universität innsbruck
Department of Computer Science

QUALITY ENGINEERING

BlockQuiz
Web-Based Block Programming Platform with Auto-Grading for Students

Fabio Plunser

B.Sc. Thesis

Supervisor: Ass. Prof. Dr. Michael Vierhauser

1st June 2026

Abstract

This thesis presents the design and implementation of BlockQuiz, a web-based learning platform for block-based programming exercises aimed at children aged 8–12. The work is motivated by the increasing importance of computational thinking in school curricula and, in particular, by the Austrian subject "Digitale Grundbildung", which creates demand for classroom-ready tools that offer guided learning tasks, immediate feedback, and a deployment model suitable for schools.

Existing systems such as Scratch, Code.org, Microsoft MakeCode, and Blockly Games demonstrate the educational value of block-based programming, but they do not combine all of the requirements that are central to this project. Some of them focus primarily on open-ended creativity, others on hardware programming or cloud-hosted delivery, and only a subset support tightly constrained exercises that can be assessed automatically. BlockQuiz addresses this gap through a teacher-oriented platform for authoring and delivering short programming tasks with clear learning goals, progressive hints, and deterministic auto-grading.

The implementation is based on SvelteKit and Google Blockly. It includes a client-side sandbox for running generated code, an authoritative server-side path that re-grades every submission inside an isolated subprocess, a typed exercise model with publish validation, a grading framework for input/output, turtle, and robot exercises, multilingual content, a teacher content-management system, and identity-provider-synchronised classes with optional OIDC and SAML Single Sign-On. The whole system follows a privacy-first deployment model that supports self-hosting in school networks.

The system was evaluated along two axes. A technical evaluation with an automated test suite and a database verification script confirms that the exercise engines, the grading layer and the submission pipeline behave correctly and that the implementation meets its requirements. A user evaluation with two pre-service teachers found the platform understandable and pedagogically appropriate for its target group and surfaced concrete usability improvements, several of which were addressed in response. The results indicate that a focused, privacy-conscious block-programming platform with automatic grading is both technically viable and positively received by the teachers expected to author its content.

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Ass. Prof. Dr. Michael Vierhauser, for his invaluable guidance, constructive feedback, and continuous support throughout this project. His expertise in software quality and educational technology significantly shaped the direction of this work.

I am also grateful to the Quality Engineering research group at the University of Innsbruck for providing an inspiring academic environment and the resources necessary to complete this thesis.

Special thanks go to all participants who took part in the usability evaluation, providing essential feedback that helped improve the platform's design and functionality.

Contents

List of Figures

List of Tables

Listings

Declaration

By my own signature I declare that I produced this work as the sole author, working independently, and that I did not use any sources and aids other than those referenced in the text. All passages borrowed from external sources, verbatim or by content, are explicitly identified as such.

Signed: . . . . . Date: . . . . . .

Declaration of AI Usage

AI tools were used in this project. They served as a coding assistant during implementation, helped search for related Scientific work, ran sentiment analysis and type checking on some of the thesis text and helped rephrasing a few sentences and paragraphs to make them less technical and easier to read. The author reviewed and edited every AI-assisted output. Final decisions about, content, structure and claims rest with the author.

1 Introduction

Computational thinking has emerged as a fundamental skill for the 21st century, increasingly recognized as essential for all learners regardless of their future career paths (40; 12). In Austria, this recognition materialized with the introduction of "Digitale Grundbildung" (Digital Basic Education) as a mandatory subject for secondary education in 2022/23, reflecting a broader European trend toward integrating computational concepts into primary and secondary curricula (5). As a result, schools need programming tools that fit actual classroom conditions. They must be understandable for beginners, manageable for teachers with limited programming experience, and suitable for an institutional context in which privacy, deployment simplicity, and curricular alignment all matter.

1.1 Motivation and Problem Statement

Many established block-based environments already support introductory programming quite well, yet they do not fully address this combination of requirements. Scratch (28) encourages creativity and experimentation, but its open-ended project structure makes it difficult to use for short, deterministically assessable classroom tasks. Microsoft MakeCode (2) offers a highly polished environment and a valuable bridge from blocks to text, but much of its design is centered on hardware integration and project-based exploration. Code.org (6) provides structured learning sequences and strong scaffolding, yet its deployment model and platform boundaries are less aligned with a privacy-first, self-hosted use case in European school settings. Teachers therefore face a practical gap between highly creative block environments on the one hand and the need for tightly guided, easy-to-manage learning tasks on the other.

This thesis addresses that gap through the design and implementation of BlockQuiz, a web-based platform for short block-programming exercises with automatic feedback. The central idea is to combine the accessibility of block-based programming with a constrained exercise model that reduces cognitive load, supports clear learning objectives, and enables deterministic grading. At the same time, the platform is designed for teachers who may not have a strong programming background and for schools that prefer local or self-hosted deployment over dependence on external services.

1.2 Goals and Contributions

The goal of the project is not to replace general-purpose creative environments, but to provide a complementary platform for guided practice. BlockQuiz focuses on short exercises in which the available blocks, the expected interaction, and the grading criteria are defined in advance. This structure allows the platform to deliver immediate and meaningful feedback while also giving teachers a manageable authoring workflow. The thesis therefore contributes a constrained exercise model for beginner-oriented programming tasks, a browser-based sandboxing and grad-

ing architecture, a teacher-oriented content management approach for exercise authoring, and a privacy-conscious deployment model that supports classroom use without relying on third-party cloud services.

1.3 Target Audience

The primary audience of the platform consists of children aged 8–12, a group for whom block-based interaction remains especially appropriate because it removes syntax barriers and keeps attention on logic, sequencing, repetition, and conditional behavior. A second important audience consists of teachers who need to create, organize, and evaluate exercises without having to program the platform itself. Finally, the system also targets administrators or school IT staff who need a solution that can be deployed and maintained with limited operational complexity.

1.4 Scope and Limitations

The implementation covers three exercise types, namely input/output style tasks, turtle-graphics exercises, and grid-based robot puzzles, together with a course model, a class model that maps cohorts of students onto courses, multilingual content, account management with self-service password reset, optional OIDC and SAML Single Sign-On for institutional deployments, and a teacher-facing authoring workflow. The platform emphasizes constrained tasks, client-side sandbox execution and Docker-based deployment. It does not attempt to provide a complete learning management system or implement a broad range of domain specific simulators.

1.5 Thesis Structure

The remainder of this thesis follows the development of the platform. Chapter 2 reviews the educational and technical background, discusses related platforms, and situates the work in existing literature on block-based programming and automatic feedback. Chapter 3 derives the requirements from the target group, the classroom context, and the intended deployment model. Chapter 4 then explains the resulting system design and the main architectural decisions. Chapter 5 describes the implementation of the platform in more detail, including the integration of Blockly, the execution sandbox, and the grading framework. Chapter 6 presents a technical and exploratory evaluation of the implemented system. Finally, Chapter 7 summarizes the thesis, reflects on limitations, and identifies directions for future work.

2 Background and Related Work

This chapter relates BlockQuiz to the literature it draws on. It reviews the research on block-based programming and learning, surveys related platforms and lays out the educational theories that informed the design.

2.1 Block-Based Programming and Learning

Block-based programming environments represent program structure through shaped, snappable pieces instead of textual syntax. The shapes encode which constructs fit together. Loop blocks accept statements, expressions return values into matching sockets and the editor refuses combinations that the underlying language would reject as a parse error. Maloney et al. describe the Scratch environment around several specific design choices. liveness with no compile or run mode, tinkerability through clickable blocks, visible execution, a deliberate absence of error messages because block shapes prevent syntactically meaningless connections, concrete data through on-stage variable monitors and a minimised command set (19). Resnick and colleagues frame the wider goals as three design principles for Scratch. Making it `more tinkerable, more meaningful and more social` than other programming environments (28).

Weintrop and Wilensky followed high-school students for ten weeks in an introductory programming course that began in Snap! (33) and transitioned to Java and they asked students what they themselves thought of the two modalities (39). Their analysis identified four reasons students gave for block-based programming being easier. The natural-language labels on blocks, the visual shape and layout of the blocks, the ease of composing programs through drag and drop and the use of blocks as memory aids since the toolbox lets a student browse the available commands. About 58 percent of the 84 mid-study responses identified blocks as easier than the text-based equivalent. Students also named drawbacks. Blocks felt less authentic, less powerful and less suited to trial-and-error exploration than text-based programming. The paper is qualitative. Its design implication is not that blocks are simply better. The authors argue that the question for a learning environment is how to keep the introductory benefits while staying honest about where blocks get in the way.

Several empirical studies have looked at what learners actually take away from short Scratch interventions. Kalelioglu and Gülbahar ran a five-week, one-hour-per-week unit with 49 Turkish fifth-grade students and reported mixed results. Their quantitative measure of problem-solving skill did not show a significant change after the unit, while the qualitative responses from the children themselves were positive about Scratch and about programming as an activity (16). Saez-Lopez and colleagues ran a longer two-year case study with 107 primary students across five Spanish schools and reported significant improvements in computational concepts such as sequences, loops, parallelism and events, together with positive attitudes toward learning with the platform (29). Meerbaum-Salant et al. developed Scratch-based learning materials for middle-school students and evaluated them in normal classes at two schools (20). Combining written tests with classroom observation, the study reported that students could successfully

learn key computer-science concepts in Scratch but also ran into recurring problems around initialisation, variables and concurrency. Across these studies the recurring pattern is that the measured learning gains are more modest than the positive responses from learners would suggest.

2.1.1 Computational Thinking as a Learning Goal

The wider research frame for these studies is `computational thinking`. Wing framed the term as a fundamental skill for everyone, characterised as the thought process of formulating a problem and expressing its solution so that an information-processing agent can carry it out (40). Grover and Pea reviewed the six years of research after Wing's article and reported on the state of the field, the gaps in the research and priorities for future inquiry into K-12 (primary and secondary school) computational thinking (12). Brennan et al. proposed an operational framework for studying computational thinking through Scratch (4). They divide the construct into three layers. Computational concepts cover sequences, loops, parallelism, events, conditionals, operators and data. Computational practices cover the way learners build a program over time. Being incremental and iterative, testing and debugging, reusing and remixing and abstracting and modularising. Computational perspectives describe how learners position themselves in relation to the medium, including expressing themselves, connecting with others and questioning the technology around them. The Brennan and Resnick framework matters for the present work in two ways. First, it argues that debugging is not a side activity but a core computational practice. A learning environment that shows only a pass or fail signal forfeits a large part of the learning opportunity. Second, the framework treats the artefact and the process around it as separate sources of evidence. A platform that records what a learner did, in addition to whether they succeeded, produces richer data than one that records outcomes alone.

2.1.2 Educational Block-Based Environments

Scratch

Scratch (30), developed at the MIT Media Lab, is one of the most influential systems in this area (28). Its educational strength lies in the way it connects programming to storytelling, games, animation and personally meaningful projects. The Scratch website hosts a large community in which learners share and remix projects, which Brennan and Resnick drew on in their work on computational-thinking assessment through project portfolios, artefact-based interviews and design scenarios (4). The same openness that makes Scratch pedagogically rich makes it difficult to use for short tasks with clear expected outcomes. Scratch does not ship with a deterministic auto-grading mechanism. Weintrop and Wilensky note that students also perceive block environments as less suited to text-style trial-and-error than text-based programming, an observation that matters for any short-task setting in which a quick second attempt is part of the intended learning loop (39).

Microsoft MakeCode

Microsoft MakeCode (22) extends the idea of block-based learning toward hardware and project-based exploration. It offers polished editors, simulators and a transition path from blocks to JavaScript or Python. This makes it particularly useful in contexts where physical computing or the blocks-to-text progression is part of the learning goal. However, its overall focus differs from a platform centered on short, tightly constrained exercises. In comparison to the present

work, MakeCode is less concerned with teacher-authored assessment tasks and more concerned with broad exploratory creation in hardware-oriented settings.

Google Blockly

Google Blockly is best understood not as a complete learning platform but as a library for building visual programming interfaces (11). This makes it particularly relevant for custom educational systems. Blockly provides a mature rendering and connection model for blocks, configurable toolboxes, internationalization support, and code generators for multiple target languages. Because it is a library rather than a fixed educational product, it enables the construction of environments in which the available blocks, the surrounding user interface and the grading logic can be aligned with specific pedagogical goals. For BlockQuiz, Blockly therefore serves as a technical foundation rather than as an end-user product.

Code.org

Code.org (6) demonstrates how strongly structured learning paths can support school teaching. Its courses combine guided exercises, immediate feedback, and clear curricular progression, which makes it an important reference point for guided beginner instruction. In contrast to more open creative environments, Code.org is designed around sequenced lessons and classroom management. This makes it particularly relevant as a comparison for BlockQuiz, since both systems emphasize guided tasks over unrestricted project work. However, Code.org is primarily delivered as a centralized online platform and is less focused on self-hosted, privacy-conscious deployment in local school environments.

Open Roberta

Open Roberta (24) is especially relevant from a European and school-oriented perspective. Developed as a browser-based platform for educational robotics, it was designed to reduce technical barriers for teachers and learners by avoiding complex local installation and configuration steps. Its visual programming model, accessibility for beginners, and compatibility with classroom contexts make it a valuable comparison point for BlockQuiz. At the same time, Open Roberta is centered more strongly on programming robots and boards, whereas BlockQuiz focuses on constrained programming exercises with integrated auto-grading and teacher-authored task sequences.

LEGO Mindstorms

LEGO Mindstorms (17) is one of the best-known robotics platforms in introductory programming education and has played an important role in bringing programming into schools through hands-on, tangible activities. Its strength lies in the connection between software and physical devices, which can be highly motivating for learners. In educational terms, it shares with Open Roberta a strong emphasis on robotics, experimentation, and visible program behavior. However, this hardware-centered approach also distinguishes it from BlockQuiz, whose main purpose is not robot control but the delivery of guided, browser-based programming exercises that can be deployed and assessed without requiring dedicated physical equipment.

2.1.3 Comparison Summary

Taken together, these platforms show that there is no single dominant design for block-based programming education. Scratch foregrounds creativity, while MakeCode, Open Roberta and

LEGO Mindstorms bridge visual and textual programming in hardware-oriented contexts. Code.org offers strongly scaffolded instruction and Blockly acts as an enabling framework. BlockQuiz positions itself with a different focus where constrained exercises, teacher authoring, automatic assessment and privacy-conscious deployment need to coexist.

The matrix in Table 2.1 shows that BlockQuiz does not compete with these platforms on creative freedom. It instead occupies a specific intersection that is otherwise empty. Per-exercise toolbox constraints under a GUI authoring workflow, automatic grading, structured per-attempt analytics that go beyond completion counts and a deployment model that is genuinely self-hostable inside school infrastructure rather than only theoretically open source. Scratch and Blockly Games optimize for open creation rather than guided practice. MakeCode targets hardware tutorials. None of them combine teacher-side authoring of constrained tasks with on-prem deployment and learner-process analytics.

Table 2.1: Comparison of Block-Based Programming Platforms

Platform	**Auto-grading**	**Teacher CMS**	**Self-hostable**	**Open Source**
Scratch	No	No	No	Yes
MakeCode	Partial	No	No	Yes
Code.org	Yes	Limited	No	Partial
Blockly	N/A (library)	N/A	Yes	Yes
BlockQuiz	Yes	Yes	Yes	Yes

2.2 Automated Assessment in Programming Education

Automated assessment has a long history in programming education and provides immediate feedback at scale. Early work in the field dates from the 1960s, with classroom systems described by Hollingsworth (14), Naur (23), and Forsythe and Wirth (10). Reviews of more recent decades document the continued development of the area (1; 7; 15; 21). Ihantola and colleagues survey systems from 2006 to 2010 and discuss the pedagogical and technical features they support (15). Across these reviews, the recurring observation is that most tools assess correctness through dynamic test execution.

2.2.1 Test-Based Grading

The most common approach executes student code against a set of test cases (7; 1), comparing actual output to expected outputs. This is well suited to input/output exercises in which correctness can be defined through a clear mapping from input to output. In practice, such grading must deal with issues such as visible versus hidden tests, normalization of whitespace or upper/lower case and the distinction between purely summative checking and formative feedback. The systematic review by Messer et al. shows that this form of grading remains the dominant approach in automated programming assessment because it is reliable for constrained tasks and scales well to large cohorts (21).

2.2.2 State-Based Grading

Not all beginner tasks produce meaningful textual output. In graphical or simulation-based exercises, correctness is often better captured through the final state of the system or through the sequence of actions that produced the state. Turtle graphics are a good example: a learner may arrive at the same picture through multiple valid block sequences, which means that grading can be based either on the final geometric state or a more prescriptive comparison of executed

commands (18). For this reason, BlockQuiz combines simple command-based and state-based grading strategies, depending on the intended learning objective.

2.2.3 Benefits of Immediate Feedback

Formative-assessment research treats feedback as one of the most influential factors on classroom learning. Black and Wiliam reviewed the empirical evidence on classroom assessment and concluded that strengthening the feedback students receive about their learning yields substantial learning gains (3). Hattie and Timperley summarised feedback research at four levels (tasks, process, self-regulation, self) and reported that feedback aimed at the task and at the learner's process is among the most powerful influences on learning (13). Shute synthesised the literature on formative feedback and reported that the most effective timing depends on task complexity and learner ability. Immediate feedback tends to be more efficient for low-achievement learners and for simple or procedural tasks, whereas delayed feedback can favour high-achievement learners and transfer to new problems (32). For the BlockQuiz target setting, beginner-level learners working on short, procedural exercises, Shute's analysis points toward the immediate end of the timing range.

2.3 Learning Theory Foundations

The design of BlockQuiz is informed by several well-established educational theories. These theories do not determine the implementation directly, but they provide a conceptual framework for understanding why constrained block-based tasks, progressive hints and visual feedback are appropriate design choices.

2.3.1 Constructionism

Seymour Papert's constructionism argues that learning becomes especially powerful when students actively construct artifacts that are meaningful to them (25). Programming can therefore be understood not merely as a way of issuing instructions to a machine, but as a medium through which learners explore ideas, observe consequences and build mental models. Although BlockQuiz focuses on shorter and more guided tasks than open creative environments, it still adopts the constructionist insight that programming is most engaging when learners can see the effects of their decisions directly. Turtle graphics, in particular, make this visible by connecting code to movement and shape.

2.3.2 Cognitive Load Theory

Sweller's Cognitive Load Theory distinguishes between the inherent difficulty of a task, extraneous load introduced by presentation or interface design and beneficial load associated with meaningful learning activity (37). One of the main strengths of block-based programming is that it reduces extraneous load by eliminating syntax errors and making valid composition visible. BlockQuiz extends this idea by constraining the toolbox on a per-exercise basis. Instead of exposing a large library of irrelevant blocks, the system presents only the constructs that are needed for a given task, which supports focused learning and reduces distraction.

2.3.3 Zone of Proximal Development

Vygotsky's Zone of Proximal Development describes the range in which learners can complete a task with guidance but not yet entirely on their own (38; 31). This idea is relevant for platforms

that provide progressive support rather than all-or-nothing assessment. In BlockQuiz, the use of staged hints, starter blocks and ordered task progression is intended to keep exercises within that zone. The platform does not merely judge whether an answer is correct, it is also designed to support learners while they move toward a correct solution.

2.3.4 Piaget's Stages of Cognitive Development

The 8 to 12 age range sits mostly inside Piaget's concrete operational stage (roughly ages 7 to 11) and overlaps the start of the formal operational stage at the upper end of the band (26). Learners in the concrete operational stage reason confidently about concrete objects and relationships but find purely abstract symbolic reasoning harder. Block-based programming maps well onto this stage. The blocks are concrete objects on the screen. Their shapes and colours stand in for the relationships that text-based syntax encodes only in symbols. The turtle and grid-robot exercises take this further. The program acts on a visible object whose position, orientation and collected items the learner observes directly. BlockQuiz uses this mapping deliberately. The visual outcome of a program is the channel through which learners check their own work, before they consult any written feedback.

2.3.5 Summary

Taken together, these theories shape four design moves in BlockQuiz. Per-exercise toolbox constraints follow Cognitive Load Theory and the empirical findings on block environments. Progressive hints follow Vygotsky and Bruner. Visible, immediate outcomes follow constructionism and Piaget. The next chapter derives the requirements that turn these design moves into concrete platform features.

3 Requirements Analysis

This chapter derives the requirements for BlockQuiz from the intended learner group, the realities of classroom teaching, the desired self-hosted deployment model and the technical consequences of executing learner programs in the browser. The purpose of the requirements analysis is not to produce an exhaustive catalog of all imaginable platform features. Instead, it identifies the set of capabilities that a realistic prototype must provide in order to support guided block-based programming exercises in a school context.

3.1 Stakeholders

The most important stakeholder group consists of learners between the ages of 8 and 12. For them, the platform must present programming tasks in a form that is visually clear, forgiving of mistakes and capable of returning immediate, understandable feedback. Teachers form the second central stakeholder group. Their interest lies less in the technical implementation than in whether they can create, adapt and organize exercises without touching source code. School IT administrators represent a third perspective, since they need a deployment model that is easy to operate, simple to back up and compatible with school privacy expectations. Finally, the thesis itself also imposes a developer-facing perspective. The architecture should remain maintainable, extensible and honest about its security assumptions.

3.2 Functional Requirements

Exercise Definition (FR-1)

At the center of the platform is the exercise model. Each exercise must have a stable identity together with learner-facing metadata such as title, description, exercise type, topic and difficulty. Because the platform focuses on guided tasks rather than unrestricted creation, an exercise must also define the subset of blocks that is available to the learner. When appropriate, it may include a starter workspace and one or more hints that support gradual progression through the task. Since the platform is intended for bilingual use, exercise content must support both German and English.

Supported Exercise Types (FR-2)

The platform focuses on exercise types that cover complementary learning situations. Input/output tasks provide a way to model simple algorithmic problems with deterministic textual results, while turtle-graphics and robot-grid exercises make program behavior visible through movement and spatial state. At the same time, the exercise architecture should not be so specialized that the addition of future exercise types would require a complete redesign.

Authoring and Course Organization (FR-3)

Authorized users must be able to create and edit exercises through a teacher interface. They also need a way to combine exercises into ordered courses so that tasks can be used as part of a broader learning path instead of only as isolated examples. Reuse is an important requirement here. The same exercise should be usable in more than one course. In addition, teachers need a way to make a course available to a whole class of students at once rather than only to individual accounts, because handling enrolment one learner at a time does not scale to a typical classroom. The system should therefore support a class abstraction that can be linked to a course so that every member of the class receives the assignment automatically. Since authoring mistakes can easily lead to confusing learner experiences, the system should also support previewing an exercise before it is published.

Execution and Feedback (FR-4)

Learner block programs must be translated into executable code and run inside a restricted environment. The execution model therefore needs a controlled API surface through which generated programs can interact with the exercise, while preventing direct access to browser capabilities that are outside the task domain. Because beginner code often contains accidental infinite loops or other non-terminating behavior, the runtime must also enforce time or step limits. After execution, the learner should receive feedback immediately so that the platform supports an iterative learning process.

Auto-Grading (FR-5)

Automatic grading is one of the defining features of BlockQuiz. Input/output exercises therefore require a grading mechanism based on specified test cases, whereas turtle and robot exercises require comparison of command traces, final state, or both. The grading result must at least distinguish between successful and unsuccessful attempts, but a binary pass/fail signal alone is not sufficient for the educational purpose of the platform. The system should therefore present feedback that is more informative and helps learners understand why a solution did or did not satisfy the task.

User Modes and Progress (FR-6)

The platform must support authenticated teacher access for authoring and management tasks, yet the learner side should remain flexible enough to support guest use where possible. This is particularly relevant for privacy-conscious deployment scenarios or informal try-out settings. Exercise progress must be persisted in a way that matches the chosen user mode, whether this means local storage for guest use or server-side persistence for authenticated users. Where a school already operates an identity provider, authentication should optionally route through that provider rather than requiring a parallel password directory, while still supporting a local email-and-password fallback for administrator recovery. When the identity provider exposes group claims, the system should be able to mirror those groups into its own class model so that institutional cohorts and BlockQuiz classes stay in sync without manual roster work.

Internationalization (FR-7)

Internationalization applies both to the general interface and to exercise content. The user interface should support German and English and exercises should be available bilingually with

a defined fallback behavior where a translation is incomplete. Since learners interact with programming blocks as part of the same environment, Blockly labels should be localized consistently as well.

3.3 Non-Functional Requirements

Security (NFR-1)

Student-generated code must not receive unrestricted access to the document object model, browser storage, or general network functionality. Data crossing trust boundaries, especially authoring input and remote API requests, must be validated carefully. In addition, authoring and administrative features require role-based access control so that privileged functions remain separate from the learner experience.

Privacy (NFR-2)

Privacy requirements follow directly from the intended school context. The system should minimize personal-data collection, support self-hosting inside school infrastructure and remain usable without mandatory reliance on third-party services wherever feasible. These requirements are not peripheral operational details; they shape the architecture and deployment strategy from the start.

Usability (NFR-3)

Since the main learner group has limited prior programming experience, the user interface must remain clear and age-appropriate. This includes constrained toolboxes, concise instructions and feedback that avoids unnecessary technical jargon. Usability is particularly important because the educational value of the platform depends on whether children can stay focused on the programming problem rather than becoming stuck in the interface.

Performance (NFR-4)

Interactive classroom use requires that exercises load quickly and that feedback appears without disruptive delays. The execution sandbox should therefore fail fast when programs exceed defined limits and the application as a whole should remain responsive on typical school hardware.

Maintainability and Extensibility (NFR-5)

The prototype should provide a technical foundation that can be extended after the thesis. This requires a separation between user interface, execution logic, grading and persistence, as well as exercise abstractions that can accommodate future task types. The system should also be reproducible across environments with minimal configuration effort so that deployment and further development do not depend on undocumented local setup steps.

3.4 Representative Use Cases

The core value proposition of the platform can be summarized through a small set of representative use cases. The first is the learner workflow: a child opens an exercise, assembles a block program, runs it and receives immediate feedback. The second is the teacher workflow: a teacher defines task text, available blocks, hints and grading configuration for a new exercise. The third

is course composition, in which a teacher selects existing exercises and arranges them into an ordered sequence. The fourth is local deployment, where an administrator sets up the application without depending on external cloud infrastructure. Together, these use cases capture the intended prototype scope. More advanced scenarios, such as deep analytics or integration with institutional learning-management systems, remain outside the current thesis.

4 System Design

This chapter turns the requirements from the previous chapter into a concrete system design. It records the principal architectural choices, lays out the data model, places the pedagogical anchors that ground each choice in learning theory and discusses the security and privacy constraints that shape the deployment posture.

4.1 Design Decisions

This section records the principal architectural choices behind BlockQuiz. The focus is on decisions that strongly influence the implementation. The visual editor technology, the execution model, the persistence layer and the separation of concerns between exercise definition, execution, grading and presentation.

4.1.1 Visual Programming Framework: Google Blockly

Table 4.1: Block Editor Selection

Option	**Pros**	**Cons**
Google Blockly	Mature, documented, accessible, customizable toolbox	Larger bundle size
Custom implementation	Full control, smaller size	Development time, maintenance burden
Scratch Blocks	Familiar Scratch aesthetics	Less flexible, tied to Scratch ecosystem

Blockly was selected because it offers the best balance between maturity and adaptability. Its toolbox model directly supports the pedagogical goal of restricting the available blocks on a per-exercise basis, which helps reduce unnecessary complexity for novice learners. The same integration can also be reused in both the learner-facing player and the teacher-facing CMS. A custom implementation would have provided greater control, but the additional development effort would have been difficult to justify within the scope of the thesis. Scratch Blocks, while visually familiar, would have constrained the platform more strongly to an ecosystem that is not primarily designed for this kind of custom guided-learning workflow.

4.1.2 Execution Mode: Client-Side Sandbox

The execution model is intentionally optimized for classroom responsiveness and low deployment complexity. A client-side iframe sandbox therefore represents a pragmatic compromise. It allows learners to receive immediate feedback without requiring the server to execute every submission,

which keeps the overall system lighter and easier to deploy. At the same time, this decision does not remove all security concerns. The design therefore pairs the responsive client-side path with a separate authoritative path on the server. When an authenticated submission is persisted, the same generated code is re-executed inside an isolated Bun subprocess worker that has no access to the database or the application session. The browser sandbox provides immediate feedback, while the subprocess re-execution provides a trusted result that becomes the stored score. This layered execution model is the central reason the implementation can combine fast learner interaction with a meaningful trust boundary, without adopting the operational complexity of a per-submission container service.

Table 4.2: Code Execution Options

Option	**Pros**	**Cons**
Client-side (iframe sandbox)	No server load, immediate response	Limited isolation compared to server-side, potential security risks
Server-side (VM/Container)	Better isolation	Added complexity, latency
WebAssembly sandbox	Best isolation	Significant development effort

4.1.3 Persistence Layer: SQLite with Drizzle ORM

Table 4.3: Database Selection

Option	**Pros**	**Cons**
SQLite	No setup, embedded, portable	Single-writer limitation
PostgreSQL	Scalable, concurrent writes	Requires server, more complex

SQLite is well suited to a prototype and to modest self-hosted school deployments because it keeps operational overhead low. When combined with Drizzle ORM, it also provides a type-safe development workflow and explicit schema management. A more scalable database system would only become necessary if the platform were extended beyond the intended deployment model.

4.1.4 Client-Server Boundary: SvelteKit Remote Functions and Async Svelte

A non-trivial design decision in this project was how the browser should talk to the server. Two SvelteKit features that were experimental at the time of writing made it possible to collapse the conventional split between REST endpoints, server load functions and form actions into a single typed boundary and the project opted into both.

The first is the `remote functions` (35) feature. Server-only code in a file ending in `.remote.ts` can export three kinds of functions: `query` for read operations, `command` for mutations and `form` for HTML-form-style mutations that integrate with progressive enhancement. Each function is declared with a Zod schema for its input, so the same definition is the implementation, the validation rule and the TypeScript type that the client sees. The client imports the function directly, there is no intermediate REST contract. and SvelteKit transparently turns the call into a fetch under the hood, validates the payload, runs the function on the server and returns the typed result. It is more like a Remote Procedure Call than REST, but it provides a clean and

typesafe client-server boundary with minimal boilerplate. The practical effect for BlockQuiz is that the per-route boundary that would otherwise be expressed as a handful of REST endpoints, a Zod schema and client wrapper collapses into a single typed function call. The remote layer in `src/lib/remote` contains the project's full server-side contract. Courses, exercises, classes, users, settings, logs, i18n and auth. Each in its own *.remote.ts file with role-checked query and command exports.

```
/**
 * Get a single course by ID. Teachers/admins receive exerciseIds and userIds in
 * addition to the base course row; learners get only the base row.
 */
export const getCourse = query(z.string(), async (id) => {
  const course = await loadCourseRow(id);
  if (!course) error(404, 'Course not found');

  if (!isTeacherOrAdmin()) {
    const actor = currentUser();
    if (actor) {
      void writeAuditLog({
        actorUserId: actor.id,
        action: 'course.open',
        category: 'user',
        details: { courseId: id }
      });
    }
    return course;
  }

  return loadCourseWithRelations(course);
});
```

Listing 4.1: SvelteKit Remote Function Example

The second is the experimental `async Svelte` (34) compiler option. With this option, a Svelte component can await a remote function anywhere in the script block or the markup. Most importantly inside of a `$derived` rune. SvelteKit handles suspension, hydration and re-fetching on its own. Where a traditional setup would require a `+page.server.ts` load function to pre-fetch the data, an explicit prop on the page component and a fall-back skeleton the same flow now reads as:

```
<script lang="ts">
  let course = $derived(await getCourse(courseId));
</script>

<!-- Markup -->
{#each publicCourseList as course}
  <CourseCard {course} />
{/each}
```

Listing 4.2: Async Svelte Data Fetching

By treating the server contract as something the component can directly await, the layer of glue between the *server* and the *component* disappears.

If an await fails in a script tag it will throw an error. That error can be caught by putting the entire component inside of an `<svelte:boundary>`. Fetching of data, showing load state and error handling can also be part of the markup directly. Because Svelte itself is a language with a compiler we do not have to await in the script tag block some of the page rendering (if fetching is not done correctly). But fetch directly in the markup

```
<svelte:boundary>
  {#await getPublicCourseList({})}
    <p>Loading...</p>
  {:then publicCourseList}
    <CourseList {publicCourseList} />
  {:catch error}
    <p>Error loading courses: {error.message}</p>
  {/await}
</svelte:boundary>
```

Listing 4.3: Svelte Async Boundary

Remote functions also provide built-in support for progressive enhancement and single flight mutations. A mutation that adds a course, for example, has to tell every other place that displays courses to re-run its query. BlockQuiz expresses this declaratively by chaining .update(queryFn) onto the command call site, with a mirroring requested(q,n). refreshAll () on the server side. The mutation declares which queries it invalidates, SvelteKit recomputes exactly those and the rest of the page is left undisturbed. This is single flight mutation without it the component would need to call await query.refresh () manually, and then instead of having one POST call which returns the updated Data we get a POST and then a GET request.

```
// courses.remote.ts
export const createCourse = command(createCourseSchema, async (data) => {
  const user = requireTeacherOrAdmin();
  const result = await createCourseImpl(user, data);
  await requested(getCourses, 10).refreshAll();
  return result;
});
```

Listing 4.4: Single Flight Mutation Example - 1

```
// CourseEditor.svelte
const result = await createCourse({
          content: formData.content,
          published: formData.published,
          demo: formData.published && formData.demo,
          exerciseIds: formData.exerciseIds,
          userIds: formData.userIds,
          classIds: formData.classIds
        }).updates(getCourses);
```

Listing 4.5: Single Flight Mutation Example - 2

The response of createCourse already contains the new course, so the client does not have to refetch and because of sveltes rune system and signal based reactivity the UI is updated with the list automatically.

These two features are explicitly experimental in upstream SvelteKit currently and their use in BlockQuiz is therefore a project-scope bet rather than a neutral framework choice. However, they do align so well with the Meta framework concept that many other projects are already using them in production. They are mentioned here because they shaped both the file layout and the interaction model in components. Also because they are still experimental we note that should either feature change shape before its stable release, that there would be some necessary refactoring. But it is unlikely that the Svelte-Core team are going to abandon this approach since the response to the primitives is extremely positive, so that the whole code needs to be refactored is not a likely scenario.

4.2 Architectural Overview

The system follows a layered architecture in which the user interface, exercise logic, grading and persistence remain conceptually separate. This separation is visible in the code base. Svelte components are primarily responsible for presentation, whereas execution, grading, typing and persistence are handled in dedicated modules. The result is not only easier to reason about technically, but also easier to explain as part of the thesis.

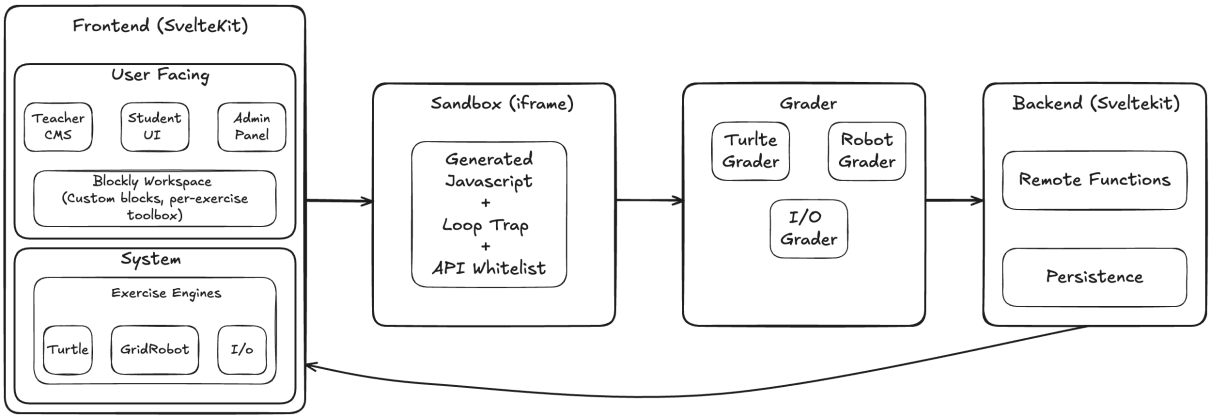

Figure 4.1: System Architecture Overview

4.2.1 Frontend Layer

User Facing

The frontend is implemented with SvelteKit and primarily supports different views depending on the user permissions. The learner interface presents an exercise together with Blockly workspace, a visual or textual output area and the resulting feedback. The teacher interface supports exercise authoring, review, course composition and a dedicated analytics dashboard that drills from a list of classes down through each class's students and from there into a per student per class detail view. The admin interface provides user management, log inspection and settings for IdP (Identity Provider) configuration and other application-level concerns. But the admin interface is a secondary concern for the thesis. The central design focus remains on the learner workflow and the teacher authoring, class-management and analytics processes.

System Components

Beneath the user-facing views the frontend hosts the editing and execution machinery. The Blockly workspace component is the shared visual editor used by both the learner player and the teacher CMS. It is configured per exercise: each exercise declares a small allow-list of blocks and a starter workspace, and the workspace component instantiates Blockly with exactly that toolbox and generator mapping. The exercise layer models the behavior of each supported task type. The Input/Output engine provides controlled input consumption and output collection for text-oriented tasks. The turtle engine manages position, orientation, pen state, drawing history, configurable start and finish points, walls that the turtle collides with at run time and an apple the turtle can eat. The robot engine manages a discrete grid with start position, direction, walls and targets and exposes fixed-action movement blocks tailored to grid puzzles rather than parameterized turtle commands. Each exercise type is associated with an exercise definition, a Blockly toolbox and generator mapping, a restricted execution API and a grading strategy that matches the exercise semantics.

4.2.2 Sandbox Layer

The sandbox executes generated code in a restricted context. In practical terms, execution takes place in an iframe with restrictive sandbox settings and all communication with the main application occurs through a message-based interface. Learner programs therefore do not receive arbitrary access to browser capabilities. They can only call the exercise-specific commands exposed by the application. Runtime guards additionally limit execution time and help detect non-terminating behavior. The resulting design does not claim absolute security, but it does establish a clear and limited trust boundary that is appropriate for the educational deployment context.

4.2.3 Grading Layer

The grading layer evaluates exercise outcomes after execution. For Input/Output tasks, this means comparing actual and expected output across defined test cases. For turtle exercises, it means comparing either the executed command trace or the resulting final state. Keeping grading separate from execution and presentation makes the platform easier to extend and allows the grading logic to be discussed independently of the user interface.

4.2.4 Backend Layer

The backend is the authoritative side of the system. It is also implemented in SvelteKit, but expressed exclusively through remote functions: a `.remote.ts` file per domain (courses, exercises, classes, users, settings, logs, i18n, auth) exports role-checked `query` and `command` entry points with Zod-validated input. This is the same boundary discussed in the design decisions above, and it is the only path through which the frontend reads or mutates server state. Persistence sits behind those functions in a single SQLite database accessed through Drizzle ORM. The schema covers the core educational entities (users, exercises, courses, course-exercise relations, classes and attempts) together with the supporting tables for versioning, audit logging and IdP group reconciliation introduced in the data model section. The backend is also where the authoritative grading path runs: when an authenticated submission arrives, the same generated program is re-executed in an isolated Bun subprocess worker that has no access to the database or session, and the trusted result returned by that worker is what gets stored as the attempt's score.

4.3 Pedagogical Design Rationale

The design decisions described above are not purely technical. They operationalize specific findings from learning theory and child-computer interaction research. The connections deserve to be stated explicitly because the platform serves an educational audience, not a generic coding tool.

The first connection runs between the per-exercise toolbox constraint and Sweller's `Cognitive Load Theory` (37). A toolbox of dozens of unrelated blocks forces the learner to make selection decisions unrelated to the task. The cognitive resources spent on selection are not available for solving the actual problem. Weintrop and Wilensky observe a related pattern in interviews with high-school students working in block-based environments. The browsable toolbox is a help when it is small, but a large set of available commands becomes a separate problem to navigate (39). BlockQuiz lets each exercise declare a small allow-list of blocks, typically five to ten. A publish-validation gate enforces that every block in the starter workspace appears in that allow-list. The learner sees only blocks that are pedagogically relevant to the current task.

The second connection runs between the progressive-hint mechanism and Vygotsky's `Zone of Proximal Development` (38), together with the related notion of scaffolding (41). Hints are not all visible at once. They unlock after a delay or on explicit learner request. The assistance stays close to what the learner needs at the moment. Each reveal event is captured per attempt, so the system reports back to the teacher how often hints were taken and at which point. A classroom intuition like "most learners struggled at exercise 3" becomes a measurable signal.

The third connection runs between the slow-motion replay feature and debugging as a core computational-thinking practice (4). Brennan and Resnick list testing and debugging as one of the computational practices in their framework. After a run, the learner plays the recorded command trace step by step at a chosen speed. The gap between the program the learner intended and the program they wrote becomes observable rather than abstract.

4.4 Age-Appropriate Design

Designing for children aged 8 to 12 requires more than scaling down an adult programming environment. At this age, learners reason confidently about concrete and visual representations but find abstract symbolic formalisms harder. Tasks need to connect to visible outcomes for younger learners. Older children in the band handle a step toward more abstract reasoning. A mixture of turtle-graphics tasks and simple input/output problems suits the range.

Read and Bekker argue that child-computer interaction is a field distinct from generic human-computer interaction (27). Children change developmentally at a rate adults do not. Adults mediate most of a child's interaction with a piece of software, in the home or in the classroom. The contexts of use differ, and so do the cultural assumptions about what counts as appropriate technology for children. These observations make designing for an eight-to-twelve learner a different exercise from designing for an adult user, even when the surface tasks look similar. Practical implications follow from this framing. The interface needs to be readable without dense text, give clear feedback on every action, and accommodate the variability of fine-motor skills across the age range. Readability and immediate visual response are not cosmetic concerns. They are core usability requirements.

BlockQuiz applies these principles in several concrete places. The learner player shows one exercise at a time. The Blockly workspace fills a large part of the screen. The result panel produces a clear pass-or-fail signal with a short, friendly explanation in the learner's locale. Hint buttons are sized to be easy to tap on a tablet. The confetti celebration on a successful submission honours the `prefers-reduced-motion` media query and shortens itself when the

operating system asks for less motion.

4.5 Privacy and Data Protection

Privacy is an important design consideration when educational software serves children. Within the European Union, the GDPR restricts the processing of personal data and emphasises data minimisation, purpose limitation and privacy by design (9). For school software, this means that architectural choices about account models, storage location and reliance on external services are institutional and legal decisions as well as technical ones.

BlockQuiz adopts a privacy-by-design stance. Guest operation is first class. A learner who opens the demo page solves an exercise without an account, the progress lives in the browser storage on the learner's device. Local-progress storage stays under the learner's control. The deployment model is self-hosted. A single container with an embedded SQLite database, no required external services for code functionality and no outgoing connections from the implementation beyond optional email transports for password reset. This does not remove every compliance consideration, but it reduces the amount of personal data the platform has to collect and transfer.

Authenticated accounts store a small set of fields. Name, email address, role and active flag. Attempts store the workspace XML, generated code, grading result, locale and hint-reveal events. The attempt records are pseudonymisable for research export and stay on the school's own infrastructure under self-hosting. In comparison to cloud-centred educational platforms, this approach lowers the barrier to deployment in school networks where data sovereignty and operational control matter.

4.6 Technical Foundation

The technical implementation builds on SvelteKit and Svelte 5 (36). Their component model and reactive primitives suit an application that combines a rich frontend, custom editing interfaces and reusable logic for exercise engines. Reactive state lives close to the domain logic and integrates cleanly with user interface components. For persistence, the project uses SQLite together with Drizzle ORM (8). SQLite suits the intended deployment scope because it avoids the operational overhead of a separate database server. Drizzle ORM adds a typed and structured data model on top of SQLite. In a school environment or single-site deployment, the simplicity of SQLite is a practical advantage rather than a limitation.

Containerised deployment through Docker matches the wider goals of the system. The platform ships as a self-contained service with minimal installation complexity. School deployments where dedicated DevOps expertise is not available stay realistic. The Bun adapter and the `bun:sqlite` client keep the runtime choice consistent between development and production.

4.7 Data Model

The data model stays close to the implemented scope of the system. Its most important entities are users, exercises, courses, relations between courses and exercises, classes that group learners into cohorts and attempt or progress records. Users store account and role information wherever authenticated access is needed. Exercises contain the task definition and grading configuration. Courses organize exercises into ordered learning sequences and the course-exercise relations capture both ordering and reuse. Classes group students into cohorts that mirror real classroom groupings and a separate course-class relation lets a teacher hand a course to an entire class in a

single assignment. Enrolment is either per student or per class and is enforced in the UI. Classes themselves can be either administrator-maintained or sourced from an identity provider's group claims and reconciled on every SSO login, which keeps the BlockQuiz roster aligned with the school's IdP without an extra provisioning protocol. Courses carry both a `published` flag and a separate `demo` flag. The demo flag is gated on published=true so that an unpublished work-in-progress course cannot leak through the public surface. Attempts or progress records connect learners to exercises and preserve the outcome of their interaction with the platform.

The repository also contains supporting entities such as audit logs, version tables, and an IdP group discovery table that helps administrators audit which group claims exist before promoting them into classes, and translation tables. These are part of the broader application context, but within the scope of the thesis they remain secondary to the core educational entities of exercises, courses, classes, users and attempts.

4.8 Security Considerations

The main threats considered in the design are accidental or malicious infinite loops, unintended access to browser capabilities from learner code and unauthorized access to protected authoring features. The response to these threats is deliberately layered. Iframe-based sandboxing limits the execution context, command whitelisting constraints the API available to generated programs and timeout or step-based execution limits keep programs from running indefinitely. In parallel, validation of application input and API payloads protects the wider application, While role-based authorization separates learner, authoring and management privileges. Where a school already operates an identity provider, the same role boundary can be sourced from the IdP through the optional OIDC or SAML SSO path. Which re-evaluates the BlockQuiz role from IdP claims on every login and, in the same callback, reconciles the learner's class memberships against the IdP-claimed groups, so that revoked group membership and removed classes propagate without manual cleanup. The browser-facing trust boundary is additionally narrowed by a server-side response-header policy. Every HTML response receives a restrictive Content-Security-Policy, together with `X-Frame-Options, X-Content-Type-Options, Referrer-Policy` and the cross-origin isolation headers. The CSP is not maximally strict. These measures reduce risk for the intended educational system, but they do not justify absolute security claims, which is why the thesis treats the security model as adequate and explicit about its own boundaries.

5 Implementation

This chapter walks through the concrete implementation of BlockQuiz. It describes the project structure, the domain model, the Blockly integration, the dual-path execution pipeline, the grading layer, the learner player, the teacher CMS and the authentication and class machinery that ties the system to a school identity provider.

5.1 Implementation Scope

The implemented system covers the full learner flow, the teacher-facing authoring workflow and the technical infrastructure required for sandboxed execution and automatic grading. The system is broad enough to demonstrate the architecture described in Chapter 4 as a coherent whole rather than as a collection of isolated components. In practice, the strongest parts of the code base are the submission pipeline, the grading logic, the typed exercise model and the shared player used by both guest and authenticated learners.

The implementation is organized as a SvelteKit application with TypeScript. The frontend uses Svelte 5 components for the learner player, the teacher-facing CMS and the Blockly integration. Persistence is handled through SQLite with Drizzle ORM, while authentication, sessions and password reset are provided through Better Auth. Development and verification are performed through bun-based commands and the same runtime choice is also reflected in deployment through the Bun adapter, the **bun:sqlite** database client and the isolated Bun subprocess that performs server-side code execution.

5.2 Project Structure and Technology Stack

The repository is divided into route components, reusable interface components, typed domain modules, grading logic, sandbox infrastructure, remote functions and database access. This separation is visible in the implementation. The learner player and the CMS live in Svelte route components, while exercise normalization, grading, authoritative re-execution and persistence are handled in dedicated library modules.

This split proved useful because the project combines several concerns that would otherwise become tightly coupled. Visual programming, learner interaction, runtime control, server-side persistence and authorization. Using SvelteKit for both interface and server integration simplified this boundary, since the same framework can host route components and typed server-side commands. At the same time, the more critical logic such as grading and sandbox coordination is kept outside user-interface files, which improves testability.

The final stack also reflects the deployment goal of a privacy-conscious, self-hostable school system. The application runs without a separate database server, does not depend on third-party APIs for its core functionality and can be shipped as a single containerized web application.

5.3 Domain Model and Persistence

One of the most important implementation tasks was turning the originally broad platform idea into a concrete and typed exercise model. In the final implementation, the central runtime model implements three exercise types: `io, turtle` and `robot`. Each type has a dedicated configuration and grading strategy, while shared learner-facing properties such as localized content, hints, toolbox restrictions and starter workspaces remain available across all types.

The resulting model is stricter than a generic JSON-based exercise description. For example, `io` exercises define normalization rules and explicit input and output test cases, whereas `turtle` and `robot` exercises use canvas or grid configuration together with structured state, command, or path based checks. In practice, this made it easier to keep authoring, grading and persistence consistent, because each exercise type has a clear set of expected fields.

At the same time, the implementation keeps a compatibility oriented **config** object in addition to the typed branches. This is a pragmatic engineering compromise. It allows generic CMS and player components to access shared properties such as toolbox selection, starter XML, or hints through a stable shape, even though the runtime representation is type-specific. The result is slightly less pure than a fully separated model but it reduces UI duplication and kept the authoring surface manageable within thesis scope.

Table 5.1: Implemented exercise types in BlockQuiz

Type	Learner interaction	Grading basis	Current maturity
`io`	Text-oriented tasks with generated JavaScript reading input and producing output	Visible and hidden stdin/stdout tests with configurable normalization	Mature and used as a main exercise type
`turtle`	Visual movement and drawing on a two dimensional canvas	Command replay, target checks, final state, and path comparison	Mature and central to the implementation
`robot`	Grid based movement with walls, targets, and optional collectibles, using fixed-action step and turn blocks rather than parameterized movement	State, command, path and collect based checks	Implemented and seeded as a full third exercise type alongside `io` and `turtle`

The same emphasis on typed structure appears in persistence. The database schema contains the expected educational entities of users, courses, exercises, course-exercise relations and attempts, classes but it also stores validation snapshots, exercise version, translations and audit-related information. For the thesis, the most interesting table is the attempt record, because attempts form the common interface between learner interaction, grading, analytics and export.

Instead of storing only a score and a timestamp, the implementation persists the replayable learner artifact: workspace XML, generated code, serialized grading result, locale, hint events, and additional analytics metadata. This makes it possible to reconstruct what a learner submitted and not merely what score they received.

Another important detail is publish validation. The final implementation does not allow exercise publication unless specific validation criteria are met. Instead, exercises are validated against concrete conditions such as bilingual title and description, parseable starter XML, presence of test cases and at least one hidden test. This is not a complete formal verification of exercise quality, but it creates a meaningful boundary between authoring flexibility and learner-facing reliability.

Table 5.2: Main categories of persisted attempt data

Category	Stored values	Purpose
Identity and actor	Attempt id, exercise id, actor type, and either authenticated user id or guest client id	Distinguishes authenticated progress from local-first guest progress while keeping a uniform attempt structure
Replayable artifact	Workspace XML and generated JavaScript code	Supports restoration, inspection, export, and later re-display of the learner state
Grading result	Serialized result object, score, pass/fail status, and timestamps	Preserves the immediate grading outcome without having to recompute it for every view
Context metadata	Locale, hint reveal events, and analytics JSON	Provides a basis for later reporting and for understanding how the learner interacted with the task

5.4 Blockly Integration

Blockly is integrated as the visual programming editor for both learners and teachers. In the learner player, the workspace is configured dynamically from the current exercise definition. The available blocks are not static, instead, the toolbox is assembled at runtime from built-in categories such as logic, loops, math and text together with engine-specific blocks for turtle or robot behavior. This directly implements the idea of constrained toolboxes discussed earlier in the thesis.

The block integration is not merely visual. Custom block generators emit JavaScript calls such as **api.move(...)**, **api.turn(...)**, or **api.collect(...)**, which become the contract between Blockly and the sandbox runtime. This was an important design choice, because it gives the generated programs a narrow and exercise-specific execution surface instead of unrestricted access to browser objects. The same block definitions can therefore be reused in the learner player, the CMS preview and the grading-related execution paths.

The robot exercise type illustrates how this contract can be specialized for a different goal. Where the turtle exercises expose parameterized `move N steps` and `turn N degrees` blocks suitable for free drawing, the robot toolbox uses fixed-action blocks such as `step forward`, `one cell`, `turn left`, and `turn right`. These blocks call the same underlying engine API but remove the need for learners to reason about arbitrary numeric arguments while solving grid puzzles. Together with the optional `collect` block, they make the robot tasks feel like a distinct puzzle activity rather than a numeric variant of the turtle exercises.

The same Blockly component is reused in the CMS editor. Teachers can choose blocks, define starter workspace and preview how an exercise appears from the learner perspective. The implementation stores and restores Blockly workspaces as XML, which allows a starter configuration to be reused consistently between authoring and learner execution. For visual exercise types, Blockly categories combine generic logic and control blocks with engine-specific movement blocks.

In practice, the Blockly integration also required several smaller engineering adjustments that are worth documenting. The workspace component adds accessibility attributes after injection so that the generated DOM is exposed as a labelled region, not just as an opaque graphics surface. In addition, a `MutationObserver` is used to keep the flyout scrollbar behavior stable when toolbox visibility changes. These details are easy to omit in a broad design description, but they matter for turning a library integration into a usable web application.

One practical advantage of this design is that the editor remains largely data-driven. Exercise definitions determine both the learner experience and the authoring preview, which reduces duplication between CMS and player code. A typical `io` exercise can expose only arithmetic,

text output and control flow blocks, whereas a turtle exercise can offer movement and pen blocks with a smaller set of generic constructs. In both cases, the learner sees only the blocks that are relevant for the current task.

5.5 Execution Pipeline and Trust Boundaries

The submission pipeline is the strongest technical part of the implementation because it connects editing, execution, grading and persistence through explicit interfaces. From the learner's perspective this appears as a simple `run` or `check` interaction, but the implementation actually performs a sequence of coordinated steps:

1. The Blockly workspace is translated into JavaScript code.
2. The player creates a sandbox request containing the code, exercise type allowed `API` methods, runtime limits and optional input data.
3. A hidden iframe receives this request through a `MessageChannel` and executes the code in an isolated browser context.
4. The sandbox returns a structured execution trace containing command logs, output streams, timing data and, for visual tasks, the final state.
5. The appropriate grader converts this trace into a uniform grading result.
6. For authenticated submissions, the server validates the shape of the submitted grading object, re-executes the code authoritatively and only then stores the attempt in the database.

This pipeline is intentionally layered. The player is responsible for user interaction, the sandbox is responsible for execution, the graders are responsible for correctness evaluation, and the server-side remote function is responsible for trust-sensitive persistence. This separation makes it easier to reason about failures and avoids hiding core logic in UI components.

The sandbox implementation is intentionally generic across exercise types. The hidden iframe is created with restrictive sandbox settings and communicates with the main application only through typed messages. Each execution request includes the allowed API methods, timeout, maximum command count and maximum loop iterations. Loops in the generated code are instrumented before execution, so the runtime can interrupt programs that would otherwise run indefinitely.

This design provides two practical benefits. First, it keeps the learner interface responsive even when a program fails or loops unexpectedly. Secondly, it establishes a clearer trust boundary than direct local execution in UI components. The browser-side sandbox is still a pragmatic prototype solution, not a formally strong isolation boundary, but it already separates learner code from the main interface and blocks many browser APIs that are irrelevant or dangerous in this context.

An important extension of this idea is the authoritative execution path on the server side. When an authenticated learner submits an attempt, the server does not simply trust the client reported score. Instead, it validates the structure of the submitted result and then re-executes the generated code inside an isolated Bun subprocess worker. The worker has no access to the database, the session, or any application module. It receives only the code, the exercise type and the runtime limits over standard input and reports the resulting trace over standard output. Inside the worker, the actual code runs in a `node:vm` context that enforces per-execution time and loop limits and the parent process additionally enforces a hard kill timeout and a maximum output size. The grading result that is stored in the database therefore comes from

this re-execution rather than from the client-reported value. This is one of the most important implementation choices of the project because it separates immediate learner feedback from trusted persistence at both the API and the operating-system level.

5.6 Grading Pipeline

The grading pipeline differs by exercise type, but all graders return the same result structure. Pass/fail status, integer percentage score, number of passed test, total number of tests and per-test feedback. This common format was an important implementation goal because it allows the UI, persistence layer and analytics code to remain largely independent of the specific grading strategy.

For `io` exercises, the executor runs the generated program once per test case, injecting the configured standard input and collecting standard output and error output. The grader then normalizes the produced text according to the exercise configuration. The supported normalization rules include trimming, whitespace collapsing, case handling, line-ending normalization, and decimal separator handling. This is more robust than direct string equality and reduces the number of irrelevant mismatches that beginners would otherwise encounter.

For visual exercises, execution produces a command trace that can be replayed on the corresponding engine. Grading then evaluates the outcome against target, state, command, path, or collect tests. A deliberate implementation decision was to grade the replayed command semantics rather than comparing rendered pixels. This keeps the grader deterministic, easier to test and less dependent on rendering details. In other words, the system asks whether the learner reached the intended logical state, not whether two canvases happen to look identical at the pixel level.

The current implementation also distinguishes carefully between visible and hidden tests. Visible tests can expose expected and actual values to learners, while hidden tests contribute to the score without revealing complete solution information. This is especially important for `io` tasks, where showing all hidden examples would make it easier to figure out a program. The grading and submission code therefore treats hidden tests not merely as a UI concern, but as part of the trusted exercise definition.

5.7 Learner Player and Attempt Capture

The learner-facing course player presents one exercise at a time, maintains progress within the course and records attempts together with additional metadata. The learner sees a top bar to switch between the task description, the Blockly workspace and the run and results panel. This has been changed after the evaluation. At first all of these areas were visible at the same time, side by side and only on mobile could you switch between them. In the evaluation, concerns were raised that the interface looked too busy and that the course navigation and task description competed for attention with the workspace and results.

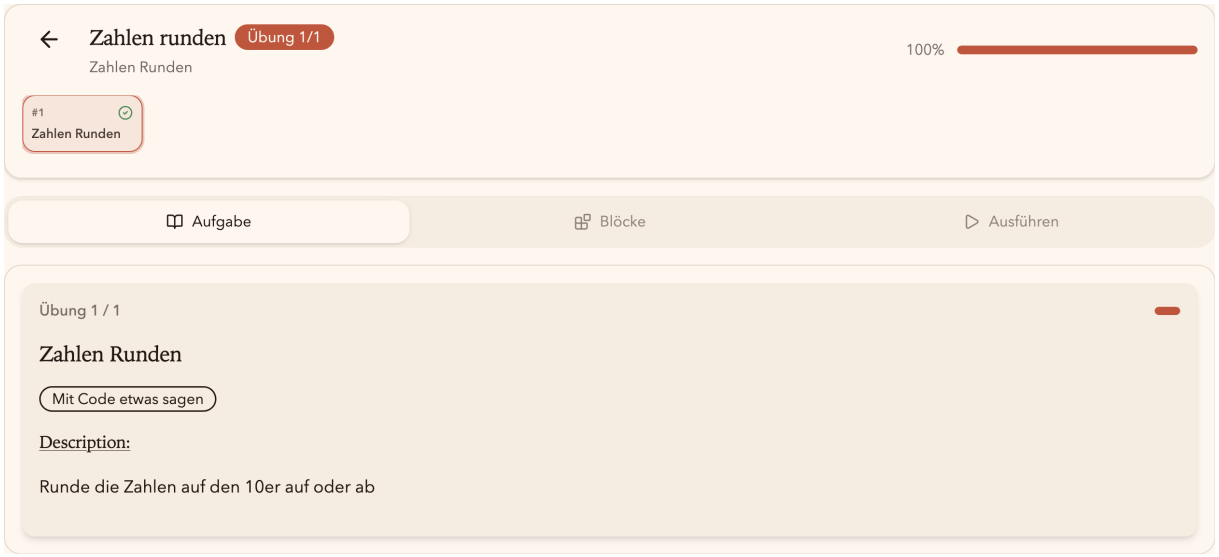

Figure 5.1: The learner player after the redesign. A top tab strip switches between the task description, the Blockly workspace and the run and results panel.

Each submission stores the Blockly workspace XML, the generated code, the grading result, the selected locale and structured metadata such as hint usage and test counts. This is more detailed than a conventional quiz record. The platform does not only preserve whether a learner passed, it preserves what was submitted and under which conditions. This makes later review and analysis more meaningful and provides useful data for the teacher.

The player also restores stored workspace state when learners return to an exercise. This is especially relevant for guest mode, where continuity must be preserved without relying on an account. The course-level player therefore does not hard-code a persistence method. Instead it accepts an injected persistence function so that the same learner interface can be used with local guest storage or authenticated server-side submission.

Hints are also treated as first-class interaction data. The player records which hints were revealed, when they were revealed and whether they were triggered by time or explicit learner action. This is a small but important example of how the implementation captures richer learning process data than a simple final score.

5.8 Teacher-facing Authoring Workflow

The CMS includes a substantial exercise editor for authoring. Teachers can define localized titles and descriptions, choose an exercise type, configure the toolbox, edit hints and manage test cases. In practice, the editor is not a single form but a multi-surface workflow that combines text editing, Blockly starter-block editing, visual canvas configuration and learner-side preview.

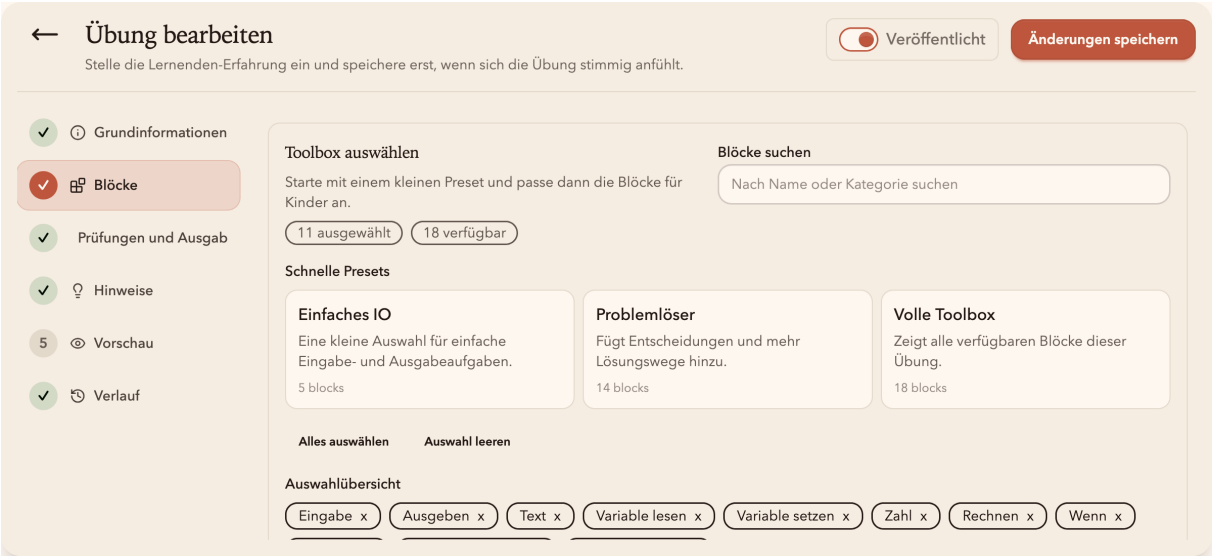

Figure 5.2: The teacher exercise editor in the CMS, showing localized content fields, exercise-type selection, toolbox configuration, hints and test-case management.

For `io` tasks, the editor offers explicit control over visible examples, stdin/stdout test cases, normalization rules and localized success messages. For visual tasks, the editor supports both automatic and manual test creation. Targets placed on the canvas become target tests automatically and drawn path overlays become path tests. Additional state or command tests can then be added manually for more specific requirements. This combination proved useful because it allows quick authoring of straightforward exercises without removing the ability to define stricter checks.

The authoring flow also includes an integrated preview. Before publication, teachers can run the current exercise definition through the same learner-facing logic that is used during normal exercise solving. This is important for thesis scope because it reduces the gap between authoring and delivery. The same execution and grading pipeline can be exercised locally before content is made public.

Before an exercise is published, the implementation performs validation checks on required fields and configuration completeness. Published exercises must have bilingual title and description fields, at least one test case, at least one hidden test, and type-specific dimensions or settings that make sense. Starter XML is also checked for parseability and the contained block ids must belong to the configured toolbox so that an exercise cannot be published with starter blocks the learner is not allowed to use. This validation is intentionally lightweight compared to a formal content-quality workflow, but it already blocks the most obvious authoring mistakes from reaching learners. The same validation function is also re-used in an automated database verification script that loads every seeded exercise after **db:seed** and asserts that each published exercise still passes publish validation.

The CMS also exposes a course-level analytics view and a dedicated multilevel class-analytics surface. The course-level view combines the summary statistics such as average score, number of attempts and unique student count with several process-oriented metrics that are derived from the analytics payload of each attempt. Average solve duration, average number of revealed hints, average workspace block count, average generated code length and the distribution of attempts across the German and English locales. The same metrics are also broken down per exercise so that teachers can spot tasks that are unusually difficult, attract many hint reveals,

or are mostly attempted in only one of the two languages.

The dedicated analytics page scales the same idea up to the classrooms level. The first level lists every class with aggregated attempt count, enroled student count, pass rate, average score, average solve duration, hint usage and time of last activity. Clicking a class opens the second level, which keeps the same metrics in the page header and then decomposes them by individual student. Clicking a student opens the third level, which follows the same student across every course assigned to the class and reports per-course attempts, exercises attempted, exercises passed, pass rate, average score and locale split. These indicators do not replace classroom observation, but they make the captured attempt data actionable for teachers along the same axes their classroom is already organised on. Note that the analytics page was not part of the original project. It was added after the first evaluation, where participants stated that it was a must-have for teachers.

5.9 Authentication, Authorization and Audit

Authentication is implemented through Better Auth with both email-and-password accounts and an optional Single Sign-On path. Sessions are stored in the application database and propagated to SvelteKit through the dedicated cookies plugin. User accounts carry a role field (student, teacher or admin) together with explicit active flag. A central server-side guard treats only login, demo, privacy, password-reset and authentication API routes as public, every other route is redirected to the login page when no valid session is present and inactive users are rejected even if they authenticate successfully. Teachers and admin-only remote functions reuse the same role check so that authorization is enforced at the data layer rather than relying on the user interface alone.

Single Sign-On is provided by the `@better-auth/sso` plugin and supports both OIDC and SAML identity providers. Provider records are kept in a dedicated `ssoProvider` table and are managed at runtime through an admin settings page rather than through environment files, which means a school administrator can register, edit or remove providers without a code change. On every successful SSO login the system inspects the IdP claims (groups, roles) and re-evaluates the mapping to a BlockQuiz role using a stored claim-to-role map for the admin, teacher groups, falling back to student when no group matches. Because the mapping runs on every login rather than only at account creation, changes to group membership in the IdP propagate without manual cleanup in BlockQuiz. The same admin settings page exposes a password login mode toggle (always versus fallback) that allows an SSO-primary deployment to hide the email-and-password form behind a small link while keeping it available for administrator recovery and the login page itself surfaces the configured providers as `Continue with SSO` button.

Password reset is implemented as an end-to-end flow. A learner who has forgotten their password enters their email address through the public login screen, which calls the **requestPasswordReset** remote function. That function delegates to Better Auth, which generates a single-use, time-limited reset token. The **sendResetPassword** hook then sends the resulting reset URL through an email. Three email drivers are available and selectable through the admin settings page. A file-based outbox driver that writes the message into a configurable directory, an SMTP driver for conventional mail relays and a Microsoft Graph driver that posts the message via the Graph API for Microsoft 365 tenants. The driver choice and its credentials can be overridden by environment variable for container deployments and an unexpected failure of the SMTP or Graph driver falls back to the file outbox so that the reset link is never silently lost. The file outbox is also the default in development and for school deployments without an outgoing mail relay. The matching `/reset-password` route accepts the token from the URL, requires confirmation of the new password and submits both values to a second remote function that

calls Better Auth's **resetPassword** endpoint. A successful reset invalidates the token, saves the password, and writes a corresponding entry to the audit log. The user is then redirected to the login page.

Sensitive content and account events are captured through a small server-side audit-log helper. Exercise creation, update, cloning, archiving, restore and import write audit records, as do course creation, role or status change and administrator-initiated password reset. Together these events form a baseline trail that an administrator can review through a dedicated logs page, which also offers a CSV export of the audit history, per user filters, search and full json content display for each record.

5.10 Class Model and Identity-Provider Synchronisation

Educational deployments rarely enrol students into courses one at a time. The natural unit is a class. The implementation therefore models classes as first-class entities, sitting alongside the existing user, course and attempt tables.

Different types of class memberships. The implementation distinguishes two kinds of class membership at the schema level. Each row in `class_users` carries a source column set to either 'sso' or 'manual' and the same user may have both for the same class. Manual rows are created by administrators through the CMS class editor and are off-limits to the SSO sync algorithm. They survive identity-provider outages and do not disappear when a user has been temporarily removed from an IdP group. SSO rows, in contrast, are owned by the `reconcileClassMembership` function and are added and removed to match the IdP-claimed group set on every login. The split lets the same class hold a stable manual roster alongside an auto-managed IdP roster without either side overwriting the other.

Reconciliation on every login. The reconciliation step runs inside the `provisionUser` hook of the Better Auth SSO plugin, immediately after the role mapping. The hook resolves the Better Auth provider id back to the local `ssoProvider` row, extracts the group claims from the IdP user-info payload, and asks `reconcileClassMembership` to compute the delta against the existing SSO-sourced memberships for that user and provider. The delta is applied transactionally, an audit-log entry of the form `class.sync` records the added and removed class ids together with the count of unmatched group claims and the reconciliation runs in a try/catch wrapper so that a class-sync failure does *not* block the login. Each observed claim value is additionally written into the `idp_group_seen` discovery table in a fire-and-forget upsert. This table is what powers the CMS IdP Group Discovery.

Admin-facing class management On the CMS side, the implemented page includes a class editor for manual classes. The course editor exposes a class-assignment widget alongside the existing per-user enrolment, so a teacher can hand a course to either named individuals or a whole class without leaving the editor.

5.11 Representative Scenario: Authoring, Learning and Feedback

To make the preceding sections concrete, this section walks through one exercise end-to-end against the actually running system. The example is the seeded I/O task "Sum to N", which is part of the default course bundle and exercises.

Authoring

A teacher (call her Jane) opens the CMS exercise editor, picks the `io` type and fills in the bilingual content fields. She then selects the toolbox: `variables_set`, `variables_get`, `math_number`, `math_arithmetic`, `controls_repeat_ext`, `text_print` and `text_prompt_ext`. She defines two visible test cases ($N = 3 \to 6, N = 5 \to 15$) and three hidden ones ($N = 10 \to 55, N = 1 \to 1, N = 100 \to 5050$). She enables the trim and case-insensitive normalization rules so that an extra trailing newline does not fail the grader. She adds two hints, one click-triggered (`"Use a loop to add the numbers one by one"`) and one delayed by 60 seconds (`"Try a controls_repeat_ext block with a counter variable"`).

When Jane clicks Publish, the **validateExercise** function runs. Bilingual content is present, the starter XML is empty (so its block-id check is trivially satisfied) and the test set contains visible cases, hidden cases are not required. The exercise becomes learner-visible.

She then has to create a course add the exercise to the course and assign the course to a specific user or class.

Learning

A learner (call him John) enters the assigned course, lands on the exercise and sees the top navigation layout. He drags blocks into the workspace and clicks Run. The Blockly XML is converted to JavaScript, sent into the hidden iframe sandbox with the configured runtime limits and the standard output stream is reflected back into the result pane. His first attempt fails on the second visible test, so the partial-progress bar read "1/2 visible tests passed" rather than a flat red error. He plays around with the blocks, tries again and now passes all tests.

Submission, regrading and badges

The submission goes through **submitAttempt**. The server validates the shape of the client-reported result, then re-executes Johns generated code inside the isolated Bun subprocess worker. The worker runs all five test cases and reports a final score back to the request handler, which is what is stored in the database. The handler then runs the badge rules engine against John's history, awards the `first_solve` badge, and returns the new badge in the response. The course player surfaces it as a small toast and the `course_complete` badge will follow once he finishes the remaining exercises.

Teacher feedback

The next morning Jane opens the course-analytics view. For "Sum to N", she reads: 18 attempts across 12 unique learners, average score 78%, average solve duration 4 minutes 12 seconds, average 1.4 hint reveals per attempt, 40% German and 60% English. The hint-reveal rate flags this exercise as the hardest in the course. She decides to add a stronger first hint and to keep the exercise as is otherwise. None of these signals come from a separate analytics service. They are derived directly from the per-attempt records the player captured during the learner flow.

What this scenario exercises

The walkthrough touches every layer described in Chapter 4 the typed exercise model (publish validation), the Blockly toolbox constraint (per-exercise allowlist), the layered execution model (browser sandbox plus subprocess regrading), the structured attempt record (workspace, generated code, hint events, locale, duration), the achievement system (badge awarded as side effect

of the same submission path) and the teacher analytics view that closes the loop. It is the intended end-to-end of the system, not a single isolated feature.

5.12 Closing the Loop

Overall, the implementation confirms that the proposed architecture is technically viable. The implementation is particularly strong where editing, execution, grading, persistence and authentication intersect. Together with the audit log, the publish-validation pipeline, the analytics view and the authoritative server-side regrading, the code base now supports a complete flow from teacher authoring through learner solving to administrator review. The implementation thereby demonstrates that a web-based block-programming platform with automatic grading, guest mode, teacher authoring, account recovery and self-hosted deployment can be built as a coherent system rather than as a collection of isolated features.

6 Evaluation

This chapter reports the evaluation of BlockQuiz along two independent axes. A technical evaluation measures whether the implementation satisfies the requirements through static checking, automated tests and structured scenarios. A user evaluation with two student teachers at the University of Innsbruck reports the first qualitative signal on whether the platform is understandable and pedagogically appropriate from the perspective of its intended authors.

6.1 Evaluation Goal

The evaluation of BlockQuiz combines two perspectives. The first is a technical perspective. Whether the proposed architecture can be implemented in a coherent, testable and operationally plausible way and whether the implementation satisfies the functional and non-functional requirements. The second is a user perspective. Whether teachers, the primary group expected to author exercises and embed the platform in lessons, find the system understandable, age-appropriate, usable and what concrete obstacles they identify when working with the implementation.

Both perspectives are reported in this chapter. The technical evaluation relies on automated tests, static checking, a database verification script. The user evaluation relies on a small qualitative study with two student teachers at the University of Innsbruck, who worked through a structured set of tasks, filled in a twenty-item scale questionnaire and discussed their impressions afterwards. The two perspectives are complementary. The technical evaluation answers whether the system works, the user evaluation answers whether the system works for the people it was built for.

6.2 Evaluation Questions and Method

The evaluation is structured around the following questions:

1. Can the implementation execute and grade the supported exercise types correctly and deterministically?
2. Does the implementation separate immediate learner feedback from trusted persistence in a technically meaningful way?
3. To what extent does the implemented system satisfy the functional and non-functional requirements?
4. Do student teachers find the platform understandable, age-appropriate and pedagogically suitable and which concrete obstacles do they identify in the prototype?
5. Which limitations remain visible after static checking, automated test, analytical inspection and qualitative user feedback?

6.3 Technical Evaluation

The project passes `bun run check` with zero reported errors and warnings. The automated test suite executes **296** tests across 27 test files via `bun -bun run test` and a dedicated database verification script asserts that the seed data is idempotent and that every published seeded exercise still satisfies the publish-validation rules. These results do not replace pedagogical evaluation with learners, but they establish a repeatable technical baseline that is rerunnable on any commit.

Table 6.1 summarises how the core requirements from Chapter 3 are met by the implemented system.

Table 6.1: Coverage of the core requirements

Requirement	**How it is satisfied**	**Status**
FR-2 Exercise types	io, turtle and robot types, each with its own configuration and grading	Met
FR-3 Authoring and courses	teacher CMS for exercises, ordered courses and class assignment	Met
FR-4 Execution and feedback	client-side iframe sandbox with controlled API and runtime limits	Met
FR-5 Auto-grading	test-based and state-based graders with a shared result structure	Met
FR-6 User modes and progress	guest and authenticated paths, optional OIDC and SAML SSO, IdP-synced classes	Met
NFR-1 Security	iframe sandbox, isolated subprocess regrading, role checks and response-header policy	Met
NFR-2 Privacy	guest mode, data minimisation and self-hosted single-container deployment	Met

6.3.1 Automated Test Suite

The tests are concentrated on the technical core of the system rather than on end-to-end browser automation. The largest tests are about the exercise engines and graders. These are the most complex and critical parts of the system and are tested on correctness of visual state changes and grading results. The sandbox tests cover particularly important invariants such as the message protocol, loop instrumentation, blocked globals and error mapping. The authoritative-execution tests focus on a security-relevant property of the system. Stored results for authenticated submissions are derived from server-side re-execution inside an isolated subprocess rather than from client trust, and the subprocess itself rejects malformed or tampered worker output.

A second concentration of tests sits around the class system and SSO, which are operationally the riskiest parts of the codebase because they have to remain correct across both manual administrator action and Idp claims deltas.

The remaining tests cover cross-cutting invariants of the platform, including the three level class-analytics aggregator, submission-payload validation, the public-route boundary, exercise canonicalization, the plain-language code readout parse, and locale parity between the German and English translation files.

The test suite does not cover everything. There is no comprehensive end-to-end browser test suite for the full learner or CMS workflows, the unit suite is complemented by a focused Playwright pass that exercises the public flow, the authenticated learner and CMS flows, the responsive tab strip, and the SSO login link rendering, but full CMS-form regression coverage is intentionally out of scope for a bachelor-thesis prototype.

6.3.2 Database Verification

The dedicated `bun -bun run db:verify` script provisions a fresh SQLite database in a temporary directory, runs db:push twice to prove idempotency, runs `db:seed` twice to prove the seed

itself is idempotent, asserts the presence of every expected table and column, asserts the seeded counts (two courses, ten exercises), exercises the attempt-actor CHECK constraint with both a valid guest insertion and an invalid user-without-user_id insertion, and finally loads every published seeded exercise and runs it through validateExercise. This last step caught a real defect during development. The new robot blocks were initially missing from the toolbox allow-list, which would have blocked publication of the seeded robot exercises. The script therefore acts as a regression gate against the exact class of authoring error that the publish-validation pass is designed to catch.

6.4 User Evaluation

In addition to the technical evaluation above, the implementation was put in front of two student teachers at the University of Innsbruck. The purpose was to obtain qualitative feedback, whether the system is understandable and pedagogically suitable from the perspective of the user group that is expected to author and embed exercises in real classroom situations and to surface concrete obstacles that the technical evaluation cannot see.

With two participants no statistical inference is attempted, the value of the data lies in the themes that emerge from the open-ended responses and the post-task discussion.

6.4.1 Method

Both participants worked through four structured tasks against a running instance of BlockQuiz seeded with the default course bundle. The tasks were designed to walk through the system in the order a teacher would naturally encounter it:

1. First orientation. Open the application, log in and explore the main surface without a specific goal.
2. Example use. Open an existing exercise as learner and solve it.
3. Feedback and help. Trigger the platform's feedback and hint mechanisms and assess whether the explanations are useful.
4. Pedagogical Use. Author a small exercise and reflect on the integration into a classroom context.

Each task carried an open-text response field. After the tasks, the participants filled in a twenty-item scale questionnaire on a five-point scale where 1 corresponds to "strongly disagree" and 5 corresponds to "strongly agree". A short debriefing discussion was held afterwards, during which the experimenter took written notes. The items from that discussion are interleaved with the questionnaire results below.

6.4.2 Participants

Both participants identify as student teachers and have direct pedagogical experience but no substantial classroom teaching record yet. Their background differ enough to provide useful contrast on the same prototype

- P1: self-describes as a "Studentin Lehramt", with pedagogical training through the teacher-education programme and occasional experience with educational software and digital learning environments.

Table 6.2: Likert ratings on the twenty questionnaire items (1 = strongly disagree, 5 = strongly agree)

Item (translated)	**P1**	**P2**
The software is easy to understand overall	4	5
Operating the software feels intuitive	3	4
I could orient myself in the application quickly	3	3
The most important functions are easy to find	4	5
The interface feels clear and not overloaded	4	3
The terms and explanations used are understandable	3	5
The software is fundamentally suitable for children/students	5	5
The tasks and content feel age-appropriate	5	5
The software can meaningfully support learning processes	5	5
The feedback from the software is helpful for learners	5	5
The software supports independent exploration and learning	5	5
The software can foster motivation or interest in the topic	5	4
I consider the software useful in an educational context	5	5
I could imagine using the software in a pedagogical situation	4	5
The effort of using the software seems appropriate relative to its benefit	5	5
The software fundamentally fits pedagogical work or learning contexts	5	4
I would recommend the software after improvements or adjustments	5	5
The software overall makes a polished impression	4	5
The software fulfils its purpose in an understandable way	5	5
Overall, I evaluate the software positively	5	5
Mean (twenty items)	**4.45**	**4.65**

- P2: self-describes as "Kindergartenpädagogin und angehende Lehrerin", with pedagogical experience from kindergarten, school placements and childcare but no or hardly any experience with educational software.

The contrast matters. P1 brings some prior familiarity with educational software, P2 brings concrete experience with younger learners and almost none with software platforms. Their reactions to the system therefore probe different parts of the design space.

6.4.3 Quantitative Results

Two patterns stand out across both participants. First, the pedagogical-fit cluster, suitability for children, age-appropriateness, support for learning processes, feedback usefulness and the global 'would recommend after improvements' item, sits consistently in the 4-5 range for both, which suggests that the platform's stated target audience and educational orientation are recognised by readers who are not its authors. Second, the onboarding-related items. 'I could orient myself in the application quickly' and 'operating feels intuitive' are the lowest in both columns. In the discussion afterwards, both expressed why they found the onboarding difficult which will be reported in a later section.

6.4.4 Qualitative Findings

The open-text responses and the debrief discussion converge on a small number of recurring themes. The themes are reported in the order they typically surfaced in the session.

Visual design and first impression

The visual design is the most consistent positive theme. P2 named 'das Layout, die Vielfalt' as the best aspect of the software, P1 wrote that 'viele kleine Details wirken gut durchdacht' and

the debrief notes record an overall impression that 'the UI is very nice and inviting and clean in general'. P2 specifically praised the colour choice and the icon set in the first orientation task: 'nettes Layout, gute Farbauswahl, nette Icons.'

Onboarding cognitive load

The strongest negative theme is the steepness of the first encounter. P2 described the volume of information at the start as 'etwas überfordernd am Anfang' and predicted that children would feel overwhelmed and distracted. P1 task-1 notes echoed this 'etwas überfordert von vielen Input.' In the debrief, both participants summarised the same point as 'too much text, too small, accessibility is missing, too much information'. The remedy both suggested converges on a single intervention, a short instructional video or guided tour. P1 wrote explicitly that 'ein Erklärvideo, das es vormacht' would have made the authoring task much easier and the debrief notes record 'better block explanation through videos and instructions'. The existing in-app hints address learner-side discovery but do not yet extend to the authoring surface, which is where the overload is worst.

Localisation gap on block labels

Both participants flagged that the Blockly block labels themselves are still rendered in English while the rest of the German UI is localised. P1 named this directly as the most important missing element the german translation of blockly blocks in the CMS. **This was fixed after the evaluation session**

Discoverability and navigation

Several individual navigation affordances were not where the participants expected them. The back-arrow was the first concrete issue mentioned in P1's task-1 notes (Zurück Pfeil nicht sofort ersichtlich) and recurs in the debrief. The starter-block section of the exercise editor did not surface for P1 (Starterblöcke nicht sichtbar) and the add check button on the I/O exercise editor was reported as non-functional (Prüfung hinzufügen Button geht nicht). The debrief notes corroborate these as concrete bugs rather than UX ambiguities. 'Live-update of checks in I/O did not work' **was fixed after the evaluation**. The starter blocks not showing in the editor also a bug which was fixed after the evaluation. Two further workflow suggestions arose from the same discussion. A 'save and return to the listing' redirect after saving an exercise or a course, and a way to attach an exercise to a course from the exercise-detail view rather than only from the course editor. Both are small but high-impact changes to the authoring path.

Bugs that block classroom use today

The debrief discussion produced the strongest piece of unsolicited qualitative feedback in the whole study. 'Due to the bugs, especially the starter workspace, currently not usable for school. But if the bugs are fixed and the problems mentioned in the evaluation are addressed, both would be happy to use it in school.' This both confirms the pedagogical fit and identifies the gating problem. The system is regarded as fit for purpose in principle but not in its current build state at the evaluation. The specific bugs called out in the debrief are the starter-workspace visibility issue, the non-updating I/O check editor, the unvalidated preview blocks, and the confusion around the I/O input block.

Missing features and forward-looking suggestions

Several suggestions in the responses point beyond bug-fixing into genuine feature requests. P1 named per-student analytics as the most important addition for pedagogical use. This has been added after the evaluation meeting. P2 suggested that the platform should rather be a standalone app rather than a website to limit distractions of the students which will, inevitably search something online than doing the exercises.

Accessibility and differentiation

Two specific accessibility concerns surfaced. P1, who flagged her own consideration for dyslexic learners, found the default font size too small and felt that the system would be hard to adapt to the needs of learners with reading or learning disabilities. She said she would deploy the software only in a more homogenous class because she finds it hard to adapt the platform for learners with different needs. P2 pointed out that the layout on the desktop and tablet should be more like the mobile layout, which is more compact and presents the exercise in stages the learner can choose in a navigation bar (Description, Blocks, Run). This is a good point and was adapted to all layouts after the evaluation.

Pedagogical fit and intended target audience

The pedagogical fit of the platform is consistent and positive. Both participants identified primary school and lower-secondary as the most appropriate target groups, with P1 adding secondary level 2 and both adding out-of-school youth work. P1 specifically saw the platform's main home in the Austrian school subject 'Digitale Grundbildung' with secondary applicability in mathematics, while P2 saw a strong fit in MINT-focused schools at the secondary level. P2's closing answer to the question: "Would you consider using the software in a pedagogical context? Why, why not?" was: "Yes, because it conveys an easy understanding to learners." P1's closing answer is the cautious version. She would deploy the software after another iteration and only in a relatively homogeneous class given her concerns about differentiation.

6.4.5 Issues Identified and Triage

Table 6.3 consolidates every concrete issue raised in the questioning into a triage list. The intent is to make the link between qualitative feedback and prioritisable engineering work.

Three issues in the triage table are bugs that the evaluation was the first encounter to surface. The non-updating I/O check editor, the unvalidated preview blocks and the broken I/O input block. These have been fixed since then. Several other entries are small UX changes that are straightforward to address within the existing remote-function layer. The two substantial entries, the german Blockly localisation and the responsive CMS layout, are larger pieces of work and have also already been addressed.

6.5 Interpretation

Taken together, the technical and the user evaluation support a limited but coherent conclusion. The proposed architecture is technically feasible and stable enough to justify the main design decisions of the thesis and the design is recognised as pedagogically appropriate by readers who are training to use such systems in real classrooms.

The technical evidence shows the runtime and grading behaviour are backed by 296 automated tests. The submission path includes an explicit trust boundary between client interaction

Table 6.3: Issues raised in the user evaluation, classified by type and engineering effort

Issue	Type	Effort	Description	Status
blockly block labels still in english under de locale	localisation / fr-7 gap	medium	requires a german message catalogue for the blockly built-in blocks alongside the existing application i18n parity test	closed
starter blocks not surfaced in the exercise editor	bug	medium	starter-workspace section should always be visible when an exercise type is selected	closed
"add check" button does not update the i/o checks list live	bug	low	a reactive query subscription is missing on the editor	closed
i/o input block confusing and non-functional	bug	high	the input-block authoring path needs both a wording revision and a behaviour fix	closed
preview blocks are not validated against the toolbox	bug	medium	preview path should reuse the same publish-validation pass that the publish action runs	closed
save action on exercise/course does not return to listing	ux / workflow	low	post-save redirect to the parent listing	open
cannot attach an exercise to a course from the exercise detail view	ux / workflow	medium	requires a small course-picker affordance on the exercise detail surface	open
authoring surface overall feels overwhelming for newcomers	ux / training	high	candidate for a short instructional video and/or a guided tour overlay; per-section progressive disclosure	open
font size too small for dyslexic learners	accessibility	medium	requires using the font settings of the browser and for accessibility itself a redesign of the whole website	open
mobile / tablet layout reuses the desktop layout	responsive design	medium	the player tab strip is responsive but the cms surfaces are not yet tablet-optimised	closed
per-student analytics page	discoverability	low	has been added after the evaluation since it was seen by the participants as an inherent feature	closed
teacher cannot tell which blocks are needed to solve an exercise	training	high	teacher training and tutorial videos would be necessary for a real deployment	open
distraction risk on shared tablets (browser tab vs. dedicated app)	deployment	high	requires platform to be PWA compatible or made as an app so that learners are discouraged to open another browser tab and browse the internet	open

and server persistence. The authoring workflow is broad enough to support real content creation rather than only hardcoded demo exercises. The class system and its SSO reconciliation are exercised by dedicated test suites that pin down both the adding and removing of users and classes. The deployment model is simple enough to remain credible for small self-hosted school installations.

The user evidence is more limited but important. Both participants rated the platform positively overall and identified its target audience and pedagogical fit consistent with the thesis's stated scope. At the same time, both reported a steep onboarding into the CMS and a small set of bugs that would block classroom use in the current build. The most actionable consequence of the study is not a rethink of the architecture but a triage list of concrete issues that need to be addressed. The strongest unsolicited message from the debrief is that both participants would consider deploying the system in their own future classrooms once the listed issues are addressed and tutorials exist for teachers to learn how to use the system.

At the same time, the evaluation should not be overstated. Technical stability is not equivalent to educational effectiveness. A passing test suite does not prove that children understand the interface, that teachers will find the CMS easy to adopt at scale or that the platform improves learning outcomes. **The user study, with n=2 student teachers and no actual child participants, is too small for any quantitative claim about usability and does not speak to learning outcomes at all.** What both perspectives together do show is that the implementation has moved beyond a conceptual mock-up, embodies the core thesis argument in executable and testable software and produces a positive impression on the users which are going to create content in the system.

6.6 Evaluation Limitations

The evaluation comes with some clear limitations.

Small qualitative sample

The user study reports findings from two participants, both of them student teachers at the same university. This is a very small sample and does not support quantitative inference. Their opinion and evaluation is important to get an idea of the pedagogical fit of the system, but it is not representative of the full range of potential users and does not speak to actual learning outcomes with real child learners.

No child or in-classroom data

Neither participant is a child in the target age range and no exercises were run with actual learners in a classroom setting. The findings therefore speak to the teacher-facing UI and to pedagogical judgements about classroom suitability, but they do not speak to whether the platform improves learning outcomes for children aged 8-12. A follow-up study with actual learners in a classroom is the most valuable next step to address this gap.

The present evaluation demonstrates technical viability, repeatable subsystem correctness and a first piece of qualitative evidence from student teachers that the design is recognised as pedagogically appropriate. A fuller study with children in a classroom remains future work.

7 Conclusion

This chapter closes the thesis. It summarises the contribution, draws out the lessons that the implementation and the user study taught, states the remaining limitations and points to the next steps that the implementation now enables.

7.1 Summary

This thesis set out to design a web-based block-programming platform for structured introductory exercises in school context. The core motivation was to combine several requirements that are often only partially addressed together. Age-appropriate visual programming, immediate feedback, teacher-oriented authoring, privacy-conscious deployment and a workflow that remains practical for classroom use.

The resulting system, BlockQuiz, was developed as a design and reference implementation rather than as a productized commercial offering. Within that scope, the thesis makes three main contributions. First, it defines a focused exercise model for guided programming tasks instead of open-ended project work. Second, it develops and implements an architecture that combines Blockly-based editing, sandboxed code execution, structured auto-grading and persistent progress capture. Third, it demonstrates that this architecture can be realized in a technically coherent system with support for bilingual content, guest-mode use, authenticated progress tracking with self-service password reset, optional OIDC and SAML Single Sign-On with identity provider synchronized classes for institutional deployment and teacher facing exercise and class authoring.

7.2 Lessons Learned

The implementation work surfaced concrete lessons.

7.2.1 What worked

The `typed discriminated-union exercise model` was the single most load-bearing decision. Forcing every exercise into one of three structured branches (io|turtle|robot), each carrying its own grader configuration, kept the rest of the system honest. The CMS, the player, the sandbox and the analytics view all branch on the same union and adding a fourth exercise type would now be a localized extension rather than a system-wide rewrite. The `message-based sandbox protocol` was the second decision worth highlighting. Once execution lived behind a typed channel it became natural to wrap it again in a separate authoritative path on the server, which is what later turned the subprocess regrading into a simple addition rather than a refactor. The `publish-validation gate` also paid off in practice, because it caught real authoring mistakes during the implementation work itself. More than once a seeded exercise was rejected because

its starter XML referenced blocks that were not in the configured toolbox, exactly the failure mode the gate was meant to prevent.

7.2.2 Compromises

Several decisions were deliberate compromises rather than ideal choices. The `browser-side sandbox` is the clearest example. It gives immediate feedback and zero per-submission server cost, which fits a school deployment where neither operations staff nor budget for a code-execution service can be assumed, but it would not be appropriate for an open-internet code playground that has to defend against motivated adversaries. The `file-based outbox` remains the default email driver for password resets because it requires no infrastructure and produces a durable record an administrator can deliver out of band, which fits the school-context reality where teachers already hand out credentials in person. Beyond a single school network, deployment can opt into the bundled SMTP or Microsoft Graph drivers through the admin settings page. The file outbox then continues to act as a fallback when an external transport fails so that a reset link is never silently lost. Finally the `complicated CMS` is a compromise between powerful exercise and course creation and usability. The inherent idea of blockly being able to program near any program makes it very difficult to abstract in a way for non technical users to understand easily. Without training how should a teacher know when they want to create a specific exercise like multiply by two which blocks they need for the system to work. But this can be solved with training videos.

7.2.3 What I would do differently

Given more time, I would like to query more teachers and students to understand better how they would use the system and what features they would like to see. Also how performant the system is when hundreds of students are using it at the same time. I would spend more time cooperating with teachers on the design of the CMS and the exercises and creating good base exercises for the system which are included in every deployment. I would rework the internationalization system to be more flexible and support more languages and maybe add the feature to auto translate everything using AI.

7.2.4 Limitations

Beyond the lessons above, three limitations of the current state of the work should be stated explicitly. First, the evaluation is not strong enough. The present results show that the implementation is testable and internally consistent, but they do not yet prove measurable learning outcomes or classroom adoption. The implemented feature set covers the full thesis scope, but it has not yet been validated with real learners or teachers in a classroom setting.

Second, the execution model is layered but not perfect. To protect against malicious actors a container-grade system would be needed. The security model is appropriate for the intended educational deployment, but it remains an explicit compromise rather than a formal isolation guarantee.

Thirdly, adding new exercise types is possible but not trivial. The system is designed to be extensible, but the current implementation only includes three exercise types and adding a new one would require changes across the CMS, player, grader and analytics.

7.2.5 Future Work

The most valuable next step is more systematic evaluation with actual learners and teachers. The implementation itself is, after the bug fixes from the evaluation, now broad enough to

support such a study without further core development. Even a small classroom pilot or a structured usability study would strengthen the thesis considerably by connecting the technical design to observed educational use.

Further work could expand the range of supported exercise types beyond the current three, or improve the existing ones. For example in the turtle exercise, currently there is no way to check if the path the teacher sets is even solvable, which can lead to frustration for the teacher having to learn which paths are possible and which are not. Adding a path solver to the grader would allow teachers to set any path and be sure that it is solvable, which would make the system more flexible and easier to use. Improving the UI in cooperation with teachers and students would also be a valuable next step, especially for the CMS which is currently the most complex part of the system.

Overall, the thesis shows that there is a viable technical path toward such a platform. In its current form, BlockQuiz already demonstrates that tightly scoped exercises, automatic feedback, end-to-end account management and privacy-oriented deployment can be combined into a coherent web-based system suitable for further educational evaluation and refinement.

Bibliography

[1] Kirsti M. Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2):83–102, 2005. doi: 10.1080/08993400500150747.

[2] Thomas Ball, Abhijith Chatra, Peli de Halleux, Steve Hodges, Michal Moskal, and Jacqueline Russell. Microsoft makecode: Embedded programming for education, in blocks and typescript. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*, pages 7–12, 2019. doi: 10.1145/3358711.3361630.

[3] Paul Black and Dylan Wiliam. Assessment and classroom learning. *Assessment in Education: Principles, Policy & Practice*, 5(1):7–74, 1998. doi: 10.1080/0969595980050102.

[4] Karen Brennan and Mitchel Resnick. New frameworks for studying and assessing the development of computational thinking. In *Proceedings of the 2012 Annual Meeting of the American Educational Research Association*, pages 1–25, Vancouver, Canada, 2012.

[5] Bundesministerium für Bildung, Wissenschaft und Forschung. Verordnung des bundesministers für bildung, wissenschaft und forschung, mit der die verordnung über die lehrpläne der mittelschulen sowie die verordnung über die lehrpläne der allgemeinbildenden höheren schulen geändert werden. `https://www.ris.bka.gv.at/eli/bgbl/II/2022/267/20220706`, 2022. BGBl. II Nr. 267/2022.

[6] Code.org. Code.org, 2026. URL `https://code.org/en-US`. Last Accessed: 28.05.2026.

[7] Christopher Douce, David Livingstone, and James Orwell. Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing*, 5(3):4, 2005. doi: 10.1145/1163405.1163409.

[8] Drizzle Team. Drizzle orm documentation. `https://orm.drizzle.team/`, 2024. Last Accessed: 28.05.2026.

[9] European Union. Regulation (eu) 2016/679 of the european parliament and of the council of 27 april 2016 (General Data Protection Regulation). `https://eur-lex.europa.eu/eli/reg/2016/679/oj`, 2016. OJ L 119, 4.5.2016, pp. 1–88.

[10] George E. Forsythe and Niklaus Wirth. Automatic grading programs. *Communications of the ACM*, 8(5):275–278, 1965. doi: 10.1145/364914.364937.

[11] Google. Blockly: A library for building visual programming editors. `https://developers.google.com/blockly`, 2024. Last Accessed: 28.05.2026.

[12] Shuchi Grover and Roy Pea. Computational thinking in k–12: A review of the state of the field. *Educational Researcher*, 42(1):38–43, 2013. doi: 10.3102/0013189X12463051.

[13] John Hattie and Helen Timperley. The power of feedback. *Review of Educational Research*, 77(1):81–112, 2007. doi: 10.3102/003465430298487.

[14] Jack Hollingsworth. Automatic graders for programming classes. *Communications of the ACM*, 3(10):528–529, 1960. doi: 10.1145/367415.367422.

[15] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, pages 86–93. ACM, 2010. doi: 10.1145/1930464.1930480.

[16] Filiz Kalelioğlu and Yasemin Gülbahar. The effects of teaching programming via scratch on problem solving skills: A discussion from learners' perspective. *Informatics in Education*, 13(1):33–50, 2014. doi: 10.15388/infedu.2014.03.

[17] LEGO. Lego mindstorms, 2026. URL `https://www.lego.com/en-us/themes/mindstorms/about`. Last Accessed: 28.05.2026.

[18] Lauri Malmi, Ville Karavirta, Ari Korhonen, Jussi Nikander, Otto Seppälä, and Panu Silvasti. Visual algorithm simulation exercise system with automatic assessment: Trakla2. *Informatics in Education*, 3(2):267–288, 2004. doi: 10.15388/infedu.2004.19.

[19] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The scratch programming language and environment. *ACM Transactions on Computing Education*, 10(4):1–15, 2010. doi: 10.1145/1868358.1868363.

[20] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. Learning computer science concepts with scratch. *Computer Science Education*, 23(3):239–264, 2013. doi: 10.1080/08993408.2013.832022.

[21] Marcus Messer, Neil C. C. Brown, Michael Kölling, and Miaojing Shi. Automated grading and feedback tools for programming education: A systematic review. *ACM Transactions on Computing Education*, 24(1), February 2024. doi: 10.1145/3636515. URL `https://doi.org/10.1145/3636515`.

[22] Microsoft MakeCode. Microsoft makecode, 2026. URL `https://makecode.microbit.org/`. Last Accessed: 28.05.2026.

[23] Peter Naur. Automatic grading of students' algol programming. *BIT*, 4(3):177–188, 1964. doi: 10.1007/BF01956028.

[24] Open Roberta. Open roberta lab, 2026. URL `https://lab.open-roberta.org/`. Last Accessed: 28.05.2026.

[25] Seymour Papert. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, New York, 1980.

[26] Jean Piaget and Bärbel Inhelder. *The Psychology of the Child*. Basic Books, New York, 1969.

[27] Janet C. Read and Mathilde M. Bekker. The nature of child computer interaction. In *Proceedings of the 22nd British HCI Group Annual Conference on People and Computers*, pages 163–170. British Computer Society, 2008.

[28] Mitchel Resnick, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: Programming for all. *Communications of the ACM*, 52(11):60–67, 2009. doi: 10.1145/1592761.1592779.

[29] José-Manuel Sáez-López, Marcos Román-González, and Esteban Vázquez-Cano. Visual programming languages integrated across the curriculum in elementary school: A two year case study using scratch in five schools. *Computers & Education*, 97:129–141, 2016. doi: 10.1016/j.compedu.2016.03.003.

[30] Scratch. Scratch, 2026. URL `https://scratch.mit.edu/`. Last Accessed: 28.05.2026.

[31] Karim Shabani, Mohammad Khatib, and Saman Ebadi. Vygotsky's zone of proximal development: Instructional implications and teachers' professional development. *English Language Teaching*, 3(4):237–248, 2010. doi: 10.5539/elt.v3n4p237.

[32] Valerie J. Shute. Focus on formative feedback. *Review of Educational Research*, 78(1): 153–189, 2008. doi: 10.3102/0034654307313795.

[33] Snap! Snap!, 2026. URL `https://snap.berkeley.edu`. Last Accessed: 28.05.2026.

[34] Svelte Team. Async svelte documentation, . URL `https://svelte.dev/docs/svelte/await-expressions`. Last Accessed: 28.05.2026.

[35] Svelte Team. Sveltekit remote functions, . URL `https://svelte.dev/docs/kit/remote-functions#Single-flight-mutations`. Last Accessed: 28.05.2026.

[36] Svelte Team. Sveltekit documentation. `https://svelte.dev/docs/kit`, 2024. Last Accessed: 28.05.2026.

[37] John Sweller. Cognitive load during problem solving: Effects on learning. *Cognitive Science*, 12(2):257–285, 1988. doi: 10.1207/s15516709cog1202_4.

[38] L. S. Vygotsky. *Mind in Society: The Development of Higher Psychological Processes*. Harvard University Press, Cambridge, MA, 1980.

[39] David Weintrop and Uri Wilensky. To block or not to block, that is the question: Students' perceptions of blocks-based programming. In *Proceedings of the 14th International Conference on Interaction Design and Children*, pages 199–208. ACM, 2015. doi: 10.1145/2771839.2771860.

[40] Jeannette M. Wing. Computational thinking. *Communications of the ACM*, 49(3):33–35, 2006. doi: 10.1145/1118178.1118215.

[41] David Wood, Jerome S. Bruner, and Gail Ross. The role of tutoring in problem solving. *Journal of Child Psychology and Psychiatry*, 17(2):89–100, 1976. doi: 10.1111/j.1469-7610.1976.tb00381.x.

8 Evaluation Questionnaire

Bachelorarbeit Evaluation

Ziel dieser Evaluation ist es, die Software aus pädagogischer Perspektive zu bewerten. Im Mittelpunkt stehen Verständlichkeit, Benutzerfreundlichkeit, pädagogische Eignung,wahrgenommener Nutzen und mögliche Verbesserungspotenziale.

Die Evaluation richtet sich an Personen mit Erfahrung in der Arbeit mit Kindern bzw. Schülerinnen und Schülern. Aufgrund der kleinen Anzahl an Teilnehmenden ist die Evaluation nicht repräsentativ, sondern dient als explorative Expertinnen-Einschätzung.

Die Fragen orientieren sich inhaltlich an etablierten Konzepten wie dem Technology Acceptance Model (TAM) und der System Usability Scale (SUS), wurden jedoch für den pädagogischen Kontext dieser Arbeit angepasst.

* Indicates required question

Hinweise für die TeilnehmerInnen

Vielen Dank, dass du an der Evaulation teilnimmst. Ziel ist es nicht, dich zu testen, sondern die Software zu bewerten. Bitte nutze die Anwendung sol, als würdest du sie in einem pädagogischen Kontext betrachten.

Achte beim Ausprobieren besonders auf folgende PunkteÖ

- Ist die Software verständlich und intuitiv bedienbar?
- Sind Aufgaben, Erklärungen und Rückmeldungen für Kinder bzw. Schüler:innen nachvollziehbar?
- Kann die Sofrware Lernprozesse sinnvoll unterstützen?
- Welche Schwierigkeiten könnten bei der Nutzung auftreten?
- Welche Verbersserungen wären für den Einsatz der Praxis sinnvoll_

Es gibt keine richtigen oder falschen Antworten. Bitte antworte möglichst ehrlich und konkret. Wenn eine Frage nicht beurteilt werden kann, kann die Option „nicht beurteilbar“ gewählt werden.

Einwillung und Datenschutz

Bitte bestätige vor beginnd er Evaulation die folgenden Punkte:

- Ich wurde über Ziel und Ablauf der Evaluation informiert.
- Ich nehme freiwillig an der Evaluation teil.
- Meine Antworten dürfen anonymisiert für die Bachelorarbeit ausgewertet werden.
- Mir ist bewusst, dass keine personenbezogenen Daten veröffentlicht werden

Anleitung für die Durchführung

Ablauf:

1. Kurze Einführung in Ziel und Kontext der Software
2. Freies Ausprobieren der Anwendung
3. Bearbeitung der vorgegebenen Evaluationsaufgaben
4. Ausfüllen des Fragebogens
5. Kurzes offenes Feedbackgespräch anhand der Leitfragen am Ende des Dokuments.

Rolle Der Evaluationsleitung:

- Die TeilnehmerIn soll möglichst selbstständig mit der Software interagieren.
- ei technischen Problemen darf unterstützt werden
- Ihaltliche Hinweise sollten nur gegeben werden, wenn die TeilnehmerIn sonst nicht fortfahren kann.
- Auffälligkeiten, Unsicherheiten und spontane Kommentare können als Beobachtungs-notizen festgehalten werden.

1. Was ist dein Pädagogischer Hintegrund? *

_____

2. Erfahrung mit Kindern oder Schüler:innen? *

Tick all that apply.

- [ ] Kindergarten
- [ ] Schule
- [ ] Kinderbetreuung
- [ ] Pädagogishce Ausbildung oder Studium
- [ ] Other: _____

3. Eigene Erfarhrung mit Lernsoftware oder digitalen Lernumgebungen: *

Tick all that apply.

- [ ] Kein oder kaum Erfahrung
- [ ] Gelegentliche Erfahrung
- [ ] Regelmäßige Erfahrung
- [ ] Sehr viel Erfahrung

Evaluationsaufgaben

Bitte bearbeite die folgenden Aufgaben nacheinander. Notiere dir gerne währenddessen Gedanken, Probleme oder Verbesserungsvorschläge.

4. Aufgabe 1: Ersete Orientierung *

Öffne die Anwendung und verschaffe dir einen Überblick. Versuche zu erkennen, was die Software leisten soll und welche Funktionen oder Bereiche besonders wichtig sind.

5. Aufgabe 2: Beispielhafte Nutzung *

Bearbeite eine beispielhafte Übung bzw. einen typischen Ablauf in der Software. Versuche dabei, die Perspektive eines Kindes oder einer Schülerin bzw. eines Schülers einzunehmen.

6. Aufgabe 3: Rückmeldungen und Hilfestellungen *

   Achte darauf, welche Rückmeldungen, Hinweise oder Erklärungen die Software gibt. Prüfe,
   ob diese verständlich, hilfreich und pädagogisch sinnvoll sind.

7. Aufgabe 4: Pädagogischer Einsatz *

   Überlege, ob und wie die Software in einem pädagogischen Kontext eingesetzt werden könnte. Berücksichtige dabei mögliche Zielgruppen, Lernziele, Chancen und Grenzen.

SkalenfragenUntitled title

Bitte bewerte die folgenden Aussagen auf einer Skala von 1 bis 5.

1. stimme gar nicht zu
2. stimem eher nicht zu
3. teils/teils
4. stimme eher zu
5. stimme voll zu
6. nicht beurteilbar n.b

8. Die Software ist insgesamt leicht verständlich. *

Mark only one oval.

1	2	3	4	5	6	
◯	◯	◯	◯	◯	◯	n.b

9. Die Bedienung der Software ist intuitiv. *

Mark only one oval.

1	2	3	4	5	6	
◯	◯	◯	◯	◯	◯	n.b

10. Ich konnte mich schnell in der Anwendung zurechtfinden. *

Mark only one oval.

1	2	3	4	5	6	
◯	◯	◯	◯	◯	◯	n.b

11. Die wichtigsten Funktionen sind gut auffindbar. *

Mark only one oval.

1	2	3	4	5	6	
◯	◯	◯	◯	◯	◯	n.b

12. Die Oberfläche wirkt übersichtlich und nicht überladen. *

Mark only one oval.

	1	2	3	4	5	6	
	◯	◯	◯	◯	◯	◯	n.b

13. Die verwendeten Begriffe und Erklärungen sind verständlich. *

Mark only one oval.

	1	2	3	4	5	6	
	◯	◯	◯	◯	◯	◯	n.b

14. Die Software ist für Kinder bzw. Schüler:innen grundsätzlich geeignet. *

Mark only one oval.

	1	2	3	4	5	6	
	◯	◯	◯	◯	◯	◯	n.b

15. Die Aufgaben oder Inhalte wirken altersgerecht *

Mark only one oval.

	1	2	3	4	5	6	
	◯	◯	◯	◯	◯	◯	n.b

16. Die Software kann Lernprozesse sinnvoll unterstützen. *

Mark only one oval.

1	2	3	4	5	6	
◯	◯	◯	◯	◯	◯	n.b

17. Die Rückmeldungen der Software sind für Lernende hilfreich. *

Mark only one oval.

1	2	3	4	5	6	
◯	◯	◯	◯	◯	◯	n.b

18. Die Software unterstützt selbstständiges Ausprobieren und Lernen. *

Mark only one oval.

1	2	3	4	5	6	
◯	◯	◯	◯	◯	◯	n.b

19. Die Software kann Motivation oder Interesse am Thema fördern. *

Mark only one oval.

1	2	3	4	5	6	
◯	◯	◯	◯	◯	◯	n.b

20. Ich halte die Software für nützlich im Bildungskontext. *

Mark only one oval.

1	2	3	4	5	6	
◯	◯	◯	◯	◯	◯	n.b

21. Ich könnte mir vorstellen, die Software in einer pädagogischen Situation einzusetzen. *

Mark only one oval.

1	2	3	4	5	6	
◯	◯	◯	◯	◯	◯	n.b

22. Der Aufwand zur Nutzung erscheint im Verhältnis zum Nutzen angemessen. *

Mark only one oval.

1	2	3	4	5	6	
◯	◯	◯	◯	◯	◯	n.b

23. Die Software passt grundsätzlich zu pädagogischen Arbeits- oder Lernkontexten. *

Mark only one oval.

1	2	3	4	5	6	
◯	◯	◯	◯	◯	◯	n.b

24. Ich würde die Software nach Verbesserungen * oder Anpassungen weiterempfehlen.

Mark only one oval.

1	2	3	4	5	6	
◯	◯	◯	◯	◯	◯	n.b

25. Die Software macht insgesamt einen * ausgereiften Eindruck

Mark only one oval.

1	2	3	4	5	6	
◯	◯	◯	◯	◯	◯	n.b

26. Die Software erfüllt ihren Zweck * nachvollziehbar.

Mark only one oval.

1	2	3	4	5	6	
◯	◯	◯	◯	◯	◯	n.b

27. Insgesamt bewerte ich die Software positiv. *

Mark only one oval.

1	2	3	4	5	6	
◯	◯	◯	◯	◯	◯	n.b

28. Was hat dir an der Software besonders gut gefallen? *

29. Was war unklar, irritierend oder schwer verständlich? *

30. Welche Probleme könnten Kinder bzw. Schüler:innen bei der Nutzung haben? *

31. Welche Funktionen, Erklärungen oder Inhalte fehlen aus deiner Sicht? *

32. Welche Verbesserungen wären für den pädagogischen Einsatz besonders wichtig? *

33. Für welche Zielgruppe würdest du die Software am ehesten geeignet sehen? *

Tick all that apply.

- [ ] Kindergarten / Vorschule
- [ ] Primarstufe
- [ ] Sekundarstufe 1
- [ ] Sekundarstufe 2
- [ ] Außerschulische Kinder- oder Jugendarbeit
- [ ] Nicht geeignet für einen pädagogischen Einsatz
- [ ] Other: _____

34. Würdest du die Software in einem pädagogischen Kontext einsetzen? Warum oder warum nicht? *

35. Welche abschließenden Hinweise oder Empfehlungen möchtest du geben? *