Design Pattern Detection in Source Code: A Neural Graph Database Approach as an Ensemble Base Model

Roy Oberhauser[0000-0002-7606-8226] and Peter Schneider
Computer Science Dept.
Aalen University
Aalen, Germany
e-mail: roy.oberhauser@hs-aalen.de, peter.schneider@studmail.htw-aalen.de

Abstract - **Software design patterns offer reusable structural solutions that support developers and maintainers in addressing common design problems. Their abstractions can support program code documentation and comprehension, yet manual pattern documentation via code or code-related artifacts (documents, models) can be unreliable, incomplete, and labor-intensive. Various automated Design Pattern Detection (DPD) techniques have been proposed, yet adoption remains limited and further investigation of viable solutions is needed. Towards more effective automated DPD, this paper contributes our Neural Graph Database approach DPD-NGDB, which also functions as a base model in our Ensembles Methods approach DPD-EM. The realization demonstrates the feasibility of our approaches, while the evaluation compares and benchmarks the DPD performance against a Gang-of-Four (GoF) software design pattern dataset, demonstrating its potential.**

Keywords – software design pattern detection; machine learning; neural graph databases; graph neural networks; ensemble methods; software design patterns; software engineering.

I. Introduction

This paper extends our previous work [1], in that it investigates further potential DPD methods, leveraging Ensembles Methods (EMs), Neural Graph DataBases (NGDBs), and Graph Neural Networks (GNNs) approaches for DPD. The breadth and depth of the evaluation is extended to and benchmarked against the entire Gang of Four (GoF) design patterns. Our prior hybrid DPD approach has been changed to an EM approach.

Program source code worldwide continues to rapidly expand, yet code comprehension remains a limiting productivity factor. Program comprehension may consume up to 70% of the software engineering effort [2]. Activities involving program comprehension include investigating functionality, internal structures, dependencies, run-time interactions, execution patterns, and program utilization; adding or modifying functionality; assessing the design quality; and domain understanding of the system [3]. Code that is not correctly understood by programmers impacts quality and efficiency.

Software Design Patterns (DPs) have been documented and popularized, including the Gang of Four (GoF) [4] and POSA [5]. The application of abstracted and documented solutions to recurring software design problems has been a boon to improving software design quality, efficiency, aiding comprehension, refactoring, reuse, reverse-engineering, and maintenance tasks. These well-known macrostructures or associated pattern terminology in code can serve as beacons to abstracted macrostructures, and as such may help identify aspects such as the author's intention or the purpose of a code segment, which, in turn, supports program comprehension.

Automated DPD in code overs various benefits, including: quicker comprehension of DP-related structural aspects of unfamiliar software; automatically documenting DPs; supplementing and validating the design documentation; reducing dependence on error-prone DP documentation; and detection of inadequately implemented DPs. Yet the challenges for automated DPD include: 1) tool support for heterogeneous programming languages, as DPs are independent of programming language; 2) internationalization and labeling, since developers may name and comment in their natural language or any way they like; 3) varying pattern abstraction levels, such as design vs. architectural patterns; 4) similarities and intent differentiation, since some similar pattern structures are primarily differentiated via their intention; 6) DP localization to indicate where in code a DP was detected; and 7) detecting variants, since each pattern implementation is unique.

Traditional DPD approaches rely on static code analysis, with tools extracting structural features such as class hierarchies, method invocations, and object instantiations to identify patterns [6]. Curated design pattern datasets, such as the Pattern-like Micro-Architecture Repository (P-MARt) [7][8], provide a collection of micro-architectures from known open source projects which applied the canonical design patterns and can serve as benchmarks for evaluating DPD techniques. While various DPD approaches have been explored [9][10], no approach has thus far achieved significant practical traction, and thus additional investigation into further possibly viable approaches and improvements is warranted.

GNNs [11] are a class of Deep Learning (DL) models that are specifically designed to work with graph-structured data. They learn representations that capture the features of individual nodes and the relationships between them. Unlike traditional neural networks (NNs) that assume independent data points, GNNs leverage the topology of graphs to propagate information across nodes via edges, making them particularly suitable for domains where relationships are key.

NGDBs [12][13] extend classical graph database systems by integrating GNNs directly into the graph storage and the query engine. An NGDB is designed to store, manage and query graphs using both traditional graph operations and

neural inference to enrich incomplete or uncertain data, perform link prediction, and extract embeddings on-the-fly. This hybrid paradigm unifies transactional graph queries (e.g., Cypher [14]) with GNN-based tasks (e.g., node classification, link prediction), enabling real-time inference powered by the rich information encoded in a Labeled Property Graph (LPG) data model.

EMs [15][16] incorporate a finite set of alternative Machine Learning (ML) algorithms and models to enhance predictive performance, especially where a single model may not perform ideally. As design patterns can exhibit significant variance and non-linear relationships, we believe that no single technique (ensemble) will likely perform well in all circumstances. Thus, a mix of models (ensembles) may improve results when faced with significant variance, diversity, and non-linear relationships in the datasets, as is often the case with DPD.

Our previous work includes: our ML-based DPD approach DPDML that utilizes semantic and static analysis metrics [17]; our hybrid DPD approach HyDPD [18], which combines our ML-based model with an expert-based graph analysis model; and HyDPD-B [1], which applies a Bayesian network probabilistic reasoning to integrate various DPD subsystems, including HyDPD-ML utilizing graph embeddings, with our expert rule system with DP rule language and micropattern detection.

This paper contributes our NGDB-based solution approach (DPD-NGDB), which is embedded as a base model in our EM approach (DPD-EM). We describe our realization, which demonstrates the feasibility of the DPD approaches. Our evaluation uses a dataset consisting of the Gang-of-Four (GoF) design patterns benchmarked against the P-MARt repository.

This paper is structured as follows: the next section discusses related work. Section III describes our solution. In Section IV, our realization is presented, which is followed by our evaluation in Section V. Finally, a conclusion is provided.

II. Related Work

Surveys including categorizations of DPD approaches include Al-Obeida et al. [9] and Yarahmadi and Hasheminejad [10]. Graph-based DPD approaches include: Yu et al. [19] transform code to UML class diagrams, analyze the XMI for sub-patterns in class-relationship directed graphs; Mayvan and Rasoolzadegan [20] use a UML semantic graph; Bernardi et al. [21] apply a DSL-driven graph matching approach; DesPaD [22] extract an abstract syntax tree from code, create a single large graph model of a project, and then apply an isomorphic sub-graph search method. Further isomorphic subgraph approaches include Pande et al. [23] and Pradhan et al. [24], both of which require UML class diagrams.

Learning-based approaches map the DPD problem to a learning problem, and can involve classification, decision trees, feature maps or vectors, Artificial Neural Networks (ANNs), etc. Examples include Alhusain et al. [25], Zanoni et al. [26], Galli et al. [27], Ferenc et al. [28], Uchiyama et al. [29], and Dwivedi et al. [30]. Thaller et al. [31] describe a micro-structure-based structural analysis approach based on feature maps. Chihada et al. [32] convert code to class diagrams, which are then transformed to graphs, and have experts create feature vectors for each role based on object-oriented metrics and then apply ML.

Additional approaches include: reasoning-based approaches such as Wang et al. [33] based on matrices; rule-based approaches like Sempatrec [34] and the ontology-based FiG [35]; metric-based approaches such as MAPeD [36], Uchiyama et al. [29], and Dwivedi et al. [37]; Fontana et al. [38] analyze microstructures based on an abstract syntax tree; semantic-analysis style includes Issaoui et al. [39]; while DP-Miner [40] uses a matrix-based approach based on UML for structural, behavioral, and semantic analysis. Singh et al. [41] combines static rules with graph analysis. GEML [42] initializes a population of random structures, applying genetic algorithms to mutate and generate new patterns from the initial population. Kouli and Rasoolzadegan [43] utilize micro-patterns with binary logic.

Graph-based code representations have emerged to capture syntactic and semantic dependencies more effectively. Liu et al. [44] propose a Code Property Graph (CPG)-based GNNs for code similarity detection, achieving high performance by learning multi-hop dependencies. Ampatzoglou et al. [45] use neural sub-graph matching with GNNs to detect design patterns in large codebases, demonstrating robustness to structural variations. Li et al. [46] integrated multi-feature fusion with GNNs, combining semantic embeddings and structural metrics for enhanced detection in real-world projects.

NGDBs represent an innovative extension, integrating GNNs with graph databases such as Memgraph, offering real-time pattern analysis [12][13]. NGDBs enable dynamic feature computation and scalable querying, which could support tasks like pattern recognition in evolving codebases [47].

Ensemble Methods have proven effective in improving robustness and accuracy for ML tasks [15][16]. For DPD, ensembles like Random Forests and stacked generalization combine multiple classifiers to leverage complementary strengths [48]. However, integrating GNNs, NGDBs, and traditional ML models in ensembles remains underexplored, particularly for addressing class imbalance and generalization, and we see an opportunity and flexibility for addressing DPD issues via its utilization.

III. Analysis and Requirements

DPD approaches can arguably be categorized into three primary approaches: 1) learning-based, where DPs are (semi-)automatically learned (e.g., via supervised learning) from provided data and requiring minimal expert intervention; 2) knowledge-based, whereby an expert defines DPs by describing elements and their associations; and 3) similarity-based, whereby DPs are grouped based on similar metrics or characteristics.

A. Analysis

DPD in object-oriented code, such as Java, involves identifying structural, creational, and behavioral patterns from the Gang of Four (GoF) catalog. However, several challenges complicate this task:

C1: Variability in Implementations: DPD must take variability into account, since patterns can be implemented in non-standard ways, with variations in naming, structure, or partial realizations. Traditional rule-based tools like P-MARt or PINOT (Pattern Identification using Optimization and Transformation) [49] often fail to detect these variants due to rigid matching criteria.

C2: Dataset Limitations: Existing "labeled" DPD datasets often consist of clean, textbook examples but lack diversity, including real-world project integrations. Imbalances between pattern and non-pattern instances further complicate ML approaches.

C3: Feature Representation: Static analysis alone misses dynamic behaviors, while graph-based representations (e.g., Abstract Syntax Tree (AST) or Call Graph (CG)) require sophisticated handling to capture inter-class relationships without losing semantic information.

C4: Scalability: DPD must handle large codebases efficiently.

C5: Generalization: DPD must generalize to unseen projects, avoiding overfitting to specific implementations. Ensemble methods and GNNs show promise in addressing these by combining complementary strengths:

- ML classifiers for feature-based detection,
- GNNs for structural resilience, and
- NGDBs for real-time querying.

B. Requirements

To address these challenges, we identify the following DPD requirements, categorized by functional and non-functional:

1) Functional Requirements (FR):

1. *DPD Coverage*: The system shall detect all 23 GoF design patterns (creational, structural, behavioral) in Java source code or bytecode.
2. *Feature Extraction*: Extract multi-modal features including numerical (e.g., method counts, complexity), graph-based (e.g., ASTs, CGs), and derived structural metrics (e.g., inheritance degrees).
3. *Model Training and Inference*: Implement training pipelines for individual models (e.g., SVM, GNN, NGDB) and an ensemble combiner (e.g., using soft voting).
4. *Dataset Management*: Extend the P-MARt dataset with real-world samples, class-level labeling, and imbalance handling via oversampling.
5. *Evaluation Framework*: Provide an evaluation framework that offers cross-validation (K-Fold, Leave-One-Project-Out (LOPO)) and performance metrics (F1-Score, confidence).
6. *Real-time Processing*: Support batch processing of multiple repositories and real-time pattern detection for integration into development workflows.

2) Non-Functional Requirements (NFRs):

1. *Performance:* The system must process medium-sized Java projects (up to 10,000 classes) within reasonable time constraints (under 30 minutes for complete analysis).
2. *Scalability:* Support horizontal scaling for batch processing of multiple repositories simultaneously.
3. *Accuracy:* Achieve F1-scores above 0.80 for common design patterns and maintain robustness against code variations.
4. *Extensibility:* Provide a modular architecture allowing addition of new pattern types, feature extractors, and an ensemble integration of further base models.
5. *Reproducibility:* Ensure deterministic results through proper random seed management and version control of models and datasets.

IV. Solution

Our DPD solution approach incorporates the features and conceptual architecture described in the following.

A. Features

Full GoF Dataset: The DPD scope encompasses the 23 GoF design patterns, covering creational, structural, and behavioral design patterns.

Class-Level Feature Aggregation: Since many design patterns are object-oriented, the feature extraction pipeline operates at the class level to preserve contextual relationships between methods and their containing classes.

Multi-Modal Feature Engineering: The feature extraction incorporates:

- Numerical features from AST analysis,
- Graph-based features from CG analysis, and
- Structural features like inheritance relationships.

Multi-model Support: The open approach can incorporate multiple varying base model types, including GNNs, NGDBs, SVMs, and can be combined with Ensemble Methods.

B. Modular Architecture

A modular, pipeline-based architecture separates concerns between feature extraction, model training, and inference. The architecture is designed to support the identified functional and non-functional requirements while maintaining flexibility for future extensions.

It consists of four primary modules, each responsible for a specific stage in the DPD process, as shown in Figure 1.

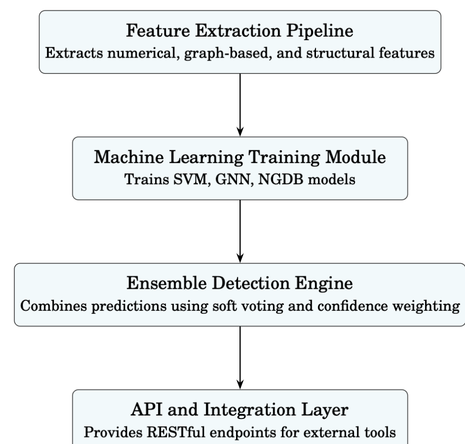

Figure 1. DPD-EM Module Architecture

1. *Feature Extraction*: Processes source code (or bytecode/binary code if desired) to extract numerical, graph-based, and structural features at both method and class levels.
2. *ML Training*: Implements training pipelines for individual models (e.g., GNN, NGDB, SVM) including training evaluation and model storage.
3. *DPD and Ensembles*: The DPD predictions from a single (or multiple) trained models can be flexibly utilized. In the multi-model case, any ensemble technique can be applied (voting, expert rules, decision trees, etc.) to leverage the complementary strengths of individual models as desired and thereby enhance overall DPD performance. While our DPD-EM ensemble method offers flexible multi-model support, when only a single ensembles classification results are used, we then refer to its specific model (e.g., DPD-NGDB), even though our modular architecture remains ensemble-enabled.
4. *System Integration and API*: Provides Web API endpoints for training, inference, and result management with standardized interfaces for external tool integration.

1) Data Flow Stages

The system processes data through a series of stages, each transforming the input into a more refined output. The stages are as follows and illustrated in Figure 2.

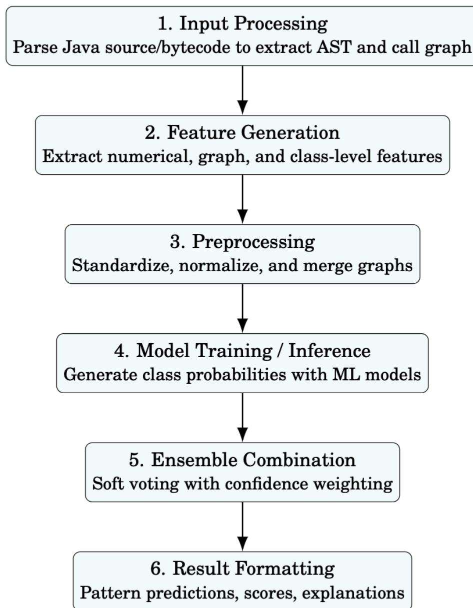

Figure 2. DPD-EM Data Flow Stages

1. *Input Processing:* Raw source code (or bytecode/binary) is parsed to extract AST and CG information.
2. *Feature Generation:* Multi-modal feature extraction generates numerical metrics, graph structures, and derived features at the class-level (which subsumes method-level).
3. *Preprocessing:* Feature standardization, normalization, and graph merging prepare data for ML models.
4. *Model Training/Inference*: Individual models are trained or used for inference, generating probability distributions over pattern classes.
5. *Ensemble Combination:* Any ensemble technique such as soft-voting with confidence weighting) to combine individual model predictions and enhance overall predictions.
6. *Result Formatting:* Standardized output includes pattern predictions, confidence scores, and detailed explanations.

V. REALIZATION

This section describes the realization of our DPD solution approach, providing details about our modules and pipeline.

A. Module Overview

The solution architecture was realized in Python, with each module addressing distinct functionality within the DPD pipeline. Any filenames listed are for reference purposes for subsequent descriptions and not intended to be comprehensive. The four modules consist of:

Feature Extraction Module (M1): Implements numerical feature extraction, including from ASTs, CGs, or static analysis metrics:

- NumericalFeaturePreprocessing.py: Handles feature standardization and preprocessing.
- GraphFeatureProcessor.py: Processes method-level CGs into class-level representations.

ML Training Module (M2): Implements training pipelines for individual models (e.g., GNN, NGDB, SVM) including evaluation and model persistence:

- TrainGNN.py: Implements the training pipeline for GNNs.
- TrainPatternPipeline.py: Implements the training pipeline for NGDB approaches.
- TrainSVM.py: Implements the training pipeline for SVMs.

DPD and Ensembles Module (M3): Implements the ensemble technique that combines predictions from multiple trained models. It currently applies soft voting and confidence-based weighting, but any other EM technique can be applied:

- Detection.py: Provides unified interfaces for applying trained models for DPD.

System Integration and API Module (M4): Provides RESTful Web API endpoints for training, inference, and result management. Offers standardized interfaces for external tool integration:

- MLAPI.py: Offers system integration via a FastAPI backend.

B. Feature Extraction Module (M1)

Accurate DPD depends on the quality and structure of the underlying features extracted from software artifacts. Our multi-stage pipeline module transforms raw code into numerical and graph-based representations suitable for supervised learning and graph-based inference. Our AST and CG extraction utilizes JavaParser for Java source code and SootUp for bytecode, but multi-language support is feasible.

1) Numerical Feature Extraction: The numerical feature extraction process (NumericalFeaturePreprocessing.py) operates on AST, CG data, and static metrics to generate quantitative representations of code characteristics. Both bytecode and source code extraction modes are supported.

Numerical features are extracted from code to support analysis and classification. The method-level features extracted include:

- The number of method parameters and object instantiations,
- The presence of modifier flags such as public, private, static, final, abstract, and synchronized,
- The complexity of return types, categorized as: void (0), primitive type (1), custom class (2), generic type (3),
- Cyclomatic complexity,
- Maximum nesting depth, and
- The number of local variables and exception handlers.

To account for dynamic behavior and interprocedural relationships, metrics derived from the static CG are incorporated, including:

- The number of incoming and outgoing calls,
- The presence of self-calls, and
- The number of unique calling methods.

Class-level aggregation is performed, which summarizes method-level features using statistical operations such as sum, mean, maximum, and minimum. This enables a holistic view of each class based on the behavior of its constituent methods. A sample of the numerical feature extraction is shown in Figure 3. A sample of the CG extraction is depicted in Figure 4.


```
66350            "file": "./JRefactory_v2.6.24/src/org/acm/seguin/awt/OrderableList.java",
66351            "class_features": {
66352                "has_static_field": false,
66353                "num_methods": 4,
66354                "has_static_method": true,
66355                "methods_returning_interface": 0,
66356                "num_fields": 1,
66357                "num_abstract_methods": 0,
66358                "num_extended_types": 1,
66359                "has_private_constructor": false,
66360                "methods_returning_self_type": 0,
66361                "num_interfaces": 0
66362            },
66363            "methods": {
66364                "OrderableList.getData([])": {
66365                    "return_type": "Object[]",
66366                    "all_instantiations": [],
66367                    "exception_handlers": [],
66368                    "max_decision_node_depth": 0,
66369                    "unique_instantiations": [],
66370                    "local_variables": [],
66371                    "modifiers": ["public"],
66372                    "decision_nodes": [],
66373                    "parameters": []
66374                },
```


Figure 3. JSON snippet showing feature extraction for OrderableList within the JRefactory project.


```
19970          {
19971              "calls": [
19972                  "list.getSelectedIndex([])",
19973                  "olm.getSize([])",
19974                  "olm.swap([item, newPos])",
19975                  "list.setSelectedIndex([newPos])"
19976              ],
19977              "id": "MoveItemAdapter.actionPerformed([ActionEvent evt])"
19978          },
```


Figure 4. JSON snippet of extracted call graph data for MoveItemAdapter within the JRefactory project.

2) Feature Standardization and Preprocessing: Feature standardization scales a feature x to the interval [0,1] applying MinMax scaling. The process operates in two modes:

a) Training Mode: computes the global minimum and maximum values across all repositories, fits MinMaxScaler parameters, and serializes them as JSON for reproducibility.

b) Usage Mode: loads previously saved scaling parameters to ensure consistent feature transformation, allowing values to exceed [0,1] when input data exceeds training distribution bounds.

3) Graph Feature Processing and Aggregation: Graph feature processing transforms method-level CGs to class-level representations suitable for GNN processing.

a) Method-level CGs to Class-level Transformation: The process extracts class identifiers from method signatures using regular expressions as shown in Figure 5.

```python
def extract_class_id(method_id):
    core = re.sub(r"\(.*\)$", "", method_id)
    idx  = core.rfind('.')
    return core[:idx] if idx > 0 else core
```

Figure 5. Class identifier extraction via regular expressions

The transformation proceeds in three steps:

1. Node Consolidation: Creates one node per unique class ID containing the class identifier and feature vector from standardized numerical features, pattern labels available in training mode, method counts indicating class complexity, and additional structural metadata.

2. Edge Aggregation: Transforms method-to-method call relationships into class-to-class dependencies using set operations to eliminate duplicates and filtering self-loops automatically. The aggregation process includes class-level feature aggregation and distribution, feature consistency validation and quality checks, and support for missing feature imputation and default values.

3. Label Propagation: In training mode, pattern labels are propagated from class-level annotations to individual method nodes. This enables: ground truth establishment for supervised learning, label consistency validation across method-class hierarchies, support for partial labeling and semi-supervised approaches, and quality assurance for label accuracy and completeness.

b) Class-Level Graph Construction: This execution step represents the core transformation that supports higher-level DPD. It involves two primary operations: node consolidation and edge aggregation.

Node Consolidation Strategy: For each unique class ID, a consolidated node is created using aggregation strategies that preserve essential information while reducing graph complexity.

Class Node Creation: Each class node contains:

- A unique class identifier preserving full namespace information.
- Aggregated numerical features computed from all methods within the class using statistical summaries (mean, sum, max, min).
- Ground truth pattern labels (in training mode) for supervised learning.

- Method counts to indicate class complexity and architectural role.
- Structural metadata on class hierarchy, interface implementation, and architectural roles.

Feature Aggregation Strategies: Multiple approaches capture different aspects of class behavior:

- Statistical aggregation (mean, median, std, quartiles) to describe feature distribution.
- Structural aggregation (counts of constructors, getters, setters, etc.) for structural insights.
- Behavioral aggregation analyzing method interaction patterns.
- Complexity aggregation combining individual method complexities into class-level measures.

Edge Aggregation and Relationship Modeling: Method-to-method call relations are transformed into class-to-class dependencies via structured aggregation:

Dependency Extraction: dependencies are extracted from

- Direct dependencies: from inter-class method calls.
- Indirect dependencies: from multi-hop paths indicating complex patterns.
- Bidirectional relationships: signaling mutual dependencies (e.g., Observer, Mediator).
- Hierarchical dependencies: derived from inheritance.

Edge Weight Computation: these are calculated from:

- Call frequency weights representing the intensity of class interactions.
- Method diversity weights indicating the variety of methods involved.

C. ML Training Module (M2)

The ML training module encompasses multiple complementary approaches, each designed to capture different aspects of design pattern characteristics through specialized architectures and training strategies. Three are currently realized: 1) GNN, 2) NGDB, and 3) SVM.

1) GNN Training: The GNN implementation (in TrainGNN.py) is a DPD approach that leverages the structural relationships inherent in software architectures. The implementation supports multiple GNN variants, each offering different strengths for capturing various aspects of design pattern implementations.

a) GNN Architecture Design: A modular and configurable architecture that supports diverse GNN approaches was implemented in the Python Class GNNModel:

Supported GNN Variants: The system supports:

- Graph Convolutional Networks (GCN) provide an implementation of spectral graph convolutions that can capture local neighborhood information and hierarchical patterns characteristic of structural design patterns.
- Graph Attention Networks (GAT) offer attention-based mechanisms that dynamically weight the importance of different neighbors, which could support identifying key relationships inherent in behavioral design patterns.
- Graph Isomorphism Networks (GIN) offer the ability to distinguish between different graph structures that could be used to detect subtle variations in pattern implementations.
- Graph Sample and Aggregation (GraphSAGE) offers a scalable inductive learning approach that can generalize to unseen graph structures, which can support DPD across diverse codebases.

Configurable Architecture Parameters:

- Supports flexible specification of network capacity with hidden channels and layer depth for deep architectures that can capture complex hierarchical patterns.
- For GAT models, configurable multi-head attention provides customizable attention head counts and attention dropout rates.
- Multiple aggregation functions, including mean, max, sum, and attention-weighted approaches can be used to combine neighborhood information.
- Dropout strategies, batch normalization, and weight decay options prevent overfitting through regularization mechanisms.
- Configurable multi-layer perceptrons for final classification offer customizable hidden dimensions and activation functions.

b) Data Loading and Preprocessing Pipeline: The training pipeline begins with a data loading and preprocessing workflow:

Graph Data Processing: The system processes class-level graph features from JSON files through the following stages:

1. *File Access and Error Handling*: file system traversal ensures resilience to missing or corrupted files via error handling mechanisms.
2. Schema Validation: Parsed JSON content undergoes schema validation and integrity checks to guarantee consistency across heterogeneous data sources.
3. *Graph Conversion*: JSON representations are transformed into internal graph data structures, including validation of graph properties for structural correctness.
4. *Feature Extraction and Preprocessing*: Node features are extracted and preprocessed through missing value imputation, outlier detection, and normalization for subsequent learning stages.

PyTorch Geometric Integration: integration with PyTorch Geometric is implemented as a series of structured steps:

1. *Tensor Conversion*: transformation of feature matrices into GPU-compatible tensor formats for integration with PyTorch and memory-efficient representation for downstream processing.
2. *Edge Index Construction*: The construction of the edge_index structure includes error-checking to identify and handle invalid or malformed edges, ensuring graph connectivity and correctness.
3. *Label Encoding*: Label encoding is performed using scikit-learn's LabelEncoder, with support for handling unseen classes and mitigating class imbalance during supervised learning tasks.
4. *Graph Validation*: A validation step includes removal of self-loops, detection of disconnected components, and verification of structural graph properties to ensure integrity and suitability for geometric learning.

Data Quality Assurance: Data quality is addressed via multiple validation and control procedures:

- *Feature Range Validation*: Detection and handling of extreme values, infinite values, and NaN entries is performed to ensure numerical stability and consistency of the feature space.
- *Graph Connectivity Analysis*: Structural issues such as isolated nodes and disconnected components are identified via graph connectivity analysis to preserve semantic coherence of the graph.
- *Label Distribution Assessment*: The distribution of class labels is analyzed to identify class imbalance and detect severely underrepresented classes, which could negatively impact supervised learning performance.
- *Data Partitioning Strategy*: Data splits for training, validation, and testing are performed using partitioning strategies that preserve the underlying graph structure, ensuring representativeness and validity in cross-validation setups.

c) Training Strategies: The training implementation incorporates strategies designed to address DPD challenges:

Cross-Validation Framework: our cross-validation implementation supports multiple strategies. Configurable K-fold strategies employ stratified sampling to ensure balanced representation across folds. Evaluation strategies for smaller datasets provide validation of generalization performance through Leave-One-Out (LOO) Cross-Validation (CV). For datasets with a temporal structure, specialized validation strategies respect temporal dependencies. Graph-aware CV employs specialized partitioning strategies that account for graph structure and avoid information leakage between training and validation sets.

Applying K-fold CV to DPD settings presents unique challenges. Software projects often cannot be arbitrarily divided, as dependencies and architecture coherence must be preserved. For this reason, a special approach such as Leave-One-Project-Out Cross-Validation (LOPO-CV) is recommended [50]. Here, each project is treated as a single validation fold. The model is trained on all other projects and tested on the one left out. While LOPO-CV increases the evaluation for cross-project scenarios, it also introduces higher variance due to project heterogeneity. Normalization of features, abstraction layers, and embedding representations can help to mitigate these issues.

Optimizer Configuration and Tuning: Optimizer support includes configuration options. Advanced Adam implementations provide configurable learning rates, weight decay, and momentum parameters, proving particularly effective for graph neural network training. Classical stochastic gradient descent with momentum enables stable training on large graphs. Adaptive learning rate methods handle the sparse gradient updates common in graph neural networks through RMSprop. Sophisticated learning rate scheduling encompasses step decay, exponential decay, and cosine annealing strategies.

d) Focal Loss Implementation for Class Imbalance: Focal Loss was specifically implemented and adapted for DPD [51]. Our implementation with the class FocalLoss is shown in Figure 6.

```python
class FocalLoss(torch.nn.Module):
    def __init__(self, alpha=None, gamma=2):
        super().__init__()
        self.gamma = gamma  # Focusing parameter
        self.alpha = alpha  # Class balancing parameter

    def forward(self, inputs, targets):
        #Computes the Negative Log-Likelihood loss ce
        ce = F.nll_loss(inputs, targets, reduction="none")
        #Convert to probability p_t
        p_t = torch.exp(-ce)
        #Loss modulated by focussing parameter gamma
        loss = (1 - p_t) ** self.gamma * ce
        #Correct imbalance in datasets
        if self.alpha is not None:
            loss = self.alpha[targets] * loss
        #Return the mean loss across all samples
        return loss.mean()
```

Figure 6. Focal Loss implementation.

This loss function provides several advantages for DPD. The $(1 - p_t)^\gamma$ term down-weights easy examples and focuses learning on difficult cases, which is particularly important for distinguishing between similar design patterns. The α parameter provides explicit class balancing to address the severe imbalance between different design patterns in typical datasets. The loss automatically adjusts focus based on prediction confidence, enabling the model to concentrate on boundary cases and ambiguous pattern implementations. The formulation provides stable gradients even with extreme class imbalance, enabling effective training on highly skewed datasets.

e) Training Monitoring and Control: The training implementation includes various monitoring and control mechanisms:

Early Stopping Implementation: early stopping employs multiple criteria including:

- Configurable patience parameters with sophisticated validation loss tracking,
- Multiple metrics including F1-score, precision, and recall,
- Overfitting detection through training-validation loss divergence analysis, and
- Automatic retention of the best models based on multiple criteria with comprehensive metadata.

Metrics Tracking: Real-time monitoring of training progress encompasses:

- Detailed tracking of accuracy, precision, recall, F1-scores, and custom metrics for DPD,
- Real-time confusion matrix computation and analysis to identify specific DPD challenges,
- Detailed analysis of loss components to understand model learning dynamics, and
- Learning curve tracking for detecting convergence issues and optimization problems.

f) Model Artifact Management: Model artifact management supports reproducibility and enables detailed analysis:

Model Serialization: Model state preservation encompasses full model state dictionaries saved in .pth format with version compatibility, architecture specifications saved as JSON for reproducibility, training hyperparameters and

configuration settings, and optimizer state preservation for training resumption.

Visualization and Analysis: Visualization artifacts include Seaborn-based confusion matrix visualizations with statistical annotations, analysis of feature distributions and their relationship to pattern detection performance, and network visualizations of class-level graphs with pattern highlighting.

2) NGDB Training Pipeline: The NGDB training approach, implemented in the Python module TrainPatternPipeline.py is shown in Figure 7. It uses Memgraph as an NGDB to store, process, and analyze heterogeneous code graphs in combination with different GNN models. Memgraph is an in-memory, Cypher-compatible graph database that natively integrates GNN modules through its MAGE (Memgraph Advanced Graph Extensions) library [52]. This approach supports analysis by combining the NGDB with NN architectures, enabling graph-based DPD analysis of software code structures.

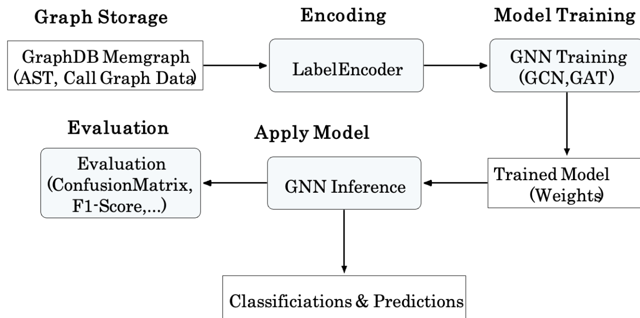

Figure 7. NGDB Training Pipeline

a) Graph Storage: The system employs Memgraph as the underlying graph database, chosen for its performance characteristics, Cypher Query Language (CQL) support, and integration capabilities with ML workflows. The database stores a heterogeneous graph representation of the software structure that captures multiple types of entities and relationships. A sample visualization of the CG import of the PMD project is shown in Figure 8. A closeup of the CG is shown in Figure 9.

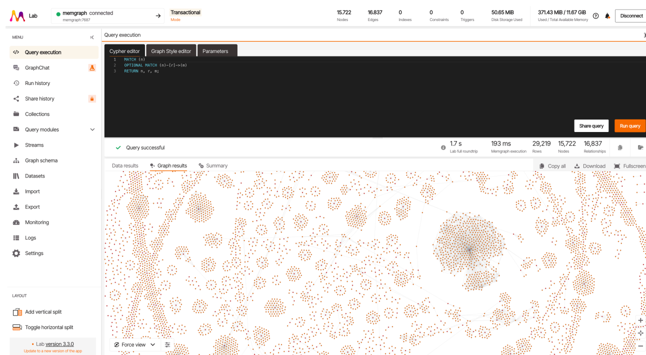

Figure 8. Screenshot of Memgraph for entire DPD dataset showing metrics above and clustering of class and method nodes by relationships.

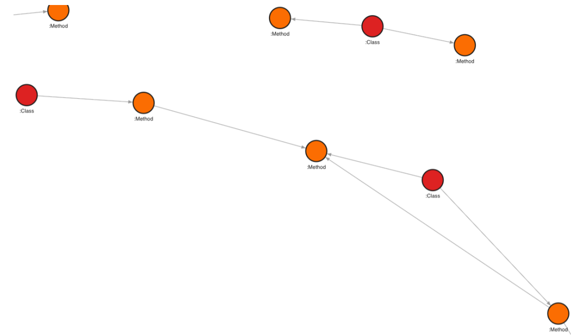

Figure 9. Closeup screenshot of Memgraph for PMD project dataset showing class (red) and method (orange) nodes and directed relationships.

Node Type Hierarchy: The graph database maintains a sophisticated node type system:

Class Nodes: These nodes represent software classes and are enriched with metadata including:

- Numerical feature vectors aggregated from method-level analysis,
- Pattern labels for supervised learning (available in training mode),
- Method counts and complexity metrics,
- Inheritance hierarchy information and interface implementation details,
- Package and namespace associations.

Method Nodes: These nodes represent individual methods and capture detailed characteristics such as:

- Method signatures and parameter types,
- Access modifiers and method-level flags (e.g., static, final),
- Cyclomatic complexity and code quality metrics,
- Call frequency, usage patterns, and invocation context,
- Exception handling structures and error management indicators.

Package Nodes: These nodes encapsulate software packages or modules, including:

- Hierarchical package organization and naming structure,
- Inter-package dependency relationships,
- Package-level metrics and aggregated characteristics.

Relationship Type System: The graph-based relationship model captures the semantic and structural interactions between software entities through multiple distinct relationship types:

DECLARES Relationships: These edges connect class nodes to their declared method nodes, encoding:

- Ownership and structural organization of methods within classes,
- Method accessibility (e.g., public, private) and scope information,
- Declaration context and compilation metadata,
- Indicators of method overrides and inheritance-related declarations.

CALLS Relationships: These edges represent dynamic or static method invocations and include:

- Call frequency and temporal invocation patterns,
- Parameter passing strategies and intra-procedural data flow,

- Exception propagation paths and error handling behavior,
- Conditional invocation patterns and control flow dependencies.

EXTENDS Relationships: These relationships model inheritance between classes, capturing:

- Inheritance depth and hierarchy complexity,
- Method override patterns and specialization,
- Abstract class associations and interface inheritance details.

IMPLEMENTS Relationships: These edges represent interface implementation and reflect:

- Interface compliance and contract fulfillment status,
- Patterns of multiple interface implementations,
- Usage of default methods and override behavior.

USES Relationships: These capture various forms of software usage dependencies, including:

- Field access and data dependency relations,
- Type usage, generic type associations, and class instantiations,
- Annotation usage and metadata-driven relationships.

b) Data Ingestion Pipeline: The data ingestion process implements a pipeline that loads and integrates multiple data sources into the unified graph representation:

Multi-Source Data Integration: The system systematically processes multiple JSON-based data sources, each contributing distinct structural and semantic information:

- *Graph Features Integration*: Incorporates class-level graph representations enriched with aggregated metrics and pattern labels, enabling structural learning at the class granularity.
- *Call Graph Processing*: Extracts method-level call relationships, capturing invocation frequency and contextual information to support interprocedural analysis.
- *Abstract Syntax Tree Data Integration*: Provides structural information about class hierarchies, inheritance relationships, and interface implementation through AST parsing.
- *Numerical Features Loading*: Supplies quantitative metrics for both classes and methods and offers statistical validation.

Graph Database Population: Ingestion functions manage the transformation and insertion of structured JSON nodes into the graph database, ingesting class-level data into a Memgraph instance. A Memgraph instance executes Cypher's MERGE clause to ensure idempotent insertion of class nodes based on a unique identifier. Node properties are assigned using dynamic key-value pairs passed via a props parameter. Metadata such as the ingestion timestamp and data version are appended to each node to support traceability, versioning, and data lineage.

Relationship Establishment: The system supports relationship creation between software entities with integrated validation and error handling mechanisms:

- *Inheritance Relationships*: Processing of EXTENDS clauses includes validation of multiple inheritance constraints and semantic correctness within the class hierarchy.
- *Method Declarations*: Methods are associated with their declaring classes through validated DECLARES relationships, incorporating access control checks and declaration context verification.
- *Call Relationships*: CALLS relationships are created between method nodes, enriched with invocation context, frequency metadata, and control flow annotations.
- *Interface Implementations*: IMPLEMENTS edges are formed based on parsed IMPLEMEMTS clauses, including validation of interface compliance and contract fulfillment semantics.

Data Consistency and Validation: To ensure semantic and structural correctness, the system performs validation procedures throughout the ingestion process:

- *Referential Integrity*: All referenced nodes and relationships are checked for existence to maintain consistency across the graph.
- *Schema Compliance*: Nodes and relationships are validated against predefined schema constraints to enforce type and structure conformity.
- *Data Quality Checks*: Inconsistencies, corrupted entries, and malformed properties are detected and handled using systematic quality control measures.
- *Duplicate Detection*: Redundant entities and relationships are identified and resolved through deduplication mechanisms to prevent semantic ambiguity.

c) Graph-Derived Feature Computation: the NGDB approach enables the computation of graph-derived features using Cypher queries:

Cypher Query Implementation: The system utilizes Cypher queries to derive composite structural metrics directly from the graph database. The query below (Figure 10) extends each Class node with additional structural and interaction-based features, enabling enhanced downstream analysis and learning.

```
mg.execute(
    """
    MATCH (c:Class)
    WHERE c.features IS NOT NULL AND c.pattern IS NOT NULL
    OPTIONAL MATCH
       (c)-[:DECLARES]->(m:Method)-[:CALLS]->(other_m:Method)
                <-[:DECLARES]-(other_c:Class)
    WITH c, collect(DISTINCT other_c.id) as
       called_classes_ids
    OPTIONAL MATCH (sub:Class)-[:EXTENDS]->(c)
    WITH c, called_classes_ids, count(DISTINCT sub) as
       in_degree_inheritance
    OPTIONAL MATCH (c)-[:EXTENDS]->(super:Class)
    WITH c, called_classes_ids, in_degree_inheritance,
         count(DISTINCT super) as out_degree_inheritance
    OPTIONAL MATCH (c)-[:DECLARES]->(meth:Method)
    WITH c, called_classes_ids, in_degree_inheritance,
       out_degree_inheritance,
         count(DISTINCT meth) as num_methods
    SET c.extended_features = c.features + [
        in_degree_inheritance, out_degree_inheritance,
        num_methods, size(called_classes_ids)
    ]
    """
)
```

Figure 10. A CQL query extends each Class node with additional structural and interaction-based features.

Specifically, the query proceeds in several stages:

1. *Filtering Relevant Classes*: Only classes for which both feature vectors (features) and pattern labels (pattern) are present are included in the analysis.
2. *CG Expansion*: The query traverses DECLARES and CALLS relationships to collect identifiers of all classes indirectly referenced via method calls, capturing inter-class interaction patterns (called_classes_ids).
3. *Inheritance Analysis (In-degree)*: By matching EXTENDS relationships pointing to the current class, the number of direct subclasses (i.e., inheritance in-degree) is computed.
4. *Inheritance Analysis (Out-degree)*: The number of direct superclasses extended by the current class (i.e., inheritance out-degree) is also calculated, supporting analysis of hierarchical complexity.
5. *Method Aggregation*: The number of methods declared by each class is counted using the DECLARES relationship to quantify class-level behavioral encapsulation.

- *Feature Augmentation*: A new property extended_features is appended to each class node. It combines the existing features vector with the four newly computed metrics: Inheritance in-degree, Inheritance out-degree, Number of declared methods, and Number of referenced external classes via method calls.

Following the execution of the Cypher query, the system augments each Class node with enriched architectural descriptors that enable a deeper analysis of software design structures and interaction patterns. These derived feature categories are extracted or inferred from graph-topological and semantic relationships:

<u>Inheritance Metrics</u>: captures the hierarchical properties of the class design, including:

- *In-degree inheritance*: The number of classes that extend a given class, indicating its centrality as a base class or abstract interface.
- *Out-degree inheritance*: The number of direct superclass relationships a class possesses, indicating hierarchy depth and potential misuse of multiple inheritance.
- *Inheritance tree depth*: The maximal depth from the root of the inheritance chain, used to detect deep or overly complex hierarchies.
- *Interface implementation count*: The number of interfaces implemented by a class, reflecting its abstraction adherence and flexibility.

<u>Method Declaration Patterns</u>: describes the intra-class behavioral structure:

- *Total method count per class:* providing a proxy for behavioral richness.
- *Distribution of method types:* distinguishing between constructors, accessors, mutators, and core logic methods.
- *Access modifier patterns:* such as the public-to-private method ratio, which may suggest encapsulation quality.
- *Abstract method statistics:* relevant for identifying abstract base classes or template patterns.

<u>Inter-Class Communication</u>: Encodes communication behavior in the CG:

- *Outgoing communication count*: i.e., the number of distinct classes this class calls.
- *Incoming communication count:* i.e., the number of classes calling this class.
- *Communication intensity and frequency:* providing insight into dependencies and possible code smells.
- *Bidirectional call detection:* to identify tight coupling or cyclic dependencies.

<u>Architectural Complexity</u>: Captures structural roles and architectural health:

- *Dependency fan-in and fan-out metrics:* used for impact and stability analysis.
- *Coupling and cohesion indicators:* inferred from communication and method sharing patterns.
- *Architectural layer classification:* by analyzing depth and connection types.
- *Pattern-specific structural indicators:* such as those associated with Singleton or Factory design patterns.

As shown in the CQL query above, a feature vector enhancement process is finally applied:

<u>Feature Concatenation</u>: The derived graph-based features are concatenated with the original static numerical features extracted from source code. Normalization and scaling are applied to ensure numerical stability.

<u>Dimensionality Management</u>: Strategies such as dimensionality capping, regularization, or feature selection are employed to prevent overfitting and ensure computational tractability.

<u>Feature Correlation Analysis</u>: Correlation matrices and mutual information metrics are used to detect and eliminate redundant or collinear features.

<u>Quality Validation</u>: The resulting feature vectors are validated for completeness, consistency, and compatibility with downstream models.

This multi-level feature integration enables the model to capture a rich representation of both static and relational program semantics, supporting tasks such as design pattern classification, anomaly detection, and architecture recommendation.

d) Class Imbalance Mitigation Strategies: Our DPD-NGDB approach integrates the following techniques to address the inherent class imbalance and memory constraints associated with design pattern datasets.

<u>Minority Class Oversampling</u> is used to mitigate the severe class imbalance in supervised learning tasks. The system employs graph-specific oversampling strategies that preserve structural integrity:

- *Configurable Oversampling Factor*: Typically, 2x oversampling with configurable parameters based on class distribution analysis
- *Graph-Aware Sampling*: Oversampling strategies that preserve graph structure and neighborhood characteristics
- *Synthetic Graph Generation*: Advanced techniques for generating synthetic graph samples that maintain pattern characteristics
- *Stratified Sampling*: Ensuring representative sampling across different pattern types and complexity levels

Memory Management and Scalability is used to support high-volume graph data. The system incorporates memory management mechanisms and scalable processing techniques:

- *Batch Processing:* Intelligent batching strategies to prevent out-of-memory errors during training.
- *Memory Monitoring:* Real-time memory usage monitoring using psutil with automatic garbage collection.
- *Incremental Processing:* Support for incremental processing of large graphs with checkpoint recovery.
- *Resource Optimization:* Dynamic resource allocation and optimization based on available system resources.

e) Heterogeneous Graph Neural Network Architecture (HeteroGNN): this constitutes a specialized neural architecture engineered to process and learn from heterogeneous graph data.

Multi-Head Attention Implementation: This is used to model the diverse semantic relationships in heterogeneous graphs. The architecture leverages advanced Graph Attention Convolution (GATConv) layers with the following design considerations:

- *Attention Head Configuration*: Multiple attention heads (typically 4-8) for capturing diverse relationship patterns.
- *Edge-Type-Specific Attention*: Specialized attention mechanisms for different relationship types (DECLARES, CALLS, EXTENDS).
- *Dynamic Attention Weighting*: Adaptive attention weights that adjust based on graph structure and the learning progress.
- *Attention Dropout*: Dropout strategies (typically 0.5) for regularization and generalization.

Message Passing for Heterogeneous Structures: The message passing paradigm is extended to accommodate structural heterogeneity, enabling effective information flow across diverse node and edge types:

- *Type-Specific Message Functions*: Different message computation functions for each relationship type.
- *Hierarchical Message Aggregation*: Multi-level aggregation strategies that capture both local and global graph patterns.
- *Temporal Message Patterns*: Support for temporal relationship patterns where available.
- *Bidirectional Message Passing*: Advanced bidirectional message passing for capturing complex dependency patterns

f) Advanced Training Configuration: To ensure model convergence, a multi-faceted training strategy is employed. This includes configured optimizers, loss functions tailored to class imbalance, and long-term training protocols with adaptive control mechanisms.

Optimizer Configuration: The optimization process is guided by adaptive strategies that enhance convergence stability and learning performance:

- *Adam Optimizer:* Advanced Adam implementation with learning rate 0.001 and adaptive moment estimation.
- *Learning Rate Scheduling*: ReduceLROnPlateau scheduler with sophisticated plateau detection and learning rate adaptation.
- *Gradient Clipping*: Advanced gradient clipping strategies to prevent gradient explosion in deep graph networks.
- *Weight Initialization*: Weight matrices are initialized using graph-aware strategies.

Loss Function Integration: To address severe class imbalance and optimize learning for minority classes, a sophisticated loss configuration is applied:

- *Focal Loss Implementation*: Gamma parameter of 2 for hard example mining with adaptive class weighting.
- *Balanced Class Weights*: Dynamic class weight computation based on current class distributions.
- *Loss Regularization*: Additional regularization terms for graph structure preservation.
- *Multi-Task Learning*: Support for auxiliary tasks that improve pattern detection performance

Extended Training Protocol: A training loop with fault-tolerance and performance tracking ensures sustainable learning across extended sessions:

- *Extended Epoch Training:* Up to 1000 epochs with careful overfitting monitoring.
- *Early Stopping Strategy:* Patience of 50 epochs with multiple stopping criteria.
- *Model Checkpointing:* Regular model checkpointing with best model preservation.
- *Training Resumption:* Support for training interruption and resumption with full state preservation.

g) Evaluation and Analysis: The evaluation framework enables an analysis of model performance through both metric-based assessment and visualization-based interpretability.

Multi-Metric Evaluation: A set of performance metrics is employed to account for the challenges posed by class imbalance and varying pattern complexities:

- *Weighted Metrics*: Class-weighted accuracy, precision, recall, and F1-scores to handle class imbalance.
- *Per-Class Analysis*: Detailed per-class performance metrics for identifying pattern-specific detection capabilities.
- *Confusion Matrix Analysis*: Confusion matrix analysis with statistical significance testing.
- *Confidence Score Analysis*: Distribution analysis of prediction confidence scores.

Visualization: The framework integrates visualizations to support interpretability and diagnostic analysis:

- *Confusion Matrix Heatmaps:* Confusion matrix visualizations with statistical annotations are offered
- *Graph Structure Visualization:* Network visualizations of learned graph representations can be shown.

3) SVM Integration: The Support Vector Machine implementation TrainSVM.py provides a complementary approach to the graph-based methods via high-dimensional feature space analysis. It focuses on leveraging rich numerical feature representations for pattern classification.

D. DPD and Ensembles Module (M3)

The DPD and Ensembles Module implements the ensemble technique combining predictions from multiple trained models, or if desired utilizing a single model. It currently applies soft voting and confidence-based weighting, but can apply any other EM technique.

4) Architecture: a stacked ensemble learning technique is applied in order to improve predictive performance by combining multiple base learners through a higher-level model known as a meta-classifier. This can be effective when the individual base models capture different aspects of the data or exhibit complementary strengths and weaknesses. The Detection.py module implements the stacked ensemble learning technique for DPD.

The process begins with a set of base models, which are independently trained on the same training data. Each model generates an individual model detection (i.e., predictions) for the input instances, resulting in a vector of intermediate outputs. These predictions are then passed to a meta-classifier, which is trained to learn a higher-order decision function based on the outputs of the base models (currently soft voting is utilized, but this can be adjusted). The meta-classifier effectively integrates the diverse perspectives of the underlying models and produces an ensemble-based detection (i.e., ensemble prediction) P_f with generalization capabilities.

5) Individual Model Detection: The detection system provides unified interfaces for applying trained models. The pipeline orchestrates: numerical feature extraction, standardization using pre-computed parameters, and graph feature merging. The following individual models are used:

SVM Detection: the function loads standardized features, applies the trained model, and handles edge cases (e.g., zero-feature vectors assigned "No Pattern", low-confidence predictions marked "Unknown").

GNN Detection: the function converts class-level graph features to PyTorch Geometric Data objects and applies trained GNN models.

NGDB Detection: the function leverages Memgraph for real-time graph queries and applies trained HeteroGNN models.

6) Ensemble-Based Detection: The ensemble methodology combines multi-model predictions (our current implementation utilizes SVM, GNN, and NGDB models via soft voting), while addressing different label spaces through label alignment. The ensemble process is as follows:

i. Common Class Identification: Finds classes that exist in all model outputs.
ii. Label Space Unification: Creates a unified label space containing all unique pattern classes.
iii. Probability Alignment: Maps each model's outputs to the unified space.
iv. Apply ensemble technique: Various techniques can be applied to combine the predictions of the base models, include stacking, bagging, boosting, etc. Soft voting is currently used.
v. Post-processing: Applies confidence thresholding and special-case handling. A function generates aligned probability matrices in which each row corresponds to a class instance and each column to a specific design pattern type. This alignment ensures that the probabilistic outputs from different models are directly comparable. Soft voting prevents bias toward patterns in a subset of models.

E. System Integration and API Module (M4)

7) Web API: The complete system is orchestrated through a FastAPI backend, providing RESTful endpoints. The system provides several key endpoints, including:

- POST /process-features for complete feature processing pipeline,
- POST /train-gnn, /train-pattern, and /train-svm for model training,
- POST /start-detection for model inference with ensemble support, and
- GET /get-metrics for model evaluation data.

The API includes error handling, asynchronous processing support, and Cross-Origin Resource Sharing (CORS) middleware for frontend integration.

8) Robustness and Integrity: Robustness, reproducibility, and overall system integrity are addressed via several supporting mechanisms in the pipeline. These include:

- Error Handling: includes file I/O validation, memory management - particularly for graph processing, and proper cross-validation data partitioning.
- Configuration Management: All configurations are serialized as JSON, including hyperparameters, training settings, and timing information to ensure reproducibility.
- Validation Framework: Multi-stage validation encompasses feature extraction consistency checks, standardization validation, and model validation with convergence analysis and overfitting detection.
- Feature Consistency: Validation of feature consistency across the graph includes range checks for feature values and detection of outliers, consistency validation between different feature sources, missing feature detection and imputation strategies, and feature distribution analysis with normalization validation.
- Label Quality: In training mode, label quality assurance encompasses label consistency validation across related classes, detection of potentially mislabeled instances, analysis of label distribution and balance, and validation of ground truth quality and completeness.

9) Output Generation and Format Optimization: The output generation process creates JSON files containing the class-level graph structure and essential metadata. This stage ensures that both the structural and contextual information are retained and formatted for efficient downstream use. Key aspects include:

Format Optimization: The JSON output is optimized for downstream processing through efficient encoding of graph structures for fast loading, preservation of metadata for debugging and analysis, compatibility with PyTorch Geometric data formats, and support for incremental loading and processing of large graphs.

Metadata Preservation: Metadata is preserved to support analysis and debugging, including original method-level information for traceability, transformation parameters and configuration settings, quality metrics and validation results, and processing timestamps with version information.

The resulting output enables efficient graph-based ML operations while maintaining the semantic relationships

present in the original code structure, creating a foundation for DPD using GNN and other techniques.

10) User Interface (UI): The frontend is implemented via Node.js with React, Vite.js, the Material UI React component, and Axios for Web APIs. Once a DPD repository is uploaded as a zip, it can be selected via dropdown (here PMD), as shown on the left in Figure 11. Thereafter, extract features can be executed (left bottom). Then the various ensemble models can be selected on the right, and then the ensemble detection can be started. At the top, the menu offers dataset management, training, models, and detection management.

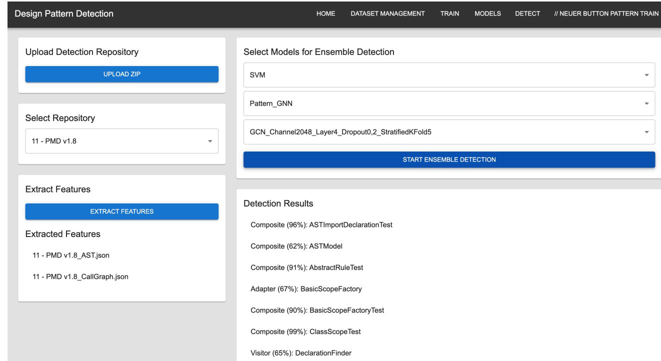

Figure 11. DPD user interface.

VI. Evaluation

The evaluation focuses on the effectiveness of our DPD-NGDB approach, analyzing GNN variants and DPD performance.

As described previously, our DPD approach is structured to inherently support ensemble methods for maximum flexibility. However, currently, all other models we attempted (GNN, SVM) for the ensemble in preliminary evaluations performed far worse or lacked consistency, and thus did not complement DPD-NGDB. Hence, our DPD-EM results are not evaluated here. As EM depends on the other models improving results, future work will focus on finding and tuning alternative models such that they address the misclassifications found in our DPD-NGDB approach.

All experiments were conducted on an Apple MacBook with M2 Pro with 32GB RAM.

A. Dataset Description

Both the training and evaluation are conducted on a dataset consisting exclusively of 23 canonical GoF design patterns and other non-labeled code. The dataset utilizes 9 projects from the Pattern-like Micro-Architecture Repository (P-MARt) repository [7]. As not all 23 GoF design patterns were exemplified in P-MARt, the dataset was supplemented with 23 pattern implementation examples from Refactoring Guru [53], which, while isolated examples without a larger project context, at least provide some training data.

The evaluation focuses on classifying any single class as a pattern instance to emphasize discriminative performance among the design patterns, and unlabeled classes are assumed to be "No Pattern" or unknown. Note that for a single pattern, multiple classes may participate (e.g., Observer), while a single class might participate in or utilize multiple patterns simultaneously (e.g., Factory and Observer). For simplicity and the labeled ground truth basis, classes are assumed to be associated with only a single pattern (e.g., either Factory Method or Observer, not both). Thus, the actual number of static pattern instantiations could be far less than the number of classes identified (labeled) as participating in a certain pattern (e.g., Observer, Broker).

The dataset comprises 30 projects with 417 unique samples across the 23 patterns, with varying distribution across pattern categories (121 creational, 182 structural, and 114 behavioral) as shown in TABLE I. Hence, the creational, structural, and behavioral design pattern types are included in the training and detection dataset with varying degrees of pattern sample frequency.

TABLE I. GoF Dataset

Pattern	Samples
Creational	**121**
Abstract Factory	72
Builder	28
Factory Method	8
Singleton	12
Prototype	1
Structural	**182**
Adapter	62
Bridge	28
Composite	58
Decorator	8
Facade	17
Flyweight	4
Proxy	5
Behavioral	**114**
Chain of Responsibility	5
Command	16
Interpreter	5
Iterator	19
Mediator	9
Memento	12
Observer	20
State	6
Strategy	7
Template Method	2
Visitor	13
Total	**417**

```
55   4 - Netbeans v1.0.x
56   Number of classes: 2238
57   Number of ghosts: 3244
58   Number of interfaces: 320
59   Number of association relationships: 41895
60   Number of aggregation relationships [1,n]: 1750
61   Number of aggregation relationships [1,1]: 6990
62   Number of composition relationships: 0
63   Number of container-aggregation relationships [1,n]: 161
64   Number of container-aggregation relationships [1,1]: 430
65   Number of container-composition relationships: 0
66   Number of creation relationships: 14131
67   Number of use relationships: 23355
68   Number of fields: 16736
69   Number of methods: 25446
70   Number of message sends: 0
71   Number of pattern models: 0
```

Figure 12. Snippet of P-MARt Netbeans project summary metrics.

An example of the pattern summary metrics per project in P-MARt is shown in Figure 12. Pattern-specific metrics are

also summarized in XML as shown in Figure 13. The XML-based documentation describes on a per-project basis the classes involved in a micro-architecture (pattern), as shown in Figure 14. We extracted the information per project to use for class labeling as our ground truth as shown in Figure 15.

```
194    Adapter
195        Adaptee: 65
196        Adapter: 86
197        Target: 37
198        Client: 64
199        -> Distribution of the micro-architectures per program:
200            JHotDraw v5.1: 1
201            JRefactory v2.6.24: 17
202            MapperXML v1.9.7: 2
203            Netbeans v1.0.x: 8
204            Nutch v0.4: 2
205            PMD v1.8: 1
206        -> Number of micro-architectures: 31
207        -> Number of roles: 4
208        -> Number of classing playing a role: 252
```

Figure 13. Snippet of P-MARt Adapter pattern summary.

```
<program type="Java">
  <name>3 - JRefactory v2.6.24</name>
  <designPattern name="Adapter">
    <microArchitectures>
      <microArchitecture number="13">
        <roles>
          <clients>
            <client roleKind="AbstractClass"><entity>org.acm.seguin.awt.OrderableList</entity></client>
          </clients>
          <targets>
            <target roleKind="AbstractClass"><entity>java.awt.event.ActionListener</entity></target>
          </targets>
          <adapters>
            <adapter roleKind="Class"><entity>org.acm.seguin.awt.MoveItemAdapter</entity></adapter>
          </adapters>
          <adaptees>
            <adaptee roleKind="Class"><entity>org.acm.seguin.awt.OrderableListModel</entity></adaptee>
          </adaptees>
        </roles>
      </microArchitecture>
```

Figure 14. Snippet of XML-based P-MARt [7] documentation of Adapter pattern in JRefactory project involving the MoveItemAdapter class.

```
1    {
2      "classes": {
3        "OrderableList": "Adapter",
4        "ActionListener": "Adapter",
5        "MoveItemAdapter": "Adapter",
6        "OrderableListModel": "Adapter",
7        "CafeSetup": "Adapter",
8        "ReloadActionAdapter": "Adapter",
9        "MultipleDirClassDiagramReloader": "Adapter",
10       "CommandLineMenu": "Adapter",
11       "ZoomAdapter": "Adapter",
12       "JumpToTypeAdapter": "Adapter",
13       "SourceBrowserAdapter": "Adapter",
14       "NewProjectAdapter": "Adapter",
15       "RefactoringAdapter": "Adapter",
16       "UndoAdapter": "Adapter",
17       "NodeViewer": "Factory Method",
18       "UMLNodeViewer": "Factory Method",
19       "NodeViewerFactory": "Factory Method",
20       "UMLNodeViewerFactory": "Factory Method",
21       "ReloaderSingleton": "Singleton",
22       "JavaParserVisitor": "Visitor",
23       "ChildrenVisitor": "Visitor",
24       "AddFieldVisitor": "Visitor",
25       "AddImplementedInterfaceVisitor": "Visitor",
26       "AddMethodVisitor": "Visitor",
27       "CompareParseTreeVisitor": "Visitor",
28       "EqualTree": "Visitor",
29       "LineCountVisitor": "Visitor",
30       "PrettyPrintVisitor": "Visitor",
31       "StubPrintVisitor": "Visitor",
32       "Node": "Visitor"
33     }
34   }
```

Figure 15. Snippet of our extracted JSON labels for the JRefactory project labeling MoveItemAdapter as class participating in Adapter pattern.

DPD results are output in our JSON format per project listing all classes, the primary pattern detected, a confidence value, and a project summary, as shown in Figure 16.

```
13347        "RERange": {
13348          "pattern": "No Pattern",
13349          "confidence": 1.0
13350        },
13351        "CharacterIterator": {
13352          "pattern": "Iterator",
13353          "confidence": 0.74
13354        }
13355      },
13356      "pattern_counts": {
13357        "No Pattern": 3305,
13358        "Adapter": 9,
13359        "Unknown": 8,
13360        "Abstract Factory": 13,
13361        "Factory Method": 1,
13362        "Iterator": 2
13363      }
13364    }
```

Figure 16. Snippet of our detection results per class and project summary.

Class imbalance across DPs was addressed via oversampling during training to ensure balanced representation. K-Fold cross-validation (K=5) is used for internal validation, with stratification to ensure balanced folds across the 23 pattern types.

B. NGDB GNN Variant Evaluation

The NGDB GNN model, implemented using Memgraph, integrates heterogeneous graph representations with GNN inference for real-time pattern detection, focusing on graph-derived features (e.g., inheritance metrics, call patterns) computed via Cypher queries. NGDB GNN variant performance was evaluated using Accuracy, Precision, Recall, and F1-Score. F1-Score was used due to dataset imbalance across pattern types. The overall performance across all patterns for each variant, averaged over the 5-fold CV is summarized in TABLE II.

TABLE II. NGDB GNN VARIANT PERFORMANCE (STRATIFIED 5-FOLD CV)

Model	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
GCN	93.45	90.67	93.45	91.13
GAT	97.72	97.73	97.72	97.68
GINSAGE	97.66	97.61	97.66	97.44

GAT achieved the highest F1-Score (97.68%) and precision (97.73%), followed closely by GINSAGE (F1: 97.44%, precision: 97.61%). GCN performed slightly lower, with an F1-Score of 91.13%. These results show substantial improvement over the standard GNN variants, meeting or exceeding the target F1-Score of 0.80 (80%) specified in the non-functional requirements (NFR3).

Pattern-specific performance from the classification reports shows high F1-scores for most patterns, such as Singleton (1.0 across variants), Proxy (1.0), and Chain of Responsibility (1.0). Abstract Factory achieved strong scores (GAT: 0.86, GINSAGE: 0.80, GCN: 0.15), while Adapter had lower performance in GCN (0.0) but improved in GAT (0.65) and GINSAGE (0.61). Behavioral patterns like Observer (GAT: 0.86, GINSAGE: 0.67, GCN: 0.0) and Command

(GAT: 0.75, GINSAGE: 0.86, GCN: 0.22) showed variability, reflecting challenges in capturing dynamic interactions.

Analysis: The results indicate that GAT outperforms other NGDB variants, likely due to its attention mechanism effectively weighting graph-derived features like call patterns and inheritance degrees. GINSAGE also performed well, combining isomorphism testing with inductive learning for robust generalization across diverse pattern implementations. GCN, while solid, lagged slightly, possibly due to less sophisticated aggregation in heterogeneous graphs, which limited its ability to capture complex relationships in patterns like Observer (F1: 0.0).

The high overall F1-Scores (91.13% – 97.68%) demonstrate the value of NGDB's Cypher-derived features in capturing relational aspects missed by standard GNNs, particularly for behavioral patterns. For instance, Observer achieved an F1-Score of 0.86 in GAT, reflecting the benefit of inter-class communication metrics. Despite excluding 'No Pattern' instances during training, the evaluation metrics reflect strong discrimination among the 23 patterns, with oversampling mitigating imbalance effectively.

Confidence metrics show reduced overconfidence compared to standard GNNs, with wrong predictions having notably lower scores (e.g., GAT: 64.79% for wrong vs. 97.16% for correct). Patterns like Singleton, Proxy, and Chain of Responsibility achieved perfect F1-scores (1.0), benefiting from distinct structural signatures enhanced by NGDB features, such as inheritance hierarchies and method declarations. Lower performance on patterns like Command (GAT: 0.75, GINSAGE: 0.86) and Adapter (GAT: 0.65) may stem from variability in implementations or insufficient feature representation for nuanced behavioral interactions.

C. DPD Evaluation of DPD-NGDB

As GINSAGE and GAT performance was equivalent, DPD-NGDB utilized GINSAGE for the rest of the evaluation.

1) DPD Performance: The confusion matrix for the DPD-NGDB model across the entire dataset using 5-fold CV demonstrates strong discrimination across the 23 GoF patterns, as shown in Figure 17. The matrix reveals minimal misclassification, with most patterns correctly identified along the diagonal. Analysis of misclassifications requires a case-by-case deeper analysis as explained in the following discussion.

Project-specific DPD tests were performed on the various P-MARt projects on which it had been trained (except for Netbeans) as listed in TABLE IV. The DPD results for the projects in the training showed an accuracy range from 0.17 to .91, precision from 0.83 to 1, recall from 0.20 to 0.91, and F1 scores from 0.55 to 0.94.

When including the Netbeans test project (left out of training), the overall values were 0.45 for accuracy, 0.96 for precision, 0.58 for recall, and 0.72 for F1. Each project has a diverse set of patterns which were detected, as shown on the right of the table, covering 18 of the 23 GoF. The testing of the remaining 5 patterns was performed in the 5-fold CV and is shown in the confusion matrix. In addition, a confidence diagram for the 23 GoF patterns (including no-pattern and unknown) is shown in Figure 18. It shows that confidence values rarely go below 0.5, and that many or most of the misclassifications occur below complete confidence and have to do with determining a class is not involved in a pattern (No pattern). This determination can be difficult even for seasoned software engineers.

2) Leave-One-Project-Out Cross-Validation (LOPO-CV): The Netbeans project was left out of the training set which used 5-fold CV. Withholding Netbeans for testing evaluates DPD for unseen test data, with the results shown in TABLE III. The F1 score of 0.17 shows relatively poor performance over the four patterns with an overall accuracy of 0.03. This shows potential issues with the generalization of our DPD results as described in the following discussion, and our relatively small training dataset may be a factor.

TABLE III. NETBEANS TEST RESULT (LOPO-CV)

Pattern	Classes	TP	FP	FN	Precision	Recall	F1
Abstract Factory	171	1	3	170	0.25	0.01	0.01
Adapter	81	7	3	74	0.70	0.09	0.15
Command	3	1	2	2	0.33	0.33	0.33
Iterator	12	0	1	12	0	0	-
Overall	267	9	9	258	0.32	0.11	0.17

3) Feature Importance: The SHAP values can provide insight into which features contributed the most to the model predictions, and is shown in Figure 19. The minimum number of unique instantiations of a class was the strongest, followed by the mean number of outgoing calls, followed by the number of fields in the class. This was followed by a number of call-related metrics. This may be an indication that the graph structure together with these metrics provide influence the ability to distinguish the various patterns.

D. Discussion

1) NGDB GNN variants evaluation: patterns like Singleton, Proxy, and Chain of Responsibility were detected most reliably, achieving perfect F1-scores (1.0) across NGDB variants. This is likely due to their distinct structural signatures such as private constructors and static access methods for Singleton. These features are well-captured by graph-derived metrics like inheritance relationships and call patterns computed via Cypher queries in Memgraph. Structural patterns like Composite and Abstract Factory also showed strong performance in NGDB (e.g., GAT F1: 0.86 for Abstract Factory), benefiting from the heterogeneous graph representations that emphasize class-to-class dependencies and method aggregations. In contrast, behavioral patterns such as Observer and Command exhibited more variability, with lower F1-scores in GCN (0.0 for Observer, 0.22 for Command) but improvements in GAT (0.86 for Observer, 0.75 for Command) and GINSAGE (0.67 for Observer, 0.86 for Command). This suggests that attention mechanisms in GAT and inductive learning in GINSAGE better handle dynamic interactions, though the static nature of the analysis limits full capture of runtime behaviors. The standard GNN variants (GCN, GAT, GINSAGE) performed poorly overall (F1: 3.54% – 7.19%), with GINSAGE slightly outperforming others due to its ability to generalize to unseen graphs, but still struggling with imbalance and feature noise.

In contrast, standard non-NGDB GNN variants (GCN, GAT, GIN, GraphSAGE) yielded poor performance far below the target, due to dataset imbalance and limited feature representation. Thus, the ensemble method, combining SVM, GNN, and NGDB via soft voting, underperformed compared to the NGDB base model alone, primarily due to the suboptimal results of GNN and SVM, which introduced noise and biased predictions. This validates the hypothesis that ensemble methods can enhance robustness only when base models are sufficiently strong, highlighting NGDB's critical role in achieving high accuracy.

2) Ensemble Methods evaluation: The methodology successfully extended the ensemble method, intended to combine SVM's feature separation, GNN's structural resilience, and NGDB's graph-derived insights, but underperformed relative to the standalone NGDB approach. It was thus not included in this evaluation. This highlights a key strength of ensembles in theory – leveraging complementary models to reduce variance and improve robustness – but in practice, the weak base models (perhaps due to the sparse training datasets) diluted NGDB's high performance. Further multi-model investigation and fine-tuning of ensembles is included in future work, as we believe it to hold promise for addressing NGDB misclassifications once their causes are apparent.

3) Requirements coverage:

- FR1: All 23 GoF DPs were detected across 9 open-source Java projects with documented DPs from the P-MARt dataset with Refactoring Guru implementations, which supporting class-level labeling and benchmarking.
- FR2: Multi-modal features including numerical, graph-based (ASTs, CGs), and derived structural metrics were extracted from the source code. from the base implementation and incorporating multi-modal features (numerical, graph-based, structural).
- FR3: Training pipelines for individual base models (e.g., SVM, GNN, NGDB) and an ensemble combiner (e.g., soft voting) were implemented.
- FR4: The P-MARt dataset was extended with additional patterns to cover all GoF patterns, class-level labeling was applied, and imbalance handling was addressed via oversampling.
- FR5: An evaluation framework was implemented that offers cross-validation (K-Fold, Leave-One-Project-Out (LOPO)) and calculates performance metrics (F1-Score, confidence). The use of stratified K-Fold (K=5) cross-validation supports the generalization of the approach, though challenges like dataset imbalance and implementation variability persisted.
- FR6: The pipelines were implemented to support batch processing of multiple repositories and real-time pattern detection for integration into development workflows was enabled via memory monitoring.
- NR1: DPD model execution performance was reasonable, within minutes on standard hardware for medium-size projects. Model training involved more time and resources but was within reasonable expectations.
- NR2: Scalability was supported via memory and resource management and optimization, e.g., via batching strategies that account for resource limitations (avoiding out-of-memory errors during training), incremental processing of large graphs with checkpoint recovery, etc.
- NR3: While F1-scores above the target of 0.80 were achieved for various P-MARt projects, the overall score was 0.72 for the GoF spectrum on over 5300 classes; robustness against code variations for unseen projects needs further work, exemplified with the large 3347 class Netbeans project that had 267 classes participating in four different DP types with a resulting F1 score of 0.17.
- NF4: Extensibility was achieved via a modular architecture that allows for the automatic addition of new pattern types, automated feature extractors, and a flexible ensemble inclusion of further base models.
- NF5: Reproducibility was addressed via logging of parameters (including random seed management) and versioning of models, datasets, and intermediate and final output results.

4) Summary: The DPD-NGDB evaluation showed that it is feasible to automatically train DPD-NGDB on a labeled training set of the 23 GoF patterns, and for it to detect the spectrum of patterns when it comes across these again. Overall, the results validate the use of NGDB for real-time, relational pattern detection, aligning with functional requirements for covering all 23 patterns and achieving high accuracy. The AST and CG features proved resilient for the different pattern types (creational, structural, and behavioral).

As to limitations, as encountered with LOPO-CV using Netbeans, DPD on unseen datasets can be challenging and further investigation regarding misclassification factors and testing on diverse datasets is needed. The relatively sparse training set and static-only analysis limit performance. Furthermore, discriminative challenges are presented among similar patterns, some differing primarily in intention. The reliance on static analysis (ASTs, CGs) limits detection of dynamic behaviors in patterns (e.g., Observer, Strategy), where runtime interactions (e.g., method invocations via reflection) might provide better hints. While dynamic analysis is promising for revealing hidden dependencies, it was not incorporated due to execution risks, high-overhead setup and execution costs, and coverage issues. In particular, pattern dataset limitations further constrain generalizability: the examples offer textbook implementations or intentional pattern-centric projects, lacking real-world diversity, partial realizations, or obfuscation variants (as initially planned but not fully evaluated). Imbalance among patterns persisted despite oversampling, skewing performance toward common patterns like Singleton and underrepresenting rare ones like Flyweight or Interpreter. Deeper analysis is required to understand any pattern misclassifications, as this could be due to multiple causes. E.g., the same object participating in multiple patterns, cases where the pattern structure differs primarily by its intent or purpose (highly context-sensitive), information about dynamic interactions is missing (e.g., due to reflection), etc. This deeper case-by-case analysis is included in future work. Environment limitations include the dependency on the Memgraph setup, which adds complexity and potential memory overhead for large heterogeneous graphs.

VII. CONCLUSION

This paper developed and evaluated our automated design pattern detection approach DPD-NGDB, based on a neural graph database and modular pipeline architecture, and made available as an ensemble base model in our DPD-EM ensemble model approach. The evaluation was benchmarked against a dataset with the 23 GoF design patterns contained in over 5300 Java classes spanning 9 open-source realistic practical Java projects (plus an additional 23 single example patterns). The dataset offered independently documented DPs and were used for the automated DPD training and testing. With 5-fold cross-validation and leave one project out, an overall F1 score of 0.72 was achieved, while the large unseen test project achieved 0.17. Furthermore, three NGDB GNN variants were evaluated, with GAT and GINSAGE showing similarly high F1 scores, and GCN somewhat lower.

The use of multi-modal features (numerical from ASTs, graph-based from CGs, structural like inheritance) outperforms traditional ML approaches (e.g., Uchiyama et al. [29], Dwivedi et al. [30]) by capturing both syntactic and relational aspects. DPD-NGDB's high F1-scores (up to 94%) surpass reported benchmarks for tools like PINOT, which achieve high precision only for well-structured code. The DPD-EM ensemble approach, while not as effective as the NGDB base model yet, provides an ongoing framework for the combination of diverse base models, contributing to the exploration of ensemble methods for DPD in software engineering [48].

Future work includes experimentation with various ensemble base models and methods; integrating runtime tracing for dynamic analysis; combining static features with execution traces to better detect behavioral patterns; full obfuscation; scalability evaluation; variant detection; usage on generalized open-source projects; expansion beyond the GoF patterns; an evaluation across multiple programming languages; and a comprehensive industrial case study.

ACKNOWLEDGMENT

The authors would like to thank Natalie Böhm, Marius Mühleck, and Sandro Moser for their assistance with various aspects of the design, implementation, and evaluation.

REFERENCES

[1] R. Oberhauser and S. Moser, "Design Pattern Detection in Code: A Hybrid Approach Utilizing a Bayesian Network, Machine Learning with Graph Embeddings, and Micropattern Rules," Proceedings of the Eighteenth International Conference on Software Engineering Advances (ICSEA 2023), IARIA, 2023, pp. 122-129.

[2] R. Minelli, A. Mocci, and M. Lanza, "I know what you did last summer: an investigation of how developers spend their time," In: Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension, IEEE Press, 2015, pp. 25-35.

[3] M. J. Pacione, M. Roper, and M. Wood, "A novel software visualisation model to support software comprehension," In: Proc. 11th Working Conference on Reverse Engineering, IEEE, 2004, pp. 70-79

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Pearson, 1995.

[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, Pattern-Oriented Software Architecture: A System of Patterns, Vol. 1. John Wiley & Sons, 2008.

[6] Y. Guéhéneuc and G. Antoniol. "Fingerprinting Design Patterns". In: 11th Working Conference on Reverse Engineering (WCRE), IEEE, 2004, pp. 172–181. DOI: 10.1109/WCRE.2004.3

[7] Y. G. Guéhéneuc, "P-mart: Pattern-like micro architecture repository," Proceedings of the 1st EuroPLoP Focus Group on pattern repositories, pp. 1-3, 2007.

[8] P-MARt Pattern-like Micro-Architecture Repository. [Online]. Available from: https://www.ptidej.net/tools/designpatterns 2025.11.28

[9] M. G. Al-Obeidallah, M. Petridis, and S. Kapetanakis, "A survey on design pattern detection approaches," International Journal of Software Engineering (IJSE), 7(3), 2016, pp. 41-59.

[10] H. Yarahmadi and S. M. H. Hasheminejad, "Design pattern detection approaches: A systematic review of the literature," Artificial Intelligence Review, 53, 2020, pp. 5789-5846.

[11] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner and G. Monfardini, "The Graph Neural Network Model," in IEEE Transactions on Neural Networks, vol. 20, no. 1, 2009, pp. 61-80, doi: 10.1109/TNN.2008.2005605.

[12] M. Besta et al., "Neural Graph Databases," Proceedings of the First Learning on Graphs Conference (LoG 2022), PMLR 198, Virtual Event, 2022.

[13] H. Ren, M. Galkin, M. Cochez, Z. Zhu, and J. Leskovec, "Neural Graph Reasoning: Complex Logical Query Answering Meets Graph Databases," In: ArXiv abs/2303.14617, 2023.

[14] N. Francis et al., "Cypher: An evolving query language for property graphs," Proc. 2018 International Conference on Management of Data, 2018, pp. 1433-1445.

[15] T. G. Dietterich, "Ensemble Methods in Machine Learning," In: Multiple Classifier Systems. Springer, 2000, pp. 1-15. DOI: 10.1007/3-54045014-9_1.

[16] G. Seni and J. F. Elder, "Ensemble Methods in Data Mining: Improving Accuracy Through Combining Predictions," In: Synthesis Lectures on Data Mining and Knowledge Discovery, Morgan & Claypool Publishers, 2010. DOI: 10.2200/S00240ED1V01Y200912DMK002.

[17] R. Oberhauser, "A Machine Learning Approach Towards Automatic Software Design Pattern Recognition Across Multiple Programming Languages," Proc. of the Fifteenth International Conference on Software Engineering Advances (ICSEA 2020), IARIA XPS Press, 2020, pp. 27-32.

[18] R. Oberhauser, "A Hybrid Graph Analysis and Machine Learning Approach Towards Automatic Software Design Pattern Recognition Across Multiple Programming Languages, " International Journal on Advances in Software, vol. 15, no. 1 & 2, year 2022, pp. 28-42. ISSN: 1942-2628.

[19] D. Yu, Y. Zhang, and Z. Chen, "A comprehensive approach to the recovery of design pattern instances based on sub-patterns and method signatures," Journal of Systems and Software, vol. 103, pp. 1-16, 2015.

[20] B. Mayvan and A. Rasoolzadegan, "Design pattern detection based on the graph theory," Knowledge-Based Systems, vol. 120, pp. 211-225, 2017.

[21] M. L. Bernardi, M. Cimitile, and G. Di Lucca, "Design pattern detection using a DSL-driven graph matching approach," Journal of Software: Evolution and Process, 26(12), pp. 1233-1266, 2014.

[22] M. Oruc, F. Akal, and H. Sever, "Detecting design patterns in object-oriented design models by using a graph mining approach," 4th International Conference in Software Engineering Research and Innovation (CONISOFT 2016), IEEE, 2016, pp. 115-121.

[23] A. Pande, M. Gupta, and A. K. Tripathi, “A new approach for detecting design patterns by graph decomposition and graph isomorphism,” International Conference on Contemporary Computing, Springer, Berlin, Heidelberg, 2010, pp. 108-119.

[24] P. Pradhan, A. K. Dwivedi, and S. K. Rath, “Detection of design pattern using graph isomorphism and normalized cross correlation,” Eighth International Conf. on Contemporary Computing (IC3 2015), IEEE, 2015, pp. 208-213.

[25] S. Alhusain, S. Coupland, R. John, and M. Kavanagh, “Design pattern recognition by using adaptive neuro fuzzy inference system,” 2013 IEEE 25th International Conference on Tools with Artificial Intelligence, IEEE, 2013, pp. 581-587.

[26] M. Zanoni, F. A. Fontana, and F. Stella, “On applying machine learning techniques for design pattern detection,” J. of Systems & Software, vol. 103, no. C, pp. 102-117, 2015.

[27] L. Galli, P. Lanzi, and D. Loiacono, “Applying data mining to extract design patterns from Unreal Tournament levels,” Computational Intelligence and Games. IEEE, 2014, pp. 1-8.

[28] R. Ferenc, A. Beszedes, L. Fulop, and J. Lele, “Design pattern mining enhanced by machine learning,” 21st IEEE International Conference on Software Maintenance (ICSM'05), IEEE, 2005, pp. 295-304.

[29] S. Uchiyama, A. Kubo, H. Washizaki, and Y. Fukazawa, “Detecting design patterns in object-oriented program source code by using metrics and machine learning,” Journal of Software Engineering and Applications, 7(12), pp. 983-998, 2014.

[30] A. K., Dwivedi, A. Tirkey, and S. K. Rath, “Software design pattern mining using classification-based techniques,” Frontiers of Computer Science, 12(5), pp. 908-922, 2018.

[31] H. Thaller, L. Linsbauer, and A. Egyed, “Feature maps: A comprehensible software representation for design pattern detection,” IEEE 26th international conference on software analysis, evolution and reengineering (SANER 2019), IEEE, 2019, pp. 207-217.

[32] A. Chihada, S. Jalili, S. M. H. Hasheminejad, and M. H. Zangooei, “Source code and design conformance, design pattern detection from source code by classification approach,” Applied Soft Computing, 26, pp. 357-367, 2015.

[33] Y. Wang, H. Guo, H. Liu, and A. Abraham, “A fuzzy matching approach for design pattern mining,” J. Intelligent & Fuzzy Systems, vol. 23, nos. 2-3, pp. 53-60, 2012.

[34] A. Alnusair, T. Zhao, and G. Yan, “Rule-based detection of design patterns in program code,” Int'l J. on Software Tools for Technology Transfer, vol. 16, no. 3, pp. 315-334, 2014.

[35] M. Lebon and V. Tzerpos, “Fine-grained design pattern detection,” IEEE 36th Annual Computer Software and Applications Conference, IEEE, 2012, pp. 267-272.

[36] I. Issaoui, N. Bouassida, and H. Ben-Abdallah, “Using metric-based filtering to improve design pattern detection approaches,” Innovations in Systems and Software Engineering, vol. 11, no. 1, pp. 39-53, 2015.

[37] A. K. Dwivedi, A. Tirkey, and S. K. Rath, “Software design pattern mining using classification-based techniques,” Frontiers of Computer Science, 12(5), pp. 908-922, 2018.

[38] F. A. Fontana, S. Maggioni, and C. Raibulet, “Understanding the relevance of micro-structures for design patterns detection,” Journal of Systems and Software, vol. 84, no. 12, pp. 2334-2347, 2011

[39] I. Issaoui, N. Bouassida, and H. Ben-Abdallah, “Using metric-based filtering to improve design pattern detection approaches. Innovations in Systems and Software Engineering,” vol. 11, no. 1, pp. 39-53, 2015.

[40] J. Dong, Y. Zhao, and Y. Sun, “A matrix-based approach to recovering design patterns,” IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans, vol. 39, no. 6, pp. 1271-1282, 2009.

[41] J. Singh, S. R. Chowdhuri, G. Bethany, and M. Gupta, “Detecting design patterns: a hybrid approach based on graph matching and static analysis,” Information Technology and Management, 23(3), pp. 139-150, 2022.

[42] R. Barbudo, A. Ramírez, F. Servant, and J. R. Romero, “GEML: A grammar-based evolutionary machine learning approach for design-pattern detection,” Journal of Systems and Software, 175, p. 110919, 2021.

[43] M. Kouli and A. Rasoolzadegan, “A Feature-Based Method for Detecting Design Patterns in Source Code,” Symmetry, 14(7), p. 1491, 2022.

[44] J. Liu et al., "Learning Graph-based Code Representations for Source-level Functional Similarity Detection," In: Proc. IEEE/ACM 45th International Conference on Software Engineering (ICSE). IEEE, 2023, pp. 345-357. DOI:10.1109/ICSE48619.2023.00040.

[45] A. Ampatzoglou et al., "Design patterns mining using neural subgraph matching". In: Proc. 37th ACM/SIGAPP Symposium on Applied Computing, ACM, 2022, pp. 1435-1444. DOI: 10.1145/3477314. 3507073.

[46] M. Li, Y. Wang, and W. Zhang, "A Multi-Feature Fusion Approach for Design Pattern Detection Based on Graph Neural Networks," In: 2024 IEEE International Conference on Software Services Engineering (SSE), IEEE, 2024, pp. 1-10. DOI: 10.1109/SSE62679.2024.10633498

[47] Memgraph. [Online]. Available from: https://memgraph.com 2025.11.28

[48] M. Y. Mhawish and M. Gupta, "Software Metrics and tree-based machine learning algorithms for distinguishing and detecting similar structure design patterns". In: SN Applied Sciences 2(1) p.11, 2020. DOI: 10.1007/s42452-019-1815-3.

[49] N. Shi and R. A. Olsson, "Reverse Engineering of Design Patterns from Java Source Code," 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06), Tokyo, Japan, 2006, pp. 123-134, doi: 10.1109/ASE.2006.57.

[50] M. Tufano et al., "An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation," In: ACM Transactions on Software Engineering and Methodology (TOSEM), 30(4), 2021, pp. 1-37. DOI: 10.1145/3450284.

[51] J. Dong, "Focal Loss Improves the Model Performance on Multi-Label Image Classifications with Imbalanced Data". In: Proceedings of the 2nd International Conference on Industrial Control Network and System Engineering Research, 2020, pp. 18-21.

[52] R. S. Pienta et al., "MAGE: Matching approximate patterns in richly attributed graphs". In: 2014 IEEE International Conference on Big Data (Big Data), IEEE, 2014, pp. 585-590.

[53] Refactoring.Guru. [Online]. Available from: https://refactoring.guru/design-patterns/java 2025.11.28

Confusion Matrix

Actual \ Predicted	Abstract Factory	Adapter	Bridge	Builder	Chain of Responsibility	Command	Composite	Decorator	Facade	Factory Method	Interpreter	Iterator	Mediator	Memento	No Pattern	Observer	Prototype	Proxy	Singleton	State	Strategy	Template Method	Unknown	Visitor
Abstract Factory	38	0	0	0	0	0	2	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
Adapter	0	85	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1
Bridge	0	0	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Builder	0	1	0	36	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Chain of Responsibility	0	0	0	0	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Command	0	0	0	0	0	27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Composite	0	0	0	0	0	0	87	0	0	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0
Decorator	0	0	0	0	0	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
Facade	0	0	0	0	0	0	0	0	10	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
Factory Method	0	1	0	0	0	0	0	0	0	26	0	0	0	0	1	0	0	0	0	0	0	0	0	0
Interpreter	0	0	0	0	0	0	0	0	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0
Iterator	0	0	0	0	0	0	0	0	0	0	0	12	0	0	2	0	0	0	0	0	0	0	0	0
Mediator	0	0	0	0	0	0	0	0	0	0	0	0	8	0	0	0	0	0	0	0	0	0	0	0
Memento	0	0	0	0	0	0	0	0	0	0	0	0	0	5	0	0	0	0	0	0	0	0	0	0
No Pattern	0	1	0	0	0	0	2	0	0	1	0	0	0	0	1146	0	0	0	0	0	0	0	2	0
Observer	0	0	0	1	0	0	0	0	0	0	0	0	0	0	7	20	0	0	0	0	0	0	0	0
Prototype	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0	0
Proxy	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0	0
Singleton	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	11	0	0	0	0	0
State	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	23	0	0	0	0
Strategy	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	18	0	0	0
Template Method	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	9	0	0
Unknown	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	284	0
Visitor	0	1	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	54

Figure 17. Confusion Matrix for DPD-NGDB GINSAGE showing strong DPD performance (diagonal) across the entire pattern dataset.

TABLE IV. DPD RESULTS USING P-MART TEST PROJECTS (LOPO-CV NETBEANS)

Project	Classes	Scored	Accuracy	Precision	Recall	F1	Pattern Types	Abstract Fac-	Adapter	Bridge	Builder	Command	Composite	Decorator	Facade	Factory	Iterator	Memento	Observer	Prototype	Singleton	State	Strategy	Template	Visitor
JHotDraw v5.1	153	116	0.91	0.99	0.91	0.94	10	x					x	x		x			x	x	x	x	x	x	
JRefactory v2.6.24	568	206	0.50	1.00	0.69	0.79	6		x		x					x					x	x			x
JUnit v3.7	89	55	0.69	1.00	0.54	0.68	4						x				x		x		x				
Lexi v0.1.1 alpha	97	16	0.31	1.00	0.53	0.88	3				x								x		x				
MapperXML v1.9.7	222	63	0.59	0.98	0.65	0.74	8	x	x				x		x				x		x		x	x	
Netbeans v1.0.x	3347	267	0.03	0.32	0.11	0.17	4	x	x			x					x								
Nutch v0.4	244	46	0.17	1.00	0.20	0.55	7		x	x		x					x	x			x		x		
PMD v1.8	472	55	0.71	0.94	0.61	0.79	8		x		x		x			x	x		x					x	x
QuickUML 2001	193	64	0.86	0.83	0.69	0.88	6	x			x	x	x						x		x				
OVERALL	5385	888	0.45	0.96	0.58	0.72	18	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

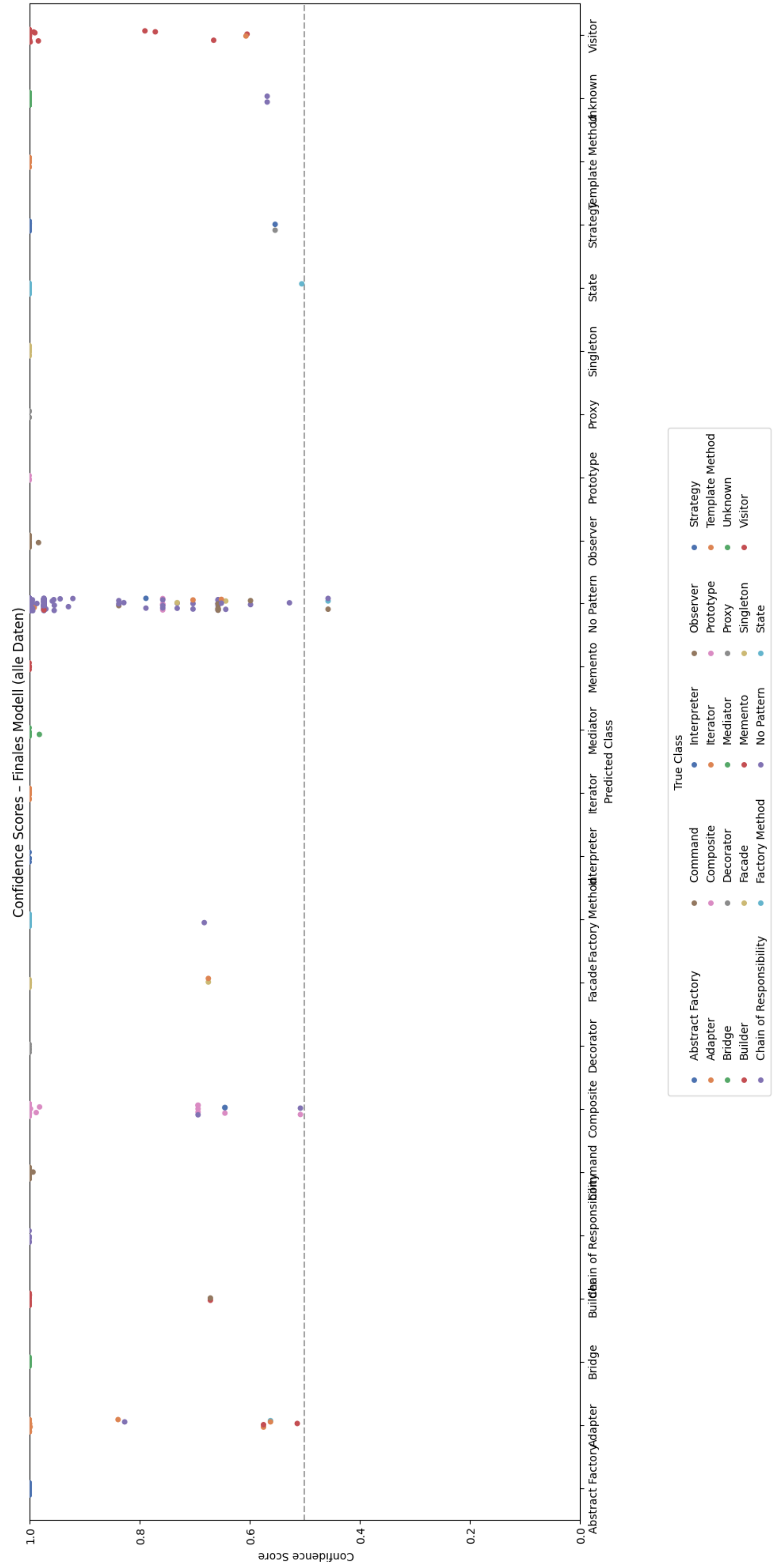

Figure 18. DPD-NGDB confidence scores for the entire dataset (NetBeans left out of training set).

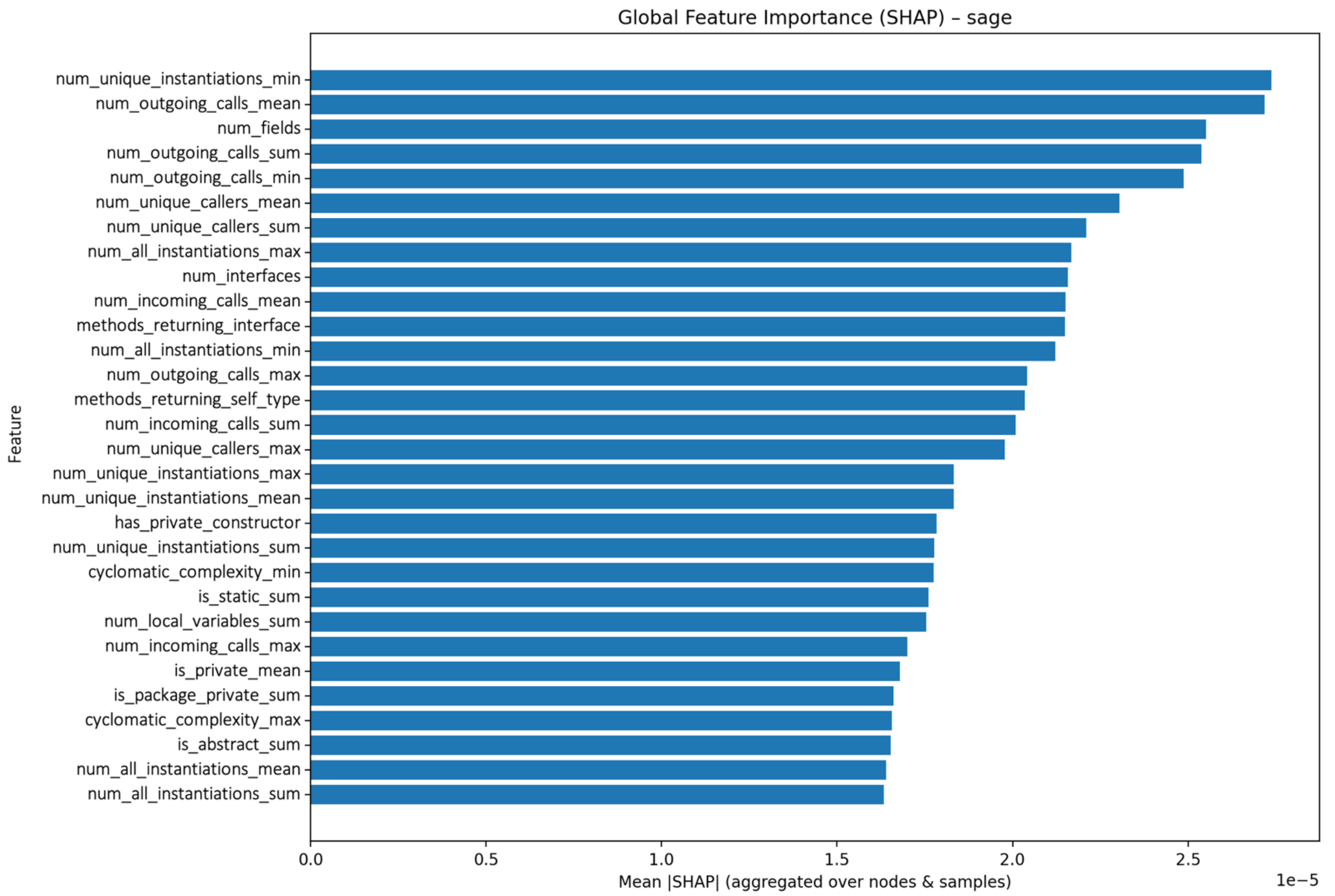

Figure 19. SHAP feature importance for DPD-NGDB across the entire dataset (generated diagram edited to replace feature numbers with names).