JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

A Model-Driven Approach To Support The Understanding Of Machine Learning Pipelines

Nicolas Lacroix[*], **Mireille Blay-Fornarino**[†], **Philippe Collet**[†], **Frédéric Precioso**[*], **and Sébastien Mosser**[§]
[*]Université Côte d'Azur, Inria, CNRS, I3S, France
[†]Université Côte d'Azur, CNRS, I3S, France
[§]McMaster Centre for Software Certification, McMaster University, Canada

ABSTRACT Artificial Intelligence in general and Machine Learning in particular is a very dynamic field, with evolving technologies and practices. Key decisions, from dataset preparation methods to model architecture and evaluation metrics, are made in an exploratory way, diverging from standard software engineering practices and guided by the data scientist's empirical knowledge and experience. In this context, (i) some practices work, while others do not; (ii) some unexpected approaches perform better than the regular ones; (iii) some anti-patterns are accidentally used, leading to model contamination and biases. Developers and data scientists have to work on code artifacts (*e.g.*, Jupyter notebooks) to identify key differences across multiple variations of the same machine learning pipelines, which is confusing and error-prone because it is only syntactic. In this paper, we defend a model-driven approach that reifies semantic information about machine learning pipelines to improve their understanding. Based on this metamodel, which captures essential steps in a given pipeline and links them to code artifacts, we define a pattern-matching language that supports data scientists in exploring corpora of machine learning artifacts. We validate the approach by identifying real-world use cases in collaboration with data scientists and applying them to the qualitative analysis of 105 Kaggle notebooks (a popular competition platform where participants submit pipelines to solve similar tasks). This work opens the door to transferring program understanding techniques to machine learning while accounting for its intrinsic exploratory nature. By relying on explicit models and a dedicated pattern language, we provide a foundation that supports systematic analysis of ML pipelines—such as pipeline comparison, practices identification, and anti-pattern detection—while remaining robust to the evolution of libraries, frameworks, and implementation technologies.

KEYWORDS Model-Driven Engineering, Machine Learning Pipelines, Domain-Specific Language, Pattern Matching.

1. Introduction

Building Machine Learning (ML) models is by nature an exploratory, collaborative, and iterative process (Zhang et al. 2020). Often relying on clone-and-own, data scientists reuse ML pipelines from past experiments and adapt them to the specific context of their analysis (Jebnoun et al. 2022; Brault et al. 2023). Yet, the resulting divergence from the theoretical ML life cycle (Martínez-Plumed et al. 2019) further complicates their understanding in a context where ML expands across various domains (Jordan & Mitchell 2015) and its societal impact grows.

Understanding how ML models are constructed is therefore critical to support ML practitioners, but this task is challenged by the high variability of the domain, ranging from the organization of high-level steps of the ML process (*e.g.*, data preparation, training, and evaluation) to library choices and concrete algorithm implementations. In this context, Jupyter notebooks have become the *de facto* standard for ML development, combining code, documentation, and visualization in a single interactive

JOT reference format:
Nicolas Lacroix, Mireille Blay-Fornarino, Philippe Collet, Frédéric Precioso, and Sébastien Mosser. *A Model-Driven Approach To Support The Understanding Of Machine Learning Pipelines*. Journal of Object Technology. Vol. 25, No. 3, 2026.
http://dx.doi.org/10.5381/jot.2026.25.3.a10

document (Granger & Pérez 2021). They represent valuable knowledge repositories, widely used in Open Science for sharing and reusing ML pipelines (Randles et al. 2017).

However, navigating many notebooks remains a challenge, as finding relevant ones requires manually inspecting each one. Moreover, numerous studies on notebook coding practices have highlighted poor practices and maintenance difficulties (Huang et al. 2025; Pimentel et al. 2019), emphasizing the need for better abstractions to enhance understanding and exploration.

In this paper, we make the following contributions:

- **A model-driven approach for analyzing and comparing sets of ML pipelines.** We propose a descriptive metamodel that structures heterogeneous information extracted from ML code into a uniform and extensible representation. Building on top of this metamodel, we introduce a dedicated query language—based on regular-expression–like constructs—designed for data scientists to express semantic queries over large pipeline corpora. Together, the metamodel and the language enable the detection of recurrent structures, meaningful variations, and deviations from expected practices.
- **An empirical case study demonstrating the expressiveness of the approach.** We qualitatively evaluate our model-driven querying framework on a real-world ML pipeline corpus drawn from an existing empirical study. Specifically, we reproduce findings and observations produced by 3 independent authors (Biswas et al. 2022) to reduce the risk of biased analysis and base our results on the current understanding of data scientists' practices. Through three motivating use cases, this qualitative evaluation shows how our approach can help data scientists filter pipelines based on their structures, validate practices compliance, and characterize emerging or anti patterns.

Section 2 motivates our work by detailing the current landscape of ML development and introducing a concrete scenario that illustrates the challenges of understanding and comparing ML pipelines. Section 3 reviews related work. Section 4 presents our overall approach, which relies on a descriptive pipeline model and a dedicated pattern language; these two core components are detailed in Section 5. Section 6 evaluates our approach by replicating the findings reported in the reference study, and shows how our queries enable the identification of good practices, deviations, and bad practices. Finally, Section 7 concludes the paper.

2. Motivation

2.1. The life of a Data Scientist

The ML landscape is characterized by its exploratory nature (Herrmann et al. 2024) and rapid democratization, leading to vast amounts of code available on platforms such as Kaggle and Google Colab, often produced by experts of specific domains (mathematics, medicine, biology, physics, etc) rather than trained software engineers.

Designing ML pipelines involves decisions made at multiple levels of abstraction, from the choice of libraries and algorithms to the ordering of processing steps. Common pain points include inconsistent data preprocessing techniques, which may introduce variations in model performance (Tschalzev et al. 2024). As a result, pipelines spanning data preparation, feature engineering, model training, evaluation, and deployment exhibit substantial—and often undocumented—variability in their structure, components, and execution flow. This variability in construction is accentuated by the fact that the same algorithm can serve different objectives within the pipeline (*e.g.*, `k`-means clustering may be used either as a dimensionality-reduction preprocessing step or as the final unsupervised decision model). Conversely, different pipelines may achieve comparable objectives through highly diverse processes. As summarized by the CACE principle (Sculley et al. 2015)—"*Changing Anything Changes Everything*"—even minor structural variations (*e.g.*, parameter choices or the ordering of ML operations) may have significant ripple effects.

Furthermore, managing the quality not only of the ML models produced, but also of how they are produced, requires working across the entire pipeline. Thus, crucial requirements, such as ML model's fairness assessment, require analyzing several interdependent steps in the pipeline while explicitly identifying the presence and configuration of specific algorithms and libraries. Similarly, to ensure organizational compliance (e.g., verifying that all pipelines implement mandatory data anonymization procedures), it is necessary to search the pipeline topology for characteristic structural patterns and associated library calls. The inherent variability of pipelines complicates systematic reasoning about these practices, particularly for non-functional properties such as fairness, code quality (Sculley et al. 2015), and performance (De Martino & Palomba 2025).

Additionally, the relatively young field of data science has not yet incorporated the lessons learned from software engineering. In practice, ML pipelines are frequently constructed through rapid prototyping and extensive "clone-and-own", a reuse strategy well documented in software engineering (Dubinsky et al. 2013) and now pervasive in ML development. Reusing code by cloning is convenient for fast experimentation—as Koenzen et al. report, "*reusing code from online sources proved to be the preferred method of reuse for our participants*" (Koenzen et al. 2020). However, this practice also introduces well-known issues of code quality and maintainability (Pimentel et al. 2019; Jebnoun et al. 2022).

Our motivating scenario is the result of discussions with experts from the ML domain. It is composed of three use cases: UC_1, UC_2 and UC_3. Each use case will be used to properly evaluate our solution in section 6. The motivating scenario involves Alice, an expert data scientist holding a PhD in Data Science. Over the past 20 years, Alice has studied and developed many ML pipelines using the "clone-and-own" method.

2.2. UC_1: Structural Filtering

Whether for searching for reusable pipeline fragments or analyzing Kaggle competitions, data scientists need to filter notebooks based on internal structural properties rather than just keywords. For instance, Alice may need to identify notebooks that strictly adhere to specific constraints, based on their particular context.

Considering the rapid growth of notebooks available online (an increase of +75% compared to 2024 to reach 2.4 million GitHub repositories using Notebooks in 2025 according to the Octoverse 2025 report[1]) and the fact that "*median notebook had 85 lines of code*" (Rule et al. 2018), we can approximate the number of scientific notebooks' lines of code hosted on GitHub to $85 \times 2,400,000 = 204,000,000$ LoC. Without an automated approach and proper tooling, Alice has no realistic way to identify ML pipeline structures from a set of notebooks, even at a lower scale involving thousands of lines of code. This exploration is necessary for data scientists to identify prototypical uses of algorithms in existing code, identify new practices, and, overall, support a better understanding of the field.

As of today, data scientists' exploration capacity is limited to (i) looking at code she already knows (emphasizing cargo cult), (ii) using keyword-based search on open-source platforms, or (iii) going through an extensive body of literature considering how fast papers are published (and released on arXiv) in the AI community. In the following, we consider that Alice, our persona, is interested in understanding whether *Graph Neural Networks (GNNs)* would be useful for the pipeline she is developing on her own, which targets *time-series forecasting*. As algorithms' performance is context-dependent, she needs to evaluate how *GNN* was used in other pipelines and in what context, to determine whether her own context *time-series forecasting* would be compatible with the classical application of *GNN*.

2.3. UC_2: Pipeline Compliance Checking

ML tutorials often present high-level patterns (*e.g.*, data preparation, remediation, sampling) using abstract vocabularies as illustration. However, validating the presence of these patterns in real-world code is challenging because of the high variability in their concrete implementations. A single conceptual step, such as "sampling" can be implemented using dozens of different functions or library calls. Given how quickly the machine learning literature evolves (*e.g.*, from Large Language Models to Small Language Models and now Tiny Recursion Models or Diffusion Large Language Models in a couple of years), it is essential to know whether a given approach is actually used before integrating it into a product. Also, since Data Science coding practices are exploratory, when a given approach is selected for designing a pipeline, it is crucial to ensure that the resulting variation remains valid.

As of today, this use case requires manual code review, including reading code written by others and auditing one's own ML pipelines. While some reverse engineering tools attempt to map code (*e.g.*, function calls) to a step in the ML life cycle (given a specific taxonomy) (Ramasamy et al. 2023; Biswas et al. 2022), they still require manual analysis. For example, pipeline compliance involves both sequential (*e.g.*, *"Feature Engineering"* should be done before *"Training"*) and transversal (*e.g.*, *"Training"* should involve a specific ML library or algorithm based on requirements) constraints. Going back to Alice, she is interested in identifying fairness biases in her product. In other words, she wants to treat the different populations in her dataset equitably. SONY has produced an implementation of standard biases mitigation techniques as tutorials to illustrate the use of their `nnabla` library. Alice wants to know how such approaches are used in actual notebooks before deciding to clone one of them: even if SONY is pushing for these methods (supported by tools and dedicated libraries), are they really reused by other practitioners in the field? Also, as adopting a new library comes with technical constraints (*e.g.*, now all code would be `nnabla` dependent), is it possible to identify the same fairness approaches in codes using the technological stack Alice is already using?

2.4. UC_3: Iterative Analysis Refinement

Best practices in ML are rarely rigid; they are subject to context-dependent variations. To automate quality assurance, it is necessary to distinguish between an acceptable refinement of a pattern (a valid variation) and a "bad smell" or anti-pattern (an error). Variations may result from acceptable or problematic processing orders, or from the use of compatible or incompatible algorithms. Determining whether a variant is acceptable or constitutes bad practice is not an easy task: it requires gradually refining the analysis to incorporate several criteria. For instance, some algorithm implementations in specific libraries are robust to missing values (*e.g.*, LightGBM) while others require explicit treatment. Likewise, removing outliers before scaling may be acceptable, whereas scaling before outlier removal can distort distance-based methods such as the Local Outlier Factor, leading to incorrect anomaly identification. However, determining whether a code, often built by aggregating codes from several sources using the "clone-and-own" method, respects properties is essential to ensuring the quality of the models themselves.

Today, although several static analysis tools and linters, such as *Pylint*, provide basic checks for the correctness of ML code (*e.g.*, syntax, types, unused variables), they operate solely at the programmatic level and remain unaware of ML-specific semantics. Consequently, determining whether a pipeline variation is acceptable or constitutes an anti-pattern still relies almost entirely on the data scientist's expertise. In practice, addressing such questions relies on a combination of the data scientist's expertise, prior experience, and external knowledge sources such as algorithm documentation, tutorials, and scientific literature (Sculley et al. 2015; Amershi et al. 2019). In addition, practitioners frequently examine existing notebooks and code examples to understand how similar pipelines have been implemented in comparable contexts. This exploratory process typically involves inspecting the ordering of operations, identifying implicit assumptions made by original authors, and reasoning about algorithmic behavior across different pipelines.

Returning to Alice, she is exploring notebooks in her domain—time series processing in a medical context where fairness is crucial. She identifies a promising notebook that addresses similar biases in patient outcome prediction. However, the pipeline scales features before detecting outliers using the Isolation Forest algorithm. Alice wonders: *Is this a deliberate design choice (perhaps the author assumes Isolation Forest is robust to scaling), or is it a structural anti-pattern that could*

[1] https://github.blog/news-insights/octoverse/octoverse-a-new-developer-joins-github-every-second-as-ai-leads-typescript-to-1/

compromise her bias detection? Since her own pipeline uses the distance-based Local Outlier Factor, she needs to determine whether this variation is safe to adopt or represents a problematic practice that could distort her fairness analysis. Therefore, she begins by querying her curated pipeline collection to structurally identify if this pattern has already been used before undertaking a deeper algorithmic study.

3. Related Work

From Prescriptive Processes to Descriptive Pipeline Abstractions Early efforts to characterize data mining and analytic projects produced prescriptive process models such as CRISP-DM, which remain useful as high-level process templates. These models emphasise tasks and decision points but are intentionally abstract and prescriptive rather than being tailored to capture the diverse, fine-grained variations found in real ML code and notebooks (Martínez-Plumed et al. 2019).

In Model-Driven Engineering, standards such as UML Activity Diagrams and BPMN provide precise semantics for control flow and activity structure; MDE approaches have been successfully applied to workflow modelling and domain-specific workflow languages (Object Management Group (OMG) 2017, 2014) to more tailored solutions (Annable et al. 2023; Salado-Cid et al. 2023). However, full-scale adoption of these standards for reverse-engineered ML code is often impractical: their richness can be inordinate for exploratory Data Science code, and they are not designed to accommodate continuously evolving, ad-hoc pipeline variants encountered in shared code repositories.

Reverse Engineering and Notebook Analysis Model-Driven Reverse Engineering (MDRE) has long shown how complex code bases can be lifted to analyzable models, notably through frameworks such as MoDisco (Bruneliere et al. 2014). However, ML pipelines are rarely encoded in homogeneous, well-structured languages, and much of today's exploratory work occurs in notebooks or scripts, which require more domain-aware extraction mechanisms.

Many researchers have addressed ML-specific challenges by enhancing tooling, notably to extract the structure of ML pipelines—*i.e.*, the main steps such as Data Preparation, Model Building, or Training—to improve understanding without requiring detailed knowledge of the underlying code. A key focus has been on documentation support, for instance, by automatically labeling notebook cells with structural headers (Venkatesh et al. 2023; Wang et al. 2020). While these approaches enrich notebooks with high-level cues, they do not reify this information into an exploitable representation for reasoning about or comparing ML pipelines.

In (Jiang et al. 2022), the authors represent notebooks as labeled data-dependency graphs, where cells are nodes and edges encode execution dependencies. A custom taxonomy of steps is used to label cells, and a graph-based navigation interface helps users explore notebooks according to this extracted high-level structure. These approaches, however, do not target capturing multiple abstraction facets simultaneously (*e.g.*, steps, fine-grained instructions, libraries, algorithm families, metrics used, functions called).

More importantly, large-scale analysis requires moving across abstractions whose relevant concepts depend heavily on the domain being studied. Time series analysis may introduce specific steps (*e.g.*, windowing), fairness verification focuses on particular evaluation aspects, and LLM construction may emphasize benchmark characterization—elements that fall outside fixed, classical taxonomies.

These limitations highlight the need for a lightweight yet structured representation that can be incrementally enriched with descriptions, without imposing a rigid schema, while remaining suitable for downstream analysis and pattern-based exploration.

Ontologies and Standardised Descriptions Ontology-based initiatives and interchange formats provide complementary, higher-level vocabularies for describing ML experiments and components. Notably, the W3C ML-Schema (and related ontology efforts (Humm & Zender 2021)) propose canonical classes and properties to describe algorithms, datasets, and experiments, and platforms such as *OpenML* organise and expose experiment metadata at scale for reproducibility and networked science (Publio et al. 2018; Vanschoren et al. 2014).

These resources are valuable when such structured metadata exist, but they typically operate at the experiment or component description level rather than systematically discovering and representing the fine-grained, evolving structural patterns present in raw pipeline code. Using such ontologies as *optional* annotation vocabularies—rather than as a mandatory top-down schema—permits richer, context-sensitive enrichments of reverse-engineered models.

Pipelines Modeling Platforms Emerging from the ACM SIGKDD, data mining pipelines modeling platforms such as RapidMiner (Kotu & Deshpande 2014), Weka (Hall 2011), and KNIME (Berthold et al. 2009) enable the graphical definition and execution of new ML pipelines, whereas Orange (Demšar & Zupan 2013) also supports interactive education.

Our approach differs in that it adopts a reverse-engineering perspective to analyze existing large-scale codebases, leaving aside their execution. Nevertheless, these platforms point out the importance of describing ML pipelines using high-level abstractions. Such models could be integrated into our *Canopus* DSL via model transformations (*e.g.*, from PMML (Grossman et al. 1999)) to identify common patterns throughout heterogeneous sources.

Scalable Model Querying Traditional model querying approaches, such as OCL, offer strong navigation capabilities but remain tightly coupled to the metamodel's structure (Cabot & Gogolla 2012). While appropriate for validation tasks, they are not well-suited to expressing topological patterns—such as multi-step flows or contextual uses of algorithms—which quickly become verbose and difficult to maintain across large collections of models.

Pattern-based languages such as *IncQuery* offer more concise graph queries (Ujhelyi et al. 2015), yet they still rely on a stable

and structurally rich metamodel. In our setting, the metamodel is intentionally lightweight, and most semantic information is carried by annotations that evolve as domain knowledge evolves. Expressing ML-related patterns in *IncQuery*'s query language (VQL) typically requires reifying these semantic elements as structural relations in the metamodel. While this reification enables *IncQuery* to leverage its incremental matching engine, it reduces flexibility and complicates handling multi-level abstractions.

Graph query languages (*e.g.*, *Cypher* (Francis et al. 2018), *Gremlin* (Daniel et al. 2016)) provide expressive primitives for large-scale pattern matching over graphs. While such languages support node tagging and attribute-based filtering, they do not offer domain-specific abstractions tailored to Machine Learning workflows. As a result, without dedicated adaptations, both execution flow and semantic annotations are treated as generic graph elements. Expressing ML-specific queries, therefore, requires practitioners to encode pipeline topology explicitly and reason over variable-length paths (*e.g.*, to state that one ML step eventually follows another) by navigating the underlying graph schema, introducing significant accidental complexity into pipeline analysis.

In summary, existing approaches either offer rich expressiveness tightly coupled to rigid metamodels or scale efficiently using general-purpose analyses, but make it more challenging to express domain-specific queries.

Gap and Positioning. Despite their strengths, existing approaches remain limited in enabling data scientists to analyze large collections of real-world ML pipelines systematically. Process-level metamodels (*e.g.*, CRISP-DM) are too coarse-grained to capture fine algorithmic decisions, while full UML-based models are unnecessarily complex for reverse-engineered code. Notebook and script analysis approaches extract valuable information, but usually commit to a fixed structural abstraction (*e.g.*, ML stages or dependency graphs), which limits their ability to support contextual, multi-perspective reasoning over ML pipelines.

Our contribution addresses this gap by combining: (i) a lightweight, UML-aligned metamodel capturing only the structural backbone of pipelines; (ii) a description mechanism enabling incremental, multi-faceted semantic enrichment; and (iii) a domain-specific pattern language designed to query large sets of such descriptive pipelines and compile into efficient queries.

4. Proposed Solution

To address the significant diversity in both the structure and content of ML pipelines, we adopt a model-driven, pattern-based analysis approach. In our context, a pattern is a structured description of elements or behaviors that one aims to identify across ML pipelines—for example, a sequence of steps or the use of specific functions. Our solution relies on:

- A descriptive pipeline metamodel, extending UML Activity diagrams with a lightweight but extensible description mechanism;
- A Domain-Specific Language (DSL), dedicated for expressing topological and semantic patterns over these pipelines;
- A toolchain that automates pipeline extraction, supports interactive exploration, and optimizes pattern execution at scale.

4.1. Big picture

Figure 1 provides an overview of the approach, which unfolds through four main stages:

(1) Model Construction Each ML pipeline is automatically transformed into a descriptive model of an ML Pipeline (*cf.* section 5). This process uses traditional reverse-engineering techniques (AST analysis) to extract program instructions, combined with a configurable set of profiling functions (*cf.* section 4.2) that identify domain-specific information such as libraries, called functions, or ML step categories.

(2) Interactive Exploration Data scientists can explore and query these models in a dedicated environment ***Colombus*** (*cf.* section 4.3). This environment supports our ***Canopus*** DSL (detailed in section 5.3). This environment also enables users to iteratively compose, refine, and apply patterns, while benefiting from graphical views that facilitate understanding of pipelines (*cf.* fig. 3).

(3) Persistence and Efficiency Both models and patterns are persisted in a relational database. This persistence enables their reuse across analyses, supports incremental corpus enrichment, and allows the system to handle large-scale pattern matching efficiently by compiling patterns into SQL queries (*cf.* section 4.4).

(4) Model Transformations and Interoperable Outputs Descriptive models of ML Pipelines can be converted into multiple external representations—such as PlantUML for visualization, JSON for export, or regular expressions—enabling downstream analyses, pattern verification, and reporting (*cf.* section 4.5).

4.2. Constructing Descriptive model of an ML Pipelines

The first component of our toolchain constructs a descriptive model of an ML Pipeline from a Jupyter Notebook. After extracting the corresponding Python code using an AST parser, each code instruction is enriched using "*profiling functions*". These functions serve as semantic descriptors: they may simply extract specific elements from the AST (*e.g.*, libraries and functions) or perform higher-level classification tasks such as identifying the ML step category of an instruction (*e.g.*, data preparation, model training). Our contribution builds upon the ability to reuse and aggregate knowledge reported in the literature through a unified descriptive modeling process.

More formally, the construction of a ***DescriptiveMLPipeline*** uses a set of *profiling functions* $\{\pi_{k_1}, \pi_{k_2}, \ldots, \pi_{k_m}\}$. For each instruction i_t in a pipeline $p = \langle i_1, \ldots, i_n \rangle$, we define its description set as $D_t = \{\pi_{k_j}(i_t)\}_{j=1}^{m}$, where each profiling function π_{k_j} associates a *description* linking the instruction i_t to

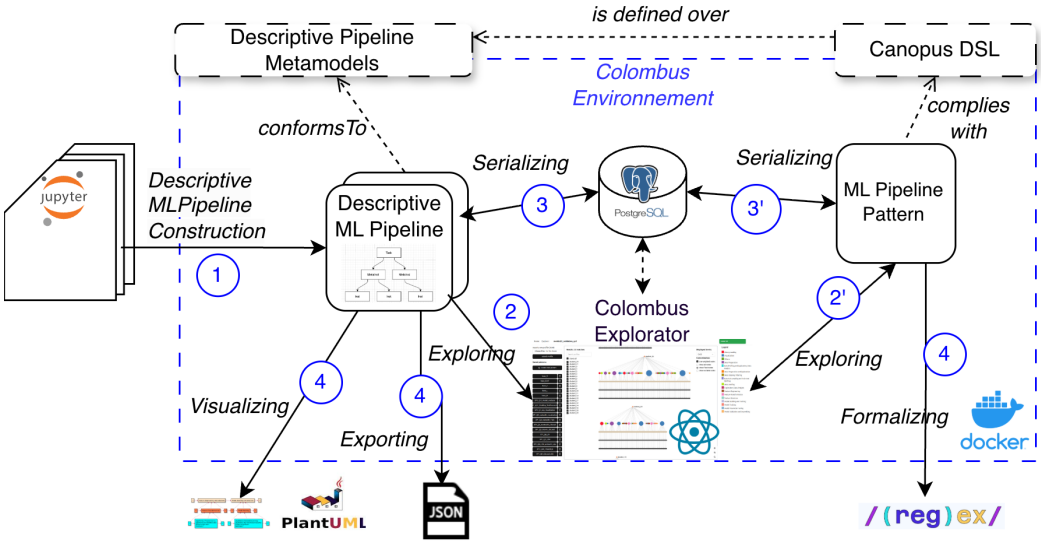

Figure 1 The ***Colombus*** environment overview, including the ***DescriptiveMLPipeline*** construction as illustrated in fig. 2, the visual explorer as represented in fig. 3 and an example of ***MLPipelinePattern*** as described in fig. 5.

a key k_j and an information value v_j. The resulting descriptive pipeline is then formally defined as a sequence of pairs: $\langle (i_1, D_1), \ldots, (i_n, D_n) \rangle$.

$$DescriptiveMLPipeline(p, \{k_1, \ldots, k_m\}) = \langle (i_1, D_1), \ldots, (i_n, D_n) \rangle.$$

The upper part of fig. 2 illustrates this enrichment process. A profiling function π_{function} extracts the raw function name (`read_csv`) from the AST, while a classification function π_{step} assigns the instruction to a step of the ML lifecycle (*e.g.*, *Data Loading*).

The result of this process is a descriptive pipeline in which each instruction is augmented with a set of descriptions computed by profiling functions. This representation preserves the original execution order while making explicit the semantic information required for higher-level architectural reasoning.

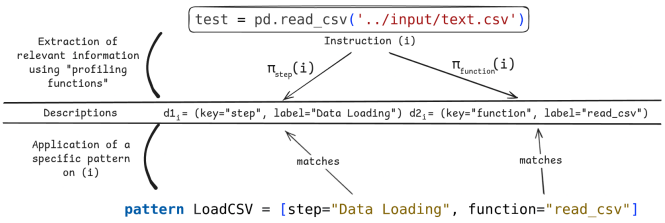

Figure 2 Illustration of two profiling functions applied to an instruction *i* and the application of a pattern *LoadCSV*

4.3. Interactive Exploration using *Colombus*

The process of our persona, Alice, is representative of the exploratory nature of data science. When data scientists look for existing solutions, they seek diverse pipelines to gain a good understanding of the different possible practices while still considering a set of constraints (*e.g.*, textual data, specific library, memory constraints). For that purpose, the ***Colombus Explorer*** provides an interactive web interface (*cf.* fig. 3) for navigating models, running pattern-based queries expressed in the ***Canopus*** DSL (detailed in section 5.3), and generating visualizations. This interface leverages a Model-To-Model transformation to Graphology (Plique 2022) for efficient visualization and navigation.

Patterns can then be iteratively refined based on the exploration results and saved for later reuse—either applied directly, combined with other patterns, or used in different analysis contexts. Concretely, fig. 3 represents the execution of a pattern on a set of ML pipelines. The defined pattern (upper section in the figure) looks for *Evaluation* steps followed by *Data Preparation* (the arrow represents the sequence as detailed in section 5.3).

4.4. Knowledge Persistence and Query Efficiency

Because each analysis of Alice consists of gathering knowledge about how data scientists previously structured their ML pipelines for a given problem, it is important to consider every artefact produced—from the created ***DescriptiveMLPipelines*** to user-defined patterns (*cf.* section 5.3)—as an integral part of the knowledge base. For example, as illustrated in fig. 2, Alice can save a pattern she has defined (using the *"Save pattern"* button shown at the top of the figure) to reuse it later, or to compose it with other patterns to build more complex queries.

As a consequence, their persistence in a PostgreSQL database is primarily intended to enable the knowledge base extension as different analyses are performed and to compose new patterns with previously created ones. Finally, storing ***DescriptiveMLPipelines*** and patterns permits reproducible analyses (as illustrated by our reproduction package[2]).

To enable efficient pattern matching, the exploration platform compiles ***Canopus*** DSL patterns—such as the structural

[2] Reproduction package available as a Zenodo artifact at https://doi.org/10.5281/zenodo.17978577

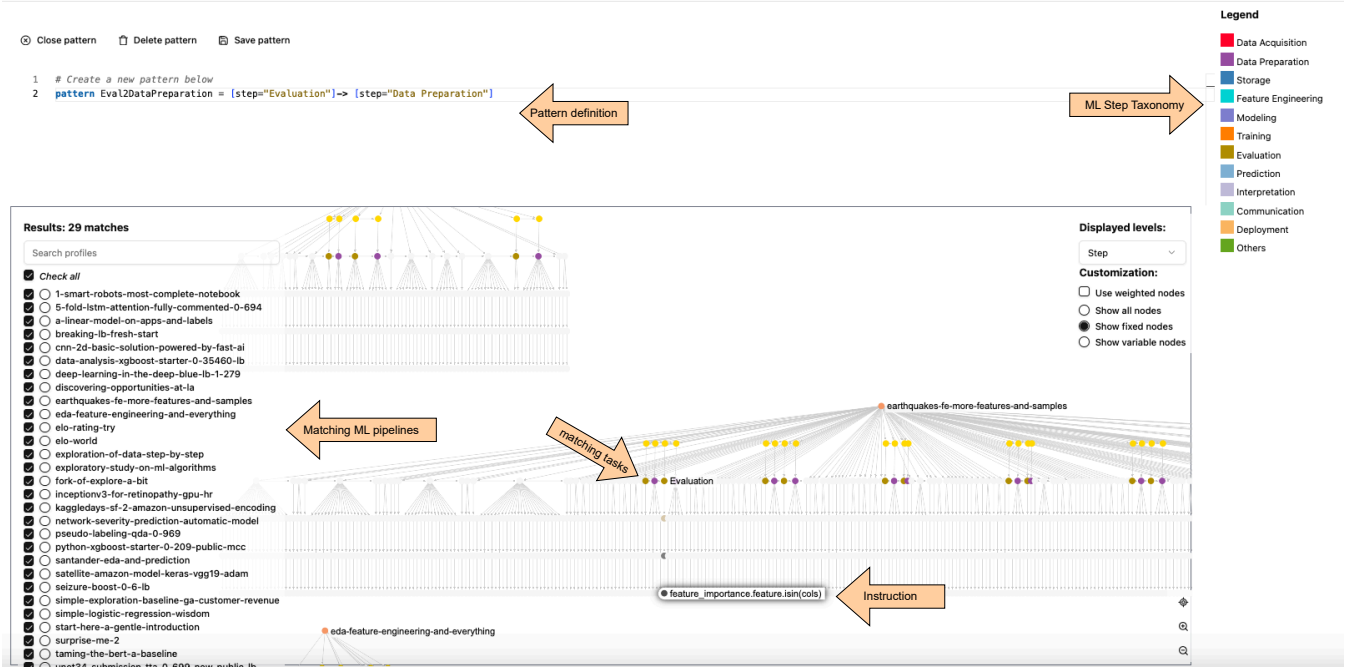

Figure 3 Screenshot of the ***Colombus*** interactive interface displaying ML pipelines, highlighting results for the pattern: *Evaluation* (brown) followed by *Data Preparation* (purple). A single pipeline flows horizontally (from left to right). Multiple pipelines are stacked vertically (from top to bottom).

sequences defined by Alice during her exploration—into SQL queries via a Model-to-Text transformation. The transformation is managed by a templating engine that automatically converts the structural constraints of a pattern (*e.g.*, a step "Evaluation" followed by any step and then a step "Data Preparation") into optimized SQL joins and filters. This ensures both efficient execution over large collections of pipelines and maintainable query generation, while keeping the process transparent to the user.

4.5. Interoperability and Model-to-Text Transformations

To support the use of ML pipeline analyses beyond the ***Colombus*** platform, we provide several Model-to-Text transformations that expose pipeline models and patterns in formats suitable for external tools and workflows.

External Pattern Verification Once an anti-pattern has been identified during interactive exploration, Alice may want to ensure that it does not reappear in future pipeline developments. This naturally motivates the integration of pattern-based checks into external processes, such as Continuous Integration workflows. Such usage scenarios have been discussed in recent work on continuous quality control for ML pipelines (Bruel et al. 2025). To support this transition from exploratory analysis to automated verification, we provide two complementary Model-to-Text transformations. The first converts ***DescriptiveMLPipelines*** into JSON representations of pipelines. The second translates user-defined *Patterns* into raw regular expressions (Regex), allowing them to be reused unchanged for pattern matching outside the modeling environment. Together, these transformations ensure interoperability by enabling the reuse of the results of Alice's exploratory analyses in external tools and workflows.

Documentation Support We further support Alice's analysis by allowing visual reporting and knowledge sharing. For this purpose, ***DescriptiveMLPipelines*** can be automatically transformed into PlantUML diagrams using a templating engine. This Model-to-Text transformation generates portable, human-readable documentation that can be shared with other data scientists.

5. Unified Metamodel for Descriptive ML Pipelines

Our approach provides an extensible metamodel for ML pipelines, built on core UML Activity concepts and organized around a separation of concerns as shown on fig. 4: pipeline structure (***DescriptivePipeline***) is handled independently from domain-specific information (***DescriptiveMLPipeline***),

Developed through a long-standing collaboration between Software Engineering and Data Science researchers, the metamodel evolved from earlier, more complex designs (Camillieri et al. 2016). Iterative work with data scientists revealed two essential needs: first, models must be easily adaptable to evolving practices; second, reasoning must be grounded in the artifacts that practitioners actually trust—namely, the source code itself (Amraoui et al. 2022; Brault et al. 2023). By modeling pipelines directly from code, we deliberately focus on capturing only the information required to support meaningful analysis,

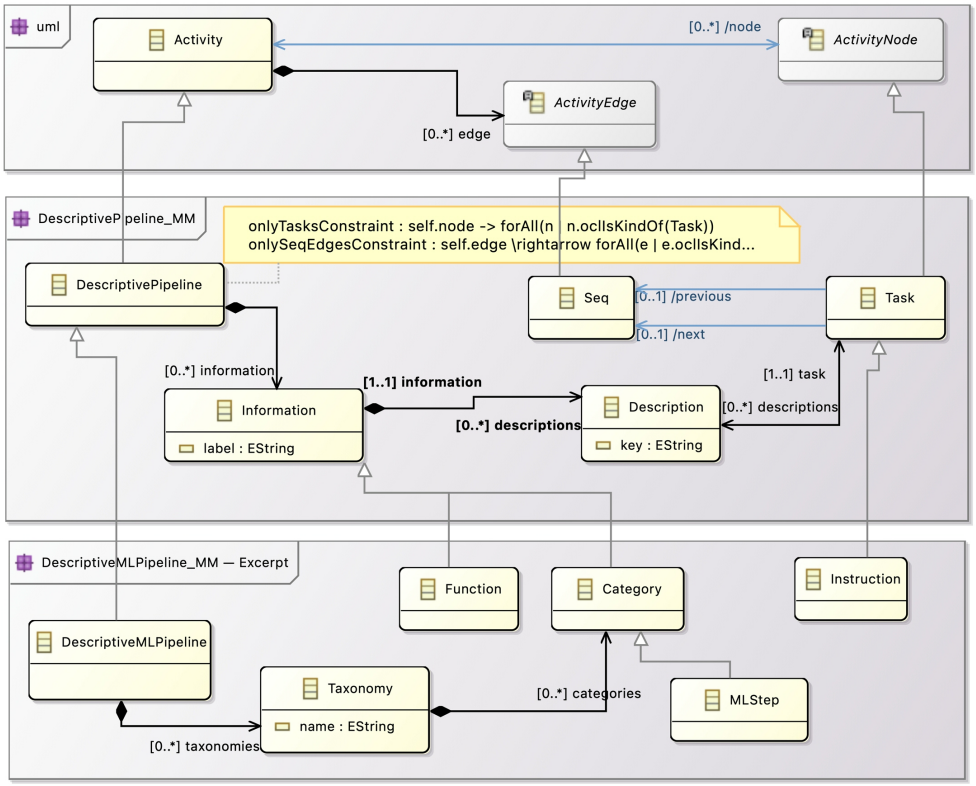

Figure 4 Metamodel for Descriptive ML Pipelines represented in ECore.

without encoding domain-specific best practices or imposing prescriptive structures.

5.1. Descriptive Pipeline Metamodel

As introduced above, the metamodel is organized around two complementary parts that reflect this separation of concerns: (i) a *foundational structure*, obtained by lightweight specializations of UML Activity concepts, and (ii) a *flexible description mechanism* that supports incremental enrichment. This separation allows us to capture the diversity found in extensive collections of ML pipelines while avoiding unnecessary structural complexity.

- **Pipeline Structure.** `DescriptivePipeline` specializes `UML::Activity` to represent the overall execution flow of an ML pipeline. `Task` and `Seq` specialize `UML::ActivityNode` and `UML::ControlFlow`, respectively, while redefining their associations to enforce a linear execution model: each `Task` may have at most one incoming and one outgoing `Seq`. This restriction captures the sequential nature of ML pipelines and provides a minimal yet sufficient control-flow backbone. This structural specialization makes pipeline semantics explicit and simplifies validation. It also facilitates querying and pattern matching by ensuring a predictable and uniform topology.
- **Task Abstraction.** `Task` represents the elementary unit of behavior within a pipeline. Tasks can be enriched with heterogeneous descriptive information via the description mechanism, allowing diverse, multi-faceted domain insights to be attached without modifying the structural core.
- **Information Hierarchy.** `Information` is an abstract root capturing any structured descriptor of the code. Unlike `UML::Comment`, which only supports unstructured text, `Information` accommodates typed extensions (*e.g.*,, visualisations, functions, metrics, ML step categories based on a specific taxonomy), allowing the metamodel to evolve with domain knowledge without modifying its structural core.
- **Description Mechanism.** A `Description` is a non-structural relation between a `Task` and an `Information` element. Its `key` attribute (*e.g.*, `step`, `function`) enables semantically meaningful, selective querying (*cf.* section 5.3), supporting domain-specific analysis without inflating the model's structural footprint.

This lightweight metamodel serves as the semantic foundation for our Pattern DSL, introduced in section 5.3.

5.2. ML-Specific Metamodel Specialization for Extensibility

The `DescriptiveMLPipeline` specialization (extending `DescriptivePipeline`) defines ML-specific `Information` types by extending the `Information` hierarchy, thereby ensuring controlled, modular extensibility.

The fig. 4 presents an excerpt of the specialization used in our evaluation (*cf.* section 6). It enables linking ML instructions to code-extracted information (*e.g.*, the function invoked)

and categorizing them according to a dedicated `Taxonomy` via profiling functions formalized in section 4.2.

Thanks to the description mechanism, additional ML-relevant insights can be introduced without refactoring existing descriptions. For instance, one may add new `Information` subclasses to represent the benchmarks used to evaluate a pipeline, or properties of the produced ML models (*e.g.*, a neural network and its number of layers, or a decision tree and its depth).

From an engineering perspective, such extensions involve: (i) defining the new subclass in the metamodel, (ii) implementing a dedicated profiling function—which can include custom Python scripts for complex AST traversal—and (iii) registering the metadata in the persisted knowledge base (currently via a dedicated SQL table).

This architecture preserves the structural core's stability while enabling targeted enrichment. Furthermore, these specialized information types support domain conformance checks, ensuring that certain metadata is attached only when it is semantically valid.

5.3. Pattern Matching over Descriptive Pipelines

To support the identification of reusable analysis structures within ML pipelines, we introduce ***Canopus*** DSL, a pattern language designed to remain easy to use for data scientists. It enables data scientists to describe recurring motifs at both the structural and semantic levels without requiring prior knowledge of model-based querying. The language complements the ***DescriptiveMLPipeline*** metamodel by providing a declarative mechanism for querying, comparing, and abstracting pipeline fragments. Patterns serve to reveal common practices, highlight meaningful variations across pipelines, and progressively enrich the knowledge base with higher-level abstractions—all while avoiding prescriptive or fixed pipeline structures.

Pattern Syntax and Semantics A pattern is defined as a sequence of activities connected by the operator `->` and optionally bounded by an input boundary assertion (`start`, `end`). Similarly to regular expressions, not specifying an input boundary assertion will look for the specific pattern anywhere in the pipeline. A pattern is further enriched with constraints on descriptions (*e.g.*, library, function, or step type). Quantifiers (`*`, `+`), grouping, and disjunction (`|`) support the expression of flexible motifs. Conditions are expressed using bracket notation, which filters activities based on the attached semantic information. A visual representation of pattern definitions is available in fig. 5.

The syntax was initially inspired by regular expressions, with a constrained activity referring to the concept of a capturing group and reusing common multiplicity operators. However, we made some operators more explicit (*e.g.*, `start`, `end`) for simplified usage[3].

The concrete syntax of ***Canopus*** DSL is defined by the grammar shown in listing 1.

```
grammar CanopusDSL;

program: importPatterns* patternDef+ EOF;
importPatterns: 'import' ID (',' ID)*;
patternDef: 'pattern' ID '=' patternExpr;
patternExpr:
  START_OP '->' middleExpr '->' END_OP
  | START_OP '->' middleExpr
  | middleExpr '->' END_OP
  | middleExpr;
middleExpr: multipExpr ('->' multipExpr)*;
multipExpr: primary multipOperator?;
primary:
  '(' expr ('|' expr)+ ')'
  | '[' condition (',' condition)* ']'
  | ID
  | WILDCARD
expr: primary ('->' primary)*;
condition: LABEL comparator STRING;
multipOperator: '*' | '+';
comparator: '=' | '!=';

ID: [A-Z][a-zA-Z0-9_]*;
LABEL: [a-z][a-zA-Z0-9_]*;
STRING: '"' (~["\r\n])* '"';
WS: [ \t\r\n]+ -> skip;
WILDCARD: '*';
COMMENT: '#' ~[\r\n]* -> skip;
START_OP: 'start';
END_OP: 'end';
```

Listing 1 Excerpt of the CanopusDSL grammar

Pattern Composition Patterns can be composed, allowing users to iteratively refine motifs by reusing previously defined patterns as building blocks. This supports incremental exploration, where initial coarse patterns are progressively specialized to reflect domain knowledge or newly observed variations. As illustrated in fig. 5, a first pattern is created to look for CSV data acquisition. It is later used in *ExamplePattern*, highlighting the composition and refinement capabilities.

Following the need for extensibility, related to the constant evolution of the ML domain, we designed the ***Canopus*** DSL to interpret domain concepts via the `Information` attached to tasks. This avoids relying on a pre-defined set of concrete information and naturally adapts to new profiling functions and emerging ML trends. As part of the ***Colombus*** exploration platform, the suggestion of supported description keys is driven by the knowledge base rather than by the language itself.

Overall, this design ensures that data scientists can run meaningful queries during incremental explorations across heterogeneous ML pipelines, without requiring recurrent language modifications.

6. Evaluation

In this section, we qualitatively evaluate the relevance of using our metamodel and exploration platform with respect to the identified scenarios. This evaluation was conducted with a data scientist representative of our motivating scenario's persona. Similar to Alice, he has 20 years of experience in the field of data science and has developed many ML pipelines during his career. Additionally, he has collaborated with industrial R&D teams in companies specializing in artificial intelligence. Our evaluation is threefold, following the 3 identified use cases:

[3] We have focused our analysis on the model transformations rather than on the expressiveness of regular expressions (Backurs & Indyk 2016), which we will address in future work.

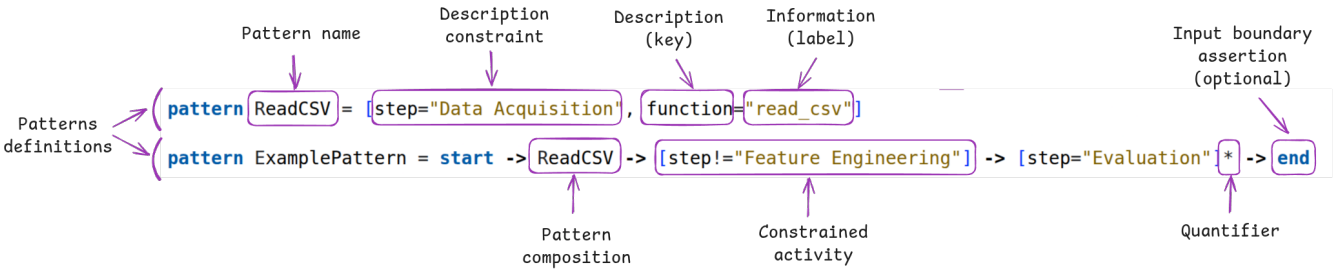

Figure 5 Illustration of two patterns, annotated with concepts from the metamodel and DSL definition

1. select a subset of pipelines based on several structural criteria (*cf.* section 6.2);
2. systematically reproduce insights about ML pipeline structures, previously collected manually by other researchers, considering them as compliance targets (*cf.* section 6.3);
3. interactively investigate ML pipelines structures to distinguish context-related refinements from anti-patterns (*cf.* section 6.4).

For that purpose, we adopt a pattern-matching approach using the introduced DSL to identify specific structures in a set of ML pipelines. The expert has access to the ***Colombus*** platform, can edit patterns, and can visualize matching results, as illustrated in fig. 3. In the results, a match is defined as "the pattern has been found at *least one* time in the pipeline".

6.1. Case Study Description

The case study is based on the peer-reviewed study "*The Art and Practice of Data Science Pipelines*" (Biswas et al. 2022), published at *ICSE* in 2022, which analyzes data science pipelines in three different contexts: as described in the literature (*In Theory*), as present in Kaggle competitions (*In-The-Small*) and as present in GitHub repositories (*In-The-Large*). In the original study, notebooks were parsed into individual code instructions and classified according to a taxonomy of ***ML steps*** comprising 12 categories, including *Data Acquisition*, *Data Preparation*, *Modeling*, *Training*, *Evaluation*, and *Prediction*. To comply with our motivating scenario, we will address the second context which gives first insights about what a "representative pipeline" looks like from a dataset of 105 Kaggle notebooks. In the selected context, the authors reported 3 key *findings* that section 6.3 reproduces. We keep the same numbering as the original study (from 3 to 5) for easier comparison:

- Finding 3: Evaluation stage is infrequent, appearing only in 36% of the pipelines in-the-small.
- Finding 4: Stages of pipelines in-the-small are often tangled with each other.
- Finding 5: Data preparation stage is occurring significant number of times between any two stages of pipelines in-the-small, which is causing pipeline jungles.

Additionally, complementary *observations* described in the original study were investigated. We make the distinction between a *finding*, which is highlighted as a key contribution in the original study, and an *observation*, which is a complementary insight reported from the same study.

- Observation 1: Among 105 programs, data acquisition and data preparation are present in almost all of them
- Observation 2: A "representative pipeline" is a sequence of *Data Acquisition*, *Data Preparation*, *Modeling*, *Training*, *Evaluation*, and *Prediction* (*cf.* figure 6 of original study)
- Observation 3: The stages are connected with "feedback loops" (*e.g.*, *Prediction* back to *Modeling*) (*cf.* figure 3 of original study)

To better reflect the naming conventions used in the literature and make a clear distinction between the stages defined in the CRISP-DM metamodel from the steps we actually study, we will use the term "step" to refer to what Biswas, Wardat and Rajan call "stage" in (Biswas et al. 2022).

Overall, this corpus contains 1,632 steps (*i.e.*, consecutive code instructions grouped by their ML step category), for a total of 5,228 individual code instructions, making manual exploration obviously tedious. Additional statistical details about the dataset are provided in table 1.

Table 1 Statistical description of the case study dataset

elements	count	mean	std	min	max
steps	1,632	15.7	20.56	1	182
instructions	5,228	50.3	67.2	2	459

For our analysis, we reuse their AST parser to extract code instructions and extend it to retrieve the function names (π_1). A second profiling function (π_2) involves their classification mapping to associate each instruction with an ML step category. To remain consistent with the methodology of the original study and enable fair comparison, we merge consecutive identical steps into a single step (*i.e.*,, *Modeling -> Modeling -> Training* is stored as *Modeling -> Training*). While this strictly adheres to their profiling methodology, it also inherits any inherent misclassifications, which we identify and discuss alongside our evaluation results.

6.2. UC_1: Structural Filtering

To validate our solution for the first use case UC_1, we must demonstrate the ability to select a subpart of the corpus based on specific criteria coming from our expert.

Based on *Observation 1*, our expert decided to better understand what kind of data had been acquired. First, he created a pattern to filter all pipelines, effectively acquiring data:

```
pattern DataAcquisition = [
    step="Data Acquisition"
]
```

Using this straightforward pattern, with only one description constraint, allowed selecting 97 pipelines. Among them, our expert applied a finer-grained filter, involving multiple description constraints, by looking for pipelines acquiring CSV data:

```
pattern DataAcquisitionCSV = [
    step="Data Acquisition",
    function="read_csv"
]
```

This filter excluded 16 more pipelines. Among the 81 remaining pipelines, our expert was interested in the ones acquiring CSV data as a first step and passing this data into the model without any preparation. This involves constraining the order of activities, the use of input boundary assertion (start) and quantifier (+) as follows:

```
pattern StartDataAcquisitionCSV = start
    -> DataAcquisitionCSV
    -> [step!="Data Preparation"]+
    -> [step="Modeling"]
```

With only one result, our expert was able to efficiently filter pipelines based on his criteria. Without ***Colombus*** and the ***Canopus*** DSL, this search would have required reading all 105 notebooks. Looking for the read_csv function could have been easy using a proper IDE. However, looking up its usage before creating a model and ensuring the data was not preprocessed would have required a thorough reading and costly manual analysis.

6.3. UC_2: Pipeline Compliance Checking

To validate the second use case UC_2, we will consider the "representative pipeline" (*Observation 2*), as well as the other study's findings, as compliance goals. The "representative pipeline" sequence is represented as follows:

```
pattern InTheSmall = [step="Data Acquisition"]
    -> [step="Data Preparation"]
    -> [step="Modeling"]
    -> [step="Training"]
    -> [step="Evaluation"]
    -> [step="Prediction"]
```

When looking for matches in ***Colombus***, no result was found. After manually investigating the content of their reproduction package, we were able to confirm these results. Accordingly, a variant of this pattern was produced after having relaxed the constraints by allowing for intermediate steps in the pipeline:

```
pattern InTheSmallFlex =
    [step="Data Acquisition"] -> *
    -> [step="Data Preparation"] -> *
    -> [step="Modeling"] -> *
    -> [step="Training"] -> *
    -> [step="Evaluation"] -> *
    -> [step="Prediction"]
```

This pattern was found in 15 matches (14.3%), suggesting that the typicality of certain sequences described in the original article may be more nuanced than initially suggested. In other words, only a small subset of the pipelines strictly follows this specific path. Overall, it highlights the advantage of our systematic approach, which enabled us to distinguish between the original study's "*perception*" and the ground truth with empirical evidence.

This previous result can be explained by their *Finding 3* highlighting infrequent evaluation in the studied ML pipelines and reproduced by the following simple pattern (38/105 matches):

```
pattern Evaluation = [step="Evaluation"]
```

This phenomenon was confirmed by retrieving 54 results (72% more results) after modifying the previous *InTheSmallFlex* pattern, by using the *zero or more* quantifier (*), to make the *Evaluation* pattern optional:

```
pattern InTheSmallOptEval =
    [step="Data Acquisition"] -> *
    -> [step="Data Preparation"] -> *
    -> [step="Modeling"] -> *
    -> [step="Training"] -> *
    -> Evaluation* -> *
    -> [step="Prediction"]
```

One important aspect reported in the study is the presence of "*feedback loops*" (*Observation 3*). Two of these loops, as illustrated in the representative pipeline, could be expressed as follows:

```
pattern FeedbackLoopPrepPred =
    [step="Data Preparation"] -> *
    -> [step="Prediction"]
    -> [step="Data Preparation"]

pattern FeedbackLoopModelPred =
    [step="Modeling"] -> *
    -> [step="Prediction"]
    -> [step="Modeling"]
```

With respectively 52 and 8 matches, we were able to confirm the presence of pipelines complying with these feedback loops and their different frequencies. The same methodology was used to confirm the concept of "*pipeline jungles*" where "*Data preparation stage is occurring a significant number of times between any two stages of pipelines*" (*Finding 5*):

```
pattern NotDataPrep = [step!="Data Preparation"]

pattern PipelineJungle =  NotDataPrep
    -> [step="Data Preparation"]
    -> NotDataPrep
    -> [step="Data Preparation"]
    -> NotDataPrep
```

With 65 matches (61.9%), this phenomenon is appearing in a majority of pipelines. Overall, the previous results confirm the notion of "*tangling stages*" (*i.e.*, "the code for one stage is interspersed with the code for other stages") as described in their *Finding 4*.

6.4. UC_3: Iterative Analysis Refinement

The last part of our evaluation, the third use case UC_3, will benefit from UC_1 and UC_2. It will highlight the iterative process, involving pattern composition and refinement, that enables

refinement of the pipeline analysis. Concretely, we first asked our expert to explore the corpus using ***Colombus*** to identify and better understand unusual practices.

When navigating the pipelines using the interactive interface, the expert was surprised to see some of them ending with *Data Preparation*. He therefore created a dedicated pattern as a basis to better understand this practice:

```
pattern EndWithDataPrep = [
    step="Data Preparation"
] -> end
```

Our expert navigated the 73 matches and identified, in some of these pipelines, the step *Prediction* appearing before our searched pattern *EndWithDataPrep*. Given the context of the notebook dataset gathered from Kaggle competitions, he understood that this new pattern corresponded to the export of results submitted to the competition organizer for evaluation. The refined pattern, including the *Prediction* step and yielding 32 results, is the following:

```
pattern SaveResults = [
    step="Prediction"
] -> EndWithDataPrep
```

With 30.5% of notebooks implementing this pattern, the taxonomy from (Biswas et al. 2022) could be extended to include a new step *Save Results*. More broadly, this exposes the challenges of defining properly and exhaustively such a taxonomy and the interest in possible adaptations to emerging practices using a pattern-driven approach.

Separately, we asked our expert to explore the different pipelines to identify a potential anti-pattern. Because the studied corpus has been curated from Kaggle competitions, our expert was interested in the identification of data contamination that could bias the results of trained models. For that purpose, he first defined the concept of data contamination, *i.e.*, the data scientist cannot train the same model a second time after having used the test data (validation data should be used instead in that case), and created the corresponding pattern:

```
pattern Contamination = [step="Modeling"]
    -> [step="Training"]
    -> [step="Prediction"]
    -> Evaluation*
    -> [step!="Modeling"]*
    -> [step="Training"]
```

Among the 4 matches, our expert was able to identify a problem related to the classification from the original study in 2 of these 4 pipelines. Indeed, he traced back the original code using ***Colombus*** and identified that each instantiation of DMatrix (*i.e.*, a data structure from the *xgboost* library), was classified as *Modeling* while it should have been labeled as *Data Preparation*. He decided to create a pattern to detect every use and potential classification issues of DMatrix:

```
pattern DMatrixUse = [
    step="Modeling",
    function="DMatrix"
]
```

Applying this pattern returned 8 matches. Although this represents only 7.6% of the corpus, these classification issues can affect the results of the original study and our understanding of data scientists' practices. Yet, this finding highlights the advantage of using pattern matching within the ***Colombus*** platform. By interactively iterating and refining patterns, our expert was able to detect such inconsistencies in the underlying taxonomy that would remain invisible in a purely statistical analysis.

6.5. Summary

As demonstrated in this evaluation, combining pattern matching with interactive exploration enables the identification of recurring practices, structural variations across pipelines, and context–specific practices. First, our expert filtered pipelines using structural and semantic constraints over activity descriptions and ordering. Second, expected pipeline structures were expressed as ***Canopus*** patterns to verify compliance and reproduce key findings reported in the literature. Finally, iterative exploration with ***Colombus*** enabled the discovery of additional practices beyond the traditional ML lifecycle and supported the progressive refinement of the analysis.

The study involved an experienced data scientist, consistent with our motivating scenario, and relied on reproducing the findings of a prior study conducted by three independent authors, thereby reducing the risk of biased interpretation. While the accuracy of the profiling function used to classify instructions affects the observations, we ensured comparability by using the same profiling mechanism as the original study. Moreover, the simplicity of the profiling function contract facilitates the creation or reuse of additional profiling functions (Lacroix et al. 2026).

Takeaways. (1) Abstraction and pattern-based exploration help experts differentiate actual pipeline practices from theoretical lifecycle models. (2) Iterative refinement between abstraction and detail is essential to understand pipeline variations. (3) Function calls alone are ambiguous; contextual descriptors are required to interpret ML practices.

7. Conclusion

In this paper, we introduced ***DescriptiveMLPipeline***, a novel metamodel-based approach for analyzing and reasoning about ML pipelines. Our solution revolves around a flexible description concept to bypass limitations of highly structured metamodels in widely variable and constantly evolving domains, as ML domain is. Such descriptions are produced by "profiling functions", referring to existing work in the reverse-engineering of ML pipelines.

The design of the ***Canopus*** DSL allows interrogating this metamodel with dedicated paterns, and the ***Colombus*** exploration platform supports the interactive analysis process through interactive visualizations and long-term persistence.

The ***DescriptiveMLPipeline*** metamodel has been implemented and validated in a case study, reusing a publicly available dataset comprising 105 notebooks. Initially motivated by discussions with domain experts, this case study addresses 3 use cases. It involves an expert who interactively navigated ML pipelines through their ML descriptions to iteratively create and compose patterns expressed using the ***Canopus*** DSL.

Overall, initial findings suggest that ***DescriptiveMLPipeline*** could assist data scientists during the

development of their ML pipelines, adapting to their practices rather than imposing others. Gathered knowledge could be shared among practitioners, contributing to greater collaboration and quality assessment in the field. Future work will first extend the evaluation study to larger-scale datasets and more diverse actors of the domain. Then, new visualizations (as model transformations) will be proposed and evaluated in terms of transparency and traceability.

Acknowledgments

This project has received funding from the French ANR Under Grant Agreement No. ANR-24-IAS2-0002-01 (TSIA 2024 – Specific Topics in Artificial Intelligence) and ANR-24-CE25-1286-01.

References

Amershi, S., Begel, A., Bird, C., DeLine, R., Gall, H., Kamar, E., … Zimmermann, T. (2019). Software engineering for machine learning: A case study. In *2019 ieee/acm 41st international conference on software engineering: Software engineering in practice (icse-seip)* (pp. 291–300).

Amraoui, Y. E., Blay-Fornarino, M., Collet, P., Precioso, F., & Muller, J. (2022). Evolvable spl management with partial knowledge: an application to anomaly detection in time series. In *Proceedings of the 26th acm international systems and software product line conference-volume a* (pp. 222–233).

Annable, N., Chiang, T., Lawford, M., Paige, R. F., & Wassyng, A. (2023). Lessons learned building a tool for workflow+. In *2023 acm/ieee 26th international conference on model driven engineering languages and systems (models)* (pp. 140–150).

Backurs, A., & Indyk, P. (2016). Which regular expression patterns are hard to match? In *2016 ieee 57th annual symposium on foundations of computer science (focs)* (pp. 457–466).

Berthold, M. R., Cebron, N., Dill, F., Gabriel, T. R., Kötter, T., Meinl, T., … Wiswedel, B. (2009). Knime-the konstanz information miner: version 2.0 and beyond. *AcM SIGKDD explorations Newsletter*, *11*(1), 26–31.

Biswas, S., Wardat, M., & Rajan, H. (2022). The art and practice of data science pipelines: A comprehensive study of data science pipelines in theory, in-the-small, and in-the-large. In *Proceedings of the 44th international conference on software engineering* (pp. 2091–2103).

Brault, Y., El Amraoui, Y., Blay-Fornarino, M., Collet, P., Jaillet, F., & Precioso, F. (2023). Taming the diversity of computational notebooks. In *Proceedings of the 27th acm international systems and software product line conference-volume a* (pp. 27–33).

Bruel, J.-M., Gouaichault, T., Teste, O., Blay-Fornarino, M., Lacroix, N., Precioso, F., & Mosser, S. (2025). Incorporating fates principles in continuous development of ml-integrated systems: Importance of requirements. In *2025 ieee 33rd international requirements engineering conference workshops (rew)* (pp. 425–431).

Bruneliere, H., Cabot, J., Dupé, G., & Madiot, F. (2014). Modisco: A model driven reverse engineering framework. *Information and Software Technology*, *56*(8), 1012–1032.

Cabot, J., & Gogolla, M. (2012). Object constraint language (ocl): a definitive guide. In *International school on formal methods for the design of computer, communication and software systems* (pp. 58–90). Springer.

Camillieri, C., Parisi, L., Blay-Fornarino, M., Precioso, F., Riveill, M., & Cancela-Vaz, J. (2016). Towards a software product line for machine learning workflows: Focus on supporting evolution. In *10th workshop on models and evolution co-located with acm/ieee 19th international conference on model driven engineering languages and systems (models 2016)*.

Daniel, G., Sunyé, G., & Cabot, J. (2016). Mogwaï: a framework to handle complex queries on large models. In *2016 ieee tenth international conference on research challenges in information science (rcis)* (pp. 1–12).

De Martino, V., & Palomba, F. (2025). Classification and challenges of non-functional requirements in ml-enabled systems: A systematic literature review. *Information and Software Technology*, *181*, 107678.

Demšar, J., & Zupan, B. (2013). Orange: Data mining fruitful and fun-a historical perspective. *Informatica*, *37*(1).

Dubinsky, Y., Rubin, J., Berger, T., Duszynski, S., Becker, M., & Czarnecki, K. (2013). An exploratory study of cloning in industrial software product lines. In *2013 17th european conference on software maintenance and reengineering* (pp. 25–34).

Francis, N., Green, A., Guagliardo, P., Libkin, L., Lindaaker, T., Marsault, V., … Taylor, A. (2018). Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 international conference on management of data* (pp. 1433–1445).

Granger, B. E., & Pérez, F. (2021). Jupyter: Thinking and storytelling with code and data. *Computing in Science & Engineering*, *23*(2), 7–14.

Grossman, R., Bailey, S., Ramu, A., Malhi, B., Hallstrom, P., Pulleyn, I., & Qin, X. (1999). The management and mining of multiple predictive models using the predictive modeling markup language. *Information and Software Technology*, *41*(9), 589–595.

Hall, M. (2011). Practical machine learning tools and techniques. *United State: Morgan Kauffman*.

Herrmann, M., Lange, F. J. D., Eggensperger, K., Casalicchio, G., Wever, M., Feurer, M., … Bischl, B. (2024). Position: Why we must rethink empirical research in machine learning. *arXiv preprint arXiv:2405.02200*.

Huang, R., Ravi, S., He, M., Tian, B., Lerner, S., & Coblenz, M. (2025). How scientists use jupyter notebooks: Goals, quality attributes, and opportunities. *arXiv preprint arXiv:2503.12309*.

Humm, B. G., & Zender, A. (2021). An ontology-based concept for meta automl. In *Artificial intelligence applications and innovations: 17th ifip wg 12.5 international conference, aiai 2021, hersonissos, crete, greece, june 25–27, 2021, proceedings 17* (pp. 117–128).

Jebnoun, H., Rahman, M. S., Khomh, F., & Muse, B. A. (2022). Clones in deep learning code: what, where, and why? *Empirical Software Engineering*, *27*(4), 84.

Jiang, Y., Kästner, C., & Zhou, S. (2022). Elevating jupyter notebook maintenance tooling by identifying and extracting notebook structures. In *2022 ieee international conference on software maintenance and evolution (icsme)* (pp. 399–403).

Jordan, M. I., & Mitchell, T. M. (2015). Machine learning: Trends, perspectives, and prospects. *Science*, *349*(6245), 255–260.

Koenzen, A. P., Ernst, N. A., & Storey, M.-A. D. (2020). Code duplication and reuse in jupyter notebooks. In *2020 ieee symposium on visual languages and human-centric computing (vl/hcc)* (pp. 1–9).

Kotu, V., & Deshpande, B. (2014). *Predictive analytics and data mining: concepts and practice with rapidminer*. Morgan Kaufmann.

Lacroix, N., Blay-Fornarino, M., Mosser, S., & Precioso, F. (2026). Using small language models to reverse-engineer machine learning pipelines structures. *arXiv preprint arXiv:2601.03988*.

Martínez-Plumed, F., Contreras-Ochando, L., Ferri, C., Hernández-Orallo, J., Kull, M., Lachiche, N., … Flach, P. (2019). Crisp-dm twenty years later: From data mining processes to data science trajectories. *IEEE transactions on knowledge and data engineering*, *33*(8), 3048–3061.

Object Management Group (OMG). (2014). *Business Process Model and Notation (BPMN) Version 2.0.2.* Retrieved from https://www.omg.org/spec/BPMN/2.0.2/

Object Management Group (OMG). (2017). *UML 2.5.1 Specification.* Retrieved from https://www.omg.org/spec/UML/2.5.1/About-UML/

Pimentel, J. F., Murta, L., Braganholo, V., & Freire, J. (2019). A large-scale study about quality and reproducibility of jupyter notebooks. In *2019 ieee/acm 16th international conference on mining software repositories (msr)* (pp. 507–517).

Plique, G. (2022). graphology.

Publio, G. C., Esteves, D., Panov, P., Soldatova, L., Soru, T., Vanschoren, J., … others (2018). Ml-schema: exposing the semantics of machine learning with schemas and ontologies. *arXiv preprint arXiv:1807.05351*.

Ramasamy, D., Sarasua, C., Bacchelli, A., & Bernstein, A. (2023). Workflow analysis of data science code in public github repositories. *Empirical Software Engineering*, *28*(1), 7.

Randles, B. M., Pasquetto, I. V., Golshan, M. S., & Borgman, C. L. (2017). Using the jupyter notebook as a tool for open science: An empirical study. In *2017 acm/ieee joint conference on digital libraries (jcdl)* (pp. 1–2).

Rule, A., Tabard, A., & Hollan, J. D. (2018). Exploration and explanation in computational notebooks. In *Proceedings of the 2018 chi conference on human factors in computing systems* (pp. 1–12).

Salado-Cid, R., Vallecillo, A., Munir, K., & Romero, J. R. (2023). Swel: A domain-specific language for modeling data-intensive workflows. *Business & Information Systems Engineering*, *66*(2), 137–160.

Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., … Dennison, D. (2015). Hidden technical debt in machine learning systems. *Advances in neural information processing systems*, *28*.

Tschalzev, A., Marton, S., Lüdtke, S., Bartelt, C., & Stuckenschmidt, H. (2024). A data-centric perspective on evaluating machine learning models for tabular data. *Advances in Neural Information Processing Systems*, *37*, 95896–95930.

Ujhelyi, Z., Bergmann, G., Hegedüs, Á., Horváth, Á., Izsó, B., Ráth, I., … Varró, D. (2015). Emf-incquery: An integrated development environment for live model queries. *Science of Computer Programming*, *98*, 80–99.

Vanschoren, J., Van Rijn, J. N., Bischl, B., & Torgo, L. (2014). Openml: networked science in machine learning. *ACM SIGKDD Explorations Newsletter*, *15*(2), 49–60.

Venkatesh, A. P. S., Wang, J., Li, L., & Bodden, E. (2023). Enhancing comprehension and navigation in jupyter notebooks with static analysis. In *2023 ieee international conference on software analysis, evolution and reengineering (saner)* (pp. 391–401).

Wang, J., Kuo, T.-y., Li, L., & Zeller, A. (2020). Assessing and restoring reproducibility of jupyter notebooks. In *Proceedings of the 35th ieee/acm international conference on automated software engineering* (pp. 138–149).

Zhang, A. X., Muller, M., & Wang, D. (2020). How do data science workers collaborate? roles, workflows, and tools. *Proceedings of the ACM on Human-Computer Interaction*, *4*(CSCW1), 1–23.

About the authors

Nicolas Lacroix is a PhD student in Software Engineering at Côte d'Azur University (France), focusing on the modeling and empirical study of machine learning pipelines to support responsible AI. You can contact the author at nicolas.lacroix@univ-cotedazur.fr.

Mireille Blay-Fornarino is Professor at Côte d'Azur University (France), focusing on software engineering approaches to improve the analysis, quality, and understanding of ML practices. You can contact the author at mireille.blay@univ-cotedazur.fr.

Philippe Collet is Professor at Côte d'Azur University (France), specializing in variability management and software comprehension in large-scale codebases. You can contact the author at philippe.collet@univ-cotedazur.fr.

Frédéric Precioso is Professor at Côte d'Azur University (France), researching data science foundations for frugal and ethical Small Language Models (FATES). You can contact the author at frederic.precioso@univ-cotedazur.fr.

Sebastien Mosser is Associate Professor of Software Engineering at McMaster University (Ontario, Canada). At McMaster, he is Associate Chair of the Department of *Computing and Software* (CAS) and Associate Director of McSCert (*McMaster Centre for Software Certification*). His research interests are related to software engineering, software composition, domain-specific languages and modelling at large. You can contact the author at mossers@mcmaster.ca.