Investigating LLMs for Feature Presence Prediction in Software Products*

Guillaume Nguyen[1][0000−0002−9724−6634], Paolo Arcaini[2][0000−0002−6253−4062], Maxime Cordy[3][0000−0001−8312−1358], Xavier Devroey[1][0000−0002−0831−7606], and Fuyuki Ishikawa[2][0000−0001−7725−2618]

[1] NADI, University of Namur, Belgium
[2] National Institute of Informatics, Tokyo, Japan
[3] SnT, University of Luxembourg, Luxembourg

Abstract. Assessing legacy systems for compliance with new requirements is critical but hindered by obsolete documentation and the loss of expert knowledge. While Software Product Line (SPL) research excels at feature model reconstruction, identifying specific features within a single legacy product for re-evaluation remains a persistent challenge and has received comparatively less attention than full feature model reconstruction. In this paper, we evaluate the ability of five Large Language Models (LLMs) to automate feature retrieval from legacy code. We propose an empirical framework using SPL benchmarks across 248 product variants to enable a controlled quantitative evaluation. Our study investigates the impact of prompting strategies, including few-shot configuration, feature name obfuscation, and source code granularity (individual function definition vs. complete code). Our results demonstrate that while LLMs can identify features with reasonable accuracy, their performance is highly sensitive to lexical cues and prompting design. We highlight a fundamental Precision-Recall trade-off and provide empirical insights into the conditions and challenges of using LLMs for legacy system re-assessment.

Keywords: Feature Identification, Software Evolution, Re-Evaluation, LLM, Compliance

1 Introduction

Many organizations rely on long-lived software systems whose original documentation, design rationale, and traceability links have progressively disappeared, while the source code remains available. When new regulatory constraints, internal policies, or operational requirements emerge, these systems must be re-

evaluated to determine whether they still provide the required functional capabilities or they need to be modified to guarantee them. We define *re-evaluation* as the evaluation of a product that was deemed compliant with a set of requirements at a specific point in time, with an evolved set of requirements (i.e., newer) at a more recent point in time. Performing such an assessment manually requires deep technical expertise and detailed program comprehension, making the process costly and difficult to scale, especially when multiple products of the same system must be analysed.

For instance, consider a legacy email system developed several years ago. Over time, new security requirements may have emerged, such as the need to support message encryption. However, documentation may be incomplete, and the original developers may be unavailable. Engineers must therefore re-evaluate the software to see whether the current implementation already supports encryption-related functionality or whether a new functionality needs to be introduced. In such situations, automatically identifying which features are implemented in the system becomes a key step in re-evaluating its compliance with evolving requirements.

From a software engineering perspective, this problem is closely related to feature location, which aims at retrieving implementation artifacts corresponding to a textual or behavioural description [4, 5, 13], as well as to requirements-code traceability and variability-aware analyses in Software Product Lines (SPLs), where functionality may be implemented differently across products [10, 11, 19, 21]. However, these approaches typically assume the availability of execution scenarios, handcrafted heuristics, explicit variability models, predefined compliance indicators, or existing traceability links [2,3,16,17]. Such assumptions rarely hold in re-evaluation scenarios, where the system must be assessed directly from its implementation and where the question is not to recover links between artifacts, but to determine whether a given functional capability is present.

Recent advances in Large Language Models (LLMs) have shown promising results for program comprehension tasks such as code summarization and feature-oriented commenting [1, 7, 14, 18]. These models are able to capture high-level semantic properties of source code and to adapt their output to different information needs expressed in natural language. This suggests that LLMs could be used as semantic query engines over source code, taking as input a functional description and determining whether the corresponding behaviour is implemented. However, current empirical studies on LLM-assisted program analysis are often conducted on relatively small benchmarks, typically evaluating a single model and without considering configurable systems or multiple product variants [8]. More importantly, many of these studies lack a well-defined ground truth regarding the functional capabilities implemented in the analysed systems, which makes it difficult to perform precise quantitative evaluations. SPLs offer a particularly suitable setting for such analyses, as the set of features implemented in each product variant is known by design through the configuration process. Therefore, leveraging SPL benchmarks enables a more systematic assessment of LLM behaviour when identifying implemented features, and allows the im-

pact of different prompting strategies and model choices to be evaluated under controlled conditions.

To investigate the capabilities of LLMs in identifying specific functionalities within re-evaluated products, we formulate the following research questions:

RQ1: To what extent does the content of the prompt influence the performance of LLMs in feature extraction?
- **RQ1a:** What is the impact of the prompting strategy (e.g., `Zero-Shot` vs. `Few-Shots`) on feature extraction accuracy?
- **RQ1b:** How does the obfuscation of feature names affect the ability of the LLMs to identify implemented features?
- **RQ1c:** To what extent does the source code granularity (e.g., individual function definitions vs. complete code) influence the results?

RQ2: How does LLMs performance vary across different SPL benchmarks?
RQ3: What performance differences can be observed across the evaluated LLMs?

To answer these questions, we conduct a large-scale empirical study in which five LLMs of different sizes are used to retrieve features from **248** products belonging to six SPLs in SPL2go [9], totalling to **800** features to be found without looking at the precise location (i.e., the unique set of predicted features overlapping with the ground truth) and **1660** features linked to specific function definitions. By using SPLs, we ensure that the ground-truth (the set of implemented features and their precise location) is directly tied with each product. We compare different prompt templates (shot strategy, code granularity, and feature name obfuscation) and evaluate the models using Precision and Recall. We do not analyse our results using a single F1-score, as we want to capture the specific effects of the prompt content; this would not be possible with the F1-score, as a prompt with better Recall yet worse Precision than another one could have the same F1-score.

In real re-evaluation scenarios, feature lists may originate from regulations, documentation, or product specifications. For this experiment, we assume that only the code is known, while the feature names and descriptions are provided by the user. We use SPL benchmarks as a controlled experimental setting because they provide an explicit and reliable ground truth regarding feature presence and feature-to-code mappings, enabling an objective evaluation of LLM behaviour.

To summarize, the contributions of this paper are as follows:

- an empirical study of multiple open-source, various-sized LLMs for capability detection in source code across entire configurable systems;
- a comparison of prompting strategies and provided content for feature retrieval and feature localization;
- a fully reproducible experimental approach for both data collection and analysis.

The replication package containing the scripts, raw data, and statistical analysis is available at `https://anonymous.4open.science/r/mercury-ED82/`.

Paper structure Sect. 2 introduces the related work. Sect. 3 presents the design of the study, while Sect. 4 discusses the experimental results. Finally, Sect. 5 provides a critical discussion of the results, and Sect. 6 concludes the paper.

2 Related Work

Concept and Feature Location in Source Code. The task of identifying where a functionality is implemented in a software system has been extensively studied under the terms *concept location*, *concern location*, and *feature location*. These approaches retrieve implementation artifacts from a textual or behavioural description.

Eisenbarth et al. [5] combine static and dynamic analyses to relate execution traces to source code elements, showing that runtime information significantly improves localization accuracy. Information-retrieval-based techniques later enabled purely textual queries: Marcus et al. [13] use Latent Semantic Indexing to capture semantic similarity between domain terms and source code, and retrieve candidate fragments. Dit et al. [4] show that most techniques rely on a combination of static analysis, dynamic analysis, and textual similarity, and that no single source of information is sufficient to consistently identify features in large systems. More recently, Fortz et al. [6] leverage machine learning techniques to analyse behavioural logs and identify variants in variability-intensive systems. Their work demonstrates the potential of learning-based approaches to detect variability-related behaviour across executions.

However, these approaches typically rely on handcrafted heuristics, execution traces, or traditional information-retrieval models, and are often evaluated on individual systems or datasets without an explicit ground truth of features embedded by design in the implementation.

Variability, Compliance, and Traceability. Recovering variability from implementation artifacts is essential for SPL re-engineering and evolution. In this context, compliance checking ensures that code satisfies regulatory, standard, or other requirements through predefined indicators [3] or static analysis of security patterns [16, 17]. While these methods focus on identifying specific signals, requirements-to-code traceability techniques use Information Retrieval techniques or LLMs to bridge the gap between natural language and software artifacts [2, 10, 11, 21].

Our work departs from these by focusing on capability detection. Instead of recovering links or detecting rigid patterns, we leverage semantic queries to determine if a high-level feature is implemented within a function definition. This approach is specifically designed for scenarios where documentation is missing, allowing for a direct functional assessment of the system from its source code.

LLMs for Configurable Systems and Reverse-Engineering. LLMs are increasingly used for program comprehension, including code summarization, search, and variability-related tasks [1, 7]. In configurable systems, they assist in

performance-oriented configuration [20] and semantic quality assessment of code descriptions [14]. Recent studies have also explored the generative potential of LLMs to reverse-engineer user stories directly from source code [18], suggesting that these models can bridge the gap between low-level implementation and functional requirements. However, despite their ability to capture multiple semantic views of code, these models remain prone to hallucinations and are highly sensitive to prompt design [8]. Furthermore, existing studies often rely on small benchmarks, lacking systematic evaluation across multiple products and models.

In parallel, traditional feature location in SPLs exploits version histories [15], variability-aware analyses [19], or clone-and-own recovery techniques [12] to map features to scattered artifacts. While these methods rely on explicit variability mechanisms or structural analyses, we investigate whether LLMs can directly detect functional capabilities across product variants through semantic queries. Our approach bypasses the need for specialized variability-aware tools, focusing instead on the model's ability to identify high-level features directly from function definitions.

Positioning of our work. In software engineering, retrieving implementation artifacts corresponding to a functional description has been addressed through feature location, traceability, and variability-aware analyses. In parallel, LLMs have shown promising results for program comprehension tasks, yet their evaluation remains limited in scale and rarely considers configurable systems.

In this work, we study whether LLMs can be used as semantic query engines to determine the presence of functional capabilities in source code. To do so, we use SPLs as a way to control which features are in a product.

3 Approach

The complete project with example prompts is available on Github[4]. As shown in Fig. 1, we have three major steps in our approach. First, we produce variants from six SPLs in SPL2go. Specifically, we chose the *Elevator*, *MinePump*, and *EmailSystem* SPLs in `C` and `Java` programming languages. The six SPLs with a total of 38 features are presented in Table 1, with the name of the SPL that was analysed, the programming language, the number of features, the number of products, the mean number of changes induced by each feature (added function definition), and the minimum and maximum number of lines of code (LOC). The definitions of the features for each SPL are presented in Table 2.

The experimental methodology follows a structured three-phase pipeline to evaluate the performances of each LLM within the context of SPL. The process begins with the **A. Produce Variants** phase, where a diverse set of SPLs is retrieved from the `SPL2go` website. By systematically processing the associated constraints, the pipeline generates the complete set of valid configurations (i.e., *products*), ensuring that every tested product strictly adheres to the logical dependencies of the feature model. Then, the principal source code for each

[4] `https://github.com/gunguyen/LLM_as_code_query_engines-Variability26`

Table 1: SPLs from SPL2go used in the experiments

SPL	Language	# features	# products	# changes	LOC (min-max)
Elevator	C	5	20	6.4	398 - 470
Elevator	Java	5	20	6	268 - 354
EmailSystem	C	8	40	7.2	92 - 260
EmailSystem	Java	8	40	16.2	143 - 476
MinePump	C	6	64	4.1	58 - 105
MinePump	Java	6	64	4.1	71 - 138

product is loaded into a specialized object list. This is followed by an automated extraction of Abstract Syntax Tree (AST) elements, which serve as the granular units for the subsequent extraction tasks. We only consider Function Definitions (in `C`) and Method Declarations (in `Java`) from those extracted AST elements; we collectively refer to those elements as "Function Definitions". To bridge the gap between source code and feature-oriented composition, we leverage the `FeatureHouse` toolchain. This specifically addresses the complexities of feature refinement and method overloading. The pipeline reconstructs precise `function-definition__wrappee__previousFeature` identifiers to accurately map overloaded functions to their respective features.

This structured approach culminates in the generation of two distinct *Ground Truth* datasets: one providing a macroscopic view of product-level accuracy and another one offering a fine-grained analysis of specific function definitions.

The **B. Prompt Engineering** phase involves executing LLM experiments across multiple prompting strategies, where the resulting model outputs are stored into `JSON` files for consistent downstream processing and troubleshooting. We provide the LLMs with the closed-set of features of each SPL to facilitate the evaluation. In a real re-evaluation scenario, the set of names and descriptions

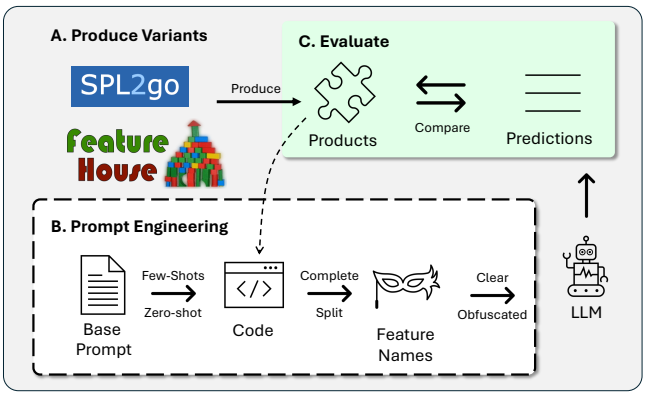


Fig. 1: The overview of the approach of this paper.

Table 2: Feature Specifications for Elevator, MinePump, and EmailSystem SPLs

Feature	Specification
Elevator SPL	
`weight`	If the elevator is empty now, reset the in-lift-buttons.
`twothirdsfull`	Car calls have precedence when the Lift is two thirds full.
`empty`	The Lift will not arrive empty at a floor and open its doors unless the button on that landing was pressed. The Lift will honour Requests from within the lift as long as it is not empty.
`overloaded`	The doors are never closed when the lift is overloaded. Elevator must not move while it is overloaded.
`executivefloor`	While there is a request from the executive Floor, the lift will not open its doors somewhere else.
MinePump SPL	
`highWaterSensor`	Sensors that detect when the water level is too high.
`lowWaterSensor`	Sensors that detect when the water level is too low.
`methaneQuery`	It queries the methane sensor which returns the percentage of methane in the air.
`methaneAlarm`	It triggers an alarm when the percentage of methane in the air is too high.
`stopCommand`	It allows to manually or automatically stop the pump.
`startCommand`	It allows to manually or automatically start the pump.
EmailSystem SPL	
`verify`	It verifies the authenticity of the message by confronting the mail signature with the public key of the sender.
`autoresponder`	It sends an automatic client-specific response when receiving a mail from a client.
`addressbook`	It checks if there are other known addresses for the client and also sends the mail to that address when a mail is sent.
`keys`	It checks if an asymmetric key pair is valid.
`encrypt`	It encrypts a mail using the public key of the receiver.
`sign`	It sign automatically a mail using a private key when it is sent to a client.
`forward`	It automatically forwards a mail when it is received.
`decrypt`	It automatically decrypts a mail using a private key when it is received and encrypted.

of the evaluated features would be provided by the user. The *base* prompt is a framework prompt as shown in Fig. 2 on which we applied the following code contents modifications:

- `Few-Shots` (FS) or `Zero-Shot` (ZS): In `Few-Shots`, we add three examples from an unrelated SPL to avoid bias (*Paycard*, also available from SPL2go). This number represents a design trade-off between providing representative demonstrations and preserving context space. The examples include code snippets, a list of features, and the expected response from the LLM in `JSON` format.
- `Complete-Code` or `Split-Code`: We send the whole code with the prompt or we send each function definition in separate prompts. For `Complete-Code`, we annotate it with all line numbers, while, for `Split-Code`, the location of the feature is embedded in the prompt (and the response from the LLM).
- `Clear` or `Obfuscated`: We either keep or obfuscate feature names to simulate different levels of domain knowledge. Obfuscation only affects the feature names and descriptions in the prompt; the source code remains unchanged.

```
=== TASK ===
You are a feature extraction assistant. You help a developer to analyze the given code
and extract its features. You provide a developer with a list of features supported in
the code

- Cycle through the list of **FEATURES** provided below.
- Generate a JSON object for each individual feature found in the **CODE**.
- Only include features that are clearly supported by the code.

Follow these strict and simplified rules:

1. "feature" field
   - Must be in the provided list of features.
   - If a feature is implemented in multiple methods add it each time.
   - Only include features that are clearly supported by the code.

2. "source" field
   - Use exact line references from the code, using line numbers provided at the
   beginning of the line for example "15:".
   - Only use the line at which a method including the feature starts
   - Example: ‘"14,22,45"‘

3. "description" field
   - Provide a concise, technical explanation of what the feature does in the code.


Generate one JSON object **per feature**.

Respond only with the list of JSON objects following the **structure** below. Do not
include comments or prose outside the JSON.
‘‘‘
[
  {
    "feature": "feature_in_list",
    "source": "14,22,45",
    "description": "This is an example of a description"
  },
  {...}
]
‘‘‘

=== EXAMPLES ===
*

=== FEATURES ===
| Feature | Description |
|---------|-------------|
...
| feature 3 | Check if an asymmetric key pair is valid |
| feature 4 | Encrypt a mail using the public key of the receiver |
| feature 5 | Sign automatically a mail using a private key when it is sent to a client  |
| feature 6 | Forward automatically a mail when it is received |
...

=== CODE ===
*Full prompts with examples, features and code are on the github
```

Fig. 2: Prompt template for the extraction of features from source code.

This setup tests whether models can associate feature descriptions with code without explicit feature names (see Fig. 2).

Finally, the **C. Evaluate** phase aggregates these results into configuration-specific tables, categorized by prompt type, code content, and obfuscation lev-

Table 3: Summary of LLMs used with Hugging Face inference service

Provider	Manufacturer	Model Name	Given name	Model Version
novita	OpenAI	Gpt-oss-20b[a]	gpt-20b	2025-08-04
novita	OpenAI	Gpt-oss-120b[b]	gpt-120b	2025-08-04
novita	Qwen	Qwen3-32B[c]	Qwen3-32b	2025-04-28
novita	Qwen	Qwen3-Coder-480B-A35B-Instruct[d]	Qwen3-480b	2025-07-22
novita	Meta	Llama-3.3-70B-Instruct[e]	Llama3-70b	2024-06-12

[a] `https://huggingface.co/openai/gpt-oss-20b` [b] `https://huggingface.co/openai/gpt-oss-120b`
[c] `https://huggingface.co/Qwen/Qwen3-32B` [d] `https://huggingface.co/Qwen/Qwen3-Coder-480B-A35B-Instruct`
[e] `https://huggingface.co/meta-llama/Llama-3.3-70B-Instruct`

els as well as LLM, SPL, and programming language. Then, we compare the prediction of the LLMs and the related ground truth from the corresponding configurations.

The previous configuration results in eight distinct prompting strategies for each product, applied to each model. Note that for `Split-Code`, when taking the location into account, we expect the LLMs to predict a feature in each function definition that was added by FeatureHouse when building the product. We chose a set of five open-source, various sized LLMs available through the HuggingFace inference service.[5] They are listed in Table 3 alongside their given name, links, and version dates.

In summary, we evaluate six SPLs (three benchmarks implemented in two programming languages), eight prompting strategies, and five LLMs, resulting in 240 experimental runs per product (6 SPLs $\times$ 8 prompting strategies $\times$ 5 LLMs). Depending on the evaluation setting, predictions are assessed either at the product level or at the function-definition level. In the latter case, each function definition is processed independently under the `Split-Code` strategy, resulting in a substantially larger number of observations.

4 Evaluation

In this section, we address our research questions through a statistical analysis of the collected data. We consider two complementary evaluation settings: (i) feature presence at the product level, where only the set of predicted features is evaluated, and (ii) feature presence at the function-definition level, where models must additionally associate features with their implementation locations. Consequently, a single feature may generate multiple observations when implemented across several function definitions.

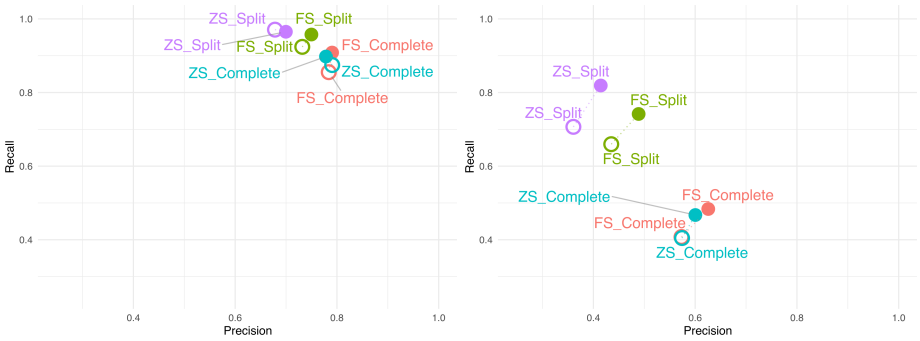
Our dataset comprises $19,839$ observations for `Complete-Code` predictions, increasing to $25,465$ when mapping predicted line numbers to function definitions. For `Split-Code`, where each function definition is processed independently, we collected $57,920$ observations. An observation corresponds to a single feature prediction produced by an LLM for a given product or function definition.

[5] `https://huggingface.co/`

4.1 RQ1: Effect of the Prompt Content

Fig. 3a illustrates the performance of the various strategies when we do not take into account the precise location of a feature in the code and Fig. 3b illustrates the performances when we do it. On the X axis, we project the Precision of the experiment (correctly predicted features) and, on the Y axis, we project the Recall of the experiment (ratio of expected features found by the models).

Considering results where the names of the features are not obfuscated (`Clear`), when excluding the location from the evaluation (Fig. 3a), we observe a Recall as high as 97% for the `Zero-Shot_Split-Code` strategy, while the highest Precision (79%) is achieved by the `Few-Shots_Complete-Code` combination. However, incorporating the predicted feature's location (Fig. 3b) leads to a performance drop across all strategies. In this context, the best Recall (82%) is reached by the `Zero-Shot_Split-Code` strategy, and the highest Precision (63%) by the `Few-Shots_Complete-Code` strategy. Obfuscating the names of the features (`Obfuscated`) further degrades both Precision and Recall.



(a) Without taking the precise location of a feature in the code

(b) Taking the precise location of a feature in the code

Fig. 3: RQ1 – Performances of prompting strategies. Full dots indicate results with non-obfuscated feature names (`Clear`), while empty dots indicate results with the obfuscated feature names (`Obfuscated`).

A notable observation occurs with the `Split-Code` strategy when location is ignored (Fig. 3a): Recall appears to increase compared to `Complete-Code` strategy, but this was expected as it is primarily due to a higher volume of predictions, which also increases the number of False Positives. Overall, these results highlight a clear trade-off: the combination `Zero-Shot_Split-Code` significantly boosts Recall, whereas the combination `Zero-Shot_Complete-Code` is more effective for maximizing Precision. This is particularly interesting as the `Few-Shots_Complete-Code` leveraged similar yet inferior results, while we would have expected superior results (higher-quality context).

RQ1a – `Few-Shots vs Zero-Shot`

The `Few-Shots` strategy outperforms `Zero-Shot` with a significant Precision increase of 2.6%, rising to 4.2% when taking location into account (paired t-test, $p < 0.001$). The `Zero-Shot` strategy significantly increases the Recall of 1.9%, rising to 2.1% when taking location into account (paired t-test, $p < 0.001$). Although significant, the observed effect size remains relatively small in practical terms.

RQ1b – `Clear vs Obfuscated`

Concerning the obfuscation of feature names, the paired t-test showed a negative impact on both Precision and Recall (paired t-test, $p < 0.001$). This leads to the conclusion that LLMs might leverage information from word matching in addition to the semantics of the presented code.

RQ1c – `Complete-Code vs Split-Code`

Providing `Complete-Code` significantly increases the Precision of 7.4% rising to 18% when taking the location into account (paired t-test, $p < 0.001$). Instead, providing `Split-Code` significantly increases the Recall of 7.6%, rising to 27% when taking the location into account (paired t-test, $p < 0.001$).

4.2 RQ2: Performance between benchmarks (SPL)

The various ways we crafted the definitions of the set of features from the three SPLs might have an impact on the ability of the LLM to link a description of a feature to a specific code. Hence, it could embed hidden effects on the performances. In Fig. 4, we can observe that there are indeed differences among the SPLs: while `C` and `Java` versions of *MinePump* and *Elevator* are quite close (although not identical), the *EmailSystem* SPL demonstrates great differences. When looking at the column Changes in Table 1 based on features related to specific function definitions, we can see that *EmailSystem* in `Java` has more than twice the amount of features to predict in specific function definitions than *EmailSystem* in `C`. This is related to the fact that the `Java` SPL of *EmailSystem* splits features into more function definitions than its `C` counterpart. This obviously negatively impacts the Recall performances of the models. Interestingly, we would have expected the Precision to rise proportionally (more chances to predict a feature in the correct function definition) but it seems that those added function definitions were difficult to link with expected features. Furthermore, the definitions of the features for this SPL were based on the `C` implementation of the features. Concerning the other differences between the three SPLs, there is probably an effect of the definition of the features as well as the readability of the code. Those effects need to be further analysed.

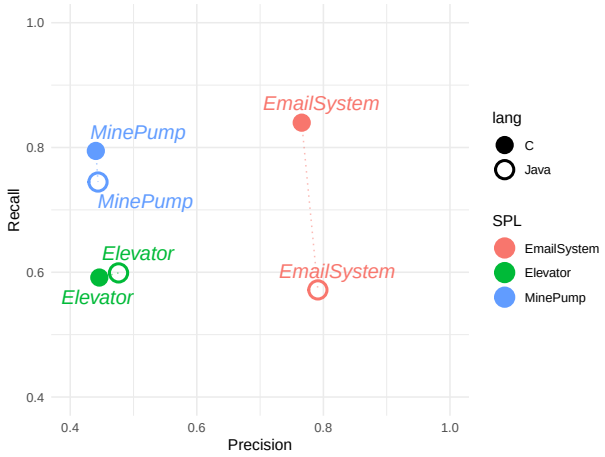


Fig. 4: RQ2 – Performances of the models between `C` and `Java` SPLs.

> **RQ2** – `C` vs `Java`
>
> We performed a paired t-test as well to assess the validity of the results. Although we mostly have significant results ($p < 0.01$), they remain hard to interpret: for Precision, `C` is better predicted than for `Java` when we take the location into account, but `Java` is better predicted when we do not. For Recall, we only have significant results when taking location into account, which shows higher scores when analysing `C` SPLs ($p < 0.001$).

On one hand, the programming language factor might embed a combination of other factors such as the number of lines of code (LOC) provided. In order to reduce this effect, we extracted the number of lines of code for each product when providing the `Complete-Code` to the LLMs and computed a linear regression model on both Precision and Recall. We took the following factors: the LOC, the language, `Zero-Shot` vs `Few-Shots`, obfuscation of the feature names and the number of expected features (ground truth). For Recall, the model explains 11.95% of the score showing significantly better scores for `C` SPLs. For Precision, the model explains 16.82% of the score showing significantly better scores for `C` SPLs as well. Thus, overall, `C` is better understood by the LLMs we used, while other factors might be at play and a deeper analysis of specific experiments is required. A more complete statistical model is presented in Sect. 4.4.

On the other hand, the description of the features and the related code could also impact the ability of the LLMs to correctly predict a feature. To assess the potential effect of the relation between a function definition and a feature description, we computed the SIDE-score [14]. This metric evaluates the semantic quality of code summaries by measuring the embedding similarity between code and textual descriptions, leveraging a fine-tuned S-BERT model to project both into a shared vector space. By computing the correlation between the Re-

```
void activatePump() {
    if (!isMethaneAlarm()) {
        original();
    } else {
        //System.out.println("Pump not activated due to methane
            alarm");
    }
}
```

Fig. 5: Implementation of `activePump`

call of the features (i.e., predictability) and the SIDE-score between expected features and their corresponding function definitions, we observed a moderate positive correlation of 33.55% with high significance ($p < 0.001$). While the SIDE-score may not be a perfect proxy for all aspects of code summarization, these results suggest a clear trend: the model's performance improves when the feature's definition closely matches the functional implementation within the function definition.

By investigating further the features that were harder to predict and the features that were predicted instead, the top 20 is almost completely composed of the *MinePump* SPL (the others are 4 for the *EmailSystem* SPL and 2 for the *Elevator* SPL). The biggest issue was for the `activePump` function definition where the `methaneQuery` was mistaken for the `startCommand` (for both `C` and `Java` SPLs).

When looking at the specific code in Fig. 5 and both definitions of `methaneQuery` and `startCommand` in Table 2, we can understand the confusion from the LLMs. Indeed, the code in Fig. 5 better suits the definition of `startCommand` than `methaneQuery`. Furthermore, `activate` is lexically close to `start` which might also influence the prediction of the LLMs.

We manually looked at the top-20 features that were not correctly predicted (i.e., at the expected function definition), we discovered two main reasons: (i) either the LLMs predicted too many features because there was a combination of semantic and lexical cues, or (ii) the predicted feature was in the name of the base function call after an overload from *FeatureHouse* (e.g., `processEnvironment__wrappee__methanequery()`) so, a purely lexical cue with no link between the definition of the predicted feature and the presented code. Nevertheless, in cases where a combination between semantic and lexical cues were at play and we would agree with the prediction of the LLMs, there is a not negligible chance that it was mainly due to lexical cues.

4.3 RQ3: Performance between models

In Fig. 6, we plot the performance in terms of Precision and Recall for each LLM used when taking the location of the feature into account or not. It reveals Recall scores as high as 95% for *Llama3-70b* and Precision scores as high as 78.9% for *gpt-20b* when we do not take the location into account. Actual performances

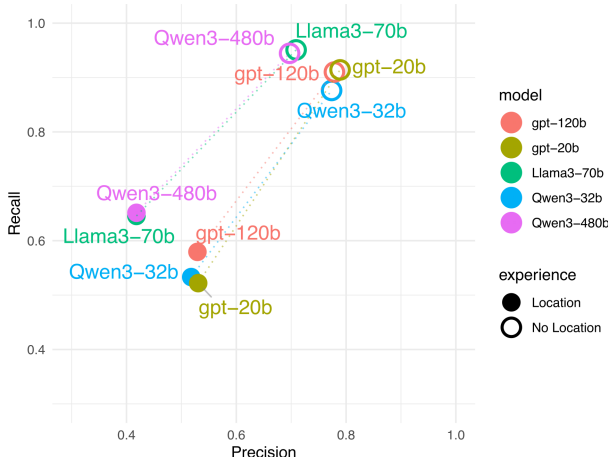


Fig. 6: RQ3 – Comparison of the performances of the models when taking or not the location of the predicted feature into account.

drop when taking the location into account with Recall scores as high as 65.1% for *Qwen3-480b* and Precision scores as high as 53% for *gpt-20b*. The fact that a smaller open-source model like *gpt-20b* performed quite well compared to bigger models is quite a surprise. Furthermore, we can observe two types of groups (more Precision vs more Recall) indicating that the choice of model has an impact on the performances. Thus, this should be taken into account when designing an approach (Precision-oriented or Recall-oriented) to assess the presence of features from source code.

> **RQ3 – Performances of the LLMs**
>
> We used a linear regression model to compare the LLMs for both Precision and Recall and it confirms the results presented in Fig. 6. Indeed, *gpt-20b*, *gpt-120b* and *Qwen3-32b* performed positively (negatively) and significantly ($p < 0.05$) in terms of Precision (Recall) than *Llama3-70b*. *Qwen3-480b* did not significantly differ from the other LLMs considering $p < 0.05$.

4.4 Overall analysis

The Generalized Linear Mixed-Effects Model (GLMM), using the six SPLs as random effects (Fig. 7), confirms the observations from the previous research questions while accounting for the variability across benchmarks. The dots on the left of the red dashed line indicate a negative effect on the specified score (Precision or Recall), whereas the ones on the right indicate a positive effect. Significance levels are shown above the dots and 95% confidence intervals are represented by the horizontal bars.

The GLMM confirms the fundamental Precision-Recall trade-off identified in RQ1. The `Split-Code` strategy is the strongest positive driver for Recall ($\beta =$

1.03, $p < 0.001$), while negatively impacting Precision ($\beta = -0.44$, $p < 0.001$). It also confirms that identifying the location of a feature is substantially more challenging than only predicting its presence.

Regarding RQ2, the GLMM isolates a significant Recall penalty for `Java` ($\beta = -0.76$, $p < 0.001$), while the effect on Precision remains negligible ($\beta = +0.06$). Likewise, non-obfuscated feature names (`Clear`) benefit both Recall ($\beta = +0.39$) and Precision ($\beta = +0.15$), confirming the importance of lexical cues during feature identification.

Finally, the GLMM confirms the model-specific behaviours observed in RQ3. Relative to *Llama3-70b*, both *gpt* and *Qwen3-32b* favour Precision over Recall, whereas *Qwen3-Coder-480b* is the only model showing no significant Recall degradation.

5 Discussion

The results presented in Sect. 4 highlight several factors influencing the effectiveness of LLM-based feature retrieval. By employing a GLMM, we move beyond descriptive observations to statistically isolate the impact of each factor while controlling for the variance inherent in different software systems (SPL).

The experiments indicate that prompting strategies play a significant role in the ability of LLMs to retrieve implemented features. Differences between `Zero-Shot` and `Few-Shots` configurations suggest that providing examples can improve the model's understanding of the expected task and output format. However, the magnitude of this effect varies depending on the model and the experimental setting, indicating that prompting alone does not fully determine retrieval performance.

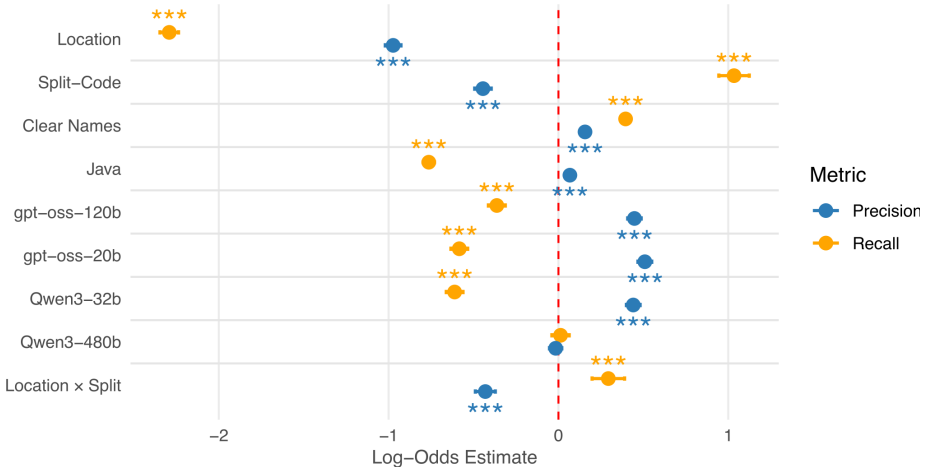


Fig. 7: Generalized Linear Mixed-Effects Models: Recall vs Precision. Considering a 95% Confidence Interval (*** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$).

The obfuscation experiment (`Clear` vs. `Obfuscated`) provides additional insights into how models identify features in source code. When feature names are obfuscated, performance decreases, confirming that lexical cues in identifiers or comments are primary drivers for feature identification. Nevertheless, models are still able to recover part of the feature set, indicating they do not rely exclusively on simple keyword matching. This suggests that LLMs combine lexical signals with structural or contextual information, though our GLMM highlights that these natural language cues act more as a discovery mechanism than a logical validation.

The comparison between `Complete-Code` and `Split-Code` reveals a fundamental behaviour shift. While providing the entire source code allows models to capture broader context, splitting the code into smaller units significantly increases Recall at the expense of Precision. While Ouf et al. [18] intuitively suggest that model performance is tied to code granularity, our GLMM analysis provides a formal quantification of this effect. By isolating the impact of the `Split-Code` strategy, we statistically prove that finer granularity is the primary driver for Recall, while simultaneously uncovering a critical Precision-Recall trade-off that purely descriptive studies often overlook.

The evaluation across several SPL benchmarks shows that model performance varies depending on the analysed system. Differences in implementation structure, feature granularity, and naming conventions influence the ability of LLMs to identify relevant functionality. Furthermore, our analysis reveals a significant penalty for `Java` SPLs that specifically reduces Recall suggesting that Java's verbosity makes feature discovery more complex than in more concise languages like `C`, while other factors such as design choices might be at play.

Finally, the comparison across multiple LLMs indicates distinct model "characteristics": some prioritize reliability (both *gpt-oss*), while others maximize exhaustiveness (*Llama3-70b*). Taken together, these findings suggest that LLM-based feature retrieval can assist engineers during the initial exploration of legacy systems. However, the variability observed indicates that these techniques should be used as supporting tools rather than fully automated solutions, particularly as LLMs appear better suited for coarse-grained feature identification than for fine-grained feature-to-code localization.

5.1 Threats to Validity

Internal validity. A potential threat lies in the design of the prompts themselves. Different prompt formulations could lead to different behaviours. Although we explore several prompting strategies, other prompt designs could further influence the results.

Construct validity. Using separate Precision and Recall metrics instead of a summarized F1-score allows us to isolate the specific impact of each experimental parameter. We complemented paired t-tests with a Generalized Linear Mixed-Effects Model (GLMM) to account for the non-independence of repeated measures across products and models. By treating the SPL as a random effect,

we decouple project-specific variance from the actual influence of prompting and language. Additionally, we integrated the SIDE-score to verify that our results reflect semantic alignment between code and descriptions rather than mere keyword matching.

Conclusion validity. The number of analysed products and observations impacts the statistical significance of the results. Our study considers 248 product variants generated from the selected SPLs, producing a large number of feature identification observations that increase the statistical power of the evaluation and reduce the risk of random fluctuations. Another potential threat concerns the non-deterministic nature of LLMs. To reduce this effect, we set the temperature to 0 (greedy decoding, $k = 1$) and $p = 0.9$, limiting variability between runs. A further confounding factor is the possibility that models rely primarily on lexical matching between feature names and source code identifiers. To assess this risk, we conducted an additional experiment with obfuscated feature names, allowing us to distinguish lexical matching from semantic understanding. Finally, we rely on the HuggingFace inference provider to correctly serve the specified model versions. To facilitate replication, we report the version dates of all evaluated models (Table 3).

External validity. Feature definitions and ground truth are derived from existing SPL benchmarks. While these datasets provide explicit feature models and traceability by design, the way features are defined or implemented in code may influence the evaluation of feature extraction performance. To mitigate this effect, we rely on three different SPL benchmarks implemented in two different programming languages, which reduces the likelihood that the results are specific to a particular dataset or language.

Another limitation concerns the representativeness of the evaluated models and systems. We mitigate this threat by evaluating several LLMs of different sizes and architectures, which allows us to observe global trends rather than behaviours specific to a single model. Nevertheless, the results may still evolve with the rapid progress of LLMs and the availability of newer models. Similarly, although the considered SPL benchmarks are widely used in the literature, they are limited in size (5 to 8 features). Therefore, the results obtained on these systems may not fully generalise to large industrial software systems having more features.

Additionally, the context window of the evaluated models may limit the amount of code that can be analysed at once. Although we experiment with different code granularities, this constraint may still affect the ability of the models to capture long-range dependencies in large files.

Another threat concerns potential benchmark contamination. Since SPL2go benchmarks are publicly available and have been widely used in research and teaching, it is not possible to completely exclude the possibility that fragments of these systems were present in the training data of the evaluated LLMs. While the feature-name obfuscation experiment partially mitigates this risk, it cannot fully rule out memorization effects.

6 Conclusion and Future Work

In this paper, we investigated the ability of LLMs to identify implemented features from source code in the context of SPLs. More specifically, we evaluated how different prompting strategies, code representations, and model choices influence feature extraction performance.

Our results demonstrate that LLMs can effectively recover high-level feature information, though their performance is governed by a fundamental Precision-Recall trade-off. We statistically confirmed that while `Split-Code` is the primary driver for Recall, it significantly degrades Precision by encouraging the model to over-predict features. Conversely, providing the `Complete-Code` favours Precision but limits Recall.

From a practical perspective, these trade-offs reflect different industrial needs. For instance, compliance audits or exploratory analyses may favour Recall-oriented configurations to identify as many potentially relevant features as possible. In contrast, software re-engineering tasks require higher Precision and often involve a human-in-the-loop to ensure reliable feature identification before modifying complex systems.

Our study also highlights several limitations of current LLM-based approaches. While models are generally able to infer which features are present in a product, their performance remains sensitive to lexical cues, code representation, and prompting strategies. Moreover, although we attempted to control generation variability by fixing the temperature and sampling parameters, recent studies suggest that such constraints may sometimes hinder the reasoning capabilities of LLMs. Future work will therefore investigate the impact of decoding parameters such as temperature, top-k, and top-p on feature extraction performance, as well as the behaviour of individual models under different inference configurations. Furthermore, computational cost and inference latency remain important aspects for assessing the practical applicability of LLM-based feature identification approaches in industrial settings.

We also plan to explore prompt tuning and model fine-tuning to improve performance on variability-aware code analysis tasks. Another promising direction consists in combining LLM reasoning with traditional program analysis techniques in hybrid approaches as suggested by Dit et al. [4]. We will also investigate larger and more complex software systems to better understand the scalability limits of LLM-based approaches and identify the point at which performance significantly degrades when analysing larger code bases and richer feature interactions. On top of that, we need to assess the level of reliance on lexical cues to identify the actual level of semantic understanding of computer code from LLMs.

Beyond feature identification from source code, an important next step will be to automatically extract high-level requirements from natural language artefacts such as regulations, standards, or specifications. Such capabilities would support the re-assessment of long-lived systems in highly regulated environments, where evolving legal frameworks and compliance requirements continuously impact existing software systems.

Acknowledgments. This research was funded in part by the CyberExcellence by DigitalWallonia project (No. 2110186), funded by the Public Service of Wallonia (SPW Recherche). Paolo Arcaini is supported by the ASPIRE grant No. JPMJAP2301, JST.

References

1. Acher, M., Duarte, J.G., Jézéquel, J.M.: On programming variability with large language model-based assistant. In: Proceedings of the 27th ACM International Systems and Software Product Line Conference - Volume A. pp. 8–14. SPLC '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3579027.3608972
2. Ali, S.J., Naganathan, V., Bork, D.: Establishing Traceability Between Natural Language Requirements and Software Artifacts by Combining RAG and LLMs, pp. 295–314. Springer Nature Switzerland (Oct 2024). https://doi.org/10.1007/978-3-031-75872-0_16
3. Bicaku, A., Tauber, M., Delsing, J.: Security standard compliance and continuous verification for industrial Internet of Things. International Journal of Distributed Sensor Networks **16**(6), 1550147720922731 (2020). https://doi.org/10.1177/1550147720922731
4. Dit, B., Revelle, M., Gethers, M., Poshyvanyk, D.: Feature location in source code: a taxonomy and survey. Journal of Software: Evolution and Process **25**(1), 53–95 (2013). https://doi.org/10.1002/smr.567
5. Eisenbarth, T., Koschke, R., Simon, D.: Locating features in source code. IEEE Transactions on Software Engineering **29**(3), 210–224 (Mar 2003). https://doi.org/10.1109/tse.2003.1183929
6. Fortz, S., Temple, P., Devroey, X., Heymans, P., Perrouin, G.: VaryMinions: leveraging RNNs to identify variants in variability-intensive systems' logs. Empirical Software Engineering **29**(4) (Jun 2024). https://doi.org/10.1007/s10664-024-10473-5
7. Geng, M., Wang, S., Dong, D., Wang, H., Li, G., Jin, Z., Mao, X., Liao, X.: Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. ICSE '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3597503.3608134
8. Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., Wang, H.: Large language models for software engineering: A systematic literature review. ACM Trans. Softw. Eng. Methodol. **33**(8) (Dec 2024). https://doi.org/10.1145/3695988
9. Jörges, A.: SPL2go – a catalog of software product lines for teaching and research. https://spl2go.cs.ovgu.de/, accessed: 2025-12-09
10. Keim, J., Corallo, S., Fuchß, D., Hey, T., Telge, T., Koziolek, A.: Recovering trace links between software documentation and code. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. ICSE '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3597503.3639130
11. Lin, J., Liu, Y., Zeng, Q., Jiang, M., Cleland-Huang, J.: Traceability transformed: Generating more accurate links with pre-trained BERT models. In: Proceedings of the 43rd International Conference on Software Engineering. pp. 324–335. ICSE '21, IEEE Press (2021). https://doi.org/10.1109/ICSE43902.2021.00040

12. Linsbauer, L., Lopez-Herrejon, R.E., Egyed, A.: Variability extraction and modeling for product variants. Software & Systems Modeling **16**(4), 1179–1199 (Jan 2016). https://doi.org/10.1007/s10270-015-0512-y
13. Marcus, A., Maletic, J.I.: Recovering documentation-to-source-code traceability links using latent semantic indexing. In: Proceedings of the 25th International Conference on Software Engineering. pp. 125–135. ICSE '03, IEEE Computer Society, USA (2003). https://doi.org/10.1109/ICSE.2003.1201194
14. Mastropaolo, A., Ciniselli, M., Di Penta, M., Bavota, G.: Evaluating code summarization techniques: A new metric and an empirical characterization. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. ICSE '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3597503.3639174
15. Michelon, G.K., Obermann, D., Linsbauer, L., Assunção, W.K.G., Grünbacher, P., Egyed, A.: Locating feature revisions in software systems evolving in space and time. In: Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A. SPLC '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3382025.3414954
16. Nguyen, G., Devroey, X.: A3S3 - Automated Android Audit of Safety and Security Signals, pp. 205–212. Springer Nature Switzerland (2025). https://doi.org/10.1007/978-3-031-94590-8_25
17. Nguyen, G., Sacré, A., Simonofski, A., Devroey, X.: Ponderarium, a place for cyber physical system conformity assessment. In: Proceedings of the 59th Hawaii International Conference on System Sciences (HICSS). p. 10. University of Hawai'i at Manoa, Waikoloa, HI, USA (2026), https://hdl.handle.net/10125/111649
18. Ouf, M., Li, H., Zhang, M., Guizani, M.: Reverse Engineering User Stories from Code using Large Language Models. In: 2025 IEEE International Conference on Collaborative Advances in Software and COmputiNg (CASCON). pp. 504–509. IEEE Computer Society, Los Alamitos, CA, USA (Nov 2025). https://doi.org/10.1109/CASCON66301.2025.00080
19. Rhein, A.V., Liebig, J., Janker, A., Kästner, C., Apel, S.: Variability-aware static analysis at scale: An empirical study. ACM Trans. Softw. Eng. Methodol. **27**(4) (Nov 2018). https://doi.org/10.1145/3280986
20. Spieker, H., Matricon, T., Belmecheri, N., Betten, J.E., Lyan, G.L.B., Borges, H., Mazouni, Q., Gross, D., Gotlieb, A., Acher, M.: Prompting for performance: Exploring LLMs for configuring software. In: 2025 IEEE 37th International Conference on Tools with Artificial Intelligence (ICTAI). p. 11411121 (2025). https://doi.org/10.1109/ICTAI66417.2025.00023
21. Tsuchiya, R., Nishikawa, K., Washizaki, H., Fukazawa, Y., Shinohara, Y., Oshima, K., Mibe, R.: Recovering transitive traceability links among various software artifacts for developers. IEICE Transactions on Information and Systems **E102.D**(9), 1750–1760 (Sep 2019). https://doi.org/10.1587/transinf.2018edp7331