https://doi.org/10.31891/2307-5732-2026-365-78
УДК 004.4'242

POLISHCHUK VOLODYMYR
Kyiv National University of Technologies and Design
https://orcid.org/0009-0000-2161-4560
e-mail: polishchuk.vl@knutd.edu.ua
SKIDAN VLADYSLAVA
Kyiv National University of Technologies and Design
https://orcid.org/0000-0002-8358-9759
e-mail: skidan.vv@knutd.edu.ua

A METHOD FOR AUTOMATED CLASSIFICATION OF UML DIAGRAM TYPES BASED ON ANALYSIS OF PLANTUML INTERMEDIATE REPRESENTATIONS

PlantUML is one of the most widely used UML formats in open-source software repositories, yet all nine standard UML diagram types share a single @startuml entry point with no explicit type declaration. Existing automated classification approaches, image-based neural networks and LLM-based methods, require rendered images, training data, or external inference services, limiting their applicability for deterministic batch processing of large textual corpora. This paper raises a question: to what extent does PlantUML's syntax encode the diagram type? A heuristic classifier is proposed that operates on PlantUML's compiled intermediate representation using a three-tier dispatch strategy combining diagram class inspection, entity-level type analysis, and symbol-based majority counting. The classifier operates directly on the compiler's output, requiring no training data, rendered images, or external APIs. Evaluation on 143,258 diagrams from the UML-in-the-Wild dataset, a corpus of PlantUML files mined from open-source repositories on GitHub and Bitbucket, yielded 95.96% overall semantic-syntactic agreement between the proposed syntactic classifier and published LLM-based semantic labels, with 93–99.7% agreement for seven of nine types. Manual inspection of 110 disagreement files revealed that neither classifier is uniformly superior: 47% of disagreements are compiler-correct, 45% are LLM-correct, and 8% are ambiguous. The component/deployment boundary, where agreement drops to 48–62%, is attributed to two structural mechanisms: keyword semantic overloading (deployment-associated keywords such as database and node used for non-infrastructure concepts) and container-vs-leaf asymmetry (node containers invisible to entity-level inspection). These represent limitations of PlantUML's expressiveness rather than classifier deficiencies.

***Keywords**: UML diagrams; PlantUML; diagram type classification; intermediate representation; heuristic classification; semantic-syntactic agreement.*

ПОЛІЩУК ВОЛОДИМИР, СКІДАН ВЛАДИСЛАВА
Київський національний університет технологій та дизайну

МЕТОД АВТОМАТИЗОВАНОЇ КЛАСИФІКАЦІЇ ТИПІВ UML-ДІАГРАМ НА ОСНОВІ АНАЛІЗУ ПРОМІЖНОГО ПРЕДСТАВЛЕННЯ PLANTUML

PlantUML є одним із найпоширеніших форматів UML у репозиторіях програмного забезпечення з відкритим кодом, проте всі дев'ять стандартних типів UML-діаграм використовують єдину директиву @startuml без явного оголошення типу діаграми. Існуючі підходи до автоматизованої класифікації — нейронні мережі на основі зображень та методи на основі великих мовних моделей (LLM) — потребують згенерованих зображень, навчальних даних або зовнішніх сервісів інференсу, що обмежує їх застосування для детермінованої пакетної обробки великих текстових корпусів. Ця стаття порушує питання: якою мірою синтаксис PlantUML кодує тип діаграми? Запропоновано евристичний класифікатор, що працює на основі скомпільованого проміжного представлення PlantUML із використанням трирівневої стратегії диспетчеризації, яка поєднує аналіз класу діаграми, аналіз типів сутностей на рівні елементів та підрахунок за більшістю на основі візуальних символів. Класифікатор працює безпосередньо з результатами компілятора та не потребує навчальних даних, згенерованих зображень або зовнішніх API. Оцінювання проведено на 143 258 діаграмах із набору даних UML-in-the-Wild — корпусу PlantUML-файлів, видобутих із в відкритого коду на платформах GitHub та Bitbucket. Результати показали 95,96% загальної семантико-синтаксичної узгодженості між запропонованим синтаксичним класифікатором та опублікованою семантичною розміткою на основі LLM, із узгодженістю 93–99,7% для семи з дев'яти типів. Ручна перевірка 110 файлів з розбіжностями виявила, що жоден класифікатор не є однозначно кращим: у 47% випадків розбіжностей правильним виявився запропонований класифікатор, у 45% — LLM-класифікатор, а 8% є неоднозначними. Межа між діаграмами компонентів та діаграмами розгортання, де узгодженість знижується до 48–62%, зумовлена двома структурними механізмами: семантичним перевантаженням ключових слів (database, node та інші використовуються для позначення неінфраструктурних елементів) та асиметрією контейнер–елемент (контейнери node невидимі для аналізу на рівні сутностей). Ці обмеження є властивостями мови PlantUML, а не недоліками класифікатора.

***Ключові слова**: UML-діаграми, PlantUML, класифікація типів діаграм, проміжна репрезентація, евристична класифікація, семантико-синтаксичне узгодження*

Стаття надійшла до редакції / Received 17.03.2026
Прийнята до друку / Accepted 11.04.2026
Опубліковано / Published 28.05.2026

Statement of the Problem

The Unified Modeling Language (UML) remains the primary standard for representing software system structure and behavior in both academic and industrial settings. Among the various tools for authoring UML diagrams, PlantUML, a text-based domain-specific language that generates diagrams from structured markup, has emerged as the most prevalent UML format in open-source software repositories [1]. The growing availability of large-scale PlantUML corpora, containing tens and hundreds of thousands of diagram files, creates a demand for automated analysis tools capable of

operating on these collections. Empirical software engineering studies, educational platforms, diagram search and retrieval systems, and quality assessment tools all require, as a prerequisite, reliable identification of the UML diagram type for each file in the corpus. At such scale, manual classification is infeasible, while existing automated approaches, image-based classifiers relying on convolutional or transformer neural networks, as well as large language model (LLM) based methods, require either rendered diagram images, labeled training datasets, or access to external inference services, making them computationally expensive, non-deterministic, and difficult to reproduce.

A fundamental characteristic of the PlantUML language is the absence of an explicit diagram type declaration. All nine standard UML diagram types, class, object, sequence, activity, state, component, deployment, use case, and timing, share a single `@startuml` entry point. The diagram type is determined internally by the PlantUML compiler through an ordered chain of syntax analyzers, where the first parser that successfully matches the input claims the diagram. This internal type resolution is not exposed to the user or to external tooling, and the author's communicative intent, whether a given diagram is meant to represent software architecture, infrastructure topology, or behavioral interaction, is never formally captured in the source file. This design raises a question of both practical and theoretical significance: to what extent does PlantUML's syntax encode the diagram type? For which of the nine UML types is the syntactic structure of the source file sufficient to unambiguously recover the author's intent, and for which types is the syntax semantically overloaded, that is, reused across diagram types in ways that make purely structural classification unreliable?

Answering this question requires two components: first, a classifier that operates exclusively on the syntactic evidence available in the compiler's parsed intermediate representation, without recourse to rendered images, domain context, or language model inference; and second, an evaluation framework that compares syntactic classification against semantic classification without treating either as infallible ground truth. Thus, the development of a deterministic, parser-based method for UML diagram type classification from PlantUML source, together with a principled framework for assessing the limits of such classification, remains an open problem in the fields of software engineering tooling and empirical analysis of software modeling practices.

Analysis of Recent Research and Publications

Research on UML usage in open-source software has intensified over the past decade, driven by the increasing availability of public repositories containing modeling artifacts. Hebig et al. [2] conducted the first large-scale mining of UML models from GitHub, establishing the Lindholmen dataset — a curated collection of UML files extracted from over 93,000 repositories. This dataset was subsequently extended by Robles et al. [3], who documented the diversity of UML formats and the challenges of cross-format analysis. A complementary perspective was provided by Reggio et al. [4], whose survey of UML practitioners established an empirical baseline for diagram type popularity: class and sequence diagrams dominate both academic and industrial usage, while deployment and timing diagrams remain comparatively rare. More recently, Romeo et al. [1] conducted a comprehensive study of UML's presence in open-source software, confirming that PlantUML has emerged as the single most prevalent UML format in public repositories, a finding that motivates the development of analysis tools specifically designed for this textual notation. Collectively, these corpus-level studies demonstrate both the scale of available PlantUML data and the need for automated processing tools capable of operating on collections of tens to hundreds of thousands of diagram files.

The problem of automated UML diagram type classification has been approached primarily through image-based methods. Shcherban et al. [5] applied convolutional neural networks (CNN) to rendered UML diagram images, achieving multiclass classification across 10 UML diagram types, the first work to demonstrate that visual features of rendered diagrams carry sufficient information for type discrimination. Torcal et al. [6] advanced this line of research by constructing a curated ground truth dataset of 2,626 labeled UML diagram images and applying Vision Transformer (ViT) architectures, reporting improved classification accuracy over CNN baselines. Both approaches, however, share fundamental limitations: they require rendered diagram images as input, depend on labeled training datasets for supervised learning, and involve computationally expensive GPU inference, making them unsuitable for direct application to textual PlantUML corpora where diagrams exist as source code, not images.

An alternative paradigm has emerged with the application of large language models (LLMs) to UML-related tasks. Bates et al. [7] demonstrated the use of multimodal LLMs for generating UML code from diagram images, showing that modern language models can interpret visual UML notation and produce structured textual output. Conrardy and Cabot [8] investigated image-based UML diagram generation using LLMs, reporting first results on converting diagram images into structured representations. While LLM-based approaches offer semantic understanding, the ability to interpret a diagram's communicative intent beyond its syntactic structure, they are inherently non-deterministic, require access to external inference services, and produce results that vary across runs, limiting their suitability for reproducible large-scale corpus analysis.

The analysis of existing approaches reveals a gap: all current methods for UML diagram type classification operate either on rendered images (requiring training data and GPU inference) or through LLM-based semantic interpretation (requiring external API access and producing non-reproducible results). No prior work exploits the compiler's own parsed intermediate representation, the structural information that the PlantUML parser generates internally during compilation, for deterministic, rule-based type classification of textual UML. This intermediate representation contains type-relevant evidence (diagram Java class, entity leaf types, visual symbol markers) that is generated as a byproduct of compilation but has not been systematically analyzed or utilized for classification purposes. The analysis of existing research confirms the relevance of developing a parser-based classification method that operates directly on PlantUML's intermediate representation, requiring no training data, no rendered images, and no external inference services.

Formulation of the Article's Objectives

The purpose of this article is to investigate the extent to which PlantUML's syntax encodes UML diagram type, by developing a purely syntactic heuristic classifier and measuring its agreement with semantic classification on a large corpus of real-world diagrams. To achieve this aim, the following tasks are formulated: (1) analyze PlantUML's internal type architecture, how nine user-facing UML types map to six internal diagram classes, and which types are syntactically distinguishable at the intermediate representation level; (2) design a three-tier heuristic classification algorithm operating on the compiler's intermediate representation, combining diagram class dispatch, entity-level LeafType inspection, and USymbol-based majority counting; (3) evaluate the classifier on 143,258 diagrams from the UML-in-the-Wild dataset using a semantic-syntactic agreement framework that does not assume either classifier as ground truth; (4) characterize the disagreement space through manual inspection to identify the structural mechanisms, syntactic overloading, factory chain ordering, container-vs-leaf asymmetry, responsible for classification divergence between syntactic and semantic approaches.

Presentation of the Main Material

PlantUML Internal Type Architecture

Diagram Type Resolution Mechanism

A defining characteristic of the PlantUML language is the absence of an explicit diagram type declaration. All nine standard UML diagram types share a single @startuml entry point, and the diagram type is determined internally by the compiler through an ordered factory chain. The PSystemBuilder class maintains a static sequence of PSystemFactory implementations, each capable of parsing a specific diagram type. When a source file is submitted for compilation, the factories are invoked sequentially; the first factory that successfully matches the input claims the diagram and produces a specific Java class representing the parsed result. This first-match-wins design has a significant consequence: factories registered earlier in the chain take priority over those registered later. In particular, ClassDiagramFactory precedes DescriptionDiagramFactory, which means that diagrams using class, interface, or package syntax are claimed as ClassDiagram even when the author intended a component diagram, a boundary condition quantified in the next sections.

The factory chain produces six distinct diagram classes for nine standard UML types, meaning that some user-facing types share the same internal representation. The complete mapping is presented in Table 1.

User Types to Internal Classes Mapping

Table 1

Mapping of UML diagram types to PlantUML's internal representation

User-facing type	Internal Java class	Classification evidence
Sequence	SequenceDiagram	Unique class
Activity	ActivityDiagram3	Unique class
State	StateDiagram	Unique class
Timing	TimingDiagram	Unique class
Class	ClassDiagram	Shared with Object
Object	ClassDiagram	Entity-level: LeafType
Use Case	DescriptionDiagram	Entity-level: LeafType
Component	DescriptionDiagram	Entity-level: LeafType
Deployment	DescriptionDiagram	Entity-level: LeafType

Four diagram types — sequence, activity, state, and timing — produce unique Java classes and can be classified trivially by checking the class of the parsed diagram object. The remaining five types form two ambiguity groups that share internal representations: the CLASS group (class and object, both represented as ClassDiagram) and the DESCRIPTION group (use case, component, and deployment, all represented as DescriptionDiagram). Notably, although PlantUML's UmlDiagramType enum defines an OBJECT value, it is never assigned, object diagrams receive the same type tag (CLASS) as class diagrams, providing no additional disambiguation.

The CLASS group encompasses class and object diagrams. Both are represented internally as ClassDiagram. Each entity within a class diagram carries a LeafType value: the object keyword produces entities with LeafType.OBJECT, while class-level keywords (class, interface, enum, abstract class, and others) produce corresponding class-like LeafType values. Distinguishing class from object diagrams therefore requires inspecting the LeafType of every entity in the diagram.

The DESCRIPTION group encompasses use case, component, and deployment diagrams. All three are represented internally as DescriptionDiagram. Use case entities carry a distinct LeafType.USECASE, providing a direct signal for use case diagram identification. Component and deployment entities, however, are both stored with LeafType.DESCRIPTION, they are indistinguishable at the LeafType level. The difference between component and deployment entities is encoded in the USymbol value, an internal marker that determines the visual shape assigned to each entity during rendering.

USymbol Analysis for the DESCRIPTION Group

The USymbol class in PlantUML's decoration subsystem assigns each entity in a DescriptionDiagram a visual shape corresponding to the keyword used in the source text (e.g., node, component, database, cloud). Since USymbol is

the only structural marker that differs between component and deployment entities, analyzing the association between USymbol values and diagram types is essential for designing a classification heuristic.

An analysis of the PlantUML Language Reference Guide (v1.2025.0) was performed to determine which USymbol values are associated with component diagrams, which with deployment diagrams, and which are used across both types. Section 7 (Component Diagram) defines the core component elements and explicitly lists certain keywords as grouping elements for component diagrams: "You can use several keywords to group components and interfaces together: package, node, folder, frame, cloud, database" [9]. Section 8 (Deployment Diagram) defines a larger set of declaration keywords. Comparing the two sections yields a three-tier classification of USymbol values by their type specificity, presented in Table 2.

Table 2

Three-tier grouping of USymbol values by diagram type specificity

Group	USymbol values
Component-exclusive	COMPONENT1, COMPONENT2, COMPONENT_RECTANGLE, INTERFACE
Deployment-exclusive	ARTIFACT, STORAGE, FILE, STACK
Shared	NODE, CLOUD, DATABASE, FOLDER, FRAME

All remaining USymbol values (RECTANGLE, AGENT, CARD, QUEUE, and others) are treated as neutral and do not influence the classification decision. The shared group constitutes the core classification challenge. The five symbols in this group — NODE, CLOUD, DATABASE, FOLDER, and FRAME — appear in both the component diagram documentation (as grouping/container elements) and the deployment diagram documentation (as infrastructure elements). In practice, DATABASE is the most frequent source of ambiguity: it is routinely used as a data store in component diagrams depicting software architecture, yet it is syntactically identical to its use in deployment diagrams representing infrastructure topology.

Heuristic Classification Method

Classifier Architecture

The classifier is implemented as a single Java class (UmlDiagramClassifier) embedded in the PlantUML v1.2025.9 source tree. No modifications were made to any existing PlantUML code, the classifier solely reads the output of the standard parsing pipeline. When a .puml file is submitted, PlantUML performs its full compilation sequence, preprocessing (macro expansion, include resolution, conditional compilation), factory chain type detection, and type-specific command interpretation, producing a fully resolved diagram object. The classifier then inspects this object and outputs one of nine UML diagram types. The tool is designed for batch execution and processes the entire evaluation corpus in a single pass.

Classification Algorithm

The classification follows a three-tier dispatch strategy, where each tier addresses one level of the type architecture described in the previous section. The tiers are applied in order; once a tier produces a classification, the remaining tiers are skipped.

Tier 1, Diagram class dispatch. If the compiled diagram is an instance of one of the four unique classes identified in Table 1, SequenceDiagram, ActivityDiagram3, StateDiagram, or TimingDiagram, the corresponding UML type (sequence, activity, state, or timing) is assigned directly. These four types are syntactically unambiguous: PlantUML uses dedicated parsers for each, and the resulting diagram classes are not shared with any other UML type. No further inspection is required.

Tier 2, Class vs. object. When the compiled diagram belongs to the CLASS group (Table 1), the classifier iterates over all entities and examines their types. In PlantUML's intermediate representation, the object keyword produces entities marked as objects, while class-level keywords, class, interface, enum, abstract class, and others, produce entities marked with their respective class-like types. The classification rule is: if every entity in the diagram is an object instance and none are class-like, the diagram is classified as an object diagram; otherwise, it is classified as a class diagram. Empty diagrams (containing no entities) default to class. The rationale is that a pure object diagram consists exclusively of object instances, whereas the presence of any class, interface, or enumeration signals a class diagram, even if object instances are also present.

Tier 3, Use case vs. component vs. deployment. When the compiled diagram belongs to the DESCRIPTION group (Table 1), disambiguation proceeds in two stages.

In the first stage, use case diagrams are identified. If any entity in the diagram is a use case element, or if only actor elements are present with no component or deployment symbols, the diagram is classified as a **use case** diagram. The use case check takes priority because the use case keyword produces a distinct entity type in the intermediate representation (LeafType.USECASE), providing a reliable signal.

In the second stage, component and deployment diagrams are distinguished using the three-tier symbol grouping established in Table 2. Each entity's visual symbol is examined and contributes to one of two counters:

- Entities with component-exclusive symbols (component, interface) increment the component counter.
- Entities with deployment-exclusive symbols (artifact, storage, file, stack) increment the deployment counter.
- Entities with shared symbols (node, cloud, database, folder, frame) also increment the deployment counter, they serve as weak deployment evidence that can be outvoted by component-exclusive symbols.

The decision rules then apply: if only deployment evidence is present, the diagram is classified as deployment; if only component evidence is present, as component; if both are present, the majority counter wins, with ties resolved in favor of component; if no distinguishing symbols are found, the default is component.

The treatment of shared symbols is a deliberate design choice. Assigning them to the deployment counter preserves the ability to detect deployment diagrams that use only infrastructure keywords (node, cloud, database), while the majority-counting mechanism ensures that a component diagram containing an incidental database is not misclassified as deployment, the component-exclusive symbols outvote the shared one. This asymmetric counting reflects the empirical observation that shared symbols appear in both diagram types but are more strongly indicative of deployment intent when no component-exclusive symbols are present.

Method Limitations

The first limitation is the component/deployment boundary. Two mechanisms make this boundary irreducible for any purely syntactic classifier. Keyword semantic overloading occurs when authors use deployment-associated keywords (database, file, artifact) to represent non-infrastructure concepts, for example, a database element depicting a data store within a software architecture diagram. The classifier counts this as deployment evidence, while the author's intent is a component diagram. Container-vs-leaf asymmetry arises because certain keywords (notably node) create grouping containers rather than leaf entities. The classifier inspects only leaf entities; bracket-notation components nested inside nodes are visible, but the enclosing nodes are not, causing component symbols to outnumber deployment symbols even in diagrams that are structurally deployment-oriented.

The second limitation is the component-to-class boundary. PlantUML's factory chain processes ClassDiagramFactory before DescriptionDiagramFactory. When a component diagram uses class, interface, or package syntax, which is common in diagrams that show both software components and their class-level interfaces, the ClassDiagramFactory claims the diagram first. These diagrams are compiled as ClassDiagram, placing them outside the reach of the DESCRIPTION group heuristic entirely. This boundary condition affects a substantial number of diagrams in the evaluation corpus (quantified in the next section).

The third limitation is the class-to-object boundary. The "all objects" heuristic classifies a diagram as an object only if every entity is an object instance. In practice, some authors use the object keyword as a lightweight alternative to class for domain modeling, producing diagrams that are semantically class diagrams but syntactically indistinguishable from object diagrams. Conversely, diagrams that mix object and class elements are classified as class, even when the author's primary intent was to show object instances.

Experimental Evaluation

Experiment Design

The classifier was evaluated on the UML-in-the-Wild dataset [10] — a publicly available corpus of 143,427 PlantUML diagrams mined from open-source repositories on GitHub and Bitbucket using the World of Code infrastructure [11]. Of these, 143,258 produced comparable classification pairs; the remaining 169 failed PlantUML compilation and were excluded.

Each diagram in the corpus carries two independent type labels. The syntactic label is produced by the proposed classifier, which inspects PlantUML's intermediate representation as described in the previous section. The semantic label is the primary_type annotation published as part of the dataset's metadata, assigned by an LLM-based classifier that interpreted each diagram's content and communicative intent to determine one of nine standard UML types. The LLM classification methodology is documented in the dataset's Zenodo record and is not described in this paper; the labels are used solely as a semantic reference point. The comparison between these two label sets measures semantic-syntactic agreement, the degree to which syntactic structure aligns with semantic intent. Neither set of labels is treated as ground truth; manual inspection validates both classifiers' errors.

Results

The overall semantic-syntactic agreement across all nine UML types is **95.96%** (137,474 of 143,258 diagrams). Table 3 presents the per-type breakdown, sorted by agreement rate.

Table 3

Per-type semantic-syntactic agreement

Diagram type	Semantic count	Agreed	Agreement	Classifier tier
Class	73,875	73,617	99.7%	Tier 2
Sequence	35,232	34,890	99.0%	Tier 1
Activity	9,793	9,595	98.0%	Tier 1
State	4,343	4,191	96.5%	Tier 1
Timing	189	178	94.2%	Tier 1
Use case	7,114	6,658	93.6%	Tier 3
Object	2,450	2,279	93.0%	Tier 2
Component	8,185	5,068	61.9%	Tier 3
Deployment	2,077	998	48.1%	Tier 3

The results reveal a clear pattern: diagram types classified by Tier 1 (unique diagram classes) and Tier 2 (LeafType inspection) achieve 93–99.7% agreement, confirming that PlantUML's syntax is highly type-distinctive for seven of nine UML types. Agreement drops sharply for the two types that depend on Tier 3 USymbol-based counting, component (61.9%) and deployment (48.1%), reflecting the syntactic overloading identified in Section 4.1.3. The full 9×9 confusion matrix (Figure 1) confirms this pattern: the matrix is near-diagonal for seven types, with visible off-diagonal mass concentrated in the component/deployment/class/usecase cluster.

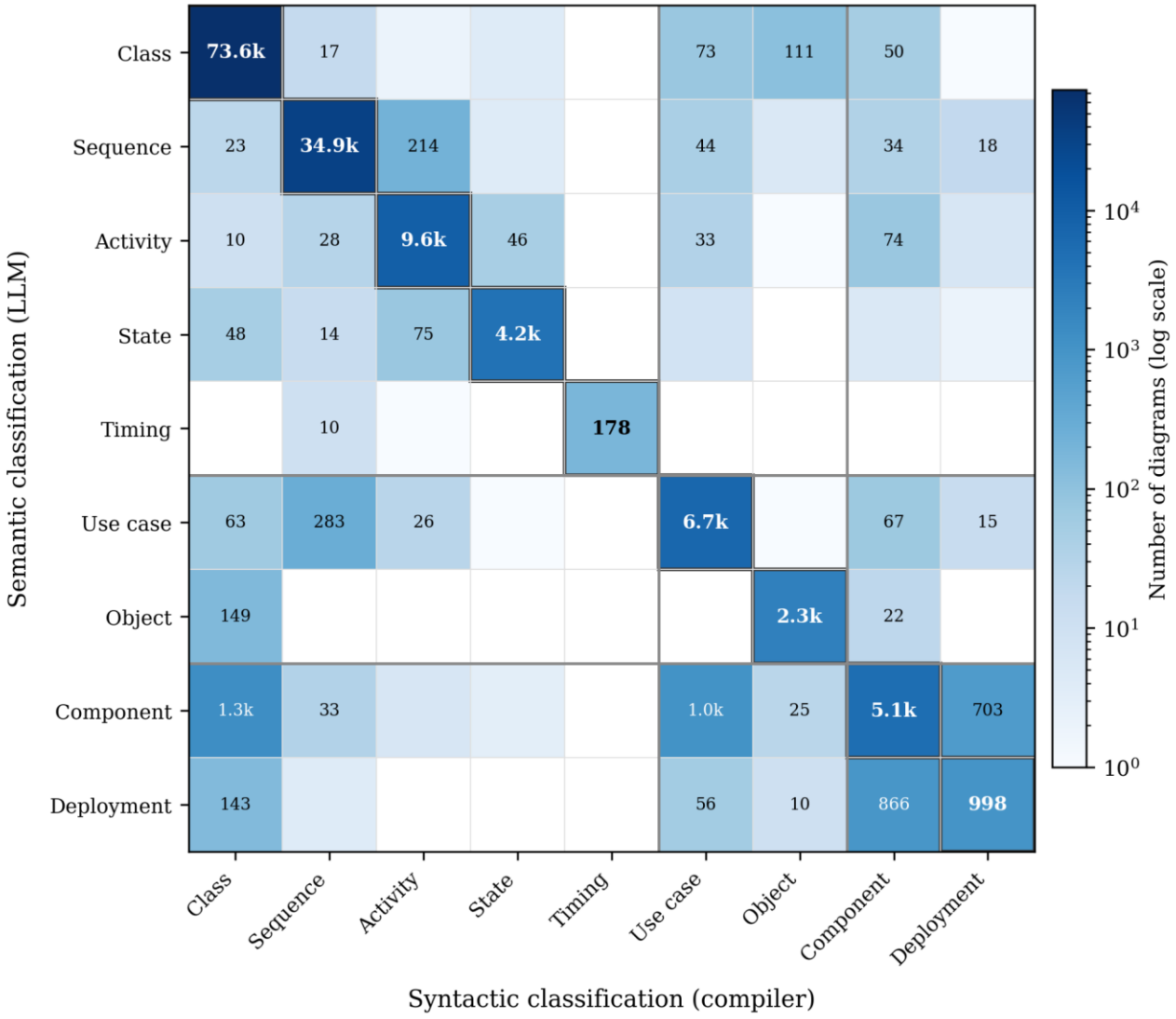

Figure 1. Confusion matrix: semantic (LLM) vs syntactic (compiler) classification

Iterative refinement

The final classifier design was reached through controlled experimentation. Table 4 summarizes the three versions evaluated on the full corpus, each isolating one design variable.

Table 4

Classifier refinement: strategy comparison on the full corpus

Version	Decision strategy	Symbol grouping	Overall	Component	Deployment
v1	Boolean presence	Original (two groups)	95.66%	52.3%	64.8%
v3	Majority counting	Original (two groups)	95.96%	61.8%	48.3%
v3	Majority counting	Three-tier (Table 2)	95.96%	61.9%	48.1%

Replacing boolean presence detection with majority counting (v1 → v2) improved component agreement by 9.5 percentage points at a cost of 16.5 points for deployment, yielding a net gain of 432 correctly classified files. The improvement arises because component diagrams containing incidental deployment symbols (e.g., a single database alongside multiple components) are no longer misclassified, the component-exclusive symbols outvote the minority. Regrouping symbols into the three-tier classification derived from the PlantUML documentation (v2 → v3) placed the heuristic on a specification-justified basis with negligible numerical change (+12 component, −5 deployment). The stability of the overall agreement at 95.96% across two different strategies confirms that this value represents the practical ceiling for entity-level syntactic classification.

Disagreement Space Analysis

The 5,784 disagreements between syntactic and semantic classification are not uniformly attributable to classifier error. To characterize the nature of these disagreements, manual inspection of 110 files across the eight largest disagreement pairs was performed. For each pair, a random sample of 10–15 files (fixed seed for reproducibility) was drawn, and each file was assessed by examining the PlantUML source, file path, and diagram content. Each file was assigned one of three labels: LLM correct (the semantic label better reflects the diagram's communicative intent), compiler correct (the syntactic label is the more appropriate classification), or ambiguous (the diagram legitimately spans type boundaries, is an empty stub, or is not a standard UML diagram). The inspection scope is determined by the disagreement space, not the full corpus: the 137,474 diagrams where both classifiers agree require no manual validation, and the inspection targets only the 5,784 files where classifications diverge. The eight selected pairs account for 4,673 of these disagreements (80.8%). The sample size of 10–15 files per pair is designed for qualitative pattern identification, determining which structural mechanisms cause each disagreement type, rather than for precise quantitative estimation of error rates. Within several pairs, the inspected samples reached saturation: the sequence → activity pair yielded 15 of 15 compiler-correct judgments, the class → object pair yielded 15 of 15 LLM-correct judgments, and the two cross-tier pairs produced 0 LLM-correct judgments across 30 files combined, indicating that additional sampling would be unlikely to alter the identified pattern. Table 5 presents the per-pair results.

Table 5

Per-type semantic-syntactic agreement

Disagreement (semantic → syntactic)	Count	Sample	LLM correct	Compiler correct	Ambiguous
component → class	1,303	15	9	6	0
component → use case	1,044	15	2	13	0
deployment → component	866	10	7	1	2
component → deployment	703	10	8	1	1
case → sequence	283	15	0	9	6
sequence → activity	214	15	0	15	0
object → class	149	15	8	7	0
class → object	111	15	15	0	0

The inspection described in the table revealed three distinct error regimes.

Compiler-correct regime (cross-tier pairs). The use case → sequence (283 files) and sequence → activity (214 files) pairs are cases where the syntactic classifier is overwhelmingly or unanimously correct. In the use case → sequence pair, the LLM misinterprets the actor keyword, which is valid in both use case and sequence diagrams, as a use case signal, even when the diagram contains sequence-specific syntax (message arrows, participants, interaction fragments). In the sequence → activity pair, the LLM misidentifies activity diagram swimlanes as sequence interactions because both depict alternating actions between participants. In both cases, PlantUML's dedicated parsers (Tier 1) produce unambiguous results: 0 of 30 inspected files supported the LLM's label.

LLM-correct regime (intra-DESCRIPTION pairs). The component → deployment (703 files) and deployment → component (866 files) pairs exhibit the reverse pattern: the LLM is correct in 15 of 20 inspected files. Two structural mechanisms cause the syntactic classifier to fail. First, keyword semantic overloading: authors routinely use deployment-associated keywords, database for data stores, file for data formats, artifact for software packages, to represent non-infrastructure concepts within component diagrams. The classifier counts these as deployment evidence, while the author's intent is a component architecture. Second, container-vs-leaf asymmetry: the node keyword creates grouping containers rather than leaf entities, making them invisible to the classifier's entity-level inspection. Deployment diagrams that use node containers with bracket-notation components inside them appear component-dominant to any entity-level classifier.

Figures 2 and 3 illustrate the two mechanisms on real diagrams from the evaluation corpus. Figure 2 presents a deployment diagram titled "Haskell Debugging System Deployment Diagram" by its author. Two node containers (VSCODE and Haskell Debugger) host four component entities representing software processes and libraries. The node keyword creates grouping containers that are not included in PlantUML's leaf entity collection; only the nested component

entities are visible to the classifier. The resulting leaf inventory contains four component entities and zero node entities, producing a component classification for a diagram that is unambiguously a deployment view. Figure 3 presents the reverse case: a software architecture diagram depicting a distributed file storage service. The author uses the component keyword for software services (DFSDatasafeService, DFSConnection) and the database keyword for data keystores, not infrastructure nodes. The classifier counts the database entities as deployment evidence, which together with node and frame containers outnumbers the component entities, resulting in a deployment classification despite the diagram's architectural intent.

```
@startuml

title <size:18>Haskell Debugging System Deployment Diagram</size>

node "VSCODE" {
  component "phoityne-vscode" <<extension>>
}

node "Haskell Debugger" {
  component "haskell-dap" <<library>>
  component "hda" <<process>>
  rectangle ghci #line.dashed {
    component "ghci-dap" <<process>>
  }
}

[phoityne-vscode] -- [hda] : <<stdio>>
[hda] --> [ghci-dap] : <<stdio>>
[hda] -l-> [haskell-dap] : <<use>>
[ghci-dap] -l-> [haskell-dap] : <<use>>

note left of "phoityne-vscode"
  hda4vsc
end note

@enduml
```

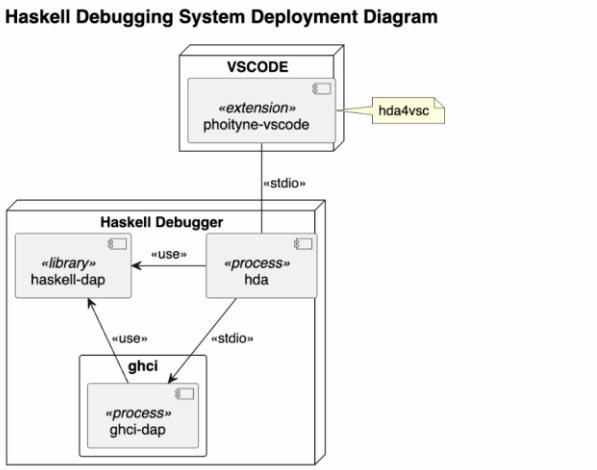

Figure 2. A deployment diagram classified as component due to container-vs-leaf asymmetry

```
skinparam rectangle {
    BorderStyle<<MainGrp>> dashed
}

left to right direction

hide stereotype

rectangle " " as main <<MainGrp>> {
  component DFSDatasafeService as svc

together {
  rectangle " " as sys <<MainGrp>> {
    component DFSConnection as sys.conn
    database "SYSTEM DFS" as sys.db
    frame sys.frame [
      For each user (user space):
      - keystore (public)
      - user protected: INBOX
      - user protected: user's DFS credentials
      - free: PUBLIC KEYs
    ]

    sys.conn --[hidden] sys.db
    sys.db -right-> sys.frame

  }

  node users {

    rectangle " " as usr1 <<MainGrp>> {
      component DFSConnection as usr1.conn
      database "USER DFS" as usr1.db
      frame usr1.frame [
        - keystore (private)
        - user protected: private user data
      ]

      usr1.conn --[hidden] usr1.db
      usr1.db -right-> usr1.frame

    }
    rectangle " " as usr2 <<MainGrp>> {
      component DFSConnection as usr2.conn
      database "USER DFS" as usr2.db
      frame usr2.frame [
        - keystore (private)
        - user protected: private user data
      ]

      usr2.conn --[hidden] usr2.db
      usr2.db -right-> usr2.frame
    }

  }
  }
```

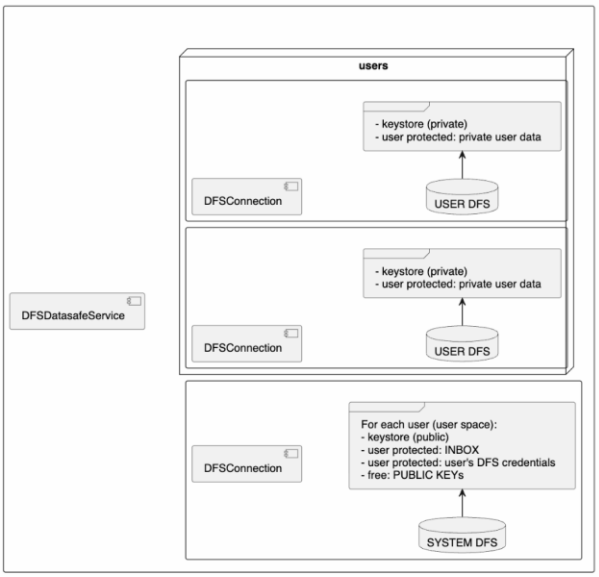

Figure 3. A component diagram classified as deployment due to keyword semantic overloading

Mixed regime. The remaining four pairs show no clear advantage for either classifier. The class → object pair (111 files) is unanimously LLM-correct: all 15 inspected files use the object keyword as a lightweight alternative to class for domain modeling, producing diagrams that are semantically class diagrams but satisfy the classifier's "all objects" heuristic. The object → class pair (149 files) splits evenly (8 LLM / 7 compiler), these are mixed diagrams combining object and class elements. The component → class pair (1,303 files) splits 9/6 in favor of the LLM, these are component diagrams using class-level syntax (interface, package) that the factory chain claims before the DESCRIPTION parser. The component → use case pair (1,044 files) is an outlier where the compiler is overwhelmingly correct (13/15): a single auto-generation tool for Helm chart dependency maps produces the majority of these files, deliberately using use case notation for Docker images. Table 6 summarizes the aggregate inspection results

Table 6

Aggregate manual inspection results (n = 120)

Judgment	Count	Proportion
LLM correct	49	45%
Compiler correct	52	47%
Ambiguous	9	8%

The 95% confidence intervals for the LLM-correct (35–54%) and compiler-correct (38–57%) proportions overlap substantially, confirming that neither classifier demonstrates a statistically significant advantage on the inspected sample.

Neither classifier is uniformly superior. The syntactic classifier achieves near-perfect agreement for seven of nine types where PlantUML's syntax is type-distinctive. The remaining disagreement space, concentrated in the component and deployment types, quantifies the degree of syntactic ambiguity inherent in PlantUML's design and identifies the specific structural mechanisms responsible.

Conclusions and Prospects for Further Research

The results of this study answer the research question posed in Section 1: PlantUML's syntax is highly type-distinctive for seven of nine standard UML diagram types. For these types, the syntactic structure of the intermediate representation reliably encodes the author's semantic intent, and a purely syntactic classifier can substitute for semantic analysis with 93–99.7% agreement. The component/deployment boundary, where agreement drops to 48–62%, represents a language-level limitation rather than a classifier deficiency: the same keywords (database, node, cloud, file, artifact) serve genuinely different purposes across diagram types, and the parser's container-vs-leaf asymmetry structurally inverts the entity-level evidence. Manual inspection confirmed that even within this disagreement space, a non-trivial proportion of files are cases where the LLM's semantic label is itself incorrect, the syntactic classifier is not the sole source of disagreement.

A three-tier heuristic classifier for all nine UML diagram types was developed, operating on PlantUML's compiled intermediate representation. The classifier is implemented as a single Java class embedded in PlantUML v1.2025.9 with no modifications to any existing source code. It is deterministic, requires no training data or external APIs, and processes the entire evaluation corpus in a single batch pass.

Evaluation on 143,258 diagrams from the UML-in-the-Wild dataset [10] yielded 95.96% overall semantic-syntactic agreement. Manual inspection of 110 files across eight major disagreement pairs identified three error regimes, compiler-correct (cross-tier), LLM-correct (intra-DESCRIPTION), and mixed, with an aggregate split of 47% compiler-correct, 45% LLM-correct, and 8% ambiguous. The semantic-syntactic agreement framework provides a principled evaluation method without assuming either classifier as ground truth.

The classifier is suited for fast, deterministic type labeling of large PlantUML corpora where 95%+ overall agreement is sufficient, empirical software engineering studies, diagram search and retrieval systems, educational platforms, and quality assessment tools. The method is tied to PlantUML v1.2025.9; internal APIs may change across versions. Diagrams that fail compilation (169 of 143,427 in the evaluation corpus) cannot be classified.

Prospects for further research include: (1) explicit diagram type declarations in PlantUML (e.g., @startcomponent, @startdeployment) that would eliminate the classification problem at the language level; (2) hybrid syntactic-semantic classifiers that combine the proposed entity-level analysis with lightweight semantic cues, keyword co-occurrence patterns, file path heuristics, diagram title parsing, to bridge the component/deployment gap without requiring full LLM inference; (3) extension of the classification method to non-UML PlantUML diagram types (mindmap, Gantt, wireframe, and others) that use distinct entry points and intermediate representations. The evaluation data are publicly available [10].

Література

1. Romeo, J., Raglianti, M., Nagy, C., & Lanza, M. (2025). UML is Back. Or is it? Investigating the Past, Present, and Future of UML in Open Source Software. Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE '25), 2342–2354. https://doi.org/10.1109/ICSE55347.2025.00155

2. Hebig, R., Ho-Quang, T., Chaudron, M. R. V., Robles, G., & Fernandez, M. A. (2016). The Quest for Open Source Projects that Use UML: Mining GitHub. Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS '16), 173–183. https://doi.org/10.1145/2976767.2976778

3. Robles, G., Ho-Quang, T., Hebig, R., Chaudron, M. R. V., & Fernandez, M. A. (2017). An Extensive Dataset of UML Models in GitHub. Proceedings of the 14th International Conference on Mining Software Repositories (MSR '17), 519–522. https://doi.org/10.1109/MSR.2017.48

4. Reggio, G., Leotta, M., & Ricca, F. (2014). Who Knows/Uses What of the UML: A Personal Opinion Survey. Proceedings of the 17th International Conference on Model Driven Engineering Languages and Systems (MODELS '14), LNCS 8767, 149–165. https://doi.org/10.1007/978-3-319-11653-2_10

5. Shcherban, S., Liang, P., Li, Z., & Yang, C. (2021). Multiclass Classification of UML Diagrams from Images Using Deep Learning. International Journal of Software Engineering and Knowledge Engineering, 31(11–12), 1683–1698. https://doi.org/10.1142/S0218194021400179

6. Torcal, J., Moreno, V., Llorens, J., & Granados, A. (2024). Creating and Validating a Ground Truth Dataset of Unified Modeling Language Diagrams Using Deep Learning Techniques. Applied Sciences, 14(23), 10873. https://doi.org/10.3390/app142310873

7. Bates, A., Vavricka, R., Carleton, S., Shao, R., & Pan, C. (2025). Unified Modeling Language Code Generation from Diagram Images Using Multimodal Large Language Models. Machine Learning with Applications, 20, 100660. https://doi.org/10.1016/j.mlwa.2025.100660

8. Conrardy, A., & Cabot, J. (2024). From Image to UML: First Results of Image Based UML Diagram Generation Using LLMs. Proceedings of the LLM4MDE Workshop (STAF 2024), CEUR Workshop Proceedings, Vol. 3727, 55–65. https://doi.org/10.48550/arXiv.2404.11376

9. PlantUML Language Reference Guide (v1.2025.0). Sections 7 (Component Diagram) and 8 (Deployment Diagram). [Online resource]. Available at: https://plantuml.com/ (Accessed: 28.03.2026)

10. Polishchuk, V. (2025). UML-in-the-Wild: A Dataset of UML Diagrams in PlantUML Notation from Open-Source Repositories. Zenodo. https://doi.org/10.5281/zenodo.18952371

11. Ma, Y., Dey, T., Bogart, C., Amreen, S., Valiev, M., Tutko, A., Kennard, D., Zaretzki, R. L., & Mockus, A. (2021). World of Code: Enabling a Research Workflow for Mining and Analyzing the Universe of Open Source VCS Data. Empirical Software Engineering, 26, 22. https://doi.org/10.1007/s10664-020-09905-9

References

1. Romeo, J., Raglianti, M., Nagy, C., & Lanza, M. (2025). UML is Back. Or is it? Investigating the Past, Present, and Future of UML in Open Source Software. Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE '25), 2342–2354. https://doi.org/10.1109/ICSE55347.2025.00155

2. Hebig, R., Ho-Quang, T., Chaudron, M. R. V., Robles, G., & Fernandez, M. A. (2016). The Quest for Open Source Projects that Use UML: Mining GitHub. Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS '16), 173–183. https://doi.org/10.1145/2976767.2976778

3. Robles, G., Ho-Quang, T., Hebig, R., Chaudron, M. R. V., & Fernandez, M. A. (2017). An Extensive Dataset of UML Models in GitHub. Proceedings of the 14th International Conference on Mining Software Repositories (MSR '17), 519–522. https://doi.org/10.1109/MSR.2017.48

4. Reggio, G., Leotta, M., & Ricca, F. (2014). Who Knows/Uses What of the UML: A Personal Opinion Survey. Proceedings of the 17th International Conference on Model Driven Engineering Languages and Systems (MODELS '14), LNCS 8767, 149–165. https://doi.org/10.1007/978-3-319-11653-2_10

5. Shcherban, S., Liang, P., Li, Z., & Yang, C. (2021). Multiclass Classification of UML Diagrams from Images Using Deep Learning. International Journal of Software Engineering and Knowledge Engineering, 31(11–12), 1683–1698. https://doi.org/10.1142/S0218194021400179

6. Torcal, J., Moreno, V., Llorens, J., & Granados, A. (2024). Creating and Validating a Ground Truth Dataset of Unified Modeling Language Diagrams Using Deep Learning Techniques. Applied Sciences, 14(23), 10873. https://doi.org/10.3390/app142310873

7. Bates, A., Vavricka, R., Carleton, S., Shao, R., & Pan, C. (2025). Unified Modeling Language Code Generation from Diagram Images Using Multimodal Large Language Models. Machine Learning with Applications, 20, 100660. https://doi.org/10.1016/j.mlwa.2025.100660

8. Conrardy, A., & Cabot, J. (2024). From Image to UML: First Results of Image Based UML Diagram Generation Using LLMs. Proceedings of the LLM4MDE Workshop (STAF 2024), CEUR Workshop Proceedings, Vol. 3727, 55–65. https://doi.org/10.48550/arXiv.2404.11376

9. PlantUML Language Reference Guide (v1.2025.0). Sections 7 (Component Diagram) and 8 (Deployment Diagram). [Online resource]. Available at: https://plantuml.com/ (Accessed: 28.03.2026)

10. Polishchuk, V. (2025). UML-in-the-Wild: A Dataset of UML Diagrams in PlantUML Notation from Open-Source Repositories. Zenodo. https://doi.org/10.5281/zenodo.18952371

11. Ma, Y., Dey, T., Bogart, C., Amreen, S., Valiev, M., Tutko, A., Kennard, D., Zaretzki, R. L., & Mockus, A. (2021). World of Code: Enabling a Research Workflow for Mining and Analyzing the Universe of Open Source VCS Data. Empirical Software Engineering, 26, 22. https://doi.org/10.1007/s10664-020-09905-9