

A Methodology for Investigating AI Patterns Prevalence in Software Repositories

Srinath Perera¹, Hasinthaka Piyumal¹, Frank Leymann², and Rania Khalaf¹

WSO2¹, University of Stuttgart²

Santa Clara, CA, USA¹, WSO2, Stuttgart, Germany²

e-mail: {srinath, hasinthaka, rania}@wso2.com¹, frank.leymann@iaas.uni-stuttgart.de²

Abstract—As Artificial Intelligence(AI)-based applications take off, a clear understanding of AI patterns can uplift the quality of AI applications. Many AI patterns have been proposed in the literature; however, their prevalence in real-life code has not yet been validated. Understanding the actual use of those patterns in practice can clarify our understanding both of the significance of these patterns and their utility. In this paper, we present a methodology to a) identify relevant patterns by mining the literature and then to b) validate their presence and prevalence in actual code repositories using active learning. To that end, we identify 14 AI pattern classes by mining 44 published AI pattern-related sources. Then we use an active learning approach to determine the prevalence of the most common pattern class across 100 GitHub open AI repositories. Using prevalence estimation, we propose bounds on the accuracy of the occurrences. The model achieves 56% accuracy and 55% recall in an 8-way classification task, significantly outperforming the 11% random-chance baseline. Furthermore, the prevalence estimation offers usable bounds for analyzing pattern applications. This methodology provides a robust foundation to start understanding how AI patterns are used in practice, a field that currently lacks empirical data.

Keywords-patterns; pattern analysis; software; software engineering

I. INTRODUCTION

Large Language Models (LLMs) [1] now provide general-purpose, high-quality Artificial Intelligence (AI) capabilities that require little to no user-provided training data. These models have unlocked many previously infeasible use cases. Several patterns and abstractions including Retrieval-Augmented Generation (RAG) [2], ReAct [3], and agent-based frameworks [4] now help developers build AI applications. Software Design patterns [5] document recurring problems and solution templates, thereby improving software quality by enabling, communicating, and educating best practices.

As we will discuss in the related work section, many AI patterns have been proposed in the literature. Those patterns aim to capture the authors' observations and experiences. Validating those patterns by understanding their usage frequency in practice can clarify their relative importance, thereby improving the quality of AI-based applications. Furthermore, estimating usage frequency will help us verify candidate patterns using the rule of three (rule of X) [6].

We have identified 769 AI design pattern candidates proposed in the literature. Using word embeddings of those pattern descriptions and clustering, we have grouped them into 78 refined pattern candidates. Then, by manual inspection, we have categorized them into 14 pattern classes.

To verify the prevalence of pattern classes in practice, we selected 100 open-source GitHub repositories that implement

real-world AI applications. From those repositories, we have extracted 2442 code communities. (A code community is a group of tightly coupled methods (procedures) in the call graph.)

Then, using an active learning-based approach [7], we built a classifier to detect the most common patterns listed above. Active learning builds a model with the help of an (e.g., human) oracle, aiming to minimize the number of data annotations (i.e., oracle queries). We proceed by starting with an initial model and iterative labeling of a few data-points that exhibit higher uncertainty in their predictions under the current model. Then, we rebuild the model with the new data and iterate. Consequently, we obtain an accuracy bound for how many times each of the above pattern classes will occur in the repositories.

The paper makes the following contributions.

- By combining methods active learning and human-in-the-loop annotations, we propose a robust methodology to detect patterns without pre-existing annotated datasets, thereby addressing a significant bottleneck in software engineering research: labeled data for niche or emerging patterns is rare.
- We propose a new chunking method to generate code embeddings useful for pattern detection by applying the Louvain method [54] to call graphs to identify "code communities" that serve as chunks.
- We propose applying prevalence estimation techniques (matrix inversion and Monte Carlo simulations) to estimate the true frequency of patterns with useful error bounds.
- We synthesized 769 pattern candidates into a refined taxonomy of 14 pattern classes. We provided one of the first empirical validations of AI pattern prevalence in real-life code by analyzing 100 open-source repositories.

The remainder of the paper is organized as follows. The section II discusses related work, and the following Section III describes the proposed methodology. Section IV describes the implementation details, followed by Section V, which describes the results. The section VI discusses the findings and lessons learned, and Section VII concludes the paper.

II. RELATED WORKS

This section discusses previous work on AI patterns and techniques for Mining Patterns from Code.

A. AI Patterns

Researchers and practitioners have proposed many AI patterns. Among post-LLM patterns, Huang [8] is a book on LLM design patterns. Gullí [9] and Liu et al. [10] discuss agent design patterns. Subramaniam [11] and Jain [12] provide the

practitioner’s perspective. The AWS web page [13] provides a detailed description of best practices for implementing AI and ML applications on AWS. Also, Databricks documentation [14] lists a collection of Agent system design patterns. Furthermore, Arslan et al. [32] and Singh et al. [15] provide detailed surveys of RAG patterns; articles [16] and [11] present practitioners’ perspectives on the same pattern.

Among pre-LLM patterns, Lakshmanan et al. [17] is a textbook on design patterns from the ML domain. Nalchigar [53] provides a detailed taxonomy of solution patterns for ML tasks such as clustering, classification, and anomaly detection. Washizaki [18] lists 15 design patterns (derived from a literature study) for building ML applications. Rodaz et al. [40] propose a mathematical formalism that sketches tasks around ML, but it is not digestible by developers. The Azure website [19] lists best practices for MLOps.

You can find a comprehensive list of sources we used to mine patterns from [20]. These include the sources mentioned above, as well as additional practitioner content, such as articles, blogs, and documentation.

B. Mining Pattern from Code

When detecting patterns in code, we need to first find a code representation and a pattern representation, and then match them. Calculating code representations involves a trade-off between losing information and maintaining generalizability. Pattern representations closer to code are often suboptimal as they fail to generalize across multiple languages. On the other hand, manually crafted pattern representations require significant effort and scale poorly to large numbers of patterns. Many techniques have been proposed for finding design patterns from code.

Search-based approaches use representations closer to code. Work such as Kramer et al. [21] and Dabain et al. [22] focused on detecting patterns using rules that describe them. These rules detect relationships between classes (e.g., inheritance hierarchies and method invocations). To reduce the overhead of pattern specification, Ghulam et al. [23] break GOF [5] patterns into 44 reusable atomic patterns, detect each using SQL, Regular Expressions, or AST parsing, and combine them to discover the patterns. Zdun et al. [24] turn microservice patterns into formal rules and provide a tool that automatically evaluates code in terms of compliance with these patterns.

A logical extension of this method is to represent patterns as subgraphs and to search the code graph for them. Mayvan [25] is an example of this approach. On the other hand, Tsantalis [26] used similarity scores between the code graph and the pattern subgraph, enabling more flexible matching. GEML [27] evolves a set of human-readable if-then-else rules using a greedy algorithm based on inexact graph matching to detect a pattern.

More recent work often focuses on ML-based statistical methods, which can avoid challenges of specifying patterns using rules or templates. Instead of specifying the pattern, they learn a pattern-identifier from sample data.

Among them, classification techniques such as those of Uchiyama et al. [28] and Dwivedi et al. [29] trained a classifier on software metrics such as Depth of Inheritance, Coupling, the number of methods, etc. Chihada et al. [30] use features extracted from an association graph to train the classifier. Zanoni et al. [31] use a combination of graph matching and ML techniques.

The downsides of initial ML techniques include the need for feature engineering, loss of information in features, and limited availability of training data.

Najam et al. [32] build a representation of the source code, apply the word2vec algorithm to construct an embedding space, and then use a classification algorithm. Pandey et al [19] use code embeddings (RoBERTa) generated from code with a KNN algorithm to achieve an F1 score of 0.91. This approach is very flexible as it can be used with minimal effort to detect new patterns if labels for the data are available.

Most of the above approaches work with benchmarks P-MART [33] and DPB [34], which primarily focus on software engineering patterns (e.g., GOF). No suitable dataset is available for detecting AI patterns.

Furthermore, a common limitation is that most benchmarks focus solely on GOF patterns. One notable exception is Fernández et al. [35], which first builds quantum circuits and then uses state-machine-based pattern-detection methods.

Our work is motivated by the results of Pandey et al. [19], which demonstrated the viability of embedding-based approaches. By retaining most of the information in code, embeddings avoid the need for feature engineering. However, embeddings are highly sensitive to the chunking method (i.e., which code is grouped together for calculating embeddings). We extended the work of Pandey et al. [19], which uses P-MART with clean code segments each including a single pattern, to detect patterns in real-life code repositories. Also, our work extends beyond well-known patterns (e.g., GOF) and limited annotated datasets by addressing the lack of training data via an active learning approach.

III. METHODOLOGY

Figure 1 depicts the methodology we used to estimate the occurrence of AI patterns in real-world AI applications as a flow chart. It has four steps. Each of the following subsections discusses four steps of the methodology in detail.

A. Step 1: Extracting pattern candidates from the literature

For mining AI patterns, we used the sources listed in the related work and practitioner articles, blogs, and documentation. You can find the detailed list from [20]. For each source, we extracted text, and for videos, we used the transcript.

We query an LLM (using Prompts [20]) to extract pattern candidates. We receive resulting pattern candidates in the template specified in the prompt.

To detect similarity between pattern candidates, we generated word embeddings for pattern description and clustered embeddings using the DBSCAN algorithm. We explored clustering algorithms DBSCAN, HDBSCAN, OPTICS, Bayesian

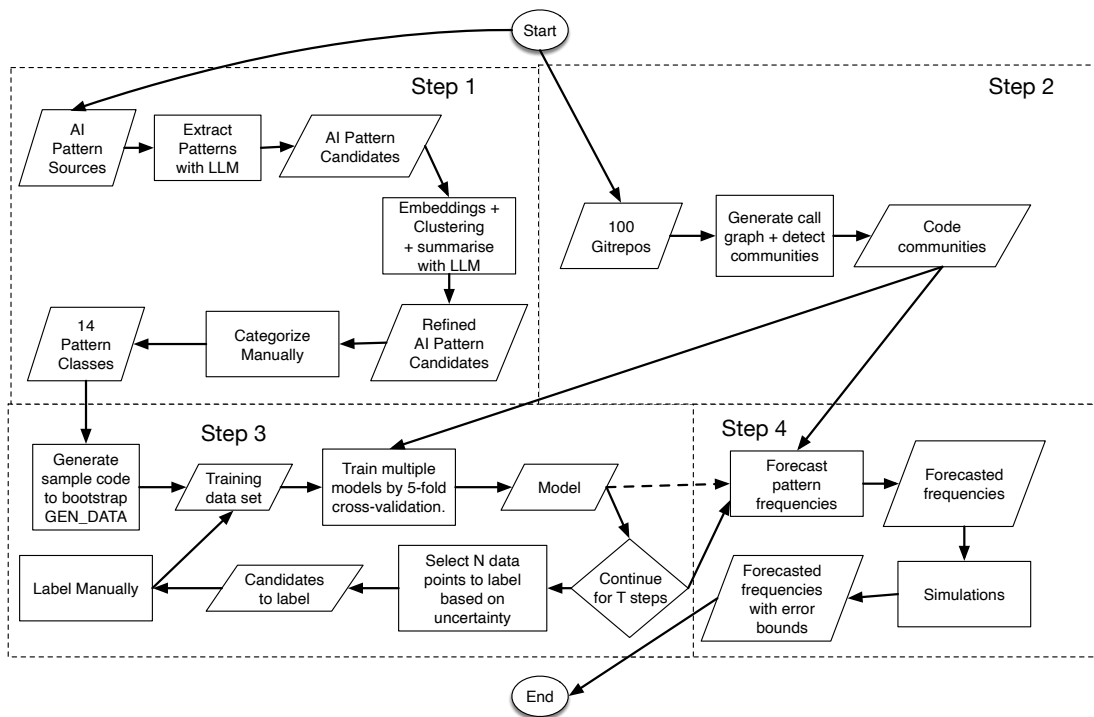


Figure 1. Proposed Methodology

Gaussian Mixtures, and K-Means. We picked DBSCAN by evaluating the even distribution of cluster sizes via silhouette score. We summarize each resulting cluster using an LLM with the prompt [20] and use them as refined pattern candidates.

Then, by manually inspecting pattern candidates, we categorize them under 14 pattern classes using the following criteria.

- We started with pattern classes: RAG, "Advanced LLM Prompting," "Forecasting with Classical Models," and "Multi-Agent Architecture," which were identified in related work.
- We tried to identify a pattern class for each refined-pattern-candidate that we identified by clustering. If we failed, we created a new pattern class. Following this approach, we added pattern classes "Evaluating LLM Results," "Multimodal Data Processing and Prompting," "Using Tools with LLMs," "LLM-based User Intent Extraction," "LLM Fine Tuning," "Training & Alignment," "Enabling Reliability, Explainability, or Robustness," "LLM-based Planning, XoT, ReAct, or Reasoning," and "MLOps" using this approach.
- We also added 2 more pattern classes: "Preprocessing Text and Numerical Data" and "Model Abstraction," while manually labeling data in step four.

The classification of patterns and their refined candidates can be found in [20].

B. Step 2: Collecting Real-world Code Communities

A code community is a group of tightly coupled methods (procedures) in the call graph. To extract code communities, we curated a dataset of 100 repositories hosted on GitHub. The selection process filtered for Python repositories that

possess between 250 and 1,000 stars. We manually verified each candidate to ensure it was a valid AI-related project. The complete list of selected repositories is provided in [20].

Manual inspection of 50 repositories with over 1,000 stars revealed that the majority were foundational frameworks, such as TensorFlow. Consequently, we targeted the 250–1,000 star range to intentionally isolate the ‘application layer’ of AI software, to understand how typical developers use AI in practice.

Embeddings will only work if chunks include full patterns and are reasonably small. For each repository, we generated a procedure call graph and detected communities using the Louvain method [54]. Since all functions within a single community are closely coupled, code within each community is likely to be related. Operating under the hypothesis that such code communities contain complete patterns, we treated all code in each code community as a single chunk and generated code embeddings for each community using `gemini-embedding-001`. We call this dataset $D_{unlabeled}$.

C. Step 3: Build a model with Active learning

We built a model to detect known code pattern classes by first creating a bootstrap version, selecting data for manual labeling via active learning criteria, and then iteratively rebuilding the model using five-fold cross-validation [36]. This cross-validation approach allows us to fully utilize all labeled data points.

To bootstrap the model, we prompted the LLM to generate multiple samples for pattern classes identified in step 1, which we will call D_{init} .

```

Input :  $D_{init}$  (Initial generated data),  $D_{unlabeled}$  (Unlabeled
        embeddings),  $T$  (Total iterations)
Output:  $M_{final}$  (Trained pattern identifier model)
1  $V_{orig} \leftarrow \emptyset$  // Initialize original verified pool
2  $V \leftarrow D_{init}$  // Initialize current working pool
3  $M_{list} \leftarrow \text{None}$ 
4 for  $i \leftarrow 1$  to  $T$  do
5    $V \leftarrow \text{handle\_rare\_classes}(V)$ 
   // Step 1: Train model using current pool
6    $M_{list} \leftarrow \text{train\_hybrid\_model\_cv}(V)$ 
   // Step 2: Active Learning - Identify uncertain samples
7    $S_{hl} \leftarrow \text{find\_uncertain\_topN}(M_{list}, D_{unlabeled})$ 
   // Step 3: Human-in-the-loop labeling
8    $D_{new} \leftarrow \text{label\_manually}(S_{hl})$ 
   // Step 4: Update pools
9    $V_{orig} \leftarrow V_{orig} \cup D_{new}$ 
10   $D_{unlabeled} \leftarrow D_{unlabeled} - S_{hl}$ 
11   $V \leftarrow V_{orig}$ 
12 return  $M_{list}$ 
    
```

Figure 2. Active Learning based Pattern Identifier

The algorithm in Figure 2 explains the active learning based approach. At each step (algorithm 2, line 5), we dropped classes with fewer than 5% of the data and 20 data points. We do not know the true labels; hence, we cannot control the distribution of labeled data without significant effort. In the tradeoff between stable results vs. forecasting more classes, we chose stability. As steps progressed and more data became available, more classes will become available.

In `find_uncertain_topN()` function, we select data points to label from $D_{unlabeled}$ by forecasting them using current models, normalizing class probabilities of different models, averaging them to create overall predicted class probabilities, and performing margin sampling. Margin is the difference between the highest and second-highest classes in the forecast, and we select data points to label based on the smallest margin.

Algorithm in Figure 3 presents the logic for building a cross-validated model. And algorithm 4 describes how to perform the final forecast as a weighted average of probability distributions, with weights based on F1 scores across folds.

D. Step 4: Pattern Occurrence Estimation based on the results

Let m denote the model trained in step 3 to detect pattern classes in a code community. We utilize the normalized confusion matrix derived from the model’s cross-validation, denoted as C . Given the set of code communities D (constructed in step 2), we define O_D as the observed distribution of pattern classes in D as predicted by m .

Assume T_D is the estimated true distribution of class in D .

$$O_{samp} = C \cdot T_D \implies T_D = C^{-1} O_{samp} \quad (1)$$

Given O_D , we can find T_D by using a linear solver.

When estimating T_D using our model, we face two sources of uncertainty: model misclassifications and sampling errors

```

// Pseudocode for train_hybrid_model_cv(..)
Input: Verified Data  $\mathcal{V}$ 
Output: Models  $M$ , F1 scores of models  $F$ 
1  $\mathcal{N} \leftarrow \{\text{"lr"}, \text{"knn"}, \text{"svc"}\}$ 
2  $C_{folds} \leftarrow \emptyset$  // fold configs
3  $\mathcal{F} \leftarrow \emptyset$ 
4  $O_{folds} \leftarrow \emptyset$ 
5 for  $f \leftarrow \text{split\_to\_folds}(\mathcal{V})$  do
6    $D_{train} \leftarrow f.train$ 
7    $D_{val} \leftarrow f.validation$ 
8    $D_{test} \leftarrow f.test$ 
9    $M_{list}, F_1 \leftarrow \text{train}(\mathcal{N}, D_{train}, D_{val}, \emptyset)$ 
   // use algorithm in Figure 4 to forecast
10   $O_{folds} \leftarrow O_{folds} \cup \text{forecast}(M_{list}, D_{test})$ 
11   $C_{folds} \leftarrow \text{get\_configs}(M_{list})$ 
12   $\mathcal{F} \leftarrow \mathcal{F} \cup \{F_1\}$ 
13 end
14 print_results( $O_{folds}, D_{test}$ )
   // Aggregate results and train final ensemble
15  $M_{list}, \_ \leftarrow \text{train}(\mathcal{N}, V, \emptyset, C_{folds})$ 
16 return  $\{M_{list}, \mathcal{F}\}$ 
    
```

Figure 3. Building a model with Cross-validation

```

// Pseudocode for forecast(...)
Input:  $x$  Input data set, Models  $M$ , F1 scores of models  $F$ 
Output:  $Y$  The final predicted class labels
/* Definitions: */
1  $C$ : The set of all possible class labels
2  $f_i \in F$ : The  $F_1$  score used as the raw weight for the  $i^{th}$ 
   model
3  $P(y = c | m_i, x)$ : The probability that model  $m_i$  assigns to
   class  $c$  for input  $x$ 
// Ensemble Forecast Calculation
4  $Y = \text{argmax}_{c \in C} \sum_{i=1}^{|M|} \left( \frac{f_i}{\sum_{j=1}^n f_j} \cdot P(y = c | m_i, x) \right)$ 
5 return  $Y$ 
    
```

Figure 4. Final Forecast

from O_D . We minimize the first error using equation 1, and the second by sampling from a multinomial distribution to estimate the true frequencies, then performing a Monte Carlo simulation to obtain error bounds. Algorithm 4 depicts our approach.

IV. IMPLEMENTATION DETAILS

At every step, we use Gemini 3 Flash as the LLM. All experiments are done in an Intel Core™ i7-10510U × 8 processor with 16.0 GiB running Ubuntu 24.04.3 LTS

We already discussed the choice of clustering algorithm used in step 1 in the methodology. Furthermore, we tried dimension reduction with PCA and UMAP. We received the best results with DBSCAN with UMAP.

In step 3, for code embedding, we experiment with embedding methods Jina Embeddings v2 (Code), CodeSage Large v2, CodeRankEmbed, RoBERTa, CodeBERT, Voyage Code-2, Gemini Text Embedding 004, and selected Gemini Text

```

Input:  $C$  (Confusion Matrix),  $O$  (Observed counts),  $\alpha$ 
      (significance level),  $N$  (Iterations)
Output:  $[L, U]$  (Confidence interval bounds)
1  $\mathcal{R} \leftarrow \emptyset$  // Initialize results collection
2  $C_{norm} \leftarrow \text{normalize}(C)$  // Normalize columns to sum to 1
3  $n \leftarrow \sum O$  // Total sample count
4  $p \leftarrow O/n$  // Observed frequencies
5 for  $i \leftarrow 1$  to  $N$  do
6   // Sample from multinomial to simulate observation noise
    $O_{samp} \leftarrow \text{sample\_multinomial}(n, p)$ 
7   // Solve linear system using Equation 1
    $E \leftarrow \text{solve}(C_{norm}, O_{samp})$ 
8    $\mathcal{R} \leftarrow \mathcal{R} \cup \{\text{clip}(E)\}$  // Store clipped estimates
9  $L \leftarrow \text{percentile}(\mathcal{R}, \alpha/2)$ 
10  $U \leftarrow \text{percentile}(\mathcal{R}, 100 - \alpha/2)$ 
11 return  $[L, U]$ 

```

Figure 5. Inversion-based Estimation via Monte Carlo Simulation

Embedding 00,4, which provided the best classification F1-score while building a bootstrapping model in active learning.

Unlike step 1, in step 3, using dimensionality reduction with the classifier model reduced the F1-score; Hence, we did not use any dimensionality reduction.

While building the model for step 3, we tried Logistic Regression (LR), Neural Networks, Support Vector Classifier (SVC), Naive Bayes, Random Forest, XGBoost, Gradient Boosting, K-Nearest Neighbour (KNN), and Decision Tree. We used LR, SVC, and KNN, which provided the best individual F1 scores in the final averaging ensemble.

While building the model in step 3, we achieved about a 4% improvement in F1 score using the averaging ensemble method in code listing 3. Building a linear regression stacked model using outputs from a neural network and a linear regression model also yielded good results. Still, we chose the averaging ensemble method because the stacked model is complex and highly susceptible to overfitting.

In step 3, we used the following workflow for manually labeling code communities. Given a code community (cc) and a predicted pattern class pc, we use the LLM to generate a code description for cc, ask the LLM to judge whether the forecasted pattern class pc is correct, and describe its decision. We then manually verified that the LLM’s judgment was correct. If the prediction (pc) is wrong, we read the description of cc and propose an alternative pattern class. [20] lists Prompts used for this step.

During active learning, in the third iteration, we tried building the model both with and without LLM-generated sample data, and it performed better with only verified data. So we only used verified data from the third iteration onward.

In step 4, for the simulation, we assume the confusion matrix of the model developed in step 3 is representative. As discussed in the methodology, we model the sampling error as independent categorical draws from a multinomial distribution, with the observed category counts as the basis for the distribution. We used 20000 as the iteration count,

increasing it until repeated tests did not change the outputs.

V. RESULTS

Manually verified data had pattern frequencies as follows: Forecasting with Classical Models (64), None (64), LLM-based Multimodal Generative Prompting (52), Preprocessing Text and Numerical Data (51), Using Tools with LLMs (31), RAG (28), Agent Architecture (20), Model Abstraction(20), Evaluating LLM Results (18), MLOps (18), Advanced LLM Prompting (14), LLM-based Planning, XoT, ReAct, or Reasoning (12), Enabling Reliability, Explainability, or Robustness (8), LLM-based User Intent Extraction (8), and LLM Fine-Tuning, Training & Alignment (4).

However, the rest of the results will only have 7 pattern classes because the model will automatically map any class less than 5% and 20 instance to the None class. Table I shows the results after the fourth iteration.

The verified data column shows the number of manually labeled data points available. As expected, classes with fewer training samples have lower F1 scores.

Table 6 shows the confusion matrix.

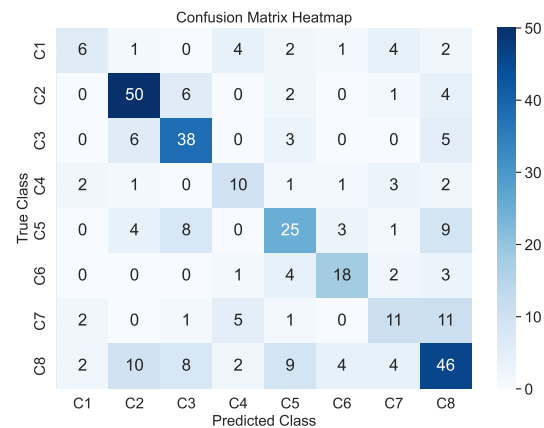


Figure 6. Weighted Vote Confusion Matrix

A strong main diagonal indicates the model’s strength. As we discussed in the methodology, most classes are often misclassified as "None", and the "None" class often gets misclassified. Hence, it is a common source of error. Considering the often misclassified patterns, "Using tools with LLM" is often misclassified as "None", which could be because "tool use" is not very prominent in code and lacks a strong signal. Furthermore, classical models and preprocessing both get confused by "None" in both directions, which may be because these patterns include a wide variety of techniques that are harder to generalize with a small set of samples.

Figure 7 depicts our results from the prevalence estimation described in step 4.

There are several zeros in bounds for the last two classes, which could occur when a class is affected by high false-positive noise from other classes or when it is often misclassified as other classes. We conducted a sensitivity test for both

TABLE I. CLASSIFICATION PERFORMANCE METRICS ACROSS CATEGORIES.

Category	Precision	Recall	F1-Score	Verified Data
C1 Agent Architecture	0.50	0.30	0.38	20
C2 Forecasting with Classical Models	0.69	0.79	0.74	63
C3 Multimodal Generative Prompting	0.62	0.73	0.67	52
C4 Model Abstraction	0.45	0.50	0.48	20
C5 Preprocessing Text and Numerical Data	0.53	0.50	0.52	50
C6 RAG	0.67	0.64	0.65	28
C7 Using Tools with LLMs	0.42	0.35	0.39	31
C8 None	0.56	0.54	0.55	85
Overall Values	0.56	0.55	0.55	

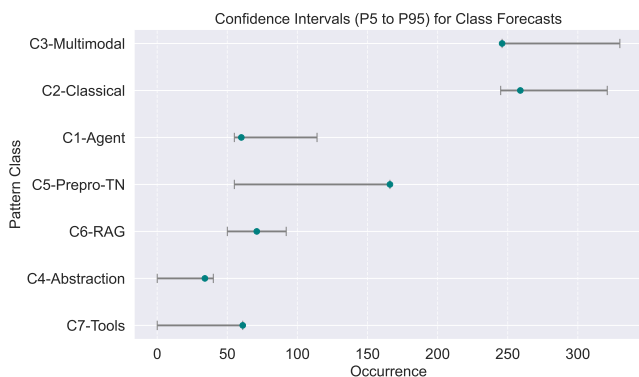


Figure 7. Estimated Pattern Class Prevalence with 95% confidence intervals

cases and found that the probable cause is that "Tool Use for LLMs" is often misclassified as "None" as shown in the confusion matrix in Figure 6.

Smaller training samples likely cause wider confidence intervals for the last two classes. When we exclude those, the confidence interval variation is 45%, which provides usable estimates for our problem. Furthermore, confidence levels indicate when estimates are unreliable, and users can address that by running more iterations of the algorithm. Since the algorithm selects labeling data based on uncertainty, there is a good chance that underrepresented classes will be labeled in subsequent iterations.

Where are the prevalence results at the current accuracy level useful? First, it can confirm the wider availability of 5/7 pattern classes. Furthermore, it can give us relative availability of different pattern classes. For example, even using given confidence levels, we can argue that pattern classes C2 and C3 are much more common than C1, C5, and C6. Such understanding can help us focus our attention on teaching these patterns. However, the current accuracy level is not enough to identify a specific pattern given a code segment, which is a future direction we plan to explore.

The following lists concrete examples organized by pattern class, found based on the model’s forecast for extracted communities.

- Agent Architecture - multi-agent debate, multi-agent orches-

tration, communication through an agent bus, coordination through group chat.

- LLM-based Multimodal Generative Prompting - Predicting object centers, dimensions, and local offsets, inferring an object’s 3D position, bounding box prediction, motion prediction, sequential detections of an object from Lidar data, resizing and padding 3D tensors, generating synthetic training data.
- Model Abstraction - abstracting multiple AI backends or local/ cloud producers, routing, adopting Prompts via template, and managing API keys
- Preprocessing Text and Numerical Data - anonymizes, binning, type conversions, data dictionary, extracting metadata, text normalization, categorical feature encoding
- Retrieval Augmented Generation(RAG) - search with vector database, using custom knowledge bases, caching, and retriever component

VI. DISCUSSION AND LESSONS LEARNED

One code community can house multiple patterns. However, to keep manual data labeling simple, we have only used a single label. However, our output provides a probability distribution across all classes, which can be used to detect multiple patterns.

Contrasting our results with those of Pandey et al. [19], which achieved an F1-score of 0.91, it is worth noting that we also tried the same algorithms and embeddings. The likely difference is the curated nature of PMART vs. the complexity of real-life code we used, which often includes additional logic. Furthermore, as we discussed in the results section, users of our methodology can assess the reliability of estimates using error bounds. If they require tighter error bounds, they can likely achieve them by running more iterations of the step three active learning algorithm, which likely yields more training data for those classes and thereby improves the error bounds. Given the limited understanding of design pattern frequency in the real world, even the bounds we reported after four iterations remain usable for 5 out of the 7 classes and represent a significant step forward.

In step 3, we folded classes that had fewer than 5% or 20 samples into the None class, which is a practical choice. Some patterns are common, while others are rare, and naturally, there are many potential patterns. To make our approach stable,

we had to draw the line somewhere. When the number of samples in a class is small, when combined with 5-fold cross-validation, the F1-score changed significantly (>10%) due to each misclassified sample, thus becoming unreliable [37].

However, we acknowledge that this practical choice reduces the granularity of results, as artificial non-class can interfere with other classes. Because the boundary between what is considered a pattern and what is not is fluid (e.g., subjective in how abstract we want the problem and solution to be), we believe our choice is still useful. If needed, practitioners can often get more granularity by labeling more data points.

We extend our work beyond basic software patterns (e.g., GOF) and limited annotated datasets by addressing the lack of training data via an active learning approach. The methodology we used is highly independent of AI patterns, and the same approach is likely to work well with other types of patterns. The ability to detect patterns by building a classifier on embeddings has been demonstrated by Pandey et al. [19] using the P-MART data set, and by this paper on real-world AI projects. It is likely that other domains also contain sufficient information in their embeddings.

Detecting patterns via embeddings is highly sensitive to the choice of chunking method. Initially, we generated embeddings per file in the repository, resulting in poor classification performance in step 3. We needed a chunking method that would keep all relevant code about a pattern in a single chunk. Given that goal, typical chunking approaches like fixed size, pooling, by function, or by class would not work. AST-like methods (e.g., CAS [[66]) also focus on code syntax structure not the dependencies. In contrast, a call graph naturally captures dependencies similar to code graph-based approaches used in related work (e.g., Mayvan [25], Tsantalis [26]). It is much more likely that code from the same pattern, which is more closely related, is more connected in the call graph. This is also a natural extension of Codegrag [38], which has used a similar idea for code retrieval. The performance improved significantly after adopting the call graph-based communities as chunks.

We tried to detect patterns by clustering the code communities collected in step 2 and then interpreting and summarizing the resulting clusters. This approach did not work well. However, it may be possible to fine-tune the embeddings using techniques such as contrastive learning improve the separation between clusters, which is also a future work we plan to explore.

Methodology uses LLM in three ways, and each can introduce bias. First, pattern candidates were extracted using the LLM and manually curated to identify pattern classes. Manual curation should help reduce bias, but will not address bias shared between LLMs and humans. Second, although we bootstrapped with LLM-generated data in the first round, after 3rd round, we dropped LLM-generated data because only using labeled data gave better results. Third, LLM-assisted manual labeling can also introduce bias. Exploring the use of LLM as a judge technique to reduce such bias is a useful direction for future research.

VII. CONCLUSION AND FUTURE WORK

In this paper, we aim to enable and shift the exploration of AI design patterns toward empirical validation using software repositories. To that end, we address three challenges. Embedding-based pattern detection is highly susceptible to chunking methods. We proposed a new chunking method based on call graph-based community detection. The performance of the resulting model validates our method. Second, AI has limited annotated pattern datasets, which we address through an active learning-based approach. Third, we address both sampling and classification errors while estimating pattern prevalence by using a matrix-inversion-based technique and Monte Carlo simulations.

The model achieves 56% accuracy and 55% recall in an 8-way classification task, which is 5 times higher than the 11% random-chance baseline. Prevalence estimation provided useful error bounds for 5 of 7 classes. In other cases, users can detect unreliable estimates through error bounds and discard them, or obtain better estimates by running more iterations in step three.

We synthesized 769 pattern candidates into a refined taxonomy of 14 pattern classes. Most patterns show more than three examples required to pass the rule of three. Frequency estimates for those classes confirm our understanding in some cases (e.g., Preprocessing Text and Numerical Data and Classical Models) and open new avenues of inquiry in others (Multimodal prompting and RAG).

This work opens several avenues for future research. First, applying the techniques in other domains to explore the universality of the approach (e.g., applying the same techniques for microservices, quantum computing), second, studying the relationships between possible bounds vs. the number of training samples for each class, and third, exploring techniques, such as SLM fine-tuning and constructive learning, to improve model performance. All information required to recreate the above work, including code, labeled data points from step 4, and code, is available in [20].

REFERENCES

- [1] Y. Chang et al., "A survey on evaluation of large language models", *ACM Transactions on Intelligent Systems and Technology*, vol. 15, no. 3, pp. 1–45, 2024. DOI: 10.1145/3641289.
- [2] P. Lewis et al., "Retrieval-augmented generation for knowledge-intensive NLP tasks", in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33, 2020, pp. 9459–9474.
- [3] S. Yao et al., "ReAct: Synergizing reasoning and acting in language models", *arXiv preprint arXiv:2210.03629*, 2023.
- [4] J. He, C. Treude, and D. Lo, "LLM-based multi-agent systems for software engineering: Literature review, vision, and the road ahead", *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 5, pp. 1–30, 2025. DOI: 10.1145/3702989.
- [5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.
- [6] C. Kohls and S. Panke, "Is that true...? thoughts on the epistemology of patterns", in *Proceedings of the 16th Conference on Pattern Languages of Programs*, 2009, pp. 1–14.

- [7] A. Tharwat and W. Schenck, “A survey on active learning: State-of-the-art, practical challenges and research directions”, *Mathematics*, vol. 11, no. 4, p. 820, 2023.
- [8] K. Huang, *LLM Design Patterns: A Practical Guide to Building Robust and Efficient AI Systems*. O’Reilly Media, 2025.
- [9] A. Singh, A. Ehtesham, S. Kumar, and T. T. Khoei, “Agentic retrieval-augmented generation: A survey on agentic rag”, *arXiv preprint arXiv:2501.09136*, 2025.
- [10] A. Gullí, *Agentic Design Patterns*. Packt Publishing, 2024.
- [11] B. Subramaniam, “Emerging patterns in building GenAI products”, 2024, Accessed: Jan. 22, 2026. [Online]. Available: <https://martinfowler.com/articles/gen-ai-patterns/>.
- [12] A. Jain, “Agentic AI architectures and design patterns”, 2024, Accessed: Jan. 22, 2026. [Online]. Available: <https://medium.com/@anil.jain.baba/agentic-ai-architectures-and-design-patterns-288ac589179a>.
- [13] Amazon Web Services, “AWS prescriptive guidance: Patterns: AI & machine learning”, 2026, Accessed: Jan. 22, 2026. [Online]. Available: <https://docs.aws.amazon.com/prescriptive-guidance/latest/patterns/machinelearning-pattern-list.html>.
- [14] Databricks, “Agent system design patterns”, 2026, Accessed: Jan. 22, 2026. [Online]. Available: <https://docs.databricks.com/aws/en/generative-ai/guide/agent-system-design-patterns>.
- [15] J. Alammar, “Retrieval-augmented generation (RAG) patterns and best practices”, InfoQ, 2024, Accessed: Jan. 22, 2026. [Online]. Available: <https://www.youtube.com/watch?v=eUY9i1CWmUg>.
- [16] Neo4j, “GraphRAG field guide: RAG patterns”, 2026, Accessed: Jan. 22, 2026. [Online]. Available: <https://neo4j.com/blog/developer/graphrag-field-guide-rag-patterns/>.
- [17] V. Lakshmanan, S. Robinson, and M. Munn, *Machine Learning Design Patterns*. O’Reilly Media, Inc., 2020.
- [18] H. Washizaki et al., “Software-engineering design patterns for machine learning applications”, *Computer*, vol. 55, no. 3, pp. 30–39, 2022. DOI: 10.1109/MC.2021.3139049.
- [19] S. K. Pandey et al., “Design pattern recognition: A study of large language models”, *Empirical Software Engineering*, vol. 30, no. 3, p. 69, 2025.
- [20] “Ai patterns github repository”, 2026, Accessed: Jan. 27, 2026. [Online]. Available: <https://github.com/wso2-incubator/ai-patterns>.
- [21] C. Kramer and L. Prechelt, “Design recovery by automated search for structural design patterns in object-oriented software”, in *Proceedings of WCRE’96: 3rd Working Conference on Reverse Engineering*, IEEE, 1996, pp. 208–215. DOI: 10.1109/WCRE.1996.558906.
- [22] H. Dabain, A. Manzer, and V. Tzerpos, “Design pattern detection using FINDER”, in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, 2015, pp. 1554–1560. DOI: 10.1145/2695664.2695755.
- [23] G. Rasool and P. Mäder, “Flexible design pattern detection based on feature types”, *Automated Software Engineering*, vol. 18, no. 3-4, pp. 339–365, 2011. DOI: 10.1007/s10515-011-0084-2.
- [24] U. Zdun, E. Navarro, and F. Leymann, “Ensuring and assessing architecture conformance to microservice decomposition patterns”, in *International Conference on Service-Oriented Computing*, Springer, 2017, pp. 411–429.
- [25] B. B. Mayvan and A. Rasoolzadegan, “Design pattern detection based on the graph theory”, *Knowledge-Based Systems*, vol. 120, pp. 211–225, 2017.
- [26] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, “Design pattern detection using similarity scoring”, *IEEE Transactions on Software Engineering*, vol. 32, no. 11, pp. 896–909, 2006. DOI: 10.1109/TSE.2006.112.
- [27] R. Barbudo, A. Ramírez, F. Servant, and J. R. Romero, “Geml: A grammar-based evolutionary machine learning approach for design-pattern detection”, *Journal of Systems and Software*, vol. 175, pp. 110–119, 2021.
- [28] S. Uchiyama, H. Washizaki, and Y. Fukazawa, “Design pattern detection using software metrics and machine learning”, in *First International Workshop on Model-Driven Software Migration (MDSM 2011)*, 2011, pp. 38–42.
- [29] A. K. Dwivedi, A. Tirkey, and S. K. Rath, “Software design pattern mining using classification-based techniques”, *Frontiers of Computer Science*, vol. 12, no. 5, pp. 908–922, 2018.
- [30] A. Chihada, V. Arnaudova, L. M. Eshkevari, G. Antoniol, and Y.-G. Gueheneuc, “Source code and design performance, design pattern detection from source code by classification approach”, *Applied Soft Computing*, vol. 26, pp. 357–367, 2015. DOI: 10.1016/j.asoc.2014.09.043.
- [31] M. Zaroni, F. A. Fontana, and F. Stella, “On applying machine learning techniques for design pattern detection”, *Journal of Systems and Software*, vol. 103, pp. 102–117, 2015. DOI: 10.1016/j.jss.2015.01.037.
- [32] N. Nazar, A. Aleti, and Y. Zheng, “Feature-based software design pattern detection”, *Journal of Systems and Software*, vol. 185, pp. 111–179, 2022.
- [33] Y.-G. Guéhéneuc, “P-mart: Pattern-like micro architecture repository”, *Proceedings of the 1st EuroPLOP Focus Group on pattern repositories*, pp. 1–3, 2007.
- [34] F. A. Fontana, A. Caracciolo, and M. Zaroni, “DPB: A benchmark for design pattern detection tools”, in *2012 16th European Conference on Software Maintenance and Reengineering*, IEEE, 2012, pp. 235–244. DOI: 10.1109/CSMR.2012.33.
- [35] M. Fernández-Osuna, M. A. Pérez-Delgado, M. Rojo-Martínez, and M. Piattini, “Exploring design patterns in quantum software: A case study”, *Computing*, vol. 107, no. 5, pp. 1–31, 2025. DOI: 10.1007/s00607-024-01365-z.
- [36] R. D. King, O. I. Orhobor, and C. C. Taylor, “Cross-validation is safe to use”, *Nature Machine Intelligence*, vol. 3, no. 4, pp. 276–276, 2021.
- [37] C. Beleites, U. Neugebauer, T. Bocklitz, C. Krafft, and J. Popp, “Sample size planning for classification models”, *Analytica chimica acta*, vol. 760, pp. 25–33, 2013.
- [38] K. Du et al., “Codegrag: Bridging the gap between natural language and programming language via graphical retrieval augmented generation”, *arXiv preprint arXiv:2405.02355*, 2024.