} \\ 2, & \text{if LLM Judge rates neighboring nodes as inconsistent} \\ 3, & \text{if LLM Judge can't decide if neighboring nodes are consistent} \end{cases} \tag{1}$$

Then the LLM Judge consistency metric that allows ambiguity defined as the following:

$$\frac{|Score(e_{ij}) = 1 | e_{ij} \in E_N(v_i)|}{|E_N(v_i)|} \tag{2}$$

Where the LLM Judge consistency metric without ambiguity is defined as:

$$\frac{|Score(e_{ij}) = 1 | e_{ij} \in E_N(v_i)|}{|Score(e_{ij}) \in \{1, 2\} | e_{ij} \in E_N(v_i)|} \tag{3}$$

8.5.2 Embedding Neighbor Consistency Metric

Our original intuition was that the domain field in a cluster of related functions would be similar, and we could create a metric for the cohesiveness of the cluster. The metric described below was fast to compute, so we extended to trying it with the function name and description fields as well from the structured JSON description. These fields were represented with embeddings from `intfloat/e5-mistral-7b-instruct`. Using the directed edges within the cluster of related functions, we construct a *cluster similarity metric* for each node in the cluster. Cosine similarities between embeddings were calculated between all nodes in the cluster up to N hops (we used N=2), and three variants of the metric were created with increasing complexity: 1) sum all similarities up to the cutoff, 2) adding a path splitting factor, reducing the contributions by the number of edges at a node, and 3) adding a decay factor to reduce the weight of similarities from further away nodes.

To more formally define this metric, we first provide a list of definitions:

- V_c – the set of nodes in a cluster.
- $Embed(v_i)$ – the embedding of some node v_i's feature in the graph. Could be the domain, name, or description of the function.
- $d_{ij} = \text{CosineSimilarity}\big(Embed(v_i), Embed(v_j)\big)$.

- $P_C(v_i, v_j, \ell)$ – a boolean function indicating that there exists a directed path between these two nodes in the cluster subgraph of length ℓ.
- L_c – the cutoff distance of how for how long of a path to explore in the cluster; we currently use 2.
- $C(v_i, L_c)$ – the count of all paths in the cluster of length $\leq L_c$ with node v_i as the starting or ending node.
- $\text{In}(v_i)$, $\text{Out}(v_i)$ – in degree and out degree of node v_i.

Equation 4 describes the simplest metric, which sums embedding similarities along paths within a cluster up to a cutoff. Equation 5 adds the path splitting factor for 1-hop distances to reduce the impact of nodes with large in degree and out degree. Equation 6 adds a decay factor (ℓ for the distance to the node) on top of that to represent the diminishing relationship between nodes that are further apart in the graph.

$$\text{Score}(v_i) = \frac{1}{C(v_i, L_c)} \sum_{v_j \in V_c} \sum_{\ell=1}^{L_c} \Big(P_C(v_i, v_j, \ell)\, d_{ij} + P_C(v_j, v_i, \ell)\, d_{ji} \Big) \tag{4}$$

$$\text{Score}(v_i) = \frac{1}{C(v_i, L_c)} \sum_{v_j \in V_c} \sum_{\ell=1}^{L_c} \left(\left(\begin{cases} \dfrac{P_C(v_i, v_j, 1)\, d_{ij}}{\text{Out}(v_i)}, & \ell = 1, \\ P_C(v_i, v_j, \ell)\, d_{ij}, & \ell > 1, \end{cases} \right) + \left(\begin{cases} \dfrac{P_C(v_j, v_i, 1)\, d_{ji}}{\text{In}(v_i)}, & \ell = 1, \\ P_C(v_j, v_i, \ell)\, d_{ji}, & \ell > 1, \end{cases} \right) \right) \tag{5}$$

$$\text{Score}(v_i) = \frac{1}{C(v_i, L_c)} \sum_{v_j \in V_c} \sum_{\ell=1}^{L_c} \left(\left(\begin{cases} \dfrac{P_C(v_i, v_j, 1)\, d_{ij}}{\text{Out}(v_i)}, & \ell = 1, \\ \dfrac{P_C(v_i, v_j, \ell)\, d_{ij}}{\ell}, & \ell > 1, \end{cases} \right) + \left(\begin{cases} \dfrac{P_C(v_j, v_i, 1)\, d_{ji}}{\text{In}(v_i)}, & \ell = 1, \\ \dfrac{P_C(v_j, v_i, \ell)\, d_{ji}}{\ell}, & \ell > 1 \end{cases} \right) \right) \tag{6}$$

8.6 Full Summary Tables with All Tested Metrics

Table 8 and Table 9 showcase the full set of UQ metrics we tested to separate highly rated scores (4 or higher) from lower rated scores (3 or lower). Results are formatted in the same manner as depicted in Section 6.2, where each column measures descriptions generated by one of the three LLMs we tested, and the PRR values are the mean across the 6 LLM Judges used to judge each. We can see that the metrics based on consistency with a function's neighbors described in Section 8.5. are inconsistent across LLMs, often being no better than random (or worse) for the `Llama3.3-70B-Instruct` descriptions.

8.7 LLM Prompt Templates

8.7.1 Ground Truth LLM Judge prompts

Figure 10 includes the text template of our LLM Judge prompt that compares only the description field of the generated outputs, and Figure 11 shows the prompt for the entire structured description. These are the rubrics sent to both the LLMs and the human raters for the formal inter-rater agreement study. The few shot examples are omitted from this appendix for space, but will be included in the replication package.

8.7.2 UQ Metric Prompts

Figure 12 shows the prompt used to have the LLM attempt to measure how consistent generated descriptions for the same function are with each other. The scale uses 1-10, similar to self-confidence, to enable a wider range of potential cutoff thresholds.

The prompts asking the LLM to measure whether neighboring nodes were consistent can be found in Figures 13, 14, and 15. Figure 13 and Figure 14 respectively show the prefix of the prompt when asking the LLM Judge to compare neighboring nodes with and without including the decompiled code in addition to the description. Figure 15 contains the rest of the prompt that both prompts shared after the differing prefix.

Table 8: Full PRR Scores for the different tested metrics on the 20 binaries sampled for information variety, including the path distances between different embeddings in the description (to try to capture cluster coherency), and LLM Judge scores trying to capture neighbor description consistency.

		Mean PRR	
Metric	GPT OSS 120B	Llama3.3-70B	Devstral2 123B
Self Confidence (Min)	0.449	0.299	0.436
Consistency Embedding Similarity	0.523	0.454	0.561
Consistency LLM Judge (Min)	0.286	0.388	0.474
Cluster Domain Sim. (Max)	0.013	-0.029	-0.043
Cluster Domain Sim. (Max - Path Split)	0.127	0.025	0.120
Cluster Domain Sim. (Max - Path Split/Decay)	0.090	0.010	0.087
Cluster Name Sim. (Max)	0.068	0.084	0.079
Cluster Name Sim. (Max - Path Split)	0.131	0.047	0.138
Cluster Name Sim. (Max - Path Split/Decay)	0.088	0.027	0.099
Cluster Description Sim. (Max)	0.040	0.013	0.073
Cluster Description Sim. (Max - Path Split)	0.131	0.031	0.136
Cluster Description Sim. (Max - Path Split/Decay)	0.088	0.014	0.099
Neighbor Consistency Judge (No Code - Ambiguity Allowed)	0.196	0.095	0.070
Neighbor Consistency Judge (No Code - No Ambiguity)	0.192	0.095	0.070
Neighbor Consistency Judge (Code - Ambiguity Allowed)	0.122	0.092	0.110
Neighbor Consistency Judge (Code - No Ambiguity)	0.122	0.092	0.110
Eigenvalue Laplacian (Max)	0.371	-0.128	0.331
Semantic Entropy (Max)	0.376	-0.086	0.412

Table 9: Full PRR Scores for the different tested metrics on the 20 binaries sampled randomly, including the path distances between different embeddings in the description (to try to capture cluster coherency), and LLM Judge scores trying to capture neighbor description consistency.

		Mean PRR	
Metric	GPT OSS 120B	Llama3.3-70B	Devstral2 123B
Self Confidence (Min)	0.450	0.277	0.384
Consistency Embedding Similarity	0.489	0.447	0.515
Consistency LLM Judge (Min)	0.323	0.453	0.443
Cluster Domain Sim. (Max)	0.126	-0.027	0.016
Cluster Domain Sim. (Max - Path Split)	0.196	0.007	0.187
Cluster Domain Sim. (Max - Path Split/Decay)	0.145	-0.031	0.163
Cluster Name Sim. (Max)	0.188	0.061	0.118
Cluster Name Sim. (Max - Path Split)	0.205	0.027	0.205
Cluster Name Sim. (Max - Path Split/Decay)	0.146	-0.017	0.175
Cluster Description Sim. (Max)	0.200	0.045	0.076
Cluster Description Sim. (Max - Path Split)	0.209	0.020	0.201
Cluster Description Sim. (Max - Path Split/Decay)	0.146	-0.019	0.171
Neighbor Consistency Judge (No Code - Ambiguity Allowed)	0.014	0.065	-0.044
Neighbor Consistency Judge (No Code - No Ambiguity)	0.013	0.065	-0.053
Neighbor Consistency Judge (Code - Ambiguity Allowed)	-0.048	0.002	-0.074
Neighbor Consistency Judge (Code - No Ambiguity)	-0.016	0.002	-0.074
Eigenvalue Laplacian (Max)	0.253	-0.192	0.121
Semantic Entropy (Max)	0.244	-0.163	0.162

```
You will be given a pair of summaries of the same code, source_summary and binary_summary.
 The source_summary is a highly accurate summary of the code created from the source code,
 and binary_summary is will be a description produced using information from a compiled
binary of this same code. Therefore, the binary_summary may have been constructed without
accurate variable names, function calls, or algorithmic/structural changes introduced in
the binary. Your task is to provide a 'total_rating' scoring how similar the
binary_summary is to the source_summary, juding if the descriptions describe similar
domains, algorithms, inputs/outputs. You must give a rating between 1 and 5, judged using
the following scale:

   1 means the binary_summary is nothing like the source_summary; they are completely
   unrelated, or the binary_summary is extremely vague.
   2 means the binary_summary captures some aspects of the source_summary, such as data
   structures used, but does not have the same domain as the source_summary and is vague.
   3 means the binary_summary is somewhat in agreement with the source_summary, at least
   correctly identifying a similar domain or general algorithm, but details will be vague.
   4 means the binary_summary is largely accurate to the source_summary, identifying the
   correct domain and algorithms, but there a some small disimilarities in names used or
   details of the description.
   5 means the binary_summary is almost exactly the same as the source_summary, showing
   practically identical understanding of the function.

Think through your score step by step and output the final result in JSON format:
{{"score" : <1 to 5 value>}}

Examples:

<Few shot examples>

End of Example.
```

Figure 10: LLM Judge prompt template on just the description field after it is extracted from the output. The few shot examples are omitted in this example.

```
You will be given a pair of summaries of the same code, source_summary and binary_summary,
 which are formatted as a structured JSON object.
The structured objects contain following fields:
- name - either the name of the function from source, or a suggested name from the
decompiled code.
- domain - a short classification of what domain this function is in
- inputs - names and types of the inputs, which are suggestions in the binary summary, and
 ground truth in the source summary.
- output_type - what type does this function return
- description - a summary of what the function does.
The source_summary is a highly accurate summary of the code created from the source code,
and binary_summary is a description produced using information from a compiled binary of
this same code. Therefore, the binary_summary may have been constructed without accurate
variable names, function calls, or algorithmic/structural changes introduced in the binary
. Your task is to provide a 'total_rating' scoring how similar the binary_summary is to
the source_summary, judging if the descriptions describe similar domains, algorithms,
inputs/outputs.
You must give your answer only as a singular value between 1 and 5, judged using the
following scale:
   1 means the binary_summary is nothing like the source_summary; they are completely
   unrelated, or the binary_summary is extremely vague.
   2 means the binary_summary captures some aspects of the source_summary, such as data
   structures used, but is otherwise vague or with limited matches in the structured
   fields.
   3 means the binary_summary is somewhat in agreement with the source_summary, at least
   correctly identifying a similar domain or general algorithm, but details will be vague.
   4 means the binary_summary is largely accurate to the source_summary, identifying the
   correct domain and algorithms, but there are some small dissimilarities in names used
   or details of the description.
   5 means the binary_summary is almost exactly the same as the source_summary, showing
   practically identical understanding of the function.

Think through your score step by step and output the final result in JSON format:
{{"score" : <1 to 5 value>}}

Examples:

<Few shot examples>

End of Example.
```

Figure 11: LLM Judge prompt template using structured scores. The few shot examples are omitted in this example.

```
You will be given a group of 2 or more summaries of the same binary decompiled code each
of which are formatted as a structured JSON object.
The structured objects will contain at least the following fields:
- name - either the name of the function from source, or a suggested name from the
decompiled code.
- domain - a short classification of what domain this function is in
- inputs - names and types of the inputs, which are suggestions in the binary summary, and
 ground truth in the source summary.
- output_type - what type does this function return
- description - a summary of what the function does.
Each summary is created from decompiled code in a binary with different contexts, which
could be other related functions, inputs from tools, or even human feedback. The summaries
 suggest names for the function and inputs, domains, and descriptions of the functions.
Your task it to evaluate how consistent the summaries are to each other. That is, do they
have similar domains or names, and are the descriptions closely aligned? The names,
domains, and descriptions do not need to be exact matches, if they are similar enough that
 is consistent. However, if descriptions are in the same domain but describe conflicting
tasks (for example, one description says a function reads a file and the other says it
write to a file), that would be an inconsistency.

Think through your judgement step by step, displaying the reasoning in less than 300 words
, and then provide a score between 1 and 10 highlighting how consistent this group of
descriptions is, with 1 representing a totally inconsistent set of metrics, and a 10
representing a set of descriptions that are essentially the same. For example, a rating of
 9 would indicate the names and domains are the same, with possibly some minor differences
 in the description logic. A
rating of 5 would indicate that the descriptions are roughly the same domain with logical
differences, or if there are 3 or more descriptions if only a subset of the descriptions
are consistent with each other. In constrast, a 2 would indicate only minor consistency -
for example if descriptions mention a similar data structure or input, but the names,
domains, and descriptions are otherwise inconsistent.

Your final response should be formatted in JSON to look like this:

{{"score" : "<Your score between 1-10 here>"}}
```

Figure 12: LLM Judge prompt to estimate consistency among generated descriptions for the same function.

```
You will be given 2 summaries functions generated from binary decompiled code, each of
which are formatted as a structured JSON object.
The summaries provided will be of two functions that are next to each other on the call
graph, where the caller function summary
is labeled as 'Parent Function Summary', and the callee function is labeled as 'Child
Function Summary'.
```

Figure 13: LLM Judge prompt to estimate consistency among generated descriptions for the same function.

```
You will be given 2 summaries functions generated from binary decompiled code, along with
the associated decompiled code for those specific functions. The summaries will be listed
as a structured JSON object, and the decompiled code will
be provided below each summary. The summaries provided will be of two functions that are
next to each other on the call graph, where the caller function summary/code is labeled as
 'Parent Function Summary and Code', and the callee function
is labeled as 'Child Function Summary and Code'.
```

Figure 14: LLM Judge prompt to estimate consistency among generated descriptions for the same function.

```
The structured objects will contain at least the following fields:
- name - either the name of the function from source, or a suggested name from the
decompiled code.
- domain - a short classification of what domain this function is in
- inputs - names and types of the inputs, which are suggestions in the binary summary, and
 ground truth in the source summary.
- output_type - what type does this function return
- description - a summary of what the function does.
Each summary is created from decompiled code with some additional context, which could
include other related functions, inputs from tools, or even human feedback. The summaries
suggest names for the function and inputs, domains, and descriptions of the functions.

Your task is to evaluate if the two descriptions are consistent with a caller-callee
context. For example, a function that takes 1 argument to extract information from human
readable text might plausibly call a string processing function. However, it would not
likely invoke image processing functions. Your goal will be to label the pair of summaries
 with 1 of 3 numerical categories.

   1 - The two descriptions seem inconsistent with a possible caller-callee relationship.
   2 - The parent description describes a domain and task that could plausibly invoke a
   function with the child's description.
   3 - The descriptions do not provide enough information to decide if they are consistent
    or not, or if you are otherwise unsure.

Think through your judgement step by step, displaying the reasoning in less than 300 words
, and then format the output in JSON to provide a 1 or 2 sentence justification for the
score, and then the score of 1,2, or 3.

Your final response should be formatted in JSON to look like this:

'''json
{{
   "justification" : "<1-2 sentences for why you scored this as 1, 2, or 3>",
   "score" : "<Your score between 1-3 here>"
}}
'''
```

Figure 15: LLM Judge prompt to estimate consistency among generated descriptions for the same function.