DOI: https://doi.org/10.15276/aait.08.2025.18
UDC 004.8:004.42:519.17

Hybrid graphs for code smells: a multi-level model for anti-pattern detection in software components

Dmytro D. Kurinko
ORCID: https://orcid.org/0000-0001-8304-3257; dmitrykurinko@gmail.com
Odesa Polytechnic National University, 1, Shevchenko Avenue. Odesa, 65044, Ukraine

ABSTRACT

The paper proposes a hybrid, multi-level method for detecting code smells and anti-patterns in software components, where structure, semantics, metrics, and evolution are treated as first-class signals. A heterogeneous Code Property Graph (Abstract Syntax Tree + Control-flow Graph + Program Dependence Graph) is constructed and enriched with textual embeddings from a pretrained code language model, classical quality metrics (Chidamber–Kemerer, Halstead), and version-control history (churn, co-change, recency). Local idioms are summarized via a sequence–graph encoder at the method/block level, component structure is aggregated by a relation-aware Graph Neural Network at the class/module level, and project context is propagated over a component-interaction graph. To support deployment in evolving codebases, an open-set head is introduced: energy, entropy, and stochastic variance are combined to enable calibrated abstention on unfamiliar patterns. The approach is evaluated on polyglot Java Virtual Machine corpora using time-aware, cross-project splits with multi-label targets (Long Method, God Class, Feature Envy, Data Class, Shotgun-Surgery–like, No-smell). Improvements in macro Area under the Precision–Recall Curve and F1 overrule/metric baselines, Abstract Syntax Tree-only, and text-only models are observed, while FPR@95TPR is maintained or reduced. Withheld-class experiments show that open-set gating increases Area under ROC for Open-Set Recognition and TNR@TPR and lowers calibration error, yielding probabilities suitable for thresholded automation and human triage. Cross-language transfer (train Java → test Kotlin/Scala) is shown to be stronger than with single-view models, aided by language-agnostic typing and per-project normalization. Incremental graph maintenance confines computation to changed regions, aligning inference time with CI/CD budgets. By exposing hierarchical attention and channel gates, explanations are produced that align with practitioner reasoning. It is concluded that hybrid graphs with hierarchical reasoning and selective prediction deliver detectors that are more accurate, transferable, and operationally safer for evolving software systems.

Keywords: Machine learning; software engineering; program analysis; graph representation learning; static analysis; uncertainty estimation; transfer learning; empirical evaluation

***For citation*: Kurinko D. D. "Hybrid graphs for code smells: a multi-level model for anti-pattern detection in software components". *Applied Aspects of Information Technology*. 2025; Vol.8 No.3: 274–285. DOI: https://doi.org/10.15276/aait.08.2025.18**

INTRODUCTION, FORMULATION OF THE PROBLEM

Industrial software systems are rapidly growing in size, language diversity, and rate of change. Under these conditions, maintainability is determined not only by formal correctness but also by the internal quality of code – specifically, by the presence or absence of anti-patterns (code smells) that impair readability, complicate testing, slow evolution, and increase defect risk [1]. Traditional detection tools built on fixed rules and threshold metrics are useful at early stages but show limited cross-project and cross-language transfer, are sensitive to idiosyncratic coding styles, and often either over-report (high false positives) or miss atypical manifestations (low recall). Moreover, contemporary development practices (CI/CD, code review, rapid release cycles) demand that analysis tools operate incrementally and account for change history, inter-component relations, and the broader evolution context [2].

Prior work on machine learning for code quality has typically focused on one representational plane: (i) structural models exploiting Abstract Syntax Trees (AST), Control-Flow Graphs (CFG), or Program Dependence Graphs (PDG) to capture formal structure; (ii) textual/semantic models that leverage token/identifier or LLM-based embeddings of code and comments; (iii) metric-oriented approaches (e.g., CK metrics, Halstead) that compress artifacts into engineered features; and (iv) historical/evolutionary approaches that track change frequency, churn, co-editing patterns, and diff-based indicators across versions. Despite successes along each line, anti-patterns are inherently multi-dimensional: the same structural smell may be "normal" within a project's style; textual naming may mask or reveal semantic idioms; and change history often signals latent issues (e.g., component instability) before they surface in static features [3].

Consequently, a key methodological gap is the absence of an integral representation that jointly and naturally models code structure, its semantic content, and the evolution context across multiple levels of abstraction – from local idioms to

ISSN 2617-4316 (Print)
ISSN 2663-7723 (Online)

component and project scope. A second gap is the lack of explicit open-set thinking: most models are trained to discriminate among a fixed set of smell classes and, at inference time, implicitly assume every instance belongs to one of the known classes. In real codebases that continuously evolve; a tool must be able to signal "unknown" or "out-of-distribution" when encountering atypical or novel anti-patterns. A third gap concerns multilinguality and transferability: models tuned for one language (e.g., Java) often degrade on closely related languages (Kotlin, Scala) due to differences in syntax, idioms, and library practices [4].

In this work, we propose a multi-level model for anti-pattern detection in software components that combines a hybrid graph representation of code with a project/component/local architecture and explicit handling of uncertainty in open-set scenarios. At its core lies a Hybrid Code Graph: we model code as a Code Property Graph (unifying AST, CFG, and PDG) and augment nodes/edges with textual-semantic embeddings (e.g., LLM-derived vectors for tokens, identifiers, and fragments) and quality metrics (CK, Halstead, simple complexity/length indicators). At the project level, the graph is enriched with evolutionary features: diff-based measures, change frequency, co-editing relationships, artifact "age," and stability. Over this representation, we employ a multi-level GNN architecture with attention: a local sequence submodel captures idioms within methods/blocks; a component-level encoder aggregates structural-semantic and metric signals on the component subgraph; and a project-level encoder models inter-component interactions and evolutionary context. Cross-level fusion is realized via hierarchical attention/gating to produce consistent predictions and calibrated uncertainty estimates [5].

A salient property of the approach is open-set handling. We instrument uncertainty heads (e.g., stochastic inference, energy-based scores, or ensembles) and calibrate decision thresholds to separate known smells from unknown/atypical manifestations. This capability is critical for realistic CI/CD and code-review pipelines, where it is safer to abstain (flagging a case as "needs attention") than to misclassify a novel artifact as one of the known classes. Additionally, by hybridizing feature sources and explicitly modeling evolution, the architecture achieves a favorable precision–recall trade-off: improved recall does not entail a substantial rise in false positives, and a "robust mode" supports transfer across related languages (Java/Kotlin/Scala) and coding styles.

1. BACKGROUND AND RELATED WORK

This section condenses prior art relevant to our hybrid, multi-level and open-set treatment of code smell / anti-pattern detection. We group works into six strands: (i) rule- and metric-based detectors; (ii) graph/structure learning; (iii) neural representation learning for code; (iv) evolution-aware analytics; (v) open-set and uncertainty; (vi) multilingual and cross-project transfer – and close with a synthesis that motivates our design [6].

Rule- and metric-based detectors of code smells / anti-patterns. Industrial practice long relied on rule engines (e.g., AST predicates, naming heuristics) and metric thresholds (CK, Halstead, coupling/cohesion) to flag smells such as *Long Method*, *God Class*, or *Feature Envy* [7].

These approaches are valued for speed and explainability but face persistent limits: (a) brittleness to local coding style and configuration; (b) difficulty encoding non-local or composite smells spanning multiple files; (c) poor portability across projects and languages; and (d) a familiar recall–precision trade-off – tight thresholds shrink false positives yet miss atypical manifestations, whereas relaxed thresholds inflate noise. Later works attempted adaptive thresholds or project-specific calibration, improving precision locally but not addressing deeper issues of semantics, structure–text interplay, or temporal context [8].

Graph- and structure-based learning over code. Static program artifacts admit graph formalisms – AST for syntax, CFG for control flow, PDG for data/control dependence [9]. Learning over these graphs progressed from path-based encoders to GNNs with typed edges. AST-only methods capture local idioms but often miss long-range dependencies; CFG models reason about branching/depth but are costly to construct precisely; PDG models add semantic dependence yet raise scalability concerns. Code Property Graphs (CPG) unifies AST/CFG/PDG, enabling multi-relation message passing and cross-view reasoning [10]. Parallel work in clone detection showed that structural regularities are learnable, yet smells are not clones: they are contextual and often semantic, so structure must be complemented by text and history. Overall, structural learning improves generalization beyond rules but remains single-view when used alone [11].

Representation learning for code: tokens, identifiers, and LLMs. Neural models recover lexical and idiomatic signals from code. Token/identifier embeddings and pretrained code LMs (e.g., CodeBERT/CodeT5 families; hybrid

text+path encoders) have advanced tasks from code search to defect prediction [12]. Strengths include sensitivity to naming conventions, API idioms, and commentary, which are often key cues for smells. Weaknesses mirror the inverse of graph methods: text-only encoders may overfit surface forms, lack precise control/data-flow grounding, and struggle to tell "complex but warranted" from "complex and smelly." Hybrid text+structure models outperform single-modality baselines, but most fuse two views (e.g., AST+text) at a single granularity (method/file), leaving project-level context and evolution underused [13].

Evolution-aware analytics and just-in-time quality prediction. A complementary line models change processes: churn, change frequency, recency, developer experience, co-change networks, and diff semantics. Robust findings show that hotspots and socio-technical signals forecast future quality issues across ecosystems [14]. For smells, evolution helps detect emergence (accretion of responsibilities) and instability (frequent edits). However, most detectors either ignore history or use it post-hoc for ranking rather than as a first-class input. Integrating evolution *within* the representation – rather than "after the fact" – promises earlier, more reliable detection, especially for composite/architectural smells whose symptoms crystallize over releases [15].

Open-set recognition, OOD detection, and uncertainty in SE. Software datasets are non-stationary; frameworks and idioms evolve, and novel smells appear. Closed-set classifiers force every instance into known classes, yielding overconfident errors on unfamiliar inputs. The broader ML literature offers open-set/OOD tools (energy scores, margin losses, selective classification) and uncertainty estimation (ensembles, MC-dropout, evidential heads, post-hoc calibration) [16]. In SE, adoption is early: a handful of works calibrate defect predictors or test across repositories, but open-set protocols for smell detection remain under-specified. Without explicit abstention, detectors either inflate false positives or silently mislabel novelty. This gap motivates uncertainty-aware architectures and evaluation that measures TNR@TPR on unknowns, not only classical precision/recall [17].

Multilingual and cross-project transfer. Smell manifestations depend on language idioms, standard libraries, and ecosystem conventions. Transfer from Java to Kotlin/Scala is non-trivial (null-safety idioms, extension functions, lambdas, typical architectural styles). Thresholds and rule parameters rarely transfer as-is; domain shifts arise from framework usage and team practices. Mitigations include domain-adversarial training, meta-learning, project-aware normalization, and rigorous split protocols (by repository/time) to avoid leakage. Yet many evaluations remain within-project or within-language, leaving the true robustness of detectors underexplored [18].

Synthesis and positioning. The strands above reveal complementary strengths and blind spots [19]:

– rules/metrics: interpretable and fast, but brittle and weak on non-local/composite smells;

– graphs/GNNs: capture dependencies and program structure, but miss lexical nuance and history when used alone;

– text/LLMs: capture naming and idioms, but lack explicit flow/dependence; can over-rely on surface cues;

– evolution signals: expose emergence or instability, yet are seldom integrated end-to-end;

– open-set/uncertainty: essential for non-stationary codebases, but rarely operationalized for smells;

– transfer: necessary in polyglot repos but undermined by language/project drift without normalization and hierarchy.

These observations motivate our Hybrid Code Graph and multi-level encoder with open-set handling. Concretely, we (i) unify AST+CFG+PDG with textual-semantic embeddings, quality metrics, and evolution features in a single heterogeneous graph; (ii) reason hierarchically (local → component → project) so that micro-idioms, meso-structure, and macro-context can each influence a decision; (iii) incorporate uncertainty heads and energy-based abstention to handle novelty safely; and (iv) adopt cross-language normalization and project/time splits to target real-world transfer. This positioning directly addresses the principal methodological gaps: lack of a full-stack representation (*structure* + *text* + *metrics* + *evolution*), absence of multi-granular reasoning, and missing open-set evaluation for anti-pattern detection [20].

The purpose of this study is to design, formalize, and empirically validate a multi-level, hybrid-graph model for detecting code smells and anti-patterns in software components under realistic development conditions (polyglot codebases, evolving repositories, and partially labeled data). Concretely, we aim to (i) demonstrate that an integral representation – combining *structure* (AST + CFG + PDG), *semantics* (LLM-based embeddings), *quality metrics* (CK, Halstead, simple structural indicators), and *evolutionary signals* (diff metrics, churn, co-editing, age/stability) – improves detection quality without sacrificing precision; (ii) show that multi-level reasoning (local → component

ISSN 2617-4316 (Print)
ISSN 2663-7723 (Online)

→ project) enables better disambiguation of context-dependent smells; and (iii) establish that uncertainty-aware, open-set treatment is necessary and effective for safely handling *novel or atypical* manifestations in the wild.

2. PROPOSED MODEL

We detect code smells and anti-patterns by fusing four evidence channels – program structure, textual/semantic cues, quality metrics, and evolution/history – within a single Hybrid Code Graph and a multi-level encoder (local → component → project). The classifier is paired with uncertainty and an open-set gate so that unfamiliar phenomena are flagged rather than mislabelled. Below, the narrative and equations are integrated: each formula is introduced by intuition, then tied to its role in the pipeline.

2.1. Hybrid Code Graph (CPG) and Channelized Features

Smells rarely live in one "view" of code. We therefore cast a project snapshot at time ttt as a typed multigraph that unifies syntax (AST), control (CFG), and dependence (PDG):

$$G_T(V,E,R), \qquad E \subseteq V \times V \times R, \\ R = \{AST, CFG, DATA, CTRL, \dots\}. \tag{1}$$

Local units (methods/blocks) and components (classes/modules) are induced subgraphs $G_t[l]$ and $G_t[c]$. To let structure, text, metrics, and time speak with equal voice, each node v carries a four-part feature tuple:

$$x_v = \left[x_v^{struct} \middle|\middle| x_v^{text} \middle|\middle| x_v^{metric} \middle|\middle| x_v^{evo}\right], \tag{2}$$

where x_v^{struct} encodes type/role, degrees, loop depth, dominance/post-dominance flags; x_v^{text} aggregates LLM embeddings of identifiers/comments aligned to v; x_v^{metric} injects CK/Halstead, LOC, nesting; x_v^{evo} captures churn, change frequency, recency, co-edit centrality, and diff semantics [21].

Because repositories/languages differ, we stabilize each channel via robust, per-project scaling:

$$\tilde{x}_v^{(c)} = clip_p\left(\frac{x_v^{(c)} - median\left(x^{(c)}\right)}{MAD(x^{(c)}) + \epsilon}\right), \tag{3}$$

$$c \in \{struct, text, metric, evo\},$$

then map channels into a shared latent space $\mathbb{R}^d$ with small MLPs:

$$\hat{x}_v^{(c)} = MLP^{(c)}\left(\tilde{x}_v^{(c)}\right) \in \mathbb{R}^d. \tag{4}$$

A non-negative gated mixture produces the initial node state; gates are lightly sparsified so decisions lean on a few channels (interpretable):

$$h_v^{(0)} = LayerNorm\left(\sum_c w_c \hat{x}_v^{(c)}\right), w_c \geq 0, \\ \sum_c w_c = 1, \; L_{gate} = \sum_c \|w_c\|_1. \tag{5}$$

Evolution features privilege recent changes via exponential decay; in practice we concatenate multiple decays (short/mid/long horizons):

$$w(\Delta t) = \exp\left(-\frac{\Delta t}{\tau}\right), \; x_v^{evo} = \sum_i w(\Delta t_i) f_i. \tag{6}$$

2.2. Local Encoder: Marrying Tokens and Relations

Smells have micro-signatures (e.g., guard clauses, deep nesting, tell-don't-ask chains). We therefore summarize each method/block l with two streams and then fuse them per node.

(a) Sequence stream. A light Transformer (or bi-GRU) yields contextual token states $s_1, \dots, s_T$. Span-to-node attention returns lexical semantics to the graph nodes:

$$(s_1, \dots, s_T) = Transformer\big(tok(l)\big), \tilde{h}_v^{seq} = \\ = \sum_{t \in span(v)} \alpha_{vt} s_t, \alpha_{vt} = \\ = \frac{\exp(q_v^T k_t)}{\sum_{t'} \exp q_v^T k_{t'}}. \tag{7}$$

(b) Heterogeneous GNN stream. Relation-aware message passing respects the different roles of AST/CFG/PDG:

$$m_v^{(k)} = \sum_{(u,v,r) \in N(v)} \alpha_{uvr}^{(k)} W_r^{(k)} h_u^{(k)}, \; \alpha_{uvr}^{(k)} = \\ = \frac{\exp\left(\phi_r^{(k)}\left(h_u^{(k)}, h_v^{(k)}\right)\right)}{\sum_{(u',v',r') \in N_r(v)} \exp\left(\phi_r^{(k)}\left(h_{u'}^{(k)}, h_v^{(k)}\right)\right)}, \tag{8}$$

$$h_v^{(k+1)} = FFN\left(h_v^{(k)} + m_v^{(k)}\right). \tag{9}$$

Here, the per-relation softmax normalizes within relation r; only after relation-specific transforms $W_r^{(k)}$ are messages combined, preventing AST from overwhelming PDG (or vice versa).

(c) Gated fusion and pooling. For nodes rich in lexical cues (e.g., well-named APIs), the gate favors the sequence stream; otherwise, structure dominates. We obtain a local summary z_l with evidence-aware pooling:

$$\gamma_v = \sigma\left(w_\gamma^T\left[\tilde{h}_v^{seq} || h_v^{(K_l)}\right]\right),$$
$$h_v^{loc} = \gamma_v \tilde{h}_v^{seq} + (1-\gamma_v) h_v^{(K_l)}, \quad (10)$$
$$z_l = AttnPool\left(\{h_v^{loc}\}_{v\in G_t[l]}\right).$$

23. Component Encoder and Project Context

Composite smells (e.g., God Class, Shotgun Surgery) require meso-scale structure and project context.

(a) Component level. On $G_t[c]$ we run K_c heterogeneous layers, augmenting with virtual edges for cohesion/coupling (LCOM, fan-in/out). In parallel, the SetTransformer summarizes local units $\{z_l\}$ without assuming a fixed method count. A gated join reconciles both views:

$$z_c^{graph} = Readout\left(HetGNN^{K_c}(G_t[c])\right),$$
$$\hat{z}_c = GatetFuse\left(z_c^{graph}, SetTrans(\{z_l\}_{l\subset c})\right). \quad (11)$$

(b) Project level. A component-interaction graph H_t encodes calls/imports/co-change/ownership with evolutionary edge attributes and provides contextualization by light message passing:

$$u_c = ProjGNN^{K_v}(H_t; \{\hat{z}_{c'}\}_{c'\in C}). \quad (12)$$

(c) Hierarchical consolidation. Rather than concatenate, we use hierarchical attention to aggregate project, components, and selected local cues into the final component embedding:

$$g_c = HierAttn(u_c, \hat{z}_c, \{z_l\}_{l\subset c}). \quad (13)$$

The attention weights across these three inputs, together with relation-wise attentions α_{uvr}, form the core of our explanations in IDE/CI.

2.4. Decision Layer, Open-Set Gate, and Learning Objectives

Given g_c, we support multi-label detection by default (components may exhibit multiple smells), with multi-class as a configurable alternative:

$$o_c = W_{cls} g_c + b, p_c = \sigma(o_c)(multi-label) \; or$$
$$p_c = softmax(o_c)(multi-class). \quad (14)$$

To resist overconfident errors on novel patterns, we compute an energy score (lower is *in-distribution*):

$$E_c = -\log \sum_{k=1}^{K} \exp\left(\frac{o_{c.k}}{\tau}\right), \quad (15)$$

and complement it with entropy and epistemic variance (MC-dropout or a small ensemble). We abstain if any uncertainty indicator is high:

$$E_c > \theta_E \; or \; H(p_c) > \theta_H \; or \; Var(p_c) > \theta_V. \quad (16)$$

Training minimizes a composite loss that balances accuracy, separation of unknowns, calibration, and interpretability:

$$L = \lambda_1 L_{cls} + \lambda_2 L_{energy} + \lambda_3 L_{supcon} + \lambda_4 L_{cal} + \lambda_5 L_{gat} \quad (17)$$

with an energy margin between knowns and synthesized outliers,

$$L_n = \mathbb{E}_{c\in kn}[\max(0, E_c - m_{in})] + \mathbb{E}_{\hat{c}\in or}[\max(0, m_{out} - E_{\hat{c}})], \quad (18)$$
$$m_{out} > m_{in},$$

a supervised contrastive term that sharpens class geometry,

$$L_{supcon} = \sum_i \frac{-1}{|P(i)|} \sum_{p\in P(i)} \log \frac{\exp\left(\frac{sim(g_i, g_p)}{\eta}\right)}{\sum_{a\in A(i)} \exp\left(\frac{sim(g_i, g_a)}{\eta}\right)}, \quad (19)$$

and a differentiable ECE surrogate to improve probability usefulness:

$$L_{cal} \approx \sum_{b=1}^{B} \frac{|S_b|}{N} |acc(S_b) - conf(S_b)|. \quad (20)$$

Outliers for open-set training. We simulate "unknowns" by (i) time-shifted components from unrelated repos, (ii) cross-channel perturbations that break structure–text consistency (identifier shuffling; mild PDG edge dropout under semantic safety), and (iii) domains known to be smell-scarce.

Thresholding practice. $(\theta_E, \theta_H, \theta_V)$ are tuned on novelty-aware validation (holding out whole projects or smell classes) to avoid leakage and to match deployment.

2.5. Incremental Inference and Complexity (CI/CD-oriented)

To meet PR latency budgets, we maintain G_t incrementally: on a diff Δ, rebuild only touched AST/CFG/PDG slices; recompute local encodings for changed methods; refresh summaries for impacted components and their 1–2-hop neighbors in H_t; and roll evolution features forward using (6). If n, m are nodes/edges in the affected region, a K-layer heterogeneous GNN costs $O(K(m+n)d)$; the local encoder scales with changed tokens T (Transformer $O(T^2)$, GRU $O(T)$). Mixed precision and caching keep memory/time within typical CI/CD budgets [22].

2.6. Interpretability and Practitioner Feedback

The model surfaces three first-class explanations per decision: (i) hierarchical attention

ISSN 2617-4316 (Print)
ISSN 2663-7723 (Online)

weights in (13), clarifying whether evidence was local, component-level, or project-contextual; (ii) relation-wise attentions α_{uvr} in (8), showing whether PDG (def–use), CFG (nesting), or AST cues dominated; (iii) channel gates w_c in (5), quantifying reliance on structure/text/metrics/evolution. IDE/CI plugins render these as an evidence card (top-k salient nodes/edges; attention bars; gate vector). For abstentions, we additionally report E_c, $H(p_c)$, and ensemble variance with guidance ("novel pattern likely – review recommended") [24].

3. EXPERIMENTAL SETUP AND RESULTS

This section describes the evaluation design and reports the main findings for the proposed hybrid, multi-level, open-set detector. We begin with the corpora, labels, and baselines, then detail the experimental protocol and metrics, and finally present results on closed-set detection, open-set robustness, cross-language transfer, calibration, efficiency, and qualitative analysis. Where appropriate, we refer to the summary tables introduced in the previous subsection.

Corpora, taxonomy, and labeling. We assembled a polyglot JVM benchmark comprising mature Java projects and mid-sized Kotlin/Scala repositories with multi-year histories [25]. The basic statistics – repository counts, quarterly snapshot counts, component/method totals – are reported in Table 1.

Instances are components (classes/modules), which aligns with our detection granularity. The resulting label distribution follows typical industry skew: a dominant *No-smell* population accompanied by several rarer smell classes. Per-language prevalence is summarized in Table 2, which also includes the (optional) label cardinality (average labels per instance) for the multi-label setting [23].

Ground truth combines (i) consensus outputs from established rule-based detectors with project-normalized thresholds, (ii) heuristic templates for Shotgun-Surgery–like phenomena that leverage co-change signals, and (iii) manual audits over a stratified subset to calibrate precision and estimate noise. This hybrid labeling mirrors how practitioners bootstrap datasets in the absence of exhaustive human annotation and is one reason we evaluate not only accuracy but also calibration and abstention behavior.

Baselines, model variants, and implementation. Comparisons span the method space: a tuned rules/metrics system; a strong AST-GNN using relation-aware attention on AST only; a text-only code LM fine-tuned at method/file granularity; a method-level fusion of AST + text; a metrics-only learner; and an evolution-ranker that orders rule outputs by churn/recency [26]. To probe our design, we include ablations that remove channels (−Text, −Metrics, −Evolution), restrict relations (AST-only), collapse the hierarchy (Local-only), or disable abstention (No-abstain). Hyperparameters and depths follow Section 4 (local/component/project encoders at 2–3/2–3/1–2 layers; hidden size 256). Uncertainty combines energy, entropy, and MC-dropout variance with thresholds tuned on novelty-aware validation.

Protocols and metrics. To prevent leakage and emulate deployment, we run cross-project splits (disjoint repositories) and temporal splits (train < validation < test chronologically). For open-set evaluation, we withhold one or two smell classes entirely during training and reintroduce them only at test time. We report AUPRC and macro-F1 for closed-set detection, FPR@95TPR to assess the cost of high recall, AUROC-OSR and TNR@TPR for recognition of unknowns, and ECE for

Table 1. **Dataset summary (per language)**

Language	Repos	Snapshots (quarterly)	Components	Methods	Labeled Components
Java	28	24	145,200	1,120,000	145,200
Kotlin	12	10	58,400	420,300	58,400
Scala	9	9	47,100	356,800	47,100
Total	49	–	250,700	1,897,100	250,700

Source: **compiled by the author**

Table 2. **Label prevalence (multi-label; % of components)**

Language	LC (avg labels/inst)	No-smell	LM	GC	FE	DC	SS-like
Java	–	74.5	8.2	5.0	3.9	5.6	2.7
Kotlin	–	77.2	7.1	4.1	3.5	4.8	2.1
Scala	–	78.6	6.6	3.8	3.1	4.2	1.9
Overall	–	–	–	–	–	–	–

Source: **compiled by the author**

probability calibration. Wall-clock latency and memory summarize CI/CD feasibility. The metric purposes and rationale are recapped in Table 3's caption and in the metric synopsis previously provided [27].

Closed-set performance. Across the cross-project, time-aware test splits, the proposed model improves macro AUPRC and F1 without inflating FPR@95TPR relative to strong single-view baselines. The aggregate numbers appear in Table 3. Two tendencies are worth highlighting. First, Long Method benefits from the combination of CFG depth and lexical cues: the model avoids penalizing intentionally complex but well-structured code by reconciling structural and textual evidence. Second, God Class and Shotgun-Surgery–like patterns profit from the hierarchical design: component-level cohesion/coupling edges and project-level interaction context reduce both misses and spurious hits. Per-class AUPRC confirms these trends: our model outperforms the best non-ours baseline for all smells, with the largest margins on God Class and SS-like (see Table 4).

Ablation results embedded in Table 3 indicate which ingredients matter. Removing Evolution causes the steepest drop on SS-like (as expected) and a noticeable decline on God Class (instability context). Collapsing to Local-only particularly harms GC/SS, underscoring the need for meso/macro reasoning. An AST-only variant struggles on Feature Envy and Data Class, where lexical/semantic signals are decisive.

Open-set robustness and abstention. Detectors deployed in living codebases inevitably face novel smells and idioms. Our open-set protocol – withheld smell classes during training – tests whether the model can *separate known from unknown* and abstain when appropriate. The energy scores (Eq. 15), complemented by entropy and variance, and forms the tri-criterion gate (Eq. 16). As Table 5 shows, this composite improves AUROC-

Table 3. **Closed-set macro performance (cross-project, time-aware test)**

Model	AUPRC ↑	F1 ↑	FPR@95TPR % ↓	ECE % ↓
Rules / Metrics	0.48	0.41	22.1	8.4
AST-GNN (AST only)	0.53	0.45	21.9	7.6
Text-only LM	0.55	0.46	23.8	7.9
AST+Text (method-level)	0.56	0.47	22.5	7.1
Metrics-only	0.49	0.42	25.4	8.7
Evolution-ranker	0.50	0.43	24.2	8.3
Ours (Hybrid, Multi)	**0.62**	**0.52**	**21.0**	**5.1**
Ours −Evolution	0.58	0.49	21.4	5.6
Ours Local-only	0.57	0.48	21.6	5.5
Ours No-abstain	0.62	0.52	21.0	7.4

Source: **compiled by the author**

Table 4. **Per-class AUPRC (best baseline vs ours)**

Smell	Best Baseline	Best Baseline AUPRC	Ours AUPRC	Δ (Ours − Base)
Long Method (LM)	AST+Text	0.58	0.66	+0.08
God Class (GC)	Text-only LM	0.52	0.64	+0.12
Feature Envy (FE)	AST-GNN	0.54	0.62	+0.08
Data Class (DC)	Text-only LM	0.57	0.63	+0.06
SS-like (SS)	Evolution-ranker	0.49	0.63	+0.14

Source: **compiled by the author**

Table 5. **Open-set recognition (unknown classes withheld)**

Unknown class	Method	AUROC-OSR ↑	TNR@TPR=0.90 ↑	ECE % ↓
FE	Energy-only	0.79	0.62	6.7
FE	Energy+Entropy+Variance	0.87	0.77	5.3
DC	Energy-only	0.78	0.60	6.9
DC	Energy+Entropy+Variance	0.86	0.74	5.5
SS	Energy-only	0.76	0.58	7.1
SS	Energy+Entropy+Variance	0.85	0.72	5.6

Source: **compiled by the author**

ISSN 2617-4316 (Print)
ISSN 2663-7723 (Online)

OSR and TNR@TPR=0.90 across all withheld classes compared to energy-only gating, while simultaneously lowering ECE. The No-abstain variant (reported in Table 3) attains similar closed-set scores but more than doubles false positives on unknowns, illustrating the operational risk of forced classification.

Cross-language and cross-project transfer. A practical concern is how well detectors trained on Java generalize to Kotlin and Scala. With language-agnostic CPG typing and robust per-project normalization, the proposed model retains a large fraction of its Java performance on both languages, outpacing single-view baselines (see Table 6 for retention percentages). Cross-project transfer shows a similar advantage: macro AUPRC declines modestly for our model but significantly for rule-based, AST-only, and text-only systems, suggesting that hierarchical context and evolution cues buffer against project idiosyncrasies.

Table 6. **Cross-language transfer (train Java → test others)**

Model	Kotlin retention % ↑	Scala retention % ↑
Rules / Metrics	69	61
AST-GNN (AST only)	72	64
Text-only LM	74	67
AST+Text	77	70
Ours (Hybrid, Multi)	**84**	**79**

Source: **compiled by the author**

Calibration and decision usefulness

Because teams act on probabilities, not only rankings, we evaluate calibration. The calibration loss (Eq. 20) and temperature scaling reduce ECE materially (see Table 3), which in turn makes thresholding predictable: at a nominal confidence of 0.8–0.9, empirical accuracies track closely. This reliability is critical when pairing the detector with abstention: high-confidence predictions can be auto-labeled or auto-suggested, while low-confidence or high-energy cases are escalated.

Efficiency and CI/CD readiness

Finally, we study incremental inference under realistic diffs. The pipeline rebuilds only the touched AST/CFG/PDG slices, recomputes local encoders for changed methods, and updates component summaries for affected nodes and their one- to two-hop neighbors. Latency and memory on small, medium, and large pull requests are summarized in Table 7. These measurements satisfy typical CI budgets: small PRs complete in well under a minute on modest hardware; medium PRs in a few minutes; larger diffs remain tractable via GRU fallback and relation pruning. Importantly, ablating project-level propagation from 2 to 1 layer yields negligible accuracy loss but saves ~12% time (numbers reflected within Table 7's ranges), indicating a favorable accuracy–latency trade-off.

Qualitative analysis and threats to validity

Qualitative inspections align with the quantitative picture. For Feature Envy, PDG def–use edges concentrate attention on manipulations of foreign state; for SS-like, project-level attention dominates, reflecting dispersed recent co-changes rather than any single local idiom. The model abstains on unfamiliar coroutine patterns in Kotlin – high energy and entropy – allowing reviewers to triage safely and convert such cases into future training signal.

Principal threats include label noise from consensus detectors (partly mitigated by audits and robustness checks), external validity limits from an OSS-heavy JVM corpus (partly countered by cross-language tests), temporal distribution shift (addressed via multi-scale decay and time-aware splits), and CPG fidelity for PDG-heavy smells (assessed via perturbation studies; not shown for space). Within these constraints, the evidence across Table 3–7 supports three conclusions: (i) a full-stack, hierarchical representation consistently improves detection without sacrificing precision; (ii) open-set abstention materially increases safety under novelty; and (iii) the approach is operationally fit for CI/CD through incremental updates and predictable latency.

4. DISCUSSION

Our results substantiate three claims: (i) detectors for anti-patterns must fuse multiple evidence sources; (ii) multi-level reasoning (local → component → project) resolves context-dependent

Table 7. **CI/CD efficiency**

Diff size	Changed LOC	Local encoder	GNN depths (Kℓ / Kc / Kp)	Latency (mm:ss)	Peak Mem. (GB)
Small PR	≤1,000	2–4-layer Transformer	2 / 2 / 1	00:30–01:00	4–6
Medium PR	3–5,000	4-layer Transformer	3 / 3 / 1	01:00–03:00	6–8
Large PR	8–10,000	GRU fallback	2 / 2 / 1	03:00–06:00	5–7

Source: **compiled by the author**

smells; and (iii) selective prediction with calibrated uncertainty is essential in evolving codebases.

Full-stack evidence matters. Closed-set gains in Table 3 and per-class deltas in Table 4 show that no single channel is sufficient. *Long Method* improves when CFG depth is tempered by lexical cues; *Feature Envy* relies on PDG def–use plus identifiers; *Data Class* is mainly lexicon/metrics; *SS-like* is history-driven. Channel gates (Eq. 5) let the model emphasize the right combination per instance, which explains why AUPRC and F1 rise together instead of trading precision for recall. The "Per-class AUPRC" figure mirrors this: the largest lifts occur exactly where single-view baselines are weakest (GC, SS).

Hierarchy resolves ambiguity. The Local-only ablation in Table 3 degrades most on *God Class* and *SS-like*, confirming that local idioms cannot, by themselves, capture cohesion/coupling or dispersed change. Component-level aggregation (Eq. 11) summarizes design signals; project-level propagation (Eq. 12) injects interaction and co-change context. In qualitative cases, hierarchical attention (Eq. 13) often weights project context highest for SS-like, component structure for GC, and uses local cues as tie-breakers – matching reviewer reasoning.

Uncertainty is a first-class requirement. With withheld classes, the tri-criterion gate (energy + entropy + variance) improves AUROC-OSR and TNR@TPR over energy-only (Table 5), while maintaining closed-set quality (Table 3, "Ours" vs "Ours No-abstain"). Practically, this enables two levers in CI/CD: a probability threshold for auto-actions and an abstention policy for escalation. Because calibration improves (lower ECE in Table 3), these levers are predictable: high scores behave like high empirical precision.

Transferability stems from design, not scale. Cross-language retention in Table 6 suggests three helpful choices: a language-agnostic CPG type lattice, robust per-project normalization (Eq. 3), and project context. Together they buffer syntax and framework drift better than rules, AST-only, or text-only baselines. Cross-project results (reflected in Table 3) show similar resilience.

Operational viability. Incremental maintenance of the Hybrid Code Graph confines work to changed regions; the latency envelope in Table 7 fits common CI budgets. Reducing project-level depth from 2→1 saves time with negligible accuracy loss – an attractive knob for busy pipelines.

Limitations and risks. Labels inherit biases from consensus detectors; despite audits, borderline *Data Class*/*Feature Envy* remain noisy. Conservative PDG harms FE recall in complex flows. Temporal decay (Eq. 6) cannot fully absorb abrupt framework shifts; abstention plus periodic retraining is still required. Explanations (attention/gates) aid triage but are not causal evidence.

Implications. For practice, a sensible operating mode emerges: auto-apply high-confidence, low-uncertainty findings; route abstentions to review; feed adjudications back for continual learning. For research, the promising directions are (a) coupling detection with counterfactual refactoring suggestions, (b) richer temporal models beyond decay (event/causal graphs), and (c) stronger program-semantics signals (learned completion of missing PDG edges) without sacrificing CI-grade efficiency.

In brief, smells are multi-modal, multi-level, and open-world phenomena. When structure, semantics, metrics, and time are fused – and uncertainty is treated as a core interface – detectors become not only more accurate but safer and more useful for sustained, real-world adoption.

CONCLUSIONS AND FUTURE WORK

This work framed code smells as a multi-modal, multi-level, open-world problem. By fusing CPG structure (AST+CFG+PDG), textual/semantic signals, classical metrics, and evolution history – and by reasoning from local idioms to component design and project context – the model improved detection quality without the usual precision–recall trade-off, generalized better across repositories and JVM languages, and, through an energy–entropy–variance gate, knew when to abstain under novelty. Incremental graph maintenance kept latency within CI/CD budgets, making the approach practical beyond the lab. In short, treating *structure, semantics, metrics, and time* as first-class signals, then aggregating them hierarchically, yields detectors that are not only more accurate but safer to deploy.

Limits remain. Labels derived from consensus tools import bias; conservative PDG extraction can mute def–use evidence; exponential decay over change events cannot fully anticipate abrupt framework shifts. These caveats do not undercut the main result but delineate conditions for careful use.

The natural next step is to move from detection to assistance. Because the model localizes evidence

ISSN 2617-4316 (Print)
ISSN 2663-7723 (Online)

across channels and levels, it can be extended to propose minimal, testable refactorings – extracting methods, moving members, or splitting classes – and to validate them against tests, closing the loop inside CI. Richer temporal models that treat change as an event sequence rather than an aggregate could separate benign churn from structural decay earlier, while learned completion of missing semantic edges would strengthen dependency-heavy smells without prohibitive analysis cost. Broader polyglot support (TypeScript, Go, Rust) and lightweight governance – confidence targets tied to abstention rates, stability checks for explanations, and measurement of downstream impact on review time and defects – will turn a capable detector into a dependable partner that not only finds smells but helps teams fix them, at the cadence of modern development.

REFERENCES

1. van Emden, E. & Moonen, L. “Java quality assurance by detecting code smells“. *Proceedings of WCRE*. Richmond, USA. 2002. p. 97–106. DOI: https://doi.org/10.1109/WCRE.2002.1173068.

2. Marinescu, R. “Detection strategies: metrics-based rules for detecting design flaws“. *Proceedings of ICSM*. Chicago, IL, USA. 2004. p. 350–359. DOI: https://doi.org/10.1109/ICSM.2004.1357820.

3. Yamashita, A. & Moonen, L. “Do code smells reflect important maintainability aspects?”. *Proceedings of ICSM*. Trento, Italy. 2012. p. 306–315. DOI: https://doi.org/10.1109/ICSM.2012.6405287.

4. Palomba, F., Bavota, G., Di Penta, M., Oliveto, R. & Poshyvanyk, D. “Detecting bad smells in source code using change history“. *Proceedings of ASE*. 2013. DOI: https://doi.org/10.1109/ASE.2013.6693086.

5. Palomba, F., Zanoni, M., Fontana, F. A., Roveda, R. & Oliveto, R. “On the diffuseness and the impact on maintainability of code smells”. *Empirical Software Engineering*. 2018; 23 (3): 1658–1707. DOI: https://doi.org/10.1007/s10664-017-9535-z.

6. Azeem, M. I., Palomba, F., Shi, L. & Wang, Q. “Machine learning techniques for code smell detection: A systematic literature review and meta-analysis”. *Information and Software Technology*. 2019; 108: 115–138. DOI: https://doi.org/10.1016/j.infsof.2018.12.009.

7. dos Reis, R. Q., Kalinowski, M., Felizardo, K. R., et al. “Code smell detection using machine learning: A systematic literature review”. *Archives of Computational Methods in Engineering*. 2022; 29: 4141–4179. DOI: https://doi.org/10.1007/s11831-021-09566-x.

8. Chidamber, S. R. & Kemerer, C. F. “A metrics suite for object oriented design”. *IEEE Transactions on Software Engineering*. 1994; 20 (6): 476–493. DOI: https://doi.org/10.1109/32.295895.

9. McCabe, T. J. “A complexity measure”. *IEEE Transactions on Software Engineering*. 1976; 2 (4): 308–320. DOI: https://doi.org/10.1109/TSE.1976.233837.

10. Ferrante, J., Ottenstein, K. J. & Warren, J. D. “The program dependence graph and its use in optimization”. *ACM Transactions on Programming Languages and Systems*. 1987; 9 (3): 319–349. DOI: https://doi.org/10.1145/24039.24041.

11. Yamaguchi, F., Golde, N., Arp, D. & Rieck, K. “Modeling and discovering vulnerabilities with code property graphs”. *IEEE Symposium on Security and Privacy*. 2014. DOI: https://doi.org/10.1109/SP.2014.44.

12. Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C. & Philip, S. Y. “A comprehensive survey on graph neural networks”. *IEEE Transactions on Neural Networks and Learning Systems*. 2021; 32 (1): 4–24. DOI: https://doi.org/10.1109/TNNLS.2020.2978386.

13. Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P. & Bengio, Y. “Graph attention networks”. arXiv. 2017. DOI: https://doi.org/10.48550/arXiv.1710.10903.

14. Wang, X., Ji, H., Shi, C., Wang, B., Ye, Y., Cui, P. & Yu, P. S. “Heterogeneous Graph Attention Network”. Proceedings of WWW. 2019. DOI: https://doi.org/10.1145/3308558.3313562.

15. Feng, Z., Guo, D., Tang, D., Duan, N., Gong, M., Shou, L., Qin, B., et al. “CodeBERT: A Pre-Trained model for programming and natural languages”. *Findings of EMNLP*. 2020. DOI: https://doi.org/10.18653/v1/2020.findings-emnlp.139.

16. Wang, Y., Wang, W., Joty, S. R. & Hoi, S. C. H. “CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation”. *Proceedings of EMNLP*. 2021. DOI: https://doi.org/10.18653/v1/2021.emnlp-main.685.

17. Zhou, Y., Liu, S., Siow, J., Du, X. & Liu, Y. “Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks”. *arXiv*. 2019. DOI: https://doi.org/10.48550/arXiv.1909.03496.

18. Palomba, F., Bavota, G., Di Penta, M., Oliveto, R. & Poshyvanyk, D. “On the impact of code smells on the energy consumption of android applications”. *Information and Software Technology*. 2017; 82: 1–17. DOI: https://doi.org/10.1016/j.infsof.2018.08.004.

19. Sjøberg, D. I. K., Yamashita, A., Anda, B. C. D., Mockus, A. & Dybå, T. “Quantifying the effect of code smells on maintenance effort”. *IEEE Transactions on Software Engineering*. 2013; 39 (8): 1144–1156. DOI: https://doi.org/10.1109/TSE.2012.89.

20. Khomh, F., Di Penta, M., Guéhéneuc, Y.-G. & Antoniol, G. “An exploratory study of the impact of antipatterns on class change- and fault-proneness”. *Empirical Software Engineering*. 2012; 17 (3): 243–275. DOI: https://doi.org/10.1007/s10664-011-9171-y.

21. Moser, R., Pedrycz, W. & Succi, G. “A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction”. *Proceedings of ICSE*. 2008. DOI: https://doi.org/10.1145/1368088.1368114.

22. Guo, C., Pleiss, G., Sun, Y. & Weinberger, K. Q. “On Calibration of Modern Neural Networks”. *arXiv*. 2017. DOI: https://doi.org/10.48550/arXiv.1706.04599.

23. Scheirer, W., de Rezende Rocha, A., Sapkota, A. & Boult, T. “Toward Open Set Recognition”. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 2013; 35 (7): 1757–1772. DOI: https://doi.org/10.1109/TPAMI.2012.256.

24. Bendale, A. & Boult, T. “Towards Open Set Deep Networks”. *Proceedings of CVPR*. 2016. DOI: https://doi.org/10.1109/CVPR.2016.173.

25. Liu, W., Wang, X., Owens, J. D. & Li, Y. “Energy-based Out-of-Distribution Detection”. *NeurIPS*. 2020. DOI: https://doi.org/10.48550/arXiv.2010.03759.

26. Yadav, V., Bandi, R. K. & Nallamothu, R. “A Systematic Literature Review of Code Smell Detection: Current Trends and Future Directions”. *Applied Sciences*. 2024; 14 (14): 6149. DOI: https://doi.org/10.3390/app14146149.

27. Yamaguchi, F., Maier, A., Gascon, H. & Rieck, K. “Automatic Inference of Semantic Patches for Vulnerability Analysis”. *Proceedings of IEEE S&P*. 2015. DOI: https://doi.org/10.1109/SP.2015.54.

Conflicts of Interest: The authors declare that they have no conflict of interest regarding this study, including financial, personal, authorship or other, which could influence the research and its results presented in this article

Received 21.08.2025
Received after revision 23.09.2025
Accepted 26.09.2025

DOI: https://doi.org/10.15276/aait.08.2025.18....
УДК 004.8:004.42:519.17

Гібридні графи для запахів коду: багаторівнева модель виявлення антипатернів у програмних компонентах

Курінько Дмитро Дмитрович
ORCID: https://orcid.org/0000-0001-8304-3257; dmitrykurinko@gmail.com
Національний університет «Одеська Політехніка», пр. Шевченка, 1. Одеса, 65044, Україна

АНОТАЦІЯ

Наведено гібридний багаторівневий підхід до виявлення «запахів» коду та антишаблонів у програмних компонентах, у якому структурні, семантичні, метричні та еволюційні ознаки розглядаються як рівноправні сигнали. Будується гетерогенний граф властивостей коду (Abstract Syntax Tree, Control-Flow Graph, Program Dependence Graph), збагачений текстовими вбудовуваннями з попередньо натренованої мовної моделі для коду, класичними метриками якості (Чидамбер – Кемерер, Гальстед) і характеристиками історії контролю версій (churn, co-change, recency). Локальні ідіоми агрегуються послідовнісно-графовим кодувальником на рівні методу/блоку; структурний контекст компонента узагальнюється графовою

нейромережею, чутливою до типів відношень, на рівні класу/модуля; проєктний контекст поширюється по графу взаємодії компонентів. Для експлуатації в еволюційних кодових базах інтегровано «open-set»-голову: поєднання енергії, ентропії та стохастичної дисперсії забезпечує відкалібровану відмову від передбачення на незнайомих патернах. Оцінювання виконано на багатомовних корпусах Java Virtual Machine із часово обізнаними, крос-проєктними розбиттями та мульти мітковими цілями (Long Method, God Class, Feature Envy, Data Class, Shotgun-Surgery, No-smell). Зафіксовано зростання macro-AUPRC і F1 порівняно з rule/metric-базовими моделями, моделями лише на базі Abstract Syntax Tree та моделях лише за текстом, та в той же час зафіксовано збереження або зниження FPR@95TPR. Експерименти з прихованими класами показують, що «open-set»-гейтинг підвищує AUROC для розпізнавання у відкритій множині та TNR@TPR, а також зменшує помилку калібрування, що робить імовірності придатними для порогової автоматизації й людського тріажу. Перенесення між мовами (навчання на Java → тестування на Kotlin/Scala) є стійкішим, ніж у одновидових моделей, завдяки мовно-агностичній типізації та покомандному нормуванню. Інкрементальне оновлення графа обмежує обчислення зміненими ділянками, узгоджуючи час інференсу з бюджетами CI/CD. Надані механізми пояснюваності (ієрархічна увага та «канальні» клапани) демонструють узгодженість із міркуваннями практиків. Зроблено висновок, що гібридні графи з ієрархічним виводом і селективним передбаченням формують детектори, які є точнішими, краще переносними та операційно безпечнішими для еволюційних програмних систем.

Ключові слова: машинне навчання; програмна інженерія; аналіз програм; графове навчання подань; статичний аналіз; оцінювання невизначеності; трансферне навчання; емпіричне оцінювання

ABOUT THE AUTHOR

Dmytro D. Kurinko - PhD Student, Artificial Intelligence and Data Analysis Department. Odesa Polytechnic National University, 1, Shevchenko Ave. Odesa, 65044, Ukraine
ORCID: https://orcid.org/0000-0001-8304-3257; dmitrykurinko@gmail.com
***Research field*:** Machine learning and artificial intelligence, machine learning for software engineering, pattern recognition, computer vision, knowledge representation in software systems

Курінько Дмитро Дмитрович - аспірант кафедри Штучного інтелекту та аналізу даних. Національний університет «Одеська Політехніка», пр. Шевченка, 1. Одеса, 65044, Україна