A Domain Specific Language for Pattern Specification and Automated Validation for Metamodel-Based Languages

Ankita Vyas
Department of Electrical and Computer Engineering
McGill University
Montreal, Quebec, Canada
ankita.vyas@mail.mcgill.ca

Gunter Mussbacher
Department of Electrical and Computer Engineering
McGill University
Montreal, Quebec, Canada
gunter.mussbacher@mcgill.ca

***Abstract*—Large collections of software artifacts often contain recurring structural patterns that can provide useful evidence for language understanding or empirical analysis, and provide requirements for language improvement of self-adaptable languages. OCL-based queries may specify such patterns but become more complex when patterns involve nested relations, repeated structures, or sequence-sensitive constraints. This paper presents the Pattern Definition Language (PDL), a domain-specific language for specifying structural patterns over EMF-based metamodels. PDL directly supports both unordered and ordered structural constraints, allowing sequence-sensitive patterns to be expressed declaratively without manually encoding positional logic. To operationalize these specifications, a model-to-text transformation automatically generates executable validators from PDL models that enable the detection of the specified pattern in artifacts. The approach is evaluated on a dataset of 29,069 DevOps pipeline YAML files using two complementary pattern discovery settings: statistical pattern mining and LLM prompting. The identified pipeline patterns are reconstructed in PDL, transformed into validators, and compared with mining-based occurrence results and independent Python-based matchers, respectively. The results show exact agreement for most evaluated patterns. All mismatch cases are explainable through differences between raw YAML-level structural matching / Python-based parsing and metamodel-level interpretation, demonstrating that the proposed workflow can validate structural pattern occurrences. The results indicate that PDL and its model-to-text transformation workflow provide a systematic basis for structural pattern specification and automated validation in metamodel-based languages.**

***Index Terms*—Model-Driven Engineering, Domain-Specific Language, Pattern Definition Language, Model-to-Text Transformation, Structural Pattern Validation**

I. Introduction and Motivation

Self-adaptable languages [1] aim to establish a feedback loop between the language engineering environment and the modeling environment where the language is used. Based on the use of the language, patterns can be identified that inform the – ideally automatic – evolution of the language. For example, the recurring pattern of nested if statements could be automatically detected and switch/case statements suggested as an improvement by a human actor or an intelligent system-based on past experience. In general, recurring structural patterns observed across large model collections can provide feedback about how a language is actually used in practice. Such evidence can help identify frequently recurring idioms, verbose structures, ordering conventions, or under-constrained language constructs that may warrant language improvement. Given a description of the improvement, the metamodel of the language as well as the existing models could then be automatically evolved. Each pattern / improvement pair succinctly describes a requirement for language evolution that must be supported by self-adaptable languages.

To support such a feedback loop, structural patterns must not only be expressible declaratively, but also validated systematically and consistently across large model datasets. We postulate that the realization of this feedback loop requires several capabilities: (i) a detection mechanism for recurring language use (i.e., a pattern), (ii) a pattern definition language to specify the pattern, (iii) a validation mechanism to find occurrences of a given pattern, (iv) the ability to improve the pattern, and (v) a mechanism to co-evolve the language's metamodel and models given the improvement. This paper focuses on the automation of steps (ii) and (iii), i.e., structural pattern specification and validation for metamodel-based languages.

Object Constraint Language (OCL) queries over the target metamodel [2] are an existing way to express structural patterns over models. For example, consider (i) a DevOps pipeline language where a *Pipeline* consists of *Jobs* which consist of *Steps* and a set of *ParameterValues* and (ii) a job-level pipeline pattern where at least one job has a *name* (line 4 in Listing 1), a *dependKW* keyword (line 5), a non-empty list of *steps* (line 6), and three parameters (*runs-on*, *name*, and *if*) (lines 7-19). OCL can express this kind of pattern by navigating from the *Pipeline* root to its *jobs* and checking the required attributes, *jobParameterValues*, and *steps*. However, the main difficulty appears when the pattern is sequence-sensitive.

For example, if the required job parameters must appear in the order *runs-on*, then *name*, then *if*, the OCL query must explicitly convert the parameter collection into a sequence and compare element positions, as shown on lines 7, 8, 13, and 16 in Listing 1. This example shows that OCL is

```
context Pipeline
def: hasPatternOccurrence(): Boolean =
self.jobs->exists(j |
    not j.name.oclIsUndefined() and
    not j.dependKW.oclIsUndefined() and
    j.steps->notEmpty() and
    let params : Sequence(ParameterValue) =
    j.jobParameterValues->asSequence() in
    params->exists(pRun |
        pRun.name = PipelineKeyword::PPL_KW_RUNS_ON and
        params->exists(pName |
            pName.name = PipelineKeyword::PPL_KW_NAME and
            params->indexOf(pName) > params->indexOf(pRun) and
            params->exists(pIf |
                pIf.name = PipelineKeyword::PPL_KW_IF and
                params->indexOf(pIf) > params->indexOf(pName)
            )
        )
    )
)
```

Listing 1: OCL Query with Ordered Criteria

expressive enough to define ordered structural checks, but the resulting query becomes longer and harder to maintain. The pattern author must manually encode positional logic through sequence operations and index comparisons.

As ordering is clearly important in imperative languages, it should therefore be explicitly supported in a pattern specification framework. This limitation motivates the need for a more declarative and reusable approach. The user should be able to specify the intended structure directly in terms of dedicated pattern language concepts, while the validation logic should be generated automatically.

To address this need, this paper presents the Pattern Definition Language (PDL), an Xtext-based domain-specific language for specifying structural patterns over EMF-based metamodels. PDL provides explicit support for unordered and ordered structural constraints, so that sequence-sensitive patterns can be represented without manually encoding positional logic in OCL or handwritten Java code. A PDL specification is supplied to an Acceleo-based model-to-text transformation, which generates an executable Java validator for the declared pattern. The generated validator enables the detection of the specified pattern in artifacts. We argue that a custom-made solution allows full control over the pattern language, transformation, and generated validation workflow. Hence, we choose to not extend OCL in this work but instead introduce PDL as a separate DSL.

The paper evaluates this approach using patterns obtained from two complementary discovery settings over a dataset of 29,069 DevOps pipeline YAML files [3], supported by an existing Xtext-based DevOps pipeline editor [3]. PDL is used to specify patterns identified through a statistical mining workflow in the first setting and patterns identified by an LLM in the second setting. Together, these evaluations assess whether PDL can express recurring structural patterns discovered through different, realistic mechanisms and whether the generated validators correctly detect the patterns in the dataset.

The contribution of this paper is not a new general-purpose model query language, but a dedicated pattern specification and validation workflow for metamodel-based languages. First, the paper introduces PDL as a declarative language for specifying structural patterns over EMF-based models, while treating ordering as a first-class pattern property. Second, it presents a model-to-text transformation that automatically generates Java validators from PDL specifications. Third, it evaluates the approach on a large DevOps pipeline dataset using both statistically mined patterns and LLM-found structural patterns, and demonstrates how generated validators can support systematic structural evidence collection for recurring pattern analysis over metamodel-based languages.

In the remainder of this paper, Section II presents background and related work. Section III introduces the PDL metamodel and explains how structural patterns are represented. Section IV gives an overview of the model-to-text transformation used to generate Java validators from PDL specifications. Section V describes the evaluation process, including the dataset, the data mining-based validation workflow, the LLM-based validation workflow, and the evaluated patterns. Section VI discusses the validation results. Finally, Section VII concludes the paper and outlines future work.

II. Background and Related Work

This section summarizes the concepts and related approaches that position the proposed PDL-based validation workflow.

A. Model-Driven Engineering, Domain-Specific Languages, and Model-to-Text Transformation

Model-Driven Engineering (MDE) treats models as primary artifacts for specification, transformation, and generation [4]. In this work, structural patterns are analyzed over EMF/Ecore-based model instances [5], [6]. Language workbenches and DSL frameworks such as Xtext [7] and MontiCore [8] provide infrastructure for defining textual domain-specific languages with parsers, editors, validation support, and metamodel-based representations. Xtext supports both the existing DevOps pipeline language used in the evaluation [3] and the implementation of PDL itself as a textual DSL [7], [9].

Model transformation and code generation are also central to the proposed approach. Prior work has studied model transformation intents and tool support more generally [10], [11], while template-based code generation has been used to generate implementation artifacts from higher-level specifications [12]. Acceleo is particularly relevant because it supports template-based text generation from EMF models [13]. In this paper, Acceleo is used specifically to generate Java validators from PDL models, thereby operationalizing declarative structural pattern specifications as executable validation artifacts.

B. Model-Level Constraint Specification and Validation

A natural baseline for structural validation over EMF models is the Object Constraint Language (OCL). OCL provides a declarative mechanism for specifying constraints and queries over model elements [2], and has also been empirically compared with Java for constraint specification over UML models [14]. OCL is expressive enough to describe structural checks, including nested and ordered constraints. However, as shown in the motivating example, ordered patterns require explicit sequence operations and positional logic, which makes

repeated pattern specification harder to write and maintain. PDL aims to address this issue by treating ordering as a first-class language concept while also supporting the generation of Java validators.

Dedicated model query languages aim to make model queries easier to express than textual OCL expressions. Stein et al. proposed Query Models [15] as a graphical notation for specifying model queries, while VMQL [16] provides a visual language for ad-hoc model querying. MOCQL [17] further explores a declarative model query language and evaluates its usability through controlled experiments. Unlike these works, this paper does not claim or measure usability improvements over OCL. Instead, it focuses on a metamodel-oriented pattern specification workflow in which structural pattern specifications are transformed into standalone Java validators for corpus-level occurrence analysis.

Several model query and validation approaches also target EMF-based models. EMF IncQuery provides graph-based query specification over EMF models [18], while VIATRA supports reactive and incremental model query and transformation capabilities [19]. The Eclipse Epsilon ecosystem provides a family of languages for model management tasks over EMF and other modeling technologies [20]. In particular, the Epsilon Pattern Language supports pattern matching through roles, guards, domains, and match models [21], [22], while the Epsilon Validation Language supports model validation through context-based constraints, messages, and fixes [23].

These approaches provide strong support for model-level querying, pattern matching, validation, and transformation. PDL is positioned differently: it does not aim to replace general-purpose query or validation languages such as OCL, IncQuery/VIATRA, EPL, or EVL. Instead, it focuses on the narrower task of specifying recurring structural patterns in a compact metamodel-oriented form and automatically generating executable validators for dataset-level occurrence counting.

C. DevOps Pipeline Analysis and Pattern Discovery

The evaluation domain of this paper is DevOps pipeline specifications. Bedekar and Mussbacher introduced a multi-platform DevOps pipeline specification language and dataset for analyzing pipeline specifications across platforms [3]. This prior work provides the metamodel-based environment and dataset used in this paper. Other research has examined CI/CD pipelines from perspectives such as reengineering [24], developer perception [25], workflow maintenance [26], and automated workflow completion [27]. These studies show that pipeline specifications are important and evolving software artifacts whose structure deserves systematic analysis. The present work complements this research by focusing on reusable structural pattern specification and generated validation over model-based pipeline instances.

The evaluation also uses statistical pattern mining to identify recurring feature combinations before reconstructing selected patterns in PDL. FP-Growth is used for frequent itemset mining over the extracted feature transactions [28]. In the proposed workflow, mining provides candidate recurring patterns, while PDL and the generated Java validators validate those patterns.

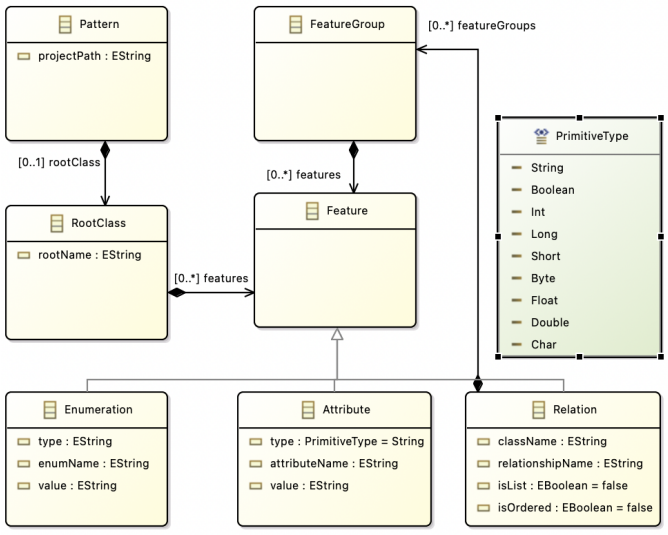

Fig. 1: Metamodel for Pattern Definition Language

D. Self-adaptable Languages

This work is connected to the broader goal of self-adaptable languages. Jouneaux et al. argue that self-adaptable languages rely on feedback loops that use evidence from language use to guide adaptations of syntax, semantics, or pragmatics [1]. In this context, recurring structural patterns can provide evidence about how a language is used in practice and can inform future language improvement. PDL and the generated validators contribute to this vision by supporting the specification and validation of such patterns over large model corpora. The proposed workflow, therefore, provides an evidence collection mechanism that can support later decisions about whether recurring structures should be simplified, constrained, reified, or otherwise adapted in future versions of a language.

III. PDL Metamodel

Figure 1 shows the metamodel of the Pattern Definition Language (PDL).[1] At the top level, the metamodel contains the `Pattern` metaclass, representing one pattern specification. It stores the `projectPath`, which identifies the Java package of the target EMF metamodel to be analyzed. The Pattern contains a `RootClass`, which defines the starting point for pattern matching. For example, in the DevOps pipeline case, the root class is *Pipeline*. The `RootClass` stores the name of the target metamodel class through `rootName` and contains the features (i.e., `Attribute`, `Enumeration`, and/or `Relation`) that must be checked from the starting point.

The main navigation construct in PDL is the `Relation` metaclass. A relation represents a traversal step from one model element to another through a reference in the target metamodel. It stores the target `className`, the

[1] https://github.com/AnkitaVyas77/Pattern-Definition-System/tree/main/Pattern-Editor/Pattern

relationshipName, and two flags that describe how the relation should be matched. The isList flag indicates whether the relation refers to a collection, while the isOrdered flag indicates whether the elements in that collection must be matched in the order specified by the pattern. This allows PDL to represent both unordered and ordered structural constraints directly at the specification level.

Constraints inside a relation are organized using FeatureGroup. A feature group collects the local conditions that must hold for the model element reached through a relation. These conditions are represented through the abstract Feature hierarchy. A feature can be a nested Relation, an Attribute, or an Enumeration. Nested relations allow the pattern to continue through deeper levels of the model hierarchy.

An Attribute represents constraints over primitive-valued properties of the current model element. It contains a type, an attributeName, and may optionally constrain its value. The type is restricted to primitive types defined by the PrimitiveType enumeration. The attributeName identifies the attribute in the target metamodel, while the optional value attribute specifies the required literal value when the pattern needs more than a presence check.

An Enumeration is used for attributes whose values are defined by an enumeration in the source metamodel, such as DevOps keywords in the pipeline case. It contains a type field for the enumeration type, an enumName field for the enumeration-valued attribute, and a value field for the required enumeration literal. This distinction allows PDL to represent both primitive attribute constraints and enumeration-based constraints based on the target metamodel structure.

For the DevOps pipeline case, this structure allows, for example, a pattern to begin at *Pipeline*, navigate to *Job* through the *jobs* relation (line 3), check job attributes such as *name* and *dependKW* (lines 5 and 6), navigate to *ParameterValue* through *jobParameterValues* (line 7), and check enumeration values such as *runs-on:*, *name:*, and *if:* (lines 8-10) as shown in Listing 2. An asterisk (*, line 7) indicates that the *jobParameterValues* relation is a collection. Since the *jobParameterValues* relation is marked as *Ordered* (line 7), the generated validator enforces the declared order of those parameter values. In this way, the PDL metamodel captures the elements needed to express structural patterns over EMF-based languages while keeping the pattern specification separate from the generated validation logic.

Note that the pattern example shown in Listing 2 is more concise than its OCL counterpart shown in Listing 1 due to *isOrdered* being a first-class concept in PDL.

IV. Generation of Java Validators

The proposed framework uses a model-to-text transformation to convert a declarative PDL pattern specification into executable Java code, i.e., a Java validator specialized for the declared structural pattern. The purpose of this transformation is to bridge the gap between high-level pattern specification and executable model-level validation.

```
Project_Path: "ca.mcgill.devops.pipeline"                              1
Pipeline {                                                             2
    (Job)jobs* {                                                       3
        {                                                              4
            (String)name,                                              5
            (String)dependKW,                                          6
            (ParameterValue)jobParameterValues* Ordered {              7
                { (PipelineKeyword)name: "runs-on:" },                 8
                { (PipelineKeyword)name: "name:" },                    9
                { (PipelineKeyword)name: "if:" }                       10
            },                                                         11
            (Step)steps* { }                                           12
        }                                                              13
    }                                                                  14
}                                                                      15
```

Listing 2: PDL Model for DevOps Pipeline Pattern Example

Figure 2 illustrates the overall transformation workflow[2] using a representative PDL pattern and selected fragments of the generated Java validator. From the input PDL model, Acceleo produces a Java validator with a stable implementation structure. This structure is reused across different patterns, while the generated matching logic changes according to the content of the input PDL model.

The generated validator contains the main components needed for dataset-level execution. It includes the required Java, EMF, and Xtext imports, the validation entry point, root-specific matching methods, relation matching methods, and the resource loading logic. The projectPath declaration in the PDL model is used to generate the package and import statements needed to connect the validator to the target language infrastructure, as shown in Fig. 2(1). This allows the validator to load input files as EMF model instances through the corresponding Xtext runtime and then evaluate them against the generated matching logic.

The transformation maps each PDL construct to corresponding Java validation code. A root class in PDL is translated into a root-level matcher that checks whether the loaded model has the expected root type, as shown in Fig. 2(2). Relations are translated into navigation logic over the EMF object graph. For example, the top-level relation (Job)jobs* in the PDL pattern is transformed into a generated relation matcher that retrieves the jobs relation and iterates over the related Job elements, as illustrated in Fig. 2(3).

Attribute constraints are translated into checks over primitive-valued properties, such as name and dependKW, as shown in Fig. 2(4).

Nested relations are handled through generated matching methods that are called from the enclosing relation matcher. This allows the validator to evaluate patterns that span multiple levels of the target metamodel. A key part of the transformation is its treatment of ordered relations. When a relation in the PDL model is marked as ordered, the generated validator not only checks whether the required elements are present. It also checks whether they occur in the sequence declared in the pattern. In Fig. 2(5), the relation (ParameterValue)jobParameterValues* Ordered is transformed into generated code that matches the required enumeration values in sequence. This order-

[2]https://github.com/AnkitaVyas77/Pattern-Definition-System/tree/main/Generic_M2T

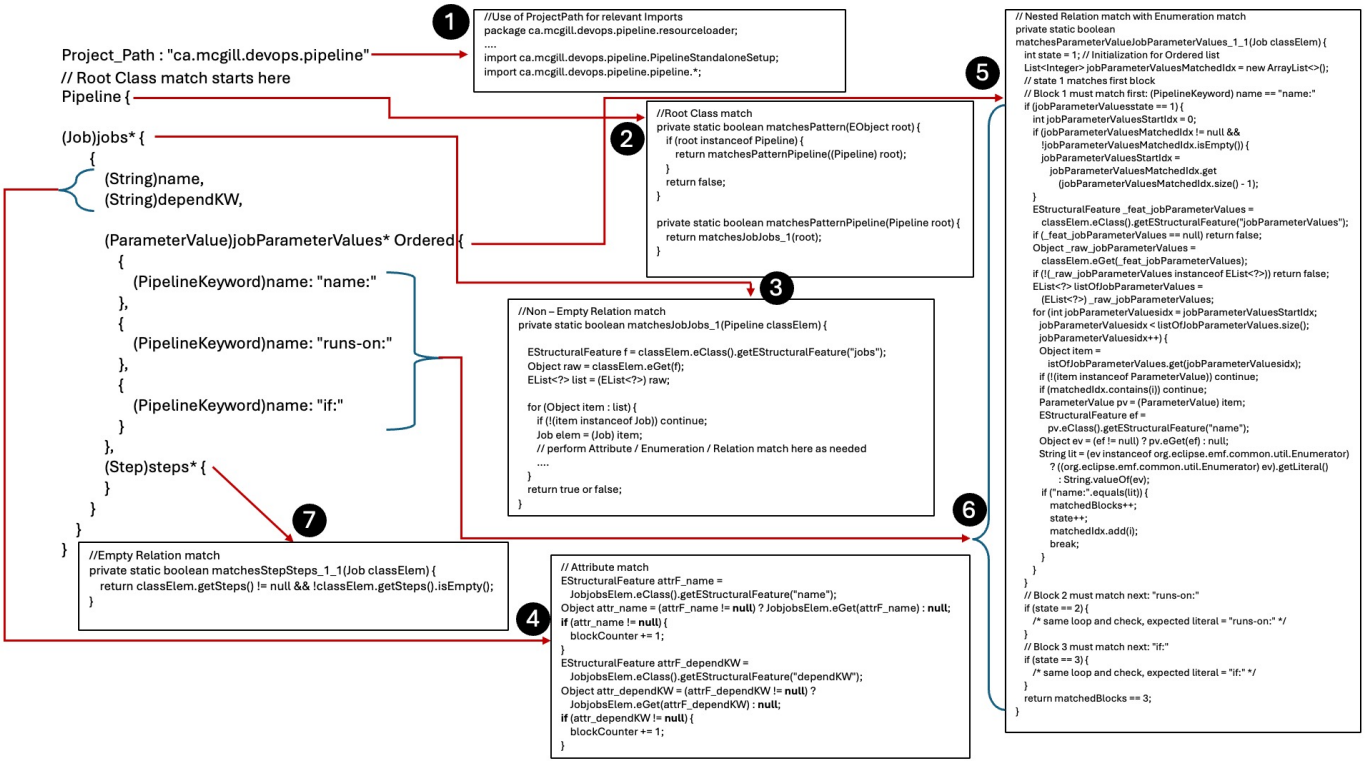

Fig. 2: Overview of the PDL-to-Java Validator Generation Workflow

sensitive behavior is generated automatically, so the user does not need to manually encode positional logic in OCL or handwritten Java code. As a result, ordered and unordered structural constraints are both supported through the same transformation workflow.

Enumeration constraints are translated into comparisons against the corresponding enumeration literals, as illustrated by the nested `ParameterValue` matcher in Fig. 2(6). Relations with empty bodies are treated as presence-only constraints. For example, the `(Step)steps*` relation is transformed into a non-empty relation check, as shown in Fig. 2(7).

For each input file in the dataset, the validator applies the generated pattern matching logic and classifies the file as matching or non-matching. The validator reports the filenames, number and percentage of files that satisfy the pattern, helping language engineers prioritize patterns for later inspection or possible language-improvement decisions.

This transformation-based design separates declarative pattern specification from validation implementation by automatically generating executable validators from PDL models. This improves consistency across validators while supporting dataset-scale validation over metamodel-based languages.

V. Evaluation

Beyond thoroughly testing the generated Java validators, we assess whether the proposed PDL-based validation workflow can handle realistic patterns found by two complementary pattern-discovery settings: a data mining-based experiment and an LLM-based experiment.[3] In both settings, candidate patterns are reconstructed in PDL and validated through the same generated Java validator workflow. The evaluation reports occurrence metrics, i.e., files evaluated, matches, and match percentage, and filename-level agreement between the generated Java validator and the corresponding reference.

Figure 3 shows the combined evaluation pipeline. The bottom branch represents the data mining-based workflow, where structural features are extracted from the validated dataset and mined using FP-Growth. The top branch represents the LLM-based workflow, where selected dataset chunks are supplied to an LLM to identify recurring YAML-level structures. The middle branch represents the shared PDL-based validation workflow. In this branch, patterns discovered by either method are reconstructed as PDL models, transformed into executable Java validators using Acceleo, and executed in the target metamodel editor environment.

The right side of Fig. 3 shows the common comparison stage. For each pattern found through the data mining-based or LLM-based experiment, the filenames produced by the corresponding Python-based workflow are compared with the filenames produced by the generated Java validator in a Python-based filename comparison script. This comparison assesses whether the reconstructed PDL specification and generated validator produce filename-level match results consistent with the corresponding reference mechanism at the model level.

[3]The complete evaluation as well as all results can be found at https://github.com/AnkitaVyas77/Pattern-Definition-System

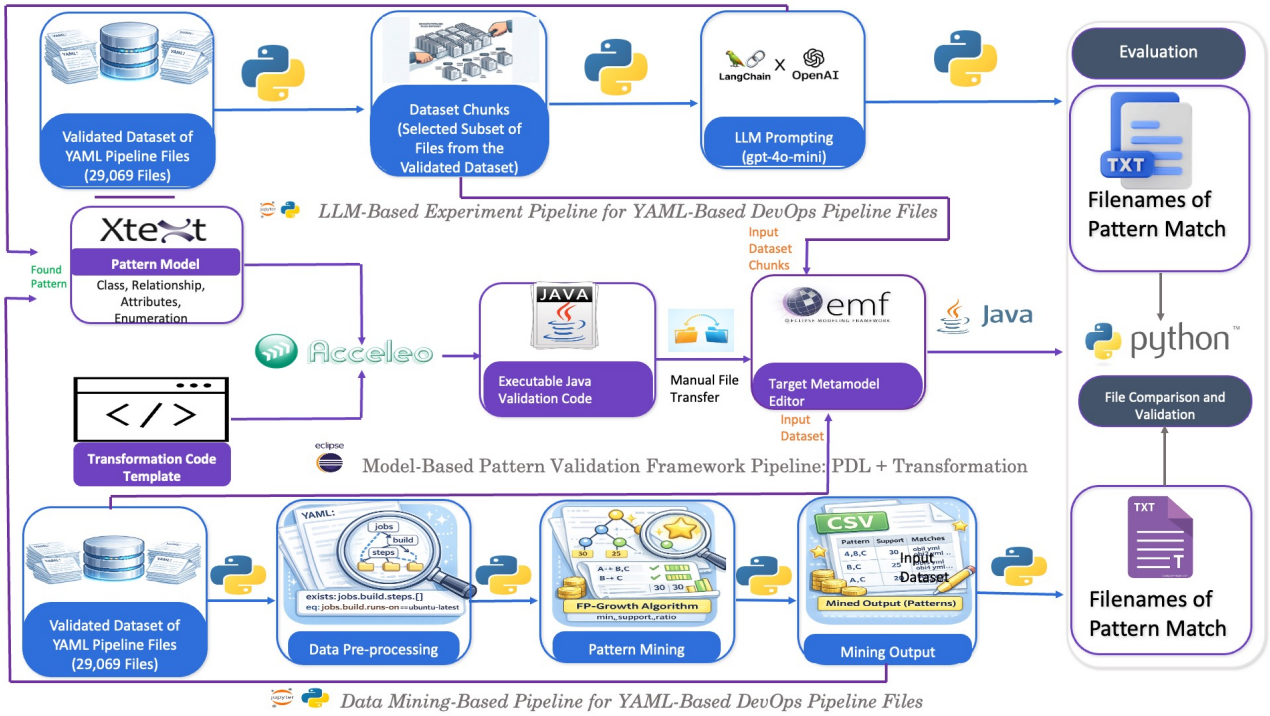

Fig. 3: Evaluation Pipeline for Data Mining-Based and LLM-Based Pattern Discovery with Shared PDL-Based Validation

A. Dataset and Experimental Setup

The evaluation uses a DevOps pipeline dataset collected from open-source projects. The original corpus contains 42,280 pipeline files [3], but only 29,069 files are valid and can be successfully parsed with the existing Xtext-based Pipeline editor. The editor converts the YAML-like DevOps files into Ecore/EMF model instances, on which the PDL-based validation workflow operates. The data mining-based setting uses the full validated dataset, while the LLM-based setting uses selected subsets of the same validated dataset. Accordingly, the reported match percentages are computed with respect to the number of files evaluated for each pattern.

B. Data Mining-Based Validation Workflow

The bottom branch of Fig. 3 shows the statistical pattern mining workflow. During a Python pre-processing step, each YAML pipeline file is traversed using a depth-first extraction procedure that preserves the hierarchy of the file as path-based features. For every visited path, the extractor emits an existence feature of the form `exists:path`. When the traversal reaches a scalar value, it also emits an equality feature of the form `eq:path==value`. Lists are represented using the wildcard segment `[]`, which captures list membership without depending on a specific index.

The extracted feature sets are then used as transactions for FP-Growth [28]. In this representation, each pipeline file corresponds to one transaction, and each extracted feature corresponds to one item in that transaction. FP-Growth is used to identify recurring combinations of structural features according to a selected minimum support threshold. For each selected mined pattern, a direct verification step rescans all transactions to confirm the support value and to produce the list of filenames containing the complete pattern. These filename-level outputs serve as the Python-based reference results for the mined patterns.

Each selected mined pattern is then manually reconstructed as a PDL specification using the Pipeline language. As shown in the middle branch of Fig. 3, the PDL model is processed by the Acceleo model-to-text transformation to generate an executable Java validator. The reconstruction process follows the structure and semantics of the Pipeline metamodel to ensure that the resulting PDL specifications reflect the corresponding EMF representation. The generated validator is executed in the target Pipeline editor environment over the same dataset. For each file, the validator loads the corresponding EMF model instance, checks the generated pattern logic, and classifies the file as matching or non-matching.

The filename list produced by the mining workflow serves as the Python-based reference output for the mined patterns. This output is later compared with the filename list produced by the generated Java validator in the common comparison stage shown on the right side of Fig. 3.

C. LLM-Based Validation Workflow

The LLM-based workflow is the second realistic source of candidate patterns and tests whether the same PDL-to-Java validation workflow can handle structures discovered outside the statistical mining process. The top branch of Fig. 3 shows the LLM-based pattern discovery workflow. Due to context window limitations, the whole dataset is not provided

to the LLM. Instead, selected subsets of pipeline files are grouped into dataset chunks and supplied directly to GPT-4o-mini [29] through prompt-based interaction.[4] The chunks are used to obtain structurally diverse candidate patterns under the model context limit, not to estimate full-corpus pattern frequencies. Therefore, LLM-based match percentages are interpreted only with respect to the evaluated chunk for each pattern and should not be directly compared with mining-based percentages computed over the full validated dataset. Unlike the statistical mining workflow, this process does not operate over extracted feature transactions. Instead, the LLM identifies recurring structural patterns directly from the raw YAML-based pipeline specifications.

The resulting candidate patterns are manually reconstructed in PDL using the Pipeline metamodel and transformed into executable Java validators through the same Acceleo-based transformation workflow used for the mined patterns. The generated validators are then executed over the same dataset chunks inside the Pipeline editor environment, where each file is loaded as an EMF model instance and evaluated against the reconstructed PDL pattern.

To provide an independent reference, a Python-based structural matcher is implemented for each LLM-found pattern over the same dataset chunk. These Python matchers operate directly over the parsed YAML representation, while the generated Java validators operate over the EMF model representation produced by the Pipeline editor. The filename lists produced by each Python matcher and the corresponding generated Java validator are then compared in the common comparison stage shown on the right side of Fig. 3.

D. Evaluated Patterns

Patterns P1 to P5 in Table I are selected from the frequent pattern mining outputs. Pattern P6 is an ordered reference pattern derived from Pattern P4 and is included to evaluate an additional case of order-sensitive matching. Patterns P7 to P11 are identified through the LLM-based discovery workflow. The patterns are selected to form a realistic and diverse set of patterns. Together, these cases cover trigger-based patterns (P1, P3, P4, P6–P11), job-level patterns (P2, P5, P7–P11), value-constrained patterns (P5), nested step structures (P2, P7–P11), and container declarations (P11).

Patterns P6, P9, and P10 represent ordered structural constraints. By comparing the Java validator results for these patterns with an independent Python reference matcher, the evaluation assesses whether the generated validators correctly implement the `Ordered` semantics of PDL. This is important because one of the main goals of PDL is to support both unordered and sequence-sensitive structural pattern validation without requiring the user to manually encode positional logic.

For each evaluated pattern, the experiment records the mined feature set or LLM-found structure, the corresponding PDL reconstruction, the number of evaluated files, the occurrence metrics produced by the generated Java validator, and the filename-level comparison with the corresponding Python-based reference output. This allows the evaluation to assess two aspects of the proposed framework. First, it examines whether realistic structural patterns discovered through different mechanisms can be expressed in PDL using the target metamodel. Second, it examines whether the generated validators detect the correct occurrences reliably when executed over the model-based representation of the dataset.

[4]https://github.com/AnkitaVyas77/Pattern-Definition-System/tree/main/LLM_Experiment/Prompts_used_for_5_LLM_patterns.txt

VI. Results and Discussion

Table I summarizes the validation results for the data mining-based patterns, the additional ordered reference pattern, and the LLM-found patterns. For each pattern, the table reports the number of matching files produced by the generated Java validator and the agreement with the corresponding reference output. For Patterns P1 to P5, the reference output is obtained from the statistical mining workflow. For Patterns P6 to P11, the reference output is obtained from manually implemented Python-based structural matchers. This comparison evaluates whether the PDL specification and the generated validator produce filename-level match results consistent with the corresponding reference mechanism when the pattern is interpreted over EMF model instances.

The results show that the generated validators produce filename-level results consistent with the corresponding reference for most evaluated cases. Patterns P2 to P5 show exact filename-level agreement between the mining-based and generated Java validator outputs. This indicates that, when the mined hierarchical features map clearly to the Pipeline metamodel, the PDL-to-Java transformation produces validators that detect the same occurrences at the model level. Patterns P3 and P4 are baseline examples of this behavior because their trigger structures align directly with the corresponding metamodel representation. Pattern P5 further shows that the generated validators preserve not only structural presence but also value constraints, since the pattern includes the requirement that the *runs-on:* value of the build job is *ubuntu-latest*.

Pattern P2 illustrates an important practical issue in mapping mined YAML features to model-level structures. The mined pattern requires both *uses* and *run* under the *steps* block, but the Pipeline metamodel can represent these elements through different concrete structures. To account for this, the pattern was reconstructed using four PDL variants, and the outputs of the corresponding generated validators were merged. After this reconstruction, the model-based output matched the mining-based output exactly, with 6,132 matching files. This result shows that PDL can support such cases, but it also shows that the pattern author, i.e., the person who reconstructs the discovered pattern as a PDL specification, must understand how the target metamodel represents the structures found in the raw dataset.

Pattern P1 is the only evaluated mined pattern that does not produce exact agreement. The mining workflow reports 6,791 files, while the Java validator reports 6,790 files, with 6,788 files common to both outputs. The five mismatches are explainable rather than random. Three files appear only in the

TABLE I: Occurrence and Filename-Level Agreement Metrics for Data Mining-Based and LLM-Found Patterns

Pattern	Method	Ordering in PDL	Main Structure	Files Evaluated	Matches	Match %	Agreement
P1	Data Mining-Based	Unordered	exists:name & exists:on.push & exists:on.push.branches	29,069	6,790	23.36%	6,788 common; 3 mining-only; 2 Java-only
P2	Data Mining-Based	Unordered	exists:jobs.build.steps.[].run & exists:jobs.build.steps.[].uses	29,069	6,132	21.09%	Exact after merging four PDL variants
P3	Data Mining-Based	Unordered	exists:on.workflow_call & exists:on.workflow_call.inputs	29,069	7,023	24.15%	Exact
P4	Data Mining-Based	Unordered	exists:name & exists:on.pull_request & exists:on.push	29,069	4,255	14.63%	Exact
P5	Data Mining-Based	Unordered	eq:jobs.build.runs-on == ubuntu-latest	29,069	3,015	10.37%	Exact
P6	Ordered version of P4	Ordered	exists:on.push followed by exists:on.pull_request	29,069	3,532	12.15%	Exact with Python reference matcher
P7	LLM-based	Unordered	name: "value" on: "value" jobs: job_name: runs-on: "value" steps: - name: "value" run: "value"	1,057	462	43.71%	462 common; 5 Python reference matcher-only
P8	LLM-based	Unordered	name: "value" on: "value" jobs: "value": runs-on: "value" steps: - name: "value" run: "value" - uses: "value"	806	227	28.16%	Exact with Python reference matcher
P9	LLM-based	Ordered	name: on: push: branches: jobs: <job_name>: runs-on: steps: - name: uses: - name: run:	519	91	17.53%	Exact with Python reference matcher
P10	LLM-based	Ordered	name: on: push: branches: pull_request: jobs: build: runs-on: steps: - uses: - name: run: - name: run:	812	26	3.20%	Exact with Python reference matcher
P11	LLM-based	Unordered	name: <workflow_name> on: <trigger_event> jobs: <job_name>: runs-on: <runner> container: image: <container_image> steps: - name: <step_name> run: <command>	1,056	15	1.42%	Exact with Python reference matcher

mining output because the raw YAML file contains a branches key with an empty branch list. The feature extractor emits *exists:on.push.branches* when the key is present, whereas the model-level validator requires an actual *includedBranches* relation containing valid Branch elements. Hence, this is an issue with the feature extractor. Two files appear only in the Java validator output because of a known parsing behavior in the Pipeline DSL, where an invalid shorthand form can still populate the corresponding branch structure in the EMF model. Hence, this is an issue with the Xtext parser for pipeline models. These cases show that filename-level comparison can reveal meaningful differences between raw YAML feature extraction and metamodel-level interpretation.

Pattern P6 evaluates the ordered matching capability of the generated validators. It is an ordered variant of Pattern P4 and requires the *push* trigger to appear before the *pull_request* trigger. The generated Java validator reports 3,532 matching files, and the independent Python reference matcher reports the same filename set. Compared with Pattern P4, which matches 4,255 files, the ordered pattern matches fewer files because it accepts only the subset where the triggers appear in the specified order. This confirms that the Ordered construct in PDL is reflected correctly in the generated validation logic and that order-sensitive constraints can significantly affect occurrence metrics.

The LLM-found patterns (P7–P11) further demonstrate that the proposed framework can validate candidate structural patterns obtained through LLM-assisted discovery, including ordered patterns. Patterns P8 to P11 show exact filename-level agreement between the generated Java validators and the reference implementation. These patterns include combinations of trigger declarations, ordered step structures, container image declarations, and step-level *uses* and *run* associations. The exact agreement indicates that the reconstructed PDL specifications capture the intended structure consistently.

Pattern P7 is the only LLM-found pattern that does not produce exact agreement. The generated Java validator reports 462 matching files, while the Python matcher reports 467 files, with five files appearing only in the Python-based output. These mismatches are explainable through the same distinction between raw YAML-level matching and metamodel-level validation observed earlier for Pattern P1. In these files, the Python matcher accepts the textual presence of the *on* key even when the trigger block is empty. In contrast, the generated Java validator evaluates the corresponding *triggers* relation in the EMF model and requires at least one valid trigger object to exist. Therefore, files containing only an empty *on:* declaration are counted by the Python matcher but rejected by the model-based validator. Some of these files also contain quoted shell commands with escaped quotation marks inside the *run:* field, which may further affect how the Pipeline DSL materializes the corresponding *Script* structure in the EMF model. These results again demonstrate that filename-level comparison can reveal meaningful differences between raw YAML structural matching and metamodel-level interpretation.

Overall, the results support the usefulness of PDL and the model-to-text transformation workflow for structural pattern validation over metamodel-based datasets. The exact agreements show that generated validators can reproduce both mined patterns and LLM-found pattern occurrences when the PDL reconstruction accurately reflects the target metamodel. The mismatch cases in Patterns P1 and P7 show that differences are traceable and can reveal meaningful distinctions between textual feature extraction and model-based interpretation. The ordered reference patterns further demonstrate that the framework can handle sequence-sensitive structural constraints without requiring manually written positional logic.

Together, these results indicate that the proposed approach provides a systematic basis for specifying realistic structural patterns in PDL and validating their occurrences through generated Java validators, independent of whether the candidate patterns originate from statistical mining or LLM-assisted discovery. In this evaluation, the same author performed the full workflow, including reconstructing mined and LLM-found candidate structures in PDL, executing the generated validators, and comparing their outputs with the corresponding reference results. Therefore, the results should be interpreted as an implementation-level consistency evaluation of the PDL workflow rather than as an independent user study of pattern reconstruction.

VII. Conclusion

This paper presents the Pattern Definition Language (PDL), a domain-specific language for specifying structural patterns over EMF-based metamodels, together with a model-to-text transformation workflow that generates executable Java validators from PDL specifications. The work is situated in the broader context of self-adaptable languages, where evidence about recurring language use can support feedback loops for future language improvement. In this context, PDL contributes to the pattern specification and validation stages of such a feedback loop by enabling (i) requirements for language evolution to be specified and, more specifically, (ii) structural patterns to be declared explicitly and checked systematically over model-based artifacts.

The evaluation uses a dataset of 29,069 DevOps pipeline files and considers two complementary realistic validation settings. The first setting compares generated validator outputs with statistically mined pattern occurrences at the filename level. The second setting evaluates patterns identified through LLM-assisted discovery by comparing generated validator outputs with independent Python-based structural matchers over the same dataset subsets. Across both settings, the results show exact agreement for most evaluated patterns, indicating that the generated validators produce filename-level match results consistent with the corresponding reference mechanism when the PDL reconstruction aligns with the target metamodel. The mismatch cases are explainable through differences between raw YAML-level structural matching / Python-based parsing and metamodel-level interpretation, showing that the workflow can also reveal representation-level discrepancies. The ordered reference patterns further show that PDL's ordered relation

construct is correctly reflected in the generated validation logic and that sequence-sensitive constraints can significantly affect occurrence metrics.

The evaluation does not measure human effort or usability. Instead, it demonstrates that the evaluated structural patterns, including nested, repeated, and sequence-sensitive constraints, can be specified declaratively in PDL and operationalized through generated Java validators. In this workflow, the pattern author specifies the required structure at the metamodel level, while traversal, ordering checks, and occurrence reporting are generated automatically by the Acceleo-based transformation.

Overall, the results indicate that PDL and its transformation workflow provide a systematic basis for structural pattern specification and automated validation in metamodel-based languages. By producing reusable validators from declarative pattern specifications, the approach supports evidence collection over large model datasets and provides a foundation for future integration into self-adaptable language workflows. The evaluation further shows that the framework is not limited to a single pattern discovery mechanism, since both statistically mined patterns and LLM-found structural patterns can be reconstructed in PDL and validated systematically through the same model-based validation workflow. Future work includes reducing the manual effort required to reconstruct discovered patterns in PDL given a partial model, evaluating the framework on additional metamodel-based languages, exploring tighter integration between LLM-assisted discovery and model-based validation, and connecting validated structural patterns to subsequent language improvement and model co-evolution steps for self-adaptable languages.

REFERENCES

[1] G. Jouneaux, B. Combemale, O. Barais, and G. Mussbacher, "Towards self-adaptable languages," in *Proceedings of the ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2021)*. Chicago, Illinois, USA: ACM, Oct. 2021, pp. 97–113, accessed on 2026/04/10. [Online]. Available: https://doi.org/10.1145/3486607.3486753

[2] Object Management Group, "OCL," accessed on 2025/04/04. [Online]. Available: https://www.omg.org/spec/OCL/2.4/PDF

[3] M. M. Bedekar and G. Mussbacher, "A multi-platform specification language and dataset for the analysis of DevOps pipelines," in *MODELS Companion '24: ACM/IEEE 27th Intl. Conf. on Model Driven Engineering Languages and Systems Companion*. ACM, 2024, pp. 264–274. [Online]. Available: https://doi.org/10.1145/3652620.3686247

[4] J. Whittle, J. R. Hutchinson, and M. Rouncefield, "The state of practice in model-driven engineering," *IEEE Software*, vol. 31, no. 3, pp. 79–85, 2014. [Online]. Available: https://doi.org/10.1109/MS.2013.65

[5] Eclipse Foundation, "EMF," accessed on 2025/04/04. [Online]. Available: https://eclipse.dev/modeling/emf/

[6] ——, "Package org.eclipse.emf.ecore," accessed on 2025/04/04. [Online]. Available: https://download.eclipse.org/modeling/emf/emf/javadoc/2.10.0/org/eclipse/emf/ecore/package-summary.html#details

[7] M. Eysholdt and H. Behrens, "Xtext: Implement your language faster than the quick and dirty way," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*. ACM, 2010, pp. 307–309. [Online]. Available: https://doi.org/10.1145/1869542.1869625

[8] H. Krahn, B. Rumpe, and S. Völkel, "MontiCore: Modular development of textual domain specific languages," in *Objects, Components, Models and Patterns*, ser. Lecture Notes in Business Information Processing, vol. 11. Springer, 2008, pp. 297–315. [Online]. Available: https://doi.org/10.1007/978-3-540-69824-1_17

[9] Eclipse Foundation, "Xtext documentation," accessed on 2026/03/27. [Online]. Available: https://eclipse.dev/Xtext/documentation/

[10] L. Lúcio, M. Amrani, J. Dingel, L. Lambers, R. Salay, G. M. K. Selim, E. Syriani, and M. Wimmer, "Model transformation intents and their properties," *Software and Systems Modeling*, vol. 15, no. 3, pp. 647–684, 2016. [Online]. Available: https://doi.org/10.1007/s10270-014-0429-x

[11] N. Kahani, M. Bagherzadeh, J. R. Cordy, J. Dingel, and D. Varró, "Survey and classification of model transformation tools," *Software and Systems Modeling*, vol. 18, no. 4, pp. 2361–2397, 2019. [Online]. Available: https://doi.org/10.1007/s10270-018-0665-6

[12] E. Syriani, L. Luhunu, and H. Sahraoui, "Systematic mapping study of template-based code generation," *Computer Languages, Systems and Structures*, vol. 52, pp. 43–62, 2018. [Online]. Available: https://doi.org/10.1016/j.cl.2017.11.003

[13] Eclipse Foundation, "Acceleo query language documentation," accessed on 2026/03/27. [Online]. Available: https://eclipse.dev/acceleo/documentation/

[14] T. Yue and S. Ali, "Empirically evaluating OCL and java for specifying constraints on UML models," *Software and Systems Modeling*, vol. 15, no. 3, pp. 757–781, 2016. [Online]. Available: https://doi.org/10.1007/s10270-014-0438-9

[15] D. Stein, S. Hanenberg, and R. Unland, "A graphical notation to specify model queries for MDA transformations on UML models," in *Model Driven Architecture*, ser. Lecture Notes in Computer Science, vol. 3599. Springer, 2005, pp. 77–92. [Online]. Available: https://doi.org/10.1007/11538097_6

[16] H. Störrle, "VMQL: A visual language for ad-hoc model querying," *Journal of Visual Languages and Computing*, vol. 22, no. 1, pp. 3–29, 2011. [Online]. Available: https://doi.org/10.1016/j.jvlc.2010.11.004

[17] ——, "MOCQL: A declarative language for ad-hoc model querying," in *Modelling Foundations and Applications*, ser. Lecture Notes in Computer Science, vol. 7949. Springer, 2013, pp. 3–19. [Online]. Available: https://doi.org/10.1007/978-3-642-39013-5_2

[18] G. Bergmann, Z. Ujhelyi, I. Ráth, and D. Varró, "A graph query language for EMF models," in *Theory and Practice of Model Transformations*, ser. Lecture Notes in Computer Science, vol. 6707. Springer, 2011, pp. 167–182. [Online]. Available: https://doi.org/10.1007/978-3-642-21732-6_12

[19] D. Varró, G. Bergmann, Á. Hegedüs, Á. Horváth, I. Ráth, and Z. Ujhelyi, "Road to a reactive and incremental model transformation platform: Three generations of the VIATRA framework," *Software and Systems Modeling*, vol. 15, no. 3, pp. 609–629, 2016. [Online]. Available: https://doi.org/10.1007/s10270-016-0530-4

[20] Eclipse Foundation, "Epsilon," accessed on 2026/03/27. [Online]. Available: https://eclipse.dev/epsilon/

[21] ——, "Epsilon pattern language," accessed on 2026/03/27. [Online]. Available: https://eclipse.dev/epsilon/doc/epl/

[22] D. S. Kolovos and R. F. Paige, "The epsilon pattern language," in *2017 IEEE/ACM 9th International Workshop on Modelling in Software Engineering (MiSE)*. IEEE, 2017, pp. 54–60. [Online]. Available: https://doi.org/10.1109/MiSE.2017.8

[23] Eclipse Foundation, "Epsilon validation language," accessed on 2026/03/27. [Online]. Available: https://eclipse.dev/epsilon/doc/evl/

[24] H. da Gião, V. Amaral, G. Engels, A. Flores, R. Pereira, S. Sauer, and J. Cunha, "A metamodel for reengineering ci/cd pipelines," in *ACM/IEEE 28th MODELS Conference*. IEEE, 2025, pp. 221–231. [Online]. Available: https://doi.org/10.1109/MODELS67397.2025.00026

[25] S. G. Saroar and M. Nayebi, "Developers' perception of GitHub actions: A survey analysis," in *27th EASE Conference*. ACM, 2023, pp. 121–130. [Online]. Available: https://doi.org/10.1145/3593434.3593475

[26] P. Valenzuela-Toledo, A. Bergel, T. Kehrer, and O. Nierstrasz, "The hidden costs of automation: An empirical study on GitHub actions workflow maintenance," in *2024 IEEE Intl. Conf. on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2024, pp. 213–223. [Online]. Available: https://doi.org/10.1109/SCAM63643.2024.00029

[27] A. Mastropaolo, F. Zampetti, G. Bavota, and M. Di Penta, "Toward automatically completing GitHub workflows," in *IEEE/ACM 46th International Conference on Software Engineering*. ACM, 2024, pp. 1–12. [Online]. Available: https://doi.org/10.1145/3597503.3623351

[28] S. Raschka, "fpgrowth: Frequent itemsets via the FP-growth algorithm," 2026, accessed on 2026/03/27. [Online]. Available: https://rasbt.github.io/mlxtend/user_guide/frequent_patterns/fpgrowth/

[29] OpenAI, "GPT-4o-mini," 2026, accessed on 2026/05/22. [Online]. Available: https://developers.openai.com/api/docs/models/gpt-4o-mini