Do LLMs Learn Structure or Names? A Study on the Robustness of Design-Pattern Detection to Identifier Shifts

Ichsan Budiman
ixb402@student.bham.ac.uk
School of Computer Science, University of Birmingham
Birmingham, United Kingdom

Leandro L. Minku
l.l.minku@bham.ac.uk
School of Computer Science, University of Birmingham
Birmingham, United Kingdom

Abstract

Large Language Models (LLMs) have recently achieved strong results for automated object-oriented Design-Pattern Detection (DPD). However, it remains unclear whether these gains come from learning pattern-defining structure (e.g., inheritance, delegation, object creation) or from exploiting identifier names (class, method, variable names) and pattern annotations. This distinction matters in practice as names are often incomplete, misleading, project-specific, or obfuscated. Developers may even label code with a given pattern name despite incorrect implementations. In such cases, an LLM that relies on names can become confidently wrong. Conversely, if code is already clean and explicity annotated with the correct pattern name, the detection model may not really be needed. In both scenarios, reliance on identifier names may defeat the purpose of design pattern detection. We conduct a study based on five LLM encoders and a dataset containing 1300 design-pattern instances collected from 214 Java projects to determine whether current LLM-based DPD genuinely learns structure or rely on name-based shortcuts. Our results show that models trained on raw code are dominated by identifier-driven attributions and suffer substantial performance degradation when tested on code with different identifier names. We also find that code anonymisation replacing identifier names by randomly set names increases reliance on code structure but does not fully eliminate identifier dependence, leaving a residual robustness gap under out-of-sample naming conventions. This study highlights the importance of understanding the features learned by LLMs before adopting them in practice.

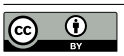
CCS Concepts

• **Software and its engineering** → *Maintaining software*; **Object oriented architectures**; • **Computing methodologies** → *Artificial intelligence*; *Machine learning approaches.*

Keywords

Design Pattern Detection, Large Language Models

ACM Reference Format:
Ichsan Budiman and Leandro L. Minku. 2026. Do LLMs Learn Structure or Names? A Study on the Robustness of Design-Pattern Detection to Identifier Shifts. In *22nd International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE '26), July 05, 2026, Montreal, QC, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3803846.3807465



ACM ISBN 979-8-4007-2584-5/2026/07
https://doi.org/10.1145/3803846.3807465

1 Introduction

Modern codebases are large, fast-changing, and highly interconnected. Keeping their structure clean is key to avoiding "code rot" and rising maintenance costs [33]. Gang of Four (GoF) design patterns help teams organise code and communicate intent [3, 12], but spotting patterns in source code at scale is hard as projects often have mixed styles, evolve over time, and spread pattern roles across many classes and files. These realities raise a practical question: can design patterns be detected automatically and reliably in code that is messy and evolving?

Existing work has attempted to detect design patterns based on machine learning (including statistical learning approaches and neural networks) [2, 4, 5, 17, 19, 24, 26, 27, 29, 30, 32]. Among the existing approaches, LLM-based approaches have recently emerged as obtaining state-of-the-art results [9, 24, 28–30, 32]. However, prior work mostly evaluates on code that is potentially annotated with the pattern names, including the use of identifiers or comments that include the pattern name. In real projects, code is noisy, names (class, method, variable) are often incomplete, misleading, or obfuscated. Methods that lean on names can look strong in clean settings but fail when names are misleading or absent. For instance, a developer might name a class using a certain design pattern name, or comment on it as implementing that pattern. However, the corresponding implementation of the pattern may be incorrect, such that the class does not actually correspond to that pattern. This may mislead the LLM if it relies on names. Meantime, if the code is fully clean and correctly annotated with the pattern's name, a Design Pattern Detection (DPD) model that relies on such names would successfully detect this pattern. However, as all classes are already correctly annotated with the patterns' names, the detection model may not really be needed. In both scenarios, reliance on identifier names may defeat the purpose of design pattern detection.

Design patterns are often structure-heavy, such that the correctness of their detection should depend on relationships (who creates, calls, inherits, delegates), not just sequence tokens. Therefore, it is unclear whether LLM-based models are really adequate for DPD, despite their state-of-the-art experimental results. This paper thus investigates what kind of features are being learned and used by LLMs to detect design patterns, and whether they truly learn structure or mostly read names. In particular, we investigate the following research questions:

(1) **RQ1:** What code features do LLM-based DPD models learn and rely on to detect design patterns?

(2) **RQ2:** To what extent does training with anonymised identifiers and removed comments encourage LLM-based DPD models to rely on name-independent (structure-oriented) features?
(3) **RQ3:** How sensitive are LLM-based DPD models to out-of-sample identifier names? Does training on anonymised data improve robustness to identifier shifts?

To answer these RQs, we study five transformer encoders (CodeBERT, GraphCodeBERT, CodeT5p, CodeGPT, and RoBERTa) based on a dataset containing 1300 design-pattern instances from 214 Java projects. All selected models have over 124M parameters. These models provide full access to their internal weights and representations, and support fine-tuning on task-specific data. These properties are fundamental requirements of our study for two reasons. First, fine-tuning the model weights enables the models to be optimised specifically for DPD. Second, we employ SHAP to analyse which tokens the models rely on when making predictions, which requires direct access to the model's internal representations. Larger closed-source models such as GPT, Gemini, and Claude do not fulfil either of these requirements, as they only permit prompt-level interaction via APIs. The study uses three variants of the DPD dataset: RAW, CLEAN, and ANON, where ANON replaces identifier names with random strings while preserving program structure and CLEAN removes comments, which sometimes make reference to the pattern names. We apply Kernel SHAP to obtain token-level attributions (RQ1). We also quantify the effect of anonymised training by comparing attribution distributions for models trained on ANON versus RAW and CLEAN (RQ2). Finally, we evaluate robustness under identifier distribution shift and missing annotations via cross-variant training and testing (RQ3).

Our results show that models trained on RAW and CLEAN are dominated by identifier-driven attributions and suffer substantial performance degradation under anonymisation. Training on ANON shifts attribution mass towards structure-oriented, but it does not fully eliminate identifier dependence, leaving a residual robustness gap under out-of-sample naming conventions. Overall, our findings warn that a model can perform well in experimental evaluation while learning fragile shortcuts (e.g., names or pattern annotations), making them potentially unreliable in real deployments.

The rest of the paper is organised as follows. Section 2 introduces preliminaries and related work. Section 3 presents our methodology, structured around the three research questions. Section 4 reports and analyses the results. Section 5 discusses threats to validity, and Section 6 concludes the paper. A replication package containing all code and data is available at https://doi.org/10.5281/zenodo.18960077.

2 Related Work

Recent work has explored LLMs as feature extractors for DPD, typically by converting code into embedding vectors and applying a lightweight classifier. Pandey et al. [28] fine-tuned several pre-trained models, including RoBERTa, CodeBERT, CodeGPT, CodeT5, Code2Vec, and used the resulting embeddings as inputs to a k-nearest neighbour DPD classifier. Their results suggest that fine tuned models can yield strong DPD performance, with RoBERTa being the best-performing model. TransDPR [29] used fine-tuned TransCoder embeddings to capture semantic regularities from C++ code, enabling pattern recognition without manual rules. DPDAtt [24] used pre-trained CodeT5+ encoder embeddings and trained three DPD classifiers (Support Vector Machines, Multilayer Perceptron, and Multinomial Logistic Regression multi-class classifiers), reporting high performance. Schindler and Rausch [32] moved to prompting GPT-3.5 & 4 directly with role-annotated examples to recover pattern roles, achieving strong precision and recall. Beyond GoF DPD, recent work applied LLM representations to domain-specific architectural patterns in Android apps. Dlamini et al. [9] combined CodeBERT embeddings with CK metrics to classify MVC/MVP/MVVM patterns in Android apps and predict whether a pattern is missing from a project.

Collectively, these LLM-based DPD approaches demonstrate that learned code representations can lead to strong DPD performance, but they are largely evaluated on code where identifier tokens remain the same in the training and test data. It is unclear whether the DPD performance achieved when using LLM representations is driven primarily by structural role configurations or by lexical cues encoded in identifier names. A strong reliance on identifier names could hinder applicability in scenarios where identifiers are unreliable or systematically altered, such as (i) cross-project deployment where naming conventions differ across teams and repositories, (ii) industrial codebases containing incomplete, inconsistent, or misleading names due to legacy evolution, rapid prototyping, or developer turnover, and (iii) settings where names are intentionally suppressed (e.g., obfuscation, anonymisation for data sharing, or privacy-preserving releases). In these cases, a detector that exploits lexical cues may perform well on RAW benchmarks yet fail to generalise when the same structural pattern is implemented under different or degraded naming conventions.

Some studies have been conducted to understand what language models learn in the context of software engineering tasks [23], as well as their sensitivity to identifier names [13]. They found that neural code comprehension models can struggle with handling complex or lengthy code [23], and can be highly sensitive to identifier names, where renaming only one identifier can cause a model to output completely irrelevant results [13]. However, these studies do not consider DPD. As a result, it remains unknown whether LLM-based DPD models rely on structure-defining patterns or exploit fragile identifier cues.

3 Methodology

3.1 Dataset and Anonymisation Setup

We adopt the DPD-F dataset of Nazar et al. [26], which provides 1,300 design-pattern instances collected from 214 Java projects, categorised into 12 patterns plus *None*. We use DPD-F because it is (i) collected from diverse, non-trivial projects rather than small curated examples, and (ii) distributed with instance-to-repository mappings which enables us to reconstruct the original code from the original repository (Allamanis and Sutton MSR'13 corpus [1]) and generate our RAW, CLEAN, and ANON variants under a controlled protocol. The patterns cover all three GoF categories: *creational* (AbstractFactory, Builder, FactoryMethod, Prototype, Singleton),

structural (Adapter, Decorator, Facade, Proxy), and *behavioural* (Observer, Visitor, Memento). The *None* category contains non-pattern and serves as a background label. Each category has 100 examples.

We will answer RQ1-3 based on multiple variants of the dataset:

- **RAW**: data with the original identifiers and comments, which may contain annotations including the pattern names.
- **CLEAN**: data without comments. This strips away natural-language hints from the comments and forces models to rely more on program structure.
- **ANON**: data without comments, and with class, method, and variable identifier names anonymised by replacing them with random strings. All occurrences of an identifier are replaced consistently across files of the same project (e.g., `singletonGetInstance` → `METH_DR4FT5`), preserving compilability and code behaviour. Anonymisation steps are specified in Algorithm 1. This setting reflects cases such as obfuscation or project-specific naming convention, and approximates cross-project testing where identifier names may not provide reliable pattern cues. It enables testing out-of-sample identifier robustness by evaluating models when identifier tokens are uninformative from those seen during training.

Algorithm 1: Identifier anonymisation for design-pattern obfuscation

Input: Java project source C; reserved identifiers $\mathcal{K}$
Output: Anonymised code C'; map $\mathcal{R}$ (reusable across files of the same project for consistent replacement)

$C \leftarrow \textsc{RemoveComments}(C)$;
$\mathcal{R} \leftarrow \emptyset$;
$C_{\text{prot}} \leftarrow \textsc{ProtectStringLiterals}(C)$;
$T \leftarrow \textsc{Parse}(C)$; // Java AST
forall *node* $n \in T$ **do**
  Extract identifiers $\mathcal{I}$ from n by type (class, method, variable) ;
  Prefix(class) ← "CLS_", Prefix(method) ← "METH_", Prefix(variable) ← "VAR_";
  forall $id, type \in \mathcal{I}$ **do**
    if $id \notin \mathcal{K}$ **then**
      $\mathcal{R}[id] \leftarrow \textsc{Prefix}(type) \cdot \textsc{RandomName}()$;
      // Each *id* maps to one *new* name, overwriting keeps consistency
$\mathcal{R}_{\text{sorted}} \leftarrow \textsc{SortByLength}(\mathcal{R}, \text{descending})$; // Avoid substring replacement (e.g. Singleton inside SingletonInstance)
forall $(id, new) \in \mathcal{R}_{sorted}$ **do**
  $C_{\text{prot}} \leftarrow \textsc{Replace}(C_{\text{prot}}, id, new)$; // Whole-code substitution: every occurrence of *id* → *new*
$C' \leftarrow \textsc{RestoreStringLiterals}(C_{\text{prot}})$;
return $(C', \mathcal{R})$;

3.2 DPD Models

Each LLM-based DPD method in this study is composed of a Pre-trained Language Model (PLM), the fine-tuning process of this PLM, and a downstream classifier.

3.2.1 Pre-Trained Language Models (PLMs). In this study, PLMs serve as the backbone for analysing whether modern DPD systems rely primarily on identifier names or on structure. We consider five widely used transformer-based encoders that differ in pre-training data and architectural bias, including both code-specific models and a general text model. These PLMs are representative of the PLM families most commonly adopted in recent LLM-based DPD pipelines that reported strong performance results:

CodeBERT [10] (125M parameters). A widely adopted code-pretrained masked-LM encoder, included as a strong baseline for code token/sequence representations.

GraphCodeBERT [15] (125M parameters). Extends CodeBERT by injecting explicit program-structure signals (e.g., data-flow) during pre-training. Included to test whether structure-aware pre-training improves robustness when identifier cues are removed.

CodeT5 [34] (220M parameters). Pre-trained with a denoising objective and a different tokenisation/training signal from BERT-style encoders. Included to assess whether denoising could yield embeddings that transfer better under identifier distribution shift.

CodeGPT [21] (124M parameters). A GPT-style model pre-trained with next-token prediction for code. Included to compare whether autoregressive pre-training changes reliance on identifiers versus structural cues.

RoBERTa (general-domain) [20] (125M parameters). Not pre-trained specifically on code. Included as a control to quantify the benefit of code-specific pre-training.

3.2.2 PLM Fine-Tuning. As with existing work [9, 28–30], we fine tune all PLMs[1]. Given a training set created from a DPD-F dataset variant[2], the fine-tuning trains the whole PLM, including its encoder with pre-trained weights, and a classifier head (using the architecture corresponding to the PLM) with randomly initialised weights. After fine-tuning, we discard the classification head and use the fine-tuned encoder as a feature extractor to compute embedding vectors, which are then fed to the downstream classifiers.

Our hyperparameter choices for fine-tuning follow standard transformer fine-tuning practice and the settings commonly reported for code-oriented PLMs. Specifically, we train for 10 epochs using AdamW, consistent with prior code-classification fine-tuning configurations and to ensure convergence under limited data [10]. We use a batch size of 16, which is a widely adopted compromise between optimisation stability and GPU memory for transformer fine-tuning. This range (16-32) is recommended in the original BERT study and is also used by later code PLMs [8, 15, 34]. We set the learning rate to 5×10^{-5}, within the typical transformer fine-tuning range (typically 2×10^{-5} to 5×10^{-5}) established by BERT and followed by subsequent transformer variants [8, 10, 20]. We fix the maximum sequence length to 512 tokens, the standard upper bound for BERT/RoBERTa-style models and the default choice adopted in CodeBERT/GraphCodeBERT fine-tuning [8, 10, 15]. Finally, we apply linear weight decay using a factor of 0.01 and dropout, which are standard regularisation settings in transformer fine-tuning to mitigate overfitting [8].

[1] Our preliminary experiments show that removing fine-tuning leads to poor results.
[2] Different variants are used for different RQs, as will be explained in Sections 3.1.

3.2.3 Downstream Classifiers. The embeddings produced by the fine-tuned PLMs are used as inputs to downstream classifiers for the task of DPD. We use four machine-learning classifiers that are competitive on high-dimensional representations, namely:

Support Vector Machine (SVM). We use an RBF-kernel SVM, which is a standard choice for modelling non-linear class boundaries and is widely used as a strong baseline for fixed embeddings [7], having also been adopted for LLM-based DPD in [24]. Following widely used practical SVM guidelines, we adopt an RBF kernel with automatic feature scaling ("`gamma=scale`") and a relatively large regularisation parameter ($C = 1000$) to prioritise fitting capacity when the embedding space is linearly inseparable [16]. Finally, we enable probability outputs via Platt scaling for calibrated posteriors used by our evaluation pipeline [31].

Multilayer Perceptron (MLP). We use a shallow MLP as a compact and standard classifier head, having also been adopted for LLM-based DPD in [24]. We adopt a single hidden layer with 100 nodes . ReLU activations are used due to their strong empirical performance [14, 25]. We train with Adam (`solver=adam`), a standard optimiser for neural models [18]. To reduce overfitting on high-dimensional embeddings, we use L2 regularisation (`alpha=1e-4`) and early stopping with a 10% validation split (`early_stopping=True`, `validation_fraction=0.1`), these are standard regularisation safeguards in practical neural classification.

Random Forest (RF). We use Random Forests as a robust non-linear ensemble methods with strong empirical performance across diverse classification problems [6, 11].This choice is also consistent with prior design-pattern detection work: Nazar et al. (DPD-F) used Random Forests in their study [26]. We set `n_estimators=100`, a common trade-off between stability and computational cost.

AdaBoost (SAMME) with RF base estimator. We also consider boosting to combine weak-to-moderate learners in a stage-wise manner, based on the SAMME multi-class formulation [35], which was also used for DPD in [26]. The base estimator is an RF with 100 trees, which increases base-learner expressiveness while preserving the ensemble's robustness [6].

3.3 Token-Level Attribution With Kernel SHAP

RQ1 and RQ2 require an understanding of which code features LLM-based DPD models rely on when detecting design patterns. This section explains the methodology for obtaining this understanding.

3.3.1 Shapley Values and Kernel SHAP. We treat "features" as the input tokens consumed by the PLM. We answer RQ1-2 by computing token-level feature attributions for each prediction using SHAP (SHapley Additive exPlanations) [22], which quantifies how much each token contributes to the model output on a given instance. For RQ1, the model is fine tuned on the full RAW variant of the dataset, and for RQ2, models are fine tuned on the full CLEAN, and on the full ANON variants. SHAP is used to analyse feature attributions for predictions on the corresponding dataset variant.

Let F be the set of features with $M = |F|$, x an instance being predicted, and f the model, which in our case is the classifier head produced during the fine-tuning process (Section 3.2.2). The Shapley value of feature $i \in \{1, \ldots, M\}$ when predicting instance x is

$$\phi_i(f, x) = \sum_{S \subseteq F \setminus \{i\}} \frac{|S|!\,(M - |S| - 1)!}{M!} \Big[f\big(x_{S\cup\{i\}}\big) - f\big(x_S\big) \Big], \quad (1)$$

which averages the marginal contribution of i over every possible subset of features $S \subseteq F \setminus \{i\}$.

Exact SHAP computation is intractable for large M, as it would require evaluating f on all 2^M possible feature coalitions. So, we use Kernel SHAP [22] to approximate Shapley values by fitting a locally weighted linear model over sampled binary masks, where the optimal weight ϕ_i are approximations of the Shapley value of the feature i for the example x. Given a mask $z \in \{0, 1\}^M$ (1 = present, 0 = absent), Kernel SHAP solves

$$\min_{\{\phi_0, \phi_1, \ldots, \phi_M\}} \sum_{z \in \mathcal{Z}} \Big(f\big(h_x(z)\big) \; - \; \phi_0 \; - \; \sum_{i=1}^{M} \phi_i \, z_i \Big)^2 \, \pi_x(z), \quad (2)$$

where $h_x(z)$ denotes a mapping from z to a valid model input and $\pi_x(z)$ denotes the Shapley kernel weighting that ensures the solution satisfies the Shapley axioms. This replaces the exact Shapley computation with a weighted regression on a sampled subset $\mathcal{Z}$ whose size we control. The cost therefore grows with $|\mathcal{Z}|$ rather than 2^M, making Shapley attribution computationally tractable.

Kernel SHAP requires a mapping $h_x(z)$ that approximates the the expected model output $v_x(S)$ when we keep the features in S fixed and marginalise out the rest. In practice, our PLM encoders cannot take inputs with missing tokens, so we approximate this expectation using complete inputs drawn from a *background distribution* [22] that represents the data distribution of X.

Kernel SHAP avoids the exact Shapley computation (which scales as 2^M for M features) by fitting a weighted linear surrogate model on a finite set of sampled coalitions (token masks) $\mathcal{Z}$. We control this cost by fixing the number of coalition samples to $|\mathcal{Z}| =$ `n_samples` per explained instance. Hence, the runtime grows with $|\mathcal{Z}|$ (and the model evaluation cost per sample) rather than 2^M, making token-level attributions feasible for long sequences.

Given a binary mask $z \in \{0, 1\}^M$ (where z_i=1 means token i is kept) and a background example b, we construct:

$$h_x(z)_i = \begin{cases} x_i & \text{if } z_i = 1 \quad \text{(token kept)}, \\ b_i & \text{if } z_i = 0 \quad \text{(token imputed)}, \end{cases}$$

so that masked tokens are replaced by background tokens rather than being removed. Evaluating $f(h_x(z))$ over sampled masks z and background points b yields Monte Carlo estimates of $v_x(S)$, which Kernel SHAP then uses in the weighted regression of Eq. (2) to estimate token attributions $\{\phi_i(x)\}$.

To keep the background both representative and computationally feasible, we build a separate background set for each dataset variant (RAW, CLEAN, ANON), ensuring that masked tokens are imputed with values drawn from the *same* variant.

3.3.2 Compute Feasibility and Parameter Choices. Kernel SHAP is computationally expensive because it evaluates the LLM many times per instance under different token masks and background imputations. On a 40 GB GPU, our runs reached 30.81 GB allocated and 36.13 GB reserved (91.4% utilisation), leaving ≈3.4 GB free GPU memory, runtime averaged 18–21 s per explained instance (172–200

instances/hour). We therefore choose settings that fit reliably within GPU memory while still producing informative explanations: k=30 background centroids, `n_samples`= 30, and sparse regularisation with `l1_reg="num_features(30)"` and multiplier 2.5. We cap the background pool at 1300 training instances per variant, fix the maximum sequence length to 512, and use batch size 1 to avoid out-of-memory failures.

3.3.3 Multi-Seed SHAP for Stability. Kernel SHAP is an approximate method and can vary slightly due to stochastic mask sampling and the weighted regression step. To account for this variance, we repeat the attribution procedure with five random seeds.

3.3.4 Two-Bucket SHAP: Syntax/structure vs. Identifiers. We answer RQ1-2 by summarising the token-level SHAP values into token buckets and attribution mass associated to (1) identifier tokens and (2) structure tokens. This enables us to check which kinds of token the fine tuned LLMs learn to most rely on.

Token Buckets. Given the token sequence produced by the model's native tokenizer, we assign each token to exactly one bucket:

- **Identifiers** $\mathcal{I}$: tokens that belong to class, method, or variable identifiers.
- **Syntax/Structure** $\mathcal{S}$: all remaining tokens (e.g., keywords such as `class`, `extends`, `implements`, `new`, operators, punctuation, literals, and control-flow tokens).

Attribution Mass. Let $\phi_i(x; m, v, s)$ be the SHAP value for token i when explaining instance x using model m trained on variant v, under SHAP random seed s. We define the absolute per-instance attribution mass in each bucket as follows:

$$\begin{aligned} S_{\text{id}}(x; m, v, s) &= \sum_{i \in \mathcal{I}} |\phi_i(x; m, v, s)|, \\ S_{\text{syn}}(x; m, v, s) &= \sum_{i \in \mathcal{S}} |\phi_i(x; m, v, s)|. \end{aligned} \tag{3}$$

To avoid differences between S_{syn} and S_{id} that are extremely small and not meaningful, we use a small tolerance threshold $\tau > 0$ (default $\tau = 10^{-6}$) to treat near-equal sums as a tie. Therefore, for a fixed SHAP seed s, the per-instance two-bucket outcome is $(u_{\text{syn}}(x; m, v, s), u_{\text{id}}(x; m, v, s)) \in \{(1, 0), (0, 1), (0.5, 0.5)\}$:

$$(u_{\text{syn}}(x; m, v, s), u_{\text{id}}(x; m, v, s)) = \begin{cases} (0.5,\ 0.5), & \text{if } |S_{\text{syn}} - S_{\text{id}}| \le \tau, \\ (1,\ 0), & \text{if } S_{\text{syn}} > S_{\text{id}} + \tau, \\ (0,\ 1), & \text{if } S_{\text{id}} > S_{\text{syn}} + \tau, \end{cases} \tag{4}$$

where S_{syn} and S_{id} denote the bucket sums in Eq. (3) computed for $(x; m, v, s)$. With multiple SHAP seeds S, we produce a final per-instance two-bucket outcome $(u_{\text{syn}}(x; m, v), u_{\text{id}}(x; m, v))$ by majority agreement across seeds (ties allowed). Pattern-level vote rates $(\bar{u}^p_{\text{syn}}, \bar{u}^p_{\text{id}})$ are then obtained by counting these outcomes over all instances of each pattern p and calculating their proportions, providing an understanding of what kind of features (structure or identifier names) the LLMs learn to rely on.

Using the bucket sums in Eq. (3), we can also define the fraction of attribution mass assigned to structure tokens:

$$\rho_{\text{syn}}(x; m, v, s) = \frac{S_{\text{syn}}(x; m, v, s)}{S_{\text{syn}}(x; m, v, s) + S_{\text{id}}(x; m, v, s) + \epsilon}, \tag{5}$$

where $\epsilon = 10^{-12}$ avoids division by zero, and $\rho_{\text{id}}(x; m, v, s) = 1 - \rho_{\text{syn}}(x; m, v, s)$. These can then be aggregated across seeds and instances of a given pattern p as follows, where S is the set of SHAP random seeds and $\mathcal{D}^p_v$ is the set of examples belonging to pattern p in dataset variant v:

$$\bar{\rho}^p_{\text{syn}}(m) = \frac{1}{|\mathcal{D}^p_v| \cdot |S|} \sum_{x \in \mathcal{D}_p} \sum_{s \in S} \rho_{\text{syn}}(x; m, v, s). \tag{6}$$

We also summarise overall reliance through the mean $\bar{\rho}_{\text{syn}}(m)$ and standard deviation $\sigma_{\text{syn}}(m)$ of the values of $\bar{\rho}^p_{\text{syn}}(m)$ across patterns. The values of $\bar{\rho}_{\text{syn}}(m)$ and $\sigma_{\text{syn}}(m)$ characterise how much attribution mass is assigned to structure tokens, and how this reliance varies across design patterns. Aggregated $\bar{\rho}_{\text{id}}(m)$ and $\sigma_{\text{id}}(m)$ can be computed similarly.

RQ1: For each of the five LLMs fined tuned on RAW, we summarise whether predictions are more identifier-driven or structure-driven by calculating these metrics using all examples from the RAW variant. To ensure a consistent SHAP feature space, the token dictionary used by the explainer is built once from a background pool sampled from RAW and then held fixed for all explanations.

RQ2: RQ2 examines which features LLM-based detectors rely under identifier anonymisation and removed comments. We conduct this analysis by repeating the process used to answer RQ1 under a matched anonymisation setting, i.e., using models fine tuned on CLEAN to compute SHAP values for predictions on CLEAN examples, and models fine tuned on ANON to compute SHAP values for predictions on ANON examples.

3.4 Predictive Pipeline and Evaluation Procedure for Cross-Dataset Robustness

RQ3 asks how sensitive LLM-based design-pattern detectors are to out-of-sample identifier names. We study cross-dataset robustness by training on a *source* dataset variant $\mathcal{D}_{src}$, and then testing on a *target* dataset variant $\mathcal{D}_{tgt}$.

3.4.1 Predictive Pipeline. We use the fine-tuned PLMs as fixed feature extractors. Given a Java source-code instance x, we can compute its embedding vector via the fine-tuned PLM encoder and use this representation as input to a downstream classifier, which outputs the predicted design-pattern label.

3.4.2 Training Process. The fine-tuning process is based on a training set $\mathcal{D}^{train}_{src}$ produced from a given dataset variant *src* (RAW, CLEAN or ANON) using the procedure described in Section 3.2.2. After the feature extractors are fine-tuned, the downstream classifier is also trained on D^{train}_{src}.

3.4.3 Evaluation Process. A model is considered robust if its predictive performance does not degrade on $\mathcal{D}^{test}_{tgt}$ compared to $\mathcal{D}^{test}_{src}$, despite it being trained solely on $\mathcal{D}^{train}_{src}$. In other words, a robust detector should maintain its performance even when the naming conventions in the test data differ from those seen during training.

We evaluate the robustness of LLM-based design-pattern detectors under naming-convention shifts using three dataset variants (RAW, CLEAN, and ANON), five PLMs, and four downstream classifiers. Algorithm 2 defines a comparable cross-variant evaluation

protocol that ensures fairness across all PLM-classifier combinations by using identical repeated stratified cross-validation (CV) splits. Because the variants are different views of the same instances, CV folds are created once per repeat on the shared row indices (stratified by the output category $\mathbf{y}$) and then reused across all variants, ensuring that fold f corresponds to the same instances in RAW, CLEAN, and ANON. This design is essential to guarantee that any observed difference in performance across variants is attributable solely to the naming-convention shift, rather than to differences in the data instances seen during training or evaluation.

Each PLM m is fine-tuned on the training split D_{src}^{train} of the source variant, then used to extract embedding vectors for both the training and evaluation variants. A downstream classifier c is fitted on the training embeddings and evaluated on the corresponding test examples across all evaluation target variants $tgt \in \mathcal{V}$, producing a train→eval performance matrix. Finally, performance metrics are averaged across folds and repeats to obtain robust estimates under each cross-variant setting. We use *stratified* 10×10 cross-validation: the data are split into 10 folds preserving class proportions (12 patterns + *None*).

Crucially, all pipeline components are held constant when comparing predictive performance for the purpose of evaluating robustness, with only the variant changing. Therefore, any observed performance change reflects sensitivity to cross-dataset distribution shift, rather than differences in the modelling procedure.

Algorithm 2: Comparable cross-variant evaluation

Input: Variants $\mathcal{V} = \{\text{RAW}, \text{CLEAN}, \text{ANON}\}$ with texts D_v and labels $\mathbf{y}$; PLMs $\mathcal{M}$; classifiers C; repeats R; folds F

Output: Comparable metrics for all $(m, c, v_{\text{train}}, v_{\text{eval}})$

$I \leftarrow \{1, \ldots, |\mathbf{y}|\}$;
; // Row indices shared by all variants
for $r \leftarrow 1$ **to** R **do**
  skf $\leftarrow$ StratifiedKFold(F, shuffle=True, seed=42+r);
  $\mathcal{F} \leftarrow \{(I_{\text{tr}}^{(f)}, I_{\text{te}}^{(f)})\}_{f=1}^{F} \leftarrow$ Split(skf, I, $\mathbf{y}$);
  // Create folds once, fold f matches across all variants
  for $f \leftarrow 1$ ***to*** F **do**
    $(I_{\text{tr}}, I_{\text{te}}) \leftarrow (I_{\text{tr}}^{(f)}, I_{\text{te}}^{(f)})$;
    forall $src \in \mathcal{V}$ **do**
      forall $m \in \mathcal{M}$ **do**
        FineTune(m, $D_{\text{src}}^{[I_{\text{tr}}]\mathbf{y}[I_{\text{tr}}]}$);
        $E_{\text{tr}} \leftarrow$ Embed(m, $D_{\text{src}}^{[I_{\text{tr}}]}$);
        forall $c \in C$ **do**
          $g \leftarrow$ Fit(c, E_{tr}, $\mathbf{y}[I_{\text{tr}}]$);
          forall $tgt \in \mathcal{V}$ **do**
            $E_{\text{te}} \leftarrow$ Embed(m, $D_{tgt}^{[I_{\text{te}}]}$);
            metrics $\leftarrow$ Evaluate(g, E_{te}, $\mathbf{y}[I_{\text{te}}]$);
            Record(r, f, m, c, src, tgt, metrics);

3.4.4 Evaluation Metrics. We compute per-class metrics and report macro-averages across classes. We report the Geometric Mean of recall and specificity (G-Mean), and specificity as the main evaluation metrics. The Matthews Correlation Coefficient (MCC) and F1-score are provided in supplementary material for reference. For all metrics, larger values are better.

4 Results and Discussion

4.1 RQ1-What Code Features Are LLMs Relying On to Detect Design Patterns?

The RAW columns in Table 1 show the reliance $\bar{\rho}$ on structure tokens per pattern and LLM model, with Figure 1 in the supplementary material further illustrating the corresponding aggregation of $\bar{\rho}_{syn}$ and $\bar{\rho}_{id}$ across patterns.

4.1.1 Global Attribution by Model. The RAW component of Table 1 shows a clear dominance of identifier features. There is high $\bar{\rho}_{id}(m)$ (and low $\bar{\rho}_{syn}(m)$) values for all five models, with CodeT5p showing the strongest dependence on identifiers and the other only slightly less. This indicates that, whenever meaningful names are available, all investigated models primarily solve the task by exploiting lexical cues rather than syntactic structure.

4.1.2 Attribution by Pattern. From Table 1, we can see that $\bar{\rho}_{syn}^{p}(m)$ is low for almost every pattern, ranging from 0.07 for CodeT5p on Prototype up to 0.23 for GraphCodeBERT on Memento. On average, identifier dominate the attribution with $\bar{\rho}_{id}(m)$ varying from 0.86 to 0.89 for different models. Table 5 in the supplementary material shows the same trend based on the pattern-level vote rates $(\bar{u}_{syn}^{p}, \bar{u}_{id}^{p})$, and further reveals that ties are uncommon (0.00–0.08). This indicates that the attribution is clearly toward one bucket, with explanations are largely name-driven.

4.1.3 Identifier-Dominated Pattern Predictions. To illustrate how identifier dominance manifests at the token level, Figures 1a and 1b plot the top token-level absolute SHAP scores for representative *Builder*, *Singleton* and *AbstractFactory* instances under a CLEAN setting[3]. For *Builder* and *Singleton*, most of the attribution mass is concentrated on repeated occurrences of the pattern name and its derivatives, with only minor contributions from surrounding syntax. Not all patterns are purely identifier-driven. For the *AbstractFactory* example, Figure 1c shows the `AbstractFactory` class name still dominates the attribution, but syntactic cues such as the presence of an `abstract` class and `public` factory methods also receive high SHAP value. Nevertheless, even in *AbstractFactory*, where tokens such as `abstract`, `class`, and `public` carry the attribution mass, the class name still accounts for the largest of the explanation. These token-level views confirm that, for many patterns, the encoder combines identifier name cues with higher-level structural hints, but the overall explanation remains dominated by identifiers.

Taken together, RQ1 shows that current LLM-style encoders behave more like name matchers than strongly structure-aware pattern detectors, a limitation that becomes critical once identifiers are out-of-sample as will be shown in RQ3 (Section 4.3).

[3]Some names appear more than once because they are used in different contexts, e.g., as class name and constructor name

Table 1: RQ1–2: Pattern-Level Syntax Reliance Ratio $\bar{\rho}_{\mathbf{syn}}(m, p)$

Pattern	CodeBERT			CodeGPT			CodeT5p			GraphCodeBERT			RoBERTa		
	RAW	CLEAN	ANON	RAW	CLEAN	ANON	RAW	CLEAN	ANON	RAW	CLEAN	ANON	RAW	CLEAN	ANON
AbstractFactory	0.15	0.16	0.46	0.14	0.17	0.47	0.14	0.18	0.49	0.15	0.19	0.42	0.13	0.15	0.45
Adapter	0.12	0.19	0.58	0.12	0.22	0.56	0.10	0.21	0.57	0.13	0.21	0.57	0.12	0.24	0.52
Builder	0.14	0.21	0.54	0.17	0.25	0.57	0.12	0.22	0.56	0.14	0.23	0.53	0.16	0.26	0.56
Decorator	0.16	0.25	0.50	0.14	0.25	0.53	0.13	0.26	0.54	0.17	0.28	0.48	0.18	0.25	0.46
Facade	0.15	0.25	0.47	0.13	0.26	0.47	0.11	0.23	0.46	0.13	0.26	0.45	0.15	0.29	0.45
FactoryMethod	0.12	0.19	0.49	0.11	0.20	0.48	0.08	0.18	0.50	0.13	0.18	0.48	0.12	0.19	0.47
Memento	0.21	0.25	0.50	0.18	0.24	0.52	0.18	0.23	0.52	0.23	0.25	0.47	0.20	0.25	0.54
Observer	0.13	0.19	0.48	0.12	0.19	0.51	0.10	0.20	0.53	0.11	0.20	0.46	0.12	0.19	0.49
Prototype	0.09	0.18	0.51	0.09	0.19	0.52	0.07	0.17	0.56	0.09	0.18	0.53	0.08	0.19	0.56
Proxy	0.15	0.24	0.47	0.16	0.29	0.50	0.12	0.28	0.51	0.16	0.28	0.44	0.16	0.26	0.48
Singleton	0.13	0.23	0.51	0.12	0.24	0.52	0.09	0.23	0.51	0.16	0.23	0.48	0.13	0.22	0.46
Visitor	0.12	0.20	0.54	0.11	0.19	0.52	0.08	0.21	0.56	0.13	0.20	0.48	0.11	0.21	0.51
$\bar{\rho}_{\mathbf{syn}}(m)$	**0.14**	**0.21**	**0.50**	**0.13**	**0.22**	**0.51**	**0.11**	**0.22**	**0.53**	**0.14**	**0.22**	**0.48**	**0.14**	**0.22**	**0.50**
$\bar{\rho}_{\mathbf{id}}(m)$	**0.86**	**0.79**	**0.50**	**0.87**	**0.78**	**0.49**	**0.89**	**0.78**	**0.47**	**0.86**	**0.78**	**0.52**	**0.86**	**0.78**	**0.50**
$\sigma_{\mathbf{syn}}(m)$	**0.03**	**0.03**	**0.04**	**0.03**	**0.04**	**0.03**	**0.03**	**0.03**	**0.03**	**0.03**	**0.04**	**0.04**	**0.03**	**0.04**	**0.04**

4.2 RQ2 - Effect Of Anonymised Training on Feature Reliance

RQ2 examines whether training on anonymised code and removed comments reduces reliance on identifier names and increases reliance on structure.

4.2.1 Overall Shift in Reliance Across Training Variants. Based on Table 1, we can see that across all model families, anonymised training produces redistribution of attribution mass away from identifiers and toward structure. Under RAW, all models show very low structure resiliance ($\bar{\rho}_{\text{syn}(m)} = 0.11 - 0.14$), indicating that prediction are primarily driven by identifier tokens. Under CLEAN, structure reliance increases ($\bar{\rho}_{\text{syn}(m)} = 0.21 - 0.22$). Wilcoxon Sign Rank tests to compare RAW to CLEAN across patterns and models lead to p-value of 1.63×10^{-11} and effect size A12 of 1.00 indicating perfectly consistent higher values for CLEAN than RAW. However, models prediction remain clearly dominated by identifier name. A large change occurs under ANON, where structure reliance rises sharply to ($\bar{\rho}_{\text{syn}} = 0.48{-}0.53$), representing a ~4x increase relative to RAW. These results indicate that identifier anonymisation strongly forces models to rely more heavily on structure. Wilcoxon Sign Rank test to compare RAW to ANON also lead to very low p-value $= 1.63 \times 10^{-11}$, and very large effect size (A12 = 1.00), indicating perfectly consistent higher values for ANON than RAW.

4.2.2 Model-Level Effects. All models follow the same direction of change in average structure reliance $\bar{\rho}_{\text{syn}}(m)$ from RAW to CLEAN to ANON, but the magnitude differs:

- CodeBERT: 0.14 → 0.21 → 0.50 ($\Delta_{\text{RAW}\to\text{ANON}} = +0.36$)
- CodeGPT: 0.13 → 0.22 → 0.51 ($\Delta = +0.38$)
- CodeT5p: 0.11 → 0.22 → 0.53 ($\Delta = +0.42$, largest shift)
- GraphCodeBERT: 0.14 → 0.22 → 0.48 ($\Delta = +0.34$, smallest shift)
- RoBERTa: 0.14 → 0.22 → 0.50 ($\Delta = +0.36$)

CodeT5p exhibits the strongest shift: it has the lowest structure reliance in RAW (0.11), indicating the highest identifier dependence, yet becomes the most structure-reliant model in ANON (0.53). In contrast, GraphCodeBERT shows the smallest increase and ends with the lowest structure reliance under ANON (0.48). Notably, although GraphCodeBERT is tied for the highest structure reliance in RAW (0.14), this advantage does not translate into stronger name-independent reliance after anonymised training.

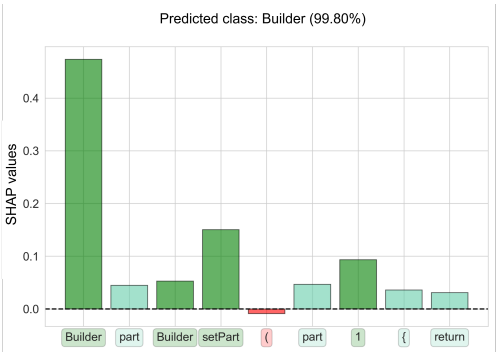
4.2.3 Effect on Patterns. Under ANON, several patterns exhibit relatively higher structure reliance across multiple models, suggesting that their structural cues remain informative after anonymisation. For instance, *Adapter* and *Builder* reach $\bar{\rho}_{\text{syn}} \approx 0.52{-}0.58$ and 0.53–0.57 across models, and *Prototype* and *Visitor* also trend higher for some encoders (up to 0.56). These patterns are consistent with recognisable structural templates that can be expressed through syntax and role relationships even when names are removed.

In contrast, *Facade* and *Factory Method* remain comparatively lower in structure reliance (typically $\approx$ 0.45–0.50 under ANON), suggesting that their distinguishing signals are either more subtle structurally, more easily confusable with nearby patterns, or more reliant on lexical cues that anonymisation disrupts.

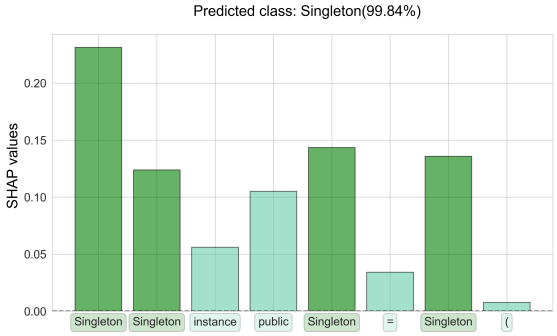
4.3 RQ3 - Cross-Dataset Robustness Under Distribution Shift

4.3.1 Models Trained on RAW. RQ1 showed that encoder models assign most of their attribution mass to identifier tokens when predicting design patterns, and RQ2 showed that training on ANON can encourage attribution mass towards syntax-structure tokens. RQ3 investigates how this behaviour impacts predictive performance when identifier names at test time are out-of-sample.

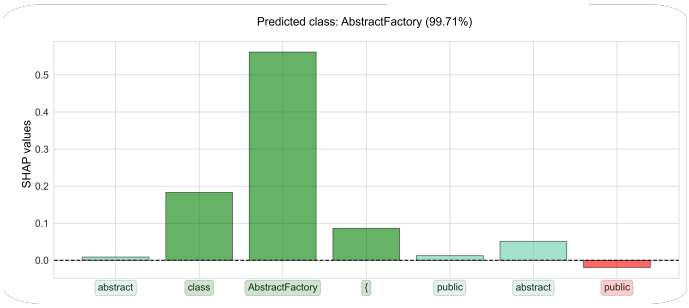
Table 2 summarises baseline performance for the encoder and classifier combinations that achieved highest G-Mean (top of table) and Specificity (bottom of table) in at least one training and testing scenario. Full results for all combinations are provided in Table 9 in the supplementary material. Table 2 shows that when the models are trained and evaluated on RAW, all detectors achieve strong performance, with G-Mean ranging from 0.94 to 0.97 and Specificity consistently at 0.99 across all model–classifier combinations. Macro-MCC and F1-score are also reported in Table 8 in the supplementary material, both exceeding 0.90. These results confirm that, when original identifiers are available, LLM-based encoders



(a) Top token SHAP (*Builder*).



(b) Top token SHAP (*Singleton*).



(c) Top token SHAP (*Abstract Factory*).

Figure 1: Token-level SHAP attributions for representative instances from three design patterns.

provide highly performing design-pattern detectors, corroborating results from the literature in that, under potentially annotated test data, LLM-based approaches represent strong DPD classifiers.

Figure 2 then contrasts this RAW baseline with performance under identifier shift (training on RAW and evaluating on ANON). Across all five encoder+classifier combinations, G-Mean collapses from the 0.94–0.97 range to approximately 0.41–0.57, indicating a substantial loss of agreement with the ground truth under out-of-sample identifier names. These differences in performance are perfectly consistent for every pattern, such that Wilcoxon Sign Rank Tests to compare RAW→RAW against RAW→ANON across LLMs, downstream classifiers and patterns leads to a p-value of $4.02e-41$, with very large effect size (A12 = 1.00). The degradation is also

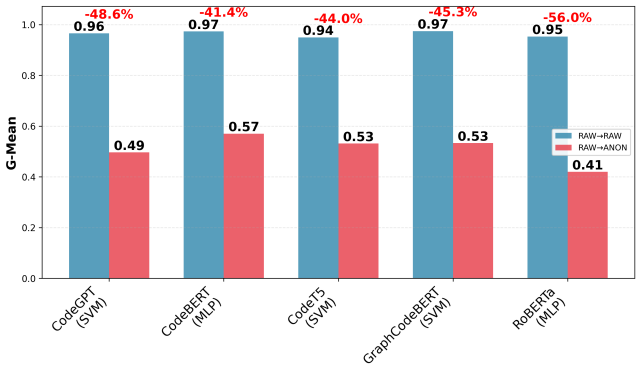


Figure 2: RAW→RAW vs RAW→ANON Performance Comparison (G-Mean Using Best Classifier per PLM)

high for other performance metrics, as shown in the supplementary material. Notably, specificity remains high under RAW→ANON (0.93–0.94), decreasing only marginally relative to RAW→RAW, as shown in Table 2. This implies that the primary failure mode is not an increase in false positives, but rather a sharp decrease in recall. Taken together with RQ1, as the classifier heavily relies on identifier names, it is unable to identify patterns when tested on ANON, where the identifier names are out-of-sample. This reveals a low robustness of out-of-sample identifier names.

Figures 3a and 3b in the supplementary material report per-pattern-specific recall for each encoder, using the top ranked classifier per PLM. Patterns such as *Builder*, *Facade*, and *Proxy* retain moderate recall relative to others, suggesting that they expose stronger structural signatures that remain partially recoverable after anonymisation. In contrast, patterns such as *Factory Method* and *Decorator* obtain low recall across encoders, indicating that their recognition depends more strongly on identifier names and does not transfer well when names are replaced.

4.3.2 Models Trained on CLEAN. Table 2 shows that detectors trained and evaluated on CLEAN achieve strong performance. For example, CodeBERT achieves G-Mean of 0.979 (SVM) and 0.979 (ADA), with Specificity consistently at 0.996–0.997, and GraphCodeBERT reaches 0.981 (SVM). However, when these same CLEAN-trained detectors are evaluated on ANON, performance drops sharply: CodeBERT falls to 0.575 (MLP) and 0.565 (SVM), GraphCodeBERT to 0.496 (MLP) and 0.450 (SVM), and RoBERTa to 0.433 (MLP). This indicates that removing comments alone does not prevent reliance on identifier, robustness remains limited once identifiers are replaced. To verify this, we compare CLEAN→CLEAN against CLEAN→ANON using a paired Wilcoxon signed-rank test over matched observations from the repeated stratified CV splits, yielding $p = 3.90 \times 10^{-18}$ with effect size A12= 1.00, indicating a perfectly consistently higher performance for CLEAN→CLEAN than for CLEAN→ANON.

4.3.3 Models Trained on ANON. Models trained on ANON typically underperform those trained on RAW and CLEAN when evaluated on their respective matched test sets, because anonymisation removes informative identifier cues. However, on anonymised test data, ANON→ANON consistently outperforms RAW→ANON and

Table 2: Cross-dataset G-Mean (Top) and Specificity (Bottom). Bold values indicate the highest score within each PLM group.

PLM	Cls.	Train RAW			Train CLEAN			Train ANON		
		Eval RAW	Eval CLEAN	Eval ANON	Eval RAW	Eval CLEAN	Eval ANON	Eval RAW	Eval CLEAN	Eval ANON
G-Mean										
CodeGPT	RF	0.962	0.958	**0.496**	**0.928**	0.976	**0.512**	0.629	0.786	0.831
	SVM	**0.966**	**0.963**	0.470	0.905	**0.977**	0.503	**0.688**	**0.835**	**0.868**
CodeBERT-base	ADA	0.973	**0.968**	0.514	0.949	0.979	0.559	0.746	0.822	0.862
	mlp	**0.973**	0.967	**0.570**	0.947	0.978	**0.575**	0.745	0.823	0.857
	RF	0.973	0.967	0.521	**0.949**	0.979	0.559	0.746	0.824	0.863
	SVM	0.973	0.968	0.514	0.949	**0.979**	0.565	**0.754**	**0.826**	**0.867**
CodeT5	mlp	0.933	**0.892**	0.519	0.862	0.909	**0.560**	**0.765**	**0.803**	0.817
	SVM	**0.950**	0.886	**0.531**	**0.869**	**0.924**	0.553	0.758	0.794	**0.841**
GraphCodeBERT-base	mlp	0.970	0.968	**0.533**	0.943	0.977	**0.496**	0.793	0.853	0.868
	SVM	**0.974**	**0.970**	0.497	**0.944**	**0.981**	0.450	**0.805**	**0.862**	**0.877**
RoBERTa-base	mlp	**0.953**	**0.932**	**0.419**	**0.854**	**0.891**	**0.433**	**0.538**	**0.631**	0.687
	RF	0.941	0.910	0.404	0.820	0.884	0.422	0.520	0.620	**0.702**
Specificity										
CodeGPT	RF	0.994	0.993	**0.936**	**0.989**	0.996	**0.938**	0.949	0.968	0.974
	SVM	**0.994**	**0.994**	0.933	0.985	**0.996**	0.937	**0.955**	**0.974**	**0.979**
CodeBERT-base	ADA	0.996	**0.995**	0.937	0.992	0.996	0.941	0.962	0.972	0.978
	mlp	**0.996**	0.995	**0.942**	0.992	0.996	**0.943**	0.962	0.973	0.978
	RF	0.996	0.995	0.938	**0.992**	0.996	0.941	0.962	0.973	0.978
	SVM	0.996	0.995	0.937	0.992	**0.997**	0.942	**0.963**	**0.973**	**0.979**
CodeT5	mlp	0.989	**0.983**	0.937	0.979	0.985	**0.941**	**0.966**	**0.970**	0.972
	SVM	**0.992**	0.982	**0.938**	**0.980**	**0.988**	0.940	0.964	0.969	**0.975**
GraphCodeBERT-base	mlp	0.995	0.995	**0.938**	0.991	0.996	**0.936**	0.969	0.977	0.979
	SVM	**0.996**	**0.995**	0.935	**0.991**	**0.997**	0.932	**0.971**	**0.978**	**0.981**
RoBERTa-base	mlp	**0.992**	**0.989**	**0.928**	**0.977**	**0.983**	**0.928**	**0.938**	**0.948**	0.954
	RF	0.990	0.986	0.927	0.973	0.982	0.927	0.936	0.946	**0.956**

CLEAN→ANON in terms of G-Mean (Table 2). Paired Wilcoxon signed-rank tests over matched observations from repeated stratified CV splits confirm both improvements, with $p < 1 \times 10^{-6}$ and A12 = 1.00 for each comparison. This is expected because ANON does not eliminate lexical cues entirely: identifiers are replaced by random strings *consistently* across files, so some anonymised names recur across training and test folds. Consistent with RQ1–2 attributions, ANON-trained models still assign non-trivial importance to identifier tokens, and this overlap benefits evaluation on ANON. Nevertheless, even under ANON→ANON, the average G-Mean is approximately 0.82, indicating that detection becomes substantially harder once identifier information is degraded, even when the shift is partial rather than fully out-of-sample.

Overall, RQ3 shows that identifier shift is a dominant driver of errors: when names are out-of-sample, current encoder-based DPD systems do not consistently fall back to structure-only signals, resulting in major performance degradation.

5 Threats to Validity

Internal Validity, Performance and attributions can vary with fine-tuning hyperparameters, random seeds, and implementation choices. We mitigate this by using a single fixed training configuration across all models, repeated stratified cross-validation, and reporting metrics aggregated across repeats/folds. Because RAW, CLEAN, and ANON are different views of the same instances, we ensure split alignment by constructing folds once using shared instance indices and reusing the same train/test row indices across all variants and all encoder+classifier combinations. To reduce inconsistencies from preprocessing, tokenisation, or embedding extraction, we use a shared pipeline and record configurations for reproducibility, for Kernel SHAP, we fix analysis-specific settings.

Construct and Statistical Conclusion Validity, Our reliance measures aggregate token-level Kernel SHAP attributions into two buckets (identifiers vs. syntax/structure). This is approximate because some tokens are ambiguous and Kernel SHAP is sensitive to the background set, the number of samples, and the masking/imputation mechanism, which can add noise for long sequences. We mitigate this by applying a consistent bucketing rule, fixing the SHAP feature dictionary from a shared background pool within each analysis, and aggregating across multiple SHAP seeds, instances, and patterns. CLEAN and ANON are controlled stress tests and may not fully reflect real-world naming noise, we therefore interpret them as probes of brittle lexical shortcuts rather than deployment simulations.

External Validity, Our experiments use DPD-F and its project mappings to reconstruct original Java code. Although DPD-F is mined from real projects with non-trivial instances, it may not represent the diversity of industrial systems. We also evaluate five encoders under a fixed "encoder + downstream classifier" pipeline, so results may not generalise to other model sizes, instruction-tuned or decoder-only LLMs used via prompting, alternative pooling, or end-to-end architectures.

6 Conclusion

This paper examined whether recent LLM-based DPD learn structure or instead rely on identifier names such as class, variable, and method name. Our analysis shows that models trained on RAW are dominated by identifier-driven attributions and suffer substantial performance degradation under anonymisation (RQ1). Training on CLEAN is unable to shift attribution mass sufficiently towards structure-oriented. ANON shifts attribution mass more considerably, but it does not fully eliminate identifier dependence, leaving a residual robustness gap under out-of-sample naming conventions (RQ2). Reliance on identifier names was shown to make models sensitive to out-of-sample identifier names, with even models trained on ANON achieving insufficiently strong G-Mean when tested on test data containing some out-of-sample identifiers (RQ3). Overall, our findings warn that a model can perform well in experimental evaluation while learning fragile shortcuts (e.g., pattern names or annotations), making them unreliable in real deployments.

Future work includes evaluating DPD models with out-of-sample identifier names in cross-project scenarios, and with additional dataset variants such as removing only comments that provide explicit hints about the design pattern a code belongs to. Furthermore, developing detectors that remain robust when identifier cues are weak, misleading, or intentionally removed is a valuable direction.

Acknowledgments

The computations described in this research were performed using the Baskerville Tier 2 HPC service (https://www.baskerville.ac.uk/). Baskerville was funded by the EPSRC and UKRI through the World Class Labs scheme (EP/T022221/1) and the Digital Research Infrastructure programme (EP/W032244/1) and is operated by Advanced Research Computing at the University of Birmingham. This work was supported by the Beasiswa Indonesia Bangkit (BIB)-LPDP scholarship from the Government of Indonesia.

References

[1] Miltiadis Allamanis and Charles Sutton. 2013. Mining Source Code Repositories at Massive Scale using Language Modeling. In *MSR*. 207–216.
[2] Awny Alnusair, Tian Zhao, and Gongjun Yan. 2014. Rule-based detection of design patterns in program code. *STTT* 16, 3 (2014), 315–334.
[3] Apostolos Ampatzoglou, Alexander Chatzigeorgiou, Sofia Charalampidou, and Paris Avgeriou. 2015. The Effect of GoF Design Patterns on Stability: A Case Study. *IEEE TSE* 41, 8 (2015), 781–802.
[4] Francesca Arcelli and Luca Cristina. 2007. Enhancing Software Evolution through Design Pattern Detection. In *Third International IEEE Workshop on Software Evolvability*. 7–14.
[5] Z. Balanyi and R. Ferenc. 2003. Mining design patterns from C++ source code. In *ICSM*. 305–314.
[6] Leo Breiman. 2001. Random Forests. *Mach. Learn.* 45, 1 (2001), 5–32.
[7] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Machine Learning* 20, 3 (Sept. 1995), 273–297.
[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT*. 4171–4186.
[9] Gcinizwe Dlamini, Usman Ahmad, Lionel Randall Kharkrang, and Vladimir Ivanov. 2024. Detecting Design Patterns in Android Applications with CodeBERT Embeddings and CK Metrics. In *Analysis of Images, Social Networks and Texts*. 267–280.
[10] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. *CoRR* abs/2002.08155 (2020). arXiv:2002.08155 https://arxiv.org/abs/2002.08155
[11] Manuel Fernández-Delgado, Eva Cernadas, Senén Barro, and Dinani Amorim. 2014. Do we Need Hundreds of Classifiers to Solve Real World Classification Problems? *JMLR* 15, 90 (2014), 3133–3181.
[12] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. 1993. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In *ECOOP*. 406–431.
[13] Shuzheng Gao, Cuiyun Gao, Chaozheng Wang, Jun Sun, David Lo, and Yue Yu. 2023. Two Sides of the Same Coin: Exploiting the Impact of Identifiers in Neural Code Comprehension. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1933–1945. doi:10.1109/ICSE48619.2023.00164
[14] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. 2011. Deep Sparse Rectifier Neural Networks. In *AISTATS*. 315–323.
[15] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2020. GraphCodeBERT: Pre-training Code Representations with Data Flow. *CoRR* abs/2009.08366 (2020). arXiv:2009.08366 https://arxiv.org/abs/2009.08366
[16] Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin. 2003. A Practical Guide to Support Vector Classification. https://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf Technical report.
[17] Dae-Kyoo Kim, Robert France, and Sudipto Ghosh. 2004. A UML-based language for specifying domain-specific patterns. *Journal of Visual Languages & Computing* 15, 3 (2004), 265–289. Domain-Specific Modeling with Visual Languages.
[18] Diederik P. Kingma and Jimmy Ba. 2017. Adam: A Method for Stochastic Optimization. arXiv:1412.6980 [cs.LG] https://arxiv.org/abs/1412.6980
[19] Weichao Liu, Cheng Zhang, Futian Wang, and Yun Yang. 2020. Combining Network Analysis with Structural Matching for Design Pattern Detection. In *EASE*. 61–70.
[20] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. arXiv:1907.11692 [cs.CL] https://arxiv.org/abs/1907.11692
[21] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Nan Duan, and Ming Zhou. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *arXiv preprint arXiv:2102.04664* (2021). arXiv:2102.04664 [cs.SE]
[22] Scott M. Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. In *NIPS*. 4768–4777.
[23] Ahmad Haji Mohammadkhani, Chakkrit Tantithamthavorn, and Hadi Hemmatif. 2023. Explaining Transformer-based Code Models: What Do They Learn? When They Do Not Work?. In *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 96–106. doi:10.1109/SCAM59687.2023.00020
[24] Rania Mzid, Ilyes Rezgui, and Tewfik Ziadi. 2024. Attention-Based Method for Design Pattern Detection. In *ECSA 2024*. 86–101.
[25] Vinod Nair and Geoffrey E. Hinton. 2010. Rectified Linear Units Improve Restricted Boltzmann Machines. In *ICML*.
[26] Najam Nazar, Aldeida Aleti, and Yaokun Zheng. 2022. Feature-based software design pattern detection. *JSS* 185 (2022), 111179.
[27] Murat Oruc, Fuat Akal, and Hayri Sever. 2016. Detecting Design Patterns in Object-Oriented Design Models by Using a Graph Mining Approach. In *CONISOFT*. 115–121.
[28] Sushant Kumar Pandey, Sivajeet Chand, Jennifer Horkoff, Miroslaw Staron, Miroslaw Ochodek, and Darko Durisic. 2025. Design pattern recognition: a study of large language models. *EMSE* 30, 3 (2025), 45 pages.
[29] Sushant Kumar Pandey, Miroslaw Staron, Jennifer Horkoff, Mirosław Ochodek, Nicholas Mucci, and Darko Durisic. 2023. TransDPR: Design Pattern Recognition Using Programming Language Models. In *ESEM*. 1–7.
[30] Dhasarathy Parthasarathy, Cecilia Ekelin, Anjali Karri, Jiapeng Sun, and Panagiotis Moraitis. 2022. Measuring design compliance using neural language models: an automotive case study. In *PROMISE*. 12–21.
[31] John C. Platt. 1999. Probabilistic Outputs for Support Vector Machines and Comparisons to Regularized Likelihood Methods. In *Advances in Large Margin Classifiers*, Alexander J. Smola, Peter Bartlett, Bernhard Schölkopf, and Dale Schuurmans (Eds.). MIT Press.
[32] Christian Schindler and Andreas Rausch. 2025. LLM-Based Design Pattern Detection. arXiv:2502.18458 [cs.SE] https://arxiv.org/abs/2502.18458
[33] Colin C. Venters, Rafael Capilla, Stefanie Betz, Birgit Penzenstadler, Tom Crick, Steve Crouch, Elisa Yumi Nakagawa, Christoph Becker, and Carlos Carrillo. 2018. Software sustainability: Research and practice from a software architecture viewpoint. *JSS* 138 (2018), 174–188.
[34] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *EMNLP*. 8696–8708.
[35] Ji Zhu, Hui Zou, Saharon Rosset, and Trevor Hastie. 2009. Multi-class AdaBoost. *Statistics and Its Interface* 2, 3 (2009), 349–360.