CodeQuest - Project Proposal (Edu. App Design)

Members:
Anthony, Mikhail, Ajani, Victor

Introduction:

We found that many computer science learning apps focus primarily on teaching specific programming languages rather than the concepts. While learning syntax is important, it often overshadows the ideas that drive computer science,leaving learners with a weaker foundation. Computer science, at its core, is problem solving.

When learners focus too heavily on syntax, they may become proficient at writing code without full understanding of why a solution works, how it scales, or how it could be improved. Concepts such as data structures, algorithmic complexity, recursion and abstraction form the backbone of the discipline. Without a firm grasp of these principles, adapting to new languages or technologies becomes significantly more difficult.

Our goal is to address this gap by teaching core concepts primarily through pseudocode rather than any single language. Pseudocode allows learners to concentrate on logic, structure, and efficiency without being distracted by language-specific rules or conventions. By separating concepts from syntax, learners can focus on understanding how problems are solved rather than just how they are written. This encourages deeper comprehension and transferable problem-solving skills.

In addition to covering foundational theory, we aim to help learners develop an intuition for efficiency and design. Understanding why one algorithm performs better than another, or why a particular data structure is appropriate in a given scenario. These insights empower learners to approach unfamiliar problems with confidence rather than relying on memorized patterns.

This approach does not dismiss the importance of programming languages. Practical implementation remains essential, and fluency in at least one language is a critical skill. However, languages are tools. The principles behind them remain constant. By prioritizing conceptual understanding first, we prepare learners to adapt quickly, think critically, and apply their knowledge in any environment.

Ultimately, our focus is not just on teaching people to code, but on teaching them to think like computer scientists by solving problems effectively regardless of the technology in front of them.

Background

Computer science education at the secondary level has increasingly been positioned as foundational preparation for university study and participation in the digital economy. Contemporary curriculum standards emphasize that high school students should develop not only programming ability, but also conceptual understanding of algorithms, abstraction, data structures, and object oriented design. The AP Computer Science A Course and Exam Description (College Board 2025) explicitly includes algorithm development, arrays and collections, object oriented programming (OOP), and program design. Broader analyses of K–12 computing reform similarly argue that rigorous engagement with computational thinking and formal reasoning is essential for preparing students for post secondary pathways (Vegas et al. 2021; Grover 2024).

Despite this curricular emphasis, computing education research continues to document persistent conceptual challenges among novice programmers. Bastian (2025) shows that students frequently rely on intuitive but incorrect reasoning strategies when interpreting code, leading to misconceptions about control flow, variable state, and loop behavior. Large scale analysis of student submissions further demonstrates that recurring logical errors reflect deeper misunderstandings of abstraction and execution models rather than simple syntactic slips (HoQ et al. 2025). Think-aloud studies of student reasoning during refactoring tasks reveal that learners who can produce syntactically correct code often struggle to articulate structural relationships between methods, classes, and data (McCracken et al. 2024). These findings suggest that the primary barrier in early computer science education is not exposure to syntax, but the development of accurate mental models of program behavior and algorithmic reasoning.

Meta-analyses of programming education research reinforce this conclusion. Malmi (2025) identifies abstraction, debugging, and algorithmic reasoning as persistent themes in student difficulty across multiple institutional contexts. Similarly, Chouhan et al. (2025) report that educators consistently cite conceptual mastery of algorithms and data structures as a major instructional challenge in introductory courses. When students transition from high school to university level computer science, these fragile conceptual foundations often result in difficulty with Big O analysis, recursion, memory models, and formal data structure trade offs.

Active learning strategies have been shown to mitigate these conceptual gaps. Córdova-Esparza et al. (2024) demonstrate that interactive, student centered approaches, particularly those involving structured problem solving with immediate feedback, significantly improve conceptual understanding in computer science courses. Broader K-12 computing reform efforts similarly advocate scaffolded progression, iterative practice, and engagement driven learning environments (Grover 2024; Vegas et al. 2021). These findings indicate that

effective tools must move beyond passive instruction and instead require learners to actively reason about code structure, execution, and efficiency.

Target Audience

The primary target audience for Code Quest is Grade 10–12 high school students who are planning to pursue computer science, engineering, or related STEM programs at the university level. This includes students currently enrolled in AP Computer Science A or equivalent provincial computer science courses, as well as motivated, self-directed learners who are preparing for post-secondary study on their own. Most of these students have some exposure to coding, often through classroom assignments, online tutorials, or block-based tools, but still feel unsure when they have to understand an unfamiliar solution, trace logic carefully, or explain *why* code behaves a certain way.

Code Quest is designed for students who are trying to cross the gap between “basic syntax practice” and the more analytical thinking expected in first-year university CS. A common issue at this stage is that students can follow examples, but struggle when they are asked to interpret pseudocode, predict outputs, handle edge cases, or translate an algorithm into working code without being shown the exact steps. The platform focuses on building those core skills, reading and interpreting logic, breaking problems into steps, and implementing programming courses and need a structured way to catch up. This may include learners who are switching into CS late, students coming from non-traditional pathways, or anyone who finds typical “write this exact program” exercises too jumpy without enough practice in understanding the underlying reasoning. Over time, Code Quest can also support teachers and tutors as a classroom or support tool, since its practice-oriented format makes it useful for quick reinforcement, targeted review, and measurable skill growth.

Learning Outcomes

Guided by curriculum expectations and well-documented areas where students typically struggle, Code Quest is designed to produce learning outcomes that are both conceptual (students understand *why* something works) and practical (students can apply it in code). Each outcome is written to be measurable through the platform’s activities, pseudocode interpretation, tracing, implementation tasks, and explanation-based questions so students aren’t only “getting the right answer,” but building reliable reasoning habits that transfer into first-year university computer science.

Algorithmic Reasoning: Students will analyze algorithms by identifying dominant operations, comparing alternative solutions, and expressing efficiency using Big-O notation. Rather than treating complexity as memorized formulas, students will practice connecting real code patterns

(nested loops, recursion, early exits) to runtime behavior, and will justify *why* one approach scales better than another.

Data Structure Understanding: Students will explain, construct, and choose between fundamental data structures—such as arrays, linked lists, stacks/queues, trees, and hash-based structures—based on performance and design trade-offs. They will be able to describe how data is stored and accessed, predict which operations are efficient or costly, and select structures that best match a problem's constraints (for example, when fast lookup matters versus when ordered traversal matters).

Execution Model Comprehension: Students will accurately trace program execution step-by-step, including control flow, variable state changes, and function calls. This includes correctly interpreting recursion, understanding the role of the call stack, and recognizing base cases and repeated subproblems. By strengthening tracing skills, students improve their ability to predict output, catch logic errors, and understand unfamiliar code—skills that are essential for university-level programming.

Object-Oriented Design: Students will apply core object-oriented principles such as abstraction, encapsulation, modularity, and class responsibility. They will practice designing simple classes from problem statements, choosing appropriate fields and methods, and explaining how objects interact. The goal is not only writing classes that compile, but demonstrating clean reasoning about structure, reuse, and maintainability.

Debugging and Code Explanation: Students will identify logical errors, interpret incorrect behavior, and explain precisely why a program produces a given result. This includes recognizing off-by-one mistakes, incorrect conditionals, faulty loop logic, and misuse of variables or data structures. Students will also practice writing short, human-readable explanations of what code is doing—building communication skills that matter in labs, exams, and collaborative work.

Collaborative Development Practices: Students will demonstrate a basic understanding of collaborative workflows, especially version control concepts (commits, branches, pull requests, resolving conflicts, and readable commit messages). Even at a beginner level, students benefit from learning how real software is developed in teams, and this outcome prepares them for group projects and future internships.

Together, these learning outcomes are meant to directly address the most common conceptual bottlenecks reported in computing education research (Bastian 2025; Malmi 2025; McCracken et al. 2024), while keeping the focus on what students actually need: the ability to read, reason about, and implement solutions confidently—not just memorize syntax.

Related Work and Limitations of Existing Tools

There already exist several educational platforms that aim to address aspects of programming education. From our research, however, we have found that they fail to

Several widely used educational platforms address aspects of programming instruction, yet do not fully target the conceptual challenges documented in the literature.

Duolingo demonstrates the effectiveness of gamified micro-lessons and streak-based motivation, but its model primarily supports vocabulary acquisition rather than structural reasoning. Codecademy provides interactive tutorials focused on syntax and guided exercises; however, instruction is often linear and may not emphasize explicit complexity analysis or abstraction reasoning. LeetCode offers extensive algorithmic practice but assumes prior conceptual mastery and provides limited scaffolding for high school learners. Block-based platforms such as Scratch are effective for early engagement but abstract away syntax and formal complexity considerations.

These platforms represent dominant delivery models in computing education, yet research suggests that syntax practice alone does not resolve misconceptions about execution, abstraction, and algorithmic reasoning (Bastian 2025; HoQ et al. 2025). Furthermore, tools designed primarily for interview preparation do not typically incorporate structured progression or active-learning scaffolds recommended in K–12 reform literature (Grover 2024).

How Code Quest Addresses Identified Gaps

Code Quest is intentionally designed around the conceptual weaknesses identified in contemporary research.

1. Syntax Sprint (Code Spelling Bee)

Users are shown pseudo code and must translate it into a programming language of their selection. Code Quest will give them feedback on their syntax errors and any other errors they might have made. This activity reinforces control flow reasoning and variable state understanding, directly targeting misconceptions documented by Bastian (2025).

2. Big-O Challenge Arena

Students are given the pseudo code to algorithms, and must determine their space and time complexities. Depending on the difficulty of the game, they might have to write their own code to match certain space and time constraints. This addresses algorithmic reasoning gaps identified by Malmi (2025) and Chouhan et al. (2025).

3. Data Structure Builder

Through drag-and-drop interaction, students construct linked lists, binary trees, and hash structures while visualizing memory relationships. This supports abstraction and structural reasoning, responding to difficulties observed in think-aloud studies (McCracken et al. 2024).

4. Recursion and Memory Lab

Interactive stack visualizations allow students to step through recursive calls and observe frame allocation and return values. This directly supports accurate mental model formation for execution flow, addressing misconceptions revealed in error-analysis studies (HoQ et al. 2025).

5. Debugging and Code Explanation Quests

Students identify logical flaws in code snippets and select explanations for observed behavior. These explanation-driven tasks align with research indicating that conceptual understanding improves when learners articulate reasoning (Córdova-Esparza et al. 2024).

6. Git Quest (Version Control Simulation)

Students simulate commits, branching, and merge conflict resolution within a simplified repository interface. This introduces collaborative development practices increasingly emphasized in modern computing curricula (Grover 2024).

Synthesis

The literature establishes that high school curricula require engagement with core computational concepts (College Board 2025), yet students consistently struggle with abstraction, algorithmic reasoning, and accurate execution models (Bastian 2025; HoQ et al. 2025; McCracken et al. 2024). Research further demonstrates that interactive, scaffolded, and active-learning environments significantly improve conceptual mastery (Córdova-Esparza et al. 2024).

Code Quest integrates these research findings into a structured, gamified platform specifically designed for high school students preparing for university-level computer science. By combining retrieval practice, visual modeling, structured progression, and immediate feedback within concept-focused learning objects, the platform addresses documented educational gaps while aligning directly with established curriculum standards.

TO SUBMIT:

CodeQuest - Project Proposal (Edu. App Design)(V2)

Introduction:

We found that many computer science learning apps focus primarily on teaching specific programming languages rather than the concepts. While learning syntax is important, it often overshadows the ideas that drive computer science,leaving learners with a weaker foundation. Computer science, at its core, is problem solving.

When learners focus too heavily on syntax, they may become proficient at writing code without full understanding of why a solution works, how it scales, or how it could be improved. Without a firm grasp of these foundational concepts, adapting to new languages or technologies becomes significantly more difficult.

Our goal is to address this gap by teaching core concepts primarily through pseudocode rather than any single language. Pseudocode allows learners to concentrate on logic, structure, and efficiency without being distracted by language-specific rules or conventions. By separating concepts from syntax, learners can focus on understanding how problems are solved rather than just how they are written. This encourages deeper comprehension and transferable problem-solving skills.

In addition to covering foundational theory, we aim to help learners develop an intuition for efficiency and design. Understanding why one algorithm performs better than another, or why a particular data structure is appropriate in a given scenario. These insights empower learners to approach unfamiliar problems with confidence rather than memorizing patterns.

Practical implementation remains essential, and fluency in at least one language is a critical skill. However, languages are tools. The principles behind them remain constant. By prioritizing conceptual understanding first, we prepare learners to adapt quickly, think critically, and apply their knowledge in any environment.

Ultimately, our focus is not just on teaching people to code, but on teaching them to think like computer scientists by solving problems effectively regardless of the technology in front of them.

Background

Computer science education at the secondary level has increasingly been positioned as foundational preparation for university study and participation in the digital economy. Contemporary curriculum standards emphasize that high school students should develop not only programming ability, but also conceptual understanding of algorithms, abstraction, data structures, and object-oriented design. The *AP Computer Science A Course and Exam Description* [1] explicitly includes algorithm development, arrays and collections, object-oriented programming (OOP), and program design. Broader analyses of K-12 computing reform similarly argue that rigorous engagement with computational thinking and formal reasoning is essential for preparing students for post-secondary pathways [2].

Despite this curricular emphasis, computing education research continues to document persistent conceptual challenges among novice programmers. Bastian (2025) shows that students frequently rely on intuitive but incorrect reasoning strategies when interpreting code, leading to misconceptions about control flow, variable state, and loop behavior [4] . Large-scale analysis of student submissions further demonstrates that recurring logical errors reflect deeper misunderstandings of abstraction and execution models rather than simple syntactic slips (HoQ et al. 2025) [5]. Think-aloud studies of student reasoning during refactoring tasks reveal that learners who can produce syntactically correct code often struggle to articulate structural relationships between methods, classes, and data [6]. These findings suggest that the primary barrier in early computer science education is not exposure to syntax, but the development of accurate mental models of program behavior and algorithmic reasoning.

Meta-analyses of programming education research reinforce this conclusion. Malmi (2025) identifies abstraction, debugging, and algorithmic reasoning as persistent themes in student difficulty across multiple institutional contexts [7]. Similarly, Chouhan et al. (2025) report that educators consistently cite conceptual mastery of algorithms and data structures as a major instructional challenge in introductory courses [8]. When students transition from high school to university-level computer science, these weak conceptual foundations often result in difficulty with Big-O analysis, recursion, memory models, and formal data structure trade-offs.

Active learning strategies have been shown to mitigate these conceptual gaps. Córdova-Esparza et al. (2024) demonstrate that interactive, student-centered approaches,particularly those involving structured problem solving with immediate feedback,significantly improve conceptual understanding in computer science courses [9].Broader K-12 computing reform efforts similarly advocate scaffolded progression, iterative practice, and engagement-driven learning environments [2]. These findings indicate that

effective tools must move beyond passive instruction and instead require learners to actively reason about code structure, execution, and efficiency.

Target Audience

The primary target audience for Code Quest is Grade 10-12 high school students who are planning to pursue computer science, engineering, or related STEM programs at the university level. This includes students currently enrolled in AP Computer Science A or equivalent provincial computer science courses, as well as motivated, self-directed learners who are preparing for post-secondary study on their own. Most of these students have some exposure to coding, often through classroom assignments, online tutorials, or block-based tools, but still feel unsure when they have to understand an unfamiliar solution, trace logic carefully, or explain *why* code behaves a certain way.

Code Quest is designed for students who are trying to cross the gap between “basic syntax practice” and the more analytical thinking expected in first-year university CS. A common issue at this stage is that students can follow examples, but struggle when they are asked to interpret pseudocode, predict outputs, handle edge cases, or translate an algorithm into working code without being shown the exact steps. The platform focuses on building those core skills, reading and interpreting logic, breaking problems into steps, and implementing programming courses and need a structured way to catch up. This may include learners who are switching into CS late, students coming from non-traditional pathways, or anyone who finds typical “write this exact program” exercises too jumpy without enough practice in understanding the underlying reasoning. Over time, Code Quest can also support teachers and tutors as a classroom or support tool, since its practice-oriented format makes it useful for quick reinforcement, targeted review, and measurable skill growth.

Learning Outcomes

Guided by curriculum expectations and well-documented areas where students typically struggle, Code Quest is designed to produce learning outcomes that are both conceptual (students understand *why* something works) and practical (students can apply it in code). Each outcome is written to be measurable through the platform’s activities, pseudocode interpretation, tracing, implementation tasks, and explanation-based questions so students aren’t only “getting the right answer,” but building reliable reasoning habits that transfer into first-year university computer science.

Algorithmic Reasoning: Students will analyze algorithms by identifying dominant operations, comparing alternative solutions, and expressing efficiency using Big-O notation. Rather than treating complexity as memorized formulas, students will practice connecting real code patterns

(nested loops, recursion, early exits) to runtime behavior, and will justify *why* one approach scales better than another.

Data Structure Understanding: Students will explain, construct, and choose between fundamental data structures,such as arrays, linked lists, stacks/queues, trees, and hash-based structures based on performance and design trade-offs. They will be able to describe how data is stored and accessed, predict which operations are efficient or costly, and select structures that best match a problem's constraints (for example, when fast lookup matters versus when ordered traversal matters).

Execution Model Comprehension: Students will accurately trace program execution step-by-step, including control flow, variable state changes, and function calls. This includes correctly interpreting recursion, understanding the role of the call stack, and recognizing base cases and repeated subproblems. By strengthening tracing skills, students improve their ability to predict output, catch logic errors, and understand unfamiliar code,skills that are essential for university-level programming.

Object-Oriented Design: Students will apply core object-oriented principles such as abstraction, encapsulation, modularity, and class responsibility. They will practice designing simple classes from problem statements, choosing appropriate fields and methods, and explaining how objects interact. The goal is not only writing classes that compile, but demonstrating clean reasoning about structure, reuse, and maintainability.

Debugging and Code Explanation: Students will identify logical errors, interpret incorrect behavior, and explain precisely why a program produces a given result. This includes recognizing off-by-one mistakes, incorrect conditionals, faulty loop logic, and misuse of variables or data structures. Students will also practice writing short, human-readable explanations of what code is doing, building communication skills that matter in labs, exams, and collaborative work.

Collaborative Development Practices: Students will demonstrate a basic understanding of collaborative workflows, especially version control concepts (commits, branches, pull requests, resolving conflicts, and readable commit messages). Even at a beginner level, students benefit from learning how real software is developed in teams, and this outcome prepares them for group projects and future internships.

Together, these learning outcomes are meant to directly address the most common conceptual bottlenecks reported in computing education research (Bastian 2025; Malmi 2025; McCracken et al. 2024), while keeping the focus on what students actually need: the ability to read, reason about, and implement solutions confidently, not just memorize syntax.

Related Work and Limitations of Existing Tools

There already exist several educational platforms that aim to address aspects of programming education. From our research, however, we have found that they fail to

Several widely used educational platforms address aspects of programming instruction, yet do not fully target the conceptual challenges documented in the literature.

Duolingo demonstrates the effectiveness of gamified micro-lessons and streak-based motivation, but its model primarily supports vocabulary acquisition rather than structural reasoning. Codecademy provides interactive tutorials focused on syntax and guided exercises; however, instruction is often linear and may not emphasize explicit complexity analysis or abstraction reasoning. LeetCode offers extensive algorithmic practice but assumes prior conceptual mastery and provides limited scaffolding for high school learners. Block-based platforms such as Scratch are effective for early engagement but abstract away syntax and formal complexity considerations.

These platforms represent dominant delivery models in computing education, yet research suggests that syntax practice alone does not resolve misconceptions about execution, abstraction, and algorithmic reasoning [4][5]. Furthermore, tools designed primarily for interview preparation do not typically incorporate structured progression or active-learning scaffolds recommended in K–12 reform literature (Grover 2024).

How Code Quest Addresses Identified Gaps

Code Quest is intentionally designed around the conceptual weaknesses identified in contemporary research.

1. Syntax Sprint (Code Spelling Bee)

Users are shown pseudo code and must translate it into a programming language of their selection. Code Quest will give them feedback on their syntax errors and any other errors they might have made. This activity reinforces control flow reasoning and variable state understanding, directly targeting misconceptions documented by Bastian (2025).

2. Big-O Challenge Arena

Students are given the pseudo code to algorithms, and must determine their space and time complexities. Depending on the difficulty of the game, they might have to write their own code to match certain space and time constraints. This addresses algorithmic reasoning gaps identified by Malmi (2025) and Chouhan et al. (2025). [8]

3. Data Structure Builder

Through drag-and-drop interaction, students construct linked lists, binary trees, and hash structures while visualizing memory relationships. This supports abstraction and structural reasoning, responding to difficulties observed in think-aloud studies (McCracken et al. 2024).[6]

4. Recursion and Memory Lab

Interactive stack visualizations allow students to step through recursive calls and observe frame allocation and return values. This directly supports accurate mental model formation for execution flow, addressing misconceptions revealed in error-analysis studies (HoQ et al. 2025).[5]

5. Debugging and Code Explanation Quests

Students identify logical flaws in code snippets and select explanations for observed behavior. These explanation-driven tasks align with research indicating that conceptual understanding improves when learners articulate reasoning (Córdova-Esparza et al. 2024).[9]

6. Git Quest (Version Control Simulation)

Students simulate commits, branching, and merge conflict resolution within a simplified repository interface. This introduces collaborative development practices increasingly emphasized in modern computing curricula (Grover 2024).

Synthesis

The literature establishes that high school curricula require engagement with core computational concepts[1],yet students consistently struggle with abstraction, algorithmic reasoning, and accurate execution models4][5][6]. Research further demonstrates that interactive, scaffolded, and active-learning environments significantly improve conceptual mastery (Córdova-Esparza et al. 2024).[9]

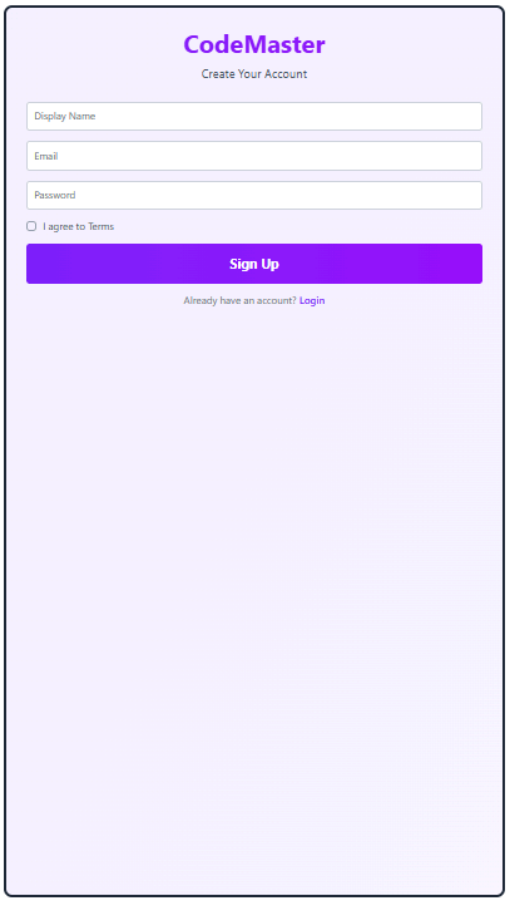
Code Quest integrates these research findings into a structured, gamified platform specifically designed for high school students preparing for university-level computer science. By combining retrieval practice, visual modeling, structured progression, and immediate feedback within concept-focused learning objects, the platform addresses documented educational gaps while aligning directly with established curriculum standards.

Storyboard:

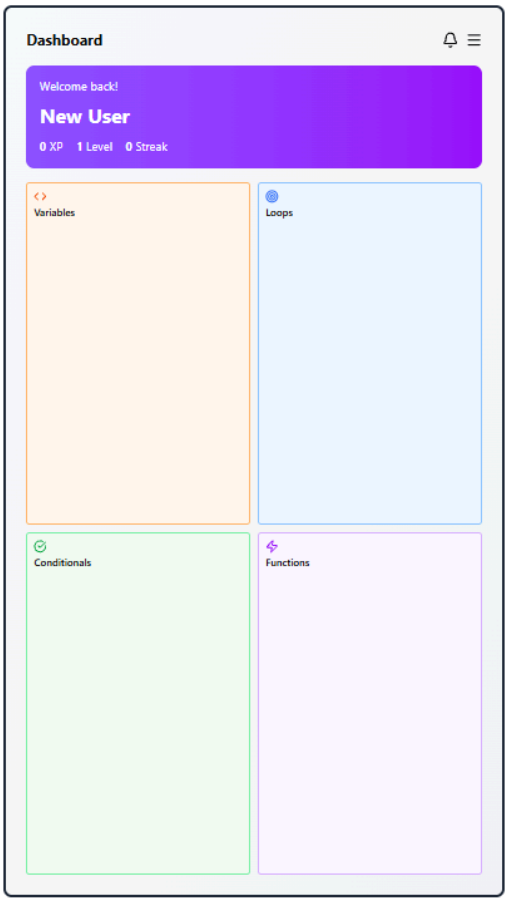
We created our storyboard using figma to show how the app functionality will work.

Use Case 1: New User Registration

A new student discovers CodeMaster and creates an account to start learning



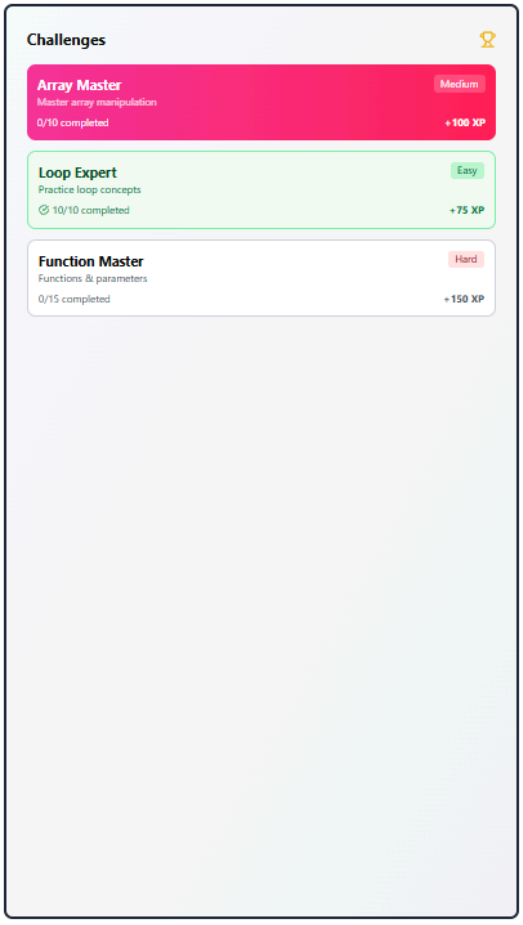
1. Signup Page



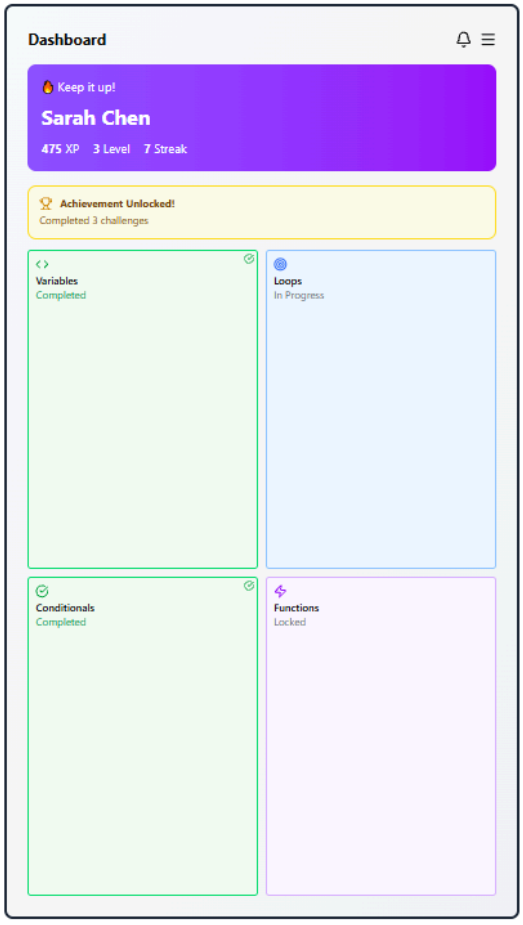
2. Dashboard (Home)

Use Case 3: Challenge Completion

Student attempts challenges to test their knowledge and earn rewards



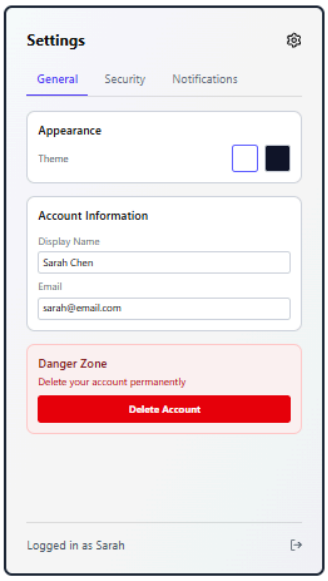
6. Challenges Overview



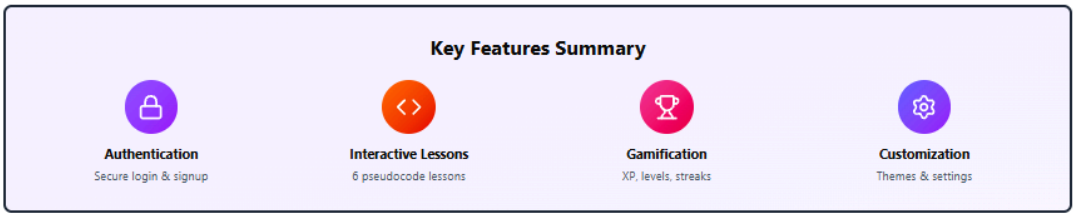
7. Progress Tracking

Use Case 4: Account Management

Student customizes their experience through settings (theme, notifications, security)



8. Settings & Preferences



CodeMaster Educational Platform - Making Learning Interactive & Engaging

Presentation Video & GitHub:

Github:
https://github.com/AnthonyCiceuOTU/CodeQuest

Presentation video:
https://www.youtube.com/watch?v=9IhLxzMe-E0

References:

[1] *AP Computer Science A Course and Exam Description*. College Board, 2025. https://apcentral.collegeboard.org/media/pdf/ap-computer-science-a-course-and-exam-description.pdf

[2] S. Grover and R. Pea. 2013. *Computational Thinking in K–12: A Review of the State of the Field*. Educational Researcher 42, 1 (2013). https://www.researchgate.net/publication/258134754_Computational_Thinking_in_K-12_A_Review_of_the_State_of_the_Field

[3] M. Bastian and A. Mühling. 2025. *Misconceptions in Programming: Intuitive Reasoning and Tracing Task Performance Across Experience Levels*. In *Proceedings of the 2025 ACM International Computing Education Research (ICER '25)*, ACM, 141–154. https://dl.acm.org/doi/10.1145/3702652.3744209

[4] (2025). *EDM 2025: Long Papers — Large-Scale Analysis of Student Submissions*. In *Proceedings of the 2025 Educational Data Mining Conference* (EDM '25). https://educationaldatamining.org/EDM2025/proceedings/2025.EDM.long-papers.85/2025.EDM.long-papers.85.pdf

[5] Juha Helminen and Lauri Malmi. 2010. *Jype: A Program Visualization and Programming Exercise Tool for Python*. In *Proceedings of the 5th International Symposium on Software Visualization (SOFTVIS '10)*, ACM Press, 153–162. https://dl.acm.org/doi/10.1145/1879211.1879234

[6] (2024). *ArXiv Preprint: Exploring Structural Reasoning and Data Structures*. arXiv:2410.20875. https://arxiv.org/abs/2410.20875

[7] A. Chouhan, S. S. Ragavan, and A. Karkare. 2025. *CS Educator Challenges and Their Solutions: A Systematic Mapping Study*. CoRR abs/2511.02876 (Nov. 2025). https://www.researchgate.net/publication/397322147_CS_Educator_challenges_and_their_solutions_A_systematic_mapping_study

[8] *Educational Data Mining Long Papers — 2025 EDM 85: Large-Scale Analysis of Student Errors and Execution Models*. In *Proceedings of the 2025 Educational Data Mining Conference* and associated dataset. https://educationaldatamining.org/EDM2025/proceedings/2025.EDM.long-papers.85/index.html

[9] T. Córdova-Esparza, N. Martínez-Martínez, P. Hernández-García, and I. Martínez-Velázquez. 2024. *Interactive Problem Solving with Immediate Feedback Improves Conceptual Understanding in Computer Science Courses*. Information 16, 6 (2024), 469. https://www.mdpi.com/2078-2489/16/6/469