Preprint. Version of record (will be) available at https://doi.org/10.1145/3832783.3834618

ScratchLog+: Live Analytics of Learners' Exercise Progress

Benedikt Fein
University of Passau
Passau, Germany
benedikt.fein@uni-passau.de

Gordon Fraser
University of Passau
Passau, Germany
gordon.fraser@uni-passau.de

Abstract

While the block-based programming language *Scratch* is a popular introductory programming environment, gathering information about student progress is challenging since the environment does not expose any kind of analytics. The *ScratchLog* tool already alleviates this somewhat, but focusses mainly on classroom management and general analytics suited for open-ended tasks. To also provide insights about student progress in goal-oriented exercises, we propose the *ScratchLog*$^+$ extension in this paper. Our tool combines *Whisker* test suites to collect information about the functional correctness of students' programs, and code embedding models to estimate structural and semantic progress towards an example solution. To make these new analytics of *ScratchLog*$^+$ accessible to teachers, they are directly integrated into the *ScratchLog* user interface as interpretable visualisations of the typically abstract code embedding vectors. We provide a screencast demonstrating the tool at https://youtu.be/lQSCDqvbpQY.

CCS Concepts

• **Social and professional topics** → **Software engineering education**; **K-12 education**; • **Software and its engineering** → **Visual languages**.

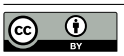
Keywords

Scratch, block-based programming, learning analytics

ACM Reference Format:
Benedikt Fein and Gordon Fraser. 2026. ScratchLog+: Live Analytics of Learners' Exercise Progress. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3832783.3834618

1 Introduction

The block-based programming language *Scratch* [11] is very popular as an introduction to programming due to its low barrier to entry, allowing students to create interactive games within the browser-based user interface. As part of the learning process, beginners frequently introduce bugs in their programs [10]. Since keeping track of all students in classroom setting is challenging, teachers may want tool assistance to identify students in need of support in fixing such bugs. However, the *Scratch* platform does not offer any such insights.



ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834618

The *ScratchLog* [2] tool was developed to collect events on how students interact with the *Scratch* interface. It allows for example for live analytics about the number of bugs in a program as found by the *LitterBox* [9] static code analysis tool. By tracking how often students add or remove blocks and how frequently they restart their programs *ScratchLog* offers insights into whether they are trial-and-error debugging or idle. This information can guide teachers when supervising students in a classroom, but mainly targets open-ended programming tasks. When students are given concrete tasks and goals, however, *ScratchLog* until now provided no insights into whether a student's program is moving towards the intended task solution, whether the student is stuck, or whether the student is active but moving astray from the given task.

To fill this gap, this paper proposes our *ScratchLog*$^+$ extension to *ScratchLog* to enable live analytics of the students' progress also in task-oriented exercises. Our approach integrates two new sources of information into *ScratchLog*: The *Scratch* testing framework *Whisker* [5] and code embedding models. By implementing a *Whisker* test suite, a teacher can verify that all intended functionality works as expected in their example solution. The same test suite can now also be executed against intermediate student programs to check whether the functionalities of subtasks are already working and which features are still missing. As *Whisker* tests are effectively system tests they provide coarse feedback, i.e., they do not indicate whether the tested functionality is fully missing or not working due to a small logic bug in the program. Furthermore, developing a comprehensive test suite may be challenging for teachers. We therefore additionally integrate code embedding models as more granular progress estimators. These machine learning models are trained to map the source code of the program into dense vector space such that structurally and semantically similar programs have similar vector representations.

Since code embedding vectors are difficult to interpret due to their high dimensionality (typically ≥ 128), *ScratchLog*$^+$ processes them further to integrate them into a new exercise progress dashboard made available in the *ScratchLog* user interface. This dashboard contains several interpretable and interactive visualisations representing the students' progress.

All components of *ScratchLog*$^+$ are available under an open-source licence (GPL-3.0) and are available at https://github.com/se2p/ScratchLog/ and https://github.com/se2p/whisker/.

2 Background

To enable live analytics of student programming behaviour in the *Scratch* environment, *ScratchLog* [2] instruments the *Scratch* user interface and virtual machine to collect various events for user interactions (e.g., starting the program execution). In case the event was caused by a code-changing interaction (e.g., adding or deleting a block), a snapshot of the resulting code is sent to *ScratchLog* as

part of the event. Using the *LitterBox* [9] static code analysis tool, *ScratchLog* can therefore not only track user interactions but also track how program quality evolves during a classroom session.

Whisker [5] is the state-of-the-art testing framework for *Scratch* programs. Test suites written in JavaScript control the program execution and allow simulating user inputs to the often interactive programs. Since these programs are often small games for which the repeated execution for each test case would take a considerable amount of time, the *Whisker* test harness can speed up the *Scratch* virtual machine while keeping the execution fully deterministic [5]. This makes it possible to continuously execute the tests for all student programs while they are being collected by *ScratchLog*$^+$.

The test results of student programs could be used to map the students' exercise progress into trajectories and thus highlight challenging parts of the exercise [12]. However, the creation of a comprehensive *Whisker* test suite verifying the functional correctness of a program can be challenging and time-intensive. As a heuristic to estimate the semantic features of a program, code embedding models can be used instead. These machine learning models map the code into a dense vector space representation that captures both semantic and structural aspects. Since most general-purpose LLMs are trained on large datasets including natural language and open-source code, they can be used to embed code. Prior work found that using the *scratchblocks* notation[1] as used on the *Scratch* community forums rather than the *Scratch*-internal JSON program representation yields better results when prompting LLMs [6]. In *ScratchLog*$^+$, we adopt this input format when prompting embedding-LLMs to compute embeddings of *Scratch* programs.

Since the code embedding vectors are highly dimensional, they need to be projected to two-dimensional space to obtain a representation that is easier to visualise and interpret. While a generic dimensionality reduction algorithm (e.g., *PCA* [15] or *t-SNE* [17]) could be used to show the students' solution attempt trajectories [16], the progress-variance-projection [14] is specifically designed to visualise student progress. It is constructed to place the starting program given to all students at the beginning of the exercise at the origin $(0, 0)$ and an example solution at point $(1, 0)$ so that the x-axis represents the students' progress. The y-axis then aims to maximise the variance in programs to capture the differences between students' solution approaches. The projection was originally used for Python exercise tasks, but its computation is independent of the specific exercise or programming language as long as the embedding vectors are computed with a suitable model.

Scratch uses programming paradigms different from most textual programming languages. For example, users can place blocks freely on the two-dimensional workspace, the execution flow is event-driven and concurrent, and the programs typically implement small games or animations rather than solving algorithmic problems. Nevertheless, common code embeddings can be adapted for this new environment [8]. By pre-training the code embedding models to name the animated figures (sprites) based on the contained within, the models can learn relevant semantic information about the code. Likely due to the different structure of *Scratch* programs, models including control- and data-flow information, e.g., *GGNN* [1], outperform models operating on only the syntactic code context [8].

[1] https://en.scratch-wiki.info/wiki/Block_Plugin/Syntax, accessed 2026-05-08

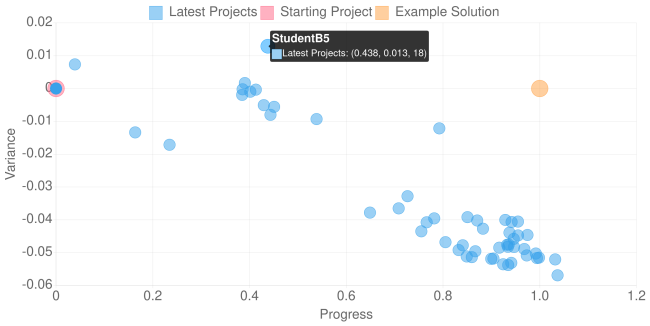


Figure 1: Progress-Variance-Projection of the current project of all students in the class.

Experiments also demonstrated that pre-trained embedding-LLMs, e.g., *Qwen 3 Embedding* [18], exhibit a similar performance to the dedicated *GGNN* model both when estimating students' programs functional correctness and when using the embeddings to construct the progress-variance-projection [8].

3 Features

To make the embeddings, their projections, and *Whisker* test results available to teachers, *ScratchLog*$^+$ extends the *ScratchLog* user interface with a new 'Exercise Progress Dashboard'. This new page contains four visualisations highlighting different aspects that may be relevant for a teacher in tracking the students' progress.

All new charts are interactive in the sense that the points provide more information on hover (e.g., student name and exact coordinates of the point, cf. Fig. 1). The plots can also be zoomed in to inspect specific areas of the chart in more detail. The charts are automatically updating to include the latest data collected by *ScratchLog*.

The data shown in the screenshots was obtained during a course implementing various non-trivial extensions to the 'Boat Race' project.[2] The ninth-grade students were given the base project as a starting point and were then tasked with adding a score system introducing variables, showing different texts on the game end screen depending on the score requiring conditional logic, and hiding no longer required figures on the game end screen which required keeping track of the overall game state [13]. The embeddings shown in the screenshot were computed with the *Qwen 3 Embedding* model [18], but we also implemented an integration with a *GGNN* code embedding model [1] adapted for *Scratch* by pre-training it to suggest the sprite names based on the code therein [8].

3.1 Class Progress Overview

As first visualisation (Fig. 1), a scatter plot is showing the progress-variance-projection of the current student projects. Its main goal is to give a quick overview over the overall progress of the class. By focussing on outliers like the highlighted 'StudentB5', a teacher can find students whose progress is behind their peers or who are moving in a different direction. In both cases an intervention could be required to either offer additional guidance or to investigate whether the student is still working on the intended task.

[2] https://projects.raspberrypi.org/en/projects/boat-race, accessed 2026-05-09

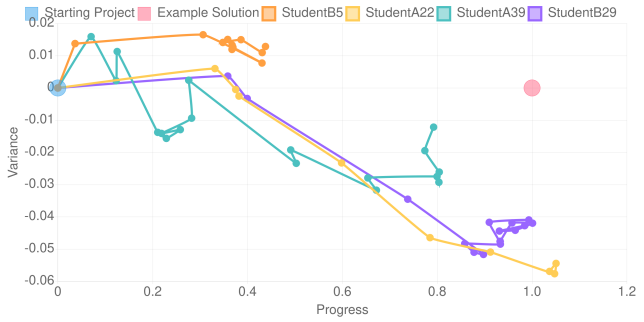


Figure 2: Progress over time during the class session for four specific students. The projects were sampled in 2-minute intervals for each student.

Figure 3: Functional correctness of student programs measured by functional tests (x-axis) compared to the embedding-estimated correctness (y-axis, 1=closest to the solution).

In case of Fig. 1, the shown student projects are the final ones created as part of the course session. While the progress of many student projects is close to the example solution (i.e., progress near 1.0), the variance diverges. For the project in this course there is not one unique solution, but multiple valid solutions exist. The teachers might have presented a slightly different variant to students in the course rather than exactly following the prepared example solution. At around progress 0.4, the plot also shows a cluster of students who only finished the first steps of the overall task. As the following gap to progress $\approxeq 0.6$ and cluster of points with progression ≥ 0.8 shows, most students that were able to implement the seemingly challenging intermediate subtask were subsequently able to implement most of the remaining features. Progress values slightly larger than 1.0 are possible due to the heuristic nature of the code embedding models.

3.2 Historic Task Progress

While the scatter plot gives an overview over the classes' current progress, it does not explain how individual students arrived at their current state. To provide the missing context, we again use the progress-variance-projection to visualise the historic states of students' code (cf. Fig. 2). A teacher can select one or more students for whom they want to inspect the history. They can also change the granularity by changing the interval with which the program states are sampled from the whole set of events collected by *ScratchLog*. For the visualisation shown in Fig. 2 we sampled the last program state per 2-minute interval for each student.

As Fig. 2 shows, 'StudentA22' followed a straightforward path towards a solution, while 'StudentB29' followed a similar path for the majority of the task. However, towards the end the cluster of program states indicates some trial-and-error to implement the remaining functionality. 'StudentA39' attempted a diverging approach at the beginning (up to progress ≈ 0.25) before moving back onto a path more similar to the other students. Towards the end (progress 0.8) this student continues to change their program without making further progress. Since the variance moves towards the example solution, they might have found a path towards a solution that more closely matches the example solution rather than the alternative approach followed by most students. The outlier detected earlier in the class overview ('StudentB5', cf. Fig. 1) started with a similar diverging approach as A39, but did not manage to move back towards the usual solution path of the other students and instead got stuck early near progress 0.4.

3.3 Embedding- and Test-Fitness

The two visualisations explained above use only the code embedding machine learning models combined with the progress-variance-projection to estimate the students' progress. Its computation only requires the teacher to provide a starter project and an example solution. By additionally providing a *Whisker* test suite, its test cases can provide a definitive measurement of a program's functional correctness. We define a program's test fitness as the proportion of passed test cases of the test suite (i.e., a program passing all tests has fitness 1.0). Similarly, we define an embedding fitness as 1 minus the normalised embedding-space distance between program and example solution vectors (i.e., again a fitness near 1.0 indicates the program is close to the solution).

Figure 3 compares these two fitnesses to find discrepancies. Programs with a low test fitness but high embedding fitness (e.g., the point near $(0.2, 0.6)$) are likely structurally similar to the example solution but still contain various logical bugs that result in test case failures. Conversely, a high test fitness with low embedding fitness indicates valid alternative solutions (e.g., the point near $(1.0, 0.6)$). As expected from the low progress observed in the previous visualisations, the program of 'StudentB5' indeed only passes one of the eleven test cases of the *Whisker* test suite.

This comparison confirms the earlier hypothesis that many students found an alternative to the example solution, since the embedding fitness generally remains below 0.8 while there exist programs with high test fitness and two students' programs even passing all test cases. It also demonstrates that the embedding distance strongly correlates with the true functional correctness of the programs (Pearson $r = 0.81$, $p = 8 \cdot 10^{-15}$), which shows that the embedding model can capture relevant structural and semantic information about the students' code. The *GGNN* embeddings exhibit a weaker correlation ($r = 0.34$, $p = 0.009$). *Qwen* likely outperforms *GGNN* in this scenario, since embedding-LLMs are designed to map semantically similar inputs into close embeddings and functionally similar programs are likely syntactically similar. The model also

User	boat crash crab invisible	boat crash gate invisible	boat touching crab increments points	boat touching crab move	boat winning crab invisible	boat winning gate invisible	game over - crab invisible	game over - gate invisible	gate spinning	points initialization
A22	✅	✅	❌	❌	✅	✅	✅	✅	✅	✅
A39	❌	❌	❌	❌	❌	❌	❌	❌	❌	❌
B5	❌	❌	❌	❌	❌	✅	❌	❌	❌	❌
B29	✅	✅	✅	✅	✅	✅	✅	✅	✅	✅

Figure 4: Detailed *Whisker* results for the latest projects of the students shown in Fig. 2.

receives the whole program as a single input whereas *GGNN* constructs per-sprite embeddings which are then mean-pooled into the program embedding [8]. Since for the BoatRace project the majority of the code resides in the Boat sprite, more nuanced sprite embedding information might be lost in the pooling step due to information-sparse embeddings of the nearly empty sprites.

3.4 Detailed Test Results

In addition to the graphical representations of student progress, the new exercise progress dashboard also includes a table of specific *Whisker* test case results for the newest student programs. As shown in Fig. 4, the table is automatically filtered to show the same students as selected for the timeline visualisation (Fig. 2). The table aims to help teachers use the failing test cases as indicators for possibly relevant code locations which should be inspected first when providing individual support for students so that the issues can be resolved more quickly.

The example of Student A39 highlights a discrepancy between the embedding-based progress estimation and the functional tests. As Fig. 4 shows, all functional *Whisker* tests fail for this student's program even though the progress-variance-projection estimates the student to be close to the solution (cf. Fig. 2). Manually inspecting the program reveals that it indeed does not implement hiding the 'Boat' and 'Crab' sprites on the game ending screen. However, while the newly added obstacle placed into the boat's path (called the 'gate' by the test cases) can spin as intended by repeatedly calling the 'rotate' block in a loop, it does so only after a specific key is pressed by the user, which is not anticipated by the tests. Similarly, the test cases related to the score system fail since the student instead implemented the crab to say something rather than awarding points when touched by the boat. In this case the student still successfully used a repeated check for the touching state with subsequent conditional behaviour. While the *Whisker* tests related to these partially implemented features fail since they did not expect this deviating program behaviour, the mapping to embedding space abstracts from the concrete behavioural aspects to instead take programming constructs which are correctly used in relevant areas of the program into account.

4 System Design

4.1 Components

As summarised in Fig. 5, *ScratchLog*⁺ consists of multiple components that interact via HTTP APIs to ultimately provide the progress visualisations to the user. The interaction between the *Scratch* UI and *ScratchLog* remains unchanged, since the collected events already contain the program code which is later required for test execution and embedding computation. While the new progress dashboard is integrated into the *ScratchLog* UI itself, the majority of the computations are delegated to the separate components.

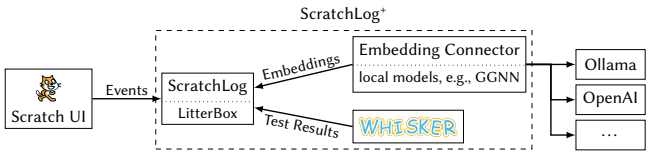


Figure 5: *ScratchLog*⁺ system component overview. Communication between components via HTTP APIs.

Whenever *ScratchLog* receives a new event about a changed program, it delegates the test execution to the separate *Whisker* service. *Whisker* has to remain a separate component that cannot be integrated into the Java-based *ScratchLog* server since it is implemented in TypeScript to be able to integrate tightly with the JavaScript-based *Scratch* virtual machine. We extend the *Whisker* application with a new HTTP API that receives a program and a test suite, runs the tests, and returns the results to *ScratchLog*. The test results are stored in the *ScratchLog* database so that they do not need to be re-executed unless the test suite of the exercise changes.

The code embeddings are computed on-demand when a user opens the new progress dashboard. This avoids unnecessary computations and thus potential costs for embeddings that are never actually accessed by the user. Since the *Scratch*-internal JSON representation of the programs is not suitable as embedding model input [6], we use *LitterBox* [9] to transform it into the model-specific format. *LitterBox* supports the *scratchblocks* representation as used for LLM inputs as well as various other formats suitable for various dedicated code embedding models. *LitterBox* is used as a direct Java dependency of *ScratchLog* and thus does not need a separate service. Following the preprocessing, the programs are sent to the embedding connector. This service is a Python application that either delegates the actual embedding computation to an OpenAI-compatible LLM API like for example a locally hostable *Ollama* instance, or can integrate other local models (e.g., *GGNN* [1]) via a direct Python dependency. It also processes the embeddings further to compute the progress-variance-projections or distances in embedding space.

4.2 Deployment

To simplify the deployment of the various subsystems, we containerised all components except for the external LLM endpoints and provide a *Docker Compose* definition file. This allows them to be started locally with a single `docker compose up -d` command. The components interact via the internal container network and both the *Scratch* and *ScratchLog* UIs become accessible to the user via their respective web-browser-based interfaces. To encourage testing of *ScratchLog*⁺, we also provide an anonymised database snapshot containing the course data used for the screenshots shown in this paper and a pre-trained checkpoint of the *GGNN* model [7]. Access to an OpenAI-compatible embedding endpoint can be configured when starting the applications.

The *Qwen 3 Embedding* (8b parameters) model [18] used to compute the embeddings for the demonstrations shown in this paper requires roughly 300 ms per program on a local server with a dedicated GPU. Therefore, the overall initial generation of the embeddings including all data pre- and post-processing for a classroom of 30 students typically takes less than 20 s. Computing the *GGNN* embeddings on a regular laptop PC without dedicated GPU requires roughly 25 ms per embedding and 4 s until all visualisations are loaded. The computed embedding vectors are cached to ensure even shorter waiting times on subsequent requests and to reduce costs when using external LLM providers. The following projection computations are not cached, due to them being computationally cheap compared to the embedding vector computation. Since the projection's construction also aims to maximise the variance component (cf. Section 2) between the given set of programs, a cache would only be useful given the exact same set, which is unlikely in a live exercise with constantly changing student programs.

5 Future Work

The expressiveness of the visualisations relies on the ability of the embedding models' ability to capture relevant structural and semantic information about the programs. As outlined in Section 3.3, the *GGNN* embedding pre-trained on sprites might not be optimal for the BoatRace-based exercise used for this demonstration. This hints at future opportunities to create more expressive whole-program embeddings to match the performance of pre-trained LLMs at a lower computational inference cost. Since prior evaluation comparing the model performance on the test fitness estimation task on two additional exercises found the performance to be somewhat exercise dependent [8], improved embedding models should be evaluated across additional *Scratch* programming exercises. Finally, while this paper provides an initial demonstration of *ScratchLog*⁺, the tool should be evaluated in a live classroom setting to collect feedback from teachers and evaluate the practical impact.

6 Conclusions

In this paper we introduced our *ScratchLog*⁺ tool aiming to ease the collection of learning analytics about students' progress in *Scratch* programming exercises by combining multiple progress indicators and combining them into interpretable visualisations. By making our tool available as open-source, we encourage its active use within the *Scratch* programming education community. Since the general approach of *ScratchLog*⁺ of using tests and code embedding models as progress indicators is not *Scratch*-specific, we hope to inspire the development of similar tools also for other programming environments.

Data Availability Statement

Our extensions to *ScratchLog* including the embedding connector are available at https://github.com/se2p/ScratchLog/. Our integration with *Whisker* can be found at https://github.com/se2p/whisker/. To ensure long-term availability, snapshots of both repositories are saved in the Software Heritage Archive [3, 4]. All components of *ScratchLog*⁺ are available under an open-source licence (GPL-3.0).

Acknowledgments

This work is supported by Deutsche Forschungsgemeinschaft (DFG) project FR 2955/3-3 'PetBlock: Didaktik und Technik für Feedback in Block-Basierter Programmierung'.

References

[1] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *International Conference on Learning Representations (ICLR)*. arXiv. arXiv:1711.00740 [cs]

[2] Laura Caspari, Luisa Greifenstein, Ute Heuer, and Gordon Fraser. 2023. ScratchLog: Live Learning Analytics for Scratch. In *Innovation and Technology in Computer Science Education (ITiCSE)*. ACM. doi:10.1145/3587102.3588836

[3] Chair of Software Engineering II, Passau. 2026. *ScratchLog*. https://archive.softwareheritage.org/swh:1:rev:fc6930c960bca5ade7c5ffc0bb0c5bbf4977d1f3;origin=https://github.com/se2p/ScratchLog;visit=swh:1:snp:16608928dc07c3bce3297f48be1332439688c781

[4] Chair of Software Engineering II, Passau. 2026. *Whisker*. https://archive.softwareheritage.org/swh:1:rev:ec4002245229701faba77e4c6ed55d73a812a17d;origin=https://github.com/se2p/whisker;visit=swh:1:snp:f69a50caf6466479433294d992e14790bc783e15

[5] Adina Deiner, Patric Feldmeier, Gordon Fraser, Sebastian Schweikl, and Wengran Wang. 2023. Automated test generation for Scratch programs. *Empirical Software Engineering* 28, 3 (May 2023). doi:10.1007/s10664-022-10255-x

[6] Benedikt Fein, Patric Feldmeier, Gordon Fraser, and Florian Obermüller. 2026. Reasoning About Bugs in Learners' Scratch Programs Using Large Language Models. In *International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. ACM. doi:10.1145/3786580.3786949

[7] Benedikt Fein and Gordon Fraser. 2026. Companion Data for ScratchLog+: Live Analytics of Learners' Exercise Progress. Zenodo. doi:10.5281/zenodo.21495924

[8] Benedikt Fein and Gordon Fraser. 2026. EmbeddedKittens: An Evaluation of Code Embeddings for Scratch. (July 2026). doi:10.48550/arXiv.2607.19291

[9] Gordon Fraser, Ute Heuer, Nina Körber, Florian Obermüller, and Ewald Wasmeier. 2021. LitterBox: A linter for Scratch programs. In *International Conference on Software Engineering: Joint Track on Software Engineering Education and Training (ICSE-JSEET)*. IEEE. doi:10.1109/icse-seet52601.2021.00028

[10] Christoph Frädrich, Florian Obermüller, Nina Körber, Ute Heuer, and Gordon Fraser. 2020. Common Bugs in Scratch Programs. In *Innovation and Technology in Computer Science Education (ITiCSE)*. ACM. doi:10.1145/3341525.3387389

[11] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. *ACM Trans. Comput. Educ.* 10, 4 (Nov. 2010). doi:10.1145/1868358.1868363

[12] Jessica McBroom, Benjamin Paassen, Bryn Jeffries, Irena Koprinska, and Kalina Yacef. 2021. Progress Networks as a Tool for Analysing Student Programming Difficulties. In *Australasian Computing Education Conference (ACE)* (2021). ACM. doi:10.1145/3441636.3442366

[13] Florian Obermüller and Gordon Fraser. 2025. Automatically Generating Questions About Scratch Programs. In *Global Computing Education Conference (CompEd)*. ACM. doi:10.1145/3736181.3747131

[14] Benjamin Paassen, Jessica McBroom, Bryn Jeffries, Irena Koprinska, and Kalina Yacef. 2021. Mapping Python Programs to Vectors Using Recursive Neural Encodings. *Journal of Educational Data Mining (JEDM)* 13, 3 (Oct. 2021). doi:10.5281/zenodo.5634224

[15] Karl Pearson. 1901. On lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 2, 11 (Nov. 1901). doi:10.1080/14786440109462720

[16] Idir Saïdi, Nicolas Durand, and Frédéric Flouvat. 2025. Analysis of Students' Attempts Trajectories in Learning Programming. In *International Conference on Educational Data Mining (EDM)*. International Educational Data Mining Society. doi:10.5281/zenodo.15870155

[17] Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing Data using t-SNE. *Journal of Machine Learning Research* 9, 86 (2008). http://jmlr.org/papers/v9/vandermaaten08a.html

[18] Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, Fei Huang, and Jingren Zhou. 2025. Qwen3 Embedding: Advancing Text Embedding and Reranking Through Foundation Models. (June 2025). arXiv:2506.05176 [cs.CL]

Received 2026-05-12; accepted 2026-06-19