EPFL

CHILI
computer-human interaction
in learning and instruction

LEARN
Center for Learning Sciences

A Web Platform for Orchestration Graph-Based Lesson Planning

Author:
Jules Delforge

Professor and Supervisor:
Pierre Dillenbourg

January 8, 2026

Contents

1 Acknowledgments **1**

2 Introduction **1**

3 Literature Review **1**
3.1 Orchestration in Educational Technology 1
3.2 Activity-Based Learning Models 1
3.3 Computational Approaches to Lesson Planning 2

4 Material and Methods **2**
4.1 System Architecture 2
4.1.1 Backend Architecture 2
4.1.2 Frontend Architecture 2
4.1.3 Development Process and Architecture Decisions 3
4.2 Our Model of a Lesson 3
4.3 Core Algorithms 4
4.4 User Interface Design 5

5 Results **6**
5.1 User Interface Implementation 6
5.2 System Capabilities and Design Trade-offs 7
5.3 Performance Analysis 7
5.4 Architecture Benefits 7

6 Conclusion **7**
6.1 Key Contributions 8
6.2 Limitations and Reflections 8
6.3 Real-World Testing and Impact 8

1 Acknowledgments

I am grateful to Pierre Dillenbourg who presented the concept of Orchestration Graphs and the first ideas about how to build off of Samuel's project. Then, he provided supervision and insightful ideas throughout the semester

I acknowledge the use of CLAUDE (Anthropic) for planning and overview when I lacked the ability to see where the issue was coming from or when I wasn't sure how to proceed.

2 Introduction

Educational lesson planning requires balancing multiple pedagogical constraints: learning objectives, time budgets, prerequisite knowledge, and organizational structures. This project builds a web-based application for designing lesson plans leveraging *orchestration graphs*, a computational framework that models student understanding as a progression through a multi-dimensional state space [3]. The work modernizes Samuel Bélisle's Qt/QML desktop application by migrating to a full-stack web platform while preserving the core pedagogical algorithms.

This work builds directly upon the orchestration graph engine developed by Samuel Bélisle [3], available at `https://github.com/Katokoda/OG_QML`. While the core pedagogical algorithms are mostly preserved, the contribution of this project lies in migrating the system to a web-based architecture and making targeted corrections to the recommendation logic. These changes aim to improve accessibility and algorithmic behavior without altering the conceptual foundations of orchestration graphs.

One may find the repo for this project here `https://github.com/Julkzein/pdb`

3 Literature Review

3.1 Orchestration in Educational Technology

Educational orchestration refers to coordinating learning activities, resources, and social configurations to achieve pedagogical goals [1, 2]. Orchestration graphs represent lesson plans as state-space transitions, where activities move students from one knowledge state to another.

3.2 Activity-Based Learning Models

Activities are organized across three social planes: **Individual** (personalized learning), **Team** (collaborative peer learning), and **Class** (whole-class instruction). Each activity has prerequisite conditions, learning effects, and temporal constraints, enabling activity sequencing and optimization.

3.3 Computational Approaches to Lesson Planning

The orchestration graph approach uses a greedy gap-filling algorithm that prioritizes efficiency (learning progress per unit time). Unlike constraint satisfaction or genetic algorithms that search for global optima, the greedy method provides locally optimal choices, trading solution quality for computational efficiency and interactive responsiveness. This design supports real-time teacher interaction but does not address multi-objective optimization or student-specific adaptation.

4 Material and Methods

4.1 System Architecture

The application uses a client-server architecture where the React frontend communicates with a Flask backend through a REST API. This separation emerged as necessary after an early attempt to integrate Samuel's Qt code directly.

4.1.1 Backend Architecture

The backend is built in pure Python 3.8+ using Flask, deliberately avoiding any Qt dependencies from Samuel's original implementation. The main challenge was extracting the orchestration algorithms from their Qt signal/slot structure while preserving exact behavior. From there were created the `*_pure.py` versions of each module that replaced Qt signals with simple Python return values. This took longer than anticipated because Samuel's code relied heavily on Qt's property binding system for state updates. I had to manually trace each signal connection to understand the update flow.

Initially, all endpoints of the Flask API were synchronous, which caused the frontend to freeze during expensive operations like auto-complete. Recommendation endpoints were refactored to use longer timeouts while keeping simple operations fast. This wasn't an optimal solution as a proper solution would use WebSockets for long operations—but it was pragmatic given time constraints.

4.1.2 Frontend Architecture

The frontend uses React 18 with TypeScript, primarily because TypeScript's type checking caught numerous bugs during development when translating between Samuel's Python data structures and JavaScript objects. Zustand was chosen for state management over Redux because the orchestration state is fundamentally a single object tree that maps well to Zustand's simple store model. React DnD handles drag-and-drop, though I initially tried react-beautiful-dnd before discovering that it couldn't handle the absolute positioning requirements of the timeline (activities must be positioned by time, not insertion order).

The most challenging architectural decision was handling state synchronization between frontend and backend. Every activity insertion triggers a backend recalculation of

the entire orchestration graph (gap evaluation, state progression, constraint checking). I initially tried optimistic updates where the UI would update immediately and roll back on validation failure, but this caused flicker when activities were rejected. The current implementation blocks on backend responses, which feels slower but prevents confusing state inconsistencies. A better solution would compute preview states client-side before committing to the backend, but this would require duplicating Samuel's algorithms in TypeScript.

4.1.3 Development Process and Architecture Decisions

The initial development approach attempted to build a baseline web application with a simplified mock orchestration engine, planning to integrate Samuel's engine later. This seemed reasonable for rapid prototyping, but I quickly realized it wouldn't work: the mock engine lacked the nuanced state progression logic, gap evaluation, and efficiency calculations that make the system functional. More importantly, building the UI around a simplified model would require significant refactoring when integrating the real engine.

Pivoting to building directly around Samuel's engine required creating the clean separation between orchestration logic and web framework described above. The Flask API layer wraps these pure Python modules, translating between HTTP requests and method calls. This separation proved essential for testing and maintainability, but created tight coupling to Samuel's data structures and method signatures, limiting flexibility for future modifications.

4.2 Our Model of a Lesson

A lesson in the orchestration graph system is represented as a sequence of activities that take the class from an initial state to a target goal state. The model reduces the state of each student's understanding into two values: p_{depth} (depth of understanding) and p_{fluency} (fluency in achieving the task), both ranging from zero to one. Each activity has prerequisite conditions $(c_{\text{depth}}, c_{\text{fluency}})$ that must be satisfied and produces effects $(e_{\text{depth}}, e_{\text{fluency}})$ that modify the student state. Activities are positioned sequentially in time and assigned to one of three organizational planes: Individual, Team, or Class. The system enforces temporal constraints through a time budget, representing lesson time. See Samuel's report [3] for more details and explanation.

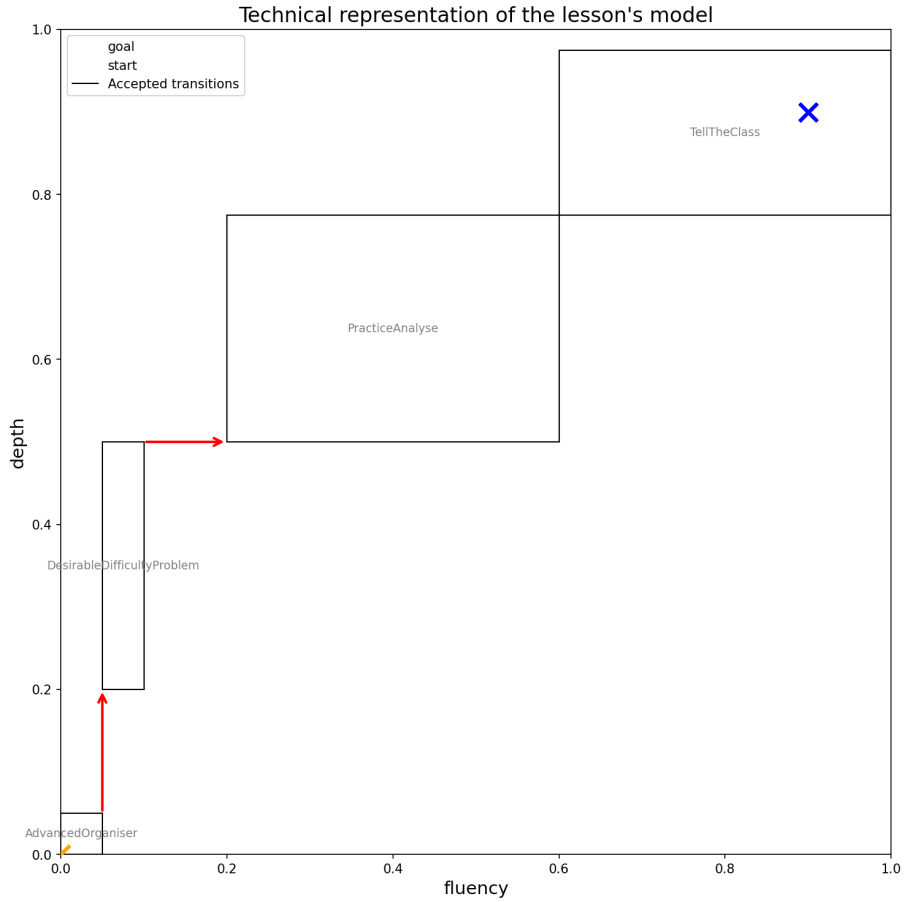

Figure 1: State-space visualization showing learning trajectory through fluency-depth space. Orange X: initial state, blue X: goal state.

4.3 Core Algorithms

The core orchestration algorithms are preserved from the original Qt/QML implementation [3]. The system evaluates learning gaps where the current state is insufficient to reach prerequisites or goals, and employs a greedy recommendation algorithm that selects activities maximizing learning progress per unit time. Some modifications were done on this algorithm.

The primary modification was fixing the distance metric for gap evaluation. The original Euclidean distance $d(s_1, s_2) = \sqrt{\sum_i (s_{1,i} - s_{2,i})^2}$ scores activities poorly when current state exceeds their prerequisites, even though exceeding prerequisites is pedagogically acceptable. Above all, it stops some valuable activities to be considered. The fix uses forward-only distance $d_{\text{forward}}(s_1, s_2) = \sqrt{\sum_{i:s_{1,i}<s_{2,i}} (s_{1,i} - s_{2,i})^2}$, which only considers dimensions requiring progress.

This bug particularly affected scenarios with unbalanced skill profiles. Consider the lesson shown in Figure 2, where after two fluency focused activities, students have high fluency (0.7) but a mid depth (0.22). Here, the engine would wrongly not propose activities

needing low fluency as the student would be "overqualified" even if this was the optimal activity in terms of building depth. However, one may argue that if an "overqualified" student is exposed to an easier exercice, they might be less inclined to follow it which could justify the original distant metric even though it is less optimal.

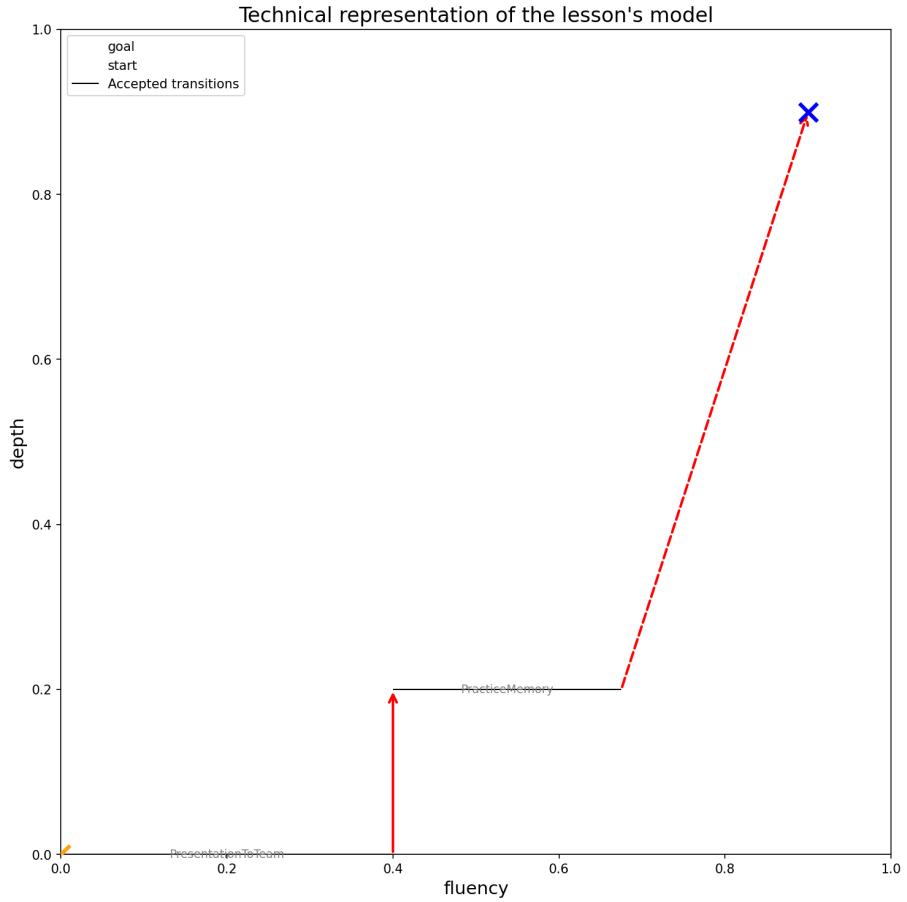

Figure 2: Example of unbalanced skill progression requiring depth-focused activities.

I also corrected a constraint validation bug in `okeyToTake()` that only checked the `noProgress` flag, allowing exhausted or over-budget activities.

4.4 User Interface Design

The interface consists of a timeline with three lanes (Individual, Team, Class), an activity library panel with draggable cards, and a management toolbar. Activities are positioned by `startsAfter` time with drag-and-drop placement, hover information, gap indicators, and real-time validation. Matplotlib generates 2D state-space visualizations as static PNGs.

The application integrates DeepSeek AI to generate age-appropriate teaching resources including explanations, prompts, materials, and assessments. The implementation uses

a two-tier prompt structure for enlarged context: a global system prompt defining the educational consultant role and output format, plus activity-specific prompts from the CSV library explaining the idea precisely by taking information from the activity object. DeepSeek was selected for cost-effectiveness. Only non-sensitive data is transmitted (activity descriptions, age group, subject) making the choice of LLM not so crucial.

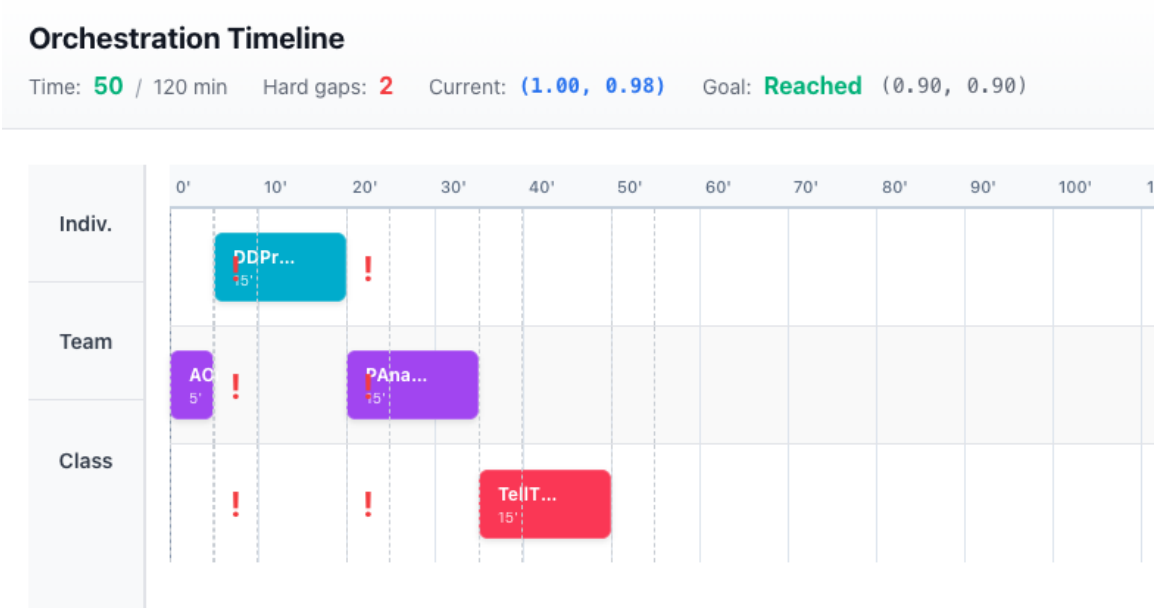

Figure 3: Orchestration timeline with activities across three organizational planes, showing real-time gap feedback and temporal positioning.

5 Results

5.1 User Interface Implementation

The web application offers automated lesson generation or manual orchestration control. The timeline (Figure 3) arranges activities in draggable activity cards across three organizational planes (Individual, Team, Class) with left-to-right time progression, displaying real-time gap metrics and goal achievement status. State-space visualization (Figure 1) plots learning trajectories in 2D fluency-depth space, with activity effects as rectangles and arrows showing state transitions between starting point (orange X) and goal (blue X).

LLM integration generates age-appropriate teaching materials including pedagogical suggestions, activity-specific examples, categorized resources (videos, worksheets, simulations), and teaching tips with concrete examples, URLs, and time management strategies.

5.2 System Capabilities and Design Trade-offs

The system supports flexible-duration activities with effects scaling through linear interpolation. The recommendation engine computes forward-only distance to prerequisites and goals, selecting activities maximizing $\eta = \Delta d/t$ (gap closure per minute). Auto-completion took 2-8 iterations for typical 60-minute lessons. The CSV-based activity library allows Excel editing but lacks validation, concurrent access, and versioning—acceptable for prototyping but requiring database migration for production.

5.3 Performance Analysis

Performance profiling on M2 Macbook Air (Python 3.11):

Operation	**10 activities**	**50 activities**	**Complexity**
Gap evaluation	0.8ms	3.2ms	$O(n)$
Recommendation calc	12ms	58ms	$O(n \cdot m)$
Graph visualization	95ms	210ms	$O(n)$
Frontend re-render	18ms	22ms	$O(1)$

Table 1: Backend performance metrics showing orchestration operations remain under 100ms for realistic library sizes.

Critical bottleneck: LLM API calls (15-45s, dominated by network). Backend orchestration operations remain <100ms for realistic library sizes ($n < 50$), making them imperceptible to users. The greedy algorithm's $O(n \cdot m)$ complexity means 100 activities would require ∼115ms per recommendation which is largely acceptable for interactive use.

5.4 Architecture Benefits

The web architecture enables browser-based access without installation and provides REST API extensibility for future integrations. However, deployment is more complex than the original single-executable Qt application, requiring separate backend and frontend hosting.

6 Conclusion

This project demonstrates that orchestration graph algorithms can be extracted from desktop frameworks and delivered through web browsers. The technical migration succeeded: the system functions equivalently to Samuel's Qt application, with the same recommendation logic and gap evaluation algorithms. The web platform requires no installation and runs on any device with a browser, which theoretically improves accessibility. However, I lack further tests with teachers to evaluate the limits of this application.

6.1 Key Contributions

This project makes three concrete contributions beyond Samuel's original work:

Web Platform Migration: I successfully decoupled the orchestration algorithms from Qt dependencies, creating pure Python modules that can be reused in other projects. This required reverse-engineering Samuel's signal/slot architecture to understand state update flows, which was more difficult than anticipated because Qt's property binding system has implicit dependencies that are not visible in the code. The resulting Flask API provides 15 endpoints covering all orchestration operations, enabling future integration with learning management systems or mobile applications.

Distance Metric Correction: The forward-only distance function fixes a subtle but impactful bug in the original recommendation algorithm. This represents an edge case than can however block an otherwise optimal activity. While this is a small algorithmic change, it's interesting to see how the engine evaluates learning gaps when students have unbalanced skill profiles.

LLM-Enhanced Teacher Materials: The DeepSeek integration generates contextualized teaching resources automatically. However, I must acknowledge this feature's limitations as the prompts for the resource generations where not further studied or engineered. This remains a proof-of-concept rather than a production-ready feature.

6.2 Limitations and Reflections

The greedy recommendation algorithm can make locally optimal choices that prevent reaching the goal. Auto-complete occasionally fails even when valid solutions exist, Samuel's thesis proposes backtracking, which I did not implement. Increasing to hire dimensions to better represent learning may need complications of the engine. The Flask API exposes internal data structures directly, creating tight coupling between frontend and backend, a better approach could use explicit API schemas like Pydantic or OpenAPI to provide a stable contract.

6.3 Real-World Testing and Impact

I tested the system during several workshops I gave and made two friends test it for workshops they were giving. The auto-generated full orchestration graph wer not personalised enough but both appreciated manually creating their orchestration graphs, one even adding an activity. LLM-generated examples received mixed feedback: the exercises given were not the best (although treated subjects were complex) but the examples were pertinent and the sources provided of great use. Maybe refining the prompts would be a good domain to explore.

References

[1] Pierre Dillenbourg. "Orchestration Graphs and Modeling of Orchestration". In: *Orchestrating Learning: A Perspective* (2011), pp. 1–24.

[2] Luis P. Prieto et al. "Orchestrating Technology-Enhanced Learning: A Literature Review and a Conceptual Framework". In: *International Journal of Technology Enhanced Learning*. Vol. 3. 6. 2011, pp. 583–598.

[3] Samuel Belisle. *Orchestration Graphs for Educational Lesson Planning*. Tech. rep. Previous work on orchestration graph implementation. CHILI Lab, EPFL, 2024.