Informatics in Education, 2025, Vol. 24, No. 3, 431–470

DOI: 10.15388/infedu.2025.21

Provenance-based Data Analysis in Block-based Programming for Application Development

Naira ARRUDA[1], Matheus Roberto DE LIMA[2], Simone MARTINS[2,*], Daniel DE OLIVEIRA[2]

[1]*Xanxerê Campus, Federal Institute of Education, Science and Technology of Santa Catarina Brazil*
[2]*Institute of Computing, Universidade Federal Fluminense, Brazil*
e-mail: naira.alice@ifsc.edu.br, matheusroberto@id.uff.br, simone@ic.uff.br, danielcmo@ic.uff.br

Received: June 2025

Abstract. Teaching programming to elementary and high school students is important for developing problem-solving and logical reasoning skills. Block-based programming frameworks, such as Scratch and Kodular, have gained popularity for introducing programming concepts in an engaging and more didactic manner. However, these frameworks lack structured tools for analysing student learning processes, which makes it difficult to track progress, identify challenges, and understand student behaviour during application development. This manuscript presents `EduPROV`, a provenance-based approach that extracts, structures, and analyses student actions from log files generated by block-based programming frameworks. By storing this data in a queryable format, `EduPROV` supports the identification of learning bottlenecks, tracking programming trajectories, and can help refine teaching strategies. `EduPROV` was evaluated in a study with elementary and high school students from three schools in southern Brazil, using Kodular as the block-based programming framework. The results show that provenance analysis helps reveal student behaviour, contributing to more informed and effective programming education.

Keywords: provenance data, block-based programming.

1. Introduction

Integrating programming skills into school curricula has received increasing attention over the past decade, with numerous studies highlighting its importance, particularly in basic education (K-12) (Sun *et al.*, 2022; Schulte *et al.*, 2025). This age range, spanning from kindergarten through 12th grade, represents a critical period for cognitive and skill development. Recognizing the importance of this developmental stage, governmental and non-governmental organizations worldwide have implemented initiatives to pro-

*Corresponding author.

mote early programming education (Szabo *et al.*, 2019; Schulte *et al.*, 2025). Beyond the explicit goal of fostering logical thinking and problem-solving, Schulte *et al.* (2025) note that programming education for K-12 students also serves economic purposes (*e.g.*, preparing students for AI-driven careers) and societal goals (*e.g.*, promoting equity and social justice). Promoting a problem-solving mindset in education is not new; its roots trace back to Papert's constructionism (Papert, 1980). This pedagogical approach, which emphasizes "learning by doing", aligns naturally with the introduction of block-based programming to children and adolescents, reinforcing that hands-on learning is a fundamental pillar of practical education.

Although the idea is interesting and promising, integrating programming skills into school curricula is a complex, yet critical, issue. Teaching programming to young students, particularly those in the K-12 age group, proves to be hard (or even impractical) when relying on traditional text-based programming languages such as C, C++, Java, or even Python. Mladenović *et al.* (2016) conducted a study that presented statistically significant differences in children's ability to learn the concept of loops when using block-based programming frameworks, compared to traditional text-based languages such as Python. Consequently, block-based programming has gained much attention for its effectiveness in educational contexts.

Block-based programming methodology is based on visual environments, *e.g.*, Scratch[1], Snap![2], Kodular[3], where colorful, intuitive blocks represent programming commands, constructs, and structures. This allows students to build code in a logical and accessible way, simply by dragging and placing these blocks (Schulte *et al.*, 2025). This approach aims to minimize the difficulties associated with the syntax and complexity of traditional programming languages. It relies on the metaphor of "programming as a puzzle piece" (Weintrop, 2019), providing a visual mechanism for young programmers to develop their programs. It provides clear guidance on applying specific language commands and constructs, such as "if", "loop", "while", and "for". An advantage of block-based programming frameworks is that students do not require prior knowledge of these language constructs. Instead, they can explore the available options in the environment and select those that apply to their problem, fostering an intuitive and exploratory learning experience.

However, even with the advantages offered by block-based programming frameworks, some students continue to face challenges in learning how to program (Mladenović *et al.*, 2016). Therefore, analysing the reasoning behind program development in a block-based framework is a top priority in identifying learning difficulties and bottlenecks. While many existing frameworks provide access to or allow downloading log files detailing users' actions, these log files are often not temporally organized or structured in a queryable way. As a result, users must create custom scripts to extract and manage data, or perform comparisons across multiple files. This process is tedious and prone to errors, as different users may interpret and extract different metadata and actions from the same log files. Such inconsistencies introduce complexity to analysing challenges and barriers in effectively teaching block-based programming (Torre *et al.*, 2024).

[1] https://scratch.mit.edu/
[2] https://snap.berkeley.edu/
[3] https://www.kodular.io/

Provenance (Freire *et al.*, 2008; de Oliveira *et al.*, 2015; Herschel *et al.*, 2017) can play a fundamental role in analysing the actions performed by children during a block-based programming session or a class. In this context, provenance represents the detailed history of data inputs, intermediate steps, and processing stages associated with a particular process (Hey *et al.*, 2011). Provenance has been successfully applied across various domains to facilitate the analysis, understanding, validation, and reproducibility of workflows and processes. For instance, it has been widely used in fields such as Bioinformatics (Salazar *et al.*, 2021), Engineering (Barbosa *et al.*, 2020), Machine Learning (Nakandala *et al.*, 2020; Pina *et al.*, 2025b,a), and Education (Marques *et al.*, 2024) to enhance the transparency and interpretability of complex systems and applications. In block-based programming teaching, provenance data becomes valuable for analysing students' learning journeys. By capturing a detailed data derivation path (*i.e.*, a queryable log) of their interactions with the programming framework, provenance enables teachers to gain insights into the learning process. It allows for identifying specific actions performed or omitted by the students, highlights patterns of interaction, and reveals their progression in skill development. This rich data empowers teachers to better understand how students engage with the programming environment, identify areas where they face difficulties, and design targeted interventions to improve their learning experience.

This manuscript introduces `EduPROV`, an approach designed to collect, store, and analyse provenance data from block-based programming sessions to gain insights into students' programming learning processes and skill development. `EduPROV` accesses log files generated by block-programming frameworks during each programming session and searches for differences in the content of these logs to identify specific student actions, such as adding components or modifying code. This process enables the analysis of how individual students or groups achieve particular outcomes or fail to do so. The insights derived from `EduPROV` analyses are helpful to identify patterns and refine their teaching methodologies to improve programming teaching. Provenance data generated by `EduPROV` is stored as a graph in a Neo4J database[4] and follows the W3C PROV standard (Belhajjame *et al.*, 2013), ensuring compatibility and interoperability with other systems that follow the same standard. `EduPROV` was evaluated through a study involving K-12 students from three different schools in the Southern region of Brazil using the Kodular block-based programming framework. The results demonstrated the applicability of provenance data in analysing programming sessions and identifying learning patterns. Such patterns can be used to develop personalized pedagogical strategies, enabling teachers to address the unique learning needs of each student.

The remainder of this manuscript is structured as follows: Section 2 discusses provenance in the context of block-based programming. Section 3 reviews related work, highlighting existing approaches and identifying gaps `EduPROV` seeks to address. Section 4 details the proposed `EduPROV` approach, outlining its architecture and provenance model. Section 5 presents the evaluation of `EduPROV`, including experimental protocol and results. Section 6 concludes the manuscript and discusses future work.

[4] `https://neo4j.com/`

2. A Brief Tour to Provenance in the Context of Block-based Programming

In this manuscript, the concept of provenance (Freire *et al.*, 2008; Herschel *et al.*, 2017) is defined as the "sequence of actions performed by a student within a block-based programming environment during the process of developing a program or application". In other words, provenance captures not only the final version of the application but also the incremental steps and decision-making processes that led to its creation. The provenance database serves as a structured repository that systematically records this data derivation path, being compliant to standards such as the W3C PROV (Belhajjame *et al.*, 2013). By storing this information in a structured and queryable form, the database eases advanced analytical queries. Such a structured representation enables teachers to reconstruct and analyse the historical trajectory of a student's programming activity, including the addition, removal, or modification of visual programming elements and code. This feature enhances the ability to analyze the underlying processes that culminated in the final version of a student's program, thereby providing insights into the learning experience.

The process that a student undertakes when developing an application within a block-based programming environment is iterative and multifaceted, typically involving a series of interconnected actions, as described in the case study presented in Section 5. Throughout this learning process, many visual components may be introduced, customized, or removed, while the underlying application logic is extended, debugged, corrected, or entirely restructured. The specification of this workflow, *i.e.*, the set of all potential actions together with those that are effectively performed, corresponds to what is commonly referred to as Prospective Provenance or *p-prov* (Freire *et al.*, 2008). This form of provenance data represents planned steps and methodological procedures that guide application development in such contexts.

On the other hand, Retrospective Provenance, or *r-prov* (Freire *et al.*, 2008), provides a record of the actual execution of the development process, capturing in detail the operations and events that lead to the creation of the application's final version. Such data include, for instance, timestamps associated with each action carried out by the user, the specific code fragments added to different components, the identification and correction of errors, and other operational details. Furthermore, *r-prov* encompasses all files generated during a programming session (*e.g.*, `.AIA` files in the Kodular framework), thereby offering an account of the entire trajectory of application development.

There are recommendations for representing provenance data, which are designed to ease interoperability across multiple approaches designed to collect, manage, and store such information at different levels of granularity. Among the most widely adopted standards in this domain is the W3C PROV recommendation (Moreau and Groth, 2013). This recommendation provides both a conceptual data model, known as PROV-DM, an ontology, PROV-O, and a corresponding notation, PROV-N, which together enable the structured, consistent, and machine-readable representation of provenance data. At its foundation, the PROV model is structured around three primary components: (i) Entities, (ii) Agents, and (iii) Activities, as well as a set of defined relationships that interconnect these components, as presented in Fig. 1.

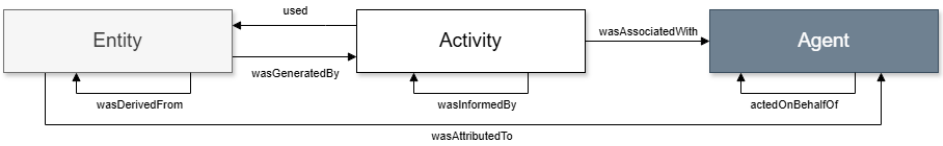


Fig. 1. An excerpt of the W3C PROV data model, adapted from Moreau and Groth (2013).

Within this recommendation, an Entity (depicted in light grey in Fig. 1) represents a physical, digital, or conceptual object. In the context of this manuscript, examples of entities include visual components of an application, *e.g.*, checkboxes, screens, and other interface elements that are used during the application development process. Activities (shown in white in Fig. 1) correspond to actions or operations carried out as part of a workflow, which interact with and potentially modify entities throughout development. Typical examples of such activities include adding, removing, or modifying visual components, capturing the temporal sequence of operations that define the application development process. Finally, an Agent (represented in dark grey in Fig. 1) refers to an individual or system responsible for executing one or more activities. Agents are critical in the provenance model, as they provide context regarding the origin and accountability.

Among the key relationships, the `wasGeneratedBy` associates an Entity with the Activity that produced it, capturing the causal relationship. For example, in a block-based programming environment, a new visual component `TextBox` would be associated with the activity `ADD` that created it. In addition, the `used` relationship connects an Activity to the Entities it consumes. The `wasAssociatedWith` relationship identifies the Agent responsible for executing an Activity, thereby attributing accountability and providing insight into the students that drive the process. Other relationships, *e.g.*, `wasDerivedFrom`, define that an Entity is a modification or transformation of another Entity. Likewise, `actedOnBehalfOf` captures delegation between Agents.

Although PROV was originally designed as a domain-agnostic recommendation, its flexible and extensible design allows it to be adapted to a wide range of specific application domains, including those focused on block-based programming. This adaptability makes it particularly suitable for capturing the nuanced interactions inherent in educational programming environments. In this regard, Section 4 provides a detailed description of the provenance model implemented within the `EduPROV` framework, illustrating how PROV has been specialized to represent the particular entities, activities, and agents relevant to block-based program development.

3. Related Work

In this section, we present a *Rapid Review* following the procedures defined by Rico *et al.* (2020). The purpose of a rapid review is to provide a structured overview of the literature by analysing the frequency, volume, and types of publications related to a

specific research topic, which in the context of this manuscript is "*Student Learning Analysis in Block-Based Programming Using Provenance Data*".

A rapid review is an alternative to a systematic literature review (SLR) (Wohlin *et al.*, 2020), offering a faster approach to synthesizing research findings. Following the guidelines outlined by Rico *et al.* (2020), we defined one primary research question (RQ1) to guide the investigation, which serves as the foundation of this section, aiming to identify and analyse relevant approaches to the topic.

Although the research question should include the term "*Provenance*", just one publication (Ribeiro dos Santos *et al.*, 2023) has used provenance data to support the analysis of educational issues; therefore, we have relaxed the scope of the research question to include studies that analyse how students learn in block-based programming environments using data collected from logs and metadata repositories.

RQ1: Which are the existing approaches for "*Student Learning Analysis in Block-Based Programming*"?

One effective approach to identify relevant publications in rapid review on the topic "*Student Learning Analysis in Block-Based Programming*" is to use keywords extracted from RQ1. These keywords include terms such as "block-based programming", and "support/analysis". In the context of this manuscript, we used Google Scholar data source and conducted a search using the search string "*(block-based programming* `OR` *block programming)* `AND` *(student)* `AND` *(learning)*". This search allowed us to gather papers to start the review process.

We also defined inclusion and exclusion criteria to determine the publications that will be considered for analysis. Our criteria consider: (i) peer-reviewed papers, (ii) papers published after 2016 (if an approach is published in more than one paper, we consider the most recent one), (iii) papers published in English or Portuguese, and (iv) studies that address the topic "*Student Learning Analysis in Block-Based Programming*". Any publication that fails to meet one or more of these established criteria is excluded from the rapid review.

In total, we identified 17 relevant publications that are associated with the topic explored in this manuscript. These studies span several academic venues, including one journal and 12 conferences or workshops. Table 1 provides metadata for each publication to show a global view of these publications compared to our proposal. It includes each manuscript's research focus, key findings, and limitations compared to `EduPROV`.

Grover and Basu (2017) analyse how effectively students learn some basic programming constructs. The analysis relies on metadata from programming sessions and identifies that students often face challenges when learning constructs like loops and using Boolean operators. While the proposed approach is interesting, it is based on limited types of metadata. It does not account for the data derivation path of an application during programming sessions, meaning it overlooks the whole learning journey of a student.

Dong *et al.* (2019) analyse the concept of tinkering within block-based programming as a particular type of student behaviour during problem-solving activities. Several

categories of tinkering based on the metadata collected from student interactions are proposed. This approach offers insights into the role of tinkering in the learning process, but it analyses this specific behaviour, not exploring other aspects of the student's learning journey.

Kesselbacher and Bollin (2019) evaluate the experiences of students learning block-based programming by accessing programming interactions within the block-based programming framework Scratch. This approach collects a series of metadata from the framework to generate metrics, but it does not structure the data properly to explore it effectively.

Table 1

A compilation of published works focusing on "*Student Learning Analysis in Block-Based Programming*" along with their respective research focus, key findings, and limitations compared to `EduPROV`

Publication	Research Focus	Key Findings	Limitations
Grover and Basu (2017)	Assess students' understanding of programming concepts in block-based environments	Students face significant conceptual challenges using block-programming.	It does not study the student learning trajectory while developing a block-based application
Dong *et al.* (2019)	Explore the concept of tinkering behaviour in block-based programming	Identifies three categories of tinkering behaviour	Focuses on how the students tinker and not other aspects of the students' learning journey
Kesselbacher and Bollin (2019)	Study on quantifying programming skills and strategies in block-based programming environments through interaction metrics.	Indicates that measurable interaction metrics can provide a fine-grained assessment of programming skills in block-based environments.	It does not generate structured data about the student´s learning trajectory to be analysed by the teachers
Krutz *et al.* (2019)	Discuss integrating stepwise refinement into block-based programming environments	Demonstrates the feasibility of integrating stepwise refinement into block programming	It does not aim to examine students' learning trajectories in the context of provenance
Zhang *et al.* (2020)	Discuss the progression of computational thinking skills through the use of block-based programming	Identifies various skills taught and assessed across different grade levels	It does not capture how the student constructs the application.
Emerson *et al.* (2020)	Develop a predictive student modelling in block-based programming environments using Bayesian hierarchical models	Predictive models incorporating student-level characteristics can support struggling students in block-based programming environments.	It represents and consumes metadata without adhering to any specific standard or recommendation for provenance representation.
Krugel and Ruf (2020)	Present a comparative study of Code.org and Scratch, focusing on learners' perspectives and motivation	Students using Code.org reported significantly higher intrinsic motivation than students using Scratch.	Its focus is not on understanding how students learn, but exploring how each framework eases the learning process.
Marwan *et al.* (2021)	Explore the impact of different scaffolding methods on students' self-regulated learning (SRL) behaviours during block-based programming.	Shows that scaffolding through sub-goal lists and adaptive feedback enhance students' progress monitoring and self-regulation in programming.	Its focus is not on investigating the learning process of block-programming students, but on which scaffolding methods are better to help them.

Continued on next page

Table 1 – continued from previous page

Publication	Research Focus	Key Findings	Limitations
Morshed *et al.* (2021)	Develop a framework for modelling students' programming abilities in block-based programming environments using progression trajectories	Identifies distinct progression trajectory patterns, showing how understanding student progression can enhance adaptive learning environments	Registers metadata without following any specific standard or recommendation for provenance representation
Kazemitabaar *et al.* (2022)	Design an intermediary programming environment to help novices transition from block-based programming to text-based programming	Highlights the effectiveness of the proposed environment in facilitating programming transitions, but acknowledged certain limitations.	Its objective is to verify if the tool developed by the authors is efficient, and not to understand the learning process of block programming.
Ribeiro dos Santos *et al.* (2023)	Develop a Provenance Model to document and identify Open Educational Resources (OER)	A framework developed to document and identify the provenance of OER	It focuses on using provenance to evaluate educational resources rather than analysing students' learning trajectories.
Tsung *et al.* (2022)	Develop a visual analytics system designed to analyse students' coding behaviours in block-based programming environments	The tool allowed educators to monitor students' overall performance on questions, identify groups of students, and common problem-solving strategies and mistakes.	It does not follow a specific standard or recommendation for provenance representation
Pereira *et al.* (2024)	Develop an interactive block programming platform to enhance programming education	User tests with students and educators indicate positive feedback on usability	The paper develops a tool to create exercises and not to understand the learning trajectories of students.
Pozzan *et al.* (2024)	Experimental analysis of block-based programming problem-solving processes using fine-grained data capture in learning analytics.	Shows the value of fine-grained process data in understanding and improving programming education	Records metadata at a fine-grained level, but it does not follow any standard or recommendation to represent this metadata, nor incorporate a data storage for querying.
Kong *et al.* (2024)	Study teaching augmentation systems designed to enhance block-based programming instruction.	Validates several design recommendations for Teaching Augmented (TA) systems	It does not present a new system or software library to understand the learning process in block programming
Torre *et al.* (2024)	Develop a systematic literature review on the analysis of learning sequences in educational contexts	Shows the growing interest in learning sequence analysis, and the need for standardized definitions and methodologies	It does not propose a new system or software library to understand the learning process in block programming
Heath *et al.* (2025)	Propose a framework to enhance code comprehension and formative assessment in block-based programming education	The proposed framework shows promise in aiding primary teachers to teach block-based programming effectively.	The metadata collected do not represent the data derivation process or follow standards or recommendations

Krutz *et al.* (2019) propose extending the Google Blockly framework by incorporating a stepwise refinement approach, leveraging provenance data from previous programming sessions. This approach uses provenance to enhance block-based programming. However, it does not aim to see students' learning trajectories within this context, distinguishing it from `EduPROV`, specifically designed to track and evaluate students' learning processes.

Zhang *et al.* (2020) gather metadata and generate indicators to determine whether a student is learning (or not) block-based programming. While this approach is a step forward in understanding the learning process, it falls short in capturing the data derivation path, *i.e.*, how the student constructs the application. Therefore, it misses some insights into the student's learning journey and development.

Emerson *et al.* (2020) propose using a series of metadata along with Bayesian hierarchical linear models to enhance the predictive performance of student modelling. The study aims to enhance adaptive support mechanisms in block-based programming frameworks by refining these predictive models. While this approach represents a step forward in understanding student learning behaviours, it consumes metadata without adhering to any specific standard or recommendation for provenance representation.

Krugel and Ruf (2020) present a comparative study of the Code.org and Scratch block-based programming frameworks. The study employs quantitative analysis relying on various metrics and metadata collected from the frameworks. It provides valuable insights, but its focus is not on understanding how students learn but on exploring how each framework eases (or does not) the learning process.

Marwan *et al.* (2021) explore the concept of Self-Regulated Learning (SRL). The authors assessed the performance of SRL through student interviews and by extracting metadata from logs generated by a block-based programming environment. Similar to `EduPROV`, their approach aims to monitor student activities by analysing metadata obtained from logs and other sources. However, logs do not explicitly represent data derivation paths or follow a specific standard or recommendation for provenance representation.

Morshed *et al.* (2021) propose a framework that collects and represents time-series data, analysing code to compare students' programs with expert-provided solutions. The framework categorized students into three learning categories: Early Quitting, High Persistence, and Efficient Completion. This study represents a step forward in understanding student learning behaviours, but it registers metadata without following any specific standard or recommendation for provenance representation.

Kazemitabaar *et al.* (2022) propose a framework designed to support children in transitioning from block-based programming to text-based programming. The framework collects metadata from programming sessions and attempts to learn from successful cases to assist new students in learning.

Ribeiro dos Santos *et al.* (2023) propose a Provenance Model for Open Educational Resources (OER) to enable the documentation and identification of the provenance of OER. It is the only approach that explicitly leverages provenance data to evaluate educational aspects, but its primary focus is on a coarse-grained evaluation of educational resources rather than analysing students' learning trajectories.

Tsung *et al.* (2022) propose a visual analytics system designed to help teachers analyse students' block-based coding behaviours, identify mistakes, and detect learning bottlenecks. The system collects a range of metadata to support this analysis and was evaluated through a study involving K-12 students. It brings a valuable study about students' learning processes, but it does not follow a specific standard or recommendation for provenance representation.

Pereira *et al.* (2024) propose NextBlocks, a Moodle plug-to capture metadata of the learning process, providing valuable insights into student performance. Although NextBlocks does not explicitly mention that it uses provenance data, it records a range of metadata commonly found in provenance databases, which facilitates analysis of student progress. Furthermore, its seamless integration with Moodle simplifies data access and analysis.

Pozzan *et al.* (2024) capture a wide range of metadata extracted from sequential program snapshots, tracing the evolution of students' learning over time. The proposed approach records metadata at a fine-grained level, similar to `EduPROV`. However, it does not follow any standard or recommendation for representing this metadata or incorporating a database for data storage and querying. This absence of structured storage makes the analytical process difficult.

Kong *et al.* (2024) conduct an exploratory study involving teachers using Scratch to identify key factors that researchers should consider when exploring design opportunities for programming classes targeted at kids. Although the approach presented in this paper does not introduce a new system or software library, it highlights the importance of analysing metadata to understand the teaching process better and enhance its effectiveness.

Torre *et al.* (2024) present a literature review highlighting some aspects of analysing learner behaviour using sequential data and analysis. They conclude that the main purpose of sequence analysis is to extract patterns from the data and identify the real-world behaviours they describe. They found several techniques to analyse the captured data, ranging from manual analysis of logs to deep learning models capable of predicting the learner´s performance.

Heath *et al.* (2025) propose an extension of the Block Model designed for use in block-based programming by teachers who are not experts in computer science. The proposed model presents an interesting framework that enables teachers to analyse students' learning behaviour through metadata collection. However, these metadata do not explicitly represent the data derivation process or follow standards or recommendations.

4. `EduPROV`: an Approach to Capture and Analyse Provenance Data from Block-based Programming Frameworks

As mentioned in Section 1, `EduPROV` is designed to collect, store, and analyse provenance data from block-based programming frameworks and platforms to gain insights into students' programming learning processes. This section provides an overview of `EduPROV`, detailing its architecture and provenance model. The architecture of `EduPROV` is depicted in Fig. 2 and consists of six components: (i) Log Extractor, (ii) Provenance Extractor, (iii) Query Processor, (iv) Graph Exporter, (v) Local Storage, and (vi) Provenance Database. Each component is important in collecting, processing, and analysing provenance data, ensuring a structured approach to understanding students' programming learning processes.

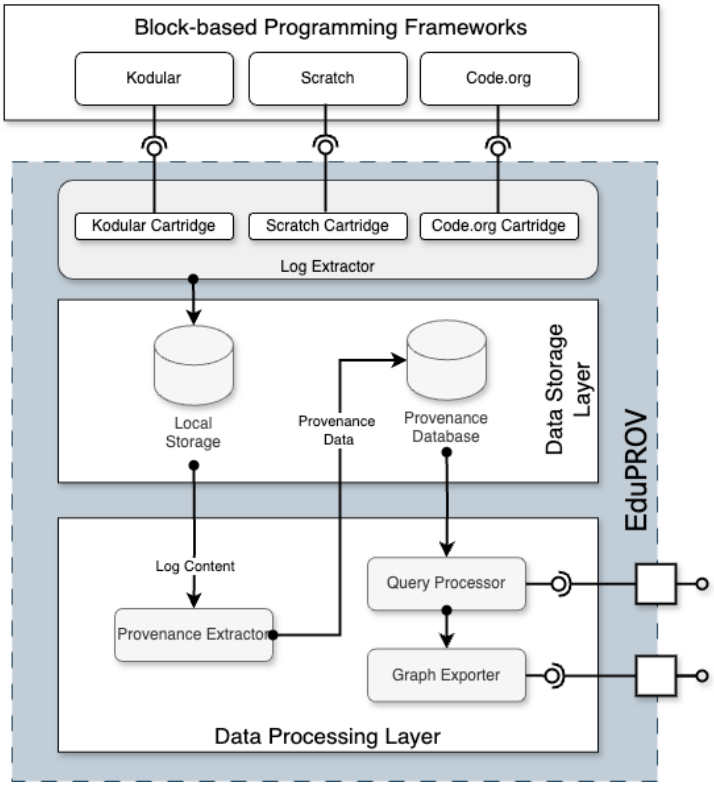


Fig. 2. The architecture of EduPROV, comprising six main components: (i) Log Extractor, (ii) Provenance Extractor, (iii) Query Processor, (iv) Graph Exporter, (v) Local Storage, (vi) Provenance Database.

The *Log Extractor* retrieves logs from block-based programming frameworks. Since different frameworks provide logs in multiple formats, *e.g.*, some through APIs and others as compressed files containing various data types, EduPROV uses the concept of a *Cartridge* to handle log file imports. A cartridge is a modular component that can be dynamically adapted (Birsan, 2005) based on the specific requirements of each framework. Therefore, a corresponding cartridge must be implemented to perform log extraction for every new framework. In the current version of EduPROV, the *Log Extractor* only includes a cartridge designed for the Kodular framework (accessing .aia files).

Another key feature of the *Log Extractor* is that it has to access and download log files from the framework periodically. EduPROV executes under the premise that log files are collected from multiple *application snapshots* (*i.e.*, versions of the application) over time, as presented in Fig. 3. This approach enables the identification of actions and modifications made by students within their applications. In the experiments detailed in Section 5, the *Log Extractor* retrieves log files at the end of each programming session (typically lasting 30–40 minutes). However, this frequency is not mandatory, as users can define how often log files should be downloaded based on their specific needs.

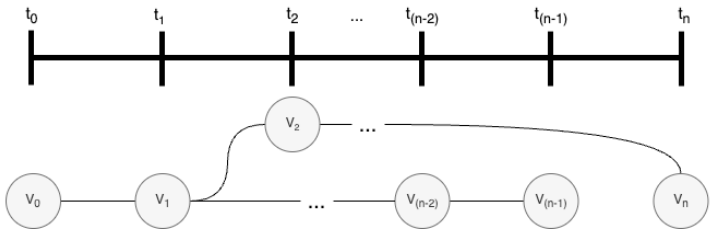


Fig. 3. Multiple application snapshots are generated over time, each representing a different version of the same application throughout development. Given that students are still in the learning phase, their development process is often non-linear. It means that some versions may be revised, modified, or even discarded as they experiment, refine their understanding, and improve their code.

Once the *Log Extractor* retrieves log files from the block-based framework, it stores their content in the *Local Storage* component of `EduPROV`. `EduPROV` adopts the Medallion Architecture[5], a well-established data design pattern that eases the logical organization of data. This architecture enables a progressive refinement of data quality and structure as it transitions through multiple stages of processing and storage. In the Medallion Architecture, the *Local Storage* component corresponds to what is referred to as the "Bronze Layer". This layer stores raw, unprocessed data from external sources, preserving its original format. The *Local Storage* is a foundational data repository, ensuring that all collected information is retained for further transformation and analysis. By maintaining an unaltered version of the logs, `EduPROV` ensures data traceability and allows for reprocessing or re-extraction, if necessary.

With all raw data stored in the *Local Storage*, the *Provenance Extractor* is activated. This component plays a key role in `EduPROV`, as it analyses and compares log files from consecutive programming sessions to identify differences (Δ), which correspond to the specific actions performed by students during each session. The *Provenance Extractor* analyses the logs of each application snapshot to detect changes at a granular level. It identifies visual components, code blocks, and other components present in one application snapshot but absent in the previous version of the same application. By doing so, it infers the nature of the modifications made by the student, classifying them into four categories: `ADD` (when new elements are introduced into the application), `REPLACE` (when the student modifies an element's values or properties), `COPY` (when elements are copied in the application), and `REMOVE` (when an element is deleted from the application). This detailed comparison provides insights into students' programming behaviour and learning trajectories, facilitating the reconstruction of the development process over time.

Let us consider an example of the difference analysis performed by the *Provenance Extractor* using the log files of two application snapshots, as presented in Fig. 4. In the initial version of the application, only the component `Label1`, containing the text "Text for Label1", was added within the `Screen1` component. In the subsequent version, a new component, `Button1`, was added, along with two additional properties for

[5] `https://www.databricks.com/glossary/medallion-architecture`

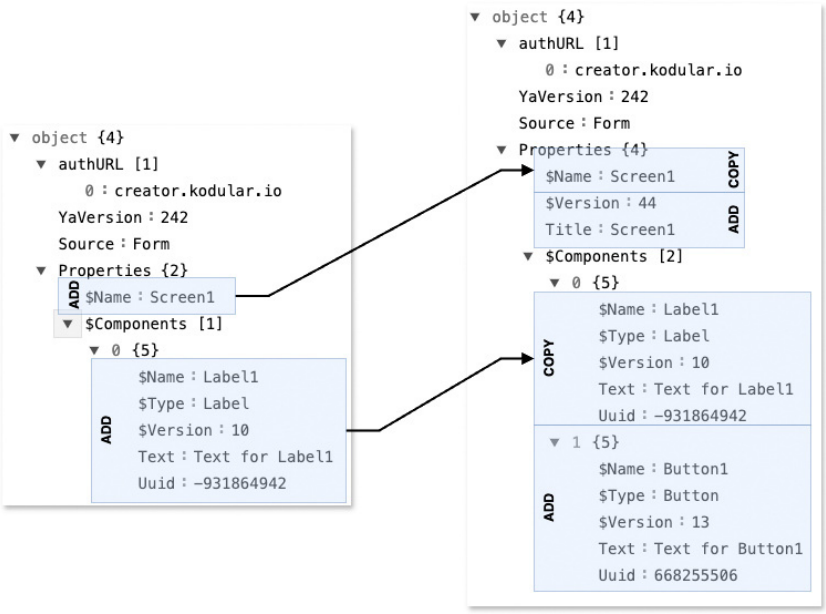


Fig. 4. Example of the difference analysis performed by the *Provenance Extractor*. The first version contains only the `Label1` component within `Screen1`, while the second version introduces a new component, `Button1`, and additional properties for `Screen1` (`$Version` and `Title`).

`Screen1`: `$Version` and `Title`. Consequently, in this example, the *Provenance Extractor* identifies 13 `ADD` activities corresponding to the newly introduced elements and seven `COPY` activities for the elements that were copied from one version of the application to the other. It is worth noticing that the *Provenance Extractor* works on the fine-grain level, *i.e.*, it compares elements and their properties.

The *Provenance Extractor* follows the procedures defined in Algorithm 1 to analyse differences between the log files of two application snapshots. This component takes as input two log files, denoted as sn_{v1} and sn_{v2}, which represent sequential versions of an application over time. Initially, it extracts a list of all elements in both logs, such as buttons, labels, and code blocks, and retrieves information about the user responsible for the modifications (line 1). Once the relevant data is collected, the *Provenance Extractor* iterates through all elements across both application snapshots (lines 2 to 52) to determine how each element and its associated properties have changed (or not). A new component has been added to the program if an element appears only in the newer snapshot sn_{v2} (lines 3 to 13). Consequently, the algorithm registers the newly added element (line 4), creates an `ADD` activity (line 8), and processes each of the element's properties accordingly (line 7). Furthermore, the relationships between the properties, elements, and the activity are explicitly documented in compliance with the PROV recommendation (lines 9 to 11).

On the other hand, if an element is found exclusively in the older snapshot sn_{v1} and is absent in sn_{v2}, it means that the element has been removed from the applica-

Algorithm 1: Provenance Extractor(sn_{v1},sn_{v2})

```
1  K_v1, K_v2, u ← getElementLog(sn_v1), getElementLog(sn_v2), NewUser(getUserLog(sn_v2));
2  foreach k ∈ (K_v1 ∪ K_v2) do
3      if k ∈ K_v2 and k ∉ K_v1 then
4          e ← newProvElement(k);
5          P ← getPropertiesLog(k, sn_v2) ;
6          foreach p ∈ P do
7              prop ← newProvProperty(p);
8              a ← newProvActivity(name : concat('ADD', prop.name));
9              wasAssociatedWith(a, u) ;
10             wasGeneratedBy(prop, a);
11             hadMember(e, prop);
12         end
13     end
14     if k ∉ K_v2 and k ∈ K_v1 then
15         e ← getProvElement(k);
16         P ← getPropertiesLog(k, sn_v1) ;
17         foreach p ∈ P do
18             prop ← getProvProperty(p);
19             a ← newProvActivity(name : concat('REMOVE', prop.name));
20             wasAssociatedWith(a, u) ;
21             used(prop, a);
22             hadMember(e, prop);
23         end
24     end
25     if k ∈ K_v2 and k ∈ K_v1 then
26         e_old ← getProvElement(k, sn_v1);
27         e_new ← newProvElement(k, sn_v2);
28         wasDerivedFrom(e_new, e_old);
29         P_s2 ← getPropertiesLog(k, sn_v2) ;
30         foreach p_2 ∈ P_s2 do
31             p_1 ← value(p_2, k, sn_v1);
32             if p_1 ≠ ∅ then
33                 if value(p_1) ≠ value(p_2) then
34                     a ← newProvActivity(name : concat('REPLACE' =, p2.name));
35                 end
36                 if value(p_1) ≡ value(p_2) then
37                     a ← newProvActivity(name : concat('COPY', p2.name));
38                 end
39                 wasAssociatedWith(a, u) ;
40                 used(p1, a);
41                 wasGeneratedBy(p2, a);
42                 hadMember(e_new, p2);
43             end
44             if p_1 = ∅ then
45                 a ← newProvActivity(name : concat('ADD', p2.name));
46                 wasAssociatedWith(a, u) ;
47                 wasGeneratedBy(p2, a);
48                 hadMember(e_new, p2);
49             end
50         end
51     end
52 end
```

tion snapshot. In this case, the algorithm retrieves the missing element (line 15) and its associated properties (line 16). Subsequently, for each removed property, a `REMOVE` activity is created (line 19) and linked to the corresponding properties, the deleted element, and the user responsible for the action (lines 20 to 22). For elements in both application snapshots, the algorithm performs a more detailed analysis to determine the type of change that has occurred. If the element remains unchanged, it is categorized as a `COPY` activity (line 37). If one or more property values have been modified, the algorithm registers a `REPLACE` activity to document the update (line 34). Additionally, if a new property has been introduced to an existing element, a corresponding `ADD` activity is recorded (line 45).

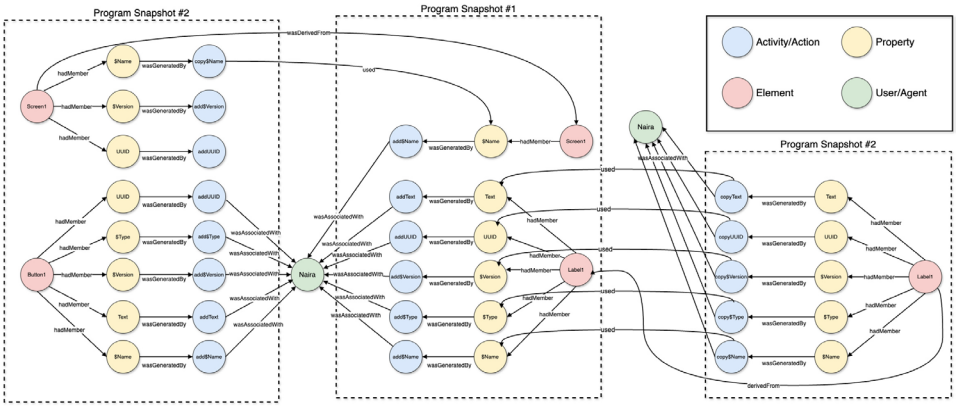


Fig. 5. The conceptual representation of the provenance graph extracted from the log files presented in Fig. 4. The user node is duplicated in the graph solely for visualization purposes to enhance readability.

Upon completing the processing phase, the *Provenance Extractor* produces a structured representation of provenance data associated with the analysed application snapshots. Fig. 5 presents a conceptual representation of the identified elements, properties, activities, and users extracted by the *Provenance Extractor*, based on the two log files presented in Fig. 4. A key feature of `EduPROV` is its ability to capture provenance data at a fine-grained level, specifically at the property level. This granularity results in the generation of large-scale provenance graphs. For instance, in the graph depicted in Fig. 5, only two application snapshots are considered, containing a limited number of elements and properties, *i.e.*, `Screen`, `Label`, and `Button`, with no code blocks. Despite this small dataset, the resulting provenance graph comprises 44 nodes.

Given the size of the provenance graph generated by `EduPROV`, an efficient storage solution is required to enable effective querying and analysis. Although various types of Database Management Systems (DBMS) could be employed for this purpose, Neo4j[6] was chosen as the storage solution due to its graph-based structure. Using Neo4j, `EduPROV` avoids additional data transformations, as the output of the *Provenance Extractor* is already formatted as a graph, ensuring seamless integration. Thus, the *Provenance Database* is responsible for storing all extracted provenance data from the log files within the Neo4j graph database. In Neo4j, `EduPROV` maintains the same graph structure as depicted in Fig. 5, consisting of four nodes and five types of edges. We have chosen to adopt this model instead of the original PROV representation, which exclusively employs `Entity`, `Activity`, and `Agent` nodes. The primary reason for this decision is that our model offers richer semantics, enabling users to perform more meaningful queries. In PROV-DM, elements and their associated properties are both mapped to `Entity` nodes, which can lead to ambiguity, as users may struggle to distinguish between structural program components (*e.g.*, buttons, labels) and their attributes (*e.g.*, text, title). To populate the provenance database in Neo4J, we

[6] `https://neo4j.com/`

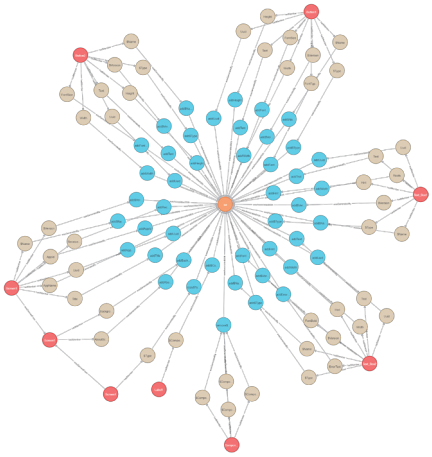


Fig. 6. An example of the provenance graph generated by EduPROV in Neo4J.
In the graph, orange nodes represent Agents/Users, blue nodes correspond to Activities, red nodes denote Elements, and brown nodes represent Properties.

have to use the Cypher query language to generate one or multiple nodes within the provenance graph. Each node can be assigned a variable, labels, and a set of properties.

Since all provenance data is stored in the *Provenance Database*, the task of retrieving relevant data is delegated to the *Query Processor* component. This component serves as the proxy between the user and the underlying storage system. Specifically, the *Query Processor* accepts queries written in `Cypher` and submits them to the graph database, and subsequently returns the corresponding results to the user. At its current stage of development, the `EduPROV` system requires users to possess a prior understanding of the graph schema and its structural organization to construct queries.

Finally, the *Provenance Exporter* is responsible for accessing the *Provenance Database* and transforming the stored provenance data into a representation that follows the PROV recommendation. Given that the provenance model of `EduPROV` implemented in Neo4J is inspired by PROV-DM, mapping data to the standard PROV notation is straightforward. Specifically, users within `EduPROV` are represented as `Agent`, actions performed during programming sessions are classified as `Activity`, and both program elements and their associated properties are categorized as `Entity`. An example of the provenance graph previously depicted in Fig. 5, now transformed into the PROV-DM notation, is presented in Fig. 7. The source code of `EduPROV` will be made available for download in our research group's GitHub repository (`https://github.com/UFFeScience/EduPROV`), allowing researchers and practitioners to access, evaluate, and extend the proposed approach.

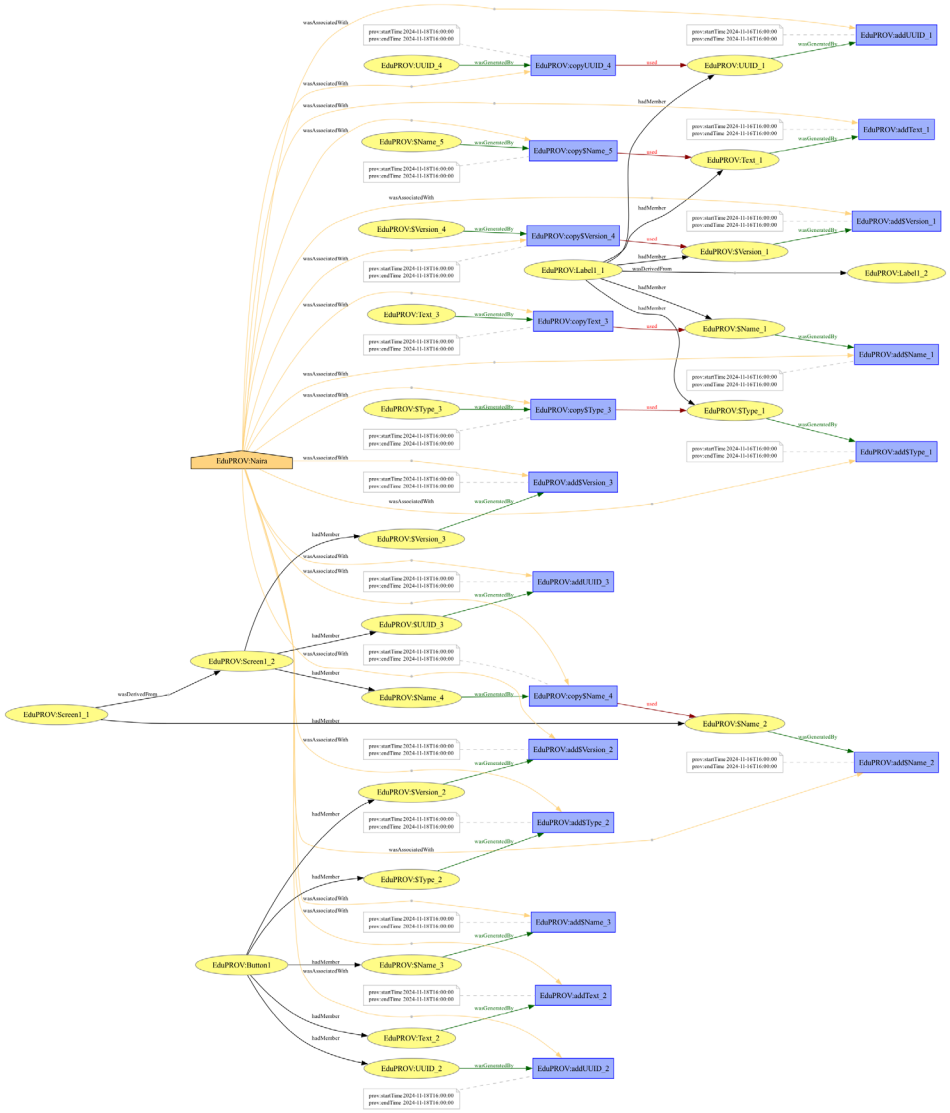


Fig. 7. An example of the provenance graph, conceptually illustrated in Fig. 5, generated by `EduPROV` using the PROV-DM notation.

5. Experimental Evaluation

This section presents an experimental evaluation of the `EduPROV` approach. The primary objective of this evaluation is to investigate the extent to which `EduPROV` can support analysing and interpreting students' learning trajectories during programming activities conducted in block-based programming environments. Specifically, the evaluation focuses on tracking individual student progress, identifying patterns of problem-

solving, and assessing engagement throughout the programming sessions. Accordingly, the following research question is posed: "How effectively can EduPROV facilitate the understanding and analysis of students' learning journeys in block-based programming environments, and to what extent does its use contribute to informed, evidence-based pedagogical interventions?".

This section begins by describing the experimental protocol used to evaluate EduPROV (Section 5.1). It then presents the profile of the participants (Section 5.2) and outlines the key metrics selected by teachers (Section 5.3) to evaluate student performance. Finally, the section demonstrates how EduPROV facilitates the analysis of students' learning trajectories (Section 5.4), providing actionable insights into their development process in block-based programming environments.

5.1. *Evaluation Protocol*

The experimental protocol followed in this manuscript is presented in Fig. 8. It is worth mentioning that the evaluation focused on showing how EduPROV can support users in analysing the learning journeys of students. The procedure begins with a 15-minute introductory session, during which students receive an explanation of the expected outcomes. Specifically, each student gets the task of developing a mobile application that calculates the area of three geometric shapes: (i) a rectangle, (ii) a triangle, and (iii) a circle. This task was intentionally designed to reinforce concepts already covered in their mathematics classes.

During the introductory session, general guidelines regarding the required components are provided to the students, as well as the expected user interface for the application. However, no step-by-step instructions or predefined solutions are provided. Following this initial briefing, students participate in four 30-minute programming sessions, conducted over four consecutive days.

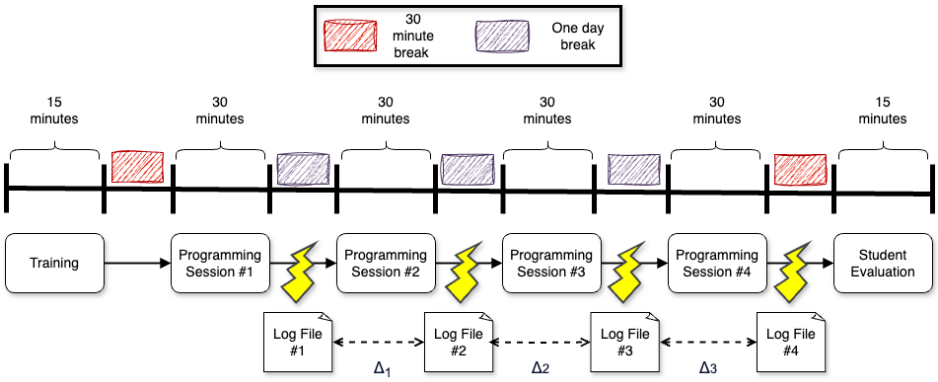


Fig. 8. The timeline of the experimental protocol followed in this manuscript. The process consists of a 15-minute introductory session and four 30-minute programming sessions, all conducted over four consecutive days. After each session, EduPROV collects, and stores log files capturing their development process, which are later analysed to evaluate programming behaviour and problem-solving strategies.

Throughout these sessions, students work using the Kodular framework without any intervention from the teacher or the study monitor. Importantly, no prior examples of similar applications are provided, nor are students informed in advance about the total number of programming sessions allocated for the task. This approach evaluates the student's ability to develop a functional application as well as their problem-solving strategies, programming behaviours, and decision-making processes throughout the development cycle.

As presented in Fig. 8, `EduPROV` collects and stores the log files generated during the student's interaction with the Kodular framework after each programming session. Therefore, each student produces four log files corresponding to the actions performed in the four programming sessions. These log files record the student's development process, capturing modifications, additions, and deletions made to the application over time.

It is important to emphasize that `EduPROV` does not capture and store provenance data continuously after each action performed by the student within Kodular. Instead, provenance is recorded only at the end of each programming session. The rationale behind this design choice lies in the computational overhead associated with capturing provenance at runtime in a fine-grained manner. Identifying and storing actions at runtime requires inserting provenance data into the database for every single operation executed by the user. This activity can easily amount to hundreds of actions within a single session, especially when the student is doing an *ad-hoc* exploration of Kodular. Prior research has demonstrated that such fine-grained provenance collection may introduce a non-negligible overhead (Wang *et al.*, 2015), which, in our context, could negatively impact both the performance of Kodular and, consequently, the overall students' experience.

To mitigate this issue, we adopt a session-level provenance capture strategy. While this strategy may not preserve the complete history of every action, thus potentially "losing" some exploratory steps, we guarantee that the final state of the application in Kodular accurately reflects the elements defined by the student as the "final" rather than temporary or *ad-hoc* interactions with the environment. This strategy is similar to the commit mechanism in version control systems (do Rego Pinto and Murta, 2023). In Git, for example, the system does not create a new version of the source code after every minor modification by the user. Instead, changes are consolidated and recorded only when the user issues a commit, at which point the new version is considered valid. Analogously, in our approach, provenance is consolidated at the conclusion of a session. It is worth noting, however, that the granularity of provenance capture remains flexible: the duration and frequency of sessions can be configured to allow for finer-grained provenance.

At the end of the experiment, students may provide feedback regarding any challenges they faced during the programming sessions. Additionally, they can suggest potential improvements or adjustments that they believe would improve the evaluation process in the future. This feedback is valuable for refining the experimental protocol and the applicability of `EduPROV`.

5.2. *Participant Analysis*

To evaluate EduPROV, 87 students were selected from three schools in Brazil's southern region. Among these institutions, two are state-managed, *i.e.*, EEB Aparício Julio Farrapo[7] and EEF Augusto Colatto[8], while the third, EMEB Nossa Senhora Aparecida[9], is managed at the municipal level. These details are summarized in Table 2.

The selected schools were chosen based on their relatively low rankings in the Brazilian Basic Education Development Index (IDEB), where scores range from 0 to 10, with 10 representing the highest level of educational performance. Additionally, these institutions have limited access to programming education, making them interesting candidates for evaluating the learning progression of students in block-based programming environments. The selected students participated in the programming sessions integrated into their regular classroom activities.

It is worth noticing that their legal guardians authorized the participation of all students in the experiment. Before the study, an informed consent form was provided detailing the research goals, activities, and data collection procedures. Only students whose guardians granted explicit permission were included in the experiment, ensuring compliance with ethical research standards and data privacy regulations. A copy of the informed consent form used in this study is provided in EduPROV's Github repository.

All participants in this study were students enrolled in the 9th grade of Elementary School II, corresponding to the K–12 educational level. Before the experimental sessions, each participant completed an open-ended questionnaire designed to collect demographic and background information, including their age, prior experience with computing, and personal expectations regarding the study. A copy of this questionnaire is provided in EduPROV's GitHub repository for reference. The age range of the students varied between 14 and 17 years old, with the majority being either 14 or 15 years old, as presented in Fig. 9. The sample was composed of a nearly balanced gender distribution, with 47% of the participants identifying as female and 53% as male.

A relatively small percentage of the participants had prior experience in programming or any background in the field of computer science, as illustrated in Fig. 10.

Table 2
IDEB scores for selected schools categorized by management type (state-managed or city-managed)

School	Management Type	IDEB	Year
EEB Aparício Julio Farrapo	State-managed	5.6	2021
EMEB Nossa Senhora Aparecida	City-managed	5.2	2021
EEF Augusto Colatto	State-managed	6.0	2021

[7] https://qedu.org.br/escola/42083907-eeb-aparicio-julio-farrapo
[8] https://qedu.org.br/escola/42102707-eef-augusto-colatto
[9] https://qedu.org.br/escola/42083559-emeb-nossa-senhora-aparecida

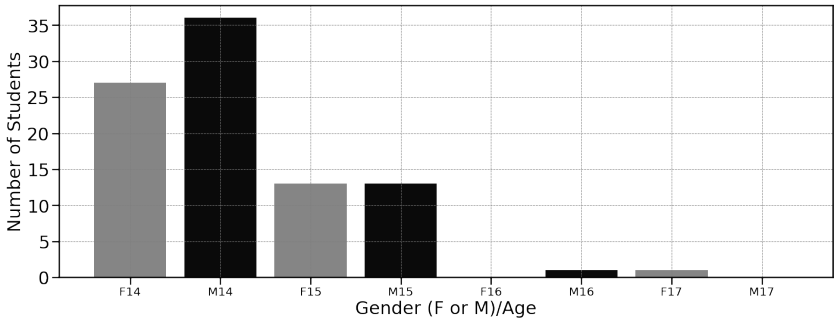


Fig. 9. Age and gender distribution of the experiment participants. The majority of participants were either 14 or 15 years old. The sample shows a nearly balanced gender distribution, with 47% identifying as female and 53% as male.

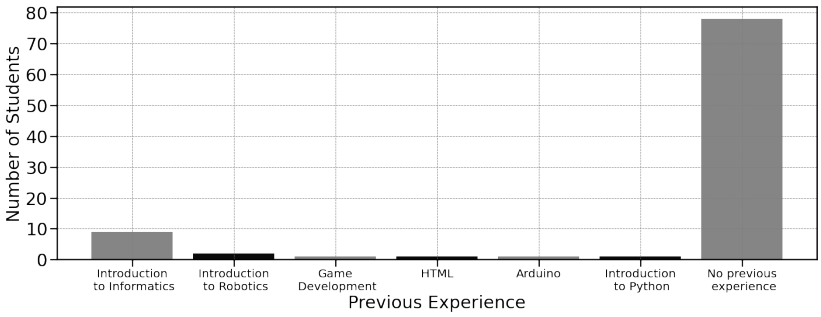


Fig. 10. Prior programming experience of study participants. A small percentage of students had previous experience with programming, mainly through introductory courses in Python, Robotics, or HTML.

Among those with previous experience, it was primarily limited to introductory courses in Python, Robotics, and HTML. An interesting aspect to discuss is that the students participating in this study had no prior experience with block-based programming environments, such as Scratch or Kodular, but were familiar with Python. This pattern can be attributed to the current state of programming education in Brazilian public schools, where structured initiatives for teaching programming are rarely supported through government-led policies. Instead, programming education often depends on the initiative and expertise of individual teachers, who seek to motivate students to engage with programming. Consequently, the specific programming language or environment introduced to students typically reflects the teacher's own background. In the context of this study, the students' prior exposure to Python illustrates how programming education in these schools is shaped mainly by the individual experiences and resources of educators, rather than by standardized curricular directives. This characteristic makes the selected group particularly relevant for evaluating `EduPROV`, as it allows for an analysis of how students with minimal prior knowledge engage with and learn block-based programming.

5.3. *Evaluation Metrics*

To conduct the analyses, the teachers from the three participating schools formulated some questions to be addressed using the provenance data provided by `EduPROV`. These questions aim to help them analyse how students develop a program and solve a problem. They outlined a list of key pieces of information that should be extracted from each student's programming session in `EduPROV`, including:

1. The type of element (*e.g.*, `Label`, `Button`, `Screen`, `Code`, *etc*.) that can be added, replaced, copied, or removed from the application.
2. Which elements and their properties are effectively added, replaced, copied, or deleted from the application.
3. The order in which the operations of adding, replacing, copying, and deleting elements took place.
4. The number of sessions it took for the student to finish the application.
5. Which user made each operation.

Based on the questions as mentioned earlier, we can establish a set of metrics to gain insights into the reasoning and decision-making processes involved in developing an application using block-based frameworks. Inspired by the metrics proposed by Feng *et al.* (2019) for analysing user interactions on web pages, we introduce several metrics that can be used to evaluate the application development process in block-based frameworks:

1. Types of elements that were effectively used in the application development.
2. Types of elements that were mandatory to be used in the application development (based on the explanations provided in the introductory session).
3. The number and list of elements and their properties placed in the block-based application by a student during each programming session.
4. The number and list of operations executed by a student during each programming session.
5. The path of operations performed to develop an application.
6. The number of programming sessions to develop the application.
7. The Exploration uniqueness refers to the use of unique elements in the application or the addition, replacement, copy, or deletion of elements in a unique order.

The metrics mentioned earlier evaluated whether the students successfully achieved the expected outcome, *i.e.*, the development of an application that calculates the area of a circle, a triangle, and a rectangle. Additionally, these metrics aimed to analyse how students reached (or failed to reach) this goal, providing insights into their problem-solving strategies and programming decisions.

To extract the defined metrics and explore the provenance graph of students' programming sessions, it is essential to formulate and execute various queries within the provenance database. These queries can be categorized into four distinct classes: (i) Class C1: Queries that retrieve entities, such as elements and properties, from a single provenance graph; (ii) Class C2: Queries that extract information related to actions within a single provenance graph, (iii) Class C3: Queries that generate statistical summaries, often us-

Table 3
Examples of provenance queries supported by `EduPROV`

Query ID	Description	Class
Q1	What are the elements a student uses to develop an application?	C1
Q2	What are the operations performed by a student to develop an application?	C2
Q3	Which operations did the student perform in a specific program session?	C2
Q4	What is the number of operations performed by user `[Student ID]` during application development?	C3
Q5	What is the number of sessions needed to develop an application?	C3
Q6	What is the number of operations performed to develop an application?	C3
Q7	What is the exploration uniqueness of user `[Student ID]`?	C4

ing aggregation functions to analyse patterns in the data, and (iv) Class C4: Queries that establish relationships between entity attributes and trace data derivation paths within the provenance graph. Table 3 presents a set of seven provenance queries designed based on the most common analyses conducted by teachers.

5.4. *Results Discussion*

Visually analysing the provenance graph to extract insights into students' program development within block-based frameworks is a tedious and error-prone process. This difficulty arises from the exponential growth of the provenance graph as students progress through their programming sessions, resulting in increasingly complex structures that are challenging to interpret visually. Fig. 11 presents the size of the provenance graph for student `ID54` over the course of just two programming sessions, highlighting the

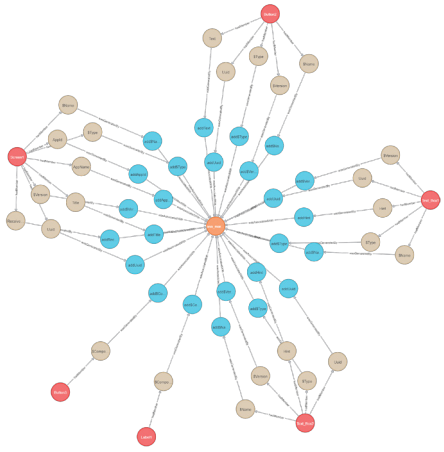


Fig. 11. Provenance graph of student `ID54` across two programming sessions. The graph captures the student's interactions, including modifications, additions, and deletions of program components.

rapid accumulation of nodes and relationships. As students continuously add, modify, and delete elements in the application, relying solely on visual inspection of the graph becomes impractical for accurately understanding development patterns and behaviours. Consequently, the integration of query mechanisms is fundamental, as they enable the systematic analysis and foster the structured interpretation of the provenance data. By using these mechanisms, teachers and users can uncover patterns in students' problem-solving strategies, engagement levels, and overall progression, which would otherwise remain obscured in the visual representation.

To evaluate the applicability of `EduPROV`, we adapted the queries outlined in Subsection 5.3 for Cypher language in Neo4J. These queries were submitted to the provenance database using the *Query Processor* component of `EduPROV`. The *Query Processor* streamlines the process of query submission and execution, as discussed in Section 4, enabling the extraction of relevant information for analysis. The rest of this section discusses the results and insights derived from executing all queries presented in Table 3. Although the Cypher queries in this experiment were manually formulated, it is expected that a LLM-based mechanism could be employed to translate natural language questions into Cypher queries automatically. Such an approach would ease the use of `EduPROV` by teachers who do not present prior expertise in databases, thereby broadening accessibility and usability, as discussed in detail in Section 6.

Table 4 summarizes the provenance graph's total number of nodes, *i.e.*, elements and their associated properties, that each student added, replaced, copied, or removed during the programming sessions. Additionally, it presents the total number of actions performed by each student, categorized into additions, removals, copies, and replacements. Furthermore, Table 4 details the number of occurrences of each element found within the students' applications and the percentage of mandatory elements successfully implemented. To evaluate student performance, we compared the results of each student with a "template solution" for the program. This "template solution" represents a possible provenance graph that captures the development process of the expected application. An expert in block-based programming created it as a reference for comparison. For each student, we compared the number of nodes and actions recorded in their provenance graph against the "template solution", allowing us to compute a normalized value that reflects their expected efficiency in constructing the application.

It is worth noticing that all developed applications were evaluated by teachers and assigned a grade. The grading scale consists of four categories: Excellent (E), Good (G), Average (A), and Poor (P). The closer a student's application is to the ideal solution, the higher the grade assigned. Table 4 visually distinguishes students based on their grades to enhance readability. Students who achieved an Excellent (E) grade are marked in dark blue, while those who received a Good (G) grade are highlighted in light blue. Students with solutions graded Poor (P) are marked in red. Solutions classified as Average (A) are not highlighted. By analysing Table 4, we observe that 39.08% of the students did not develop functional applications (grade P), meaning their implementations failed to compute the area of the given geometric shapes correctly. On the other hand, the majority of students (60.91%) successfully created a functional version of the application (grades A, G, and E).

Table 4

Summary of the elements added, removed, copied, and replaced, and operations performed during programming sessions

Student	Nodes (Elements and Properties)	Nodes Normalized	Actions	Actions Normalized	% of Mandatory Elements	# of Screen	# of Horizontal Arrangement	# of Label	# of TextBox	# of Button
EMEB Nossa Senhora Aparecida										
ID01	220	1.5	32	1.5	40.0%	1	0	0	3	7
ID02	149	1	21	1	40.0%	1	0	0	3	3
ID03	122	0.8	16	0.8	20.0%	1	0	1	2	4
ID04	85	0.6	14	0.7	60.0%	1	0	0	5	1
ID05	85	0.6	13	0.6	20.0%	1	0	0	1	3
ID06	120	0.8	19	0.9	60.0%	1	0	0	3	3
ID07	150	1	24	1.1	60.0%	1	0	1	3	3
ID08	113	0.7	16	0.8	40.0%	1	0	0	3	3
ID09	194	1.3	31	1.5	70.0%	1	0	0	4	3
ID10	144	1	18	0.8	60.0%	1	0	0	3	2
ID11	136	0.9	14	0.7	30.0%	1	0	2	2	1
ID12	219	1.4	30	1.4	60.0%	1	0	0	4	3
ID13	144	1	25	1.2	60.0%	1	0	1	2	4
ID14	111	0.7	13	0.6	20.0%	1	0	0	3	2
ID15	196	1.3	29	1.4	70.0%	1	0	0	4	4
ID16	226	1.5	30	1.4	60.0%	1	0	0	3	8
ID17	77	0.5	9	0.4	0.0%	1	0	1	0	1
ID18	95	0.6	15	0.7	40.0%	1	0	0	2	2
ID19	129	0.9	23	1.1	70.0%	1	0	0	4	3
ID20	145	1	13	0.6	30.0%	1	0	1	2	3
ID21	115	0.8	17	0.8	60.0%	1	0	1	2	3
ID22	170	1.1	27	1.3	70.0%	1	0	0	3	7
ID23	157	1	24	1.1	70.0%	1	0	0	4	3
ID24	131	0.9	12	0.6	30.0%	1	0	1	0	3
ID25	183	1.2	27	1.3	70.0%	2	0	1	3	4
ID26	361	2.4	45	2.1	70.0%	1	0	0	2	4
ID27	113	0.7	13	0.6	50.0%	1	0	1	2	2
ID28	147	1	26	1.2	70.0%	1	0	0	4	4
EEF Augusto Colatto										
ID29	204	1.3	33	1.6	40.0%	1	0	1	2	2
ID30	165	1.1	21	1	80.0%	1	0	2	2	4
ID31	186	1.2	25	1.2	70.0%	1	0	2	2	2
ID32	150	1	24	1.1	60.0%	1	0	1	3	2
ID33	160	1.1	23	1.1	80.0%	1	0	2	2	2
ID34	103	0.7	16	0.8	50.0%	1	0	1	2	2
ID35	133	0.9	19	0.9	50.0%	1	0	0	2	3
ID36	134	0.9	21	1	40.0%	1	0	1	3	2

Continued on next page

Table 4 – continued from previous page

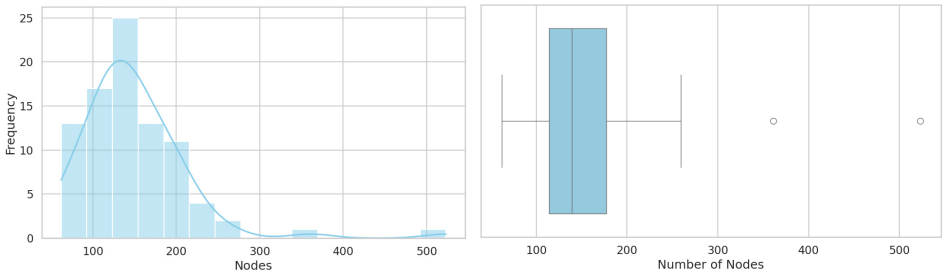
Student	Nodes (Elements and Properties)	Nodes Normalized	Actions	Actions Normalized	% of Mandatory Elements	# of Screen	# of Horizontal Arrangement	# of Label	# of TextBox	# of Button
ID37	186	1.2	24	1.1	70.0%	1	0	5	2	4
ID38	147	1	22	1	60.0%	1	1	1	2	3
ID39	124	0.8	22	1	60.0%	1	0	1	2	3
ID40	139	0.9	22	1	80.0%	1	0	2	2	2
ID41	162	1.1	22	1	50.0%	1	0	0	2	3
ID42	226	1.5	32	1.5	50.0%	1	0	1	2	2
ID43	116	0.8	16	0.8	60.0%	1	0	1	2	2
ID44	180	1.2	28	1.3	70.0%	1	0	1	2	3
ID45	185	1.2	25	1.2	70.0%	1	1	2	2	4
ID46	203	1.3	24	1.1	60.0%	1	0	1	2	3
ID47	199	1.3	35	1.6	80.0%	1	0	2	2	4
ID48	145	1	21	1	60.0%	1	0	1	2	3
ID49	259	1.7	28	1.3	30.0%	1	0	3	2	7
ID50	176	1.2	23	1.1	50.0%	1	0	0	2	3
ID51	74	0.5	8	0.4	20.0%	1	0	1	2	3
ID52	201	1.3	31	1.5	50.0%	1	0	1	2	2
ID53	66	0.4	7	0.3	20.0%	1	0	1	2	2
ID54	63	0.4	7	0.3	20.0%	1	0	1	2	2
ID55	148	1	14	0.7	50.0%	2	0	1	2	2
ID56	128	0.8	16	0.8	50.0%	2	0	1	2	2
ID57	109	0.7	12	0.6	20.0%	2	0	1	2	2
ID58	203	1.3	26	1.2	30.0%	2	0	1	3	4
ID59	202	1.3	19	0.9	50.0%	2	0	2	4	3
ID60	133	0.9	23	1.1	50.0%	1	0	1	2	2
ID61	158	1	22	1	70.0%	2	0	2	3	3
ID62	125	0.8	15	0.7	60.0%	2	0	1	2	2
ID63	179	1.2	21	1	60.0%	2	0	1	2	2
ID64	124	0.8	16	0.8	70.0%	2	0	1	2	2
ID65	112	0.7	13	0.6	60.0%	2	0	1	2	2
ID66	258	1.7	33	1.6	90.0%	2	8	2	6	4
ID67	170	1.1	23	1.1	70.0%	2	0	2	3	3
ID68	90	0.6	14	0.7	70.0%	2	0	2	2	4
ID69	90	0.7	7	0.3	50.0%	1	0	1	2	2
ID70	112	0.7	12	0.6	20.0%	1	0	1	3	4
ID71	89	0.6	15	0.7	60.0%	1	0	1	2	3
EEB Aparķcio Julio Farrapo										
ID72	104	0.7	10	0.5	30.0%	1	0	0	2	2
ID73	119	0.8	10	0.5	0.0%	1	0	0	2	1
ID74	167	1.1	22	1	50.0%	1	0	1	2	3
ID75	138	0.9	18	0.8	50.0%	1	0	0	2	2
ID76	131	0.9	20	0.9	70.0%	1	0	0	4	2

Continued on next page

Table 4 – continued from previous page

Student	Nodes (Elements and Properties)	Nodes Normalized	Actions	Actions Normalized	% of Mandatory Elements	# of Screen	# of Horizontal Arrangement	# of Label	# of TextBox	# of Button
ID77	62	0.4	7	0.3	10.0%	1	0	0	2	1
ID78	118	0.8	14	0.7	10.0%	1	0	0	1	2
ID79	169	1.1	28	1.3	70.0%	1	2	0	3	2
ID80	65	0.4	9	0.4	20.0%	1	0	0	2	1
ID81	125	0.8	10	0.5	30.0%	1	0	0	2	3
ID82	523	3.5	87	4.1	40.0%	2	0	0	10	6
ID83	77	0.5	7	0.3	20.0%	1	0	0	2	1
ID84	118	0.8	11	0.5	30.0%	1	0	0	1	3
ID85	150	1	22	1	80.0%	1	3	3	3	2
ID86	69	0.5	8	0.4	0.0%	1	0	0	2	0
ID87	115	0.8	20	0.9	80.0%	1	0	2	3	3

We generated some statistics using the results presented in Table 4 that can help in the analysis. Fig. 12(a) and Fig. 12(b) present the distribution of the number of nodes, *i.e.*, both visual elements and associated properties, that were added, removed, or modified by students during their programming sessions. From these distributions, one can state that most students managed between 100 and 200 nodes during their sessions, indicating a generally high level of engagement and active participation in the task. Only a small subset of students managed fewer than 100 nodes. Interestingly, among these students, only three (`ID68`, `ID69`, and `ID71`) were able to develop applications of satisfactory quality, suggesting that a minimal level of interaction with the programming environment may limit the potential for developing successful applications. On the other hand, a few students (*e.g.*, `ID26` added 361 elements), outliers in the distribution, managed more than 200 nodes, representing a higher-than-average level of interaction with the system. It is worth noticing that this group included the sole student who developed an



(a) (b)

Fig. 12. (a) Distribution of the number of nodes added, removed, or modified during programming sessions (b) Boxplot with the number of nodes added, removed, or modified during programming sessions.

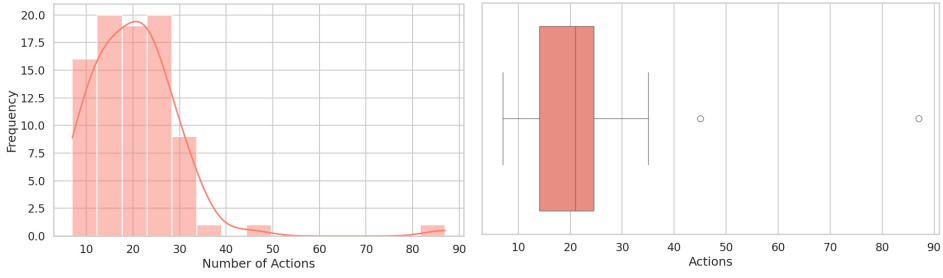
excellent application (ID66). This observation suggests a positive correlation between the intensity of engagement, as measured by node-level interactions, and the quality of the resulting application.

With respect to the number of actions, Fig. 13(a) and Fig. 13(b) show the distributions of actions performed per student during the programming sessions. Similar to the analyses made regarding the number of nodes, the majority of students concentrated their activity within a range of 10 to 30 actions. This suggests a balanced level of interaction with the programming environment, consistent with engagement in the development task. A small subset of students performed fewer than 10 actions. Among them, only student ID69 was able to produce an application of acceptable quality, whereas the remaining students in this group developed applications of poor quality. This highlights that insufficient interaction with the environment generally hampers the ability to construct functional or meaningful applications, although there may be outlier cases.

On the other hand, students who performed more than 30 actions also tended to produce low-quality applications. This finding suggests that a high volume of actions does not necessarily reflect productive engagement. Instead, it shows that these students may not have followed a structured plan for their development. Their activity appears to have been largely exploratory and *ad-hoc*, characterized by frequent additions, deletions, and modifications of nodes without a clear design strategy. Such behaviour points to trial-and-error programming rather than systematic problem-solving, which, in turn, can hinder the development of coherent and practical applications.

A deeper analysis of the provenance graphs for each student also shed further light on the findings aforementioned. The graphs revealed significant differences in the ways students approached programming tasks. For example, Student ID66 actively explored multiple strategies for implementing and optimizing the application, refining the solution iteratively over time. In contrast, Student ID26 appeared to work in a more unstructured manner, frequently adding elements at random and modifying arbitrary parameters (such as repeatedly adjusting the width of a label) without paying attention to the aaplication's core functionality.

Although the number of nodes and actions does not directly determine a student's performance, an interesting pattern emerged from the analysis. All students who achieved



(a) (b)

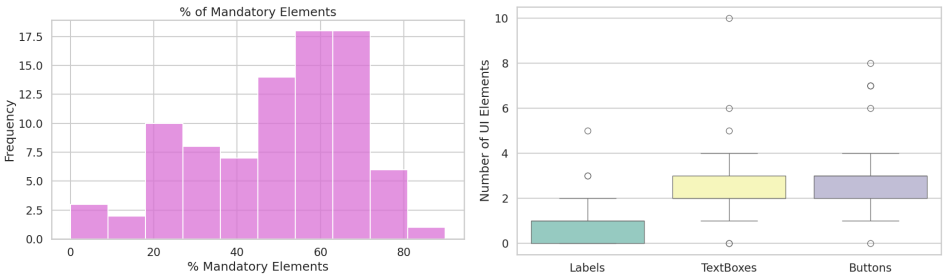
Fig. 13. (a) Distribution of the number of actions performed during programming sessions (b) Boxplot with the number of actions performed during programming sessions.

higher evaluation grades, i.e., classified as Good (G) or Excellent (E), added at least 90 nodes to their programs. This suggests that while the number of nodes and actions cannot be treated as a definitive metric for measuring success, there appears to be a threshold of engagement. In particular, a minimum level of experimentation and active manipulation of interface components may be positively associated with the production of higher-quality solutions. Furthermore, as presented in Fig. 14(a), most students effectively incorporated the mandatory elements required for the assignments, with the majority managing between 40 and 60 of these elements. This consistency indicates that students generally understood and complied with the essential structural requirements of their tasks.

Another layer of analysis, made possible through querying the provenance graph, concerns the identification of the number and type of visual elements added by each student. As shown in Fig. 14(b), and except for a few outliers, the number of different visual elements used by students typically did not exceed four. This observation highlights that the bulk of the nodes reported in Fig. 12 were not entirely new visual elements but rather properties whose values were repeatedly modified during programming sessions. This behaviour is consistent with the iterative nature of programming tasks, where students often fine-tune parameters and adjust existing elements to achieve desired outcomes.

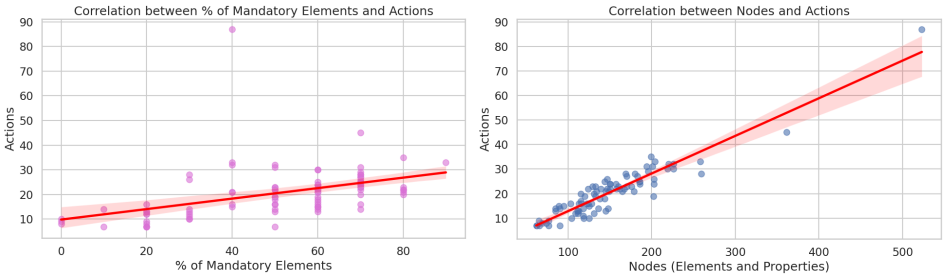
We also analysed the correlations between two different aspects of student activity: the percentage of managed mandatory nodes and the number of actions performed (Fig. 15(b)), as well as the correlation between the total number of nodes and the number of actions (Fig. 15(a)). The results provide insights into the relationship between students' programming behaviours and their outcomes. As presented in Fig. 15(a), there is a clear correlation between the total number of nodes and the number of actions performed. This finding is expected, given that every action, whether adding, modifying, or deleting elements and their properties, directly contributes to changes in the overall node count. In other words, the more actions a student performs, the greater the number of nodes that are created, modified, or removed within the application. This correlation highlights the natural dependency between these two variables.

However, when focusing exclusively on the mandatory nodes (Fig. 15(b)), the correlation with the number of actions becomes weaker. This weaker relationship suggests



(a) (b)

Fig. 14. (a) Distribution of the number of mandatory nodes and (b) the number of visual elements per type added, removed or modified by students.



(a) (b)

Fig. 15. (a) Correlation between the number of nodes and number of actions
(b) Correlation between the number of actions and the number of mandatory nodes.

that simply performing a large number of actions, or conversely, very few actions, does not necessarily lead to the effective management of mandatory nodes. In practical terms, a student may be highly active in modifying and experimenting with non-mandatory features while neglecting essential requirements, or may complete the mandatory nodes with relatively few actions. This observation underscores the importance of distinguishing between the quantity of engagement (total actions and nodes) and the quality of engagement (focused management of mandatory nodes).

Complementing the results presented in Table 4, Table 5 provides information regarding the number of nodes added by students across individual programming sessions. While the earlier analysis did not reveal a direct correlation between the overall number of nodes or actions and student performance, examining these metrics at the session level may introduce another perspective. By observing how students progressively add elements over time, teachers can gain insights into their programming practices. For example, a steady increase in the number of nodes across sessions may reflect systematic development and iterative refinement of the application. In contrast, abrupt fluctuations could reveal exploratory or trial-and-error behaviours. This session-based view, therefore, may enable teachers to better distinguish between students who are making meaningful progress with their solutions and those who may be engaging in unstructured experimentation.

By analysing Table 5, one can state that except for student `ID09`, all students who received grades E (Excellent) and G (Good) actively engaged with the task until at least the third programming session, with most continuing their work until the final (fourth) session. This pattern suggests that students who remained engaged throughout the experiment were more likely to explore different solutions and refine their implementations until the last moment. In contrast, students who produced lower-quality programs (grade P) or abandoned the development process did so within the first or second session. This finding highlights the critical role of the first programming session in determining student engagement, whether a student continues working towards a functional program or disengages early in the process.

Another observation from Table 5 is that students who encountered difficulties in developing functional applications, and whose final implementations were of poor quality, often presented a recurring pattern of behaviour. Rather than systematically approaching the problem or engaging with the logical structure of the application, these

students tended to make *ad-hoc* modifications, such as randomly adding new elements or renaming existing components, without any clear connection to the underlying functionality of the application. In many cases, these students did not even attempt to implement application logic. Instead, their effort was concentrated almost exclusively on the visual layer of the application, focusing on the placement of visual components rather than advancing toward a functional solution.

Table 5

Summary of the number of elements added by students in each programming session

Student	# of Nodes in Session 1	# of Nodes in Session 2	# of Nodes in Session 3	# of Nodes in Session 4	# Session Started Coding
EMEB Nossa Senhora Aparecida					
ID01	97	7	-	-	1
ID02	72	0	0	-	1
ID03	57	1	-	-	1
ID04	34	4	-	-	1
ID05	40	0	-	-	1
ID06	53	3	-	-	1
ID07	68	3	-	-	2
ID08	54	0	-	-	1
ID09	93	1	-	-	1
ID10	68	1	-	-	1
ID11	57	7	-	-	1
ID12	94	9	-	-	1
ID13	50	13	-	-	1
ID14	50	2	-	-	1
ID15	91	3	-	-	1
ID16	92	14	-	-	1
ID17	36	-	-	-	1
ID18	45	0	-	-	1
ID19	62	0	-	-	1
ID20	66	3	4	4	1
ID21	55	-	-	-	0
ID22	75	5	-	-	1
ID23	76	0	-	-	1
ID24	62	2	-	-	1
ID25	75	14	-	-	1
ID26	67	46	46	-	1
ID27	54	0	-	-	1
ID28	71	0	-	-	1
EEF Augusto Colatto					
ID29	57	14	20	-	2
ID30	68	9	0	-	2
ID31	62	20	2	-	1

Continued on next page

Table 5 – continued from previous page

Student	# of Nodes in Session 1	# of Nodes in Session 2	# of Nodes in Session 3	# of Nodes in Session 4	# Session Started Coding
ID32	62	5	2	-	1
ID33	34	21	13	-	2
ID34	48	3	0	-	2
ID35	47	14	0	-	2
ID36	37	21	0	-	2
ID37	59	10	14	-	2
ID38	65	4	0	-	2
ID39	46	5	4	-	2
ID40	61	4	0	-	2
ID41	66	10	0	-	2
ID42	100	9	-	-	2
ID43	42	11	0	-	2
ID44	67	15	0	-	2
ID45	68	14	3	-	2
ID46	55	37	0	-	2
ID47	59	32	0	-	2
ID48	42	20	2	-	2
ID49	83	27	7	-	2
ID50	51	23	4	-	2
ID51	26	2	1	3	1
ID52	44	17	6	20	1
ID53	26	3	-	-	1
ID54	26	2	-	-	1
ID55	42	17	2	4	1
ID56	35	18	0	3	1
ID57	36	1	7	3	1
ID58	26	19	21	21	2
ID59	47	12	2	28	1
ID60	32	12	14	-	1
ID61	35	9	24	-	1
ID62	39	9	2	4	1
ID63	47	28	0	4	1
ID64	35	17	0	3	1
ID65	37	17	0	-	1
ID66	38	24	8	42	1
ID67	36	7	4	27	1
ID68	35	4	1	0	1
ID69	34	0	3	-	1
ID70	35	9	4	-	1
ID71	36	0	4	-	1
EEB Aparkcio Julio Farrapo					
ID72	48	1	-	-	1
ID73	54	2	0	-	1
ID74	62	11	3	-	1

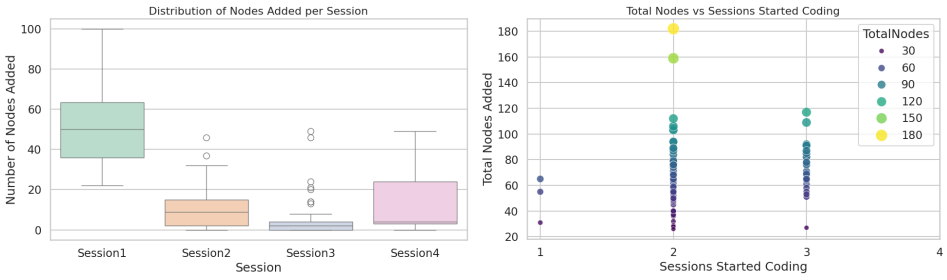
Continued on next page

Table 5 – continued from previous page

Student	# of Nodes in Session 1	# of Nodes in Session 2	# of Nodes in Session 3	# of Nodes in Session 4	# Session Started Coding
ID75	56	9	-	-	0
ID76	49	10	0	-	1
ID77	24	2	1	-	2
ID78	32	20	-	-	1
ID79	48	26	2	-	1
ID80	22	3	3	-	1
ID81	33	20	1	-	1
ID82	71	13	49	49	1
ID83	25	1	0	-	1
ID84	52	3	0	-	1
ID85	34	31	0	-	2
ID86	29	2	-	-	0
ID87	34	11	5	-	1

An analysis of Fig. 16(a) and Fig. 16(b) further reinforces this interpretation. The data reveal that the first session is the one in which students, on average, added the largest number of nodes. However, students who effectively began programming components during this initial session showed a different behaviour: they tended to add relatively fewer nodes, suggesting that their attention was directed toward the development of application logic rather than merely adding new elements. This contrast highlights a critical distinction between students who experiment only with the interface and those who engage in developing useful applications.

It is also important to note that the majority of students seem to dedicate the early stages of their interaction, particularly in the first session, to exploring the environment rather than coding. Evidence of this exploratory phase can be seen in the shift of activity between sessions: most coding typically began in sessions 2 and 3, during which the number of newly added nodes decreased. This reduction suggests a transition from exploratory behaviour toward more goal-directed coding activity. On the other hand, a small group of students delayed coding until session 4. Interestingly, these students did not add new nodes during that session, a behaviour that may indicate limited engage-



(a) (b)

Fig. 16. (a) Correlation between the number of nodes and number of actions
(b) Correlation between the number of actions and the number of mandatory nodes.

ment with the task. Taken together, these findings suggest that the timing and nature of student engagement in early sessions are critical predictors of eventual success.

The subsequent evaluation aimed to identify the uniqueness of exploration presented by individual students during their programming sessions. Specifically, it focused on detecting unique choices made by each student throughout their learning journey. To conduct this analysis, we selected the provenance graph of student `ID66`, as this student demonstrated the highest program quality among the 87 participants. It is important to mention that student `ID66` had no prior experience in programming. Due to this lack of background, the student initially adopted a trial-and-error approach, frequently adding and removing visual elements to evaluate their effects on the application's functionality. However, unlike random and unfocused actions, this iterative process was guided by a clear intent to refine and enhance the program logic.

As presented in Fig. 17, this exploratory behaviour led to a distinctive sub-graph that captures patterns of element deletions followed by subsequent re-additions (highlighted sub-graph in Fig. 17). These modifications included components such as buttons and their associated logic structures. This iterative cycle of adjustments, characterized by continuous refinement and testing, emerged as a unique characteristic among students with no prior computing experience. In contrast, students with previous experience in programming typically presented different interaction patterns, often following a more structured and linear development process.

Finally, aiming to uncover hidden correlations and provide teachers with insights to support students better, a clustering analysis was conducted based on data from the programming sessions generated by `EduPROV`. The K-Means algorithm (Han *et al.*, 2011) was chosen for this purpose. K-Means is a widely used clustering technique that partitions a dataset into different groups by assigning each object to the cluster whose centroid is closest to it. Instead of predefining the number of clusters (K), we

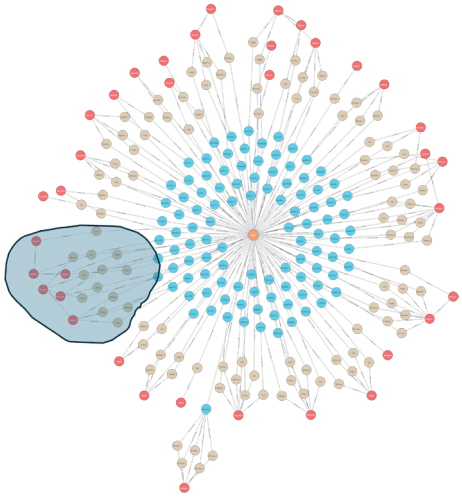
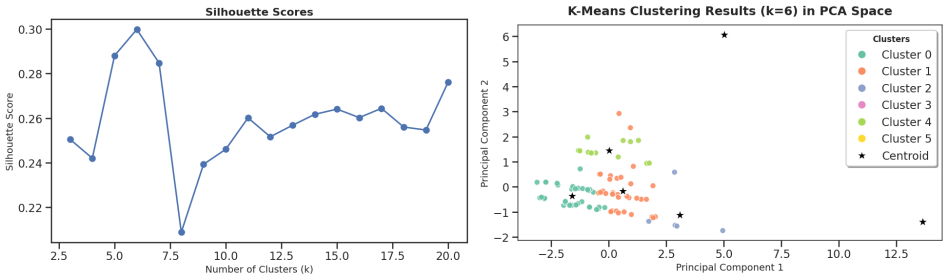


Fig. 17. Provenance graph of the student `ID66`, illustrating their unique exploration and iterative approach during programming sessions.



(a) (b)

Fig. 18. (a) Evolution of silhouette score for different cluster sizes
(b) Clusters generated by K-Means based on the features provided in Table 4 and Table 5.

used the scikit-learn (`https://scikit-learn.org/`) in a script that evaluates multiple possible values of K and selects the most suitable one based on the silhouette score, a widely used metric for assessing cluster cohesion. The clustering process was performed on features extracted from Table 4 and Table 5. To improve the quality of clustering and reduce data dimensionality, we applied Principal Component Analysis (PCA). PCA transforms the original dataset into a new set of orthogonal components, preserving as much variance as possible while reducing redundancy, thereby enhancing the interpretability of results.

Fig. 18(a) shows the evolution of the silhouette score for different values of K tested in our analysis. We evaluated $K \in [2, 20]$, and the optimal value identified was $K = 6$. Fig. 18(b) illustrates the resulting clusters using 2D scatterplots, where the axes correspond to Principal Component 1 (PC1) and Principal Component 2 (PC2). Consistent with the other analyses presented in this section, the clustering results neither grouped students according to the quality of their generated programs nor revealed a clear correlation. The first indication of this limitation was that the selected value of K did not align with the four predefined categories of student performance (excellent, good, average, and poor). This finding suggests that the clustering approach may not have been sensitive enough to capture the subtle patterns and complexities inherent in the provenance graph. Consequently, future work within `EduPROV` will investigate graph-based clustering techniques to enhance student classification and analysis.

5.5. *Limitations of the Experimental Results*

As an empirical research, the results presented in this manuscript present limitations due to the design choices of the experiment. In this subsection, we discuss the limitations of the experimental results.

The first limitation of the results is associated with the relatively small number of schools included in the experiment, as well as the fact that all schools are located within the same geographic region of the country. Although the quality of schools may vary considerably across different regions of Brazil, we selected schools based on their clas-

sification according to the IDEB, the standard evaluation framework provided by the Brazilian Federal Government, in an effort to mitigate regional disparities in school quality. With respect to the sample size, it is planned to extend the study to include additional schools in future investigations. Thus, the experimental results discussed in this section should be interpreted with caution, as their generalizability is limited due to these constraints.

Another limitation of the study is related to the number of programming sessions set for the experiment. In the current experiments, students participated in four programming sessions, each held over consecutive days. While this design enabled an initial evaluation of the `EduPROV` approach, it does not provide insights into long-term learning. To address this limitation, future studies are planned in which `EduPROV` will be evaluated over the course of an entire academic year, allowing for a long-term analysis of learning. Accordingly, the experimental results presented in this section should be interpreted with caution, as the limited duration of the experiment inherently constrains their generalizability.

The third limitation concerns the usability of `EduPROV` from the perspective of teachers. In its current version, effective use of `EduPROV` requires a background in computer science and databases. To mitigate this issue during the study, we provided direct support to teachers in conducting the analyses. Nevertheless, this dependency may restrict the types of queries that teachers can formulate and submit to the database on their own, thereby representing a potential barrier to the broader adoption of `EduPROV`. To overcome this challenge, we are integrating Large Language Models (LLMs) (Zhao *et al.*, 2023) into `EduPROV`. This will allow teachers to express their queries in natural language, after which the *Query Processor* will forward these inputs, using a proper prompt, along with the graph database schema, to the LLM. The LLM will then generate the corresponding `Cypher` query, which can be automatically submitted to the provenance database. This feature is currently under development and is designed to reduce the technical expertise required to use `EduPROV`, thereby lowering the barrier to entry for teachers.

6. Conclusion and Future Work

Integrating programming classes into school curricula has become a widely recognized need. Governments and non-governmental organizations worldwide are spending more and more resources and efforts to introduce programming skills to children as early as possible. The benefits of early programming education include fostering creative thinking, encouraging structured knowledge organization, and promoting the evaluation of the solution developed. By engaging in programming activities, students develop computational thinking and enhance their problem-solving abilities, which are fundamental skills in today's world.

While programming can be introduced through traditional text-based languages such as Python, Perl, C, and Java, block-based programming frameworks like Kodular and Scratch are the most common and practical approaches for children and teenagers. Despite the advantages offered by block-based programming, these frameworks lack tools

that allow teachers to analyse student learning processes. Teachers often face challenges in identifying students' specific difficulties, understanding the bottlenecks in their learning, and evaluating their progress. Although block-based programming environments generate log files containing students' interactions, these files are unstructured and require manual comparison across multiple sessions. This process is tedious, error-prone, and impractical for users who aim to monitor and support student learning.

This manuscript introduces `EduPROV`, an approach designed to bridge this gap, which is intended to extract provenance data from log files generated by block-based programming frameworks. The core objective of `EduPROV` is to capture and structure students' programming activities, including the elements they manipulate and their actions. This structured information is stored in a provenance database, which educators can query to support their analysis. By exploring provenance data, `EduPROV` provides teachers with insights into students' learning processes, enabling them to refine their teaching strategies and offer targeted support to students struggling with programming concepts.

To evaluate the applicability of `EduPROV`, we conducted a study involving 87 students from three different schools in Brazil. This study aimed to investigate how provenance data analysis can assist teachers in understanding students' programming behaviours and improving their instructional methodologies. The study's results demonstrated that analysing provenance data enables the identification of student behaviours that contribute to learning and recognizing critical programming sessions that play a key role in maintaining student engagement with programming tasks.

Future work will expand `EduPROV` in several ways. First, we plan to expand the number and diversity of participating schools beyond those chosen in the experimental evaluation. Although this manuscript focused on schools classified according to the IDEB framework in Brazil, a broader sample will allow us to better capture variations in school quality across different contexts. Second, the evaluation of `EduPROV` will be conducted over longer periods. While the current design already provides some insights, novel studies will follow students throughout an entire academic year, enabling an analysis of long-term learning effects. Finally, ongoing development is directed toward enhancing the usability of `EduPROV` for teachers. Future work includes integrating LLMs to allow teachers to formulate queries in natural language, which the system will then translate into `Cypher` and execute automatically. This improvement is expected to reduce barriers to adoption and facilitate the integration of `EduPROV` into everyday classroom practice. We also plan to explore advanced clustering techniques, including graph-based clustering, to classify students more effectively and uncover additional patterns in their learning behaviours hidden within the provenance graph structure.

Funding

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001. The authors would like to thank CNPq and FAPERJ.

References

Barbosa, C.H.S., Kunstmann, L.N.O., Silva, R.M., Alves, C.D.S., Silva, B.S., Filho, D.M.S., Mattoso, M., Rochinha, F.A., Coutinho, A.L.G.A. (2020). A workflow for seismic imaging with quantified uncertainty. *Computers & Geosciences*, 145, 104615. https://doi.org/10.1016/j.cageo.2020.104615

Belhajjame, K., B'Far, R., Cheney, J., Coppens, S., Cresswell, S., Gil, Y., Groth, P., Klyne, G., Lebo, T., McCusker, J., *et al.*(2013). Prov-dm: The prov data model. *W3C Recommendation*, 14, 15–16.

Birsan, D. (2005). On Plug-ins and Extensible Architectures: Extensible application architectures such as Eclipse offer many advantages, but one must be careful to avoid "plug-in hell.". *Queue*, 3(2), 40–46. https://doi.org/10.1145/1053331.1053345.

de Oliveira, D., Silva, V., Mattoso, M. (2015). How Much Domain Data Should Be in Provenance Databases? In: Missier, P., Zhao, J. (Eds.), *7th USENIX Workshop on the Theory and Practice of Provenance, TaPP 2015, Edinburgh, Scotland, UK, July 8–9, 2015*. USENIX Association. https://www.usenix.org/conference/tapp15/workshop-program/presentation/de-oliveira

do Rego Pinto, F.C., Murta, L.G.P. (2023). On the assignment of commits to releases. *Empir. Softw. Eng.*, 28(2), 32. https://doi.org/10.1007/S10664-022-10263-X

Dong, Y., Marwan, S., Catete, V., Price, T., Barnes, T. (2019). Defining Tinkering Behavior in Open-ended Block-based Programming Assignments. In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. SIGCSE '19. Association for Computing Machinery, New York, NY, USA, pp. 1204–1210. 9781450358903. https://doi.org/10.1145/3287324.3287437

Emerson, A., Geden, M., Smith, A., Wiebe, E., Mott, B., Boyer, K.E., Lester, J. (2020). Predictive Student Modelling in Block-Based Programming Environments with Bayesian Hierarchical Models. In: *Association for Computing Machinery*. UMAP '20. Association for Computing Machinery, New York, NY, USA, pp. 62–70. 9781450368612. https://doi.org/10.1145/3340631.3394853

Feng, M., Peck, E., Harrison, L. (2019). Patterns and Pace: Quantifying Diverse Exploration Behavior with Visualizations on the Web. *IEEE Transactions on Visualization and Computer Graphics*, 25(1), 501–511. https://doi.org/10.1109/TVCG.2018.2865117

Freire, J., Koop, D., Santos, E., Silva, C.T. (2008). Provenance for Computational Tasks: A Survey. *Computing in Science & Engineering*, 10(3), 11–21. https://doi.org/10.1109/MCSE.2008.79

Grover, S., Basu, S. (2017). Measuring Student Learning in Introductory Block-Based Programming: Examining Misconceptions of Loops, Variables, and Boolean Logic. In: *Proceedings of the 2017ACM SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '17. Association for Computing Machinery, New York, NY, USA, pp. 267–272. 9781450346986. https://doi.org/10.1145/3017680.3017723

Han, J., Kamber, M., Pei, J. (2011). Data Mining: Concepts and Techniques. (Morgan Kaufmann, 2011). http://hanj.cs.illinois.edu/bk3/

Heath, J., Whyte, R., Sentance, S. (2025). FLARE: A Framework Supporting Code Comprehension and Formative Assessment in Block-Based Programming Education. In: *Proceedings of the 9th Conference on Computing Education Practice*. CEP '25. Association for Computing Machinery, New York, NY, USA, pp. 25–28. 979-8-4007-1172-5. https://doi.org/10.1145/3702212.3702219

Herschel, M., Diestelkämper, R., Ben Lahmar, H. (2017). A survey on provenance: What for? What form? What from? *VLDB J.*, 26(6), 881–906. https://doi.org/10.1007/S00778-017-0486-1

Hey, T., Tansley, S., Tolle, K., Beck, L. (2011). *O Quarto Paradigma: descobertas científicas na era da eScience*. Inventando o Futuro. Oficina de Textos. 9788579750311.

Kazemitabaar, M., Chyhir, V., Weintrop, D., Grossman, T. (2022). CodeStruct: Design and Evaluation of an Intermediary Programming Environment for Novices to Transition from Scratch to Python. In: *Proceedings of the 21st Annual ACM Interaction Design and Children Conference*. IDC '22. Association for Computing Machinery, New York, NY, USA, pp. 261–273. 9781450391979. https://doi.org/10.1145/3501712.3529733

Kesselbacher, M., Bollin, A. (2019). Quantifying Patterns and Programming Strategies in Block-Based Programming Environments. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 254–255. https://doi.org/10.1109/ICSE-Companion.2019.00101

Kong, M., He, O., Louis Mauriello, M., Pollock, L. (2024). "Anything That Can Be Streamlined Would Be Great": Validating Elementary School Teachers' Preferences for a Block-Based Programming Teaching Augmentation System. In: *Proceedings of the 24th Koli Calling International Conference on Computing Education Research*. Koli Calling '24. Association for Computing Machinery, New York, NY, USA. 979-8-40071038-4. https://doi.org/10.1145/3699538.3699539

Krugel, J., Ruf, A. (2020). Learners' perspectives on blockbased programming environments: Code.org vs.

Scratch. *In Workshop in Primary and Secondary Computing Education (WiPSCE'20)*, 12, 28–30. http://doi.acm.org/10.1145/3421590.3421615

Krutz, J., Siy, H., Dorn, B., Morrison, B.B. (2019). Stepwise refinement in block-based programming. *J. Comput. Sci. Coll.*, 35(5), 91–100. https://dl.acm.org/doi/abs/10.5555/3381613.3381623

Marques, F., Lignani, L., Quadros, J., Amorim, M., Viana, W., Ogasawara, E., dos Santos, J. (2024). ProBee: A Provenance-based Design for an Educational Game Analytics Model. *Technology, Knowledge and Learning*. https://doi.org/10.1007/s10758-024-09758-x

Marwan, S., Shabrina, P., Milliken, A., Menezes, I., Catete, V., Price, T.W., Barnes, T. (2021). Promoting Students' Progress-Monitoring Behavior during Block-Based Programming. In: *Proceedings of the 21st Koli Calling International Conference on Computing Education Research*. Koli Calling '21. Association for Computing Machinery, New York, NY, USA. 9781450384889. https://doi.org/10.1145/3488042.3488064

Mladenović, M., Krpan, D., Mladenović, S. (2016). Introducing programming to elementary students novices by using game development in Python and Scratch. In: *EDULEARN16 Proceedings*. 8th International Conference on Education and New Learning Technologies. IATED, pp. 1622–1629. 978-84-608-8860-4. https://doi.org/10.21125/edulearn.2016.1323

Moreau, L., Groth, P. (2013). *Provenance: An Introduction to PROV*. Morgan & Claypool Publishers. https://doi.org/10.2200/S00528ED1V01Y201308WBE007

Morshed, F.F., Tian, X., Emerson, A., B. Wiggins, J., Bounajim, D., Smith, A., Wiebe, E., Mott, B., Boyer, E., K., Lester, J. (2021). Progression Trajectory-Based Student Modeling for Novice Block-Based Programming. In: *Proceedings of the 29th ACM Conference on User Modeling, Adaptation and Personalization*. UMAP '21. Association for Computing Machinery, New York, NY, USA, pp. 189–200. 9781450383660. https://doi.org/10.1145/3450613.3456833

Nakandala, S., Zhang, Y., Kumar, A. (2020). Cerebro: A Data System for Optimized Deep Learning Model Selection. *Proc. VLDB Endow.*, 13(11), 2159–2173. http://www.vldb.org/pvldb/vol13/p2159-nakandala.pdf

Papert, S. (1980). *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, Inc., USA. 0465046274.

Pereira, D., Barbosa, F., Morgado, C. (2024). NextBlocks: An interactive block programming platform. In: *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1*. ITiCSE 2024. Association for Computing Machinery, New York, NY, USA, pp. 590–596. 979-8-4007-0600-4. https://doi.org/10.1145/3649217.3653609

Pina, D.B., Kunstmann, L.N.O., de Oliveira, D., Mattoso, M. (2025a). Breadcrumbs for your Deep Learning Model: Following Provenance Traces with DLProv. *Softw. Impacts*, 23, 100730. https://doi.org/10.1016/J.SIMPA.2024.100730

Pina, D.B., Kunstmann, L.N.O., Chapman, A., de Oliveira, D., Mattoso, M. (2025b). DLProv: a suite of provenance services for deep learning workflow analyses. *PeerJ Comput. Sci.*, 11, 2985. https://doi.org/10.7717/PEERJ-CS.2985

Pozzan, G., Padova, C., Montuori, C., Arfé, B., Vardanega, T. (2024). Experimental Analysis of First-Grade Students' Block-Based Programming Problem Solving Processes. In: *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1*. ITiCSE 2024. Association for Computing Machinery, New York, NY, USA, pp. 143–149. 979-8-4007-0600-4. https://doi.org/10.1145/3649217.3653586

Ribeiro dos Santos, R., Prado Santos, M.T., Ciferri, R.R. (2023). ProvOER model: A provenance model for Open Educational Resources. *Heliyon*, 9(2), 13311. https://doi.org/10.1016/j.heliyon.2023.e13311

Rico, S., Ali, N., Engström, E., Host, M. (2020). Guidelines for conducting interactive rapid reviews in software engineering – from a focus on technology transfer to knowledge exchange. *Zenodo*, 12. https://doi.org/10.5281/zenodo.4327725

Salazar, V., Cavalcante, J., de Oliveira, D., Thompson, F.L., Mattoso, M. (2021). BioProv – A provenance library for bioinformatics workflows. *J. Open Source Softw.*, 6(67), 3622. https://doi.org/10.21105/JOSS.03622

Schulte, C., Sentance, S., Sparmann, S., Altin, R., Friebroon-Yesharim, M., Landman, M., Rücker, M.T., Satavlekar, S., Siegel, A., Tedre, M., Tubino, L., Vartiainen, H., VelÁzquez-Iturbide, J.A., Waite, J., Wu, Z. (2025). What We Talk About When We Talk About K-12 Computing Education. In: *2024 Working Group Reports on Innovation and Technology in Computer Science Education*. ITiCSE 2024. Association for Computing Machinery, New York, NY, USA, pp. 226–257. 9798400712081. https://doi.org/10.1145/3689187.3709612

Sun, L., Guo, Z., Zhou, D. (2022). Developing K-12 students' programming ability: A systematic literature review. *Education And Information Technologies*, 27, 7059–7097.

https://doi.org/10.1007/s10639-022-10891-2
Szabo, C., Sheard, J., Luxton-Reilly, A., Simon, B., Becker, B., Ott, L. (2019). Fifteen Years of Introductory Programming in Schools: A Global Overview of K-12 Initiatives. In: *Proceedings Of The 19th Koli Calling International Conference On Computing Education Research*. https://doi.org/10.1145/3364510.3364513.
Torre, M.V., Oertel, C., Specht, M. (2024). The Sequence Matters in Learning – A Systematic Literature Review. In: *Proceedings of the 14th Learning Analytics and Knowledge Conference*. LAK '24. Association for Computing Machinery, New York, NY, USA, pp. 263–272. 9798400716188. https://doi.org/10.1145/3636555.3636880
Tsung, S., Wei, H., Li, H., Wang, Y., Xia, M., Qu, H. (2022). BlockLens: Visual Analytics of Student Coding Behaviors in Block-Based Programming Environments. In: *Proceedings of the Ninth ACM Conference on Learning @ Scale*. L@S'22. Association for Computing Machinery, New York, NY, USA, pp. 299–303. 9781450391580. https://doi.org/10.1145/3491140.3528298
Wang, J., Crawl, D., Purawat, S., Nguyen, M., Altintas, I. (2015). Big data provenance: Challenges, state of the art and opportunities. In: *2015 IEEE International Conference on Big Data (Big Data)*. IEEE.
Weintrop, D. (2019). Block-based programming in computer science education. *Commun. ACM*, 62(7), 22–25. https://doi.org/10.1145/3341221
Wohlin, C., Mendes, E., Felizardo, K., Kalinowski, M. (2020). Guidelines for the search strategy to update systematic literature reviews in software engineering. *Information and Software Technology*, 127, 106366.
Zhang, L., Nouri, J., Rolandsson, L. (2020). Progression of Computational Thinking Skills in Swedish Compulsory Schools with Block-based Programming. In: *Association for Computing Machinery*. ACE'20. Association for Computing Machinery, New York, NY, USA, pp. 66–75. 9781450376860. https://doi.org/10.1145/3373165.3373173.
Zhao, W.X., *et al.* (2023). A survey of large language models. *arXiv:2303.18223*, 1(2).

N. Arruda earned her Master's degree in Computer Science from the Universidade Federal Fluminense (UFF) in 2025. Her research focuses on block-based programming. She is currently a professor of computer science at the Federal Institute of Santa Catarina – Xanxerê Campus.

M.R. de Lima earned his Bachelor's degree in Information Systems from the Universidade Federal Fluminense (UFF) in 2024. He is currently working as a Salesforce Developer at GLOBAL HITSS.

S. Martins has been a Professor at the Institute of Computing, Universidade Federal Fluminense (UFF), until 2025, when she retired. She received her Doctor of Science degree from the Pontifical University of Rio de Janeiro in 1999. Her research focuses on combinatorial optimization, data mining, and high-performance computing. She is currently a collaborating professor in the postgraduate program in computing at UFF.

D. de Oliveira has been a Professor at the Institute of Computing, Universidade Federal Fluminense (UFF), since 2013. He received his Doctor of Science degree from the Federal University of Rio de Janeiro (UFRJ) in 2012. His research focuses on scientific workflows, provenance, cloud computing, high-performance computing, and distributed and parallel databases. He has led and participated in research projects funded by Brazilian agencies, including CNPq, CAPES, and FAPERJ. Prof. de Oliveira serves on program committees for national and international conferences and workshops and is a member of IEEE, ACM, and the Brazilian Computer Society.