Empirical Software Engineering (2025) 30:69
https://doi.org/10.1007/s10664-025-10625-1

Design pattern recognition: a study of large language models

Sushant Kumar Pandey[1,2] · Sivajeet Chand[1] · Jennifer Horkoff[1] · Miroslaw Staron[1] · Miroslaw Ochodek[3] · Darko Durisic[4]

Accepted: 5 February 2025 / Published online: 18 February 2025

Abstract

Context As Software Engineering (SE) practices evolve due to extensive increases in software size and complexity, the importance of tools to analyze and understand source code grows significantly.

Objective This study aims to evaluate the abilities of Large Language Models (LLMs) in identifying DPs in source code, which can facilitate the development of better Design Pattern Recognition (DPR) tools. We compare the effectiveness of different LLMs in capturing semantic information relevant to the DPR task.

Methods We studied Gang of Four (GoF) DPs from the P-MARt repository of curated Java projects. State-of-the-art language models, including Code2Vec, CodeBERT, CodeGPT, CodeT5, and RoBERTa, are used to generate embeddings from source code. These embeddings are then used for DPR via a k-nearest neighbors prediction. Precision, recall, and F1-score metrics are computed to evaluate performance.

Results RoBERTa is the top performer, followed by CodeGPT and CodeBERT, which showed mean F1 Scores of 0.91, 0.79, and 0.77, respectively. The results show that LLMs without explicit pre-training can effectively store semantics and syntactic information, which can be used in building better DPR tools.

Conclusion The performance of LLMs in DPR is comparable to existing state-of-the-art methods but with less effort in identifying pattern-specific rules and pre-training. Factors influencing prediction performance in Java files/programs are analyzed. These findings can advance software engineering practices and show the importance and abilities of LLMs for effective DPR in source code.

Keywords Large language model · Design pattern recognition · Software reengineering · Deep learning

1 Introduction

The complexity and scalability of modern software applications is continually evolving. Design patterns (DPs) (Gamma et al. 1995; Kuchana 2004) offer robust and proven solutions

Communicated by: Steffen Herbold and Eunjong Choi

This article belongs to the Topical Collection: *Predictive Models and Data Analytics in Software Engineering (PROMISE).*

Extended author information available on the last page of the article to recurring design challenges. Gamma et al. (1995) define DPs as *"recurring solutions to common problems in a given context and system of forces"*. Since their popularisation by the 'Gang of Four' (GoF) DPs, these patterns serve as architectural templates or skeleton structures of software design to guide developers in developing high-quality, maintainable, scalable software systems; this is particularly crucial in large-scale software. Research on GoF DPs shows mixed results, some patterns improve software quality, while others may have negative effects (Ampatzoglou et al. 2013). This makes it crucial to explore their real impact further. Design Pattern Recognition (DPR) can accelerate the development process, reducing the possibility of errors and improving the overall quality of the software system (Zhang and Budgen 2011). Moreover, DPR can also help in code understanding and maintainability of a software system.

Despite the advantages of recognizing DPs as part of software development, manually identifying and categorizing patterns within large and complex codebases can take time and effort. Automatic detection of DPs has assisted software developers in quickly and correctly comprehending and maintaining unknown source code, ultimately leading to higher productivity (Walter and Alkhaeir 2016; Christopoulou et al. 2012; Scanniello et al. 2015). However, most existing methods often rely on rule-based techniques, which require manually created rules, or ML-based techniques which require extensive labelled training data, difficult to find or create. Many state-of-the-art DPR approaches also rely on creating the abstract syntax tree (AST) (Tsantalis et al. 2006; Guéhéneuc and Antoniol 2008), but extracting AST for all programs that could not compile would be challenging (in the context of code snippets in continuous integration). Tsantalis et al. (2006) provided an approach that provides 100% recall for many patterns using a similarity score of the graph vertices. However, this approach requires substantial pre-processing.

Further studies use machine learning-based approaches for DPR (Nazar et al. 2022; Zanoni et al. 2015). These approaches involve training a machine learning model on a dataset of known DPs and then using the model to identify similar patterns in new code. While promising, these approaches are still in the early stages of development and require source code metrics and significant training data to be effective. These approaches are targeted for specific languages and are usually limited to general DPs that cannot recognize DPs for particular domains, for instance, the automotive domain (Pandey et al. 2023). Subsequently, there are challenges associated with existing DPR methods: 1) ML-based methods need a significant amount of training set, which is hard to collect in real-world applications; 2) existing methods work on known and recognized DPs, requiring DP-specific rules; and 3) static-analysis approaches require a program to compile to extract AST of a program which can be challenging in real-world applications. In this paper, we address challenges 1) and 3) in particular.

The emerging natural language processing (NLP) field has witnessed the growth of large language models (LLMs) and their capability as powerful tools across diverse applications in the SE domain. This study aims to test the ability of LLMs for DPR. We conduct DPR without performing any domain-specific pre-training, pre-processing of LLMs, and without using any DPR metrics, and without requiring code to compile. By refraining from pre-training the LLM, we seek to show the model's ability to capture semantic, contextual, and structural information from source code. Our approach works by comparing embeddings of programs to a small set of known DPs, thus requiring few examples compared to related work.

Our previous work presented an introductory study using LLMs for DPR (Chand et al. 2023). We pre-trained two widely utilized LLMs on two extensive automotive software codebases, AndroidAuto and GENIVI. Subsequently, we fine-tuned these LLMs by employing the k-means clustering algorithm. Our experiment involved small-scale examples extracted from open-source programs implementing Singleton, Prototype, and Builder DPs. Our findings revealed that Code2Vec significantly outperformed Word2Vec in accurately predicting the correct DPs. The mean F1-score achieved by Code2Vec was 0.78, surpassing Word2Vec, which achieved a mean F1-score of 0.69.

As shown by Unger and Tichy (2000), focusing on documenting DPs improves communication and maintainability in software systems. Our study focuses on detecting DPs in Java code using LLMs, and these models (e.g., RoBERTa) have been pre-trained on large datasets that may include documents where patterns are described, helping to uncover and potentially document patterns that were not previously documented in the code.

In this study, we build on these previous results and through more experiments, we analyze the impact of various model choices on performance different DPs, investigating the factors that influence the performance of LLMs. We also explore the consistency of results across different DPs, providing insights into the models' generalization capabilities in DPR.

In this study, we evaluate the ability of five LLMs to recognize five different GoF patterns. Four of the LLMs are deliberately not pre-trained (Code2Vec cannot be used without pre-training). Methodologically, we process programs from the P-MARt (Pattern-like Micro-Architecture Repository) repository, pass them to the LLMs to get the embedding, then use k-NN(k-Nearest Neighbors) to classify the resulting embedding as containing or not containing DPs, calculating precision, recall and F-1 scores. Although our ultimate aim to support recognition of any, domain-specific DP with no rules and few examples, in the study we start by evaluating the ability of our approach to recognize a sub-set of popular GoF patterns, giving the availability of widely used repositories for these patterns, and enabling us to compare our results with the state of the art. Next steps will include applying the method to further, domain-specific examples, such as those found in automotive (Pandey et al. 2023). We shared the replication package[1] of this study to maintain transparency and of this study.

The study will address the following research questions (RQs). These research questions examine different dimensions of the same set of results.

RQ-1: *Which LLM performs the best in DPR for the selected patterns?* To address this RQ, we compare the performances of different LLMs by calculating the average precision, recall, and F-1 scores across five selected DPs. We found that RoBERTa and CodeGPT produce the maximum (0.91) and second maximum (0.79) mean F1-score, respectively, although the performance of other LLMs is comparable. The results show that Code2Vec and CodeT5 show high variance in DPR, whereas CodeBERT has a smaller variance across DPs.

RQ-2: *Amongst those evaluated, which DPs are the most easily detectable by the LLMs?* The results found that Abstract Factory, Builder and Prototype patterns were detected with high F1-scores across all LLMs. The results also show that Builder and Singleton have high variance in their detection, and the Factory method has the smallest variance across LLMs. Results show that the performance of LLMs is comparable with state-of-the-art methods.

RQ–2.1: *How do the results for various DPR compare to state-of-the-art methods?* Results show no significant difference in the performance of LLMs with state-of-the-art methods. LLMs produce the highest (1) F1-score for Builder DP. For Abstract Factory and Factory Method, the state-of-the-art (not LLM-based) methods produce the highest F1-score. Despite this, LLM approaches retain the benefit of not needing pattern-specific rules or extensive labelled training data.

[1] https://github.com/sushantkumar007007/Design-Pattern-Recognition-using-Large-Language-Models-

RQ-3: *Are pattern instances consistency classified or miss-classified by different LLMs?* We found that some examples of Abstract Factory, Factory Method, Prototype, and Singleton are consistently correctly classified across LLMs. Very few Factory Methods and Singleton patterns implemented files are always misclassified. We found that consistency in classificaton depend on the types of patterns.

RQ–3.1: *What factors contribute to any inconsistencies?* This RQ looks at individual pattern examples to investigate the factors that contribute to different classifications across different LLMs. The study found that along with semantics, syntactic information, the context of implementation, keywords, and implementation style are the factors contributing to DPR by LLMs.

We summarize the contributions of this study:

(i) We introduce an approach to address some limitations in existing DPR methods by utilising state-of-the-art LLMs combined with the k-NN classification method.
(ii) Our study includes a comparison with state-of-the-art DPR methods. This comparative analysis provides a benchmark for understanding the advancements achieved through LLMs.
(iii) Finally, our study recommends which LLMs can have the most promise for DPR. In future work, these LLMs can then be further used with pre-training using domain-specific codebase for domain-specific DPR tasks.
(iv) The study reveals the factors involved in DPR tasks across different DPs by LLMs.

This paper is organized as follows. Section 2 provides an overview of related work in the field. Section 3 elucidates the preliminary information. Our methodology is explained in Sect. 5. Section 6 reveals our empirical results, followed by a thorough discussion in Sect. 7. Section 8 examines potential threats to the study. We conclude our findings and future work in Sect. 9.

2 Related Work

We organize the discussion of the related work section into two subsections. First, we explore DPR methods; then, we focus on the evolution of language models in software engineering, providing an overview of source code analysis.

2.1 Design Pattern Recognition Methods

Tsantalis et al. (2006) suggested a novel approach for DPR based on the similarity scoring between graph vertices. They represented the program structure in a graph structure. They employed three projects from P-MARt, and used Singleton, Factory Method, and Prototype DPs in their experiments. Their method showed high performance by achieving a 95.9% F1-score across Java projects. Barbudo et al. (2021) introduced an approach to DP detection. Their method uses evolutionary machine learning techniques, utilising diverse software properties. Initially, they applied an evolutionary algorithm to extract DP characteristics, followed by applying a rule-based classifier for predictions. The study focused on 15 DPs commonly found in Java programs from the P-MARt repository, achieving F1-score ranges from 0.56 to 1. Inspired by their work, we have utilized some of these DPs in our investigation. Furthermore, we compare the performance of various LLMs with their proposed method. In another approach, Rasool and Mäder (2011) performed a study, utilising various search-based techniques in DP detection and compared them with state-of-the-art methods DPR methods. Across seven open-source projects and twenty-two patterns, their findings revealed mean F1-scores ranging from 0.97 to 1 in different projects. Their study is based on creating semi-formal definitions for each pattern and then identifying patterns based on features. While their approach successfully identified DPs in specific projects, it showed challenges in other patterns and projects. In our study, we employed some of the projects explored in their work and conducted a comparative analysis.

Further work by Xiong and Li (2019) in DP detection from source code present a practical methodology by using idiomatic implementations within the Java language. Formalizing DPs under the layered knowledge graph (LKG), their model encapsulates language-independent DP concepts and Java language. Their approach facilitates search strategies through static analysis and inference techniques. The runtime performance shows the practical applicability of their proposed approach. We aim to extend this work by addressing the limitations by incorporating semantic information for a more comprehensive DP detection and a more diverse set of DPs, which was not explicitly considered in the original study.

A further contribution to DPR is presented by Bernardi et al. (2014). This study presents the importance of developing knowledge about DP instances to enhance program comprehension. The proposed approach employs source code parsing to trace the roles played by system components by using high-level structural properties such as inheritance, dependency, etc. The method also extends its utility to detecting DP variants by identifying overridden pattern properties. They investigated seven open-source projects, including systems of varying sizes, showing the efficacy of the approach, ranging recall and precision from 7.74 to 1.0 and 0.5 to 1, respectively. Our motivation lies in improving DP detection methodologies by explicitly incorporating structural properties and semantic information, which helps a more comprehensive understanding of DP implementations in Java code.

Some approaches have considered using various forms of machine learning for DPR. In work presented by Zanoni et al. (2015), the authors extend previous work by integrating machine learning techniques into an analysis framework. The proposed methodology combines graph matching and machine learning, implemented through the MARPLE-DPD tool, to enhance DPR. They conducted experiments on five DPs across ten open-source software projects. They tried seventeen classifiers and found an F1-score range from 0 to 0.91 for Singleton DP. Pettersson et al. (2010) presented a study for DP detection. Their research found that the high impact of existing variations on prediction performance hinders comparisons across techniques. To address this, the authors introduce a benchmark method, for DP occurrence detection metrics. Heuzeroth et al. (2003) introduces an innovative approach to improve program comprehension through a tool capable of automatically generating static and dynamic analysis algorithms from DP specifications. To achieve this, the authors define static and dynamic patterns' specifications as predicates, representing legacy code through predicates encoding its attributed abstract syntax trees. The static analysis, derived from the specification of static pattern aspects, performs a query on the legacy code representation, producing potential pattern candidates. Finally, the dynamic specification summarizes expected state sequences during pattern use.

Recent advancements in DPR have been seen through the integration of LLMs, as exhibited by the work of Parthasarathy et al. (2022). In their study, a pre-trained model from Facebook was employed and evaluated on industrial controller-handler DPs implemented in C++. The model showed satisfactory performance, achieving an F1-score ranging from 0.473 to 0.85. Similarly, Pandey et al. (2023) utilized the same LLM, fine-tuning it for creational DPs. Testing on two modules developed by an automotive company, the model accurately predicted DP in 10 out of 16 C++. Further recent work by Pandey et al. (2024) explored four LLMs for DPR, but only used cosine distance on ten Singleton-implemented C++ examples from GitHub finding that all four LLMs perform inconsistently, with further exploration needed. These recent works highlight the growing significance of LLMs in advancing the field of DPR and show promising results. Inspired by these studies, we are motivated to apply an LLM-based approach to Java codebases and explore the potential of state-of-the-art LLMs for enhanced DPR.

2.2 Language Models in Software Engineering

The first work that motivated using LLMs for SE tasks was conducted by Hindle et al. (2016). The paper presented the software development process using natural languages, exhibiting patterns and regularities; the language models can be applied to different programming tasks, such as code repair. Allamanis and Sutton (2013) concluded that language models are the new dimensions of software project analysis and understanding, driven by the large-scale and statistical understandings gained from extensive corpora. The very first language model that showed remarkable success in many programming tasks was Code2Vec (Alon et al. 2019), a study by Tian et al. (2020) applied Code2Vec for program repair. They investigated different representation learning approaches; they found that Code2Vec produces an AUC value of about 0.8 in predicting patch correctness on a deduplicated dataset of 1,000 labeled patches. The motivation behind including Code2Vec in our study (this and previous study) is its high performance in program analysis tools in various SE tasks. As part of our previous study, we pre-trained the Code2Vec model using two automotive codebases, and we utilized this pre-trained model in this study.

Svyatkovskiy et al. (2020) introduce a tool called IntelliCode Compose, a multilingual code completion tool for integrated development environments (IDEs). It uses a generative transformer model trained on extensive datasets of 1.2 billion lines of source code in Python, C#, JavaScript, and TypeScript. IntelliCode is deployed as a cloud-based web service. The trained model achieved an average edit similarity of 86.7% and perplexity of 1.82 for the Python language, they used the GPT-C model, which is a variation of GPT-2. Motivated by the successful application of GPT models, we have also applied CodeGPT, a variant of the GPT model, which is a multi-layer generative pretrained transformer model for code.

A further study (Haque et al. 2022) focuses on source code summarization. The paper conducted experiments to assess the correlation between various word overlap metrics and human-rated similarity of predicted and reference summaries. To improve the evaluation of source code summarization, the study explores an alternative for semantic similarity metrics, which provides valuable insights for improving the evaluation of code summarization. They found that the embedding-based approach produces 0.82, 0.52, and 0.54 similarity accuracy and completeness and outperforms the n-gram-based approach. Motivated by this study, we also used embeddings of programs to feed into a k-NN classification layer for DPR.

AlphaCode, a novel system for code generation presented by Li et al. (2022), addresses the challenge of improving the problem-solving capabilities of transformer-based neural network models in programming tasks. It uses trained transformer-based networks to generate millions of diverse programs. AlphaCode achieved an average ranking in the top 54.3% in recent programming competitions. Our work is based on the same principles as AlphaCode (using transformer models) but is aligned with software architectural needs and requirements. The paper (Silva et al. 2020) introduces RefDiff 2.0, a refactoring detection tool; it recognizes the value of identifying refactoring operations in source code changes for understanding software evolution. The RefDiff 2.0 utilizes a novel algorithm based on the code structure tree, a language-agnostic representation of source code, and enables its applicability to various programming languages. The evaluation shows that RefDiff produces precision and recall in Java, as 96% and 80%, respectively.

A recent study presented by Xiao et al. (2023), that extensively investigates three prominent pre-trained models for source code, i.e., CodeBERT, CodeGPT, and CodeT5, across four critical software engineering tasks, 1) Code summarization, 2) Bug fixing, 3) Bug detection, and 4) Code search. The results show the effectiveness of decoder-only models, particularly CodeGPT, in specific generation tasks based on state-of-the-art evaluation metrics. The findings challenge the assumption that the most frequently used models are universally suitable for all applications, emphasizing the need for modified solutions to address developers' diverse needs. The paper concludes that transformer-based models are efficient for code-related tasks. This study motivates us to employ the suggested LLM (CodeGPT) in DP detection tasks. Wang et al. (2023) presented a large-scale empirical study and evaluation of six state-of-the-art techniques for code search and four for code generation with and without performing pre-training of LLMs and analyzing a dataset comprising over 22,000 natural language queries. The study shows the critical findings in code search techniques performing model pre-training, which reveal high effectiveness. The results were also observed for code generation techniques, the study found that GraphCodeBERT + CodeT5 produces Bulescore of 30.0/30.2 and 36.9/37.4, and GraphCodeBERT + CodeT5 + CodeBERT produces Bulescore 31.8/32.0 37.9/38.9 in two different scenarios. Motivated by this study, we employed CodeT5, and CodeBERT in this work.

While previous studies have explored LLMs in various SE tasks, our work takes a different approach by investigating LLMs for DPR in Java code. We employ a combination of LLMs and k-NN classification techniques, improving current DPR methodologies. This study investigates the implications of LLM for DPR, which may help us build a better DPR tool than existing methods and help us understand LLMs' ability to solve similar SE problems (e.g., code smell detection, code refactoring).

3 Background

This section discusses the background of Large Language Models, including pre-training and fine-tuning.

3.1 Large Language Models

LLMs are used as a powerful tool in source code analysis. They are trained on vast amounts of source code, textual, and code snippet data. The seminal paper *Attention is All You Need* by Vaswani et al. (2017) laid the foundational architecture for LLMs by introducing the Transformer architecture. Unlike traditional recurrent neural networks (RNNs) and long short-term memory networks (LSTMs), Transformer-based models leverage the self-attention mechanism that enables parallel processing of input data, thereby enhancing computational efficiency and scalability. The attention mechanism allows the model to weigh the importance of different words in a sentence, facilitating a deeper understanding of contextual relationships and dependencies.

LLMs, such as GPT models (e.g., CodeGPT) and BERT (e.g., CodeBERT), have shown remarkable abilities to understand context, semantics, and syntactic structures (Li et al. 2023;

Chen et al. 2021). Utilizing the contextual information generated from these models can extend DPR methods more efficiently.

In general, when utilizing LLMs for SE tasks, the following steps are typically taken. Training/Tuning: An LLM is taken off-the-shelf an is either pre-trained or fine-tuned with task-specific data to perform better on specific tasks. More information about this step is described in Sec. 3.2. Step 1: Code, for which there is a classification question such as which, if any, DP is contained, is collected and preprocessed. Preprocessing will remove unnecessary information like comments and spaces. For example, consider a program implementing the Singleton pattern as shown in list 1, the question is whether this code is a Singleton, or contains another DP, or none? Step 2: The preprocessed code is converted into an embedding using the pre-trained/fine-tuned LLM. For instance, using the RoBERTa-based model, the code is tokenized into a sequence of tokens, and each token is embedded into a vector that captures its semantic meaning. The result is a vectorized representation of the code. Step 4: The embedding is classified using a classifier, in our case k-NN. This classifier can be seen as an extra layer to the LLM, which takes the embedding and gives an answer with is either yes/no, or mapping the embedding to one of a number of categories, e.g., DPs. Step 5: steps 1 to 4 are used to receive individual answers on specific pieces of code in practice. Before use in practice, the overall performance of the model is first evaluated by repeating steps 1 to 4 many times for many examples, with different runs using different example code files for pre-training/fine-tuning and evaluation, producing performance scores such as F1.

Listing 1 Sample Example

```
public class Singleton {
    private static Singleton instance;
    private Singleton() {}
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

3.2 Pretraining and Fine-Tuning

Transformer-based models undergo two primary training phases: pre-training and fine-tuning. During pre-training, the model engages in self-supervised learning by predicting parts of the input data, such as the next word in a sentence or filling in masked words. This approach allows the model to learn general language patterns, grammar, and factual knowledge from vast amounts of text data without explicit labels. This phase equips the model with a broad understanding of language, making it adaptable to various downstream tasks. Fine-tuning, on the other hand, involves training the pre-trained model on a specific dataset tailored to a particular task, such as sentiment analysis, machine translation, or question-answering (Radiya-Dixit and Wang 2020). This two-step process allows LLMs to achieve high performance and versatility across diverse applications by building upon the foundational knowledge acquired during pre-training.

In our case, the task of focus is DP recognition in Java code, allowing the model to adjust its general knowledge to focus on features relevant to this task using labeled DP examples. Technically, fine-tuning can be performed in two ways. Since Transformer-based LLMs are artificial neural network (ANN) models, one approach is to add inference layers on top of the pre-trained model and continue training with a labeled dataset. Another approach is to use the embeddings generated by the pre-trained model, which encodes language features, as input to a task-specific model. This second method offers flexibility in designing a task-specific model architecture, better suited to the problem and dataset size, and helps mitigate overfitting, especially with smaller datasets. In our work, we adopt this second method, using k-NN as the task-specific model.

In our previous study (Chand et al. 2023), we pre-trained two LLMs Code2Vec and Word2Vec models on two automotive code bases, AndroidAuto and Genivi, using these models for DPR. The objective during pre-training was for the model to develop a general understanding of the automotive domain and DPs used, including their syntax, semantics, and common code patterns. Based on our previous results we found, that Code2Vec produces a mean F1-score of 0.78, which outperforms Word2Vec, which only produced a mean F1-score of 0.69 on Java-based Singleton, Prototype, and Builder DPs. Because of these results, in this study, we use only the pre-trained Code2Vec model, with four other state-of-the-art LLMs. Table 1 provides a summary of the LLMs that considered pre-training and fine-tuning, along with the datasets used for the fine-tuning process.

In this study, we opted not to perform pre-training on state-of-the-art LLMs, CodeBERT, CodeT5, CodeGPT, and RoBERTa, as shown in Table 1, due to their extensive existing pre-training on large-scale, diverse datasets that contain a variety of programming languages, including Java. We are interested in understanding the performance of these models without the additional step or burden of pre-training with task-specific data. Furthermore, Liu et al. (2022) established that fine-tuning is more effective than pre-training for specific applications rather than investing resources in redundant pre-training. Thus, in this study we focus mainly on fine-tuning. Note that Code2Vec is an exception, as this model cannot be used without pre-training, thus we use the pre-trained version of the model from Chand et al. (2023).

4 Proposed Technique

In this section, we detail our approach to applying LLMs for DPR using k-NN.

4.1 Overview

Fig. 1 shows the overview of the suggested DPR method using LLMs and k-NN classification technique. The figure has the three phases, A, B, and C. Phase A represents Java code programs where the present or absence of a specific DP is under question. When evaluating our method with code from P-MARt, each program has labels indicating whether it implements a particular pattern(s), allowing us to evaluate the effective of our approach. In practice, the label of a program will be unknown, but an answer as to whether a DP is contained will be given in phase C.

In phase B, the code is fed into LLMs, producing tokens for each keyword (Tok_1 to Tok_m) and unique tokens like empty space/special character/unknown keywords ([SEP]), as shown in Fig. 1. These tokens are combined into embedding vectors based on the LLM's

Table 1 Summary of Pre-training and Fine-tuning for LLMs applied in this study

LLM Model	Pre-training Performed	Codebase for Fine-tuning	Fine-tuning with k-NN
CodeBERT	No	P-MARt	Yes
CodeT5	No	P-MARt	Yes
CodeGPT	No	P-MARt	Yes
RoBERTa	No	P-MARt	Yes
Code2Vec	Yes (AndroidAuto + Genivi)	P-MARt	Yes

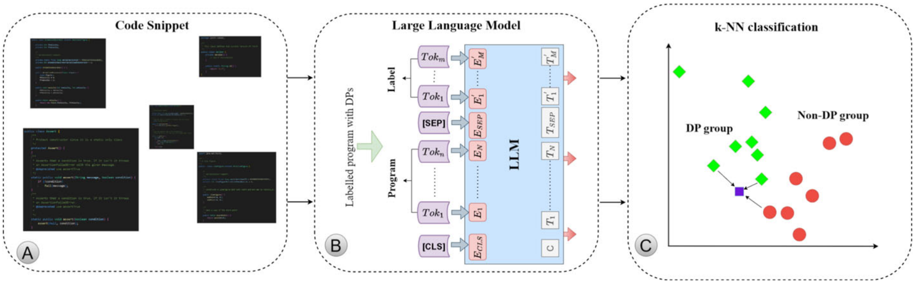


Fig. 1 Overview of Design Pattern Recognition using Large Language Model combined with k-NN classification technique

bucket/token size. LLMs generate embeddings that capture semantic information of code, representing the contextual understanding of each keyword.

In phase C, embedding vectors are used as input to the k-NN classification method, predicting the DP in question. In our figure, the green diamond represents the program embedding of a specific DP (e.g., Singleton), the red circle denotes the non-implemented DP group (e.g., non-singleton), and the purple rectangle is the embedding vector of an unknown DP program. The k-NN approach employs Euclidean distance to predict whether a program implements a specific DP. This distance-based prediction helps in recognizing implemented patterns in the code.

4.2 Design Pattern Recognition Model

This subsection describes the specific steps we have undergone to pre-train and fine-tune the chosen LLMs. It then elaborates on the k-NN model and how k-NN is applied with LLMs using Euclidian distance. Further, the section will also discuss the selection of different state-of-the-art methods for comparison to our results, and our selection of performance measures.

4.2.1 Large Language Model using k-NN for DPR

Large language models generate rich and context-aware representations for code elements as embedding vectors. We use this as input to k-NN, treating this step as fine-tuning to the original LLM. In the context of our study, we consider two programs, denoted as P_1 and P_2, each implemented with distinct DPs, DP_1 and DP_2, respectively, illustrated in Fig. 2. Their respective embedding vectors are E_1 and E_2. For a program P_k, the Euclidean distances from P_1 and P_2 are d_1 and d_1, respectively. Based on our analysis, the determined DP in P_k is based on comparing distances d_1 and d_2, where $d_2 < d_1$. Utilizing the k-NN approach, if P_k is close to programs implementing DP_2, the DP in P_k is predicted as DP_2.

We decided to rely on k-NN as our model of choice to implement the task-specific layer of our DPR models due to its inherent alignment with semantic search objectives, simplicity, and robust performance in high-dimensional feature spaces. Since it is based on measuring the distance between the examples (in our case programs), it allows for debugging and insights into the model's decision-making process by retrieving the most similar examples from the training dataset (still the user must compare both and draw conclusions). Also, k-NN is generally known for its effectiveness in pattern recognition tasks, making it a balanced choice that combines performance, ease of use, and adaptability (Cover and Hart 1967;

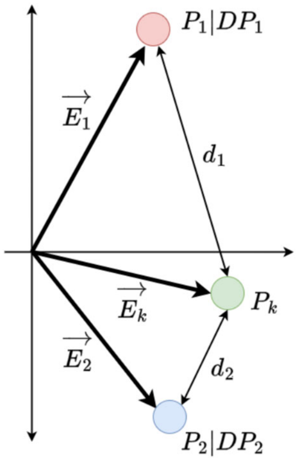


Fig. 2 Illustration of Euclidean distance between embedding vectors

Altman 1992; Zhang et al. 2004; García et al. 2015). Several other machine-learning models, like decision trees, are known for their built-in interpretability and explainability, but since the input features are a multidimensional embedding vector generated by the base model, knowing how each feature contributes to the final decision still does not allow human experts to interpret the decision made by classifier.

Algorithm 1 Design Pattern Recognition.

```
Input ← Set of Java programs $P$ with known DP labels, LLM model parameters $\Theta$, and Java Programs $P'$ with unknown labels.
Output → Labels of programs, and Trained model $M$ with optimized parameters $\Theta$
Initialize $M$ with LLM model using pre-trained weights    ▷ LLM from open source
procedure ExtractEmbeddings($P$, $M$)    ▷ Embedding for each of the $P$ with unknown labels
    for each program $p_i$ in $P$ do
        Tokenize $p_i$ using LLM
        Get embeddings $M_E(p_i)$ from LLM
    end for
end procedure
procedure CalculateDistanceMatrix($M_E(P)$)
    Initialize distance matrix $D$ with zeros
    for each pair of programs $p_i, p_j$ do
        Calculate Euclidean distance: $D_{ij} \leftarrow$ Distance($M_E(p_i), M_E(p_j)$)
    end for
end procedure
procedure TrainModel($P, L, M, \Theta$)    ▷ Embedding vectors is the train set, and the predicted label $L$
    for each program $p_i$ in $P$ do
        Find $k$-nearest neighbors of $p_i$
        Predict label for $p_i$ based on majority label of neighbors
        Calculates the Precision, Recall, & F1-score
    end for
    Optimize parameters $\Theta$ by minimising misclassification error
end procedure
TunedModel($\Theta'$) ← $P'$
TunedModel($P, L, M, \Theta'$) and Predicted labels of $P'$    ▷ Predicted labels $L$ of programs, Tuned model saved
```

Algorithm 1 presents the pseudocode of the steps involved in our DPR method using an LLM and k-NN.

The input to the algorithm is the set of Java program P, with a set of Label L, a set of Java programs P', with unknown DPs, and LLM with initial parameter Θ. The algorithm's output is the predicted labels of unknown programs trained model M with optimized parameters, including parameters related to k-NN Θ'.

1) Embeddings Extraction: We used a pre-trained model to extract embeddings from the P-MARt Java projects, as shown in lines 4 to 9. These embeddings represent the code in a high-dimensional vector space, where similar code snippets are located closer to each other. The procedure takes P, and M as the input, the LLM tokenizes each program p_i, and produces the embedding $(M_E(p_i))$ vector from the encoder block. The LLM processes the programs and generates embeddings that capture syntactic and semantic features. This procedure includes the following steps internally, a) Tokenize each program in P, b) Input the tokenized program into the pre-trained LLM, c) Retrieve the program's embedding vectors from the model's output layers, d) Return the vector representations for each program.

2) Distance Matrix Computation: The k-NN algorithm calculates a distance matrix based on the Euclidean distances between the embeddings of different input programs (from lines 10 to 15). The choice of Euclidean distance is motivated by its effectiveness in measuring geometric relationships between two programs in n-dimensional space and helps to find dissimilarity between program embeddings. Furthermore, Euclidean distance aligns with the geometric intuition of measuring straight-line distances between points, providing a meaningful interpretation of the dissimilarity between programs. The shorter the distance, the more similar the programs, and vice versa (Qian et al. 2004).

The algorithm takes the set of embedding vectors for all programs $(M_E(P))$. Then, takes the pair of embeddings of Java programs p_i and p_j, calculates the distance, and stores it in a zero initialized matrix D. The distance between the pair of embeddings $Distance\big(M_E(p_i), M_E(p_j)\big)$ is stored in D_{ij}.

3) k-NN Classification: After obtaining these embeddings and distance calculation, we use the k-NN algorithm to fine-tune the model, lines 16 to 21, i.e., TrainModel(). K-NN is a non-parametric method that classifies new examples based on the majority class of their closest neighbors in the embedding space. This allows us to classify code snippets in terms of DPs based on their closeness to other annotated snippets in the dataset. During the fine-tuning process, the embedding space is adjusted to be more sensitive to the structural, semantic, and syntactic features specific to DPs. For each program in the dataset, the k-NN algorithm identifies its k-nearest neighbors based on the previously computed distance matrix. The majority label among its neighbors determines the predicted label for the program.

After fine-tuning using k-NN, we evaluated the model's performance by comparing the predicted classifications against the true labels of DPs from the P-MARt repository. The performance of the model is evaluated based on the prediction rate. The trained model takes Java programs with unknown DP labels as input, and outpus a prediction of the DP lables for these programs.

Finally, the trained and most optimized model $\big(TrainModel(P, L, M, \Theta')\big)$ is saved. Our approach has optimized the parameters Θ by minimizing misclassification error, where Θ represents the network's weights. The misclassification error is quantified through a loss function in classification tasks. During training, the model calculates the gradient of the loss concerning each parameter and updates Θ in the opposite direction to reduce the error. This process is repeated iteratively until the error converges to a minimum. This final model $(\big(TunedModel(P, L, M, \Theta')\big))$ can then be used for DPR in practice.

5 Evaluation Methodology

This section outlines the details of our experimental methodology evaluating the performance of the approach described in Sec. 4 using different LLMs, including data collection, utilizing LLMs, our experimental setup, and evaluation metrics.

5.1 Selected Large Language Models

We have chosen five recent LLMs for this study: one from our previous study (Chand et al. 2023) and four selected using specific inclusion and exclusion criteria. Specifically, we included models that were newer than 2018 and trained on Java programs and/or documents. We looked for models that have a recent history of successful use in SE tasks, chose models that were available for pre-training (for future studies), and were open-source. We excluded LLMs which were not publicly accessible or available for fine-tuning or pre-training. For example, this includes models which are only accessible from a user interface (e.g., GPT-4).

The chosen LLMs are detailed in the following.

(i) Code2Vec[2] In contrast to the conventional approaches that rely on bag-of-words or token-based representations, Code2Vec captures the structural complexities of code by considering paths within a program's Abstract Syntax Tree (AST). At its core, Code2Vec adopts a Continuous Bag of Paths (CBOP) methodology, where it processes sets of code paths and learns to embed them into continuous vector representations. With a neural network architecture containing 100 million parameters, Code2Vec shows a high capacity for capturing small relationships within code structures. The Code2Vec model has proven effective in various code understanding tasks (Ciniselli et al. 2021), for instance, code completion (Ciniselli et al. 2021), code clone detection (Yuan et al. 2022), code summarization (Bui et al. 2021), bug detection (Li et al. 2019), and other broader software comprehension task. We have used the Code2Vec model in the previous work (Chand et al. 2023), where we pre-trained the original model (Alon et al. 2019) using two codebases from the Automotive domain (AndroidAuto, Genivi). As part of this work, we pre-trained the model until 43 epochs, stopping when there was a saturation in the pre-training performance (F1-score = 80.08%). This version of the model is used as part of this study.

(ii) *Code Bidirectional Encoder Representations from Transformers (CodeBERT[3])*: This state-of-the-art LLM is developed by Microsoft Research (Feng et al. 2020). It uses the BERT architecture, CodeBERT is designed for code generation tasks, showing its application across various SE tasks. The model uses a masked language model (MLM) objective during pre-training, where code segments are masked, and the model is tasked with predicting the masked tokens. This methodology enables CodeBERT to acquire contextualized representations of code elements, allowing it to capture small relationships and complex semantics within a program. The model, which has over 110 million parameters, has been pre-trained in diverse programming languages, including Java, with high accuracy in understanding Java syntax and semantics, making it a valuable tool for our study. Furthermore, a pre-trained version of has been applied in different applications, for instance, automated software repair (Mashhadi and Hemmati 2021), code clone detection (Arshad et al. 2022), and program understanding (Zeng et al. 2022).

[2] https://github.com/sivajeet/DPR-using-code2vec-and-word2vec:

[3] https://huggingface.co/microsoft/codebert-base/tree/main

(iii) *Code Generative Pre-trained Transformer (CodeGPT[4]):* This LLM has been developed by AISE-TUDelft, and emerges as a popular state-of-the-art LLM. It is used as a multilingual code generation tool: CodeGPT uses transformer architecture to understand and generate code snippets across various programming languages. CodeGPT has been applied in code search, bug detection, code summarization (Parvez et al. 2021), code generation (Lu et al. 2021; Tipirneni et al. 2022) and bug fixing (Tufano et al. 2019).

(iv) *Transformative Code Understanding with Text-To-Text Transfer Transformer (CodeT5[5]):* It is also a state-of-the-art LLM, with the complex architecture of the Text-To-Text Transfer Transformer (T5) and functioning as a code understanding tool. The model uses a text-to-text framework, where input and output are treated as text, allowing it to handle many code-related tasks seamlessly. The model can operate across multiple programming languages, including Java, making it suitable for our study. CodeT5 has been successfully applied to various SE applications, for instance, code understanding generation (Wang et al. 2021), code smell detection (Kovačević et al. 2023), source code understanding (Zhang and Lu 2003), etc. The model is publicly available for pre-training and fine-tuning.

(v) *Robustly optimized BERT approach (RoBERTa[6]):* This LLM is a transformer-based language model originally developed by Liu (2019). It is trained with large mini-batches and learning rates, contains more than 125 M parameters, and uses BPE as a tokenizer. It has been successfully applied in various SE tasks, such as code understanding (Wang et al. 2022), and code completion (Ciniselli et al. 2021). Our experiment used the base-RoBERTa model from HuggingFace. The architecture includes attention mechanisms and positional embeddings, allowing the model to learn complex text sequences.

5.2 Dataset

We utilized the nine projects from the P-MARt repository,[7] a curated collection of diverse Java projects that implemented GoF DPs. According to the TIOBE[8] index, Java is currently one of the most popular programming languages in the world across various domains.

The P-MARt repository is often used for DPR studies and is widely accepted by the SE community (Almadi 2022; Xiong and Li 2019; Bernardi et al. 2014; Moreira et al. 2022; Naghdipour et al. 2023). It contains a diverse annotated and ground truth set of GoF DPs implemented in Java. The P-MARt repository comprises 9 distinct projects with a total of 4,374 files. P-MARt includes an XML file that lists paths to files implementing various DPs and antipatterns. The details of the nine projects from P-MARt repository are shown in Table 2. Lexi v0.1.1 is the smallest project, with just 24 programs/files and a total of 7,101 lines of code (LOC). On the other hand, the Netbeans v1.0.x project is notably larger, comprising 2,558 programs/files and approximately 31,000 LOC. This variation in project sizes demonstrates the diversity within our dataset.

[4] https://huggingface.co/AISE-TUDelft/CodeGPT-Multilingual/tree/main

[5] https://huggingface.co/Salesforce/codet5-base/tree/main

[6] https://huggingface.co/RoBERTa-base/tree/main

[7] https://www.ptidej.net/tools/designpatterns/

[8] https://www.tiobe.com/tiobe-index/

Table 2 P-MARt project description

Project	No. of Files	LOC	No. of Classes	No. of Methods
JHotDraw v5.1 (JHD)	155	8,419	136	1,393
JRefactory v2.6.24 (JRfac)	569	55,871	556	4,690
Lexi v0.1.1 alpha (Lexi)	24	7,101	23	601
QuickUML 2001 (Q-UML)	156	9,249	142	1,264
MapperXML v1.9.7 (M-XML)	213	14,928	195	2,307
Netbeans v1.0.x (NB)	2,558	317,542	2,238	25,446
JUnit v3.7 (Junit)	79	4,886	69	856
Nutch v0.4 (Nutch)	173	23,579	149	1,832
PMD v1.8 (PMD)	447	41,321	423	3,752
Total	4,374	482,896	3,931	42,141

5.3 Design Pattern Selection

For this, our initial experiments into evaluating the effectiveness of various LLMs for DPR, we limit our DP selection to those that are both widely recognized in the SE community and frequently used in prior works such as SparT (Xiong and Li 2019), DBF (Bernardi et al. 2014), RaM (Rasool and Mäder 2014), DPD (Tsantalis et al. 2006), and GEML (Barbudo et al. 2021). Specifically, we focus on creational DPs from the GoF for this initial investigation. Rahman et al. found that the creational DPs positively impact software quality more than behaviour and structural DPs (Rahman et al. 2023).

Although it is interesting to evaluate the ability of an LLM to detect different types of patterns (creational, behavioral, structural); here, our goal is to determine if LLMs are promising for DPR compared to state-of-the-art approaches. Thus, to manage the complexity of the study, we focus on only creational patterns. If our results show that LLM DPR is promising, future work should focus on comparing results between higher-level pattern types.

We evaluate DPR with five creational DPs: 1) Singleton, 2) Builder, 3) Prototype, 4) Abstract Factory, and 5) Factory Method. These creational patterns were chosen due to their proven effectiveness in enhancing code organization, maintainability, and scalability (Wedyan and Abufakher 2020; Ampatzoglou et al. 2012).

These five patterns offer diversity in complexity and abstraction levels. Singleton is a simple pattern that ensures a class has only one instance, easily detectable due to its use of a public class with a private constructor. Builder and Prototype represent mid-level patterns that abstract object construction. Abstract Factory and Factory Method, on the other hand, introduce higher levels of abstraction and flexibility, which make them more complex to identify. Furthermore, these patterns are frequently used in real-world industrial applications (Mehra 2021).

Table 3 shows the DPs utilized in our study from P-MARt projects. The last column in the table represents the overall count of Java modules/classes corresponding to each DP. The table reveals that Builder DPs have the least number of file/module (9), while the Prototype DP has the maximum number of file/module counts (34). We also utilized programs from these projects that do not implement any DPs, selecting a total of 50 such examples to act as negative examples for each DP. For the validity of our negative examples, we excluded any files listed in the XML as having a pattern or anti-pattern, selecting 50 files that did not implement any DPs. This process allowed us to identify programs that could act as negative examples, as

Table 3 Design Patterns overview, which is collected from different projects from P-MARt repository

Design Pattern	Q-UML	Lexi	JRfac	NB	Junit	JHD	M-XML	Nutch	PMD	# DPs
Abstract Factory	NA	NA	NA	14	NA	NA	2	NA	N/A	16
Builder	1	3	4	NA	NA	NA	NA	NA	1	9
Factory Method	NA	NA	3	NA	NA	1	6	NA	NA	10
Prototype	6	NA	1	1	26	NA	NA	NA	NA	34
Singleton	NA	2	11	2	2	2	3	1	1	24

*Note** NA indicates DP not present

they were not associated with the patterns. We have shared these selected negative examples in our repository. The examples range from 21 to 399 LOC, with the majority sourced from JRefactory v2 (35 examples) and QuickUML (8 examples).

For each DP, we grouped all programs that implemented the specific pattern, such as Singleton, as positive examples. For the negative group, we selected Java programs that did not implement any DPs collected from various repositories as mentioned in Sect. 5.2, along with programs that implemented patterns other than Singleton. This process was repeated for each DP. To address the class imbalance issue, we kept the proportion of negative examples between 40% and 60% in each experiment's fine-tuning set, helping to prevent skewed performance and overly optimistic results. The negative examples from the negative group were selected randomly during each experiment. We followed this strategy for all DPs.

5.4 Evaluation Metrics

To evaluate the performance of our DPR approach with different LLMs, we used different metrics: t-SNE visualisation, precision, recall, and F1-score. These measures assess the models' effectiveness by considering different aspects of the classification outcomes.

1) *t-SNE (t-Distributed Stochastic Neighbor Embedding) plots:* We used t-SNE to visualize the embeddings extracted from LLMs. t-SNE is a dimensionality reduction technique that is effective for visualising high-dimensional data in a lower-dimensional space, typically 2D or 3D. The importance of t-SNE plots facilitates the interpretation of complex embeddings. By observing the relationships between program embeddings of different programs in the t-SNE plots, we can see patterns, clusters, and similarities among different DPs.

2) *Clustering metrics:* We used the Silhouette Score (Rousseeuw 1987) and Davies-Bouldin Index (Maulik and Bandyopadhyay 2002), to get insight from clustering made by embedding extracted from LLMs.

Silhouette Score: This metric measures the overlap and separation of clusters formed by the embeddings, ranging from -1 to +1. A score close to +1 indicates that the program samples labeled with the DP are well-clustered, make clear groups, and are far from neighboring DP clusters. If a score close to -1 suggests that the programs labeled with the DP may have been assigned to the wrong cluster. A score of zero indicates that the programs lie on or very close to the decision boundary between two neighboring clusters.

Davies-Bouldin Index: This metric quantifies the average similarity ratio of each cluster with the cluster that is most similar to it, ranging from 0 to infinity. Lower values of this index indicate better clustering, which means higher separation between clusters. A higher value indicates that clusters are more similar to each other and less distinct, indicating poorer clustering quality.

3) *Precision:* This metric indicates the accuracy of positive predictions. Precision in our study represents the model's ability to accurately identify Java programs that implement a specific DP. A high precision indicates that it is likely correct when the model predicts a program implementing a particular DP.

4) *Recall:* This metric shows the ability of the model to capture all instances of Java programs that implement a specific DP. A high recall indicates that the model effectively identifies most of the true instances.

5) *F1-score:* It is the harmonic mean of precision and recall. It considers both false positives and false negatives, making it suitable for tracking misclassification.

6) *Other metrics* We used the pair-wise t-test, which is a statistical hypothesis test to determine whether a significant difference exists between the means of two related groups or samples. The paired t-test reports for variation within the data by considering the paired nature of the observations. We conducted this test to determine the performance difference between state-of-the-art methods and LLMs-based approaches across DPs. We also use $(\frac{F-score_{max}-F-score_{min}}{F-score_{mean}})$ to calculate the variability of an LLM across different DPs. We used standard deviation, a common method to investigate the selection of LLMs and DPs in terms of variability.

To explore the consistency of predictions made by LLMs combined with k-NN classification (RQ-3), we analyzed how these models perform across DP-implemented program files when different sets of negative examples are selected.

5.5 Baseline State-of-the-art Methods

We compared our method against five well-known state-of-the-art methods, inspired in part by a recent survey (Moreira et al. 2022). They were selected based on various criteria: dataset used, programming language, performance, use of relevant DPs, and utilization of P-MARt for evaluation.

The five selected methods are SparT by Xiong and Li (2019), DPD by Tsantalis et al. (2006), DPF by Bernardi et al. (2014), RaM by Rasool and Mäder (2014), and GEML by Barbudo et al. (2021). These models have consistently shown a range of performances, achieving precision, recall, and F1-scores ranging from 0.43 to 1.0, 0.64 to 1.0, and 0.54 to 0.97, respectively.

5.6 Experimental Setup

We use the Python to conduct experiments, Pytorch framework, and different Python libraries such as Transformers, Pandas, Numpy, Scikit, etc. The experiments were conducted on the free version of Google Colab GPU; it has NVIDIA Tesla T4 (16GB RAM) CPU, and virtual CPUs System RAM: 12GB.

We employed Code2Vec, which we pre-trained on two automotive codebases. The base version of Code2Vec[9] used in our study is derived from Code2Seq (ICLR 2019 Alon et al. 2019), which utilizes LSTMs for encoding and decoding sequences. For the remaining state-of-the-art models, CodeBERT, CodeT5, CodeGPT, and RoBERTa, we used pre-trained versions from HuggingFace's model hub with default hyperparameters unless otherwise specified. Details of these models is specified in Table 4. The table provides an overview of the various models and their pre-trained versions, with the last column describing the specific

[9] https://github.com/tech-srl/code2vec

Table 4 Experimental setup and model parameters

Model	Version	Pre-training Dataset	Hyperparameters
Code2Vec	Custom Pre-trained	AndroidAuto, Genivi	Embedding Dim: 200, LSTM Encoder, 30 Epochs
CodeBERT	`huggingface/roberta-base`	Pre-trained (multi-code)	Learning rate: 2e-5, Batch size: 16, Others: default
CodeT5	`huggingface/t5-small`	Pre-trained (multi-code)	Learning rate: 3e-5, Batch size: 16, Others: default
CodeGPT	`huggingface/gpt2`	Pre-trained (multi-code)	Learning rate: 2e-5, Batch size: 16, Others: default
RoBERTa	`huggingface/roberta-base`	Pre-trained (multi-code)	Learning rate: 1e-5, Batch size: 16, Others: default

hyperparameters used for each model. Given the limited number of programs implementing each DP (e.g., nine programs for Builder), we conducted each experiment ten times to ensure the reliability of our findings and mitigate potential random biases. Then, we calculated the mean of each performance measure, i.e., precision, recall, and F1-score, over the five runs.

In our experimental setup, we opted for k=3 in the k-NN model for several reasons: 1) this choice balances bias and variance, allowing the model to capture local patterns without using excessive sensitivity to noise (Mucherino et al. 2009). 2) we do not have that many examples of positive or negative classes in the data. 3) the computational efficiency of the k-NN algorithm is maintained with a smaller "k". Given the limited number of programs for each DP, we adopted an 80-20 split for training and testing datasets.

6 Results

This section addresses the research questions framed in Sect. 1. Each subsection provides insights into individual RQs, presenting findings and justifications.

6.1 LLM Performance in DPR

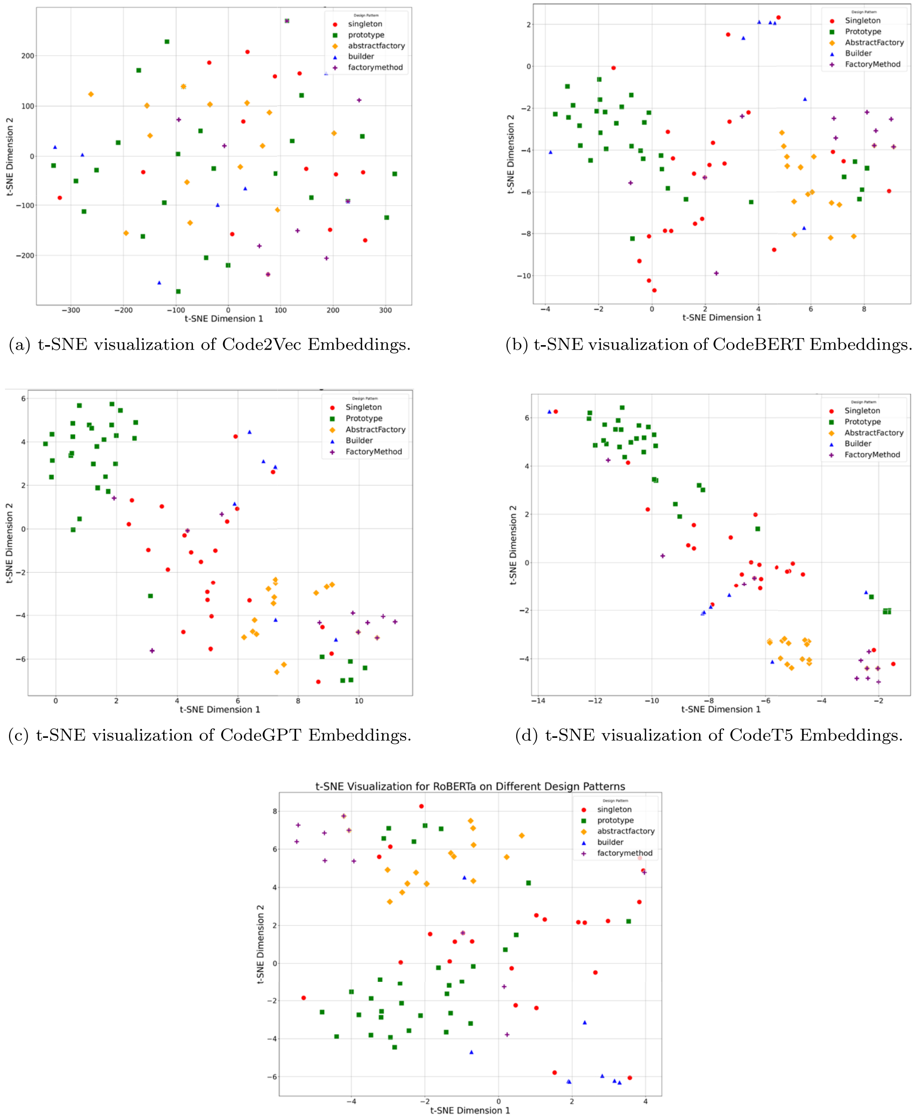
Fig. 3a shows the t-SNE visualization produced by Code2Vec for different Java programs with five different DPs. The scatter and the lack of linearly separated groups indicate that the Code2Vec model's embedding space does not form clear clusters/groups based on the DPs. These results may indicate, for example, that the instances of each DP have high variability, making it challenging for the model to create distinct clusters or groups.

The t-SNE visualization of the CodeBERT model is shown in Fig. 3b. Unlike the scattered points of the Code2Vec visualization, CodeBERT has relatively clear groups of Java programs implementing specific DPs. The green square dots representing Prototype implementations appear to form a distinct cluster, suggesting that CodeBERT may effectively capture information for Prototype instances. Similarly, well-defined groups of the Abstract Factory and Factory Method implemented programs indicate the model's ability to differentiate between examples of these DPs. Singleton programs make one big group (red dots), but a few dots are scattered in 2-D space and require further investigation. These findings suggest that CodeBERT's embeddings have better discriminative ability, especially for Prototype, Abstract Factory, and Factory Method DP, compared to Code2Vec.

The t-SNE visualizations for CodeGPT, CodeT5, and RoBERTa are shown in Fig. 3c, d, and e respectively. CodeGPT and CodeT5 exhibit groups, with only a few instances scattered outside their respective clusters. Both models form well-defined clusters of Builder DP instances despite the small number of examples (9), showing their ability to capture structural similarities. RoBERTa displays a higher scatter for the Builder pattern DP but forms clusters for other DPs.

These t-SNE diagrams illustrate that CodeGPT, CodeT5, and RoBERTa demonstrate relatively better clustering and discrimination capabilities, compared to the other LLMs. The variations for each model are due to each LLM's different architectural and training data specifics; they also indicate the significant impact of the choice of language model on the effectiveness of DPR.

To investigate further, we evaluated the overall Silhouette Score and Davies-Bouldin Index for each LLM as shown in Table 5. In our results, the majority of models, Code2Vec, CodeBERT, and CodeGPT, showed negative Silhouette Scores, as shown in the second columns of



(a) t-SNE visualization of Code2Vec Embeddings.

(b) t-SNE visualization of CodeBERT Embeddings.

(c) t-SNE visualization of CodeGPT Embeddings.

(d) t-SNE visualization of CodeT5 Embeddings.

(e) t-SNE visualization of RoBERTa Embeddings.

Fig. 3 t-SNE visualizations of different embeddings for 93 Programs with Five Different Design Patterns

Table 5, which indicates that the clusters formed by the embeddings are not well-separated, which is aligned with the results from t-SNE of these three LLMs. CodeBERT had the lowest score (−0.11), indicating imperfect clustering performance, while CodeT5 achieved the highest score (0.05), showing some overlap among clusters, which is aligned with the t-SNE of CodeT5.

The Davies-Bouldin Index value is shown in the third column of Table 5; this metric evaluates the average similarity ratio of each cluster with the cluster that is most similar to it. The results show that CodeT5 has the lowest value (2.6611), indicating it produced more

Table 5 Clustering performance metrics for five LLM models

LLM Model	Silhouette Score	Davies-Bouldin Index
Code2Vec	−0.0479	3.1738
CodeBERT	−0.11	6.83
CodeGPT	−0.04	2.89
CodeT5	0.05	2.6611
RoBERTa	0.0023	3.3818

distinct clusters than the other LLMs. In contrast, CodeBERT yielded the highest index (6.83), reflecting higher similarity among clusters, which aligns with its negative Silhouette Score. These clusters are formed after the dimensional reduction of high-dimensional embedding vectors, which could result in the loss of some information relevant to DP clustering.

We show results including precision, recall, and F1-score for each DP and LLM in two Tables 6 and 7. Still, row-wise and column-wise comparisons (e.g., blue or bold text) were made considering both tables together to ensure consistency in analysis. These two tables summarize the performance metrics for each LLM across different patterns; RoBERTa is the top performer, producing the highest mean precision, recall, and F1-scores at 0.92, 0.89, and 0.91, respectively. The table also reports the standard deviation for each result. Despite RoBERTa's training on extensive text, which could include knowledge about DPs, semantic and syntactic information in programs, and contextual details within code, the results are unexpected. This is as RoBERTa is not explicitly intended to understand source code, but language in general. This result may be due to its robustness to code noise and variability. CodeGPT also shows high prediction performance; it has the second-best performance with mean precision, recall, and F1-scores of 0.79, 0.79, and 0.79, respectively.

Tables 6 and 7 also report the mean for each performance metric across the five LLMs. RoBERTa and CodeT5 exhibit a high standard deviation, especially for DPs with fewer examples, such as Builder and Factory Method. In contrast, patterns with more examples, like Prototype, display lower variation. On the other hand, CodeGPT consistently shows low standard deviation across all patterns, indicating more stable performance regardless

Table 6 Performance metrics of design pattern recognition using Code2Vec and CodeBERT

DP	Code2Vec			CodeBERT		
	Pr.	*Re.*	*f-s.*	*Pr.*	*Re.*	*f-s.*
Abstract Factory	0.58±0.22	0.65±0.23	0.61±0.20	**0.81**±0.01	**0.81**±0.01	**0.81**±0.01
Builder	**0.82**±0.06	0.82±0.16	**0.81**±0.07	0.80±0.16	0.77±0.15	0.78±0.15
Factory Method	0.71±0.04	**0.85**±0.11	0.77±0.09	0.79±0.04	0.78±0.08	0.78±0.10
Prototype	0.64±0.04	0.69±0.08	0.67±0.05	0.78±0.23	0.78±0.10	0.78±0.16
Singleton	0.65±0.17	0.61±0.10	0.62±0.01	0.73±0.04	0.72±0.12	0.73±0.07
Mean	0.68	0.72	0.70	0.77	0.78	0.77

(**Note*:** *Pr.*, *Re.*, *f-s* are precision, recall, and F1-score, respectively. Bold blue text values indicate the highest values for each performance measure in row-wise comparisons across different DPs. The gray cell values with bold text present the maximum values of each performance measure when comparing different LLMs, representing column-wise comparisons. Cells with a gray background and bold blue text denote values simultaneously the maximum in row and column-wise comparisons. A green cell with bold text also represents the maximum mean values across all three performance measures. Here "±" indicated the standard deactivation for each performance measure)

Table 7 (Table continued) Performance comparison for CodeGPT, CodeT5, and RoBERTa

DP	CodeGPT			CodeT5			RoBERTa		
	Pr.	*Re.*	*f-s.*	*Pr.*	*Re.*	*f-s.*	*Pr.*	*Re.*	*f-s.*
Abstract Factory	0.87±0.02	0.89±0.03	0.88±0.03	**0.84**±0.11	**0.92**±0.06	**0.88**±0.08	**0.93**±0.17	**0.93**±0.16	**0.92**±0.11
Builder	0.77±0.06	0.77±0.05	0.77±0.05	0.83±0.07	0.67±0.24	0.72±0.24	**1**±0.26	**1**±0.20	**1**±0.19
Factory Method	0.78±0.07	0.79±0.07	0.79±0.07	0.72±0.21	0.67±0.11	0.70±0.11	**0.87**±0.32	0.77±0.37	**0.82**±0.31
Prototype	**0.87**±0.02	**0.87**±0.02	**0.87**±0.02	0.82±0.10	0.85±0.12	0.83±0.08	**0.88**±0.06	0.84±0.24	0.86±0.17
Singleton	0.68±0.01	0.70±0.01	0.70±0.01	0.82±0.10	0.58±0.14	0.68±0.11	**0.92**±0.16	**0.90**±0.26	**0.91**±0.17
Mean	0.79	0.79	0.79	0.81	0.73	0.76	**0.92**	**0.89**	**0.91**

of the dataset size. Code2Vec and CodeBERT demonstrate moderate standard deviation, particularly for patterns with fewer examples like Builder, where the limited data likely impacts fine-tuning impact. This variation could be attributed to the sensitivity of certain language models to the size and diversity of the fine-tuning examples.

The mean F1-score of different LLMs on five DPs is summarized in Table 8. The gray cell with bold text indicates the model that produces the highest F1-score across all patterns in each column. Similarly, blue bold text in each row highlights the maximum F1-score any model achieves on each pattern. The table also exhibits standard deviation in the last column (μ) to investigate variability across patterns and LLMs.

Reading from left to right, Code2Vec produces the maximum F1-score (0.81) for the Builder pattern but has the lowest performance for the Abstract Factory pattern (0.61). The model results show a standard deviation of 0.081, indicating higher variability in performance across different DPs. CodeBERT demonstrates the best and worst F1-scores for the Abstract Factory and Singleton patterns, respectively, with values of 0.81 and 0.73, and 0.025 standard deviations, showcasing medium variability in detecting DPs among all LLMs.

The CodeGPT model achieves the maximum and minimum F1-scores of 0.88 and 0.68, respectively, for Abstract Factory and Singleton DPs, and a standard deviation of 0.66. CodeT5 presents the best and worst F1-scores for the Prototype and Singleton patterns, with values of 0.83 and 0.68, respectively. The standadard deviation is 0.082, higher than CodeBERT and RoBERTa.

RoBERTa achieves the highest F1-score (1) for the Builder DP but the lowest (0.78) for the Factory Method pattern, with a 0.074 standard deviation, suggesting relatively high variability in detecting DPs compared to CodeBERT but lesser than Code2Vec.

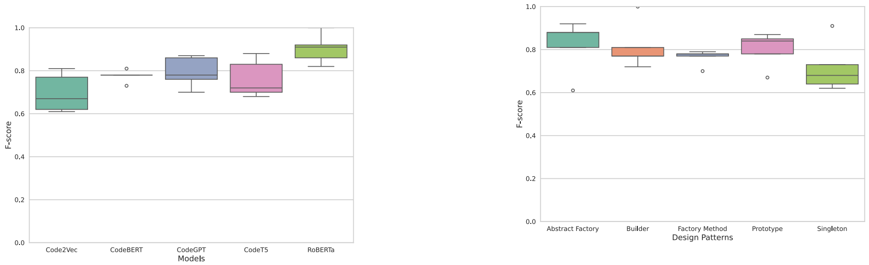
Finally, when comparing performance variation, CodeBERT and CodeT5 have the minimum and maximum standard deviations, respectively, indicating that CodeBERT is the most reliable.

We also investigated the variation in the F1-score during DP detection visually for each LLM as shown in Fig. 4. These boxplots show performance variability across multiple runs and experimental settings. The box plot in Fig. 4a shows the variation in F1-scores across DPs for the five LLMs across different experimental settings. Code2Vec and CodeT5 exhibit an extensive range of F1-scores, from 0.61 to 0.81 and 0.68 to 0.88, respectively. The inherent complexities of certain DPs could cause this broad range. Conversely, CodeBERT consistently produces F1-scores within a narrower range (0.73 to 0.81), indicating stable

Table 8 Combined F1-scores of LLMs utilizing k-NN for DPR task

DP	Code2Vec	CodeBERT	CodeGPT	CodeT5	RoBERTa	Mean	μ
Abstract Factory	0.61	**0.81**	**0.88**	**0.88**	**0.92**	**0.81**	0.111
Builder	**0.81**	0.77	0.77	0.72	**1**	0.80	0.043
Factory Method	0.77	0.78	**0.79**	0.70	0.78	0.76	**0.014**
Prototype	0.67	0.78	**0.87**	0.85	0.84	0.80	0.032
Singleton	0.62	0.73	0.70	0.68	**0.91**	0.71	0.043
Mean	0.70	0.77	0.79	0.76	**0.91**	NA	NA
μ	0.081	**0.025**	0.066	0.082	0.074	NA	NA

Note*: The bold text in each gray cell indicates the maximum F1-score produced by each LLM (column-wise comparison). The bold blue text value indicates the maximum F1 score across LLMs (row-wise comparison). μ indicates the Standard Deviation, and the green cell with bold text indicates the minimum standard deviation



(a) F1-score range comparison for each LLM.

(b) F1-score range comparison for each Design Pattern.

Fig. 4 F1-score range comparisons using boxplots for LLMs and Design Patterns

performance across various DPs. The minimum range observed for CodeBERT suggests its effectiveness remains relatively consistent. RoBERTa yields an intermediate F1-score range from 0.78 to 1, as shown by the lime green box in Fig. 4a.

To compare the performance of different LLMs for DPR statistically, we conducted t-tests on the F1 Scores obtained from each model across various DPs. The null hypothesis (H_0) in is that there was no significant difference in the performance of the compared LLMs. This can be represented as:

$$H_0 : \mu_1 = \mu_2$$

Where: μ_1 and μ_2 represent the F1-scores means of the compared LLMs.

The alternative hypothesis (H_1) was that there is a significant difference in the performance of the compared LLMs, which can be represented as:

$$H_1 : \mu_1 \neq \mu_2$$

Where: The significance level (α) for the t-tests was set at 0.05. A p-value less than 0.05 means that there is a significant difference in performance between the compared LLMs, while a p-value greater than 0.05 indicates that there is no significant difference. After the t-test, we found that the t-statistic and p-value for each DP comparison were calculated for Code2Vec, CodeBERT, CodeGPT, CodeT5, and RoBERTa.

Across all DPs, the p-values for each LLM comparison exceeded the significance level of 0.05, indicating no significant difference in performance between LLMs. Additionally, we combined the results to compare the overall performance of the LLMs across all DPs. The t-statistic values and p-values for each LLM are shown in Table 9.

RQ-1. Summary: RoBERTa has the highest F1 Score, but it has a reasonably high standard deviation. CodeGPT has comparable performance with two other LLMs (CodeBERT, CodeT5), but has a low standard deviation, which indicates more stability. Thus, LLM selection can depend on a users' desired trade-off between performance and consistency. However, despite the differences indicated by our visual and quantitative results, we find that there is no statistically significant difference in the performance across LLMs.

Table 9 Results of t-tests comparing the performance of different large language models

Large Language Model	t-statistic	p-value	Conclusion
Code2Vec	0.6809	0.5027	There is no significant difference in performance
CodeBERT	0.8969	0.3791	There is no significant difference in performance
CodeGPT	−0.0519	0.9591	There is no significant difference in performance
CodeT5	0.3118	0.7580	There is no significant difference in performance
RoBERTa	−1.9610	0.0621	There is no significant difference in performance

6.2 Design Patterns Easily Detectable by LLMs

In this section, we re-examine our previous results from the DP perspective to understand which patterns are easily detectable (RQ-2).

Figure 4b shows the box plot, which indicates the range of the F1-scores for every DP. The figure shows that there are outliers in all DPs. Abstract Factory has the maximum variance, from 0.61 to 0.92, followed by Singleton, which has a range from 0.62 to o.91. This indicates that when using most LLMs in different experimental settings, it was difficult to classify these two patterns correctly. Whereas the Factory Method is easy to detect by most of the LLMs in different settings as it has the smallest range in the box plot, from 0.77 to 0.88, as shown in Fig. 4b.

The last column of Table 8 shows each DP's standard deviation for the F1-score. The table indicates that the Factory Method DP exhibits the lowest standard deviation (0.014 in the lime green cell), followed by the Prototype (0.032), suggesting that LLMs consistently detect these patterns. Conversely, the Abstract Factory pattern exhibits the maximum standard deviation (0.111), indicating more difficulty for LLMs in accurately identifying implemented design DPs. The results shown by the box plot and table is aligned.

RQ-2. Summary: Builder, Abstract Factory, and Singleton are patterns easily detected by LLMs, each achieving more than the mean F1-score of 90%. Factory method and Abstract Factory have the highest and lowest variability, respectively.

6.3 DPR Performance Compared to the State-of-the-art

In this section, we compare our performance with state-of-the-art DPR approaches. Tables 10, and 11 compare precision and recall, respectively, to values from the the state-of-the-art, while Table 12 compares the overall F1-scores. The tables highlight the maximum and second maximum performance metrics values in bold within gray and green cells.

The state-of-the-art method (DBF) achieves the highest precision for detecting the Abstract Factory (0.97) and Factory method (0.96) patterns, as shown in Table 10. The RoBERTa model produces the second-highest precision of 0.93%. For the Builder DP, the maximum (1) and second maximum (0.83) precision values were obtained from RoBERTa, and CodeT5,

Table 10 Precision comparison of LLMs versus state-of-the-art methods

Design Patterns	Large Language Models + k-NN					State-of-the-art				
	Code2Vec	*CodeBERT*	*CodeGPT*	*CodeT5*	*RoBERTa*	*SparT*	*DBF*	*RaM*	*DPD*	*GEML*
Abstract Factory	0.58	0.81	0.86	0.84	**0.93**	N/A	**0.97**	0.50	N/A	N/A
Builder	0.82	0.80	0.77	**0.83**	**1**	N/A	0.73	0.52	N/A	N/A
Factory Method	0.71	0.79	0.78	0.72	**0.87**	N/A	**0.96**	0.51	0.64	0.81
Prototype	0.64	0.78	0.87	0.82	**0.88**	0.86	0.43	0.51	**1**	N/A
Singleton	0.65	0.73	0.68	0.82	0.92	N/A	**0.94**	0.54	**1**	**0.94**
Mean	0.68	0.77	0.79	0.81	**0.92**	N/A	**0.82**	0.51	N/A	N/A

Note*: In a gray cell, bold text denotes the highest precision value for a DP, while in a lime-colored cell, bold text indicates the second-highest precision value

Table 11 Recall comparison of LLMs versus State-of-the-art methods

Design Patterns	Large Language Models + k-NN					State-of-the-art				
	Code2Vec	*CodeBERT*	*CodeGPT*	*CodeT5*	*RoBERTa*	*SparT*	*DBF*	*RaM*	*DPD*	*GEML*
Abstract Factory	0.65	0.81	0.86	0.92	**0.93**	N/A	**0.98**	0.64	N/A	N/A
Builder	**0.82**	0.77	0.77	0.67	**1**	N/A	0.73	0.61	N/A	N/A
Factory Method	**0.85**	0.78	0.78	0.67	0.77	N/A	0.73	0.62	N/A	**0.88**
Prototype	0.69	0.78	0.87	0.83	0.84	**0.94**	0.73	0.64	**1**	N/A
Singleton	0.61	0.72	0.70	0.58	0.90	N/A	0.82	0.63	**1**	**0.95**
Mean	0.72	0.78	**0.79**	0.73	**0.89**	N/A	0.78	0.63	N/A	N/A

Note*: In a gray cell, bold text denotes the highest recall value for a DP, while in a lime-colored cell, bold text indicates the second-highest recall value

Table 12 The F1-score comparison of LLMs versus state-of-the-art methods

Design Patterns	Large Language Models + k-NN					State-of-the-art				
	Code2Vec	*CodeBERT*	*CodeGPT*	*CodeT5*	*RoBERTa*	*SparT*	*DBF*	*RaM*	*DPD*	*GEML*
Abstract Factory	0.61	0.81	0.88	0.86	**0.92**	N/A	**0.97**	0.57	N/A	N/A
Builder	**0.81**	0.78	0.77	0.76	**1**	N/A	0.73	0.54	N/A	N/A
Factory Method	0.77	0.78	0.79	0.78	0.78	N/A	**0.88**	0.56	N/A	**0.84**
Prototype	0.67	0.78	**0.87**	0.85	0.84	**0.89**	0.54	0.57	N/A	N/A
Singleton	0.62	0.73	0.64	0.70	**0.91**	N/A	0.85	0.58	N/A	**0.94**
Mean	0.70	0.77	0.79	0.76	**0.91**	N/A	**0.84**	0.58	N/A	N/A

Note*: In a gray cell, bold text denotes the highest F1-score value for a DP, while in a lime-coloured cell, bold text indicates the second-highest F1-score value respectively. In comparison, the highest precision obtained by any state-of-the-art method is produced by the DBF method, with a 0.73 value, showing that for the Builder pattern, the LLM approaches produce better results.

For Factory Method DPs, the maximum (0.96) and second maximum (0.87) precision were obtained by DBF and RoBERTa, respectively, as shown in the third row of Table 10. For the Prototype and Singleton DPs, the DPD method produces a perfect precision value, i.e., 1. RoBERTa produces the second maximum precision value for Prototype DP i.e., 0.88, whereas, for Singleton, DBF yields the second-highest precision (0.94). In the evaluation, the LLM-based techniques obtained the maximum precision for one out of the five DPs and the second-highest precision for the remaining four patterns. Finally, RoBERTa achieves the highest mean precision (0.92) across all DPs, as reported in the last column of Table 10. The DBF and DPD methods produce the highest precision for two DPs.

Table 11 presents the recall comparison of different LLMs combined with k-NN with existing state-of-the-art. The DBF method produces the maximum prediction metric for Abstract Factory in terms of recall (0.98), whereas the RoBERTa model produces the second maximum recall (0.93) – 0.05 difference. However, for Builder DP, a method based on LLMs produces maximum (RoBERTa) and second maximum (Code2Vec) recall values, i.e., 1.0 and 0.81, respectively. The GEML, a state-of-the-art method, produces the maximum recall value for Factory Method DP, i.e., 0.88, although Code2Vec is the second highest with a recall value of 0.85. Code2Vec was only 3.52% behind the GEML technique. The DPD method shows the highest performance on Prototype and Singleton DPs with recall value 1, as shown in the fourth and fifth rows of Table 11. SparT is the second-best performer in terms of recall, with a 0.94 value. The RoBERTa model produces a second maximum recall value of 0.91 for the Singleton pattern.

During the evaluation, the LLM-based method produces a maximum recall for one out of the five DPs while securing the second-highest recall across the other three DPs. RoBERTa achieves the highest mean recall across all DPs, with CodeGPT securing the second-highest recall value, as shown in the last row of Table 11.

The F1-scores obtained from different LLMs and state-of-the-art methods for each DP are shown in Table 12. As the Table reports, the DBF method produces the maximum F1-score for Abstract Factory (0.97), outperforming the second-highest F1-score achieved by the RoBERTa model (0.92). For Builder DP, RoBERTa and Code2Vec, both LLM-based approaches, obtained the maximum and second-maximum F1-score values of 1.0 and 0.81, respectively. RoBERTa shows better performance over Code2Vec. While state-of-the-art methods DBF and GEML produce high F1-scores of 0.88 and 0.84 for the Factory Method DP, no LLM-based methods are among the top two performers. SparT leads in the Prototype pattern with a maximum F1-score of 0.89, while CodeGPT follows closely with a second-highest F1-score of 0.87, lagging behind the SparT method. Finally, for the Singleton pattern, GEML secures the maximum F1 score (0.94), showing a better score than the second-highest performer, RoBERTa, which achieves an F1 score of 0.91. In the evaluation, the LLM-based approaches outperform state-of-the-art methods regarding the F1-score for one out of the five DPs while achieving the second-highest F1-score in the remaining four DPs. Ultimately, RoBERTa appears as the top LLM-based method with the highest mean F1-score across all DPs, reaching 0.90, as shown in the last row of Table 2.

Overall, although we split our analysis examining both precision and recall, in addition to the combined F1-score, we do not see obvious trends, i.e., with LLM-based approaches having overall higher or lower precision or recall compared to state-of-the-art approaches.

To investigate the statistical difference between the performance of state-of-the-art DPR methods and LLM-based models, a paired t-test was conducted to compare their performance across all measures: F1-score(F), precision (P), and recall (R). The optimal performance for each DP was selected for both state-of-the-art and LLM-based methods.

For example, for the Abstract Factory DP, the highest precision achieved by the state-of-the-art method (DBF) was (0.97), and by the LM-based method (RoBERTa) was (0.92). The significance level for all tests was set to (0.05).

Our null hypotheses for precision ($H_{\emptyset_P}$), recall ($H_{\emptyset_R}$), and F1-score($H_{\emptyset_F}$) are stated as follows:

$$H_{\emptyset_P} : \mu_{\text{state-of-the-art}} = \mu_{\text{LLM}}$$

$$H_{\emptyset_R} : \mu_{\text{state-of-the-art}} = \mu_{\text{LLM}}$$

$$H_{\emptyset_F} : \mu_{\text{state-of-the-art}} = \mu_{\text{LLM}}$$

Where $\mu_{\text{state-of-the-art}}$ and μ_{LLM} are the mean performance metrics for the state-of-the-art and LLM-based methods, respectively. Table 13 displays the t-statistic (t) and p-value (p) for each performance measure. As the results indicate, each performance measure has no significant difference between the state-of-the-art and LLM-based models ($p > 0.05$). Consequently, we fail to reject the null hypotheses, suggesting no substantial difference in the performance metrics between the state-of-the-art and LLM-based methods.

RQ−2.1. Summary: For two out of five patterns, LLM-based methods produce a higher F1-score than state-of-the-art methods and the second-best performance by LLM-based methods for four patterns. The performance of LLM approaches (without fine-tuning) is comparable to state-of-the-art methods. Factory method and However, statistical analysis shows that the difference between our results and the state-of-the-art results is significant.

6.4 Pattern Instance Classification Consistency and Contributing Factors

This section will answer RQ-3 and RQ−3.1 by investigating into the consistency of correct and incorrect classifications for various instances (programs) the five GoF DPs using five LLMs. The program files are divided into two types of file examples: positive examples are files that implement specific DPs, while negative examples are those that do not implement the particular DP. For example, when looking for Singletons, a program file that implements the Singleton pattern is considered a positive example, whereas a file that does not implement the Singleton pattern, regardless of whether it implements other patterns, is classified as a negative example. We look at how many example programs were consistently and inconsistently classified across all examined patterns.

Table 13 Statistical Analysis using t-test on state-of-the-art methods versus LLM-based models

Performance Measure	t-statistic	p-value	Conclusion
Precision	−0.306	0.771	No significant difference
Recall	0.434	0.682	No significant difference
F1-score	−0.160	0.878	No significant difference

We observed that, across all experimental settings (i.e., selecting different negative examples), six out of 16 programs were consistently correctly identified as Abstract Factory DP across all five LLMs, as shown in the first row of Table 14. In contrast, consistent classification for the Builder DP was challenging, with none of the 9 programs consistently classified across all five LLMs in different experimental settings. The Factory Method pattern showed consistency for two class implementation programs out of a possible 10. Five out of 34 Prototype-implemented programs were correctly classified across all LLMs, as indicated in the fourth row of Table 14. Three programs implementing Singleton DP are consistently classified as Singleton in different settings across LLMs out of a possible 24.

We further explored partially classified programs, consistently identified as implementing DPs correctly in at least 80% of all experiments across LLMs. Conversely, we analyzed partially misclassified programs that exhibit at least 80% misclassification across all experiments, as shown in the fifth and sixth columns of Table 14. As the Table reports, for Abstract Factory, out of 16, 9 programs are partially consistently classified, while only one program is partially misclassified, indicating that LLMs can detect Abstract Factory classes in most programs. Results are similar for other patterns, which most patterns at least partially consistently classified.

Through looking at the details of specific files, we reveal potential reasons or factors contributing to the correct and incorrect prediction of programs across all LLMs. For example, we examine the two programs that implemented Factory Methods and had this pattern detected consistently in our results (in our repository, Factory Method (5), Factory Method (8)), both having no `Factory Method` keyword in the program for easy identification, but having clear instantiation mechanisms and encapsulating the creation logic for Factory Method patterns. We shows Factory Method (5) in Fig. 5a. The file follows the standard Factory Method naming convention when creating method `createDefaultResponder()`, as shown in the first arrow indicated in Fig. 5a. This method explicitly represents the creation of a `Responder` instance, which is a basic characteristic of the Factory Method pattern. Both this program, as well as Factory Method (8) incorporate common Factory Method features, such as abstract event-handling methods, listener registration, and an event-type list, which are easily visible and might be captured by all the LLMs. Thus, it is not surprising these programs are consistently classified correctly.

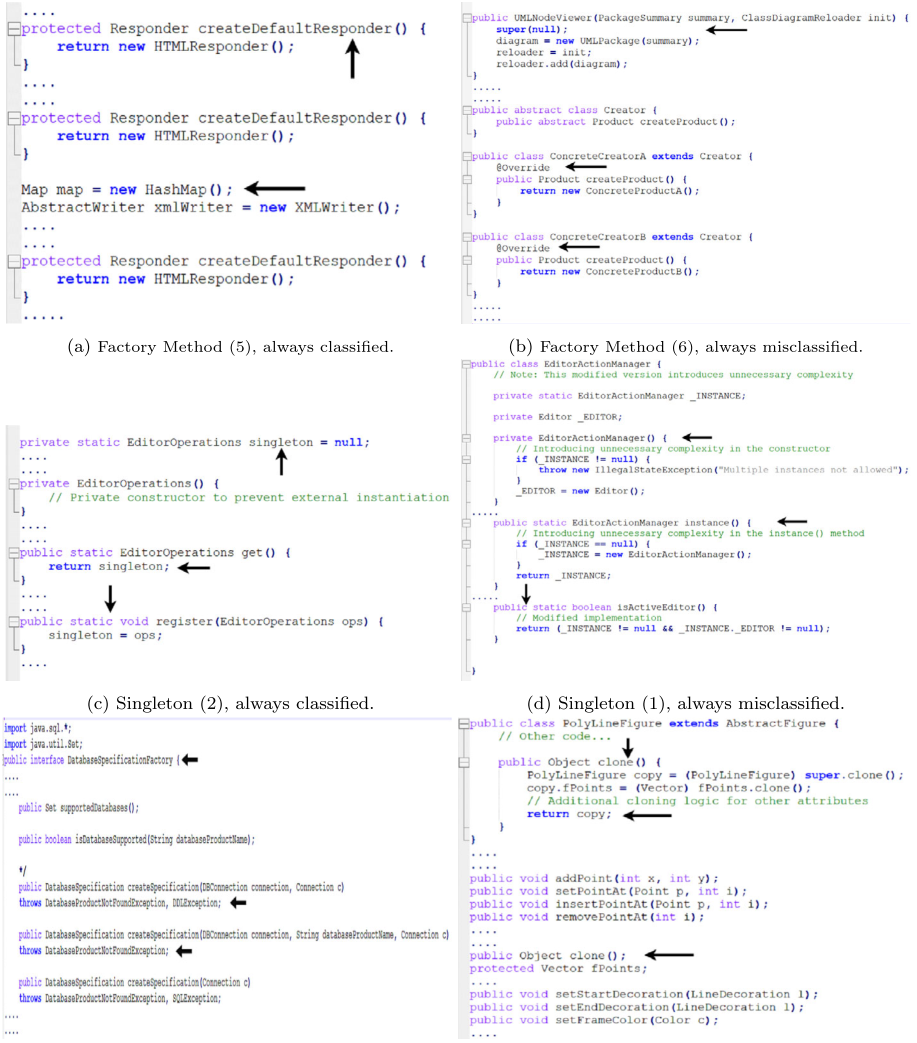
The two Factory Method implemented programs, Factory Method (6) and Factory Method (7), which are consistently misclassified by all LLMs irrespective of selected negative examples, are from the Quick UML project. Both these programs are small, 27 LOC and 109 LOC, respectively, with no keyword of `Factory Method` in the program. The Factory Method (6) program is shown in Fig. 5b, showing a simple creation of public class and straightforward implementation of objects by extending classes (highlighted by the three arrows).

In addition, we examine some Singleton examples. The three programs correctly classified as Singleton in every experiment are from the JRFactory project. The first file (Singleton (2)) is a small file that has 117 LOC, has the `private static instance` (Singleton), which indicates a Singleton pattern and single point of access. The file also provides Singleton initialization; three Singleton keywords are in the file, as shown in Fig. 5c. The second file is (Singleton (8)), which is a small Singleton implemented file of 54 LOC; it has five `Singleton` keywords and `public static PackageListFilter get()` methods, which is a clear indication of Singleton implemented file. The third file (Singleton (10)) has 293 LOC; it has a class that maintains a `private static instance` and provides a `public static method (get())` for access. However, there are eight Singleton keywords in the file. These programs used a static variable, which implies a shared, global state accessible across the application. This is a fundamental property of Singleton-implemented

Table 14 Classified and misclassified consistency of DPs implemented programs across LLMs

Design Pattern	Total implemented DP	Consistently classified	Consistently misclassified	Partially consistently classified	Partially consistently misclassified
Abstract Factory	16	5	0	9	1
Builder	9	0	0	7	2
Factory Method	10	2	2	6	0
Prototype	34	5	0	21	8
Singleton	24	3	1	10	10

Note*: The “Consistently classified” and “Consistently misclassified” columns contain the count and names of files consistently classified or misclassified across all LLMs. The "Partially consistently classified" and "Partially consistently misclassified" columns include the count and names of files that are classified or misclassified at least 80% of the time in all experiments. The full list of specific files for each count can be found in our replication package



```
....
protected Responder createDefaultResponder() {
    return new HTMLResponder();
}
....
....
protected Responder createDefaultResponder() {
    return new HTMLResponder();
}

Map map = new HashMap();
AbstractWriter xmlWriter = new XMLWriter();
....
....
protected Responder createDefaultResponder() {
    return new HTMLResponder();
}
.....
```

(a) Factory Method (5), always classified.

```
public UMLNodeViewer(PackageSummary summary, ClassDiagramReloader init) {
    super(null);
    diagram = new UMLPackage(summary);
    reloader = init;
    reloader.add(diagram);
}
.....
.....
public abstract class Creator {
    public abstract Product createProduct();
}

public class ConcreteCreatorA extends Creator {
    @Override
    public Product createProduct() {
        return new ConcreteProductA();
    }
}

public class ConcreteCreatorB extends Creator {
    @Override
    public Product createProduct() {
        return new ConcreteProductB();
    }
}
.....
.....
```

(b) Factory Method (6), always misclassified.

```
private static EditorOperations singleton = null;
....
....
private EditorOperations() {
    // Private constructor to prevent external instantiation
}
....
....
public static EditorOperations get() {
    return singleton;
}
....
....
public static void register(EditorOperations ops) {
    singleton = ops;
}
....
```

(c) Singleton (2), always classified.

```
public class EditorActionManager {
    // Note: This modified version introduces unnecessary complexity

    private static EditorActionManager _INSTANCE;

    private Editor _EDITOR;

    private EditorActionManager() {
        // Introducing unnecessary complexity in the constructor
        if (_INSTANCE != null) {
            throw new IllegalStateException("Multiple instances not allowed");
        }
        _EDITOR = new Editor();
    }
.....
    public static EditorActionManager instance() {
        // Introducing unnecessary complexity in the instance() method
        if (_INSTANCE == null) {
            _INSTANCE = new EditorActionManager();
        }
        return _INSTANCE;
    }
.....
    public static boolean isActiveEditor() {
        // Modified implementation
        return (_INSTANCE != null && _INSTANCE._EDITOR != null);
    }


}
```

(d) Singleton (1), always misclassified.

```
import java.sql.*;
import java.util.Set;
public interface DatabaseSpecificationFactory {
....
....
   public Set supportedDatabases();

   public boolean isDatabaseSupported(String databaseProductName);

   */
   public DatabaseSpecification createSpecification(DBConnection connection, Connection c)
   throws DatabaseProductNotFoundException, DDLException;

   public DatabaseSpecification createSpecification(DBConnection connection, String databaseProductName, Connection c)
   throws DatabaseProductNotFoundException;

   public DatabaseSpecification createSpecification(Connection c)
   throws DatabaseProductNotFoundException, SQLException;
....
....
```

(e) Abstract Factory (7), always classified.

```
public class PolyLineFigure extends AbstractFigure {
    // Other code...

    public Object clone() {
        PolyLineFigure copy = (PolyLineFigure) super.clone();
        copy.fPoints = (Vector) fPoints.clone();
        // Additional cloning logic for other attributes
        return copy;
    }
}
....
....
public void addPoint(int x, int y);
public void setPointAt(Point p, int i);
public void insertPointAt(Point p, int i);
public void removePointAt(int i);
....
....
public Object clone();
protected Vector fPoints;
....
public void setStartDecoration(LineDecoration l);
public void setEndDecoration(LineDecoration l);
public void setFrameColor(Color c);
....
```

(f) Prototype (19), always classified.

Fig. 5 Excerpts from Example programs, which are always classified/misclassified by all five LLMs showing notable DP features

programs and would have easily been captured by LLMs. However, the LLMs could have also looked for the Singleton keywords in the script and used them to facilitate a prediction. As these programs are small, there would be less information loss when calculating the mean embedding of a program; this could also be the reason for consistent classification.

In another example, the program Singleton (1) within the Lexi project implements a Singleton but is consistently misclassified. The file contains 1,226 LOC, belonging to the `EditorAction Manager` class, as shown in Fig. 5d. It implemented multiple classes and has no Singleton keyword in the file. Additionally, when instantiated separately, interfaces or abstract classes related to `EditorActionManager`, can be reasons for misinterpreting the code as a non-Singleton, as highlighed by the arrows Fig. 5d. The large file size can be a challenge for accurate pattern recognition, especially when considering embeddings of each line, leading to a potential loss of contextual information when combining each line embedding and then calculating the mean. The complex class hierarchy and complex interactions within the code could also be challenging for LLMs to interpret.

We also examine some Abstract Factory examples. Five Abstract Factory-implemented programs (Abstract factory (7) to Abstract factory (11)) are always correctly classified across all five LLMs; these programs are small and range from 80 LOC to 304 LOC. And no Abstract Factory example is consistently misclassified, as shown in Table 14. The file Abstract Factory (7), for example, clearly defines the interface name `DatabaseSpecificationFactory`, which is the primary property of Abstract Factory, as shown in Fig. 5e; it declares the factory method (`createSpecification`), which returns the object, conforming `DatabaseSpecification`. Similar to Abstract Factory (7), the file Abstract Factory (8) also clearly defines the interface definition and factory implementation `DefaultFactory` and methods `createMethodNode`, `create ConstructorNode`, and `createFieldNode`. All design elements are consistent, with a simple implementation of Abstract Factory DP. The other three programs also have similar and precise implementations of the Abstract Factory.

Finally, we examine some examples of Prototype implementation. Five Prototype-implemented programs are always correctly classified, irrespective of the selection of negative examples. The file Prototype (19) has 326 LOC and clearly defines the `clone (copy)` function as `PolyLineFigure` of itself, as shown in Fig. 5f; the new object is created by copying an existing object. The file also provides methods (`addPoint`, `setPointAt`, `removepointAt`, etc.) for dynamic modification of object classes, which is typically in practice for Prototype implementation. It also consistently used prototype-specific terminology such as `clone` and `copy`. Prototype (20), also consistently correctly classified, has 92 LOC, and it also has a clear definition of Prototype creation. It has a `clone ()` method implemented and dynamic creation of instances, for instance, methods for moving the figure (`basicMoveBy`), drawing the background (`drawBackground`), and drawing the frame (`drawFrame`). Similar observations were also found in the other three programs. The consistent classification of these programs as Prototype DP by all five LLMs could be because they implement dynamic creation, keywords, and customization of instance attributes, which are aligned with the principles of the Prototype pattern and easily detectable in the programs. The file size could also be a factor, as there would not be more information loss when generating the program embedding.

RQ-3, 3.1. Summary: Abstract Factory and Builder have zero consistently misclassified programs. Abstract Factory has the maximum number of files that are correctly classified. The majority of programs with each DP fall under partially classified. Instances of consistent classification and misclassification could depend on how closely the implementation of DP corresponds to standard DP features, the size of programs, keywords, and complex structures.

6.5 Computational Costs

Table 15 shows the fine-tuning time using k-NN for five LLMs across DPs. The table reports that RoBERTa consistently takes the least fine-tuning time for each DP. One potential

Table 15 Average fine-tuning time using k-NN, in seconds, for each LLM across various DPs

Design Patterns	Large Language Models + k-NN				
	Code2Vec	*CodeBERT*	*CodeGPT*	*CodeT5*	*RoBERTa*
Abstract Factory	18.41	29.24	90.51	52.12	**4.99**
Builder	15.17	43.05	38.37	44.39	**6.84**
Factory Method	14.89	36.87	39.73	51.71	**5.05**
Prototype	76.16	73.07	90.51	105.86	**5.02**
Singleton	41.16	73.22	87.90	93.02	**6.31**

Note*: In a gray cell, bold text denotes the lowest time

reason for this could be that RoBERTa is optimized for efficient embedding extraction and has smaller architectural complexity than other models like CodeGPT or CodeBERT, which may result in faster fine-tuning times. The table also reveals that all models take the longest fine-tuning time for the Prototype DP. While a larger number of examples for the Prototype might be one of the reasons, another possible reason could be the higher variability or complexity within the Prototype examples. CodeGPT exhibits the highest fine-tuning time among the models. This might be because of the large model size and the computational overhead of generating embeddings.

To understand the real-life applicability of LLMs for different practitioners, we calculated the average prediction time from the trained LLMs, as shown in Table 16. RoBERTa consistently takes the least time, less than a second, as shown in the Table, making it ideal for real-world applications. CodeT5 also performs efficiently, with prediction times less than a second. However, CodeGPT takes significantly longer, from 6.81 to 9.37 s. This higher time could be due to its larger architecture and focus on text generation. Overall, RoBERTa and CodeT5 are more suited for fast, real-time predictions. Bernardi et al. (2014) has suggested five approaches to detect DPs; the detection times for the five approaches suggested ranged between 77.992 sec to 1,402.169, which are comparatively expensive to our approaches. Xiong and Li (2019) proposed five additional methods to detect DPs, where the detection times ranged from 0.82 to 6.22 secs, which is quite comparable to our suggested approaches.

7 Discussion

Based on performance measures, RoBERTa is the best LLM in DPR without explicit pre-training, which is surprising. However, there is no significant difference between the

Table 16 The range of average prediction times (in seconds) for unseen java examples

Large Language Model	DP Prediction time
Code2Vec	0.79 to 14.02
CodeBERT	2.38 to 8.71
CodeGPT	6.81 to 9.37
CodeT5	≤ 1
RoBERTa	≤ 1

Note: This shows how long it would take for each LLM to predict an implemented DP in the real world performance of LLMs, statistically, all five LLMs are equally good at recognizing patterns. RoBERTa has the highest performance measure, so there could be some possibility that RoBERTa's training on diverse documents potentially contains knowledge about DP implementation guidelines, validation, and recognition. The results show that the model can capture semantic and syntactic information, along with the Java code's complex representations and DP structure. Our pre-trained Code2Vec shows high variance in performance across different DPs, indicating challenges in capturing certain pattern complexities. CodeBERT, on the other hand, consistently produces high F1-scores with less variability, showing its stable performance in understanding and recognizing various DPs. As CodeBERT supports code-centric training and employs a joint embedding approach, it also captures the information available in the comments of the source code and makes the prediction.

CodeGPT, based on the GPT architecture, is a general-purpose language model designed for various tasks related to natural language understanding. CodeGPT is trained on diverse text data, including programs, but not exclusively on programming languages. Maybe due to that, its performance is moderate, although it shows a contextual understanding of language and captures dependencies and relationships between patterns. Maybe its general-purpose nature may lead to varied performance across different DPs. CodeT5 uses text-to-text transfer, allowing it to handle various NLP tasks. It is trained on multimodal data that contains both text and code, which makes it potentially more suitable for our study. CodeGPT and CodeT5, being general-purpose models, may require domain-specific understanding for code-specific tasks. This could be a possible factor for moderate-level performance in DPR.

We found high variance in the prediction of various DPs; for instance, Abstract Factory pattern examples were mainly consistently classified or classified with a high degree of consistency, while nearly half of all Singleton examples were inconsistently or partially inconsistently classified by the LLMs. Part of the reason for this finding may be the size of the fine-tuning dataset; however, the number of Singleton examples in this set was relatively large.

We found a few factors affecting the prediction of DPs: file size, how the patterns are implemented (i.e., whether they use common conventions), the size of the files, use of keywords, and file complexity. The performance of LLMs may vary depending on these factors as well as the complexity of the DPs, the quality of training data, and the specific characteristics of the programming language. All these factors need to be further explored. In addition, in this work, we have focused only on creational design patterns, due to their popularity and utility. However, our results may not hold for other types of patterns. Now that we have found promising results for creational patterns, exploring the ability of LLMs to detect behavioral and structural patterns is a key area of future work.

Overall, based on our results, we recommend RoBERTa and CodeGPT as the first choices for pre-training for domain-specific DPR tasks.

The findings of this study emphasize the potential of LLMs in extending the field of DPR. By applying pre-trained models without explicit pre-training, we have shown their usefulness in extracting semantic embeddings from Java programs and applying them to DP detection tasks. The results produced by other LLMs are comparable for our selected DPs, we find no statistical differences between our results and the state of the art. However, we achieve our results (mainly) without pre-training, with fine-tuning on only a limited dataset, and without use of DP-specific detection rules. However, it is also essential to acknowledge the limitations of LLM-based approaches. Our approach explicitly avoids using pattern-specific rules for ease of use, but this also means that such rules cannot be shown to users to support classification results (as is done in Barbudo et al. (2021)). However, in our case, using k-NN allows us to see the most similar cases on which the classification was based, and manually examining these files can provide information justifying the classification. Further investigation of factors such as dataset characteristics, programming language syntax, and codebase complexity on LLM performance must also continue. The balance between the innovation application of LLMs and practical applicability is essential, ensuring these modes can effectively help software engineering practices in real-world scenarios.

While our approach addresses some of the limitations of existing methods by automating the identification of GoF patterns, we accept that our study focuses mainly on these 'standard' patterns. Specialized domain-specific patterns, such as those found in automotive software, were not evaluated in this study due to the lack of publicly available codebases. Thus, without further empirical validation, we do not claim that our approach generalizes to such patterns. Future work will be needed to explore the performance of our method in more specialized domains with smaller and more specialized codebases.

We employed k-NN for fine-tuning, a non-parametric and lazy-learning-based supervised learning model. While k-NN offers simplicity, several other limitations arise when applied to our study, especially when dealing with high-dimensional embeddings generated by LLMs. K-NN's performance can be affected by large-size embeddings. Second, when the number of positive and negative class examples is small, as in our study, k-NN's reliance on distance metrics can strengthen the impact of class imbalance. With fewer positive or negative examples, the nearest neighbors are less likely to represent the actual class distribution. However, we tried to maintain the size of positive and negative classes approximately equal (40% to 60%), which might have addressed this issue. Additionally, k-NN is computationally expensive, requiring calculating distances for every new sample. Although our study does not have many examples, maybe in the future, approximate nearest neighbor (ANN) search algorithms should be explored and used to get similar performance in less time.

The practical implications of this study benefit software practitioners and researchers. This study opens up new routes for improving software development practices by showing the feasibility of using LLMs for DPR without explicit pre-training, saving extensive time and resources. Practitioners can apply LLM-based approaches to identify DPs more efficiently and help in software maintenance and refactoring tasks. One can envision our approach integrated into an IDE, where developers can indicate the presence of domain-specific patterns in code files, and this is used by the tool to identify such patterns in other files in real-time, with occasional updates, while the developers are working.

While this study mainly evaluates the DPR performance and applicability of LLMs for detecting design patterns, the aspect of explainability remains underexplored. Explainability guides the ability of models to provide understandable reasoning or insights into classification decisions. Existing methods, such as GEML, which uses graph-based representations to generate rules that not only classify DPs but also reveal their specific reasons and variations. Such important insights can help developers understand how patterns are utilized and adapted.

In contrast, LLMs can discover complex relationships among DP elements by analyzing the contextual and syntactic of code. This capability may extend to identifying implementation variants or suggesting improvements in DPR, though these aspects require further investigation. Exploring techniques such as attention-based visualizations or saliency maps for LLMs could offer more insights into how LLMs interpret the structural and behavioral elements of design patterns, which may bridge the gap between performance and practical applicability.

Furthermore, this study's findings highlight the importance of combining advanced machine learning techniques into software development workflows to address complex challenges, such as code comprehension. Using LLMs in real-world software development can ultimately enhance productivity, code quality, and overall software maintainability, benefiting developers and end-users.

8 Threats to Validity

This section presents potential threats to the validity of our study and explains how we mitigate these threats. Our approach aligns with the framework proposed by Wohlin et al. (2012), ensuring thorough examination of various validity.

Internal Validity The internal validity threat may arise from the confounding factors influencing the relationship between LLMs and DPs. We carefully controlled experimental settings to address this threat, ensuring consistency in evaluating LLMs. We maintained a standard dataset size by adding an almost equal number (40% to 60%) of negative examples for every DP and executing experiments under similar computational environments. Due to limited transparency of the training datasets used for LLMs in our study, we cannot completely rule out the possibility of overlap with the P-Mart dataset, which could lead to potential data leakage. However, our use of LLMs is restricted only to extracting code embeddings for DPR, with classification performed by a separate kk-NN using labels associated with these embeddings. This approach mitigates direct dependency on LLMs for the classification task itself. Additionally, the embeddings represent code structures generated through self-supervised learning (e.g., masked token prediction), not task-specific exposure to DPs. This approach limits the impact of any overlap, as the LLMs would have seen code in a different context than our DPR analysis. Althouth the LLMs may have seen P-MARt code, this code would have been processed without DP labels, i.e., the DP labels for each file are stored outside the files and would not have been linked by the LLM.

External Validity External validity concerns the generalizability of our findings beyond the specific experimental environment. Our study employed a diverse set of DPs and included a selection of LLMs. We only focused on five Java-based DPs, which limits the generalizability of our findings to other types of patterns and programming languages. Patterns implemented in other languages (e.g., C++, Python) may differ in structure and behavior, leading to different results. Future work should expand the dataset to include a wider variety of DPs and programming languages to assess the models' ability to generalize across diverse scenarios.

We have compared our results over the five selected DP to DPR results from the state-of-the-art. However, we only compare these results for our selected patterns, while other works often produce results on further patterns. Thus, although we find our results are not statistically significantly different to the state-of-the-art, we do not know if this would hold for additional patterns. Further work should evaluate this finding on further patterns.

There are a few examples in the P-MARt repository for each DP, which also restricts the generalizability of our findings in to other Java projects. We also provided a replication package to facilitate the reproducibility of our results and findings. However, we cannot generalize our findings to other GoF DPs and programming languages.

Construct Validity We used suitable performance metrics for this study, including t-SNE, F1-scores, precision, and recall, and explained how they align with the objectives of DP recognition. The lack of clear clusters using t-SNE visualization does not necessarily mean that the model has yet to learn meaningful representations; reducing high-dimensional information to a 2D plot might be a primary challenge, and there could be information loss related to DP implementation. We also conducted a paired-wise t-test to measure significant differences between the performance of different LLMs and state-of-the-art methods. All these measures are widely accepted in the SE community.

Conclusion Validity The conclusion validity threats derive from errors in concluding the study. It is important to consider that some existing methods include a wider range of DPs. While our study focuses on the popular five creational patterns with practical significance, this narrower scope could affect direct performance comparisons. Expanding the scope to other types of patterns in future work will provide a more comparison with broader state-of-the-art methods. To ensure the validity of our conclusions, we performed classical statistical analyses, including t-tests, to assess the significance of observed differences. We also provided detailed descriptions of our approach and evaluation methodology and made conclusions based on the experimental evidence. Standard evaluation metrics also contribute to the authenticity of our conclusion, as these metrics are widely accepted in the SE and NLP communities.

9 Conclusion and Future Work

Our study thoroughly investigates the performance of large language models in recognizing DPs within Java code. We used a benchmark P-MARt repository and five state-of-the-art language models out of four without explicit pre-training, which are RoBERTa, CodeBERT, CodeGPT, and CodeT5. And one pre-trained model (Code2Vec) from our previous study. We used extracted embeddings of programs from LLMs and then applied the k-NN to fine-tune the DPR task. We found that the RoBERTa model produces a mean of 0.91 F1-score across all DPs, outperforming the other LLMs. CodeGPT archives the second-best positions in DP detection by yielding a 0.77 mean F1 score. The models also show variability in recognizing DPs; CodeBERT shows the smallest variability, followed by CodeGPT. However, CodeT5 and Code2Vec exhibited maximum variability, indicating the effect of pattern complexities and Java file characteristics on LLM performance. We found that the detection performance of LLMs is comparable with state-of-the-art methods; there is no significant difference between the performance of LLMs and existing state-of-the-art methods.

Moreover, domain-specific pre-training of these LLMs would make them more powerful in designing pattern recognition tools for a specific domain. The study also found that the Abstract Factory is consistently detectable by DPs. However, the Abstract Factory and Singleton patterns have high variance across LLMs. Factory Methods show the most minor variance in prediction performance across LLMs. The study also began to investigate various factors in the instance programs that could influence DPR using LLMs.

In the future, we plan to pre-train LLMs for domain-specific DPR (RoBERTa, CodeGPT), building more effective recognition of DPs. We also plan to extend our study to Python/C++ languages to investigate the effect of syntactic information in DPR. Investigating the impact of file size, pattern implementation complexity, etc., further optimizes LLMs for DPR in the practical SE domain. We also plan to explore alternative classification approaches that address the limitations of the k-NN technique. This study also finds several future research directions. Based on our findings, we suggest the following directions for exploration:

1. **Detectability of DPs:** Investigate why certain DPs, such as Abstract Factory, are more detectable than others, like Singleton. This could involve examining the characteristics and complexity of code structures associated with each pattern.
2. **Complex DPs:** Examine the detectability of more complex DPs, such as Visitor or Mediator. Understanding how the complexities of these patterns affect the recognition process and it could provide insights into improving the detection approach.
3. **Impact of pattern applications:** Further explore how the implementation of DPs affects its detectability. Such investigation could provide useful facts LLMs can help detect domain-specific patterns.
4. **Contextual factors:** Investigate how contextual factors, such as the programming language or style used, may influence the performance of LLMs in DPR.
5. **Explainability:** Our current DPR approach lacks explainability, which poses challenges in practical use and developer trust. Techniques like attention visualization, saliency maps, or rule extraction, can be used to implement explainability and also provide insights. Future research could explore combining LLMs with explainability techniques to improve their practical applicability in different industries.

These investigation areas can help build an understanding of DPR and suggestion in software engineering.

Acknowledgements The authors would like to thank the external reviewers for their detailed review, and suggestions.

Author Contributions **Sushant Kumar Pandey (corresponding author):** Research Design and Planning, Data Analysis, Writing - Original Draft, Writing - Review & Editing, and Visualization. **Sivajeet Chand:** Research Design and Planning, Data Analysis, Review & Editing. **Jennifer Horkoff:** Supervision, Research Design, and Planning, Review & Editing. **Miroslaw Staron:** Supervision, Research Design and Planning, Review & Editing. **Miroslaw Ochodek:** Research Design and Planning, Review & Editing. **Darko Durisic:** Panning, Review & Editing.

Funding This study received partial funding from CHAIR (Chalmers AI Research Center), Vinnova, Software Center, Volvo Cars, AB Volvo, Chalmers University of Technology, the University of Gothenburg, and the University of Groningen. Additionally, support was provided by the National Science Centre, Poland, under the research project titled "Source-code-representations for machine-learning-based identification of defective code fragment" (OPUS 21), registered with grant number 2021/41/B/ST6/02510.

Data Availability To support open science, we publish a full replication package on GitHub, including all the program's implemented DPs; the program does not implement DPs from P-MARt and scripts. This replication package is available at: https://github.com/sushantkumar007007/Design-Pattern-Recognition-using-Large-Language-Models-. The P-MARt repository is here https://www.ptidej.net/tools/designpatterns/.

Declarations

Ethical Approval This study did not involve human participants or animals, and therefore, ethical approval was not required.

Conflict of interest We confirm that no known Conflict of interest are associated with this publication, and there has been no significant financial support for this work that could have influenced its outcome.

Informed Consent Not applicable, as the study did not involve human participants.

References

Allamanis M, Sutton C (2013) Mining source code repositories at massive scale using language modeling. In: 2013 10th working conference on mining software repositories (MSR), pp 207–216

Almadi SH (2022) Toward investigating the violations roles of pattern grime occurrence in software design patterns: Violations roles of pattern grime. In: Proceedings of the 26th international conference on evaluation and assessment in software engineering, pp 336–341

Alon U, Zilberstein M, Levy O, Yahav E (2019) Code2vec: Learning distributed representations of code. Proc ACM Program Lang 3(POPL): 40:1–40:29. https://doi.org/10.1145/3290353

Altman NS (1992) An introduction to kernel and nearest-neighbor nonparametric regression. Am Stat 46(3):175–185

Ampatzoglou A, Frantzeskou G, Stamelos I (2012) A methodology to assess the impact of design patterns on software quality. Inf Softw Technol 54(4):331–346

Ampatzoglou A, Charalampidou S, Stamelos I (2013) Research state of the art on gof design patterns: a mapping study. J Syst Softw 86(7):1945–1964

Arshad S, Abid S, Shamail S (2022) Codebert for code clone detection: a replication study. In: 2022 IEEE 16th International Workshop on Software Clones (IWSC), pp 39–45

Barbudo R, Ramírez A, Servant F, Romero JR (2021) Geml: A grammar-based evolutionary machine learning approach for design-pattern detection. J Syst Softw 175:110919

Bernardi ML, Cimitile M, Di Lucca G (2014) Design pattern detection using a dsl-driven graph matching approach. J Softw: Evol Process 26(12):1233–1266

Bui ND, Yu Y, Jiang L (2021) Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. Proceedings of the 44th international ACM SIGIR conference on research and development in information retrieval, pp 511–521

Chand S, Pandey SK, Horkoff J, Staron M, Ochodek M, Durisic D (2023) Comparing word-based and ast-based models for design pattern recognition. In: Proceedings of the 19th international conference on predictive models and data analytics in software engineering, pp 44–48

Chen M, Tworek J, Jun H, Yuan Q, Pinto HPdO, Kaplan J, Edwards H, Burda Y, Joseph N, Brockman G et al (2021) Evaluating large language models trained on code. arXiv:2107.03374

Christopoulou A, Giakoumakis EA, Zafeiris VE, Soukara V (2012) Automated refactoring to the strategy design pattern. Inf Softw Technol 54(11):1202–1214

Ciniselli M, Cooper N, Pascarella L, Mastropaolo A, Aghajani E, Poshyvanyk D, Di Penta M, Bavota G (2021) An empirical study on the usage of transformer models for code completion. IEEE Trans Software Eng 48(12):4818–4837

Cover T, Hart P (1967) Nearest neighbor pattern classification. IEEE Trans Inf Theory 13(1):21–27

Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, Shou L, Qin B, Liu T, Jiang D et al (2020) Codebert: A pre-trained model for programming and natural languages. arXiv:2002.08155

Gamma E, Helm R, Johnson R, Johnson RE, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Pearson Deutschland GmbH

García S, Luengo J, Herrera F (2015) Data Preprocessing in Data Mining, Springer

Guéhéneuc YG, Antoniol G (2008) Demima: A multilayered approach for design pattern identification. IEEE Trans Software Eng 34(5):667–684

Haque S, Eberhart Z, Bansal A, McMillan C (2022) Semantic similarity metrics for evaluating source code summarization. Proceedings of the 30th IEEE/ACM international conference on program comprehension, pp 36–47

Heuzeroth D, Mandel S, Lowe W (2003) Generating design pattern detectors from pattern specifications. In: 18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings, pp 245–248

Hindle A, Barr ET, Gabel M, Su Z, Devanbu P (2016) On the naturalness of software. Commun ACM 59(5):122–131

Kovačević A, Luburić N, Slivka J, Prokić S, Grujić KG, Vidaković D, Sladić G (2023) Automatic detection of code smells using metrics and codet5 embeddings: a case study in c. Authorea Preprints

Kuchana P (2004) Software architecture design patterns in Java. CRC Press

Li Y, Choi D, Chung J, Kushman N, Schrittwieser J, Leblond R, Eccles T, Keeling J, Gimeno F, Dal Lago A et al (2022) Competition-level code generation with alphacode. Science 378(6624):1092–1097

Liu Y (2019) Roberta: a robustly optimized bert pretraining approach. arXiv preprint arXiv:1907.11692 364

Liu Z, Xu Y, Xu Y, Qian Q, Li H, Ji X, Chan A, Jin R (2022) Improved fine-tuning by better leveraging pre-training data. Adv Neural Inf Process Syst 35:32568–32581

Li Y, Wang S, Nguyen TN, Van Nguyen S (2019) Improving bug detection via context-based code representation learning and attention-based neural networks. In: Proceedings of the ACM on Programming Languages 3(OOPSLA), pp 1–30

Li Y, Zhang T, Luo X, Cai H, Fang S, Yuan D (2023) Do pre-trained language models indeed understand software engineering tasks? IEEE Trans Softw Eng

Lu S, Guo D, Ren S, Huang J, Svyatkovskiy A, Blanco A, Clement C, Drain D, Jiang D, Tang D et al (2021) Codexglue: A machine learning benchmark dataset for code understanding and generation. arXiv:2102.04664

Mashhadi E, Hemmati H (2021) Applying codebert for automated program repair of java simple bugs. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), pp 505–509

Maulik U, Bandyopadhyay S (2002) Performance evaluation of some clustering algorithms and validity indices. IEEE Trans Pattern Anal Mach Intell 24(12):1650–1654

Mehra L (2021) Software Design Patterns for Java Developers: Expert-led Approaches to Build Re-usable Software and Enterprise Applications (English Edition) (BPB Publications)

Moreira R, Fernandes E, Figueiredo E (2022) based comparison of design pattern detection tools. In: 29th International Conference on Pattern Languages of Programs (PLoP), pp 1–16

Mucherino A, Papajorgji PJ, Pardalos PM, Mucherino A, Papajorgji PJ, Pardalos PM (2009) K-nearest neighbor classification. Data Mining in Agriculture:83–106

Naghdipour A, Hasheminejad SMH, Barmaki RL (2023) Software design pattern selection approaches: a systematic literature review. Softw: Pract Exp 53(4):1091–1122

Nazar N, Aleti A, Zheng Y (2022) Feature-based software design pattern detection. J Syst Softw 185:111179

Pandey SK, Chand S, Horkoff J, Staron M (2023) Design patterns understanding and use in the automotive industry: An interview study. In: International conference on product-focused software process improvement, pp 301–319

Pandey SK, Staron M, Horkoff J, Ochodek M, Durisic D (2024) In: 2024 IEEE 21st International Conference on Software Architecture Companion (ICSA-C), pp 183–190. https://doi.org/10.1109/ICSA-C63560.2024.00041

Pandey SK, Staron M, Horkoff J, Ochodek M, Mucci N, Durisic D (2023) Transdpr: Design pattern recognition using programming language models. In: 2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) pp 1–7

Parthasarathy D, Ekelin C, Karri A, Sun J, Moraitis P (2022) Measuring design compliance using neural language models: an automotive case study. In: Proceedings of the 18th international conference on predictive models and data analytics in software engineering, pp. 12–21

Parvez MR, Ahmad WU, Chakraborty S, Ray B, Chang KW (2021) Retrieval augmented code generation and summarization. arXiv:2108.11601

Pettersson N, Löwe W, Nivre J (2010) Evaluation of accuracy in design pattern occurrence detection. IEEE Trans Software Eng 36(4):575–590

Qian G, Sural S, Gu Y, Pramanik S (2004) Similarity between euclidean and cosine angle distance for nearest neighbor queries. In: Proceedings of the 2004 ACM symposium on applied computing, pp. 1232–1237

Radiya-Dixit E, Wang X (2020) In: International Conference on Artificial Intelligence and Statistics (PMLR), pp 2435–2443

Rahman M, Chy MSH, Saha S (2023) A systematic review on software design patterns in today's perspective. In: 2023 IEEE 11th International Conference on Serious Games and Applications for Health (SeGAH), pp 1–8

Rasool G, Mäder P (2014) A customizable approach to design patterns recognition based on feature types. Arab J Sci Eng 39:8851–8873

Rasool G, Mäder P (2011) Flexible design pattern detection based on feature types. In: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), pp 243–252

Rousseeuw PJ (1987) Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. J Comput Appl Math 20:53–65

Scanniello G, Gravino C, Risi M, Tortora G, Dodero G (2015) Documenting design-pattern instances: a family of experiments on source-code comprehensibility. ACM Transactions on Software Engineering and Methodology (TOSEM) 24(3):1–35

Silva D, da Silva JP, Santos G, Terra R, Valente MT (2020) Refdiff 2.0: A multi-language refactoring detection tool. IEEE Trans Softw Eng 47(12):2786–2802

Svyatkovskiy A, Deng SK, Fu S, Sundaresan N (2020) Intellicode compose: Code generation using transformer. In: Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, pp 1433–1443

Tian H, Liu K, Kaboré AK, Koyuncu A, Li L, Klein J, Bissyandé TF (2020) Evaluating representation learning of code changes for predicting patch correctness in program repair. In: Proceedings of the 35th IEEE/ACM international conference on automated software engineering, pp 981–992

Tipirneni S, Zhu M, Reddy CK (2022) Structcoder: Structure-aware transformer for code generation. arXiv:2206.05239

Tsantalis N, Chatzigeorgiou A, Stephanides G, Halkidis ST (2006) Design pattern detection using similarity scoring. IEEE Trans Software Eng 32(11):896–909

Tufano M, Watson C, Bavota G, Penta MD, White M, Poshyvanyk D (2019) An empirical study on learning bug-fixing patches in the wild via neural machine translation. ACM Transactions on Software Engineering and Methodology (TOSEM) 28(4):1–29

Unger B, Tichy WF (2000) In: Proceedings of the international workshop on empirical studies of software maintenance

Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I (2017) Attention is all you need. Advan Neural Inform Process Syst 30

Walter B, Alkhaeir T (2016) The relationship between design patterns and code smells: an exploratory study. Inf Softw Technol 74:127–142

Wang S, Geng M, Lin B, Sun Z, Wen M, Liu Y, Li L, Bissyandé TF, Mao X (2023) Natural language to code: How far are we? In: Proceedings of the 31st ACM joint European software engineering conference and symposium on the foundations of software engineering, pp. 375–387

Wang D, Jia Z, Li S, Yu Y, Xiong Y, Dong W, Liao X (2022) Bridging pre-trained models and downstream tasks for source code understanding. In: Proceedings of the 44th international conference on software engineering, pp 287–298

Wang Y, Wang W, Joty S, Hoi SC (2021) Codet 5: identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv:2109.00859

Wedyan F, Abufakher S (2020) Impact of design patterns on software quality: a systematic literature review. IET Software 14(1):1–17

Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A et al (2012) Experimentation in software engineering, Springer, vol 236

Xiao Y, Zuo X, Xue L, Wang K, Dong JS, Beschastnikh I (2023) Empirical study on transformer-based techniques for software engineering. arXiv:2310.00399

Xiong R, Li B (2019) Accurate design pattern detection based on idiomatic implementation matching in java language context. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp 163–174

Yuan D, Fang S, Zhang T, Xu Z, Luo X (2022) Java code clone detection by exploiting semantic and syntax information from intermediate code-based graph. IEEE Trans Reliab

Zanoni M, Fontana FA, Stella F (2015) On applying machine learning techniques for design pattern detection. J Syst Softw 103:102–117

Zeng Z, Tan H, Zhang H, Li J, Zhang Y, Zhang L (2022) An extensive study on pre-trained models for program understanding and generation. In: Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis, pp 39–51

Zhang C, Budgen D (2011) What do we know about the effectiveness of software design patterns? IEEE Trans Software Eng 38(5):1213–1231

Zhang D, Lu G (2003) Evaluation of similarity measurement for image retrieval. In: International conference on neural networks and signal processing, 2003. Proceedings of the 2003, vol 2, pp 928–931

Zhang K, Schiele B, Yeung DY (2004) In: Proceedings of the 21st International Conference on Machine Learning (ICML), pp 941–948

Authors and Affiliations

Sushant Kumar Pandey[1,2] · **Sivajeet Chand**[1] · **Jennifer Horkoff**[1] · **Miroslaw Staron**[1] · **Miroslaw Ochodek**[3] · **Darko Durisic**[4]

✉ Sushant Kumar Pandey
s.k.pandey@rug.nl

Sivajeet Chand
sivajeet@student.chalmers.se

Jennifer Horkoff
jennifer.horkoff@gu.se

Miroslaw Staron
miroslaw.staron@gu.se

Miroslaw Ochodek
Miroslaw.Ochodek@cs.put.poznan.pl

Darko Durisic
darko.durisic@volvocars.com

[1] Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden

[2] Bernoulli Institute for Mathematics, Computer Science and Artificial Intelligence, University of Groningen, Groningen, The Netherlands

[3] Poznan University of Technology, Poznań, Poland

[4] Research & Development, Volvo Cars, Gothenburg, Sweden