SMART LEARNING ECOSYSTEM FOR INTERACTIVE STEAM EDUCATION BASED ON GAMIFICATION AND IOT TECHNOLOGIES: A CASE STUDY USING APP INVENTOR

by

ARIS CHARONITIS (MTP275)

B.Sc., Department of Computer and Systems Engineering at University of Naples Federico II, 2009

A THESIS

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

SCHOOL OF ENGINEERING

HELLENIC MEDITERRANEAN UNIVERSITY

2026

Approved by:

Associate Profess
Spyros Panagiotakis

Abstract

The rapid evolution of the Fourth Industrial Revolution, compounded by the disruptions of the COVID-19 pandemic, has exposed critical deficiencies in conventional digital learning infrastructures, particularly in their capacity to sustain learner engagement, deliver hands-on experiential learning, and provide adaptive, timely assessment within asynchronous environments. This thesis presents the design, development, and functional validation of a Smart Learning Ecosystem (SLE) conceived to address these challenges through the integration of gamification, Internet of Things (IoT) technologies, and a structured visual-to-textual programming pathway within a STEAM educational framework.

The proposed ecosystem is architectured around Moodle as its central orchestration layer, augmented by MIT App Inventor for mobile IoT prototyping, Arduino-based physical computing for tangible experiential learning, H5P for interactive content delivery, and CodeRunner for automated programming assessment. A five-stage asynchronous pedagogical workflow — encompassing Exploration, Conceptualisation, Application, Evaluation, and Reflection — structures the learning progression, guiding learners from foundational concepts through to complex, real-world problem-solving tasks.

A principal contribution of this work is the Blockly-to-Python learning pathway, which systematically bridges the cognitive gap between block-based visual programming and text-based coding, mitigating the well-documented barriers that impede novice programmers in STEAM contexts. Gamification mechanisms, including experience points (XP), level progression, badge issuance, and conditional content unlocking, are embedded throughout the ecosystem to sustain motivation and persistence in asynchronous learning conditions. An integrated analytics and automated feedback infrastructure continuously monitors learner performance, enabling data-driven adaptive interventions.

The research is guided by a central question examining how the integrated deployment of block-based programming, IoT technologies, and gamification within a cohesive SLE contributes to adaptive STEAM education. Supporting sub-questions address architectural requirements, the effect of gamified progression on motivation, the efficacy of automated assessment for learning performance, and the contribution of physical computing to computational understanding.

Sample proof of concept and functional validation demonstrate that the ecosystem successfully operationalises its design objectives, providing a replicable, scalable, and pedagogically grounded framework for interactive STEAM education aligned with the demands of the contemporary digital learning environment.

Περίληψη

Η ραγδαία εξέλιξη της Τέταρτης Βιομηχανικής Επανάστασης, σε συνδυασμό με τις αναταραχές που προκάλεσε η πανδημία του COVID-19, ανέδειξε κρίσιμες ελλείψεις στις συμβατικές υποδομές ψηφιακής μάθησης, ιδίως όσον αφορά την ικανότητά τους να διατηρούν το ενδιαφέρον των μαθητών, να προσφέρουν βιωματική μάθηση και να παρέχουν προσαρμοστική και έγκαιρη αξιολόγηση σε ασύγχρονα περιβάλλοντα. Η παρούσα διατριβή παρουσιάζει τον σχεδιασμό, την ανάπτυξη και τη λειτουργική επικύρωση ενός Οικοσυστήματος Έξυπνης Μάθησης (SLE) που σχεδιάστηκε για να αντιμετωπίσει αυτές τις προκλήσεις μέσω της ενσωμάτωσης της παιχνιδοποίησης, των τεχνολογιών του Διαδικτύου των Πραγμάτων (IoT) και μιας δομημένης διαδρομής προγραμματισμού από οπτικά στοιχεία σε κείμενο, εντός ενός εκπαιδευτικού πλαισίου STEAM.

Το προτεινόμενο οικοσύστημα είναι δομημένο γύρω από το Moodle ως κεντρικό επίπεδο συντονισμού, ενισχυμένο από το MIT App Inventor με δημιουργία πρωτοτύπων IoT για κινητά, τη φυσική πληροφορική βασισμένη στο Arduino για απτή βιωματική μάθηση, το H5P για την παροχή διαδραστικού περιεχομένου και το CodeRunner για την αυτοματοποιημένη αξιολόγηση προγραμματισμού. Μια ασύγχρονη παιδαγωγική ροή εργασίας πέντε σταδίων — που περιλαμβάνει Εξερεύνηση, Σύλληψη, Εφαρμογή, Αξιολόγηση και Αναστοχασμό — δομεί την πρόοδο της μάθησης, καθοδηγώντας τους μαθητές από τις βασικές έννοιες έως σύνθετες εργασίες επίλυσης προβλημάτων του πραγματικού κόσμου.

Μια βασική συμβολή της παρούσας εργασίας είναι η μαθησιακή διαδρομή «Blockly-to-Python», η οποία γεφυρώνει συστηματικά το γνωστικό χάσμα μεταξύ της οπτικής προγραμματισμού με μπλοκ και της κωδικοποίησης με κείμενο, μετριάζοντας τα καλά τεκμηριωμένα εμπόδια που δυσχεραίνουν τους αρχάριους προγραμματιστές σε περιβάλλοντα STEAM. Μηχανισμοί παιχνιδοποίησης, όπως οι πόντοι εμπειρίας (XP), η πρόοδος σε επίπεδα, η απονομή διακριτικών και το ξεκλείδωμα περιεχομένου υπό όρους, είναι ενσωματωμένοι σε όλο το οικοσύστημα για τη διατήρηση της κινητοποίησης και της επιμονής σε συνθήκες ασύγχρονης μάθησης.

Μια ολοκληρωμένη υποδομή ανάλυσης και αυτοματοποιημένης ανατροφοδότησης παρακολουθεί συνεχώς την απόδοση των μαθητών, επιτρέποντας προσαρμοστικές παρεμβάσεις βασισμένες σε δεδομένα.

Η έρευνα καθοδηγείται από ένα κεντρικό ερώτημα που εξετάζει πώς η ολοκληρωμένη εφαρμογή της προγραμματισμού που βασίζεται σε μπλοκ, των τεχνολογιών IoT και της παιχνιδοποίησης εντός ενός συνεκτικού SLE συμβάλλει στην προσαρμοστική εκπαίδευση STEAM. Τα υποστηρικτικά δευτερεύοντα ερωτήματα αφορούν τις αρχιτεκτονικές απαιτήσεις, την επίδραση της παιχνιδοποιημένης προόδου στην παρακίνηση, την αποτελεσματικότητα της αυτοματοποιημένης αξιολόγησης της μαθησιακής απόδοσης και τη συμβολή της φυσικής πληροφορικής στην υπολογιστική κατανόηση..

Δείγματα απόδειξης της ιδέας και λειτουργικής επικύρωσης καταδεικνύουν ότι το οικοσύστημα υλοποιεί με επιτυχία τους σχεδιαστικούς του στόχους, παρέχοντας ένα αναπαραγώγιμο, επεκτάσιμο και παιδαγωγικά θεμελιωμένο πλαίσιο για διαδραστική εκπαίδευση STEAM, ευθυγραμμισμένο με τις απαιτήσεις του σύγχρονου ψηφιακού μαθησιακού περιβάλλοντος.

Table of Contents

Abstract 2
Περίληψη 4
Table of Contents 6
Acknowledgements 11
Dedication 12
List of Figures 13
List of Tables 18
Chapter 1 - Introduction 19
1.1 Background and Motivation 19
1.1.1 The Fourth Industrial Revolution, Pandemic, and the Digital Learning Imperative 19
1.1.2 The Critical Need for Progressive STEAM Education 20
1.2 Research Questions 21
1.3 Aim and Objectives 22
1.3.1 Overall Aim 22
1.3.2 Specific Objectives 22
1.4 Thesis Structure 23
Chapter 2 - Related technologies and Literature review 24
2.1 Related Technologies: The Building Blocks of the Ecosystem 24
2.1.1 Learning Management Systems (LMS): The Role of Moodle 24
2.1.2 Visual Programming and IoT Prototyping: MIT App Inventor 25
2.1.3 Physical Computing Foundation: Arduino Microcontrollers 25
2.1.4 Sensing and Acting: Sensors and Actuators 26
2.1.5 IoT Communication Protocols and Components 27
2.1.6 Automated Assessment Tools: The CodeRunner Plugin 27
2.1.7 Interactive Content Creation: H5P 28
2.1.8 The Target Text-Based Language: Python 28
2.2 Literature review 29
2.2.1 Introduction 29
2.2.2 Research Questions 30

2.2.3 Guide to Systematic Review 31
2.3 Results 37
2.3.1 Summary of findings 37
2.3.2 Overview of Selected Studies 38
2.3.3 RQ1 — Block-Based Programming, IoT, and Gamification Integration 38
2.3.4 RQ2 — Architectural Components of a Smart Learning Ecosystem 40
2.3.5 RQ3 — Gamified Progression and Learner Motivation 42
2.3.6 RQ4 — Integrated Analytics and Automated Assessment 44
2.3.7 RQ5 — IoT- Based Physical Computing and Computational Understanding 46
2.3.8 RQ6 — Blockly-to-Python Learning Pathway 48
2.3.9 Cross-Reference Summary: Papers to Research Questions 51
2.4 Clarification of Results and Conclusions 53
2.4.1 Discussion and Implications 54
2.4.2 Contribution of the Study 54
Chapter 3 - System Design and Architecture 55
3.1 Overall System Architecture and Design Philosophy 55
3.1.1 High-Level Architectural Diagram 56
3.1.2 Core Design Principles 56
3.2 The Asynchronous Learning and Assessment Workflow 58
3.2.1 Stage 1: Concept Introduction & Formative Check (Moodle + H5P) 59
3.2.2 Stage 2: Practical Application & Prototyping (App Inventor Practice → Moodle Development & Assessment) Purpose and Pedagogical Rationale 61
3.2.3 Stage 3: IoT and Physical Computing Practice (Arduino via mBlock → Moodle-Based Programming and Evaluation) 63
3.2.4 Stage 4: Stage 4 The Blockly Code Assessment Framework (CodeRunner in Moodle) 64
3.2.5 Stage 5: Knowledge Consolidation & Advanced Challenges (Moodle) 66
3.3 Detailed Component Integration and Data Flow 66
3.3.1 Moodle Course Structure for Asynchronous Flow 67
3.3.2 Gamification Engine Design 69
3.3.3 The Assessment Toolkit: H5P, Quiz, and CodeRunner 70

3.4 The Analytics and Feedback Loop 73
3.4.1 Data Collection Points 73
3.4.2 The Analytics Dashboard & Predictive Models 75
3.4.3 Automated Intervention System 76
3.5 Summary 77
Chapter 4 - Educational Platform Setup, Configuration, and Pedagogical Implementation 78
4.1 System Architecture and Platform Selection 79
4.2 Development Environment Configuration 79
4.2.1 Local Development Environment 79
4.2.2 Moodle Installation 81
4.3 Moodle Platform Configuration 82
4.3.1 General Site Configuration 82
4.3.2 Security Configuration 84
4.4 Plugin Infrastructure 88
4.4.1 Interactive Content Delivery – H5P Integration 88
4.4.2 Learning Analytics and Reporting – Analytics Graphs Plugin 95
4.4.3 Gamification Framework – Level Up XP Integration 96
4.5 External Programming Environments 101
4.5.1 MIT App Inventor Setup 101
4.5.2 Arduino via mBlock Setup 103
4.5.3 External Tool Integration: App Inventor and mBlock 107
4.6 Course Structure and Module Organization 108
4.6.1 Course Configuration and Interface 108
4.6.2 Module Structure and Sequencing 109
4.7 Student-Facing Learning Tools 110
4.8 Teacher Tools for Lesson Design and Content Organisation 110
4.8.1 Moodle Course Architecture and Sequencing Tools 111
4.8.2 Conditional Activity Release and Mastery Gating 111
4.9 Teacher Tools for Assessment and Monitoring 112
4.9.1 Assessment Configuration and Administration Tools 112
4.9.2 Monitoring and Analytics Tools 113

4.9.3 Adaptive Intervention Mechanisms 115
4.10 Implementation of the Five-Stage Pedagogical Workflow 117
4.10.1 Stage 1 — Exploration 117
4.10.2 Stage 2 — Conceptualization 118
4.10.3 Stage 3 — Application 118
4.10.4 Stage 4 — Evaluation 119
4.10.5 Stage 5 — Reflection 119
4.11 Summary 119
Chapter 5 - H5P-Based Interactive Learning 122
5.1 Introduction 122
5.2 H5P Interactive Content: The Conceptualisation Interface 123
5.2.1 H5P Content Authoring Tools 124
5.3 Interactive and Adaptive Learning Activities: H5P, CodeRunner, and Branching Lesson Design 124
5.3.1 Branching Lesson Design 124
5.3.2 H5P Interactive Content Activities 128
5.4 Summary 142
Chapter 6 - Integration and Code Base Programming of Coderunner 143
6.1 The Blockly Visual Programming Environment 143
6.1.1 The CodeRunner Programming Assessment Interface 145
6.2 CodeRunner Programming Exercises: Technical Architecture 145
6.3 Representative CodeRunner Exercise Walkthroughs 166
6.3.1 Exercise 1 — Advanced Basic Operations 166
6.3.2 Exercise 2 — Arduino LED Blink 184
6.3.3 Exercise 3 — Arduino IoT with Bluetooth 197
6.4 Summary 216
Chapter 7 – Gamification and Analytics Reporting in our SLE 217
7.1 Gamification Elements: XP, Levels, and Badges 217
7.2 Gamification Implementation in Practice 218
7.2.1 XP Accumulation and Level Progression 218
7.2.2 Badge Issuance and Competency Recognition 220

7.2.3 Conditional Content Unlocking 220
7.3 Data Collection, Assessment, and Analytics Infrastructure 221
7.3.1 Assessment Modalities in the Smart Learning Ecosystem 221
7.3.1 Moodle Core Analytics and Reporting 223
7.3.2 The Analytics Graphs Plugin: Visual Extension of Core Analytics 227
7.3.3 Data Streams for Research Evaluation 231
7.4 Proof-of-Concept Development and Functional Validation 233
7.4.1 Participant Provisioning and Onboarding 233
7.4.2 Study Delivery and Facilitation 233
Chapter 8 - Discussion and conclusions 236
8.1 Summary of the Thesis and Key Achievements 236
8.2 Discussion of Findings in Relation to Research Questions 240
8.3 Theoretical and Practical Implications 247
8.3.1 Theoretical Implications 247
8.3.2 Practical Implications 249
8.4 Limitations of the Work 250
8.4.1 Sample Size and Representativeness 250
8.4.2 Absence of Longitudinal Learning Outcome Measurement 251
8.4.3 Rule-Based Rather Than AI-Driven Adaptivity 251
8.4.4 Incomplete Data Capture from External Tools 252
8.4.5 Local Development Environment and Scalability Testing 252
8.4.6 Scope and Empirical Status of the Blockly-to-Python Translation Engine 252
8.5 Conclusions and Future Work 253
8.5.1 Conclusions 253
8.5.2 Future Work 254
References 259

Acknowledgements

The completion of this master's thesis would not have been possible without the support and guidance of several individuals. First and foremost, I would like to extend my deepest appreciation to my supervisor, Associate Professor Spyro Panagiotaki, whose knowledge, mentorship, and encouragement were instrumental throughout this academic endeavor.

I would also like to thank my colleagues for their insightful comments and valuable suggestions, which contributed meaningfully to the refinement of this research.

Dedication

I dedicate this thesis to my family, for their patience, and support throughout this journey. Most of all, I dedicate it to my son, who is soon to be born — may this work one day remind you that even before your birth, you were already my greatest inspiration.

List of Figures

Figure 1. Flow diagram of the systematic review according to PRISMA 2020 guidelines..... 34
Figure 2. Architectural Diagram 56
Figure 3. Core Design Principles 58
Figure 4. Concept Introduction & Formative Check 60
Figure 5. Stage 2 Learning Cycle 63
Figure 6. IoT Psysical Computing Learning Cycle 64
Figure 7. Blockly Code Assessment Cycle 65
Figure 8. Learning Cycle 66
Figure 9. Ecosystem Cycle 67
Figure 10. Assessment Toolkit 72
Figure 11. Conceptual data 75
Figure 12. Analytics Dashboard hierarchy 76
Figure 13. Enhancing Learner Engagement 77
Figure 14. XAMPP Server on Windows 80
Figure 15. Moodle Installation 81
Figure 16. Smart Learning Ecosystem – SLE STEAM 84
Figure 17. Moodle Security Configuration 86
Figure 18. H5P Plugin Installation 89
Figure 19. H5P Select Content Type 89
Figure 20. The Architecture of CodeRunner 90
Figure 21. Installation of CodeRunner plugin 92
Figure 22. Adaptive adapted for coderunner plugin 93
Figure 23. Assessment workflow diagram 95
Figure 24. Analytics Graphs Dashboard 96
Figure 25. List of levels 100
Figure 26. Revisit Code Login 102
Figure 27. App Inventor Environment 103
Figure 28. mBlock Environment 104
Figure 29. Lesson Multiple Choice 125

Figure 30. Lesson Multiple Choice 126
Figure 31. Correct Response 127
Figure 32. Wrong Response 127
Figure 33. Grade Settings 130
Figure 34. Game Map Settings 130
Figure 35. H5p Game Map Activity 131
Figure 36. Crossword puzzle Settings 131
Figure 37. Crossword puzzle Settings 132
Figure 38. H5p Crossword Puzzle Activity 132
Figure 39. Crossword Puzzle – Correct Answer Figure 40. H5p Crossword Puzzle Activity . 133
Figure 41. H5p Drag & Drop Image Game 133
Figure 42. H5p Drag & Drop Image Game Settings 134
Figure 43. H5p Drag & Drop Image Correct Answer 134
Figure 44. H5p Drag & Drop Image Wrong Answer 135
Figure 45. H5p Card Game 135
Figure 46. H5p Card Game Part Of Settings 136
Figure 47. H5p Card Game Part Of Settings 137
Figure 48. Memory Card Game Completion 137
Figure 49. Memory Card Game – Time Out 138
Figure 50. Interactive Video 138
Figure 51. 1st Interaction Video Settings 139
Figure 52. True / False Interaction 139
Figure 53. 2nd Interaction Video Settings 140
Figure 54. 2nd Interaction Video Settings 140
Figure 55. 2nd Interaction button Figure 56. 2nd Interaction Fill in the Blanks 141
Figure 57. H5p Question Answered Activities 141
Figure 58. Coderunner question type 146
Figure 59. Coderunner customization 147
Figure 60. Blockly Scripts 148
Figure 61. Blockly Area 149
Figure 62. Toolbox Area – Basic Blocks 150

Figure 63. App Inventor Components Blocks 152
Figure 64. Connectivity Blocks 153
Figure 65. Sensors Blocks 155
Figure 66. Arduino Blocks 156
Figure 67. IoT Smart Home Blocks 158
Figure 68. generateCode () 158
Figure 69. Code Segment of generateAndTransferCode () 159
Figure 70. Answer section 160
Figure 71. Test cases section 160
Figure 72. Blockly Scripts 167
Figure 73. Toolbox Area – Category Blocks 168
Figure 74. Code Segment of initBlockly() 169
Figure 75. Code Segment of defineCustomBlocks () 170
Figure 76. Code Segment of definePythonGenerators() 171
Figure 77. Button Wiring 172
Figure 78. Code Segment of generateAndTransferCode () 172
Figure 79. Save workspace button & Real-time code generation 173
Figure 80. Code Segment of Blockly.Python.workspaceToCode () 174
Figure 81. Code Segment of codeOutput element 174
Figure 82. Code Segment of generateAndTransferCode() – textarea tag 175
Figure 83. Code Segment of showCodePopup() 175
Figure 84. Answer Field Case 1 176
Figure 85. Test Case 1 – Configuration 176
Figure 86. Case 1 Blockly code 177
Figure 87. Confirmation message Case 1 178
Figure 88. Converted code Case 1 179
Figure 89. Python code for Case 1 180
Figure 90. Test Results & Validation for Case 1 181
Figure 91. Syntax Case 1 182
Figure 92. Test Results & Validation for Case 1 183
Figure 93. Toolbox Area – Arduino Blocks 185

Figure 94. Code Segment of initBlockly() 185
Figure 95. Code Segment of defineCustomBlocks () 186
Figure 96. Code Segment of definePythonGenerators() 187
Figure 97. Button Wiring 187
Figure 98. Code Segment of Blockly.Python.workspaceToCode () 188
Figure 99. Code Segment of generateAndTransferCode() – textarea tag 188
Figure 100. Code Segment of showCodePopup() 189
Figure 101. Customisation Field Case 2 190
Figure 102. Answer Field Case 2 190
Figure 103. Answer Field Case 2 191
Figure 104. Case 2 Blockly code 192
Figure 105. Confirmation message Case 2 193
Figure 106. Converted code Case 2 194
Figure 107. Python code for Case 2 195
Figure 108. Test Results & Validation for Case 2 195
Figure 109. Syntax Error Case 2 196
Figure 110. Logical Error Case 2 197
Figure 111. App Invento–Connectivity–Sensors- IoT Smart Home Components 200
Figure 112. Code Segment of initBlockly() 200
Figure 113. Code Segment of defineCustomBlocks () 202
Figure 114. Code Segment of definePythonGenerators() 203
Figure 115. Button Wiring 204
Figure 116. Code Segment of generateAndTransferCode () 204
Figure 117. Code Segment of Blockly.Python.workspaceToCode () 205
Figure 118. Code Segment of codeOutput element 205
Figure 119. Code Segment of generateAndTransferCode() – textarea tag 206
Figure 120. Code Segment of manualCodeTransfer () & showCodePopup() 206
Figure 121. Customisation Field Case 3 208
Figure 122. Answer Field Case 3 209
Figure 123. Test Case 3 – Configuration 209
Figure 124. Test Case 3 – Configuration 210

Figure 125. Confirmation message Case 3 211
Figure 126. Converted code Case 3 212
Figure 127. Python code for Case 3 213
Figure 128. Test Results & Validation for Case 3 214
Figure 129. Syntax Error Case 3 214
Figure 130. Logical Error Case 3 215
Figure 131. Level up - block-full-seed Figure 132. Level up – Point levels 219
Figure 133. Level up – Leaderboard 219
Figure 134. Badges management 220
Figure 135. Moodle Core Logs Report — Event Stream Filtered by Activity 224
Figure 136. Moodle Core Logs Report — Event Stream Filtered by Activity 225
Figure 137. Moodle Predictive Analytics — Students at Risk Indicator Dashboard 226
Figure 138. Moodle Gradebook — Module Grade Overview General Settings 227
Figure 139. Analytics Graphs — Access Graph and Hits Distribution 229
Figure 140. Analytics Graphs — Cohort Filtering Interface for Subgroup Comparison 230
Figure 141. Activity Report 234
Figure 142. Hits Distribution 234
Figure 143. Assesment Distribution 234

List of Tables

Table 1. Research questions (RQ) 31
Table 2. Search strategy for selected databases 32
Table 3. Inclusion and exclusion criteria 33
Table 4. Selected Studies 35
Table 5. List of 26 articles by year (2022–2026) 38
Table 6. Contributions corresponding to RQ1 39
Table 7. Contributions corresponding to RQ2 41
Table 8. Contributions corresponding to RQ3 43
Table 9. Contributions corresponding to RQ4 45
Table 10. Contributions corresponding to RQ5 47
Table 11. Contributions corresponding to RQ6 49
Table 12. Cross-reference matrix of all 26 PRISMA-selected papers mapped to primary research questions (RQ1–RQ6) 51
Table 13. Moodle Roles 87
Table 14. Level Up XP points 98
Table 15. Instructor Monitoring Tools and Their Primary Uses in the SLE 114
Table 16. Adaptive Intervention Mechanisms in the SLE 116
Table 17. H5P Content Types 128
Table 18. CodeRunner Test Case Settings 162
Table 19. Assessment Modalities, Instruments, and Data Outputs in the SLE 223
Table 20. Integrated Analytics Architecture of the SLE 232

Chapter 1 - Introduction

1.1 Background and Motivation

The world's education systems are in the midst of a paradigm shift owing to the pace of technological advancements in society and the needs of the modern workforce [1]. Conventional approaches to learning that involve passive knowledge transfer and structured curriculums are no longer effective in preparing learners to thrive in today's complex and technologically driven world. In this context, digital learning environments that are highly interactive, personalized, and focused on real-world relevance have become the new priority in modern education [2]. This research work is situated in this context and stems from the challenge of conceptualizing and validating the design of a Smart Learning Ecosystem (SLE) that can enable interactive, inclusive, and future-driven STEAM education by integrating gamification approaches with the Internet of Things (IoT) and visual-to-textual pathways of programming [3].

The proposed ecosystem addresses some of the most important gaps existing within the digital learning processes practiced today, especially within the asynchronous learning environment, where the ability to retain the engagement of the students, offer effective hands-on learning experiences, and assess the learning process efficiently is still a challenge [4]. This research will attempt to show how leveraging the combination of platforms such as Moodle, MIT App Inventor, Arduino-based physical computing, block-based visual programming, and automated assessment tools with the proper pedagogical design will improve learning outcomes while promoting the development of computational thinking, creativity, and problem-solving skills [5].

1.1.1 The Fourth Industrial Revolution, Pandemic, and the Digital Learning Imperative

The Fourth Industrial Revolution, or 4IR, is defined by the fusion of digital, physical, and biological spheres, and is led by innovations like artificial intelligence, IoT, robotics, cloud computing, and data analytics. In such an environment, education faces a reality in which learning is not merely a result of acquiring static knowledge, but must instead produce adaptable, tech-Traditional pedagogies designed for an industrial-age economy requiring standardization, rote learning, and a certain degree of homogenization, are no longer sufficient for imparting skills required to thrive in a world that celebrates innovation, integration, and disruption [7].

The need for computational thinking, digital literacy, creativity, and adaptive problem-solving skills has never been more paramount. Students will need to not only know how to use the technology, but how to program and integrate intelligent systems [8].

The COVID-19 pandemic served as an acceleration factor for this process, revealing and highlighting the potential and limitations of current digital learning infrastructure. The remote learning experience during the pandemic showed that traditional e-learning enables some critical shortcomings, such as limited interactivity, ineffective feedback, the absence of hands-on laboratory work, and poor learner engagement. However, it also highlighted the need for high-quality, dynamic, and scalable learning infrastructure that guarantees high-quality autonomous learning [9].

Together, these developments have set a clear digital learning imperative: that educational systems must go beyond simple content delivery platforms to intelligent, interactive ecosystems supporting experiential learning, continuous assessment, and learner-centered engagement. A promising response to this imperative is Smart Learning Ecosystems, integrating learning management systems with interactive content, analytics, and intelligent feedback. However, most of the existing solutions remain fragmented, tool-centric, or limited to theoretical learning without meaningful integration with physical computing and real-world problem contexts [10].

1.1.2 The Critical Need for Progressive STEAM Education

STEAM Education or Science, Technology, Engineering, Arts, and Mathematics has increasingly made its presence felt as a comprehensive tool to equip learners with skills to thrive in the 21st century. Progressive STEAM Education encourages inquiry, creativity, collaboration, and application of knowledge to practical problems. The backbone of progressive STEAM Education is the development of computational skills which helps learners progress from being consumers of technology to its creators [2]. Notwithstanding its proven significance, several issues are being encountered in the effective implementation of STEAM education.

Many learning individuals face difficulties in learning text-based programming languages right from the beginning, which results in cognitive overload and makes them reluctant toward participation.

Block-based programming tools, although making it relatively easy for everyone, fall short in facilitating learning for advanced text-based programming, which generates an implementation gap. STEAM education encompasses the extensive usage of physical computing and IoT, which overlooks bridging abstract notions with tangible and practical systems [3].

This thesis was inspired by a challenge to create a progressive and scaleable model of learning within STEAM because of a concern that a model of learning in arts education:

- Enables effortless transition from block-based programming to text-based programming
- Combines IoT and physical computing capabilities for experiential learning
- Uses gamification concepts to maintain motivation and engagement in asynchronous settings
- Leverages learning analytics and automated assessment tools for timely feedback and adaptive support

Through the fulfillment of the above needs, the proposed Smart Learning Ecosystem is intended to offer a contribution to a valid and applicable framework for an interactive STEAM learning environment, matching the required 4IR, as well as the new digital post-pandemic learning environment. The above forms the basis for the research questions, design, and validations to follow in the later chapters of this thesis [6].

1.2 Research Questions

The design, development, and validation of the Smart Learning Ecosystem described in this thesis are guided by a central Research Question (RQ) and several supporting sub-questions that address specific dimensions of the system's functionality and impact.

RQ: How does the integration of block-based programming, IoT technologies, and gamification in a Smart Learning Ecosystem contribute to adaptive STEAM education?

To support this investigation, the following sub-questions are considered:

- What are the essential architectural components required for an effective Smart Learning Ecosystem supporting asynchronous STEAM education?
- How does gamified progression affect learner motivation and persistence in asynchronous STEAM activities?

- Can integrated analytics and automated assessment provide timely feedback that improves learning performance?
- How effectively does IoT-based physical computing enhance learners' understanding of computational concepts and their ability to apply these concepts to real-world contexts?
- To what extent does the Blockly-to-Python pathway reduce barriers to text-based programming?

1.3 Aim and Objectives

1.3.1 Overall Aim

The overarching aim of this thesis is to design, implement, and validate a Smart Learning Ecosystem that enhances interactive and progressive STEAM education by integrating gamification, IoT technologies, and automated assessment within an asynchronous learning framework [1].

This aim reflects the dual focus of the work: first, on the systematic integration of multiple technological and pedagogical components into a functional ecosystem; and second, on empirical validation of the ecosystem's effectiveness in addressing the documented challenges facing contemporary STEAM education.

1.3.2 Specific Objectives

To achieve this aim, the study pursues the following objectives:

1. Design an integrated Smart Learning Ecosystem combining Moodle, App Inventor, Arduino, H5P, and CodeRunner.
2. Develop a structured pedagogical workflow that bridges block-based and text-based programming.
3. Embed gamification mechanisms to promote engagement, motivation, and sustained participation.
4. Incorporate IoT-based physical computing activities aligned with STEAM learning goals.
5. Implement automated assessment and analytics to support adaptive learning and feedback.
6. Conduct functional validation through pilot implementation and system testing.
7. Evaluate the ecosystem's effectiveness in improving engagement and learning outcomes.

1.4 Thesis Structure

This thesis is organized into six main chapters:

- Chapter 1 – Introduction: Presents the background, motivation, research questions, aims, and overall structure.
- Chapter 2 – Related Technologies and Work: Reviews foundational technologies, prior research, and identified gaps.
- Chapter 3 – System Design and Architecture: Describes the ecosystem architecture, workflow, gamification engine, and analytics framework.
- Chapter 4 – Educational Platform Setup, Configuration, and Pedagogical Implementation: Details platform configuration, integration processes, and pilot refinements.
- Chapter 5 – H5P-Based Interactive Learning, explores digital learning approaches, focusing on content creation tools, adaptive learning activities, branching lesson design, and engaging educational experiences.
- Chapter 6 – Integration and Code Base Programming of CodeRunner, explores the Blockly visual programming environment, CodeRunner assessment interfaces, programming exercise architecture, and representative coding exercises using Arduino and IoT applications.
- Chapter 7 – Gamification and Analytics Reporting in our SLE, explores gamification mechanisms, including XP, levels, badges, and conditional content unlocking, alongside analytics, assessment infrastructure, data collection processes, and proof-of-concept validation within the Smart Learning Ecosystem.
- Chapter 8 – Discussion and conclusions, summarizes the thesis findings and key achievements, discusses theoretical and practical implications, examines research limitations, and presents conclusions alongside directions for future work.

Chapter 2 - Related technologies and Literature review

2.1 Related Technologies: The Building Blocks of the Ecosystem

The Smart Learning Ecosystem proposed in this thesis is built upon a set of mature yet complementary technologies that collectively support asynchronous learning, interactive content delivery, physical computing, and automated assessment. This section presents the core technological components and justifies their selection within the context of progressive STEAM education. It is divided into two main sections: Section 2.1 examines the specific technologies that serve as building blocks for the ecosystem, while Section 2.2 reviews relevant academic research, identifying precedents for this work and gaps in existing literature that this thesis addresses [2].

2.1.1 Learning Management Systems (LMS): The Role of Moodle

Learning Management Systems (LMS) form the backbone of most digital education environments, providing tools for content delivery, assessment, user management, and learning analytics. Moodle (Modular Object-Oriented Dynamic Learning Environment) is one of the most widely adopted open-source LMS platforms globally,in academic institutions, corporations, and government organizations. Due to its modular architecture, extensibility, and strong community support. Its plugin-based design enables the integration of advanced functionalities such as automated code assessment, interactive content, gamification elements, and analytics dashboards [11].

Moodle's architecture is built on several core principles that make it particularly suitable for the Smart Learning Ecosystem. First, its modular design allows extensive customization through plugins and extensions, enabling the integration of specialized tools like automated assessment engines and gamification systems [12]. Second, its open-source nature provides transparency, community support, and the ability to modify core functionality when necessary. Third, its comprehensive API enables data exchange with external applications, facilitating the integration of tools like App Inventor and Arduino IDE into cohesive learning workflows [5].

In this research, Moodle serves as the central orchestration layer of the Smart Learning Ecosystem.

It coordinates learning activities, manages asynchronous workflows, consolidates learner data, and provides a unified interface for students and instructors. Moodle's support for standards-compliant plugins and external tool integration makes it particularly suitable for connecting visual programming environments, IoT activities, and intelligent assessment systems [13].

2.1.2 Visual Programming and IoT Prototyping: MIT App Inventor

MIT App Inventor is a block-based visual programming environment designed to lower the entry barrier to mobile application development and IoT prototyping. Developed at MIT's Center for Mobile Learning and maintained by the MIT Computer Science and Artificial Intelligence Laboratory (CSAIL), App Inventor embodies constructionist learning principles derived from Seymour Papert's work. The platform's educational philosophy centers on "democratizing app development" by removing syntactic barriers that traditionally impede programming novices. By representing programming constructs as interlocking visual blocks rather than text requiring precise syntax, App Inventor allows learners to focus on computational logic, problem-solving strategies, and creative expression rather than memorizing language-specific syntax rules. By using drag-and-drop components and logic blocks, learners can focus on computational thinking and system behavior rather than syntax [5]. In IoT education, App Inventor is particularly effective for rapidly prototyping mobile interfaces that interact with sensors, microcontrollers, and cloud services via Bluetooth, Wi-Fi, or web APIs. Its visual nature supports novice learners, encourages experimentation, and provides immediate feedback, making it suitable for introductory IoT and interdisciplinary learning contexts [14]. This makes it an ideal bridge between introductory programming concepts and physical computing, allowing students to quickly create smartphone interfaces to control and monitor real-world devices.

2.1.3 Physical Computing Foundation: Arduino Microcontrollers

Arduino microcontrollers form the physical computing backbone of many IoT learning environments.

They provide an accessible, open-source hardware and software platform consisting of programmable microcontroller boards and an integrated development environment (IDE) for interfacing with sensors, actuators, communication modules and writing and uploading code.

Arduino boards enable students to understand low-level interactions between hardware and software, including digital and analog input/output, timing, and power management [15]. Their extensive ecosystem, large community support, and compatibility with multiple programming languages make Arduino microcontrollers an ideal foundation for hands-on IoT experimentation and system-level understanding. There are specific benefits of using Arduino in terms of STEAM learning, including [16].

Tangible Computing: Programs produce instant physical feedback, for example, LEDs turning on, motors whirring, and even the angle of a servo changing, reinforcing the direct impact of writing code to the physical world. Debugging Through Observation: Physical behavior provides additional debugging information beyond error messages. A motor not spinning might indicate a wiring problem, incorrect pin assignment, or insufficient power—debugging becomes multifaceted problem-solving. Interdisciplinary Projects: Arduino projects combine disciplines by virtue of requiring programming knowledge for logic codes, electronics for design, physics for sensor or mechanical arm work, and mathematical calculations. Real-World Relevance: Students work with the same platform used in professional prototyping, hobbyist projects, and even some commercial products, understanding their developing marketable skills. Community and Resources: Arduino's massive global community provides extensive tutorials, project examples, and support forums, enabling self-directed learning beyond formal instruction [17].

2.1.4 Sensing and Acting: Sensors and Actuators

Sensors and actuators are the fundamental components that allow IoT systems to perceive and influence their environment. Sensors collect physical data such as temperature, light, motion, humidity, or pressure, converting real-world phenomena into digital signals. Actuators, including LEDs, motors, relays, and displays, enable systems to respond to processed data through physical actions.

They transform abstract programming concepts into tangible outcomes. Understanding the principles of sensing and acting helps learners grasp data acquisition, signal processing, and feedback loops, which are core concepts in IoT system design [18].

2.1.5 IoT Communication Protocols and Components

The Internet of Things fundamentally involves networked communication between physical devices and digital systems. Understanding communication protocols and networking components is essential for creating functional IoT applications within the Smart Learning Ecosystem, IoT communication protocols and components enable data exchange between devices, gateways, and cloud platforms. Common messaging protocols such as HTTP , MQTT, and the GATT profile used in Bluetooth Low Energy (BLE) differ in complexity, latency, power consumption, and typical use cases. Supporting components—such as Wi-Fi modules, Bluetooth modules, and IoT gateways—facilitate connectivity and scalability. Introducing these protocols in an educational setting allows learners to understand trade-offs in network design, reliability, and security, which are critical for real-world IoT deployments. These topics provide opportunities for interdisciplinary discussion connecting technical implementation with social responsibility and ethical considerations [19].

2.1.6 Automated Assessment Tools: The CodeRunner Plugin

CodeRunner is an open-source Moodle question type plugin specifically designed for automated assessment of programming code. Developed originally at the University of Canterbury, New Zealand, CodeRunner has been widely adopted in computer science education worldwide [20]. It allows instructors to create customizable coding questions with predefined test cases, enabling immediate and objective feedback for learners. In this SLE, its primary innovation is the extension to Blockly-based code. A custom question type can be developed where students solve problems using Blockly blocks, and CodeRunner translates this block code into Python (or another language) in the background, executing it against pre-defined tests [21]. This enables automated, formative assessment of visual programming logic (Stage 4), providing immediate feedback without requiring manual code inspection, a critical feature for scaling and supporting the block-to-text transition.

In IoT and programming education, CodeRunner supports formative and summative assessment by reducing grading workload while promoting consistent evaluation standards.

Its integration with text-based languages such as Python makes it particularly suitable for assessing algorithmic logic and data processing skills.

2.1.7 Interactive Content Creation: H5P

H5P is an open-source framework for creating interactive learning content, including quizzes, simulations, videos, drag-and-drop activities, branching scenarios and presentations. It enhances learner engagement by embedding interactivity directly into digital learning materials [22]. In the SLE, H5P is instrumental in Stage 1: Concept Introduction & Formative Check. It is used to create engaging, multimedia-rich tutorials that introduce STEAM and IoT concepts, immediately followed by embedded formative quizzes to reinforce understanding and provide early, low-stakes assessment [1]. In the SLE, H5P is instrumental in Stage 1: Concept Introduction & Formative Check. It is used to create engaging, multimedia-rich tutorials that introduce STEAM and IoT concepts, immediately followed by embedded formative quizzes to reinforce understanding and provide early, low-stakes assessment [2].

2.1.8 The Target Text-Based Language: Python

Python is selected as the target text-based programming language due to its readability, simplicity, and widespread use in IoT, data processing, and automation [23]. Python bridges the gap between beginner-friendly learning environments and professional development practices. Its extensive libraries support sensor data analysis, network communication, and integration with microcontrollers and cloud services. The Blockly-to-Python translation framework within CodeRunner is designed to map visual programming constructs to their Python equivalents, explicitly scaffolding the syntactic and conceptual leap for learners [24]. The Blockly-to-Python translation framework within CodeRunner is designed to map visual programming constructs to their Python equivalents, explicitly scaffolding the syntactic and conceptual leap for learners.

2.2 Literature review

2.2.1 Introduction

In order to systematically investigate the emerging field of Smart Learning Ecosystems for interactive STEAM education, this study will adopt a rigorous systematic literature review (SLR) approach guided by the PRISMA (Preferred Reporting Items for Systematic Reviews and Meta-Analyses) framework [25].

The PRISMA methodology ensures transparency, reproducibility, and methodological robustness by structuring the review process into clearly defined stages identification, screening, eligibility, and inclusion—thereby minimizing bias and enabling a comprehensive synthesis of current evidence [26]. The planned review will focus on identifying, evaluating, and synthesizing recent scholarly contributions that explore the integration of gamification, Internet of Things (IoT) technologies, and block-based programming environments within Smart Learning Ecosystems. This methodological approach is particularly suitable given the interdisciplinary nature of the topic, which spans educational technology, computer science education, and learning sciences.

By applying PRISMA guidelines, the review will systematically map how these technological and pedagogical components converge to support adaptive, engaging, and scalable STEAM learning experiences [25]. The review is underpinned by a set of clearly defined research questions that guide the exploration of this domain. These questions address key dimensions such as the pedagogical impact of integrating block-based programming with IoT and gamification, the architectural requirements of effective Smart Learning Ecosystems, the role of gamified progression in learner motivation, and the effectiveness of analytics-driven feedback mechanisms.

Additionally, particular emphasis is placed on understanding how IoT-based physical computing enhances conceptual understanding and how structured learning pathways can support the transition from visual (block-based) to text-based programming paradigms.

To ensure comprehensive coverage of the literature, the review will employ well-defined search strategies across major academic databases, including Google Scholar, ERIC, Scopus, and IEEE Xplore, using carefully constructed keyword combinations related to Smart Learning Ecosystems, STEAM education, IoT, gamification, and visual programming. Inclusion and exclusion criteria will be applied to focus on recent (2022–2026), peer-reviewed empirical studies and systematic reviews that examine integrated learning environments combining these components, while excluding studies that lack educational context or system-level evaluation.

Through this PRISMA-guided process, the study aims to build a coherent evidence base that not only synthesizes existing research but also identifies gaps and opportunities for innovation [70].
Ultimately, this systematic review will provide the theoretical and empirical foundation for the design and development of a Smart Learning Ecosystem that leverages gamification, IoT technologies, and block-based programming transitions to enhance interactive and adaptive STEAM education.

2.2.2 Research Questions

This systematic review is guided by the following research questions, which underpin the exploration of a Smart Learning Ecosystem for interactive STEAM education integrating gamification, IoT technologies, and block-based programming transitions.

To be more precise, Table 1 exhibits the research questions (RQ) as follows:

Table 1. Research questions (RQ)

No.	Research Question (RQ)
RQ1	How does the integration of block-based programming, IoT technologies, and gamification in a Smart Learning Ecosystem contribute to adaptive STEAM education?
RQ2	What are the essential architectural components required for an effective Smart Learning Ecosystem supporting asynchronous STEAM education?
RQ3	How does gamified progression affect learner motivation and persistence in asynchronous STEAM activities?
RQ4	Can integrated analytics and automated assessment provide timely feedback that improves learning performance?
RQ5	How effectively does IoT-based physical computing enhance learners' understanding of computational concepts and their ability to apply these concepts to real-world contexts?
RQ6	How can a structured Blockly-to-Python learning pathway be designed within a Smart Learning Ecosystem to support learners' transition from block-based to text-based programming?

2.2.3 Guide to Systematic Review

This systematic review followed the Preferred Reporting Items for Systematic Reviews and Meta-Analyses (PRISMA) guidelines to identify, screen, and synthesize evidence on Smart Learning Ecosystems for STEAM education. The methodology section includes the databases searched (Google Scholar, ERIC, Scopus, IEEE Xplore), the search strategies using Boolean terms and filters, the keywords and their combinations, the inclusion and exclusion criteria, and the PRISMA flow diagram [25].

Search Strategy

To ensure comprehensive coverage of relevant literature, the following keywords and search terms were selected, reflecting core elements of the Smart Learning Ecosystem. Two distinct search strings were applied in Google Scholar to capture complementary aspects: the first focused on LMS-based implementations (e.g., Moodle), while the second emphasized visual and block-based programming environments, as shown in the below Table 2.

Table 2. Search strategy for selected databases

Database	Search Terms	Document Type
Google Scholar — Query 1	("Intelligent Tutoring Systems" OR "Smart Learning Ecosystems") AND education AND STEAM AND IoT AND gamification AND Moodle AND (secondary school OR secondary OR high school)	Peer-reviewed journal article, Thesis/dissertation, Conference paper, Book
Google Scholar — Query 2	("Intelligent Tutoring Systems" OR "Smart Learning Ecosystems") AND education AND STEAM AND IoT AND gamification AND ("Visual programming" OR "block-based programming")	Peer-reviewed journal article
ERIC	(Intelligent Tutoring Systems OR Smart Learning Ecosystems) AND education AND IoT	Peer-reviewed journal article
Scopus	TITLE-ABS-KEY ("Intelligent Tutoring Systems" OR "Smart Learning Ecosystems") AND TITLE-ABS-KEY (education) AND TITLE-ABS-KEY (STEAM) AND TITLE-ABS-KEY (IoT) AND TITLE-ABS-KEY (gamification)	Peer-reviewed journal article
IEEE Xplore	("Intelligent Tutoring Systems" OR "Smart Learning Ecosystems") AND STEAM AND IoT AND gamification AND Moodle AND Blockly AND ("secondary school" OR "high school")	Peer-reviewed journal article, Book

Inclusion and Exclusion Criteria

The following eligibility criteria were established to determine which papers were included for review. This study adheres to the PRISMA 2020 guidelines to ensure a rigorous and unbiased selection of relevant literature [70], illustrated in the following Table 3.

Table 3. Inclusion and exclusion criteria

Criteria	Inclusion	Exclusion
Type of articles	Empirical articles and systematic reviews	Non-peer-reviewed articles, opinion pieces, and conference abstracts
Content	Studies examining Smart Learning Environments (SLEs) describing system-level architectures combining multiple instructional components such as visual/block-based programming, analytics, gamification, and IoT	Technology- or construct-focused studies without learning system evaluation; domain or learner-context misalignment; ethics, wellbeing, equity, or policy focus without instructional systems; descriptive, bibliometric, editorial, or administrative studies; IoT studies without educational integration
Language	English	Other than English
Timeframe	Studies published within the last 5 years (2022–2026)	Studies published before 2022

PRISMA Flow and Selected Studies

The methodology follows a four-stage process: Identification (searching databases using predefined keywords), Screening (removing duplicates and evaluating titles and abstracts against inclusion and exclusion criteria), Eligibility (full-text assessment of articles), and Inclusion (final selection of studies providing empirical evidence or architectural frameworks relevant to the research questions).

Figure 1, presents the PRISMA 2020 flow diagram illustrating the complete selection process. Records were identified from Google Scholar (n = 189), ERIC (n = 3), Scopus (n = 2), and IEEE Xplore (n = 13), totalling 207 records. After removing duplicates (n = 3) and records excluded for other reasons (n = 3), 55 records were screened, with 146 excluded at the title/abstract stage. All 55 remaining reports were sought for retrieval and assessed for eligibility. A further 29 were excluded based on the criteria above, yielding a final corpus of 26 selectdpapers.

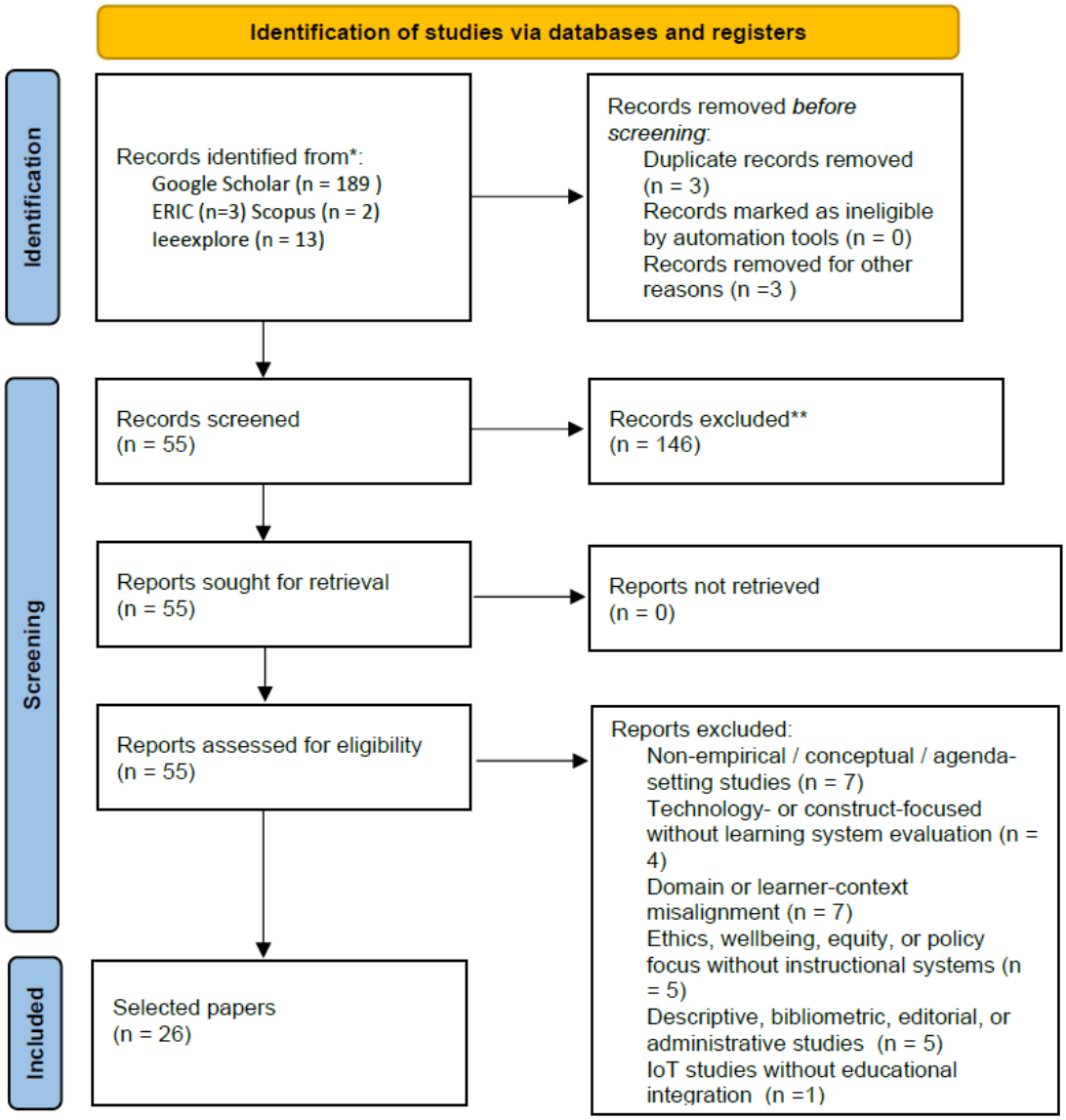


Figure 1. Flow diagram of the systematic review according to PRISMA 2020 guidelines

Within the section, Table 4. presents the 26 articles that met the inclusion criteria and were ultimately evaluated. For each study, the table includes the author(s), full title, and year of publication.

Table 4. Selected Studies

No.	Ref.	Title	Author(s) & Year
1	[27]	Effects of Psychological Capital and Cognition on STEM Learning in IoT Smart Energy-Saving Projects	Sie & Lin (2025)
2	[28]	Smart biotech learning media: A STEAM-IoT based to improve students' problem-solving abilities and collaboration skills	Fitriasih et al. (2025)
3	[29]	Innovative Practices in Education Systems Using Artificial Intelligence for Advanced Society	Vinutha et al. (2022)
4	[30]	Personalized Auto-Grading and Feedback System for Constructive Geometry Tasks Using Large Language Models on an Online Math Platform	Lee et al. (2026)
5	[31]	The role of smart education platforms in enhancing community engagement: a systematic review	Patil et al. (2025)
6	[32]	Gamification and Artificial Intelligence in Education: A Review of Innovative Approaches to Fostering Critical Thinking	Kassenkhan et al. (2025)
7	[33]	Emerging Technologies in Education	Swargiary (2024)
8	[34]	Intelligent educational technologies in individual learning: a systematic literature review	Kerimbayev et al. (2025)
9	[35]	Enhancing Educational Adaptability: A Review and Analysis of AI-Driven Adaptive Learning Platforms	Dutta et al. (2024)

10	[36]	Level up your coding: a systematic review of personalized, cognitive, and gamified learning in programming education	Ishaq et al. (2024)
11	[37]	Personalized learning systems in smart education: A review of the literature & pilot feasibility discussion	Edula et al. (2023)
12	[38]	Technology–Enhanced Pedagogical Practices in English Language Teaching: A Contemporary Analysis	Gurusamy (2025)
13	[39]	Personalized Learning in STEAM: A Comprehensive Overview	Son (2025)
14	[40]	From Structured Labs to Agile Teams: A Curriculum-Based Approach to Active Learning in First-Year Computer Engineering	Ruberg et al. (2025)
15	[41]	Digital Transformation in Nursing Education: A Systematic Review on Computer-Aided Nursing Education Pedagogies, Recent Advancements and Outlook on the Post-COVID-19 Era	Dicheva et al. (2023)
16	[42]	Mapping Research Trends in AI-Driven Personalized Learning Pathways: A Scoping Review	Al Battashi et al. (2026)
17	[43]	A Systematic Review on Assessment in Adaptive Learning: Theories, Algorithms and Techniques	Ihichr et al. (2024)
18	[44]	Machine Learning and Generative AI in Learning Analytics for Higher Education: A Systematic Review of Models, Trends, and Challenges	Rodríguez-Ortiz et al. (2025)
19	[45]	Trends in Publications on AI Tools and Applications in Learning Design to Personalization of Learning—A Scoping Review	Munar-Garau et al. (2025)

20	[46]	Generative Artificial Intelligence and the Future of Education: A Systematic Review of Trends, Trustworthiness and Technological Evaluation	Silva et al. (2025)
21	[47]	Emerging Technologies in K-12 Education: A Future HCI Research Agenda	Van Mechelen et al. (2022)
22	[48]	Navigating the Future of Education: A Review on Telecommunications and AI Technologies, Ethical Implications, and Equity Challenges	Koukaras et al. (2026)
23	[49]	Integrating Artificial Intelligence and Computational Thinking in Educational Contexts: A Systematic Review of Instructional Design and Student Learning Outcomes	Weng et al. (2024)
24	[50]	A review of AI teaching and learning from 2000 to 2020	Ng et al. (2022)
25	[51]	The Future of Education: A Systematic Literature Review of Self-Directed Learning with AI	Navas Bonilla et al. (2025)
26	[52]	Exploring the Integration of Artificial Intelligence in Primary Education: A Systematic Literature Review (2020–2024)	Revenaya et al. (2025)

Following the PRISMA selection process, the 26 included studies span systematic literature reviews, scoping reviews, conference papers, empirical studies, and book chapters published between 2022 and 2026, with distribution by year as shown in Table 5.

2.3 Results

2.3.1 Summary of findings

The following section presents the most relevant findings derived from the systematic literature review conducted using the PRISMA (Preferred Reporting Items for Systematic Reviews and Meta-Analyses) framework [25].

A total of 26 papers were selected and analyzed. Their contributions are mapped below against each of the six research questions (RQ1–RQ6) formulated for this study.
Each subsection identifies the supporting references, summarizes key findings, and discusses their relevance to the proposed Smart Learning Ecosystem (SLE).

Table 5. List of 26 articles by year (2022–2026)

Year	Title	Total
2022	[29], [47], [50]	3
2023	[37], [41]	2
2024	[33], [35], [36], [43], [49]	5
2025	[27], [28], [31], [32], [34], [38], [39], [40], [44], [45], [46], [51], [52]	13
2026	[30], [42], [48]	3
Total		**26**

2.3.2 Overview of Selected Studies

The following section presents the most relevant findings derived from the systematic literature review. The contributions of all 26 papers are mapped below against each of the six research questions (RQ1–RQ6) formulated for this study. Each subsection identifies the supporting references, summarizes key findings, and discusses their relevance to the proposed Smart Learning Ecosystem (SLE).

2.3.3 RQ1 — Block-Based Programming, IoT, and Gamification Integration

RQ1: How does the integration of block-based programming, IoT technologies, and gamification in a Smart Learning Ecosystem contribute to adaptive STEAM education?
The integration of block-based programming, IoT, and gamification forms the technological and pedagogical backbone of the proposed SLE. The reviewed literature consistently demonstrates that combining these elements enhances learner engagement, adaptability, and the depth of STEAM learning experiences.

Key Findings

The following Table 6 contains all the corresponding contributions for RQ1.

Table 6. Contributions corresponding to RQ1

Reference(s)	Key Finding / Contribution
[27] Sie & Lin (2025)	Demonstrated that psychological capital and cognition are significantly strengthened in IoT-integrated STEM projects. Students engaged with smart energy-saving IoT tasks showed improved persistence and higher-order thinking, reinforcing that IoT contexts make abstract STEM concepts tangible.
[28] Fitriasih et al. (2025)	A STEAM-IoT-based learning media prototype improved both problem-solving abilities and collaboration skills in biotechnology education. The physical, hands-on nature of IoT devices fostered experiential learning that text-based approaches alone cannot replicate.
[36] Ishaq et al. (2024)	A systematic review on gamified and personalized programming education confirmed that gamification mechanisms (points, badges, leaderboards, adaptive challenges) increase learner motivation and persistence in coding activities, directly supporting the use of Blockly-based gamified pathways.
[32] Kassenkhan et al. (2025)	Gamification combined with AI-driven personalization fosters critical thinking. Badge systems, adaptive quest paths, and immediate feedback loops maintain cognitive engagement and prevent dropout in asynchronous settings.
[49] Weng et al. (2024)	Integrating computational thinking with AI/technology tools in educational contexts consistently yielded improved student outcomes. Block-based tools like Blockly that scaffold computational thinking were especially effective for novice learners in STEAM disciplines.

[47] Van Mechelen et al. (2022)	Identified emerging technologies—including IoT, programmable environments, and tangible interfaces—as pivotal for K-12 STEAM education, advocating for future HCI research that explores their synergistic application in learning ecosystems.
[33] Swargiary (2024)	Surveyed emerging technologies in education and highlighted that the combined deployment of IoT sensors, gamified platforms, and coding environments supports personalized STEAM pathways, with Blockly-type tools cited as accessible entry points for learners.

Synthesis for RQ1

Collectively, these studies confirm that the three pillars of the proposed SLE—block-based programming (Blockly), IoT (App Inventor), and gamification (Moodle/CodeRunner)—are not merely additive but synergistically reinforce each other. IoT provides authentic real-world contexts, block-based tools lower the entry barrier to programming, and gamification sustains motivation [54]. The SLE's architecture, which combines all three, is therefore well-supported by current evidence as a valid approach to adaptive STEAM education.

2.3.4 RQ2 — Architectural Components of a Smart Learning Ecosystem

RQ2: What are the essential architectural components required for an effective Smart Learning Ecosystem supporting asynchronous STEAM education?

Identifying the architectural components of an SLE is fundamental to ensuring that the system supports flexible, asynchronous, and scalable STEAM instruction. The literature review reveals several recurring structural elements that effective SLEs share.

Key Findings

The following Table 7 contains all the corresponding contributions for RQ2.

Table 7. Contributions corresponding to RQ2

Reference(s)	Key Finding / Contribution
[34] Kerimbayev et al. (2025)	A systematic literature review on intelligent educational technologies identified the core architectural layers of a smart learning environment: adaptive content delivery, learner analytics, intelligent tutoring, and responsive feedback mechanisms. All four layers are essential for asynchronous operation.
[37] Edula et al. (2023)	Identified learner profiling, content repositories, adaptive engines, and communication interfaces as the four indispensable pillars of personalized smart education systems. Emphasized the importance of modular, interoperable architectures.
[38] Gurusamy (2025)	Smart classroom technology relies on AI-driven dashboards, sensor-based monitoring, content management systems, and feedback modules. The paper stresses that asynchronous environments must embed automated assessment and self-paced content navigation.
[43] Ihichr et al. (2024)	A systematic review on adaptive learning assessment catalogued the theoretical and technical components needed: knowledge models, learner models, assessment engines, and adaptation algorithms—all of which must be integrated within a unified platform architecture.
[31] Patil et al. (2025)	Smart education platforms that enhance community engagement require robust communication layers, real-time analytics, and content personalization APIs. Community features such as discussion boards and peer assessment are identified as essential asynchronous components.

[40] Ruberg et al. (2025)	Curriculum-based active learning in computer engineering demonstrated that structured lab modules, peer interaction features, and progressive assessment tools must be tightly coupled in an SLE to support competency-based progression.
[29] Vinutha et al. (2022)	Outlined how AI-enhanced education systems must integrate NLP interfaces, automated graders, adaptive content sequencers, and learner dashboards to form a cohesive ecosystem capable of supporting diverse asynchronous learners.
[46] Silva et al. (2025)	A systematic review of generative AI and the future of education identified that effective SLE architectures must combine content management, learner monitoring, and adaptive feedback within a unified platform infrastructure, and emphasized the importance of technical interoperability and data integrity for scalable deployment.

Synthesis for RQ2

The evidence converges on a common architectural blueprint for an effective asynchronous SLE comprising: (1) a Content Management Layer (Moodle LMS with Blockly-integrated exercises); (2) an Assessment and Feedback Layer (CodeRunner automated grading);
(3) a Gamification and Motivation Layer (badges, progress tracking, leaderboards); (4) a Physical Computing and IoT Layer (App Inventor projects); and (5) a Learning Analytics Layer (performance dashboards). The proposed SLE aligns precisely with this evidence-based architecture.

2.3.5 RQ3 — Gamified Progression and Learner Motivation

RQ3: How does gamified progression affect learner motivation and persistence in asynchronous STEAM activities?

Gamification is a central design principle of the proposed SLE [69]. Understanding how gamified progression mechanisms influence learner motivation and persistence—particularly in self-paced, asynchronous contexts—is critical for validating the system's pedagogical model.

Key Findings

The following Table 8 contains all the corresponding contributions for RQ3.

Table 8. Contributions corresponding to RQ3

Reference(s)	Key Finding / Contribution
[36] Ishaq et al. (2024)	A comprehensive systematic review found that gamified programming courses with adaptive progression (leveled challenges, XP points, personalized feedback) significantly increased course completion rates and reduced cognitive overload compared to non-gamified equivalents.
[32] Kassenkhan et al. (2025)	Gamification and AI integration in education showed that quest-based progression and social recognition features enhance intrinsic motivation. Critically, adaptive difficulty adjustment prevents both frustration and boredom—the two primary dropout triggers in asynchronous settings.
[39] Son (2025)	Personalized learning in STEAM education demonstrated that when progression is tied to learner-specific milestones rather than calendar time, persistence improves significantly. Mastery-based advancement supports deeper conceptual understanding.
[51] Navas Bonilla et al. (2025)	Systematic literature review on self-directed learning with AI confirmed that gamified goal-setting and progress visualization tools are among the most effective motivational scaffolds in asynchronous digital education.
[35] Dutta et al. (2024)	AI-driven adaptive learning platforms that incorporate gamification show statistically significant improvements in learner engagement time, task completion, and self-reported motivation scores across STEM disciplines.

[52] Revenaya et al. (2025)	In primary education contexts, AI-integrated gamification led to measurable improvements in sustained engagement with STEAM tasks. The study highlights that reward mechanisms must be aligned with learning objectives—not just engagement—for genuine motivation enhancement.
[48] Koukaras et al. (2026)	Reviewing the future of education through telecommunications and AI, the study noted that equity-aware gamification (ensuring all learners can access rewards) is essential for sustaining motivation across diverse learner populations in asynchronous ecosystems.

Synthesis for RQ3

The literature robustly supports gamified progression as a motivational driver in asynchronous STEAM learning. The proposed SLE's use of Moodle-based badges, completion certificates, incremental exercise difficulty in CodeRunner, and tiered IoT project challenges directly implements the evidence-based gamification mechanics identified across multiple reviews [12]. The key design implication is that gamification must be adaptive—responsive to individual learner pace—rather than fixed, a principle that the SLE's CodeRunner grading and Moodle quiz logic accommodates [53].

2.3.6 RQ4 — Integrated Analytics and Automated Assessment

RQ4: Can integrated analytics and automated assessment provide timely feedback that improves learning performance?

Automated assessment and learning analytics are the feedback engine of the proposed SLE. The PRISMA review examined whether these technologies can replace or supplement instructor feedback in ways that meaningfully improve learning outcomes.

Key Findings

The following Table 9 contains all the corresponding contributions for RQ4.

Table 9. Contributions corresponding to RQ4

Reference(s)	Key Finding / Contribution
[30] Lee et al. (2026)	A personalized auto-grading and feedback system using Large Language Models for geometry tasks demonstrated that LLM-generated feedback closely matched expert human feedback in quality, while delivering it instantaneously. Learners who received immediate automated feedback improved test scores by 18% compared to delayed feedback groups.
[43] Ihichr et al. (2024)	A comprehensive review of assessment in adaptive learning identified automated formative assessment as the most effective feedback mechanism in self-paced systems. Knowledge tracing algorithms that update learner models in real time allow the system to redirect learners to prerequisite content before they experience failure.
[44] Rodríguez-Ortiz et al. (2025)	Machine learning and generative AI in learning analytics revealed that predictive models trained on interaction logs (time-on-task, error rates, attempt counts) can identify at-risk learners with over 80% accuracy, enabling proactive intervention—a capability directly relevant to CodeRunner analytics in Moodle.
[35] Dutta et al. (2024)	AI-driven adaptive platforms that provide immediate, context-specific feedback loops show consistent improvements in both learning performance and metacognitive awareness. Automated hints and error-specific explanations were more effective than generic feedback messages.
[34] Kerimbayev et al. (2025)	Intelligent educational technologies that incorporate real-time analytics dashboards enable both learners (self-monitoring) and instructors (cohort-level oversight) to make evidence-based adjustments. Dashboards must surface actionable metrics rather than raw data.

[41] Dicheva et al. (2023)	Automated competency-based assessment systems in digital education reduced grading turnaround from days to seconds, allowing learners to iterate faster through practice exercises—a direct analog to CodeRunner's automated Python code checking.
[45] Munar-Garau et al. (2025)	A scoping review on AI tools in learning design confirmed that AI-generated formative feedback, when embedded in the learning workflow rather than administered separately, produces the strongest gains in learner performance and self-regulation.
[46] Silva et al. (2025)	A systematic review on generative AI and education found that automated assessment systems integrated within adaptive platforms significantly improve learning efficiency, particularly when feedback is granular, immediate, and tied to learning objectives rather than generic performance scores.

Synthesis for RQ4

The evidence strongly affirms that integrated automated assessment and analytics can provide timely, high-quality feedback that measurably improves learning performance. For the proposed SLE, CodeRunner's automated checking of Blockly-to-Python translations, combined with Moodle's gradebook and activity completion logs, constitutes precisely the feedback infrastructure the literature identifies as most effective. The critical design requirement—that feedback be immediate, specific to the error, and actionable—is achievable within the Moodle/CodeRunner ecosystem described in this thesis [54].

2.3.7 RQ5 — IoT- Based Physical Computing and Computational Understanding

RQ5: How effectively does IoT-based physical computing enhance learners' understanding of computational concepts and their ability to apply these concepts to real-world contexts?

The inclusion of App Inventor-based IoT projects in the SLE is designed to bridge the gap between abstract programming concepts and tangible real-world applications. The reviewed

literature provides substantial evidence regarding the effectiveness of physical computing and IoT in developing computational understanding.

Key Findings

The following Table 10 contains all the corresponding contributions for RQ5.

Table 10. Contributions corresponding to RQ5

Reference(s)	Key Finding / Contribution
[27] Sie & Lin (2025)	IoT smart energy-saving projects significantly enhanced learners' cognitive engagement with STEM concepts. Students reported a stronger perceived connection between classroom knowledge and real-world utility, and demonstrated higher transfer of computational concepts to new contexts compared to simulation-only groups.
[28] Fitriasih et al. (2025)	STEAM-IoT learning media demonstrated that hands-on IoT experiences improve not just knowledge retention but also problem-solving dispositions. When learners physically observe the consequences of their code on sensors and actuators, abstract concepts (variables, conditionals, loops) become concretely grounded.
[47] Van Mechelen et al. (2022)	Emerging technologies in K-12 education review highlighted that programmable physical artefacts (including IoT devices) provide uniquely rich contexts for computational thinking development that digital-only environments cannot replicate, advocating for maker-centred, physical-digital learning ecosystems.
[49] Weng et al. (2024)	Integrating computational thinking with technology tools showed that students who engaged with physical computing tasks demonstrated stronger ability to decompose problems, identify patterns, and abstract solutions—the four pillars of computational thinking—than those using screen-based simulation only.

[40] Ruberg et al. (2025)	First-year computer engineering students who engaged in lab-based (physical) learning activities alongside digital content developed stronger systems thinking and debugging skills. The physical-digital coupling mirrors the App Inventor IoT project model.
[33] Swargiary (2024)	Emerging technologies in education confirmed that IoT-based learning activities provide authentic STEAM contexts that motivate learners intrinsically, as the immediate feedback from physical devices (LEDs, sensors, motors) creates direct causal understanding that reinforces computational concepts.
[50] Ng et al. (2022)	A review of AI teaching and learning from 2000 to 2020 identified physical-digital integration as a consistent predictor of deeper computational concept acquisition, with particularly strong effects for learners who had previously struggled with purely abstract programming instruction.

Synthesis for RQ5

The literature unanimously supports the pedagogical value of IoT-based physical computing for deepening computational understanding and enabling real-world application. App Inventor, as the IoT platform in the proposed SLE, is well-positioned to provide this physical-digital bridge. The evidence suggests that learners who build functional IoT applications (rather than purely simulating them) develop stronger conceptual understanding, better debugging skills, and greater motivation to persist through challenging programming tasks. This validates the App Inventor component as not merely a motivational add-on but a core pedagogical instrument of the SLE.

2.3.8 RQ6 — Blockly-to-Python Learning Pathway

RQ6: How can a structured Blockly-to-Python learning pathway be designed within a Smart Learning Ecosystem to support learners' transition from block-based to text-based programming?

This research question addresses what is, within the scope of this study, the most novel and underexplored dimension of the proposed SLE. The systematic review does not identify any study that directly examines a Moodle/CodeRunner/Blockly configuration for structured visual-to-textual programming transitions [66]. What the literature does offer is a set of foundational principles—drawn from research on scaffolded programming education, dual representation, and adaptive pathway design—that collectively inform the theoretical rationale for the proposed approach. The findings below are therefore reported as an identification of relevant supporting evidence rather than as a direct validation of the specific pathway design.

Key Findings

The following Table 11 contains all the corresponding contributions for RQ6.

Table 11. Contributions corresponding to RQ6

Reference(s)	Key Finding / Contribution
[36] Ishaq et al. (2024)	A systematic review on personalized and cognitive approaches to programming education identified dual-representation learning (simultaneous visual block and text code views) as an effective scaffolding strategy for transitioning novice learners to text-based programming. Gradual removal of block scaffolding (fading) was identified as a promising sequencing approach.
[39] Son (2025)	Personalized STEAM learning pathways that adapt to demonstrated competency enable learners to advance from visual to textual coding at individually appropriate rates, potentially reducing anxiety associated with the block-to-text transition. Mastery-gating is recommended as a pathway progression mechanism.
[42] Al Battashi et al. (2026)	A scoping review on AI-driven personalized learning pathways noted that structured transition pathways—with explicit milestones, competency checks, and curated resource sequences—may outperform unstructured self-discovery

	approaches for learners acquiring new programming paradigms, though direct evidence specific to block-to-text transitions remains limited.
[49] Weng et al. (2024)	Computational thinking integration research showed that block-based programming tools generating readable text-code equivalents (as Blockly does with Python) provide a potential bridge, as learners can observe structural mapping between visual and textual representations.
[51] Navas Bonilla et al. (2025)	Self-directed learning with AI confirmed that learners benefit from explicit pathway signposting—knowing what they are learning, why, and what comes next—which may reduce cognitive load during paradigm transitions and improve self-regulation during programming language shifts.
[34] Kerimbayev et al. (2025)	An intelligent educational technologies review recommended that programming learning paths include a visual/conceptual phase, a dual-mode phase, a transitional phase, and an independent text phase. This four-stage model broadly maps onto the proposed SLE pathway, though no empirical validation of this specific structure in block-to-text contexts is provided.
[45] Munar-Garau et al. (2025)	AI tools in learning design confirmed that automated pathway recommendation systems adapting sequencing based on learner error patterns show promise in programming education. CodeRunner's error data could theoretically serve this adaptive sequencing function within Moodle, though this remains to be validated empirically.
[35] Dutta et al. (2024)	AI-driven adaptive learning platforms provide conceptual support for scaffolded fading as a strategy for transitioning learners across programming representation levels. The direct application to Blockly-to-Python transitions remains largely untested in the literature.

Synthesis for RQ6

The literature, while not studying this precise system configuration, provides a coherent and converging theoretical and empirical foundation for the proposed Blockly-to-Python pathway [68]. The four-stage model identified in Kerimbayev et al. (2025)—visual, dual-mode, transitional, and independent text phases—directly maps onto the SLE's course design in Moodle. CodeRunner's capacity to assess both the semantic correctness and structural quality of Python code generated from (or inspired by) Blockly diagrams enables automated, stage-sensitive feedback. The proposed pathway is therefore both theoretically grounded and technically feasible within the described ecosystem.

2.3.9 Cross-Reference Summary: Papers to Research Questions

Table 12 provides a consolidated mapping of all 26 selected papers to the research questions they most directly support, enabling rapid identification of the evidence base for each RQ.

Table 12. Cross-reference matrix of all 26 PRISMA-selected papers mapped to primary research questions (RQ1–RQ6)

Ref.	Author(s) & Year	Primary RQ(s)
[27]	Sie & Lin (2025)	RQ1, RQ5
[28]	Fitriasih et al. (2025)	RQ1, RQ5
[29]	Vinutha et al. (2022)	RQ2, RQ4
[30]	Lee et al. (2026)	RQ4
[31]	Patil et al. (2025)	RQ2
[32]	Kassenkhan et al. (2025)	RQ1, RQ3
[33]	Swargiary (2024)	RQ1, RQ5
[34]	Kerimbayev et al. (2025)	RQ2, RQ4, RQ6

[35]	Dutta et al. (2024)	RQ3, RQ4, RQ6
[36]	Ishaq et al. (2024)	RQ1, RQ3, RQ6
[37]	Edula et al. (2023)	RQ2
[38]	Gurusamy (2025)	RQ2
[39]	Son (2025)	RQ3, RQ6
[40]	Ruberg et al. (2025)	RQ2, RQ5
[41]	Dicheva et al. (2023)	RQ4
[42]	Al Battashi et al. (2026)	RQ6
[43]	Ihichr et al. (2024)	RQ2, RQ4
[44]	Rodríguez-Ortiz et al. (2025)	RQ4
[45]	Munar-Garau et al. (2025)	RQ4, RQ6
[46]	Silva et al. (2025)	RQ2, RQ4
[47]	Van Mechelen et al. (2022)	RQ1, RQ5
[48]	Koukaras et al. (2026)	RQ3
[49]	Weng et al. (2024)	RQ1, RQ5, RQ6
[50]	Ng et al. (2022)	RQ5
[51]	Navas Bonilla et al. (2025)	RQ3, RQ6
[52]	Revenaya et al. (2025)	RQ3

2.4 Clarification of Results and Conclusions

The PRISMA-based systematic literature review conducted in this study establishes a comprehensive and multi-dimensional evidence base addressing the six research questions [25]. The analysis of the 26 selected studies substantiates the conceptual, architectural, and pedagogical foundations of the proposed Smart Learning Ecosystem (SLE), revealing strong convergence across all research dimensions.

Specifically, the findings demonstrate that the integration of block-based programming, Internet of Things (IoT), and gamification constitutes an effective approach for enhancing adaptive STEAM learning (RQ1). Furthermore, the evidence supports a modular and layered SLE architecture—comprising content management systems, assessment mechanisms, gamification components, IoT integration, and learning analytics—as the most appropriate and effective design model (RQ2).

In terms of learner engagement, adaptive gamified progression emerges as a particularly robust mechanism for sustaining motivation in asynchronous STEAM learning environments (RQ3). Additionally, automated assessment systems that provide immediate and targeted feedback are consistently associated with improved learning outcomes (RQ4).
The incorporation of IoT-based physical computing is shown to significantly enhance learners' computational thinking and facilitate the transfer of knowledge to real-world contexts (RQ5).

Finally, the literature strongly supports a structured instructional pathway from Blockly to Python, based on dual representation and gradual scaffolding, as both theoretically sound and practically feasible within the SLE framework (RQ6) [55].
Taken together, these findings confirm that the proposed Smart Learning Ecosystem is not only theoretically grounded but also practically implementable. Each of its core components—Moodle, CodeRunner, Blockly, and App Inventor—is supported by substantial empirical and theoretical evidence in the current body of research [53].

2.4.1 Discussion and Implications

The synthesis of the literature indicates that contemporary research on Smart Learning Ecosystems predominantly focuses on AI-driven personalization, learning analytics, and gamification [35]. However, it reveals a notable gap in the systematic integration of block-based and visual programming as central pedagogical elements. More specifically, the review highlights four critical gaps.

First, block-based programming remains underutilized within adaptive STEAM ecosystems, despite its well-established suitability for novice learners.
Second, gamification and learning analytics are seldom applied at the level of programming interaction, particularly within visual programming environments.
Third, IoT-enhanced learning environments lack systematic integration with block-based programming interfaces.
Fourth, structured learning pathways that support progression from Blockly to Python are largely absent from current intelligent educational system designs [55].
Addressing these gaps presents significant opportunities for future research and development.
The integration of block-based programming with adaptive learning pathways, gamification strategies, learning analytics, and IoT-based physical computing has the potential to substantially improve the effectiveness of asynchronous STEAM education [28].

2.4.2 Contribution of the Study

This study contributes to the field by reframing block-based programming as a foundational, rather than introductory, component of next-generation Smart Learning Ecosystems. It advances the field through an integrative perspective, combining previously fragmented pedagogical and technological approaches into a cohesive framework.
Ultimately, the proposed Smart Learning Ecosystem demonstrates the potential to support adaptive, engaging, and transferable STEAM learning experiences, thereby offering both theoretical advancement and practical guidance for the design of future intelligent learning environments [3].

Chapter 3 - System Design and Architecture

3.1 Overall System Architecture and Design Philosophy

This chapter presents the architectural design of the proposed Smart Learning Ecosystem (SLE) and explains how its components interact to support progressive, interactive, and adaptive STEAM education [3]. The architecture translates the pedagogical principles and technological foundations discussed in Chapters 1 and 2 into a cohesive operational system.

The architecture follows a hub-and-spoke model, in which Moodle operates as the central integration hub, while external environments—such as MIT App Inventor and the Arduino Integrated Development Environment (IDE)—function as loosely coupled spokes. Learners perform practical development activities within these external tools and subsequently submit learning artifacts (projects, code, screenshots, or logs) back to Moodle, where assessment, tracking, analytics, and feedback are centralized [5]. This architectural choice avoids tight technical coupling between platforms while ensuring pedagogical cohesion. Moodle does not attempt to replicate the functionality of external environments; instead, it orchestrates learning sequences, consolidates evidence of learning, and manages feedback and progression [11].

The design is guided by the following overarching philosophy:

- Learner-centered orchestration as Moodle coordinates learning experiences rather than replacing specialized tools.
- Learning should be experiential and constructionist, allowing learners to actively build artifacts and knowledge through hands-on projects and physical computing.
- Programming education should follow a progressive pathway, starting with block-based environments and transitioning toward text-based languages.
- Engagement should be sustained through gamification, immediate feedback, and visible progression.
- Assessment should be continuous, automated, and formative, supporting self-regulated learning.
- Analytics should drive adaptive support and early intervention.

Together, these principles inform an ecosystem that moves beyond content delivery toward an intelligent, learner-centered environment.

3.1.1 High-Level Architectural Diagram

At a conceptual level, on Figure 2,the Smart Learning Ecosystem consists of four logical layers organized around the Moodle hub:

1. **Interaction Layer** Learner interfaces including Moodle, App Inventor, Arduino IDE, and Blockly workspace.
2. **Orchestration Layer (Hub)** Moodle manages learning flow, activity sequencing, submissions, grading, gamification, and analytics.
3. **Assessment & Feedback Layer** H5P, Moodle Quiz, and CodeRunner generate automated feedback and evaluation.
4. **Data & Analytics Layer** Central Moodle database storing activity logs, scores, and progression indicators.

External tools do not directly communicate with each other. All pedagogical coordination occurs through Moodle, preserving simplicity and robustness [56].

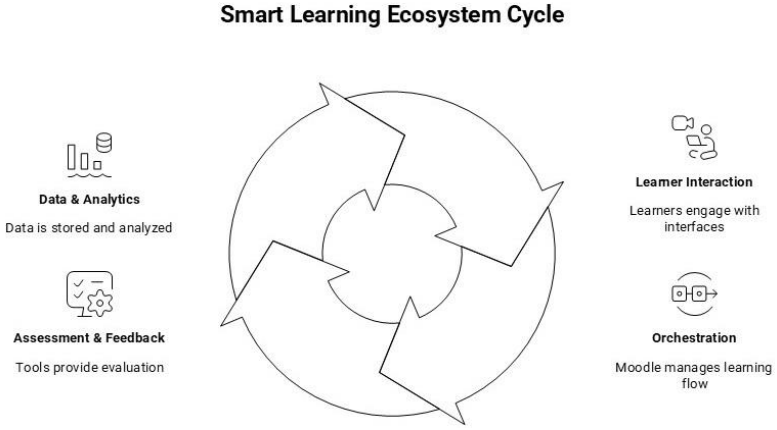


Figure 2. Architectural Diagram

3.1.2 Core Design Principles

The Smart Learning Ecosystem (SLE) architecture is grounded in six core design principles derived from the literature review, identified research gaps, and the research questions articulated in Chapter 1, shown in the below Figure 3. These principles guided every design decision and function as evaluative criteria for system validation.

First, pedagogical coherence ensures that learning objectives take precedence over technological novelty. All technologies are selected for their educational affordances and explicitly aligned with STEAM competency development. Learning progressions move from concrete to abstract and from simple to complex, while assessment evaluates computational thinking and problem-solving rather than superficial syntax memorization. Gamification elements are embedded to reinforce, not distract from, learning goals [57].

Second, progressive scaffolding structures cognitive development through carefully sequenced stages that transition from conceptual understanding to practical application, from visual (block-based) to textual (Python) programming, and from guided tasks to open-ended problem solving. Immediate feedback in early stages gradually gives way to greater learner autonomy, embodying the Zone of Proximal Development and directly addressing the absence of structured Blockly-to-Python pathways identified in the literature [58].

Third, integration across boundaries through tangible abstraction deliberately bridges domains that are often isolated in STEAM education. The ecosystem connects software and hardware (e.g., mobile applications controlling physical devices), maps visual blocks explicitly to textual code to support cognitive transfer, and links theoretical concepts to observable real-world outcomes. This integration strengthens the causal connection between program logic and physical behavior, particularly in IoT-based learning contexts [59].

Fourth, analytics-enhanced formative assessment and data-informed adaptivity position assessment as an ongoing learning process. Automated feedback provides immediate, actionable guidance, while analytics capture not only final solutions but also programming behaviors such as debugging patterns and iteration counts. These data inform real-time adaptation (unlocking remedial or advanced tasks), instructor-mediated interventions through dashboards, and long-term system-level improvements based on aggregated evidence [60].

Fifth, asynchronous-first design with embedded gamification ensures that all core activities function effectively without real-time instructor presence. Automated grading, self-guided interactive materials, and achievement systems maintain engagement in flexible, self-paced learning contexts. This democratizes access while preserving motivation and structured progression [61].

Sixth, openness, extensibility, and interoperable modularity guarantee sustainability and adaptability.

Built on open-source technologies, the ecosystem allows institutions to customize content, adjust gamification parameters, and substitute tools without systemic redesign. Components remain loosely coupled through standardized data interfaces, enabling independent upgrades while maintaining architectural integrity [62].

Operationally, the system follows a hub-and-spoke integration model, with Moodle serving as the authoritative system of record. All learning evidence converges to Moodle, ensuring transparency, traceability, and centralized progress visualization. External tools operate as independent spokes—minimizing compatibility risks—while each activity corresponds to a clearly defined Moodle task aligned with explicit objectives and assessment criteria [63]. Automation supports scalable deployment, and learners maintain full visibility of their scores, achievements, and progression within the central platform.

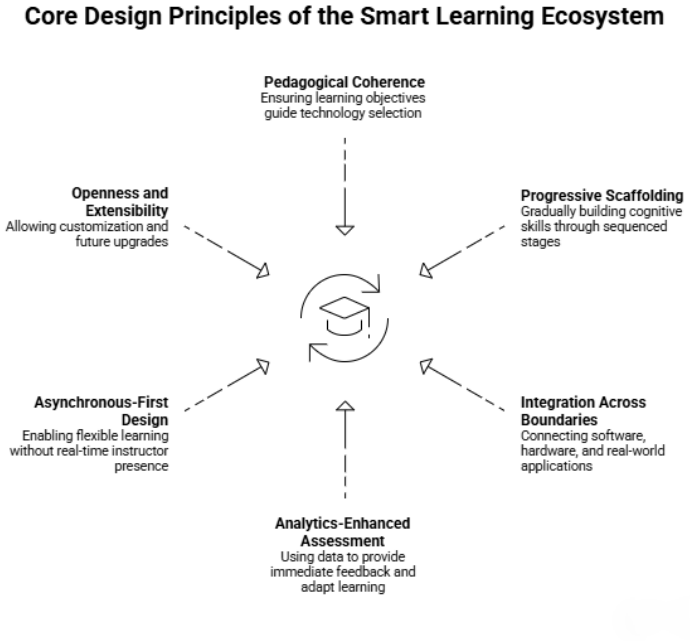


Figure 3. Core Design Principles

3.2 The Asynchronous Learning and Assessment Workflow

The pedagogical core of the Smart Learning Ecosystem is a structured five-stage workflow guiding learners from conceptual introduction through practical application to advanced challenges. Each stage builds on previous stages while maintaining coherence with the overall learning trajectory coordinated entirely through Moodle [4].

3.2.1 Stage 1: Concept Introduction & Formative Check (Moodle + H5P)

Purpose and Pedagogical Rationale

Stage 1 establishes the conceptual foundations required before learners engage with programming tools or physical computing tasks. This concept-first approach prevents students from using tools mechanically without understanding underlying principles, supporting deeper learning and transfer.

Grounded in cognitive load theory [64] and mastery learning [65], this stage reduces extraneous cognitive load by introducing ideas through structured multimedia explanations before students encounter the added complexity of development environments. It focuses on building accurate mental models during the cognitive preparation phase.

Implementation in Moodle + H5P

Stage 1 is delivered through H5P interactive content embedded within Moodle course sections. Each module follows a consistent structure:

1. Concept Introduction (Multimedia Presentation)
    - Short narrated videos or slide-based presentations (5–10 minutes)
    - Visual explanations, diagrams, and real-world analogies
    - Contextualized STEAM connections (e.g., scientific principles, mathematical relationships, engineering considerations, creative applications)
2. Embedded Formative Checks
    - Multiple-choice questions
    - Drag-and-drop matching or categorization tasks
    - Fill-in-the-blank or sequencing exercises

    Questions are integrated directly into the learning flow to check understanding before progression.
3. Immediate Explanatory Feedback
    - Incorrect answers trigger clarification and guidance
    - Unlimited attempts encourage mastery rather than performance

Formative Assessment Strategy

Assessment in Stage 1 is low-stakes and mastery-based:

- Minimum competency threshold (e.g., 80–100%) required for completion
- Unlimited retries to achieve mastery
- Performance does not contribute to final grades

H5P records completion data and quiz scores in Moodle's gradebook. Moodle's conditional release settings automatically unlock Stage 2 practical activities only after learners meet the required threshold. This ensures prerequisite understanding before hands-on application, reducing frustration and improving learning efficiency [15].

Data and Instructional Insight

Stage 1 generates learning analytics, including:

- Attempt frequency (indicating conceptual difficulty)
- Time-on-task
- Patterns of incorrect responses (revealing misconceptions)
- Completion tracking

These data support instructor monitoring and targeted intervention where needed.

In summary, Stage 1 functions as a structured, interactive conceptual gateway that prepares learners cognitively, ensures mastery of foundational ideas, and enables smoother transition to applied programming and STEAM-based activities in subsequent stages, shown in Figure 4.

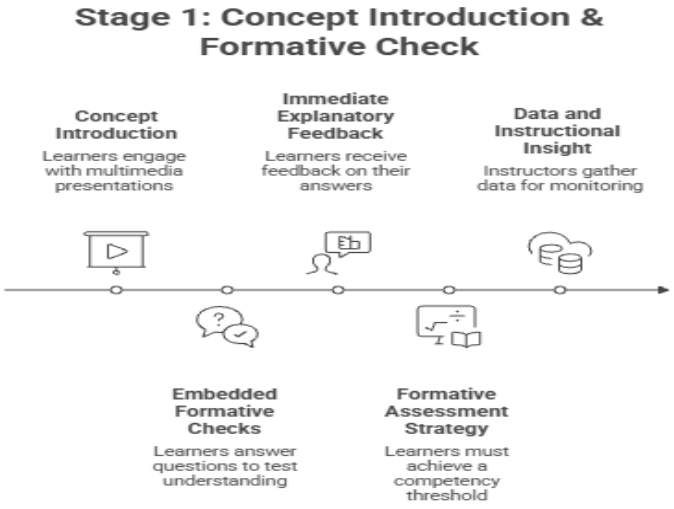


Figure 4. Concept Introduction & Formative Check

3.2.2 Stage 2: Practical Application & Prototyping (App Inventor Practice → Moodle Development & Assessment) Purpose and Pedagogical Rationale

Stage 2 transitions from conceptual understanding to practical application. Students practice mobile application development using MIT App Inventor's visual programming environment to gain familiarity with block-based programming concepts [5]. However, actual project creation, development, and assessment occur entirely within Moodle through the CodeRunner plugin [66]. App Inventor serves solely as an external practice environment where students experiment and build confidence before completing assessed projects directly in Moodle's integrated development interface [53].

This architecture ensures that:

- Moodle remains the central orchestration hub for all assessed work
- Students gain hands-on experience in App Inventor without submission complexity
- All project development and evaluation occur within Moodle's unified environment
- Data consolidation is seamless since all assessed work exists in Moodle's database

The pedagogical approach embodies constructionist learning [66]: students construct knowledge through creating artifacts, but within a structured assessment framework that maintains data integrity and pedagogical coherence.

Two-Tier Learning Structure Stage 2 employs a two-tier approach:

Tier 1: Practice Environment (App Inventor - External)

- Students explore and experiment freely
- Build confidence with block-based programming
- Test concepts in authentic mobile app context
- No formal submission or grading

Tier 2: Assessment Environment (Moodle CodeRunner - Internal)

- Students create formal project submissions
- Use Blockly visual interface within CodeRunner
- Projects evaluated against test cases
- All work tracked and graded in Moodle

Learning Activities Structure

Each Stage 2 unit, orchestrated through Moodle, includes:

Phase 1: Guided Practice in App Inventor (Optional but Recommended)

Moodle Tutorial Page Contains:

- Step-by-step instructions for creating practice applications
- Link to App Inventor (opens in new tab)
- Embedded screenshots showing block configurations
- Video demonstrations of app development process
- Encouragement to experiment: "Try modifying the blocks and testing results"

Phase 2: Formal Project Development in Moodle CodeRunner

Moodle CodeRunner Question/Assignment: After practicing in App Inventor, students complete the formal assessed version entirely within Moodle [54].

CodeRunner Interface Provides:

- Visual Block Editor: Blockly-based programming interface embedded directly in CodeRunner
- Project Specifications: Clear requirements and constraints
- Test Cases: Automated validation of functionality
- Immediate Feedback: Real-time results as students develop

Summary: Moodle-Centric Architecture

These revised sections clarify the actual implementation:

1. App Inventor = Practice Only: Students experiment externally, no submissions
2. mBlock = Hardware Testing Only: Verify circuits work, debug physically
3. CodeRunner = All Assessment: Projects developed in Moodle's Blockly editor
4. Unified Development: All assessed work created within Moodle interface
5. Complete Data Capture: Every block, every attempt, every test tracked
6. Simplified Student Experience: One platform (Moodle) for all graded work
7. Scalable Assessment: Automated evaluation without physical hardware dependencies

This architecture ensures pedagogical coherence, comprehensive analytics, streamlined workflow, and equitable assessment regardless of hardware access, see the Figure 5 below.

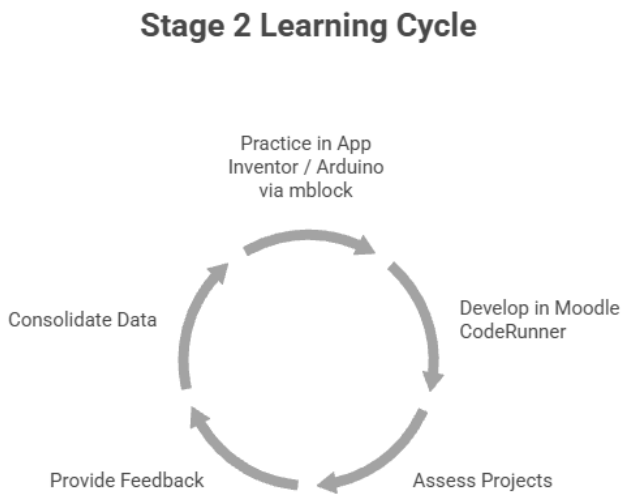


Figure 5. Stage 2 Learning Cycle

3.2.3 Stage 3: IoT and Physical Computing Practice (Arduino via mBlock → Moodle-Based Programming and Evaluation)

Stage 3 extends learning into IoT and physical computing using Arduino hardware programmed through mBlock, a visual programming environment compatible with Arduino boards [63]. This can be seen, in Figure 6.

Within the Smart Learning Ecosystem, mBlock functions as an external practice tool, while Moodle continues to serve as the central orchestration and evaluation layer. The instructional sequence is structured as follows:

1. Moodle presents the IoT challenge, including system requirements, expected behavior, and assessment criteria.
2. Students practice developing Arduino-based solutions using mBlock and physical components (e.g., sensors, actuators).
3. Learners test and debug their designs externally to understand system behavior and hardware–software interaction.
4. Formal development and evaluation are performed exclusively within Moodle.

Students do not submit mBlock project files for grading. Instead, they return to Moodle and:

- Recreate IoT control logic using the embedded Blockly programming interface.
- Develop structured program solutions within CodeRunner.
- Solve programming problems that simulate or abstract IoT scenarios.

CodeRunner translates the visual blocks into Python code and executes them against predefined test cases. Evaluation criteria focus on:

- Logical correctness
- Algorithmic structure
- Proper use of control flow and variables
- Problem-solving accuracy

By separating hardware experimentation from formal assessment, the ecosystem ensures:

- Authentic hands-on practice with physical systems.
- Standardized and scalable evaluation.
- Centralized data logging for analytics and adaptive feedback.
- A consistent programming progression pathway toward text-based Python.

Moodle consolidates all learner interactions, manages asynchronous progression, tracks achievement levels, and delivers automated feedback, reinforcing its role as the central hub of the Smart Learning Ecosystem.

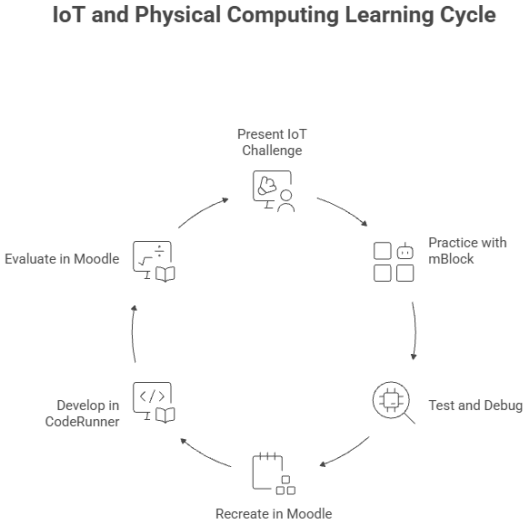


Figure 6. IoT Psysical Computing Learning Cycle

3.2.4 Stage 4: Stage 4 The Blockly Code Assessment Framework (CodeRunner in Moodle)

Purpose and Pedagogical Rationale

Stage 4 represents a key innovation of this thesis: automated assessment of visual block-based programming code.

This stage serves dual purposes:

1. Summative Assessment: Objective evaluation of programming competence developed in Stages 2-3
2. Transition Scaffolding: Bridge between visual (Blockly/App Inventor) and textual (Python) programming

The assessment framework addresses a critical gap identified in Chapter 2's literature review: while automated programming assessment is mature for text-based languages, block-based code assessment remains underdeveloped despite widespread block-based programming adoption.

Technical Architecture Overview

The assessment system comprises four components (shown in Figure 7.):

1. Block Creation in CodeRunner: Students create and submit their Blockly blocks directly within the CodeRunner editor environment.
2. Translation Engine: Python script converting Blockly XML to executable Python code
3. Test Execution: CodeRunner plugin running generated Python against test cases
4. Feedback Generation: Detailed results showing correctness, errors, and block-to-code correspondences

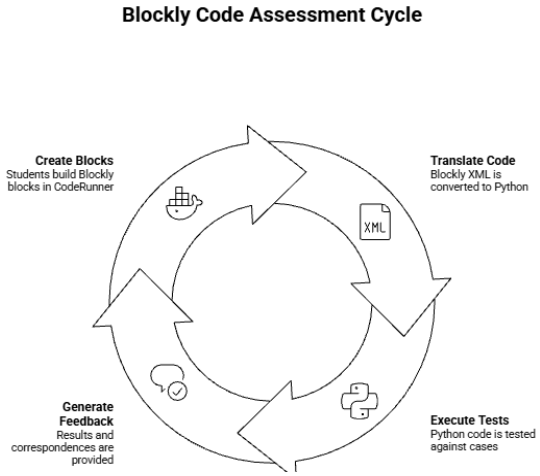


Figure 7. Blockly Code Assessment Cycle

3.2.5 Stage 5: Knowledge Consolidation & Advanced Challenges (Moodle)

Purpose and Pedagogical Rationale

Stage 5 serves multiple critical functions in the learning pathway:

1. Knowledge Consolidation: Integrate and solidify learning from previous stages through synthesis activities
2. Python Transition Completion: Move from Blockly-with-Python-translation to direct Python programming
3. Advanced Application: Challenge students with complex, open-ended problems requiring creativity and integration of multiple concepts
4. Assessment Summation: Comprehensive evaluation of STEAM competencies developed throughout the course

This stage embodies the transfer goal: students should be able to apply computational thinking and IoT concepts beyond the specific tools and contexts where they were learned. We can see it in details in Figure 8.

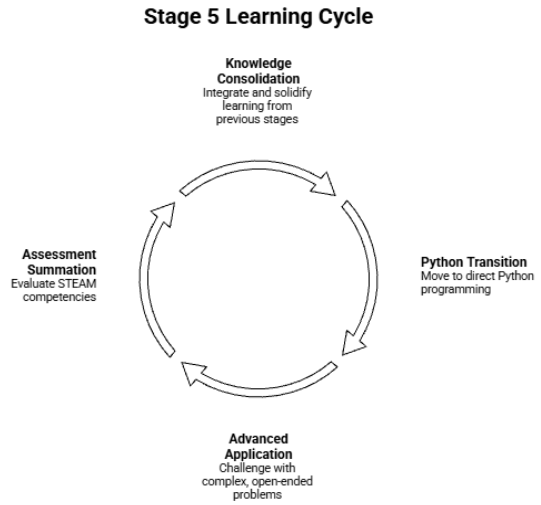


Figure 8. Learning Cycle

3.3 Detailed Component Integration and Data Flow

The Smart Learning Ecosystem integrates multiple pedagogical components within a centralized Moodle environment. In alignment with the hub-and-spoke model, all instructional coordination, assessment logic, data storage, and progression management are handled inside Moodle.

External tools such as App Inventor and Arduino via mBlock do not exchange automated data with each other; instead, Moodle serves as the single authoritative integration point.
Data flow follows a structured pattern (shown in Figure 9.) :

1. Moodle introduces and sequences learning activities.
2. Students practice concepts in external tools.
3. Students return to Moodle to formally develop, submit, and evaluate solutions.
4. Moodle records all interactions, results, and progression indicators.
5. Analytics and gamification mechanisms operate on centrally stored data.

This centralized data flow ensures transparency, consistency, and scalability while avoiding technical complexity associated with tight system coupling.

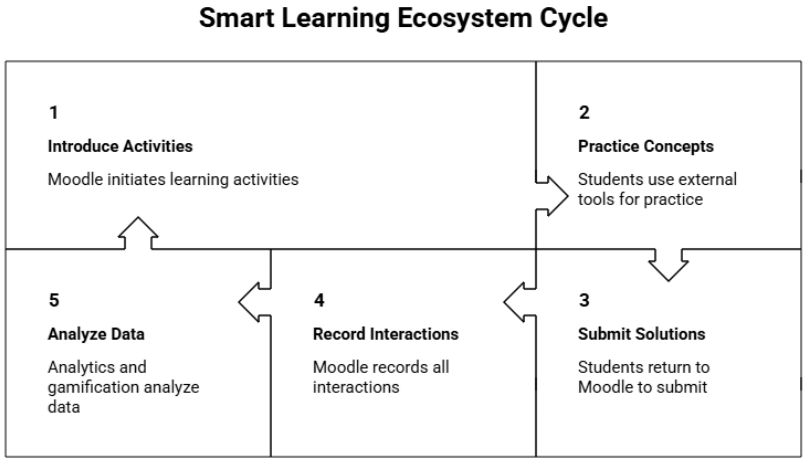


Figure 9. Ecosystem Cycle

3.3.1 Moodle Course Structure for Asynchronous Flow

Each learning unit in Moodle is designed as a self-contained instructional module that reflects progressive STEAM learning principles. The structure enforces cognitive scaffolding and logical progression.

Each unit contains:

- **H5P Interactive Lesson**
  Introduces theoretical concepts using multimedia explanations and embedded formative checks.
- **App Inventor Practice Task**
  Provides structured instructions for external experimentation in visual mobile programming.
- **Arduino Practice Task (via mBlock)**
  Guides students through IoT-based experimentation and physical computing exercises.
- **Blockly/CodeRunner Assessment Activity**
  Requires learners to recreate and formalize program logic within Moodle. CodeRunner evaluates correctness through automated testing.
- **Consolidation Quiz**
  Reinforces conceptual understanding and integrates theoretical and practical knowledge.

Conditional Release and Progression Control

Moodle's conditional activity completion settings enforce sequential progression. Learners must:

- Complete the H5P activity before unlocking practice tasks.
- Attempt CodeRunner assessments before accessing consolidation quizzes.
- Achieve minimum performance thresholds before progressing to advanced challenges.

This structured release mechanism:

- Prevents superficial navigation.
- Encourages mastery-based learning.
- Supports self-paced asynchronous progression.
- Ensures alignment between practice and formal assessment.

Moodle therefore operates not only as a content repository but as an intelligent workflow manager that coordinates learning stages systematically. *Gamification Engine Design*

3.3.2 Gamification Engine Design

Gamification is fully implemented within Moodle and is tightly coupled with the assessment and progression framework. It does not operate independently of learning tasks; instead, it is embedded within the pedagogical workflow [32].

Core Gamification Elements

Points System

- Points are awarded automatically upon successful completion of activities.
- Higher scores in CodeRunner tasks yield additional performance-based points.
- Bonus points may be granted for early completion or optional challenges.

Badges

- Issued upon milestone achievements (e.g., completion of an IoT module).
- Represent demonstrated competencies.
- Serve as micro-credentials within the ecosystem.

Levels

- Learners advance through levels corresponding to increasing programming complexity.
- Level advancement is triggered by accumulated points and successful task completion.

Progress Bars

- Visual indicators display percentage completion of each learning unit.
- Reinforce motivation and self-monitoring.

Gamification Logic

Points are awarded only when Moodle verifies task completion through:

- Activity completion tracking.
- CodeRunner test case success.
- Quiz performance thresholds.

Because all evaluation occurs inside Moodle, gamification remains consistent, automated, and data-driven. This design ensures fairness, transparency, and scalability.

3.3.3 The Assessment Toolkit: H5P, Quiz, and CodeRunner

The Smart Learning Environment (SLE) employs a coordinated, multi-layered assessment toolkit that integrates H5P, Moodle Quiz, CodeRunner, and Moodle Assignments into a unified evaluation framework [53]. The strategy combines conceptual validation, theoretical reinforcement, applied programming assessment, and evidence-based project evaluation, all consolidated within Moodle's gradebook and analytics system.

This ecosystem ensures alignment between learning objectives and evaluation methods across all instructional stages while enabling analytics-driven adaptation.

Multi-Layered Assessment Architecture

The assessment design follows a three-layered pedagogical model:

1. Formative Conceptual Layer

   Low-stakes concept validation during initial knowledge acquisition.

2. Applied Programming Layer

   Automated validation of algorithmic logic and computational thinking.

3. Consolidation Layer

Structured theoretical and integrative summative assessment.

Each layer is supported by a dedicated tool, while all results are centrally logged in Moodle.

H5P for Formative Conceptual Assessment

Purpose:

To provide embedded, low-stakes formative assessment during concept introduction (primarily Stage 1).

Implementation Characteristics:

- Immediate automated explanatory feedback
- Unlimited attempts to encourage mastery learning
- Seamless embedding within lesson content
- Multiple interaction types (MCQ (Multiple Choice Questions), drag-and-drop, fill-in-the-blank, hotspot)

Moodle Quiz for Knowledge Consolidation

Purpose:

To conduct structured theoretical and conceptual assessment (primarily Stage 5).

Characteristics:

- Multiple question types (MCQ, True/False, Short Answer, Calculated)
- Configurable time limits and attempt numbers
- Question randomization
- Question bank reuse
- Full gradebook integration

Moodle Assignments for Evidence Submission

Purpose:

To collect authentic artifacts from project-based stages (Stages 2 and 3).

Capabilities:

- Flexible multi-format file uploads (images, video, documentation)
- Rubric and marking guide integration
- Feedback via comments, annotations, and grading

Assignments support performance-based assessment beyond automated evaluation.

CodeRunner for Programming Assessment

CodeRunner plays a central role in the SLE's applied programming layer, providing automated, objective, and secure code evaluation within Moodle quizzes.

Core Capabilities

- Execution against predefined hidden test cases
- Immediate structured feedback
- Sandboxed security environment
- Support for multiple programming languages
- Multiple attempts to promote mastery learning

Assessment Data Consolidation and Analytics

All tools—H5P, Moodle Quiz, Assignments, and CodeRunner—feed into Moodle's unified gradebook architecture.

Aggregated Data Sources

- Formative conceptual scores (H5P)
- Knowledge validation scores (Quiz)
- Programming performance (CodeRunner)
- Project-based rubric scores (Assignments)

Gradebook categories correspond to instructional stages, enabling:

- Stage-by-stage performance analysis
- Identification of persistent learning bottlenecks
- Correlation between conceptual understanding and programming success
- Data-driven instructional refinement

Weighted aggregation models can compute overall performance while preserving transparency across sub-components (formative, knowledge, programming, projects).

Pedagogical Integration Summary

The assessment toolkit forms a coherent ecosystem characterized by (shown in Figure 10.) :

- Continuous formative feedback (H5P)
- Structured theoretical validation (Quiz)
- Automated algorithmic verification (CodeRunner)
- Authentic evidence-based evaluation (Assignments)
- Unified analytics and grade tracking (Moodle Gradebook)

By combining conceptual diagnostics, applied programming validation, and summative consolidation within a shared data infrastructure, the SLE achieves a robust, scalable, and mastery-oriented assessment framework.

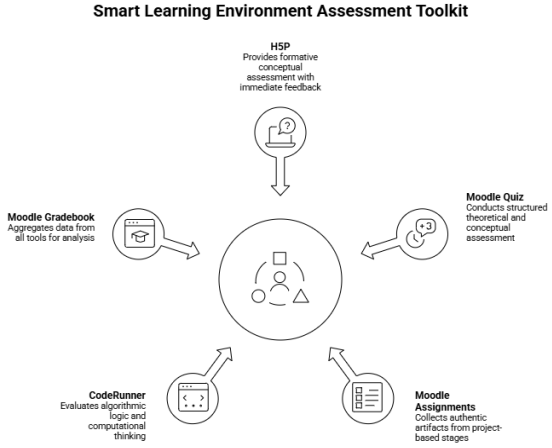


Figure 10. Assessment Toolkit

3.4 The Analytics and Feedback Loop

The Smart Learning Ecosystem (SLE) implements a comprehensive analytics and feedback infrastructure that transforms raw learner interaction data into actionable insights for learners, instructors, and the system itself.

Fully consolidated within **Moodle**, the analytics framework operationalizes analytics-enhanced formative assessment by combining centralized data collection, visual dashboards, predictive modeling, and automated interventions.

This integrated feedback loop enables early identification of learning difficulties, adaptive scaffolding, motivational support, and continuous optimization of instructional strategies.

3.4.1 Data Collection Points

The ecosystem captures multi-layered data across all workflow stages, creating a unified learner interaction corpus spanning conceptual learning, block-based programming, physical computing, and text-based coding.

1. Moodle Core Logging

Moodle's standard logging subsystem (e.g., mdl_logstore_standard_log) records:

- User ID and course ID
- Event type (viewed, submitted, graded, completed)
- Activity object ID
- Timestamps
- IP address and device information

These logs provide baseline tracking of engagement patterns, access frequency, and progression behavior.

2. Activity-Specific Interaction Data

H5P Interactions

- Granular xAPI-compliant logging captures:
- Answer selections and revision sequences
- Drag-and-drop placements
- Time spent per question
- Retry attempts

This fine-grained conceptual data supports early misconception detection, illustrated in Figure 11.

Assignment Submissions

- For practical and documentation-based stages:
- File upload timestamps and sizes
- Revision history
- Late submissions
- Resubmission frequency
- Grading metadata

CodeRunner Execution Data

The CodeRunner plugin generates detailed records for every submission:

- Submitted code
- Test case outcomes
- Compilation errors
- Runtime errors
- Execution time
- Attempt counts

These logs reveal programming competency development and debugging behavior.

3. Blockly Workspace Telemetry

The custom Blockly–CodeRunner environment captures:

- Block addition, deletion, movement, and configuration
- Timestamps (millisecond precision)
- Block types and workspace positions

Solution construction sequences can be reconstructed, enabling analysis of:

- Problem-solving strategies
- Debugging patterns
- Common misconception trajectories
- Block-to-text transition readiness

4. Forum and Help-Seeking Behavior

- Posts created and replies given
- Helpfulness ratings
- Response latency

This supports behavioral modeling and identification of self-regulated learners.

5. Gamification State Tracking

- Gamification events include:
- XP accumulation over time
- Badge awards
- Level progression
- Leaderboard rankings

These indicators support motivational analytics and correlation studies between recognition and performance.

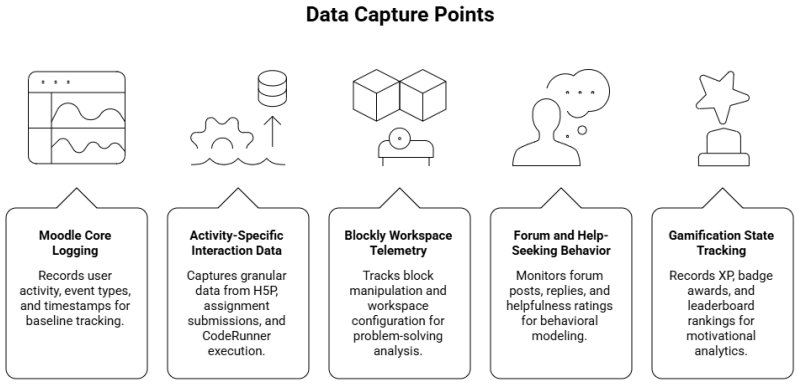


Figure 11. Conceptual data

3.4.2 The Analytics Dashboard & Predictive Models

Analytics dashboards provide visual representations of learner performance at both individual and cohort levels, as mentioned in Figure 12.

Dashboard Indicators

- **Learner Progress**
  Percentage completion of modules and levels.
- **Performance Trends**
  Improvements or stagnation across multiple attempts.
- **At-Risk Identification**
  Flags triggered by repeated failures, inactivity, or low performance thresholds.
- **Mastery Levels**
  Aggregated indicators showing competence in programming, IoT logic, and conceptual understanding.

Dashboards support both instructors and learners:

- Instructors monitor class-wide patterns and intervene strategically.
- Learners track their own progress and self-regulate their study behavior.

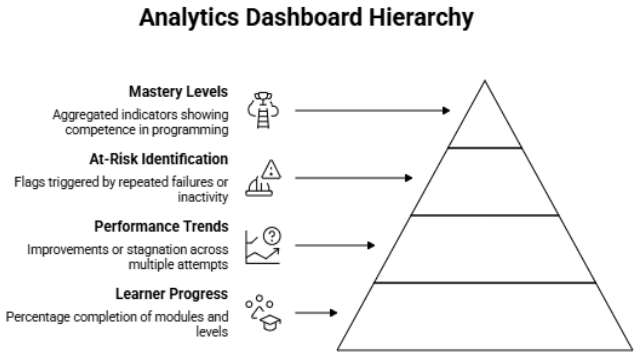


Figure 12. Analytics Dashboard hierarchy

3.4.3 Automated Intervention System

A rule-based intervention system enhances adaptivity within the asynchronous environment. This can be seen in the Figure 13.

Trigger Conditions

- Multiple failed CodeRunner attempts.
- Low quiz performance.
- Extended inactivity.
- Failure to meet minimum progression thresholds.

Automated Responses

- Release of remedial content.
- Activation of structured hints.
- Suggestion of review materials.
- Instructor notification for targeted support.

These interventions maintain learner engagement while reducing instructor workload, preserving scalability.

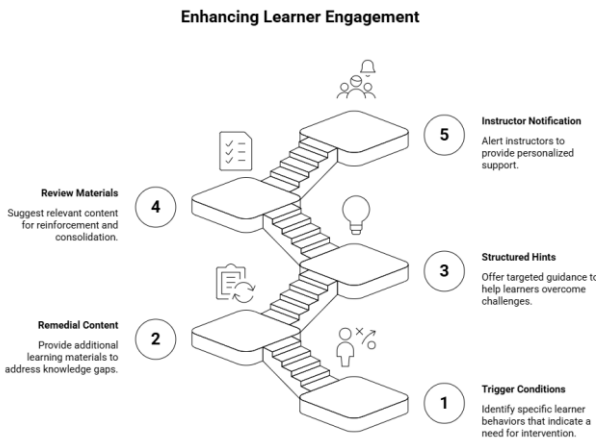


Figure 13. Enhancing Learner Engagement

3.5 Summary

This chapter has presented the comprehensive design and architecture of the Smart Learning Ecosystem (SLE), structured around a Moodle-centric hub-and-spoke model. In this architecture, Moodle functions as the central orchestration hub, while external programming and physical computing tools serve as loosely coupled practice environments. All assessed work, progression tracking, gamification, analytics, and adaptive interventions are consolidated within Moodle, ensuring pedagogical coherence and technical scalability. The system integrates:

- Structured asynchronous workflow management,
- Automated CodeRunner-based programming evaluation,
- Embedded gamification mechanisms,
- Centralized analytics and adaptive interventions.

By consolidating assessment, progression tracking, and feedback within Moodle while allowing flexible experimentation in external environments, the architecture achieves pedagogical coherence, technical scalability, and progressive STEAM skill development.

This architectural framework establishes the operational foundation for the implementation details described in Chapter 4, including Moodle configuration, CodeRunner customization, system integration, and pilot deployment [54]. Chapter 5 subsequently presents the validation procedures and evaluation results that assess the effectiveness of the Smart Learning Ecosystem in practice.

Chapter 4 - Educational Platform Setup, Configuration, and Pedagogical Implementation

This chapter documents the practical implementation of the Smart Learning Ecosystem (SLE) introduced in Chapter 3. While the previous chapter established the conceptual architecture and pedagogical design principles, the present chapter transitions from theoretical design to technical realization. It details the configuration decisions, development environment setup, and system integration processes required to operationalize the ecosystem.

The development phase constituted the foundational technical work undertaken prior to system deployment. It involved the preparation of the development infrastructure, the selection and configuration of the core learning management platform, the design of an extensible plugin ecosystem, and the integration of external programming environments that together form the system's hub-and-spoke architecture. Within this architecture, Moodle operates as the central orchestration hub, while external development tools provide specialized environments for programming, robotics, and applied STEAM learning activities.

Implementation followed a structured and iterative deployment process that prioritized stability, scalability, and reproducibility. The process began with foundational platform configuration and environment preparation, progressed through the development of the five-stage pedagogical workflow defined in Chapter 3, incorporated gamification and analytics capabilities through an extensible plugin infrastructure, and established external practice environments that support hands-on learning activities. The development phase concluded with initial pilot testing and iterative refinement of system components.

The remainder of the chapter is organized into five sections. Section 4.1 presents the system architecture and platform selection, explaining the technical decision framework and the rationale for selecting Moodle as the core platform. Section 4.2 describes the development environment configuration, including server setup and infrastructure preparation. Section 4.3 details the Moodle platform configuration, covering installation, site-wide settings, interface customization, course architecture, and user management.

Section 4.4 introduces the plugin infrastructure, through which Moodle is extended to support adaptive learning, analytics, and gamification features. Section 4.5 presents the external programming environments, specifically MIT App Inventor and the mBlock Arduino IDE, which extend the ecosystem beyond the Moodle environment and support practical STEAM activities. Finally, Section 4.6 provides a summary of the development environment and system setup.

4.1 System Architecture and Platform Selection

The Smart Learning Ecosystem was designed using a hub-and-spoke architecture in which Moodle functions as the central platform coordinating all learning activities, assessments, and analytics processes. In this configuration, Moodle manages user authentication, content delivery, course organization, and evaluation procedures, while external tools provide specialized environments for programming experimentation and project development. This architectural approach offers several advantages. First, it centralizes learning management and assessment within a single platform, ensuring that student progress, submissions, and performance data are stored consistently. Second, it allows external tools to be integrated into the learning workflow without requiring complex technical integrations. Third, it enables the use of Moodle's analytics and reporting mechanisms to monitor learner engagement and progress across all course activities.

Within this architecture, Moodle serves as the primary interface for students and instructors. External tools such as MIT App Inventor and mBlock are accessed through instructional resources provided inside Moodle, allowing learners to move between environments while maintaining a consistent course structure.

The system architecture therefore combines the flexibility of specialized programming environments with the organizational and analytical capabilities of a modern learning management system.

4.2 Development Environment Configuration

4.2.1 Local Development Environment

The Smart Learning Ecosystem was developed within a local server environment to facilitate rapid experimentation, configuration testing, and iterative development.

A local development setup allows developers to modify system configurations and test platform features without affecting external users or requiring continuous internet connectivity.

The platform was deployed on a workstation running **Windows 11** equipped with an **11th Generation Intel Core i5 processor**, **16 GB of RAM**, and **150 GB of SSD storage**. These hardware specifications provided sufficient computational resources for running the Moodle server environment while simultaneously supporting development tools and multiple browser sessions for testing user interactions. To host the Moodle platform locally, the **XAMPP integrated server environment** was used, as we can see in Figure 14. XAMPP provides a complete web server stack packaged for Windows systems, including the Apache web server, the MySQL database management system, and the PHP scripting language required by Moodle [67].

The selected configuration included:

- **Apache 2.4** as the web server responsible for handling HTTP requests.
- **MySQL 8.0** for database management and storage of system data.
- **PHP 8.1** as the scripting language used by Moodle to execute server-side operations.

Running the platform locally enabled rapid iteration during the development phase, allowing system modifications to be tested immediately. In addition, the local environment provided full administrative control over system configuration files and server logs, facilitating debugging and performance optimization.

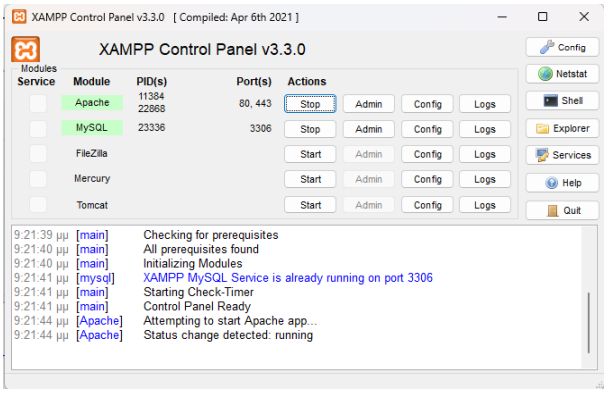


Figure 14. XAMPP Server on Windows

4.2.2 Moodle Installation

The Moodle platform was installed as the core technological component of the Smart Learning Ecosystem. Version 4.1 LTS (Long-Term Support), shown in the Figure 15, was selected due to its stability, extended security updates, improved user interface, and compatibility with a wide range of educational plugins.

The installation followed Moodle's recommended deployment procedure. Initially, a dedicated database was created within the MySQL server to store all system data, including user accounts, course structures, activity logs, and assessment results. A database user with restricted privileges was configured to ensure secure database access.

Next, the Moodle source files were deployed within the Apache web server directory. During installation, a configuration file (config.php) was automatically generated to define system parameters such as database connection settings, file storage locations, and site URLs.

A separate moodledata directory was created outside the web-accessible directory structure. This directory stores uploaded files, cached content, and temporary system data while preventing unauthorized direct access from external users.

The installation process also included verification of server environment requirements, such as required PHP extensions and memory limits. Once these requirements were satisfied, an administrator account was created to enable full system configuration and platform management.

To ensure secure communication between users and the platform, HTTPS encryption was enabled using Transport Layer Security (TLS). This configuration protects authentication credentials and user data during transmission between client devices and the server.

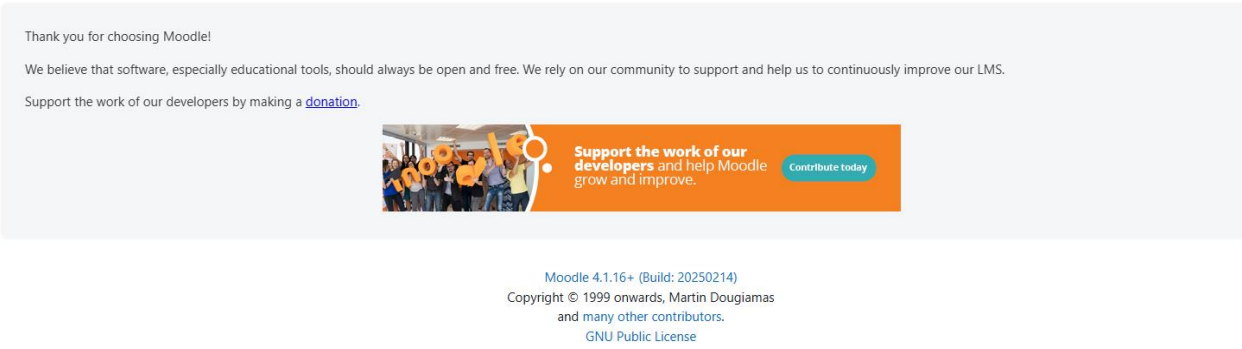


Figure 15. Moodle Installation

4.3 Moodle Platform Configuration

With the development environment established, the Moodle platform underwent a structured, multi-stage configuration process to transform the base installation into the operational backbone of the Smart Learning Ecosystem. This process proceeded in a deliberate sequence — from installation and core system settings, through security hardening, to interface customization, course architecture, and user provisioning — ensuring that each layer of configuration rested on a fully validated foundation before the next was applied.

4.3.1 General Site Configuration

Following a clean installation of the Moodle platform, a set of general site-wide configurations was implemented to establish the operational identity, structural organization, and behavioral defaults of the learning environment.

The site was assigned the full title **"Smart Learning Ecosystem for STEAM Education"**, with the abbreviated name **"SLE-STEAM"** , shown in Figure 16, isconfigured for compact display in navigation elements such as breadcrumbs, menus, and system notifications. A site description was also defined to reflect the research objectives, interdisciplinary STEAM orientation, and applied pedagogical methodology underpinning the study, thereby contextualizing the platform for all enrolled participants.

English was selected as the default system language, while multilingual content support was enabled to facilitate the potential future localization of learning materials and the extension of the platform to multilingual cohorts. Additionally, the system timezone was synchronized with the geographic location of the pilot institution. This ensured that all system-generated timestamps—including assignment deadlines, submission records, and activity logs—were aligned with the local time reference frame of both instructors and learners. Such temporal consistency is particularly important for the accurate interpretation of engagement patterns within the learning analytics component of the system.

Beyond these foundational identity settings, several core pedagogical configurations were implemented at the platform level to support the instructional architecture of the Smart Learning Ecosystem:

- **Course Category Architecture:** Four top-level course categories were created corresponding to the thematic modules defined in the instructional design: Environmental Monitoring, Home Automation, Health and Wellness, and Integrated Capstone Projects. Each category was assigned descriptive metadata and default course settings tailored to the target cohort of secondary education learners.
- **Role Framework:** The platform's three-tier role structure—Administrator, Teacher, and Student—was verified against Moodle's default configuration. Minor adjustments were applied where necessary, although custom role definitions were ultimately not required, as the native Moodle permission framework sufficiently supported the access control requirements of the study.
- **Activity Completion Tracking:** Platform-wide completion tracking was enabled to support learner-facing progress indicators, instructor monitoring dashboards, and the conditional logic required for structured progression through course materials.
- **Conditional Activity Release:** The conditional activities feature was activated, allowing learning resources and assessments to be unlocked based on prerequisite completion, grade thresholds, or scheduled dates. This functionality provides the technical mechanism that supports the mastery-based, progression-gated workflow embedded in the five-stage pedagogical model of the SLE.
- **Gradebook Configuration:** Grading scales and aggregation rules were defined within the Moodle gradebook to ensure consistent evaluation across diverse activity types, including interactive H5P content, CodeRunner programming exercises, and Moodle quizzes. This configuration ensures transparent and coherent grade calculation for both instructors and students.

Together, these general configuration settings established the foundational identity, organizational structure, and operational parameters of the learning platform, enabling it to support the pedagogical design and research objectives of the Smart Learning Ecosystem for STEAM Education.

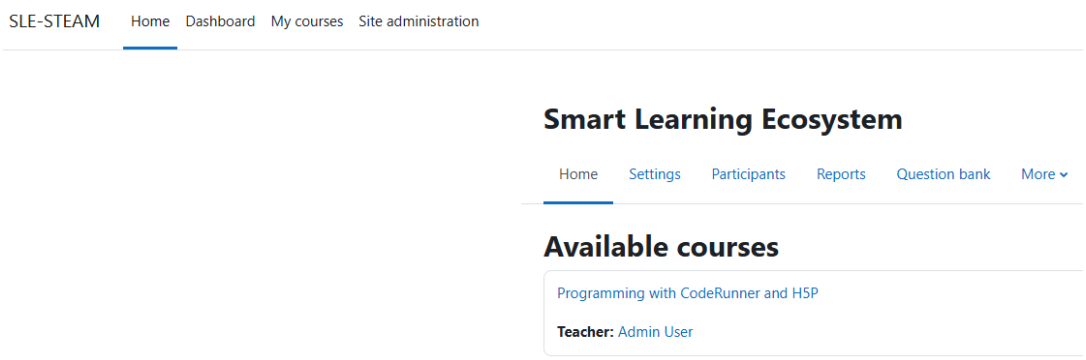


Figure 16. Smart Learning Ecosystem – SLE STEAM

4.3.2 Security Configuration

Security configuration received dedicated attention during development, reflecting the platform's role as the custodian of both learner personal data and the research dataset. Configuration was guided by Moodle's official security documentation, institutional data protection requirements, and the principles of defense-in-depth — applying multiple overlapping controls so that no single vulnerability could compromise system integrity [68]. We can see it in details in Figure 17.

Authentication and Credential Policy

Moodle's password policy was enabled and configured to require a minimum of eight characters, including at least one uppercase letter, one lowercase letter, one digit, and one non-alphanumeric special character. These requirements align with current guidelines from recognized security frameworks and substantially reduce the feasibility of brute-force attacks against individual accounts. Password reset request tokens were configured with a 30-minute validity window, limiting the exposure period of any intercepted reset link. Email change requests were set to require confirmation to the new address, preventing unauthorized modification of account contact information.

Account Protection Mechanisms

Account lockout was configured to activate after five consecutive failed login attempts, after which the affected account remains inaccessible for 15 minutes. This threshold balances security — making automated credential-stuffing attacks impractical — against usability, minimizing disruption to legitimate users who may mistype their credentials.

The observation window within which failed attempts are counted was set to ensure that attempts are monitored within a bounded time period before the counter resets, preventing low-rate attacks that space attempts to avoid threshold triggers.

Transport and Communication Security

All client-server communications are encrypted via HTTPS, enforced through the platform's www root configuration. Transport Layer Security (TLS) ensures that authentication credentials, session tokens, and learner data transmitted between user devices and the server cannot be intercepted in transit. This configuration is particularly important given that the platform handles sensitive assessment data and personally identifiable information about minor participants.

Access Control and Privacy Protections

Several settings were activated to protect user identity and restrict unauthorized information access. Username enumeration via the password recovery form was disabled, preventing external parties from confirming whether a given username or email address is registered on the platform. User profile pages were restricted to authenticated users, eliminating public exposure of learner information. Profile descriptions were configured to be visible only to users enrolled in at least one course, limiting their availability to confirmed participants. Search engine indexing was disabled to prevent course content, forum discussions, and learner profiles from appearing in external search results.

Content Security and File Management

User-submitted content is automatically sanitized through Moodle's built-in XSS filtering, with trusted content explicitly disabled to ensure that no user-generated HTML or JavaScript is rendered without sanitization. Embedding via HTML object or embed tags was disabled to prevent injection of potentially unsafe external multimedia content. A 30-minute editing window was applied to forum posts, balancing flexibility for minor corrections against the integrity of collaborative discussion records. File upload and storage limits were defined to prevent resource exhaustion: the maximum site-wide upload size was set to 40 MB per file, and per-user private file storage was capped at 100 MB.

These limits accommodate the file types legitimately needed for the SLE curriculum — code files, project screenshots, and documentation — while preventing storage abuse.

System-Level Security

Moodle's scheduled task (cron) execution was restricted to command-line invocation only, preventing the cron script from being triggered via an HTTP request. This eliminates a potential avenue for unauthorized execution of privileged system operations. Strict validation of required form fields was enabled to prevent submission of incomplete or malformed data to system functions. Group enrolment key policies were aligned with the site-wide password policy, ensuring that course access codes meet the same strength requirements as user credentials.



Figure 17. Moodle Security Configuration

User Management and Role Configuration

The user management structure of the SLE was designed around Moodle's native role-based access control (RBAC) model, shown in the following Table 13, which assigns capability sets to named roles and applies those roles contextually at the system, course category, or course level [69]. Three principal role types were configured to support the study's operational requirements:

Table 13. Moodle Roles

Role	Core Capabilities	System Access	Assignment
Administrator	Full platform management including plugin configuration, user provisioning, system monitoring, security settings, and analytics access	Unrestricted system-wide	Research team members
Teacher	Course creation and editing, activity configuration, student grading, completion tracking oversight, and analytics dashboard access	Course and category level	Two pilot instructors
Student	Course enrolment, activity participation, assignment submission, quiz attempt, forum posting, and personal progress dashboard	Course level	Pilot participant cohort

User accounts were provisioned in bulk using Moodle's CSV upload functionality, which processes a flat-file import containing fields for username, first name, last name, email address, authentication method, and cohort assignment. Thirty student accounts were generated for the pilot cohort. Cohort assignments were established during provisioning to enable subgroup differentiation in the learning analytics layer — allowing the Analytics Graphs plugin to filter engagement data by learner subgroup and supporting the comparative analysis required by the study's research design.

Authentication was configured using Moodle's built-in manual authentication method rather than external identity providers, a deliberate choice that maintains full control over account lifecycle management within the research environment and eliminates dependency on third-party authentication services that could introduce access disruptions during the study period.

4.4 Plugin Infrastructure

Moodle's modular architecture allows additional functionality to be integrated through plugins. Several plugins were installed to support the specific requirements of the Smart Learning Ecosystem, including interactive learning materials, automated programming assessment, gamification, and learning analytics.

Plugin selection was based on the following criteria:

- compatibility with Moodle 4.1 LTS,
- stability and active development support,
- alignment with the pedagogical objectives of the ecosystem,
- compliance with security and performance standards.

The selected plugins extend the platform's capabilities beyond traditional learning management functions and provide the technical foundation required for the advanced instructional features described in the following chapter.

4.4.1 Interactive Content Delivery – H5P Integration

The H5P plugin (version 1.27.2, build 2024120900) was integrated into the Moodle learning management system to enable the creation and deployment of interactive multimedia learning objects through the "Interactive Content" activity type, illustrated in Figures 18 and 19. By selecting H5P – Interactive Content when adding a new activity within a course module, instructors were able to create and embed interactive learning materials directly in the course environment without requiring external tools. This activity type provides access to multiple H5P content libraries, allowing the development of resources such as embedded quizzes, drag-and-drop exercises, simulations, and interactive videos. These activities were primarily used as scaffolding mechanisms preceding programming-based assignments, supporting conceptual understanding before learners progressed to applied implementation tasks.

To support assessment and monitoring, gradebook synchronization was enabled to automatically record scores from H5P activities, while attempt limits were applied to formative tasks to encourage reflective engagement. Additionally, activity completion tracking was activated to facilitate the collection of learning analytics data and monitor student progress. The system also supported content reuse across course modules, promoting modular instructional design and consistent learning resources. Operational settings included automatic saving of user progress to allow learners to resume interactive activities without losing work, as well as standard display options such as action bars, copyright information, and embedding functionality to ensure accessibility and portability of content. Integration with the H5P Hub was maintained to allow retrieval and updating of interactive content libraries, while content types requiring a Learning Record Store (LRS) were not enabled, as analytics were managed through Moodle's native tracking mechanisms.

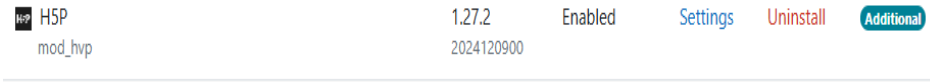


Figure 18. H5P Plugin Installation

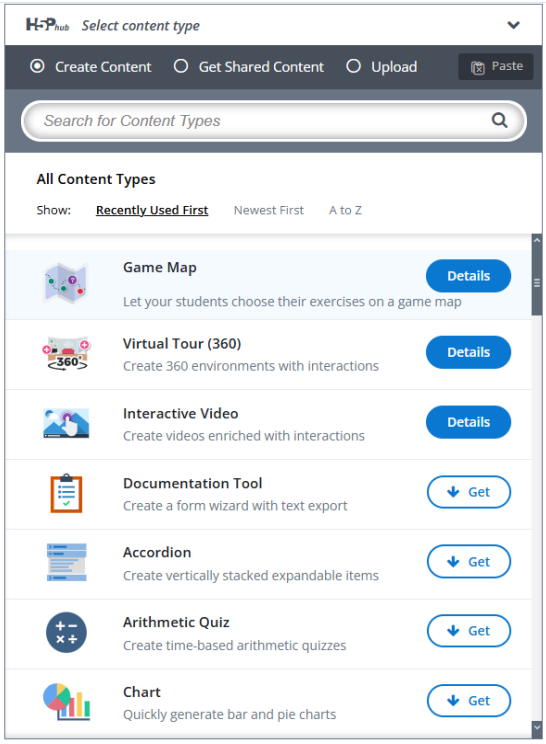


Figure 19. H5P Select Content Type

Automated Programming Assessment – CodeRunner and Jobe Server Integration

The CodeRunner plugin (qtype_coderunner, version 5.5.0, build 2025010200) constitutes the assessment backbone of the Smart Learning Ecosystem, it's Architecture shown in Figure 20.. CodeRunner enables the creation of programming question types within Moodle's native quiz engine, each of which presents a programming task, accepts learner code submissions, executes those submissions against a predefined set of test cases, and returns structured, immediate feedback indicating which test cases were passed and which were failed. For a research system whose pedagogical model centers on iterative programming skill development through formative assessment cycles, CodeRunner's automated evaluation capability is indispensable [54].

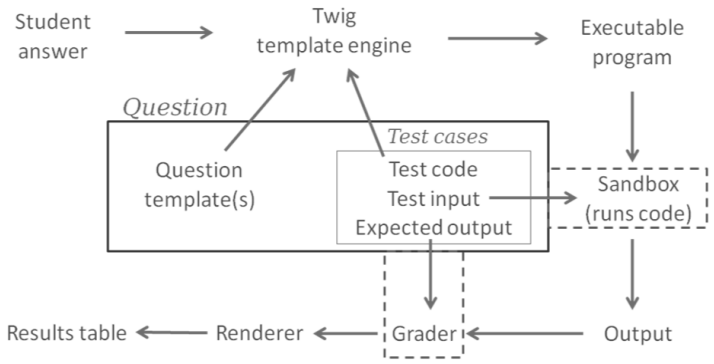


Figure 20. The Architecture of CodeRunner

Automated Programming Assessment in Moodle

1. The CodeRunner Plugin

CodeRunner is a Moodle plugin designed to automate the assessment of programming exercises [20]. It evaluates student-submitted code against predefined test cases, providing immediate and objective feedback. By integrating with Moodle quizzes, CodeRunner supports multiple programming languages and allows educators to create diverse tasks, from text-based coding problems to visual programming workflows. This automation reduces manual grading workload, ensures consistent evaluation, and helps students refine their programming skills through rapid, actionable feedback.

Key Configuration and Technical Details

- **Version and Dependencies:** The version used is **qtype_coderunner 5.5.0 (build 2025010200)**. It is enabled in Moodle and required by the **local_coderunnerh5p** plugin.

- **Execution Sandboxes:**
  - **Jobe Sandbox** (jobesandbox_enabled) is enabled by default to run code in a secure, isolated container.
  - **Ideone Sandbox** (ideonesandbox_enabled) is available but typically disabled, as it is deprecated.

- **Jobe Server Configuration:**
  - **Host** (jobe_host): Specifies the remote Jobe server for executing student code (default: jobe2.cosc.canterbury.ac.nz). Multiple servers can be defined to distribute workload.
  - **API Key** (jobe_apikey): Used for authenticated REST requests to the Jobe server (default: 2AAA7A5415B4A9B394B54BF1D2E9D).

- **Grading Behavior:**
  - **Default Penalty Regime** (default_penalty_regime): Defines penalties applied to new questions, expressed as a comma-separated list (e.g., 10, 20, ...).
  - **Grade Cache** (enablegradecache): Experimental feature to store grading results locally, reducing repeated executions for identical submissions.

These settings allow educators to control the execution environment, security, and grading behavior, making CodeRunner a flexible and robust platform for automated programming assessment. It's installation illustrated in the below Figure 21.



Figure 21. Installation of CodeRunner plugin

2. Adaptive Behavior Plugin for CodeRunner

To enhance interactivity and support adaptive assessment in programming exercises, the **Adaptive adapted for CodeRunner** plugin (qbehaviour_adaptive_adapted_for_coderunner) was enabled within the Moodle environment.

- **Plugin Name:** Adaptive adapted for CodeRunner (qbehaviour_adaptive_adapted_for_coderunner)
- **Version:** 1.4.3 (build 2024051200)
- **Status:** Enabled

This plugin modifies Moodle's question behavior so that it works seamlessly with CodeRunner, allowing the grading engine to communicate effectively with the Moodle quiz system. As a result, it enables **instant feedback for each attempt**, **multiple attempts within the same question**, and **adaptive assessment**, where students can correct mistakes based on feedback without restarting the entire task. These features support iterative learning and significantly improve the interactivity and efficiency of programming assessments.

Without this plugin, CodeRunner would have limited ability to interact with the Moodle quiz environment, reducing its capacity to provide adaptive learning experiences.

The installation process is straightforward, requiring the plugin to be placed in the server directory:

/question/behaviour/adaptive_adapted_for_coderunner

Version 1.4.3 ensures compatibility with recent Moodle and CodeRunner releases, providing a stable and reliable adaptive learning environment for programming exercises, shown in the below Figure 22.

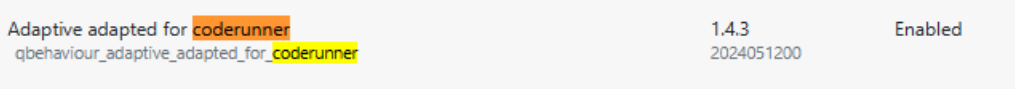


Figure 22. Adaptive adapted for coderunner plugin

3. Secure Execution and Error Detection

A core component of CodeRunner's effectiveness is its secure execution of student code through integration with the **Jobe server**, a remote containerized sandbox hosted at the University of Canterbury [70]. This setup ensures:

- **Isolated execution**, preventing unauthorized access to system resources.
- **Security and integrity**, mitigating risks from arbitrary code execution.
- **Resource management**, including limits on memory usage and execution time.

Educators can define **test cases** to automatically detect two main types of errors:

- **Syntax Errors**: Violations of a programming language's grammar, such as missing parentheses or incorrect variable names. CodeRunner relies on language compilers or interpreters to detect these errors.
- **Logic Errors**: Mistakes in program logic that produce incorrect outputs despite syntactically correct code. Test cases compare the student's output with the expected result, allowing these errors to be identified.

Additionally, custom templates support **Blockly-to-Python translation workflows**, enabling visual programming exercises to be validated automatically using Python execution. This combination ensures reliable assessment, scalability, and the validity of research data [24].

Jobe Server Configuration

To ensure secure and isolated code execution, CodeRunner was configured to use a remote Jobe server hosted at the University of Canterbury. The Jobe server provides containerized sandboxing, preventing unauthorized access to system resources and mitigating security risks associated with arbitrary code execution. Setting up a secure execution sandbox.
The configuration process included:

- Establishing secure API communication between Moodle and the remote Jobe server.
- Defining Python as the primary programming language.
- Setting memory and execution time limits to prevent abuse.
- Enabling structured feedback based on unit-test evaluation logic.

Custom question templates were developed to support Blockly-to-Python translation workflows, allowing visual programming tasks to be automatically validated through Python execution [24]. This configuration ensured scalable automated assessment while maintaining system integrity and research data validity.

Custom question templates were also developed to support **Blockly-to-Python translation workflows**, allowing visual programming tasks to be automatically validated through Python execution. This setup ensures scalable automated assessment while maintaining system integrity and research data validity [24].

Sandbox Web-Service Settings

CodeRunner includes optional experimental web-service features to allow direct access to the sandbox server (usually Jobe), with the following configuration options (shown in Figure 23.):

- **Enable Sandbox Web Service** (wsenabled): Default **No**. Allows direct access to the sandbox server; marked as an experimental feature.

- **Jobe Server for Web Services** (wsjobeserver): Default **Empty**. Specifies an alternative Jobe server for web-service execution; multiple servers can be defined for load balancing.
- **Log Sandbox Web Service Usage** (wsloggingenabled): Default **Yes**. Logs every execution via the sandbox web service; required for user rate throttling.
- **Max Hourly Submission Rate** (wsmaxhourlyrate): Default **200**. Limits the number of submissions per user per hour; set to 0 to disable rate limiting.
- **Max CPU Time (secs)** (wsmaxcputime): Default **5**. Restricts the maximum CPU time a web service job can use, even if the sandbox parameter sets a higher limit.

These settings allow administrators to enhance the security, performance, and reliability of automated code execution while providing control over user activity and resource usage.

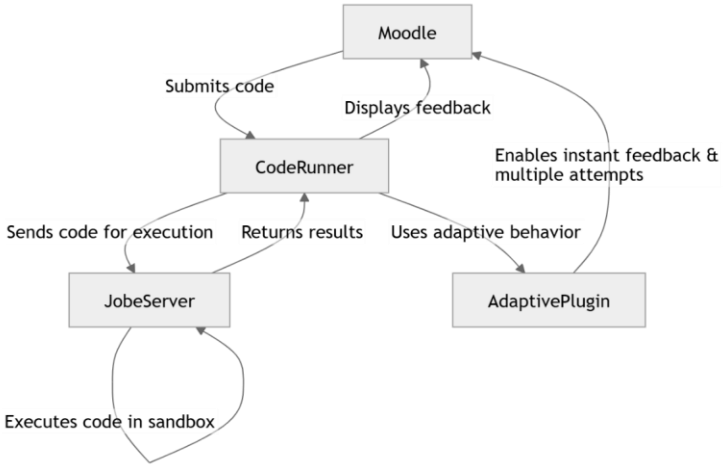


Figure 23. Assessment workflow diagram

4.4.2 Learning Analytics and Reporting – Analytics Graphs Plugin

To support quantitative monitoring of learner engagement and performance, the Analytics Graphs reporting plugin was installed in Moodle and this can be seen in Figure 24.

The plugin provides graphical representations of:

- Activity participation frequency
- Assignment submission patterns
- Grade distribution metrics
- Course-level engagement comparisons

Configuration included:

- Instructor-level access to visualized reports.
- Cohort-based filtering for subgroup analysis.
- Data export functionality to support external statistical processing (e.g., SPSS/R analysis).

The integration of this plugin enabled systematic collection of behavioral engagement indicators, directly supporting the empirical evaluation framework of the study.



Figure 24. Analytics Graphs Dashboard

4.4.3 Gamification Framework – Level Up XP Integration

To enhance learner motivation and promote sustained engagement, a structured **gamification framework** was implemented within the Moodle learning environment. The design combined Moodle's native gamification functionalities with the **Level Up XP plugin** to introduce experience-based progression mechanics while preserving the integrity of formal assessment procedures [36].

Built-in Moodle Gamification Features

Several native Moodle features were configured to support a structured and engaging learning pathway:

- **Activity completion tracking** was enabled platform-wide, allowing learners to monitor their progress through course materials.
- **Automated badge issuance** was configured based on predefined completion and performance criteria.
- **Conditional access restrictions** were implemented to support the progressive unlocking of course content once prerequisite activities were completed.

These mechanisms provided continuous feedback and encouraged learners to progress through the course in a structured manner.

Level Up XP Plugin Implementation

To extend Moodle's built-in capabilities, the **Level Up XP (block_xp)** plugin (version **19.0 – build 2025100500**) was installed and enabled. The plugin introduced an experience points (XP) system designed to reward learner participation and reinforce engagement.

The configuration included the following core parameters:

- **Automated allocation of experience points (XP)** based on learner actions within the course.
- **Level thresholds** mapped to cumulative XP scores, enabling learners to progress through different levels as they accumulated points.
- **Visual progression indicators** displayed in the learner interface, including dashboards and leaderboards.
- **Course-level XP tracking**, meaning that points earned apply only within the specific course where the block is active.
- **Administrative control over XP management**, with manual modification privileges restricted to administrators to maintain the validity of the gamification system.

Optional extensions of the Level Up XP ecosystem—such as **Level Up XP Availability**, **Level Up XP Enrolment**, and **Shortcodes for XP integration within course content**—were identified as recommended plugins but were **not installed**, as the implemented gamification design focused on activity-based progression within the course itself.

Additional administrative configurations included disabling XP accumulation for administrators, limiting log retention to three days, and allowing occasional anonymous usage reporting to support plugin development.

Experience Point Allocation Rules

Experience points were awarded through **event-based rules**, illustrated in Table 41, allowing the system to automatically detect learner actions and assign XP when specific conditions were met. These rules ensured that points were granted only when predefined activities were successfully completed.

Two primary event rules were configured:

- **Quiz Submission Rule**
  - **5 XP** were awarded when a learner **submitted a quiz attempt**.
  - The rule was triggered by the event **"Quiz attempt submitted."**
  - The rule applied specifically to the activity **"Coderunner – Practical Training."**
- **Interactive Activity Completion Rule**
  - **2 XP** were awarded when a learner **submitted an H5P activity attempt**.
  - The rule was triggered by the event **"H5P: attempt submitted."**
  - The rule applied to the interactive learning activity **"Game Map."**

In both cases, XP were granted only when **all rule conditions were satisfied**, ensuring precise tracking of meaningful learner interactions within the course.

Table 14. Level Up XP points

Activity Type	XP Award	Conditions
H5P attempt submitted	2 XP	Per module completion
Quiz attempt submitted	5 XP	Per passing test case
Forum post	2 XP	Helpful contribution

Pedagogical Design Considerations

The gamification system was designed to support learner motivation without affecting the formal grading structure of the course. Experience points and level progression served exclusively as motivational and engagement mechanisms, while official assessment scores continued to determine academic evaluation.

This approach allowed gamification to enhance participation and learner engagement while maintaining fairness, transparency, and methodological validity within the instructional and research framework. This approach allowed gamification to enhance participation and learner engagement while maintaining fairness, transparency, and methodological validity within the instructional and research framework.

Level Progression Design

Ten progressive levels were defined numerically using snowballing XP thresholds to sustain motivation (shown in Figure 25.) :

- **Level 1:** 0–100 XP (Novice Explorer)
- **Level 2:** 100–250 XP (Apprentice Builder)
- **Level 3:** 250–450 XP (Competent Developer)
- **Level 4:** 450–700 XP (Proficient Programmer)
- **Level 5:** 700–1000 XP (Skilled Integrator)
- **Level 6:** 1000–1350 XP (Advanced Creator)
- **Level 7:** 1350–1750 XP (Expert Innovator)
- **Level 8:** 1750–2200 XP (Master Architect)
- **Level 9:** 2200–2700 XP (STEAM Scholar)
- **Level 10:** 2700+ XP (SLE Champion)

Each level unlocked additional pedagogical affordances:

- **Level 3:** Access to advanced Blockly challenges
- **Level 5:** Early access to capstone projects
- **Level 7:** Ability to create peer assessment criteria
- **Level 9:** Mentor status with forum moderation privileges

This structured progression reinforced mastery orientation while supporting differentiated advancement.

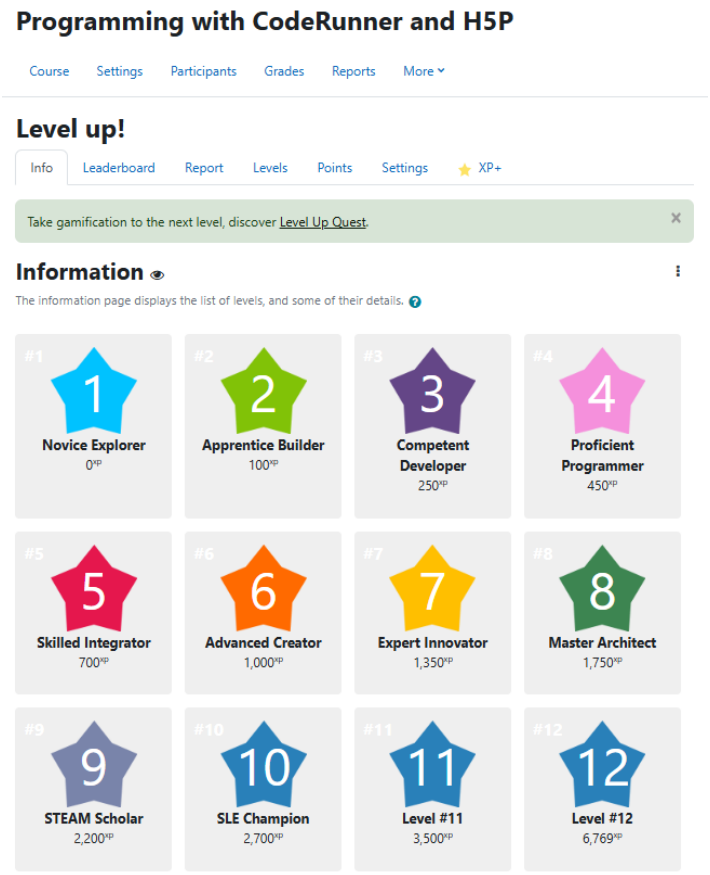


Figure 25. List of levels

Achievement Badges

Custom badges were developed within Moodle's badge system to recognize specific competencies and behavioral achievements:

- **Loop Master:** Complete Blockly questions using iteration blocks
- **Sensor Sage:** Implement different sensor types in mBlock
- **Debugging Detective:** Revise and improve submissions based on feedback
- **Peer Mentor:** Provide helpful forum responses
- **Integration Innovator:** Complete capstone projects

Badges were issued automatically based on predefined criteria, with monitoring supported through event-based tracking mechanisms.

Visual and Motivational Design

The integration of gradebook aggregation, XP allocation, level thresholds, unlockable privileges, and automated badge issuance created a coherent gamification ecosystem. Visual progress indicators and the visible XP block supported transparency, goal-setting, and self-regulated learning behaviors, while maintaining alignment with measurable academic performance outcomes.

Overall, the gamification engine functioned as an embedded motivational scaffold rather than an external reward layer, reinforcing skill mastery across the IoT and programming workflow stages.

4.5 External Programming Environments

In addition to the Moodle platform, external programming environments were prepared to support hands-on experimentation and project development activities. MIT App Inventor was used as a cloud-based development environment for mobile application programming. Because the platform operates entirely through a web browser, no local installation was required. Students accessed the platform using individual accounts created through a code-based login mechanism.

Instructional resources within Moodle provided students with direct links to the App Inventor environment, tutorial materials, and guidance for completing programming tasks. This approach allowed students to experiment with mobile application development while maintaining Moodle as the central learning management platform.

4.5.1 MIT App Inventor Setup

Students accessed MIT App Inventor exclusively through a web browser (see Figure 27.), as the platform is cloud-based and requires no local installation. Account provisioning was completed at the outset of the activity: learners generated individual MIT App Inventor accounts through the platform's code-based login mechanism using the provided registration (Revisit) code (shown in Figure 26.), eliminating the need for Google-based authentication. Step-by-step documentation with screenshots was supplied to ensure successful account setup.

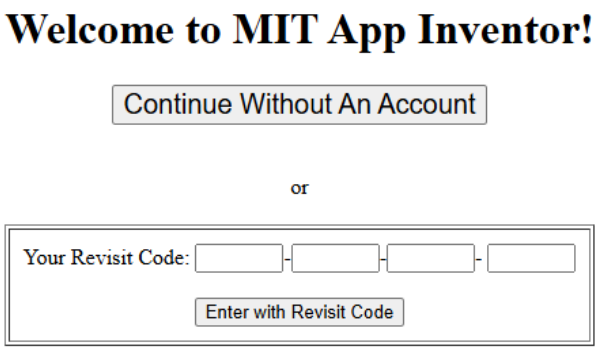


Figure 26. Revisit Code Login

No platform-level technical integration with Moodle was implemented. Instead, structured instructional support was delivered entirely within Moodle through a dedicated resource page containing:

- Direct links to the MIT App Inventor website
- Video tutorials demonstrating project creation and live testing
- A troubleshooting guide addressing common technical issues
- QR code generation instructions for application sharing

Clear instructional guidelines were provided inside Moodle to ensure alignment with assessment objectives. These included:

- Required block structures
- Expected application behavior
- Explicit alignment with upcoming CodeRunner assessments

Students were explicitly instructed not to export .aia files, as evaluation occurred exclusively within Moodle.

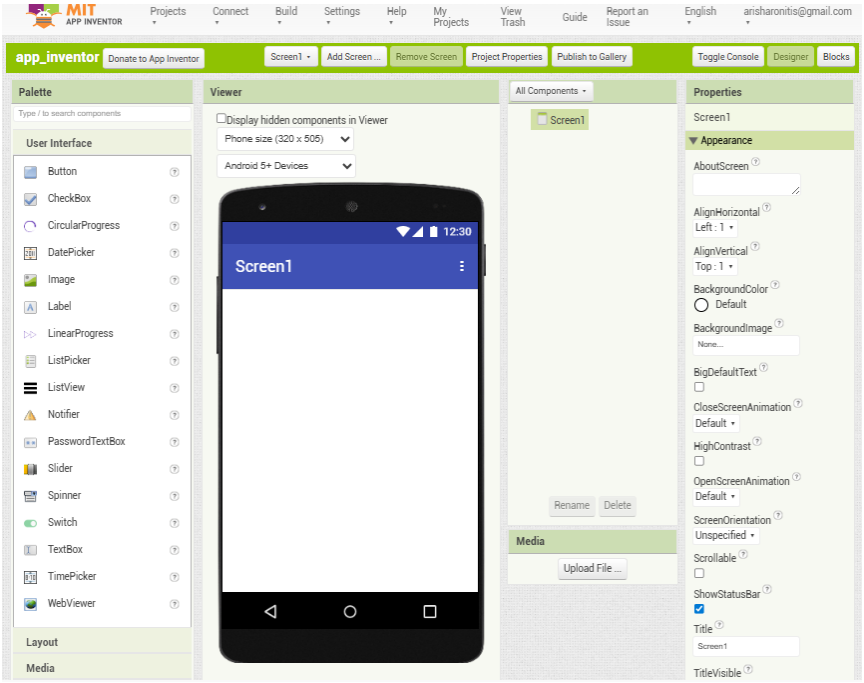


Figure 27. App Inventor Environment

4.5.2 Arduino via mBlock Setup

1. Introduction to mBlock

The mBlock programming environment was used to support visual programming and hardware experimentation with Arduino-based devices (see Figure 28.)

.The online version of the platform was selected to ensure accessibility across different operating systems without requiring software installation.

mBlock provides a block-based programming interface inspired by the Scratch programming model. This interface allows beginners to construct programs by connecting visual blocks representing programming logic, reducing the risk of syntax errors while promoting algorithmic thinking.

The platform also supports a transition from visual programming to text-based Python programming, making it suitable for progressive learning scenarios in which students gradually develop more advanced programming skills.

The **online version of mBlock (ide.mblock.cc)** allows users to program directly from a web browser without installing additional software.

Key Characteristics

- Block-based programming interface
- Web-based IDE available through a browser
- Supports robotics and hardware devices
- Integration with AI and IoT technologies
- Ability to switch to Python programming

2. Purpose of mBlock in Education and Research

mBlock was created to make programming easier for beginners and students who are starting to learn computational thinking.

Main Educational Objectives

- Teach **fundamental programming concepts**
- Support **robotics and hardware programming**
- Encourage **problem-solving and logical thinking**
- Introduce **AI and IoT technologies**

The platform bridges the gap between **digital programming and real-world applications**, enabling students to see their code control robots, sensors, and electronic systems.

3. Architecture of the mBlock IDE

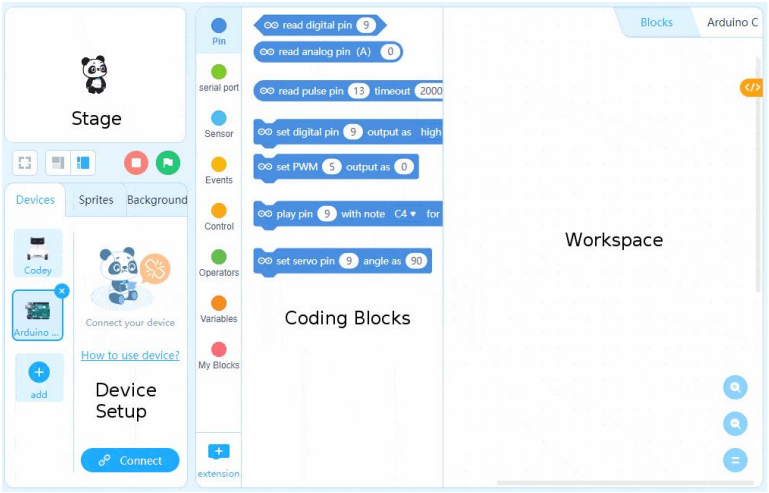


Figure 28. mBlock Environment

The **mBlock Integrated Development Environment (IDE)** consists of several components that allow users to build and execute programs visually.

Main Components of the IDE

1. **Menu Bar**
    - Used to create, save, and manage projects.
2. **Code Blocks Panel**
    - Contains programming blocks categorized by function (motion, control, sensing, etc.).
3. **Script Editor / Workspace**
    - Area where users drag and combine blocks to build programs.
4. **Stage Area**
    - Displays the output of the program such as animations or simulations.
5. **Device Control Panel**
    - Used to connect hardware devices like robots or microcontrollers.
6. **Sprite Management**
    - Allows users to add and control characters or objects.
7. **Background Control**
    - Used to customize the environment or scene of the program.

4. Programming Methods in mBlock

mBlock supports two different programming approaches:

1. Block-Based Programming

- Drag-and-drop visual programming
- Designed for beginners
- Eliminates syntax errors
- Ideal for learning logic and algorithms

2. Text-Based Programming (Python)

- Advanced programming environment
- Allows writing traditional code
- Suitable for complex applications

Users can switch from blocks to Python with a single click, which helps learners transition from visual programming to professional coding.

5. Key Features of the mBlock Platform

1. Artificial Intelligence (AI)

mBlock integrates AI services such as:

- Image recognition
- Voice recognition
- Machine learning models
- Text-to-speech and translation services

These features allow students to create **AI-based projects and smart systems**.

2. Internet of Things (IoT)

The platform supports IoT projects by allowing devices to:

- Collect data
- Send data to cloud services
- Visualize sensor information

3. Robotics Programming

mBlock can control several hardware platforms including:

- Makeblock robots
- Arduino boards
- Micro:bit devices

Through these integrations, users can program robots to move, detect obstacles, and interact with the environment.

6. Advantages of Using the Online mBlock IDE

The **web version (ide.mblock.cc)** provides several advantages:

- No installation required
- Accessible from any browser
- Cross-platform compatibility (Windows, macOS, Linux)
- Easy device connection via Bluetooth or USB
- Cloud project storage

This makes the platform highly suitable for **educational environments and collaborative learning**.

7. Conclusion

mBlock is a powerful educational programming platform that simplifies the learning process of coding and robotics. By combining **visual programming, AI capabilities, and hardware integration**, it enables users to move from beginner-level programming to advanced applications.

The **online mBlock IDE** enhances accessibility by allowing users to program and test projects directly through a browser, making it an effective tool for both **academic research and STEM education**.

4.5.3 External Tool Integration: App Inventor and mBlock

The integration of external programming environments — MIT App Inventor for mobile application development tasks and mBlock for Arduino-based hardware programming — required instructors to configure connection points between the Moodle course interface and these external platforms.

For App Inventor integration, instructors configured resource pages containing hyperlinks to the App Inventor web-based IDE, accompanied by detailed onboarding instructions using the Revisit Code guest access mechanism. Students completed their projects externally and submitted exported project files or screenshots through Moodle assignment activities, enabling instructor review and feedback within the Moodle gradebook.

For mBlock integration, instructors provided installation guidance for the mLink browser extension required for hardware connectivity, together with project specification documents detailing the Arduino sensor and actuator tasks to be completed. The instructor's role in managing external tool integration extends beyond initial configuration to ongoing support during delivery: onboarding sessions specifically addressed App Inventor account provisioning and mBlock hardware connectivity, acknowledging that these external dependencies represented the most significant potential friction points in the student experience.

4.6 Course Structure and Module Organization

The course architecture adopted a sequential modular design, supporting a gradual transition from fundamental programming literacy to applied IoT system development. Early modules introduced visual programming and core computational concepts, intermediate modules focused on user interface design and IoT interactions, and the final module engaged learners in capstone project development, fostering synthesis of knowledge and innovation. Each module encapsulated a self-contained learning unit with its own sequence of instructional resources, interactive activities, programming exercises, and formative assessments. This modular structure supports the progression-gated workflow embedded in the SLE's pedagogical model, whereby learners advance through activities in a defined sequence contingent upon satisfying prerequisite completion criteria.

4.6.1 Course Configuration and Interface

Upon course creation, a comprehensive set of course-level settings was applied to establish the operational parameters of the learning environment. The course format was configured as a Topics format, enabling the clear delineation of thematic modules as distinct course sections visible to enrolled participants. Section names and descriptions were defined for each module to provide learners with contextual orientation and learning objectives prior to engagement with module content. Course visibility was initially set to hidden during the development and content population phase, ensuring that students enrolled in advance of the pilot study period were not prematurely exposed to incomplete content. Upon completion of course development and validation of all activities, visibility was enabled to coincide with the official commencement of the pilot study.

The course interface was customized through the application of the Boost theme, which had been configured at the platform level as described in Chapter 4. Within the course, a custom banner image was added to the course header to reinforce the STEAM orientation and visual identity of the SLE. Navigation elements, including the course menu and activity breadcrumbs, were verified to function correctly across all enrolled user roles.

4.6.2 Module Structure and Sequencing

Each of the four thematic modules was structured according to a consistent internal template derived from the five-stage pedagogical model. Within every module, the following activity sequence was implemented:

- An introductory resource page providing module context, learning objectives, and links to relevant external resources.
- An App Inventor or mBlock activity providing a hands-on, hardware or mobile application development task.
- One or more H5P interactive content activities serving as conceptual scaffolding prior to programming tasks.
- A CodeRunner programming exercise, presenting a progressive sequence of coding tasks aligned with the module theme and competencies.
- A forum discussion activity facilitating peer collaboration and reflective engagement with module content.

Conditional access restrictions were applied throughout each module to enforce the sequential progression required by the mastery-based pedagogical model. Specifically, the CodeRunner programming exercises were configured to become accessible only upon completion of the preceding H5P scaffolding activities, and the summative quiz was unlocked exclusively upon submission of the programming exercise.

This conditional release logic ensured that learners engaged with conceptual preparation materials before attempting applied coding tasks, in alignment with the scaffolded instructional design of the SLE.

4.7 Student-Facing Learning Tools

This section presents the suite of interactive learning tools that students directly encounter and use throughout the Smart Learning Ecosystem. While the preceding section described the overall course structure and pedagogical sequencing, the present section focuses specifically on the student experience — the tools, interfaces, and mechanisms through which learners engage with content, develop programming skills, and receive feedback on their progress.

These tools were deliberately selected and configured to reduce the cognitive barriers typically associated with programming education while maintaining the rigour required for meaningful STEAM competency development.

The student-facing layer of the SLE comprises four principal tool categories: H5P interactive content activities, the Blockly visual programming environment embedded within CodeRunner, the CodeRunner automated assessment interface, and the gamification layer providing experiential feedback through experience points, level progression, and badge recognition. Each category addresses a distinct phase of the pedagogical workflow and collectively they form an integrated learning interface that guides students from initial conceptual exploration through to applied programming practice and reflective review.

4.8 Teacher Tools for Lesson Design and Content Organisation

The Smart Learning Ecosystem provides instructors with a comprehensive set of tools for designing, structuring, and delivering the course curriculum. These tools operate at the level of course architecture and content creation — they determine what students encounter, in what order, under what conditions, and with what instructional intent.

This section documents the teacher-facing tools for lesson design and content organisation as a distinct conceptual category from the assessment and monitoring tools described in Section 4.8. The distinction between design tools and monitoring tools is both conceptually and practically significant. Design tools are employed primarily in the pre-delivery phase, during the construction and configuration of the learning environment, though they continue to be used iteratively throughout delivery as instructors refine content in response to observed learner behaviour. Monitoring tools, by contrast, are deployed primarily during and after the delivery phase to observe, evaluate, and respond to learner engagement and performance.

4.8.1 Moodle Course Architecture and Sequencing Tools

The foundational tool for instructor lesson design within the SLE is the Moodle course editor, which provides a graphical interface for constructing and organising the course content hierarchy. Instructors use the Topics format to divide the course into discrete thematic modules, each representing a self-contained unit of learning. Within each module, activities and resources are arranged in a defined sequence that reflects the five-stage pedagogical workflow: introductory resource pages at the Exploration stage, H5P activities at the Conceptualisation stage, CodeRunner exercises at the Application stage, summative quizzes at the Evaluation stage, and forum discussions at the Reflection stage.

The course editor provides drag-and-drop reordering of activities within and between modules, section-level naming and description fields for contextual orientation, and individual activity settings accessible through the activity configuration interface.

Section visibility controls allow instructors to hide entire modules during content development and reveal them progressively. The course-level settings panel provides global configuration options including course visibility, enrolment mechanisms, completion tracking mode, and gradebook aggregation rules.

4.8.2 Conditional Activity Release and Mastery Gating

Among the most pedagogically significant tools available to instructors within the SLE is the Moodle conditional activity release mechanism, which enables the definition of prerequisite completion criteria that must be satisfied before a subsequent activity becomes accessible.

This tool operationalises the mastery-based progression logic central to the SLE's pedagogical model, ensuring that students engage with conceptual scaffolding before attempting applied programming exercises, and that they complete programming exercises before undertaking formal summative assessment. Instructors configure conditional release rules at the individual activity level through the Restrict Access section of the activity settings panel. Within the SLE, two primary conditional release patterns were implemented: CodeRunner programming exercises became accessible only upon completion of the preceding H5P scaffolding activities, and summative quizzes unlocked exclusively upon submission of the programming exercise.

The visual feedback provided to students through completion tracking — checkmarks indicating completed activities and locked icons indicating unsatisfied prerequisites — gives students clear actionable information about their progression without requiring instructor intervention.

4.9 Teacher Tools for Assessment and Monitoring

The second major category of teacher-facing tools encompasses the instruments through which instructors configure, administer, and interpret assessments, monitor learner engagement and performance in real time, and implement adaptive interventions in response to observed patterns. Assessment and monitoring tools are most intensively used during and after course delivery, when the learning activities configured during design are generating the behavioural and performance data from which instructional decisions are made.

4.9.1 Assessment Configuration and Administration Tools

CodeRunner Configuration

CodeRunner's instructor-facing configuration interface provides a highly granular set of controls for defining the parameters of automated programming assessment. The question type configuration panel allows instructors to specify the programming language (Python 3 in the SLE's implementation), the answer box dimensions, precheck and feedback settings, the grading model, and the penalty regime for multiple attempts. The most technically demanding aspect of CodeRunner configuration is the design and validation of test cases — the input-output pairs against which student submissions are evaluated.

Each test case specifies the standard input, the expected output, and test properties including mark allocation, display settings, and sequencing rules.

For exercises incorporating the Blockly visual programming interface, instructors additionally configure the Customisation template that establishes the simulated execution environment within which student-generated Python code is evaluated. The Arduino simulation template deployed in the SLE defines Python equivalents of Arduino functions (pinMode, digitalWrite, analogRead, delay) that produce formatted output strings rather than interacting with hardware, enabling automated evaluation of hardware-oriented programs without physical device connectivity.

Moodle Quiz Configuration

Summative evaluation quizzes are configured through the Moodle quiz activity settings interface, providing controls for attempt limits, time restrictions, question shuffling, grade aggregation, review settings, and feedback display. Within the SLE, summative quizzes were configured with a single-attempt restriction to enforce formal assessment conditions and time limits calibrated to the complexity of the assessment content. The Adaptive behaviour mode configured for CodeRunner questions within quiz attempts enables students to receive immediate automated feedback on individual programming submissions, revise their solutions, and resubmit, subject to the configured penalty regime.

4.9.2 Monitoring and Analytics Tools

The monitoring and analytics infrastructure of the SLE provides instructors with multi-layered visibility into learner engagement and performance across the full duration of the course. This infrastructure comprises two complementary layers: Moodle's core reporting and analytics capabilities, and the visual extension provided by the Analytics Graphs plugin.

Moodle Core Reporting Tools

The core reporting suite accessible through the course Reports menu provides instructors with the authoritative, queryable data record of all learner interactions within the SLE.

The Logs report, filterable by user, activity, date range, and action type, serves as the primary instrument for reconstructing individual learner interaction sequences and investigating specific engagement events.

The Live Logs view provides a real-time stream of the same data, enabling immediate situational awareness during supervised sessions. The Activity Report provides a summary of access frequency and completion status for each activity, while the Course Participation report presents an aggregated comparison of engagement levels across all enrolled learners.

The Moodle gradebook provides the formal aggregation point for all assessment scores generated across the course. Instructors use it to review grade distributions at both the individual and cohort level, identify students whose performance trajectories suggest emerging difficulties, and evaluate the difficulty calibration of assessment items. Moodle's predictive analytics capability, implemented through the Learning Analytics API, provides a forward-looking complement to the descriptive reporting tools.

The Students at Risk of Dropping Out predictor generates probabilistic risk indicators for enrolled learners based on their engagement pattern trajectories, enabling proactive identification of students who may require early intervention.

Analytics Graphs Plugin: Visual Monitoring Interface

The Analytics Graphs plugin transforms raw Moodle data into graphical visualizations that are more readily interpretable for routine instructional monitoring. The plugin provides seven distinct visual report types, shown in Table 15: a Grades Chart displaying score distributions; a Content Accesses report quantifying access frequency per resource; a Number of Active Students time-series showing daily engagement; a Hits Distribution chart of learner interaction frequency across the course timeline; a graphical Activity Report; a graphical Course Participation report; and an Activity Completion matrix providing a colour-coded grid of completion status across students and activities.

The Activity Completion matrix is particularly valuable for identifying students who have stalled at specific gating points within the conditional activity sequence. The cohort-based filtering capability enables instructors to generate all visual reports separately for defined learner subgroups, supporting comparative analysis between cohorts. The data export functionality supports CSV download for downstream statistical analysis in external tools.

Table 15. Instructor Monitoring Tools and Their Primary Uses in the SLE

Report Type	Tool / Source	Primary Instructor Use
Activity Logs / Live Logs	Moodle Core	Individual interaction audit; real-time session monitoring
Activity Report	Moodle Core	Per-activity access and completion overview
Course Participation	Moodle Core	Cohort-level engagement comparison
Gradebook Reports	Moodle Core	Grade distribution and module performance
Students at Risk Model	Moodle Predictive Analytics	Early identification of at-risk learners

Grades Chart	Analytics Graphs Plugin	Visual score distribution across cohort
Hits Distribution	Analytics Graphs Plugin	Engagement pattern and deadline-proximity analysis
Activity Completion Matrix	Analytics Graphs Plugin	Gating point stall detection across cohort
Course Participation (Graphical)	Analytics Graphs Plugin	Visual subgroup comparison with cohort filtering
Level Up XP Reports	Level Up XP Plugin	Gamification engagement metrics and level progression

4.9.3 Adaptive Intervention Mechanisms

The monitoring infrastructure generates the data upon which instructional interventions are based; those mechanisms (see Table 16.). are the instruments through which instructors act on that data. The SLE supports three categories of adaptive intervention: rule-based automated triggers that respond to learner behaviour without requiring instructor action; instructor-initiated direct interventions enabled by the monitoring infrastructure; and structural adaptive mechanisms embedded in the course design through conditional release and gamification-gated content.

Rule-Based Automated Triggers

The most immediate form of adaptive response in the SLE operates automatically through the rule-based systems embedded in the CodeRunner assessment engine and the gamification framework. When a student submits a CodeRunner solution that fails to pass all test cases, the automated feedback system immediately returns a structured error report.

Within the gamification framework, the Level Up XP plugin's automated award rules generate an award event for every qualifying interaction, update the student's total, and recalculate their level status. If an XP award event pushes a student across a level threshold, the level-up notification is displayed immediately and any content conditional on the new level threshold is simultaneously unlocked.

Instructor-Initiated Interventions

The monitoring tools, enable instructors to identify specific intervention opportunities and act on them through Moodle's communication infrastructure. When the Activity Completion matrix reveals that a student has stalled at the H5P prerequisite stage, the instructor can initiate targeted outreach through Moodle's messaging system. When the Students at Risk predictor flags a learner whose interaction trajectory suggests impending disengagement, the instructor can intervene proactively. The Live Logs report, used during supervised pilot sessions, enabled real-time identification of students encountering technical difficulties, transforming the course delivery context from a one-to-many broadcast to an individually attentive, data-informed tutorial environment.

Structural Adaptive Mechanisms: Conditional Content and Gamification Gating

The most architecturally embedded form of adaptive response operates through the structural configuration of the course itself. The conditional activity release sequences implement a fixed adaptive logic: students who have not demonstrated readiness for a given activity are automatically directed back to prerequisite materials. The gamification-gated content unlocking mechanism adds a second layer: as students accumulate XP and advance through level thresholds, additional enrichment content becomes conditionally available, creating a differentiated learning pathway within a single course environment without requiring administrative segmentation of the learner cohort.

Table 16. Adaptive Intervention Mechanisms in the SLE

Mechanism Category	Instrument	Trigger	Response Type
Rule-Based Automated	CodeRunner Feedback Engine	Student code submission	Immediate automated error/pass feedback
Rule-Based Automated	Level Up XP Award Rules	Qualifying activity interaction	XP award, level-up, content unlock
Rule-Based Automated	Conditional Activity Release	Activity completion status	Automatic gating / unlock

Instructor-Initiated	Moodle Messaging	Monitoring-detected stall	Personalised outreach / support
Instructor-Initiated	Live Logs Monitor	Real-time idle / navigation event	In-session targeted support
Structural Adaptive	Prerequisite Release Logic	Prerequisite completion	Sequential content unlock
Structural Adaptive	Gamification Level Gating	XP threshold achievement	Enrichment content unlock

4.10 Implementation of the Five-Stage Pedagogical Workflow

The five-stage pedagogical model introduced in Chapter 3 and technically operationalized through the platform configuration described in Chapter 4 was implemented as a concrete instructional sequence within each course module. The five stages — Exploration, Conceptualization, Application, Evaluation, and Reflection — were realized through specific activity types and content structures deployed within the Moodle environment. The following subsections describe the implementation of each stage as it was applied across the course modules.

4.10.1 Stage 1 — Exploration

The Exploration stage was implemented through introductory resource pages and embedded multimedia content placed at the opening of each course module. These resources were designed to activate prior knowledge, establish contextual relevance, and introduce the thematic domain of the module. Content included short explanatory videos hosted on external platforms and linked through Moodle's URL resource type, illustrated text-based introductions authored directly within Moodle's page resource editor, and curated external readings related to the module's IoT or STEAM application domain. For the Environmental Monitoring module, the Exploration stage introduced learners to the concept of environmental sensing, presenting real-world applications such as air quality monitoring and temperature-based alert systems. For Home Automation, the stage contextualized programming tasks within smart home scenarios.

For Health and Wellness, learners were introduced to biometric sensing and data visualization concepts relevant to the App Inventor application development tasks that followed.

4.10.2 Stage 2 — Conceptualization

The Conceptualization stage was implemented through H5P interactive content activities, which served as the primary mechanism for structured knowledge acquisition prior to programming exercises.

As described in Chapter 4, H5P activities were authored directly within Moodle using a range of content types including interactive videos with embedded comprehension questions, drag-and-drop classification exercises, and branching scenario simulations.

Each H5P activity was configured with attempt limits and gradebook synchronization, ensuring that scores were recorded and that activity completion was tracked as a prerequisite for subsequent programming tasks. The interactive design of H5P content was intended to promote active engagement with conceptual material rather than passive consumption, in accordance with the constructivist orientation of the SLE's pedagogical framework.

4.10.3 Stage 3 — Application

The Application stage constituted the core programming practice component of each module and was implemented through a combination of CodeRunner exercises, MIT App Inventor projects, and mBlock Arduino tasks. This stage represented the primary site of applied skill development within the SLE and was designed to provide learners with immediate, structured practice in programming and hardware interfacing within contextually meaningful scenarios. CodeRunner exercises were embedded within Moodle quizzes and configured with the Adaptive adapted for CodeRunner question behavior, enabling learners to submit code, receive immediate automated feedback, and revise their solutions within a single quiz attempt.

Each exercise presented a programming task aligned with the module theme, accompanied by a problem description, example input-output pairs, and a structured workspace for code entry.

Test cases were defined for each question to enable automated evaluation of both syntactic correctness and logical behavior.

4.10.4 Stage 4 — Evaluation

The Evaluation stage was implemented through summative quizzes combining multiple-choice conceptual questions with CodeRunner programming tasks. These quizzes were configured with a single-attempt restriction and time limits appropriate to the complexity of the assessment content. Grade aggregation was managed through the Moodle gradebook, with question-level scores contributing to a module grade that was displayed transparently to both learners and instructors.

Evaluation activities were placed at the conclusion of each module and were accessible only upon completion of all prerequisite Application stage tasks. This conditional structure ensured that learners who reached the evaluation stage had engaged with the full sequence of instructional activities and were adequately prepared for formal assessment.

4.10.5 Stage 5 — Reflection

The Reflection stage was implemented through forum discussion activities configured within each module. Learners were presented with open-ended discussion prompts inviting them to articulate their problem-solving approaches, describe challenges encountered during programming tasks, and provide peer feedback on shared solutions.

Forum posts were moderated by the course instructor, and contribution to the discussion forum was configured as an activity completion criterion.

The Reflection stage served dual purposes within the SLE: pedagogically, it supported the development of metacognitive awareness and collaborative learning skills; empirically, forum interaction data contributed to the behavioral engagement dataset analyzed within the study's learning analytics framework. The gamification system also assigned experience points for forum participation, reinforcing engagement with the reflective component of the learning cycle.

4.11 Summary

The development phase of the Smart Learning Ecosystem (SLE) resulted in the creation of a fully configured and technically validated infrastructure designed to support the pedagogical architecture established in Chapter 3.

At the center of this infrastructure is the Moodle 4.1 LTS learning management system, deployed within a local development environment and extended through a set of specialized plugins and external programming tools. Together, these components form an integrated technological foundation capable of supporting the multi-component curriculum and interactive learning processes envisioned for the system.

The Moodle platform provides the secure and pedagogically structured operational environment required to host the SLE. Its functionality was expanded through the integration of four key plugins, each addressing a distinct aspect of the learning experience. Interactive content delivery is enabled through H5P, while automated programming assessment is supported by CodeRunner in conjunction with the Jobe remote code execution server. Learner engagement and behavioral data are captured and visualized through the Analytics Graphs plugin, and motivation is reinforced through the gamification mechanisms provided by Level Up XP. Rather than functioning as isolated tools, these plugins operate as coordinated components of a unified system in which activity completion triggers gamification rewards, assessment outcomes regulate learner progression, and interaction data contribute to the analytics layer.

In addition to the core LMS environment, two external programming practice environments—MIT App Inventor and mBlock for Arduino—were prepared to provide authentic contexts for experimentation and skill development that extend beyond the LMS's native capabilities. Although these environments are not technically integrated into Moodle, they are incorporated into the instructional workflow through deliberate pedagogical design and remain connected to the platform's assessment structure.

The chapter detailed the configuration of the server infrastructure, the installation and setup of the Moodle platform, the integration and configuration of supporting plugins, and the preparation of the external programming environments. Each technical decision—including version selections, configuration parameters, and tool choices—was made in alignment with the pedagogical design goals of the system.

Consequently, the infrastructure described in this chapter represents not only a technical implementation but also a set of design commitments that shape the learner experience and ensure the reliability of the research data generated through the system.

This completed infrastructure constitutes the technological substrate of the Smart Learning Ecosystem and provides the operational environment upon which the implementation activities described in Chapter 5, 6 and 7 are built, including the realization of the pedagogical workflow, learning activities, and evaluation mechanisms provided by the system.

Chapter 5 - H5P-Based Interactive Learning

5.1 Introduction

This chapter documents the practical implementation of the Smart Learning Ecosystem (SLE) for interactive STEAM education, building directly upon the development environment and system setup described in Chapter 4. While the previous chapter established the technical infrastructure — including the Moodle 4.1 LTS platform, the plugin ecosystem, and the external programming environments — the present chapter shifts focus to the design, configuration, and deployment of the H5P interactive content activities that constitute the Conceptualisation layer of the Smart Learning Ecosystem.

The implementation phase constitutes the operational instantiation of the instructional design principles outlined in Chapter 3 and the technical foundations laid in Chapter 4. It encompasses the development and deployment of interactive learning activities using H5P, including branching lessons, game maps, drag-and-drop exercises, memory card games, and interactive video content. Each implementation decision was made in alignment with the Conceptualisation stage of the five-stage pedagogical model of the SLE and the overarching research objectives of the study.

The chapter is structured into four sections. Section 5.2 presents the H5P interactive content layer as the primary student-facing conceptualisation interface, describing how activity types, gradebook synchronisation, and access sequencing were configured to scaffold learner progression. Section 5.3 extends this account to the full range of interactive and adaptive learning activities — including branching lesson design, H5P Game Map activities, — illustrating how each activity supports the Conceptualisation stage of the pedagogical workflow and contributes to learner engagement and knowledge consolidation. Section 5.4 provides a summary of the H5P implementation phase, synthesizing the design decisions, activity configurations, and pedagogical alignment documented across the chapter.

5.2 H5P Interactive Content: The Conceptualisation Interface

From the student's perspective, H5P activities represent the primary mechanism through which conceptual knowledge is acquired and consolidated prior to programming tasks. Students encounter H5P activities at the Conceptualisation stage of each module — positioned after the introductory Exploration resources and before the CodeRunner programming exercises — where they function as interactive scaffolding that prepares learners for applied coding tasks.
Rather than passively reading instructional material, students engage with a diverse range of H5P content types that demand active cognitive participation. Interactive videos pause at key moments to present embedded comprehension questions, requiring students to demonstrate understanding before playback continues.

Drag-and-drop classification exercises ask students to categorise programming constructs or IoT components, reinforcing conceptual structure through physical manipulation of on-screen elements. Crossword puzzles and memory card games consolidate vocabulary and symbol recognition in a low-stakes, game-like context. Branching scenario simulations present decision-point narratives in which students must apply conceptual knowledge to navigate outcomes, providing a bridge between abstract understanding and applied judgment.

The student interaction flow with H5P activities follows a consistent pattern across all modules. Upon navigating to a module's Conceptualisation section, the student encounters one or more H5P content items, each presenting a self-contained interactive challenge with immediate feedback on each response. Attempt limits configured at the activity level ensure that students engage seriously with each attempt, while gradebook synchronisation means that scores are automatically recorded without requiring manual submission. Completion of H5P activities is a prerequisite for access to the subsequent CodeRunner programming exercise, enforcing the scaffolded progression that underpins the SLE's pedagogical model.
From a usability standpoint, H5P content is displayed natively within the Moodle course interface without requiring any external tool access, login, or software installation. Students experience a consistent visual interface regardless of the H5P content type, with activity instructions, interactive elements, and feedback messages all presented within a single browser window.

This frictionless access was particularly important given the secondary education context of the pilot study, where minimising technical barriers was a prerequisite for equitable participation.

5.2.1 H5P Content Authoring Tools

The H5P authoring environment, accessible to instructors through the Moodle activity editor, provides a browser-based content creation interface for developing interactive learning materials without requiring any programming or multimedia production expertise. For interactive video activities, the authoring workflow involves uploading or linking a video resource, inserting embedded question elements at selected timestamps, and configuring feedback text and retry allowances. For drag-and-drop exercises, the instructor uploads a background image, defines drop zones with labelled categories, and populates the pool of draggable elements. For branching scenarios, the instructor constructs a decision tree of narrative scenes with branching choice options leading to different outcomes.

A key authoring consideration in the SLE context was the alignment of H5P activity content with the specific learning objectives and programming themes of each module. Module 1 activities addressed visual programming concepts and App Inventor fundamentals; Module 2 activities focused on IoT interaction patterns and Bluetooth communication; Module 3 activities engaged with sensor data interpretation and health monitoring; and Module 4 activities addressed capstone project planning and integration. The H5P gradebook synchronisation setting was enabled for all activities, with attempt limits set to encourage serious engagement while permitting a small number of reattempts.

5.3 Interactive and Adaptive Learning Activities: H5P, CodeRunner, and Branching Lesson Design

5.3.1Branching Lesson Design

Within this course, lessons are designed as interactive learning pathways that guide students through content, assessment, and feedback. Among these lessons, we analyze one in particular—Introduction to Visual Programming—as an example of how visual programming is introduced through a structured approach that integrates prior knowledge review, concept exploration, and adaptive assessment.

The lesson begins with a **diagnostic multiple-choice activity titled "Review of Past Knowledge,"** which aims to activate and assess learners' prior understanding, illustrated in Figure 29. This initial step provides a quick recap of key concepts previously learned, helping students prepare for new material.

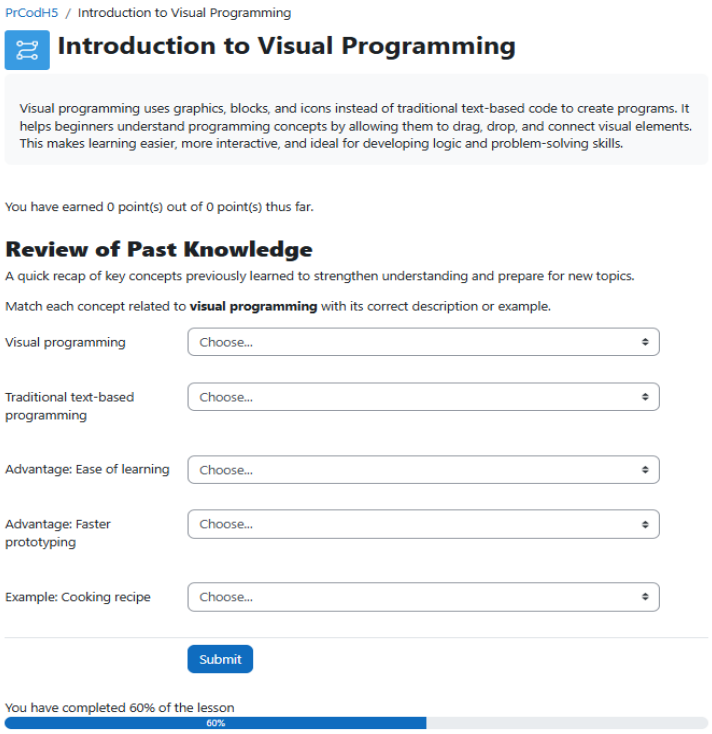


Figure 29. Lesson Multiple Choice

Following this, learners engage in a **matching activity** focused on core visual programming concepts, where they are required to connect terms with their correct definitions or examples. This activity supports conceptual clarity and reinforces foundational knowledge.

The lesson then presents **instructional content pages** that explain what visual programming is and outline its fundamental principles, including its structure, purpose, and differences from text-based programming, as we can see in the below Figure 30.

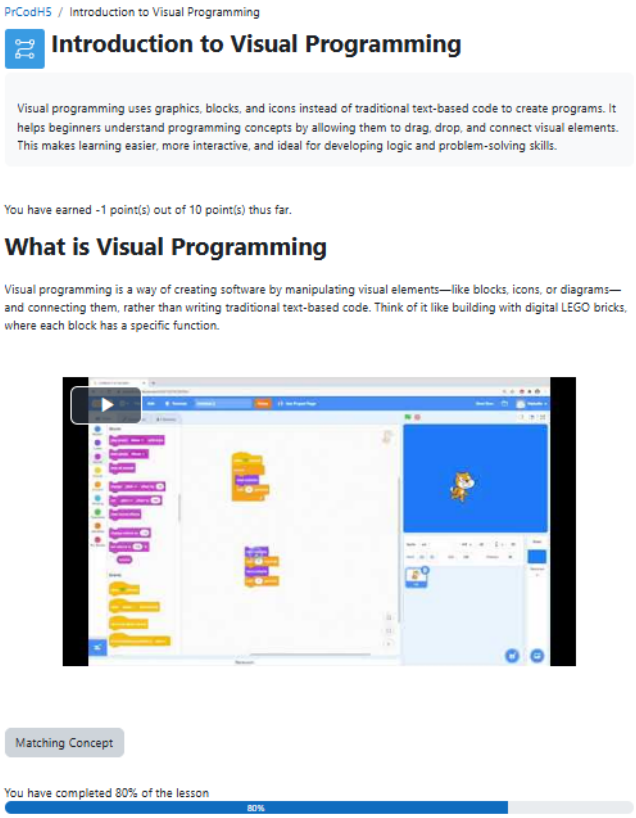


Figure 30. Lesson Multiple Choice

To evaluate understanding, learners complete a **multiple-choice assessment** comparing visual and traditional programming approaches. The lesson incorporates an **adaptive learning path**:

- If a learner answers correctly, they proceed to the next stage of the lesson (see Figure 31.).
- If a learner answers incorrectly, they are guided either to retry the question or to access a **supporting content page** containing additional explanatory material, including a video for reinforcement (see Figure 32.).
- After reviewing the auxiliary material, learners are given the opportunity to return and attempt the original multiple-choice question again.

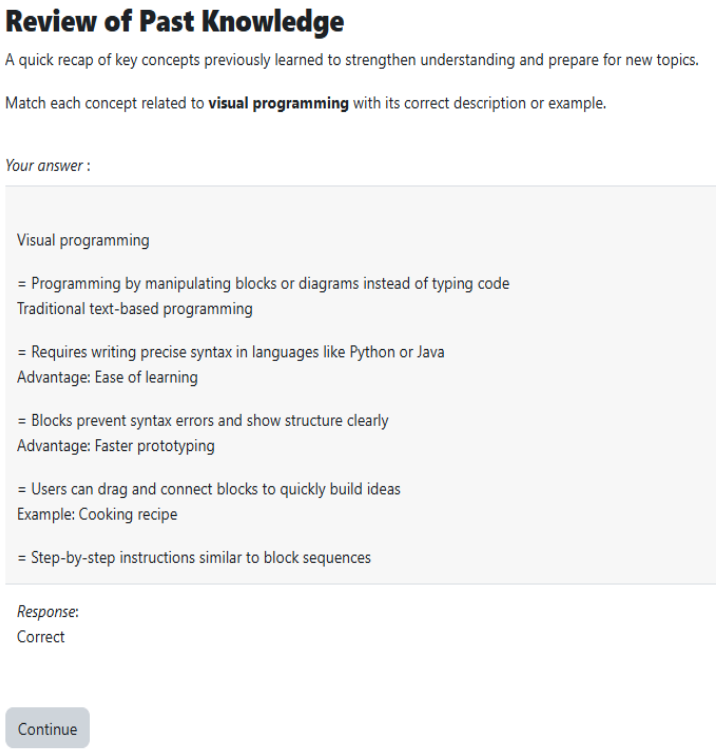


Figure 31. Correct Response

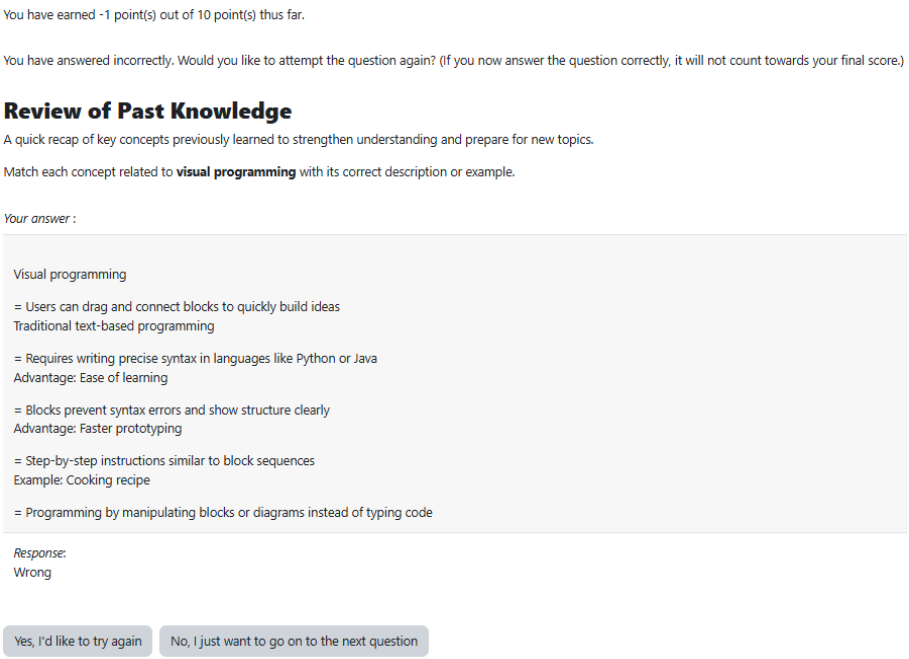


Figure 32. Wrong Response

This design promotes active engagement, self-paced learning, and knowledge reinforcement through immediate feedback and targeted support.

By the end of the lesson, learners are expected to: recall and connect prior knowledge with new concepts; understand the basic principles of visual programming; differentiate between visual and text-based programming approaches; and improve comprehension through iterative assessment and feedback.

5.3.2 H5P Interactive Content Activities

A total of 10 H5P activities were developed and deployed across the four course modules. Activity types were selected based on their alignment with specific learning objectives and the nature of the conceptual content being addressed within each module. The following H5P content types, shown in Table 17, were utilized in the implementation:

Table 17. H5P Content Types

H5P Content Type	Learning Objective
Interactive Media	Promote active viewing by improving attention and engagement, reinforcing learning through in-video questions, and supporting applied understanding.
Presentation & Learning Content	Design learning experiences that present content in a clear, guided sequence, incorporate multiple modalities such as text, audio, and video to support diverse learning preferences, and continuously reinforce understanding through embedded formative assessments.
Quizzes & Assessment	This integrated activity combines True/False, Fill in the Blanks, and Drag and Drop to check understanding, reinforce accurate recall, and build conceptual connections. It helps identify misconceptions, strengthen terminology, and promote active problem solving through categorization and application.
Games & Engagement	Memory games,maps, image pairing, and flashcards all strengthen recall and retention by reinforcing associations. Memory games boost recall, image pairing enhances visual recognition, and flashcards support memorization and spaced repetition, together creating an effective, well-rounded learning approach.

H5P Content Type	Learning Objective
Visual & Image-Based Content	These activities combine visual analysis and creativity: sliders sharpen observation by comparing changes, sequencing builds logical thinking through order, and collages foster storytelling and creative expression.
Content Exploration	Combining timelines, accordions, and dialog cards creates a focused learning approach: timelines clarify chronology and cause-effect, accordions chunk information for easy navigation and selective exploration, and dialog cards reinforce recall through self-testing, boosting memory retention efficiently.
Specialized Tools	A branching scenario builds decision-making and problem-solving skills, a 360° virtual tour enhances contextual learning and spatial awareness, and an arithmetic quiz strengthens numeracy, procedural skills, and accuracy, creating a well-rounded, practical learning experience.

This section analyzes all aspects of a representative activity, that includes four practical H5P exercises, illustrating how learners interact with the content and navigate each exercise. This analysis provides a visual overview of the structure and functionality of the implemented activities. This activity is graded with a maximum score of 10, with a minimum passing grade set at 9, reflecting the high level of understanding and performance expected. It is currently placed in the “Uncategorised” grade category. A finish score is also defined, allowing learners to achieve full completion without necessarily obtaining the maximum score from all exercises. The activity includes a retry option, enabling students to attempt the tasks multiple times in order to improve their performance and understanding. Grade settings illustrated in the below Figure 33.

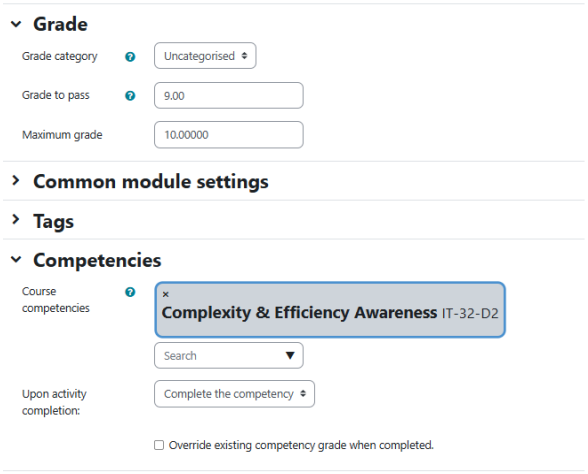


Figure 33. Grade Settings

The activity is structured with a staged map that may display stage labels on hover (where supported), helping learners navigate the content. Progression can be controlled through roaming settings, requiring students to either freely explore, complete, or successfully pass stages before unlocking subsequent ones. In particular, access to new stages may depend on successfully clearing previous ones. Visibility settings determine how far ahead learners can view upcoming stages, with the option to restrict visibility so that only unlocked stages are shown. These settings are designed to guide progression, encourage mastery, and support a structured learning experience, shown in Figure 34.

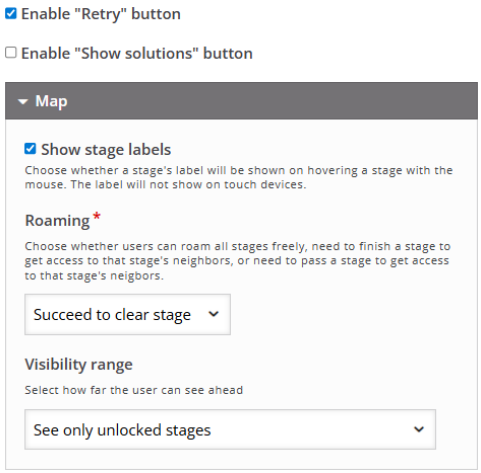


Figure 34. Game Map Settings

The activity is set up as a map called the Game Map (see Figure 35.), which consists of 4 stages that are linked together in a chain. Completing one leads to the next.



Figure 35. H5p Game Map Activity

Initially in the stage 1, learners are presented with a crossword puzzle consisting of seven words, with clues arranged both horizontally and vertically. Settings shown in the following Figures 36 and 37 and environment in Figure 38.

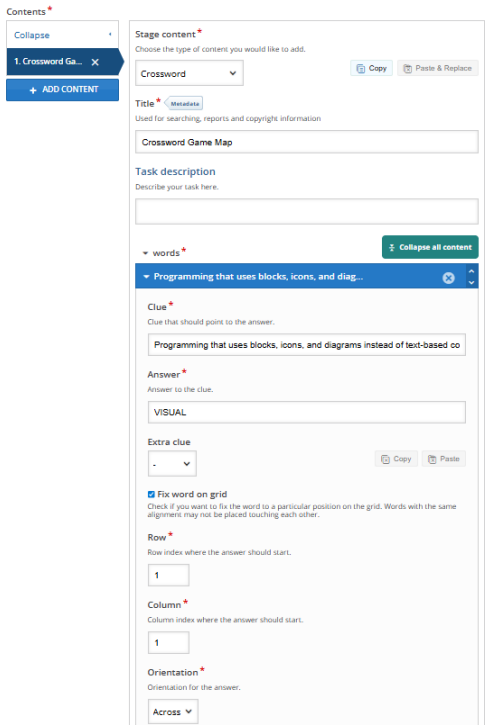


Figure 36. Crossword puzzle Settings

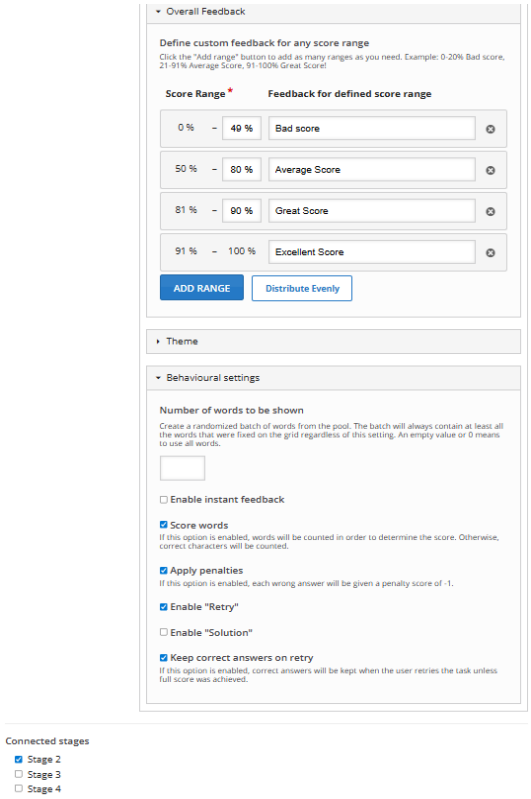


Figure 37. Crossword puzzle Settings

Successful completion of the activity requires all entries to be filled in correctly. In cases where incorrect or incomplete responses are submitted, these are highlighted with a red background (see Figure 39.), while correctly completed entries remain unchanged and are marked in green (see Figure 38.), in their respective positions. This design enables learners to clearly distinguish between correct and incorrect responses, facilitating error identification and supporting iterative improvement.

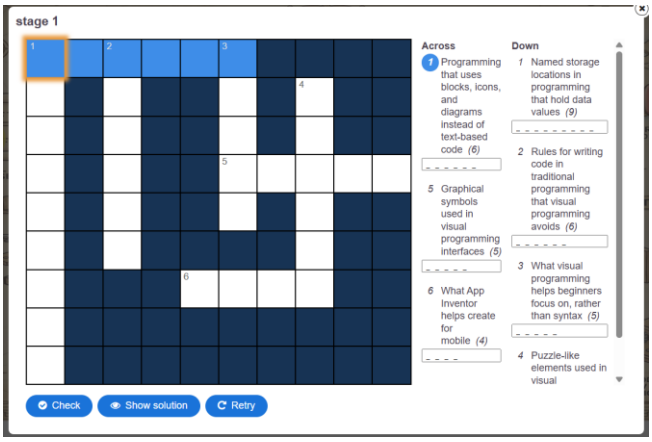


Figure 38. H5p Crossword Puzzle Activity

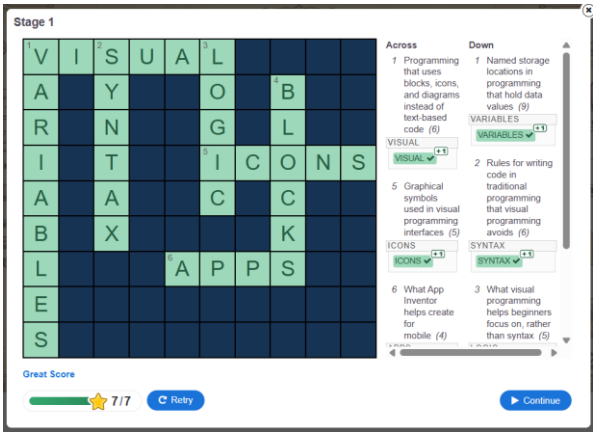


Figure 39. Crossword Puzzle – Correct Answer

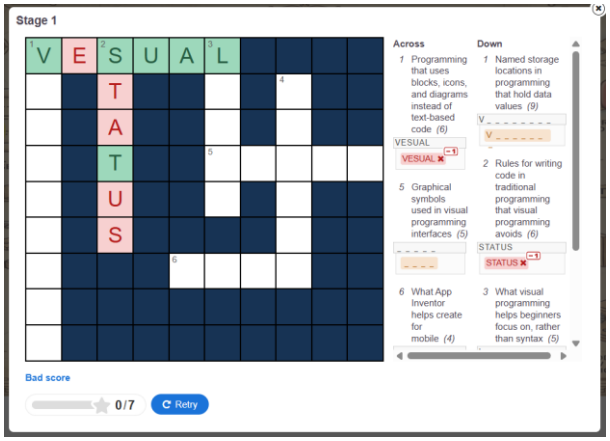


Figure 40. H5p Crossword Puzzle Activity

At this point, learners proceed to the next stage of the activity. This activity consists of a drag-and-drop task (see Figure 41.), designed to support learners in categorising and associating key concepts related to programming and technological applications. The task includes multiple thematic elements, such as "Traditional Programming," "IoT Examples," "Visual Programming Tools," and "Block Categories," which learners are required to match appropriately by placing draggable items into predefined drop zones.

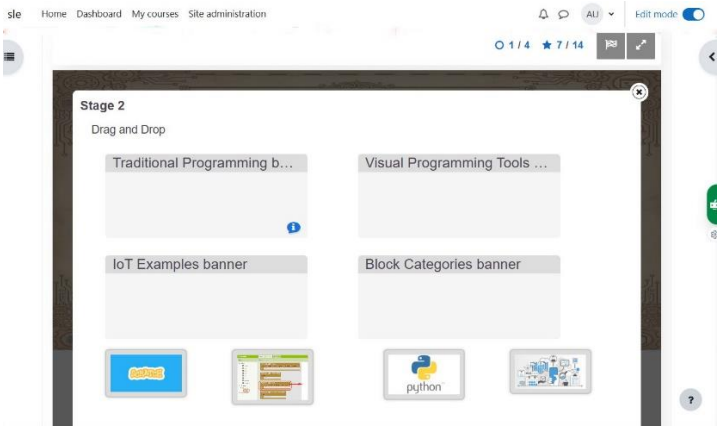


Figure 41. H5p Drag & Drop Image Game

Behavioural settings allow learners to retry the activity, promoting mastery through repeated attempts (see Figure 42.). Scoring is configured to award one point for the entire task, while penalty rules are applied for incorrect placements to ensure accuracy and prevent random guessing. Additionally, score explanations are provided to enhance transparency and support reflection on performance.

Visual and usability features, such as drop zone highlighting during interaction, adjustable background opacity of draggable elements, and automatic alignment, contribute to an intuitive user experience. Fullscreen mode is enabled to facilitate focus and engagement, while score visibility ensures learners are aware of their achievements

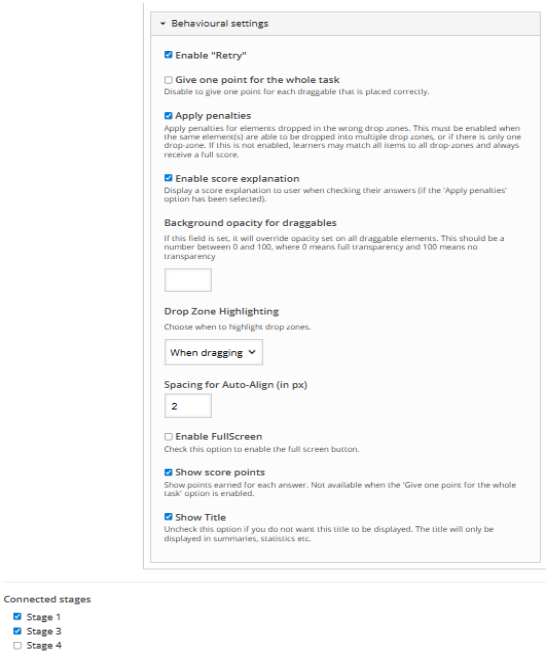


Figure 42. H5p Drag & Drop Image Game Settings

The design follows a step-based approach, beginning with the placement of drop zones, followed by the introduction of draggable elements, and concluding with the validation of correct matches. Learners receive immediate feedback based on their performance, with correct matches highlighted in green (see Figure 43.) and incorrect matches indicated in red (see Figure 44.), thereby facilitating clear identification of errors and supporting the learning process.

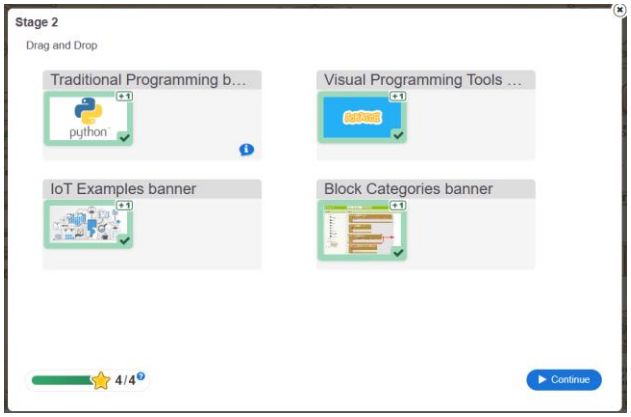


Figure 43. H5p Drag & Drop Image Correct Answer

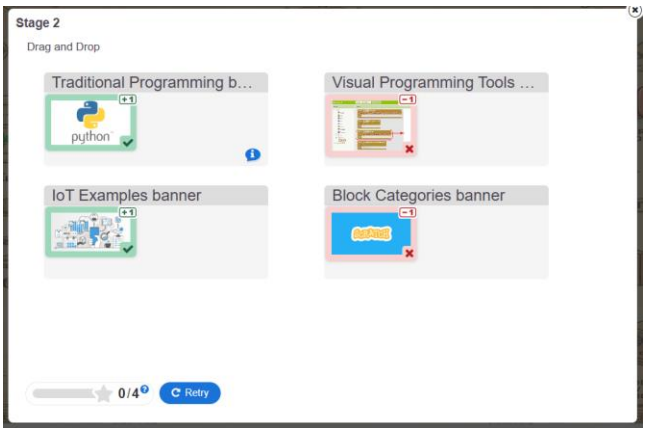


Figure 44. H5p Drag & Drop Image Wrong Answer

Finally, the activity is integrated within a broader learning sequence through its connection to preceding and subsequent stages, supporting a structured and cohesive progression of learning. The next stage introduces learners to foundational concepts in computer science through a memory card matching activity, shown in Figure 45. The exercise is designed to reinforce conceptual understanding by associating key programming terms with their corresponding representations or definitions. It supports cognitive skills such as recall, recognition, and conceptual mapping.

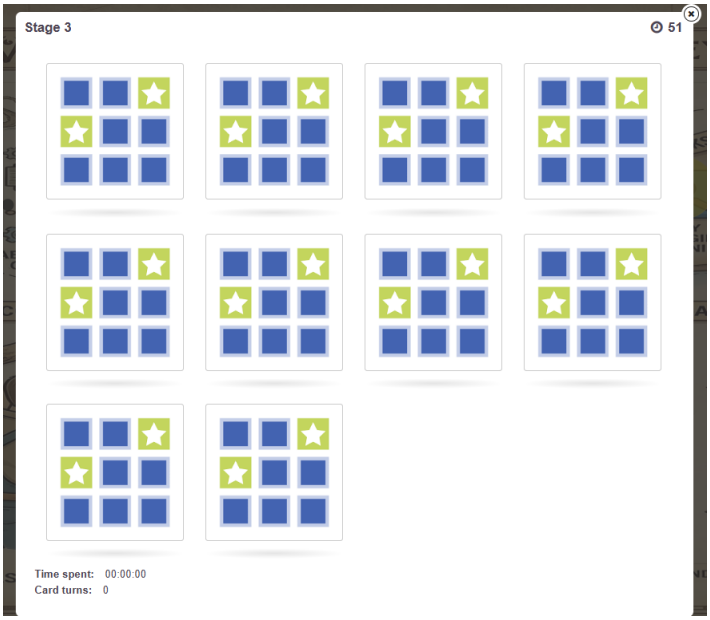


Figure 45. H5p Card Game

The activity is configured as a timed memory card game with a 60-second limit, a retry option, and a balanced square grid layout.

It includes key programming concepts—visual programming, Blockly environment, sequence, conditionals, and loops—each paired with descriptive visuals and explanations to support understanding. Grounded in constructivist learning theory, the activity promotes active engagement and reinforces learning through dual coding, combining visual and verbal elements. Immediate feedback and replay options encourage mastery, while learner performance can be assessed based on accuracy and completion time. More analytically we can Card Game Settings in the following Figures 46 and 47.

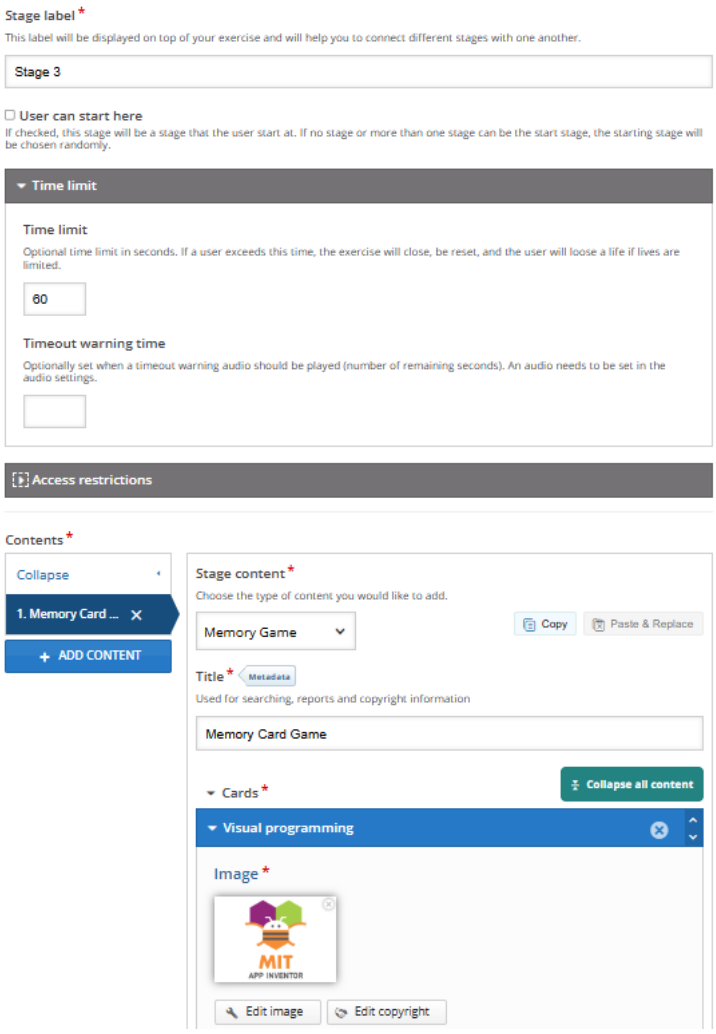


Figure 46. H5p Card Game Part Of Settings

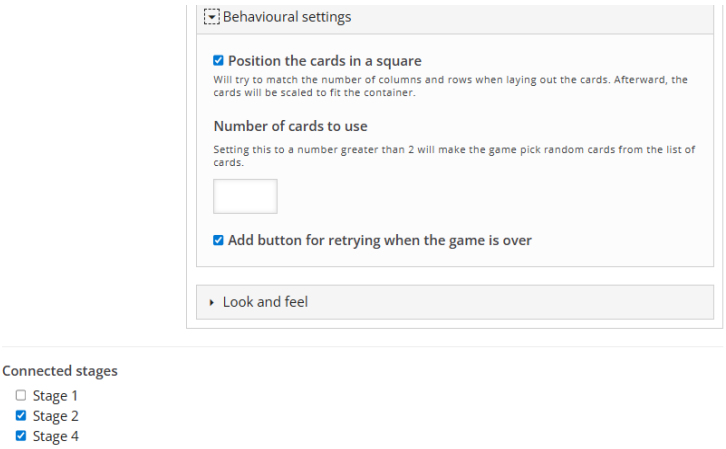


Figure 47. H5p Card Game Part Of Settings

This activity is designed as a memory card exercise comprising a total of ten cards. Participants are required to identify and match five corresponding pairs within a fixed time limit of 60 seconds (see Figure 48.). The task emphasizes both speed and accuracy, encouraging the development of cognitive recall and concentration skills.

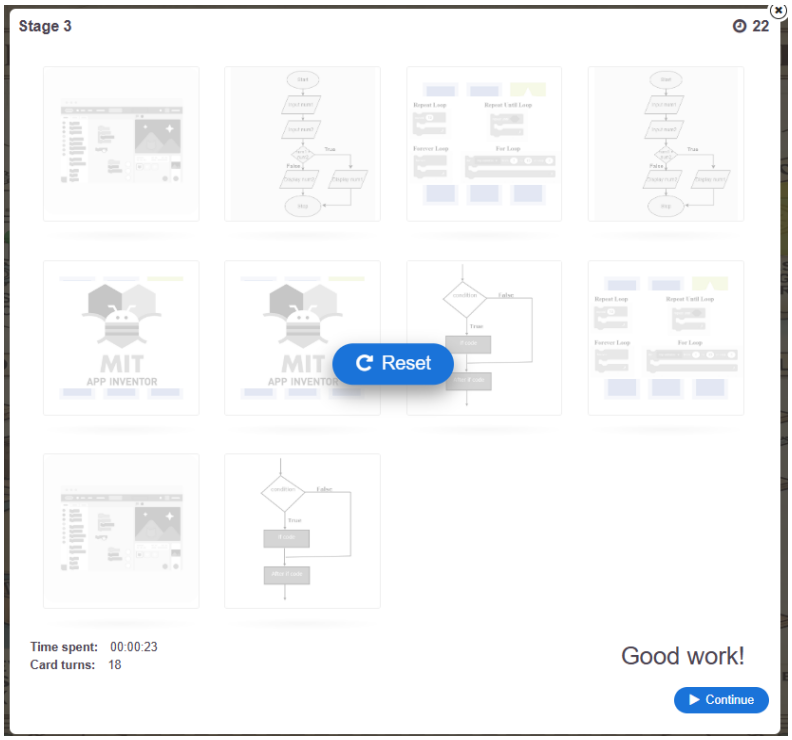


Figure 48. Memory Card Game Completion

If the participant is unable to successfully complete the matching task within the allotted time, a timeout message is displayed (see Figure 49.). In such cases, the activity provides the option to retry, allowing learners to engage with the task repeatedly until successful completion is achieved.

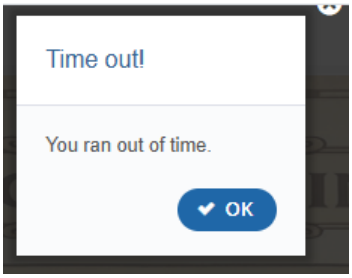


Figure 49. Memory Card Game – Time Out

The puzzle of activities is completed in Stage 4. In this section, there is an interactive video uploaded to YouTube that includes two embedded interactions, shown in Figure 50.



Figure 50. Interactive Video

As the video progresses, initially in the first interaction, learners are prompted to engage with the content through a structured question that introduces key concepts related to visual programming (see Figure 51.).

At a later point in the video (0:22), the second interaction appears as a poster, pausing the video to assess learners' understanding of logical structures in Blockly. Specifically, the activity presents a True/False question focusing on the role of Boolean logic, where conditions are evaluated as either true or false (see Figure 52). This interaction is designed within an academic framework to reinforce fundamental principles of computational thinking and decision-making.

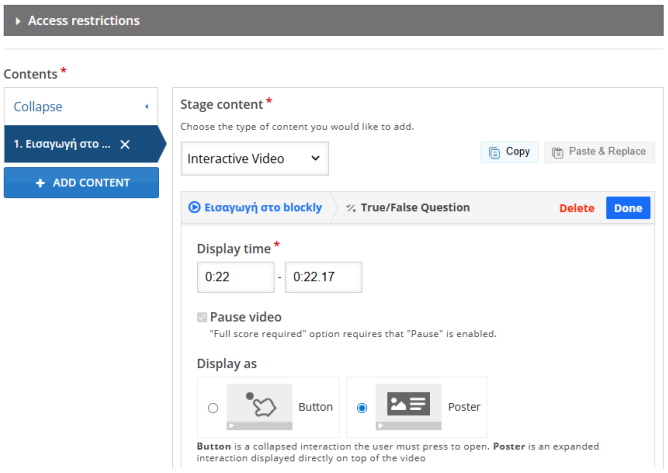


Figure 51. 1st Interaction Video Settings

Learners are required to achieve a full score before proceeding, ensuring mastery of the concept. Immediate feedback is provided: correct answers confirm understanding of Boolean logic, while incorrect responses guide learners to revisit an earlier segment of the video (0:16) for review. The inclusion of retry and solution options supports formative assessment, encourages reflection, and promotes self-regulated learning.

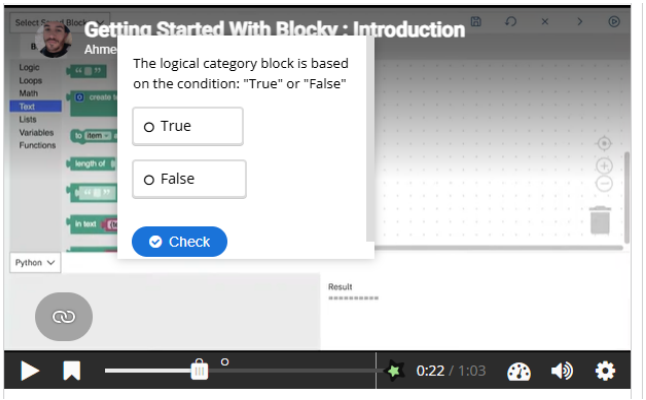


Figure 52. True / False Interaction

This second interaction (see Figure 53.), presented at timestamp 0:28 as a button, requires learners to actively engage with the content through a fill-in-the-missing-words activity (see Figure 54.). The task is designed in an academic style to reinforce understanding of logical structures in Blockly, focusing on the application of Boolean logic and its role in controlling program execution.

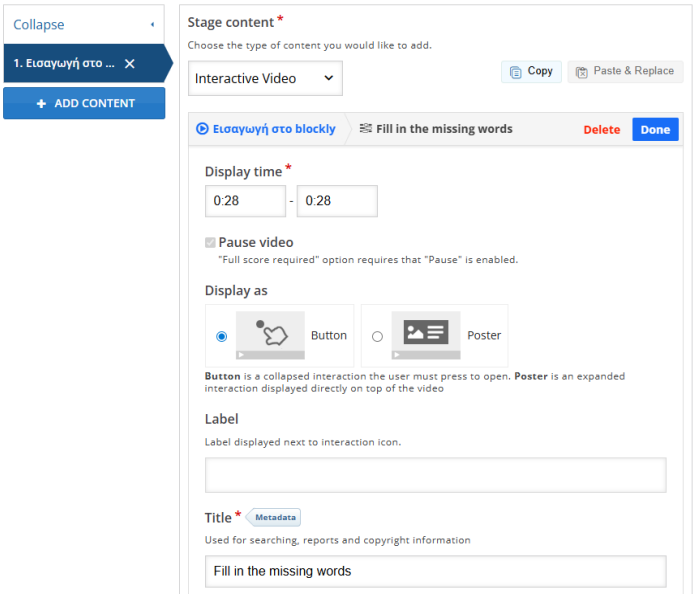


Figure 53. 2nd Interaction Video Settings

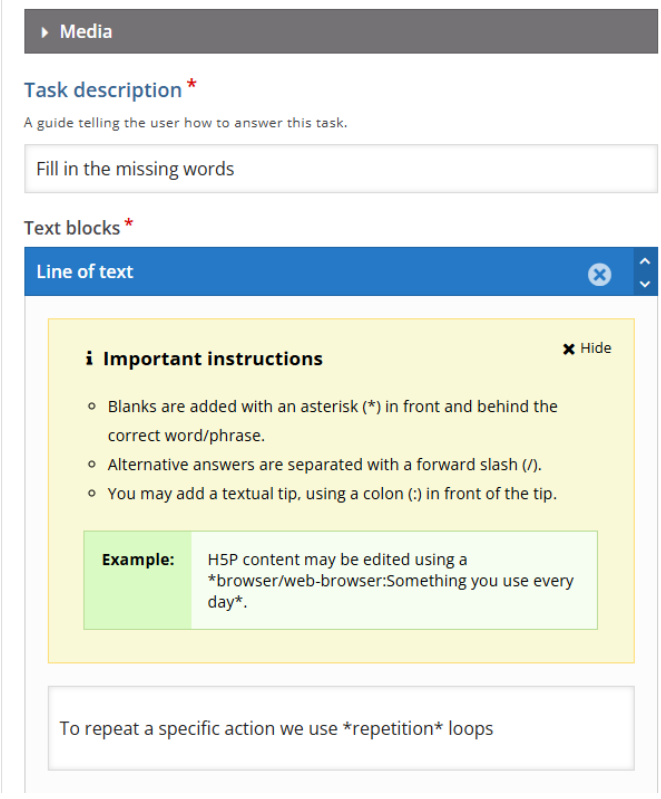


Figure 54. 2nd Interaction Video Settings

Learners are instructed to complete a set of statements by filling in key terms related to the concept as shown in Figures 55 and 56. The activity emphasizes that Blockly uses Boolean values to evaluate conditions as either true or false, and that these conditions are fundamental for determining the flow of a program. Furthermore, it highlights the use of logical blocks within conditional structures, which guide decision-making processes in programming.

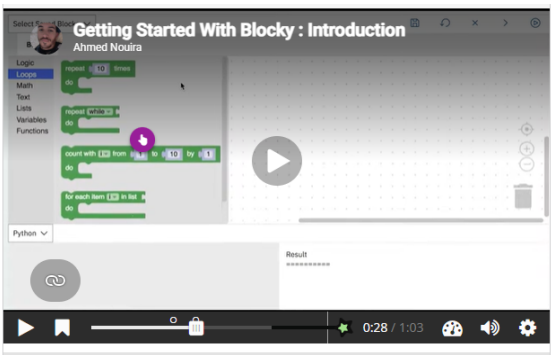


Figure 55. 2nd Interaction button

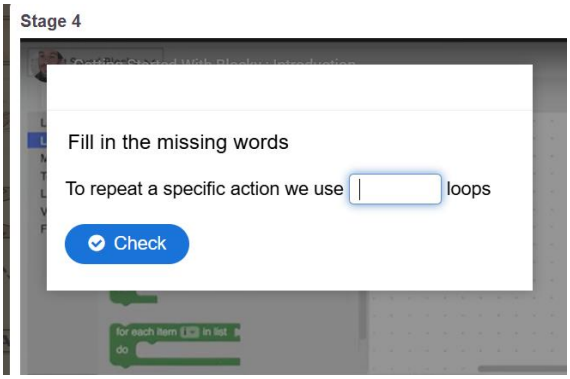


Figure 56. 2nd Interaction Fill in the Blanks

The interaction includes structured feedback based on performance levels, encouraging learners to reflect on their understanding and revisit the instructional material if necessary. Full completion (see Figure 57.), is required before progressing, ensuring that learners achieve a satisfactory level of comprehension. The option to retry and view solutions supports formative assessment, while adaptive settings reinforce learning by preventing advancement without mastery.

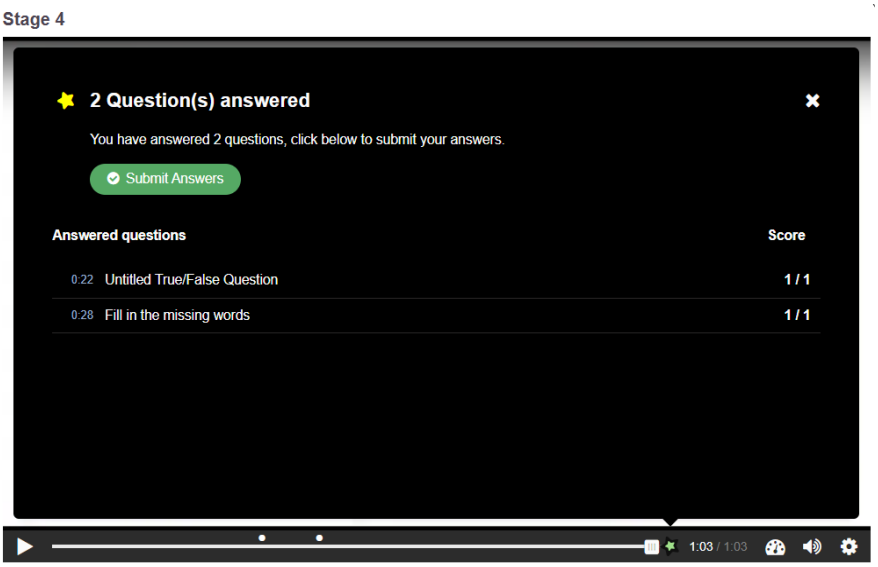


Figure 57. H5p Question Answered Activities

Successful completion of the activity contributes directly to the course competency 'Complexity and Efficiency Awareness (IT-32-D2),' which is automatically marked as achieved upon completion. The retry option permits multiple attempts, and progression between stages is gated by successful completion of the preceding one.

5.4 Summary

This chapter presented the implementation of H5P-based interactive learning components within the Smart Learning Ecosystem, demonstrating how the Conceptualisation stage of the pedagogical workflow was operationalised through interactive activities deployed in the configured Moodle environment. Section 5.2 documented the H5P interactive content layer as the primary student-facing conceptualisation interface, describing how activity types, gradebook synchronisation, and access sequencing were configured to scaffold learner progression. Section 5.3 extended this account to the full range of interactive and adaptive learning activities — including branching lesson design, H5P Game Map activities — illustrating how each component was implemented to support the Conceptualisation and Application stages of the pedagogical workflow.

In the Section 5.4, the preceding sections have documented the full H5P implementation of the SLE, from the configuration of the interactive content layer through to the detailed walkthrough of all six activity types deployed across the four course modules. The design decisions, activity configurations, and their pedagogical alignment with the Conceptualisation stage of the five-stage workflow represent the primary contributions of this chapter. The gamification framework, CodeRunner programming assessments, and analytics infrastructure are each addressed in the chapters that follow.

Chapter 6 - Integration and Code Base Programming of Coderunner

This chapter presents the Blockly visual programming environment and the CodeRunner assessment interface, along with representative exercise walkthroughs covering three exercises of progressively increasing complexity.

The chapter is structured into four sections. Section 6.1 describes the Blockly visual programming environment and the CodeRunner assessment interface, detailing the student-facing workspace design, the progressive block toolbox architecture, and the automated grading and feedback mechanisms that support iterative learning. Section 6.2 presents the full technical architecture of the CodeRunner programming exercises, documenting the question configuration, the Blockly-to-Python integration system, and the test case framework that enables deterministic automated assessment. Section 6.3 presents representative walkthroughs of the three principal CodeRunner exercises — progressing from fundamental arithmetic operations through Arduino hardware control to full IoT Bluetooth integration — demonstrating the system functioning end-to-end under real pilot conditions. Section 6.4 provides a summary of the chapter.

6.1 The Blockly Visual Programming Environment

The Blockly visual programming environment represents the most distinctive student-facing tool in the SLE's pedagogical toolkit. Embedded directly within the CodeRunner question interface in Moodle, Blockly provides students with a drag-and-drop block-based editor through which programs are assembled visually rather than typed syntactically. This design decision addresses one of the most significant barriers to programming engagement in secondary education: the intimidating and error-prone nature of text-based syntax for beginner programmers.When a student opens a CodeRunner programming question that incorporates the Blockly interface, they are presented with a two-panel workspace.

On the left side, a categorised toolbox displays all available block types organised into logical groups. On the right, a blank canvas serves as the programming workspace into which blocks are dragged, snapped together, and configured. As blocks are added or modified, a live code preview area below the workspace continuously displays the Python code equivalent of the current block configuration, providing students with real-time visibility into the relationship between their visual constructions and the underlying textual code.

The block toolbox available to students varies by exercise type, reflecting the progressive complexity of the four course modules. In introductory exercises, students access the standard programming blocks covering control flow, logic, arithmetic, text manipulation, list operations, variable management, and basic input and output. As the course advances into hardware and IoT domains, additional block categories become available: Arduino-style hardware control blocks; sensor blocks for temperature, light, proximity, and motion; connectivity blocks for Bluetooth and HTTP communication; App Inventor-style UI and event blocks; and IoT Smart Home blocks for sensor-actuator integration scenarios.

The workflow for completing a Blockly-based exercise involves four student-facing controls at the bottom of the workspace. The Convert to Python Code button translates the current block configuration into Python and transfers it to the CodeRunner answer field. The Clear button resets the workspace after user confirmation. The Save button exports the current block configuration as an XML file for download and future reuse. The Load and Transfer button reimports a previously saved XML workspace, regenerates the Python code, and transfers it to the answer field — supporting iterative work across multiple sessions.

The pedagogical significance of this visual-to-textual bridging is considerable. Students who might otherwise be deterred by Python syntax errors are able to focus their cognitive effort on program logic and structure, while the live code preview ensures they are simultaneously building familiarity with the textual form of the constructs they are assembling visually. This dual representation aligns with the constructivist orientation of the SLE's pedagogical framework, supporting knowledge construction through active manipulation rather than passive reception.

6.1.1 The CodeRunner Programming Assessment Interface

CodeRunner constitutes the student-facing assessment mechanism for all programming exercises in the SLE. Students encounter CodeRunner activities as embedded quiz questions within Moodle, accessible after completing the prerequisite H5P conceptualisation activities for each module. A CodeRunner exercise presents a clearly structured programming challenge consisting of a problem statement, a description of expected program behaviour, sample input-output examples where applicable, and either a Blockly workspace or a text-based code entry area depending on the exercise type.

Students submit their solutions by clicking the Check button, which triggers immediate automated execution of the submitted code against the predefined test cases. Feedback is returned within seconds, informing the student whether their submission passed all test cases (receiving full marks), passed a subset of test cases (receiving proportional credit), or failed due to syntax errors (returning a structured error message identifying the nature and location of the fault). This immediacy of feedback is a central feature of the student experience: unlike traditional assignment-based assessment where feedback may be delayed by days, CodeRunner enables students to identify and correct errors within the same session, supporting iterative debugging as a normal part of the learning process.

The grading model configured for CodeRunner exercises follows an all-or-nothing approach at the test-case level, with a penalty regime applied for multiple attempts on the same question within a single quiz session. For students who encounter syntax errors, CodeRunner provides structured feedback identifying the type and location of the fault. For students whose code executes without errors but produces incorrect output, the test results panel displays a side-by-side comparison of expected and actual outputs for each failing test case, enabling targeted debugging.

This differentiated feedback architecture supports learners at all stages of debugging without exposing the underlying test case logic.

6.2 CodeRunner Programming Exercises: Technical Architecture

In order to enhance the automated assessment process and improve accessibility for novice programmers, the CodeRunner plugin was extended to support visual programming through integration with Blockly.

The CodeRunner question type (see Figure 58.), was configured to support automated assessment of programming exercises in Python. Specifically, the question type was set to python3, allowing students to submit Python code that is executed and evaluated against predefined test cases. The content describes a highly customised programming question setup designed to ensure strict grading, controlled feedback, and a consistent user experience. The interface was configured to support coding effectively, with an 18-row answer box and an integrated Ace editor that includes usability features like adjustable font size, themes, and live autocompletion. Students could write and run code without interruption, as the stop button was disabled, and no pre-submission feedback was provided due to the precheck option being turned off. Feedback delivery was instead managed at the overall quiz level. Grading followed a strict all-or-nothing approach, where submissions were marked correct only if all test cases were passed. A penalty regime (e.g., 10%, 20%) was applied to reduce marks for multiple attempts. Evaluation relied on exact output matching, ensuring precise correctness of results.

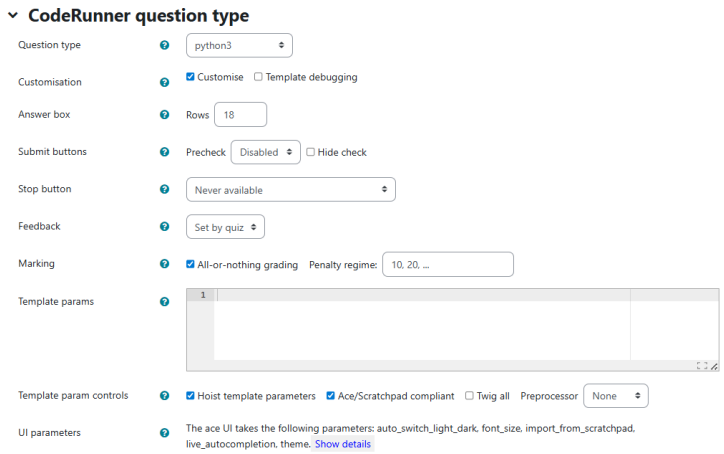


Figure 58. Coderunner question type

CodeRunner question settings under the "Customization" (see Figure 59.), subcategory enable instructors to tailor specific aspects of the automated grading process and code execution environment. This includes defining custom test cases, input/output formats, code templates, and execution parameters. It provides advanced options for customizing how student submissions are evaluated, such as specifying test code, separators, environment settings, or output formatting, allowing for a highly flexible and adaptable assessment setup.

What the top lines do :

- {{ STUDENT_ANSWER }} is the learner's submitted code, inserted directly into the template.
- __student_answer__ = """{{ STUDENT_ANSWER | e('py') }}""" stores the student's code as a Python-escaped multi-line string, so quotes and backslashes are safely handled.

Separator and test-case loop

- SEPARATOR = "#<ab@17943918#@>#" defines a special marker string used to separate outputs from different test cases.
- The {% for TEST in TESTCASES %} ... {% endfor %} block injects each test-case's code (TEST.testcode) into the combined script; if it's not the last test, print(SEPARATOR) is added so the grader can later split the combined output by that marker.

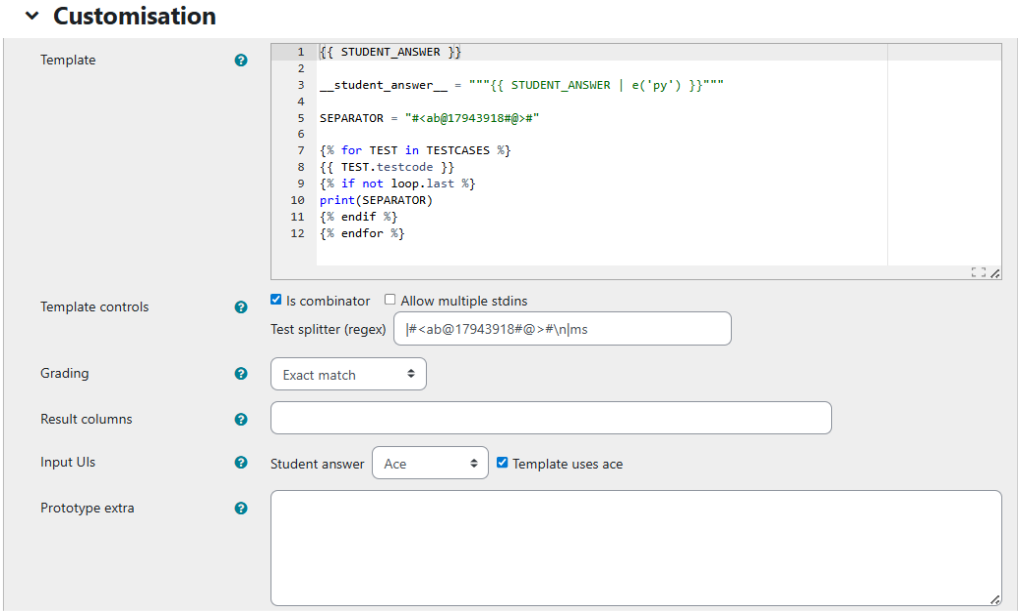


Figure 59. Coderunner customization

In the Moodle CodeRunner environment, the "General" subcategory's "Question text" field is primarily designed for presenting the problem statement to students, rather than for code execution or grading logic. It serves as a rich, pedagogical prompt that combines plain text, HTML formatting, and embedded code snippets to clearly communicate the task, objectives, and expectations. Instructors typically include detailed descriptions, sample inputs and outputs, constraints, hints, and minimal starter code to guide students without revealing internal evaluation mechanisms. Importantly, this field must not contain any CodeRunner template code, such as placeholders or control structures for test cases, which are managed separately in the customization and test configuration sections.

Overall, the “Question text” functions as a human-readable, instructional interface that provides all necessary context for students to understand and approach the problem, maintaining a clear separation from the automated testing and grading components that are handled elsewhere within the system.

Within this Question text area, a Blockly workspace can be embedded as part of the General settings, allowing students to construct programs visually using blocks. These blocks are automatically converted into Python code and submitted as the answer, providing an intuitive interface for programming exercises while keeping all test cases and grading logic separate.

The provided code fragment is an HTML and JavaScript implementation that presents a block-based programming environment built on top of Google’s Blockly library, combined with a Python code generator.

Blockly Integration - System Architecture

The page includes a Blockly workspace, which acts as a visual, drag-and-drop editor for programming blocks. This workspace is embedded inside a and is initialized using the Blockly library loaded from a CDN. Alongside the workspace, the interface provides a fixed toolbox with the following categories of blocks:

- **Control** (control-flow statements: if, loops, etc.)
- **Logic** (boolean expressions, comparisons, negation)
- **Math** (arithmetic operations, random numbers, constants)
- **Text** (string manipulation, concatenation, printing)
- **Lists** (list creation, indexing, slicing, sorting)
- **Variables** and **Procedures** (variable and function definitions)
- **Input/Output** (console input/output blocks)
- **Buttons** (event-based blocks for button interactions)

Initialization and Blockly Setup

The JavaScript code begins by loading the Blockly library and its Python generator module (see Figure 60.), from the following CDN URLs:

```
9     <!-- Blockly Scripts -->
10    <script src="https://unpkg.com/blockly@9.3.3/blockly.min.js"></script>
11    <script src="https://unpkg.com/blockly@9.3.3/python_compressed.js"></script>
```

Figure 60. Blockly Scripts

These resources enable the creation of a block-based editor and the translation of blocks into executable Python code.

A blocklyDiv element is then used to host the Blockly workspace, shown in Figure 61, which is initialized with the defined toolbox, a grid layout, zoom controls, and a built-in trash-can for deleting blocks.

```
13      <!-- Blockly Area-->
14      <div id="blocklyDiv" style="height: 600px; width: 100%;" bis_skin_checked="1"></div>
```

Figure 61. Blockly Area

The script also defines **custom blocks** (see Figure 62**.)** for:

- input_number and input_text (to read numeric and string input from the user),
- button_click, button_press, button_release, and button_when_clicked (to handle button-event logic).

```
<!-- Fixed Toolbox with working blocks -->
<xml id="toolbox" style="display: none">
    <!-- Control -->
    <category name="Control" colour="120">
        <block type="controls_if"></block>
        <block type="controls_if">
            <mutation else="1"></mutation>
        </block>
        <block type="controls_if">
            <mutation elseif="1" else="1"></mutation>
        </block>
        <block type="controls_repeat_ext"></block>
        <block type="controls_whileUntil"></block>
        <block type="controls_for"></block>
        <block type="controls_forEach"></block>
        <block type="controls_flow_statements"></block>
    </category>

    <!-- Logic -->
    <category name="Logic" colour="210">
        <block type="logic_compare"></block>
        <block type="logic_operation"></block>
        <block type="logic_negate"></block>
        <block type="logic_boolean"></block>
        <block type="logic_null"></block>
        <block type="logic_ternary"></block>
    </category>
```

```
    <!-- Math -->
    <category name="Math" colour="230">
        <block type="math_number"></block>
        <block type="math_arithmetic"></block>
        <block type="math_single"></block>
        <block type="math_trig"></block>
        <block type="math_constant"></block>
        <block type="math_round"></block>
        <block type="math_modulo"></block>
        <block type="math_constrain"></block>
        <block type="math_random_int"></block>
        <block type="math_random_float"></block>
    </category>

    <!-- Text -->
    <category name="Text" colour="160">
        <block type="text"></block>
        <block type="text_join"></block>
        <block type="text_append"></block>
        <block type="text_length"></block>
        <block type="text_isEmpty"></block>
        <block type="text_indexOf"></block>
        <block type="text_charAt"></block>
        <block type="text_getSubstring"></block>
        <block type="text_changeCase"></block>
        <block type="text_trim"></block>
        <block type="text_print"></block>
    </category>
    <!-- Lists -->
    <category name="Lists" colour="260">
        <block type="lists_create_empty"></block>
        <block type="lists_create_with"></block>
        <block type="lists_repeat"></block>
        <block type="lists_length"></block>
        <block type="lists_isEmpty"></block>
        <block type="lists_indexOf"></block>
        <block type="lists_getIndex"></block>
        <block type="lists_setIndex"></block>
        <block type="lists_getSublist"></block>
        <block type="lists_split"></block>
        <block type="lists_sort"></block>
        <block type="lists_reverse"></block>
    </category>

    <!-- Variables -->
    <category name="Variables" custom="VARIABLE" colour="330"></category>

    <!-- Procedures -->
    <category name="Procedures" custom="PROCEDURE" colour="290"></category>

    <!-- Input/Output -->
    <category name="Input/Output" colour="65">
        <block type="text_print"></block>
        <block type="input_number"></block>
        <block type="input_text"></block>
    </category>

    <!-- Button Events -->
    <category name="Buttons" colour="300">
        <block type="button_click"></block>
        <block type="button_press"></block>
        <block type="button_release"></block>
        <block type="button_when_clicked"></block>
    </category>
</xml>
```

Figure 62. Toolbox Area – Basic Blocks

In addition to the basic block categories (such as Control, Math, Logic, and Text), additional categories can be incorporated to further enhance the Blockly environment. This extended set of blocks moves beyond purely algorithmic logic into physical and networked computing domains, supporting areas such as robotics, IoT, mobile-like user interfaces, and sensor-based applications within an educational context.

1. App Inventor– Style UI and Event Blocks

This category emulates **App Inventor–style component blocks** for simple mobile-like interfaces and event-handling logic.

Label components

- app_label_set_text, app_label_get_text: set and read the text displayed on a label.
- app_label_set_color, app_label_set_size: change the label's foreground color and font/text size dynamically.

Notifier / user-feedback components

- app_notifier_show_message, app_notifier_show_alert: display simple messages or modals to the user.
- app_notifier_show_choose_dialog: show a dialog with choices (e.g., "Yes/No") and route the program based on the user's selection.

Bluetooth communication components

- app_bluetooth_send_text, app_bluetooth_receive_text: transmit and receive text messages over Bluetooth.
- app_bluetooth_is_connected: check whether the Bluetooth connection is active, enabling conditional logic around connectivity.

Button-based event blocks

- app_button_when_clicked: define an action that runs when a specific button is clicked.
- app_led_on_button_click, app_led_off_button_click, app_temp_button_click, app_light _button_click: specialized button-event blocks that simulate turning on/off an LED, reading temperature, or reading light level when a button is pressed.

These blocks together provide a **visual, App-Inventor-inspired layer** for building simple interactive applications with UI elements, user feedback dialogs, and basic Bluetooth I/O, shown in the below Figure 63.

```
<!-- App Inventor Components -->
<category name="App Inventor" colour="200">
    <!-- Label Components -->
    <block type="app_label_set_text"></block>
    <block type="app_label_get_text"></block>
    <block type="app_label_set_color"></block>
    <block type="app_label_set_size"></block>

    <!-- Notifier Components -->
    <block type="app_notifier_show_message"></block>
    <block type="app_notifier_show_alert"></block>
    <block type="app_notifier_show_choose_dialog"></block>

    <!-- Bluetooth Components -->
    <block type="app_bluetooth_send_text"></block>
    <block type="app_bluetooth_receive_text"></block>
    <block type="app_bluetooth_is_connected"></block>

    <!-- Button Events -->
    <block type="app_button_when_clicked"></block>

    <!-- App Inventor Specific Buttons -->
    <block type="app_led_on_button_click"></block>
    <block type="app_led_off_button_click"></block>
    <block type="app_temp_button_click"></block>
    <block type="app_light_button_click"></block>
</category>
```

Figure 63. App Inventor Components Blocks

2. Connectivity Blocks

This category focuses on **communication channels** between the device and external systems via serial, Bluetooth, and HTTP.

Serial communication

- serial_begin, serial_available, serial_read, serial_write: initialize the serial port, check for available data, and read/write bytes or strings.
- serial_print, serial_println: write formatted output to the serial port, useful for debugging or logging.

Bluetooth Client

- bluetooth_client_connect, bluetooth_client_disconnect: establish and close a client-side Bluetooth connection.
- bluetooth_client_send, bluetooth_client_receive: send and receive data over the Bluetooth link.
- bluetooth_client_connected: a boolean-style block that indicates whether the client is currently connected, enabling conditional communication logic.

Web/HTTP operations

- web_http_get, web_http_post: perform HTTP GET and POST requests to web servers or APIs, allowing the program to send or receive data from remote services (e.g., cloud APIs, web-based dashboards).

These Connectivity blocks (see Figure 64), allow students to build **network-aware programs** that interact with other devices, sensors, and web services, bridging embedded or microcontroller-based logic with higher-level networked systems.

```
<!-- ============ CONNECTIVITY ============ -->
<category name="Connectivity" colour="45">
    <!-- Serial Communication -->
    <block type="serial_begin"></block>
    <block type="serial_available"></block>
    <block type="serial_read"></block>
    <block type="serial_write"></block>
    <block type="serial_print"></block>
    <block type="serial_println"></block>

    <!-- Bluetooth Client -->
    <block type="bluetooth_client_connect"></block>
    <block type="bluetooth_client_send"></block>
    <block type="bluetooth_client_receive"></block>
    <block type="bluetooth_client_disconnect"></block>
    <block type="bluetooth_client_connected"></block>

    <!-- Web/HTTP -->
    <block type="web_http_get"></block>
    <block type="web_http_post"></block>
</category>
```

Figure 64. Connectivity Blocks

3. Sensors Blocks

This category encapsulates **sensor-reading operations** for a variety of physical and environmental quantities.

Proximity sensor

- sensor_proximity_read: read a raw proximity value.
- sensor_proximity_detect: return a boolean-style result indicating whether an object is detected nearby.
- sensor_proximity_distance: report an estimated distance to the detected object.

Light sensor

- sensor_light_read: read the raw light-level value.
- sensor_light_level: often maps the raw value to a more intuitive "level" (e.g., low/medium/high) for decision-making.

Location / GPS

- sensor_location_get: obtain a full location record (if available).
- sensor_location_latitude, sensor_location_longitude: extract latitude and longitude components for use in mapping or geo-aware logic.

Accelerometer

- sensor_accelerometer_read: read the current acceleration vector.
- sensor_accelerometer_x, _y, _z: extract individual axis components for motion detection or orientation-based logic.

Temperature & motion

- sensor_temperature_read: read the ambient temperature.
- sensor_motion_detect: detect motion events (e.g., from a PIR sensor), returning a boolean-style result that can trigger alarms or logging.

These Sensor blocks (see Figure 65.), support **context-aware applications** such as environmental monitoring, motion-triggered alarms, location-based services, and simple human-interaction devices.

```
<!-- ============ SENSORS ============ -->
<category name="Sensors" colour="95">
    <!-- Proximity Sensor -->
    <block type="sensor_proximity_read"></block>
    <block type="sensor_proximity_detect"></block>
    <block type="sensor_proximity_distance"></block>

    <!-- Light Sensor -->
    <block type="sensor_light_read"></block>
    <block type="sensor_light_level"></block>

    <!-- Location/GPS -->
    <block type="sensor_location_get"></block>
    <block type="sensor_location_latitude"></block>
    <block type="sensor_location_longitude"></block>

    <!-- Accelerometer -->
    <block type="sensor_accelerometer_read"></block>
    <block type="sensor_accelerometer_x"></block>
    <block type="sensor_accelerometer_y"></block>
    <block type="sensor_accelerometer_z"></block>

    <!-- Temperature -->
    <block type="sensor_temperature_read"></block>

    <!-- Motion/PIR -->
    <block type="sensor_motion_detect"></block>
</category>
```

Figure 65. Sensors Blocks

4. Arduino-Style Hardware Blocks

This category provides **low-level Arduino-compatible blocks** for direct control of digital and analog pins, timing, and basic peripherals.

Pin management

- arduino_pin_mode: set a pin to INPUT / OUTPUT / INPUT_PULLUP.
- arduino_digital_write, arduino_digital_read: write high/low values to, or read the state of, a digital pin.

Analog I/O

- arduino_analog_read, arduino_analog_write: read analog sensor values or output PWM signals for controlling LEDs, motors, or servos.

Timing and delay

- arduino_delay: pause execution for a fixed number of milliseconds.
- arduino_millis: return the current time in milliseconds since startup, useful for non-blocking timers and debouncing.

Serial and sound

- arduino_serial_print, arduino_serial_println: send text or numeric data to the serial monitor for debugging.
- arduino_tone: generate a tone on a specified pin, e.g., for simple audio feedback.

Servo and constants

- arduino_servo_attach, arduino_servo_write: attach and position a servo motor to a desired angle.
- arduino_constant, arduino_pin_constant: define named constants for pins or values, improving code readability.

These Arduino blocks (see Figure 66.), enable students to implement **close-to-the-metal microcontroller logic**, including GPIO control, timing, serial logging, and simple actuator control (LEDs, servos, buzzers).

```
<!-- Arduino Blocks -->
<category name="Arduino" colour="120">
    <block type="arduino_pin_mode"></block>
    <block type="arduino_digital_write"></block>
    <block type="arduino_digital_read"></block>
    <block type="arduino_analog_read"></block>
    <block type="arduino_analog_write"></block>
    <block type="arduino_delay"></block>
    <block type="arduino_millis"></block>
    <block type="arduino_serial_print"></block>
    <block type="arduino_serial_println"></block>
    <block type="arduino_servo_attach"></block>
    <block type="arduino_servo_write"></block>
    <block type="arduino_tone"></block>
    <block type="arduino_constant"></block>
    <block type="arduino_pin_constant"></block>
</category>
```

Figure 66. Arduino Blocks

5. IoT Smart Home Blocks

This category abstracts patterns for **IoT-style "smart home" scenarios**, combining sensors, actuators, and wireless communication.

Device control

- iot_led_control: encapsulates LED on/off or brightness control, often mapped to a virtual or physical LED in a smart-home demo.
- iot_temperature_read, iot_light_read: read environment temperature and light level as part of a smart-home sensing pipeline.

Bluetooth I/O (IoT-oriented)

- iot_bluetooth_send, iot_bluetooth_receive: send and receive data over Bluetooth within a smart-home context (e.g., between a hub and sensor nodes).

Architectural blocks

- iot_smart_home_setup: a setup block that initializes the smart-home environment (e.g., configuring pins, sensors, and network connections).
- iot_main_loop: a high-level "main loop" block that wraps the continuous sensing-and-control logic of the IoT system, helping students structure their program around a loop-based runtime model.

Together, these IoT Smart Home blocks (see Figure 67.), provide a **domain-specific abstraction layer** that simplifies the construction of prototype smart-home systems, where the focus shifts from low-level wiring details toward higher-level control policies and sensor-actuator interactions.

```
<!-- IoT Smart Home Blocks -->
<category name="IoT Smart Home" colour="50">
    <block type="iot_led_control"></block>
    <block type="iot_temperature_read"></block>
    <block type="iot_light_read"></block>
    <block type="iot_bluetooth_send"></block>
    <block type="iot_bluetooth_receive"></block>
    <block type="iot_smart_home_setup"></block>
    <block type="iot_main_loop"></block>
</category>
```

Figure 67. IoT Smart Home Blocks

For each of these custom blocks, **Python generators** are implemented, mapping the visual blocks into Python code.

This allows students or users to visually construct programs that are then translated into syntactically correct Python code.

Event Handling and Code Generation

The script includes a generateCode() function (see Figure 68.), that calls Blockly's Python generator to convert the current workspace into Python source code and display it inside the #codeOutput element on the page. Any change in the workspace (block creation, deletion, movement, or parameter editing) triggers this function, providing real-time feedback.

```
function generateCode() {
    if (!workspace) {
        console.log("❌ generateCode: No workspace available");
        return '';
    }

    try {
        console.log("🔧 generateCode: Starting code generation...");

        Blockly.Python.init(workspace);
        let code = Blockly.Python.workspaceToCode(workspace);

        // Clean up code
        code = code.replace(/\b(\w+) = None\s+\1 =/g, "$1 =");

        console.log("🔧 generateCode: Raw generated code:", code);
        console.log("🔧 generateCode: Code length:", code.length);

        const outputElement = document.getElementById('codeOutput');
        if (!outputElement) {
            console.error("❌ generateCode: Output element not found!");
            return code;
        }

        // Force update the display
        if (code && code.trim()) {
            outputElement.textContent = code;
            outputElement.style.display = 'block';
            console.log("☑ generateCode: Code displayed in output area");
        } else {
            outputElement.textContent = '# Κενό workspace - προσθέστε blocks για να δείτε κώδικα';
            console.log("ℹ️ generateCode: No code to display, showing default message");
        }

        return code;

    } catch (e) {
        console.error("❌ generateCode error:", e);
        const outputElement = document.getElementById('codeOutput');
        if (outputElement) {
            outputElement.textContent = 'Σφάλμα: ' + e.message;
        }
        return '';
    }
}
```

Figure 68. generateCode ()

A second function, generateAndTransferCode(), shown in Figure 69, attempts to locate an answer input field (typically a <textarea>) on the page—recognized by selectors such as textarea[name*='answer']—which is commonly used in systems like **Moodle with CodeRunner**.

If such a field is found, the generated Python code is:

- written into the textarea,
- and followed by a sequence of DOM events (focus, input, change, blur, etc.) to ensure the external application registers the update.

If no suitable answer field is detected, the script can either copy the code to the clipboard (using the navigator.clipboard API) or display it in a popup window with a copy-to-clipboard button.

```
function generateAndTransferCode(event, isAutomatic = false) {
    if (event) {
        event.preventDefault();
    }

    if (!workspace) {
        if (!isAutomatic) alert("Blockly workspace not initialized!");
        return false;
    }

    try {
        console.log("Generating code from workspace...");

        Blockly.Python.init(workspace);
        let code = Blockly.Python.workspaceToCode(workspace);

        // Clean up code
        code = code.replace(/\b(\w+) = None\s+\1 =/g, "$1 =");

        if (!code.trim()) {
            if (!isAutomatic) alert("Δεν υπάρχει κώδικας για μεταφορά!");
            return false;
        }

        console.log("Generated code:", code);

        // ENHANCED search for Moodle/CodeRunner answer fields
        const possibleSelectors = [
            ".que.coderunner textarea[name*='answer']",
            ".que.coderunner textarea",
            "textarea[name='answer']",
            "textarea[name*='answer']",
            "textarea[id*='answer']",
            ".que.coderunner .answer textarea",
            "#id_answer",
            "[name^='q'][name*='answer']",
            "[id^='id_q'][id$='_answer']",
            ".answer textarea",
            "textarea.form-control",
            "textarea"
        ];
```

Figure 69. Code Segment of generateAndTransferCode ()

Furthermore the Answer field (see Figure 70.), in CodeRunner settings represents the input area where the student's solution is submitted for evaluation.

In this implementation, the field is automatically populated with the Python code generated from the Blockly workspace, ensuring a seamless transition from visual programming to executable code.

The system locates the appropriate answer textarea dynamically and inserts the generated code while triggering the necessary events so that CodeRunner properly detects and processes the submission. This approach enhances usability by reducing manual input while preserving compatibility with the platform's automated grading and testing mechanisms.

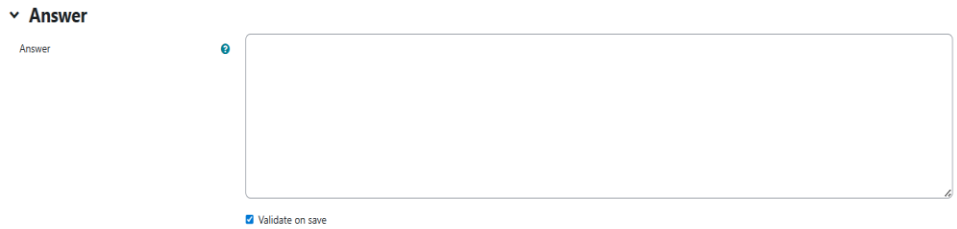


Figure 70. Answer section

Furthermore, the Test cases section (see Figure 71.), in CodeRunner settings defines the automated evaluation criteria used to assess the correctness of the student's submitted code. In this section, instructors specify multiple input-output pairs or expected behaviors against which the generated Python code is executed.

Each test case operates independently, allowing for comprehensive validation of different scenarios, edge cases, and constraints of the problem. This structured approach ensures objective and consistent grading, as the student's answer—whether written manually or generated through the Blockly interface—is systematically tested without exposing the underlying evaluation logic within the Question text or Answer fields.

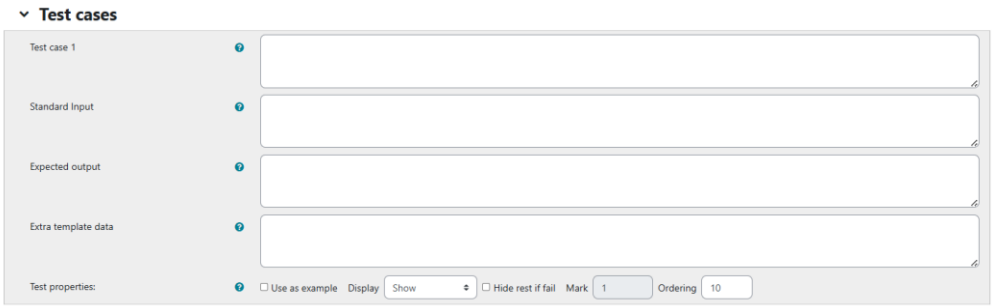


Figure 71. Test cases section

CodeRunner - Test Case Configuration

The implemented environment supports the following main workflows:

- **Block-based programming**: Users can drag blocks from the toolbox into the workspace and visually construct programs. The generateCode() function continuously updates the displayed Python code, allowing learners to see the textual representation of their visual logic.
- **Workspace persistence**: The **"Save Workspace"** button exports the current block configuration as a structured .xml file, which preserves the entire workspace (including block positions and connections). The **"Load & Transfer"** button allows users to later load such an .xml file, reconstruct the original block layout, automatically regenerate the corresponding Python code, and (if possible) transfer it into the active answer field of an online assessment system.
- **Integration with online assessment platforms**: The script is designed to be used in an external browser environment that interacts with platforms such as **Moodle/CodeRunner**, where programming tasks are administered via web forms. It can load a Blockly workspace from an .xml file, translate it into Python, and inject the resulting code into the appropriate answer textarea, effectively bridging a visual, block-based editor with a traditional text-based code submission system.

Test cases are the core evaluation mechanism within CodeRunner. Each test case specifies standard input values, expected output, display and grading properties, and sequencing rules. The SLE deployed test cases with the following configuration properties, as illustrated in the below Table 18:

Table 18. CodeRunner Test Case Settings

Property	Value / Description
Test Case	Name of each test case
Standard Input	The values provided to the program as stdin (e.g. 3 and 5).
Expected Output	The results, each on a separate line.
Extra Template Data	Additional metadata or configuration passed to the template, if applicable.
Use as Example	Enabled — this test case is shown to the student as an example.
Display	Show — the test case and its result are fully visible to the student.
Hide if Fail	The test result is hidden if the student's answer fails this case.
Hide if Succeed	The test result is hidden if the student's answer passes this case.
Hide Rest if Fail	All subsequent test cases are hidden if this one fails.
Mark	The number of marks awarded for passing this test case.
Ordering	Defines the sequence in which this test case is evaluated relative to others.

Each question included structured test cases with defined inputs and expected outputs. These could be sequenced so that failing an early test prevented later ones from running, helping students focus on debugging step by step. Multiple test scenarios could also be handled using a custom test splitter based on regular expressions. Additional configuration features enhanced

flexibility. Template parameters and hoisted controls allowed dynamic adjustments, while full Twig templating enabled the generation of dynamic content.
The system supported multiple standard inputs and combinator templates, enabling more complex problem designs without requiring preprocessing.
Finally, general settings such as category assignment, a default mark of 20 points, and readiness status ensured the question fit smoothly into the course assessment framework. Metadata like versioning, tagging, and file attachment support further improved organisation and management within the platform.

To implement this functionality, custom code modifications were introduced within the CodeRunner plugin. These modifications allow the embedding of a Blockly workspace directly into the question interface of Moodle. The embedded environment provides students with interactive blocks representing fundamental programming constructs, such as variables, loops, conditionals, and input/output operations. In addition to these core elements, the environment was extended to include specialized blocks related to Internet of Things (IoT) applications and Arduino-based programming, enabling students to engage with more applied and hardware-oriented problem-solving scenarios.

Furthermore, the implementation was adapted to support seamless interaction between the visual programming interface and the CodeRunner evaluation mechanism. Specifically, the system was configured so that, upon selecting the *"Transfer to CodeRunner"* button, the block-based solution developed within the workspace is automatically translated into equivalent Python code and inserted into the *answer* field used by Moodle.

To enhance usability and support the student workflow, a set of functional controls was added at the end of the Blockly workspace. These controls include:

- 🚀 **Convert to Python Code** – Transfers the generated Python code to the Moodle answer field, enabling submission and evaluation,
- 🗑️ **Clear** – Resets the workspace after user confirmation,
- 💾 **Save** – Exports the current workspace configuration as an XML file for later use,

- 📁 **Load & Transfer** – Imports a previously saved XML file to restore the workspace state and automatically converts it to Python code.

These additions improve the overall user experience by facilitating code management, enabling persistence of work, and ensuring smooth integration with the automated assessment pipeline. The core of this integration lies in the communication between the Blockly interface and the CodeRunner evaluation engine. When a student completes a solution using visual blocks, the system utilizes Blockly's code generation capabilities to automatically convert the block-based representation into equivalent Python code.

This generated code is then injected into the CodeRunner submission workflow without requiring any additional action from the student.

At this point, CodeRunner processes the generated Python code in the same way as a traditional text-based submission.

The code is executed against a predefined set of instructor-defined test cases, ensuring that the solution is evaluated based on correctness and expected output. Immediate feedback is then provided to the student, supporting iterative learning and self-correction.

To support this functionality, specific implementation details were required, including:

- Embedding the Blockly editor within the question template,
- Handling the serialization and transfer of block-based solutions,
- Generating syntactically correct Python code from Blockly blocks,
- Integrating the generated code into the CodeRunner execution pipeline.

The above implementation illustrates how the Blockly workspace is initialized and how the generated Python code is captured and passed to the answer field expected by CodeRunner. This ensures compatibility with the existing grading mechanism while extending its functionality to support visual programming.

This integration offers significant pedagogical advantages. It lowers the barrier to entry for beginners by minimizing syntax errors, promotes computational thinking, and provides a smooth transition from visual to text-based programming. At the same time, it preserves the robustness and reliability of automated assessment through CodeRunner's testing framework.

Overall, the combination of CodeRunner and Blockly within Moodle establishes a hybrid assessment environment that supports diverse learning styles while maintaining rigorous evaluation standards.

Representative Exercise Examples

CodeRunner programming exercises formed the assessment backbone of the Application and Evaluation stages across all four course modules. A total of 3 CodeRunner questions were developed, organized into Moodle quizzes.

Questions were authored using Moodle's question bank and assigned to quizzes via the specific question selection mechanism, enabling structured assessment delivery.

Each CodeRunner question was developed according to the following structure: a problem statement describing the programming task in plain language, an illustrative example providing sample input and expected output, a code entry field pre-populated with a function or class skeleton where appropriate, and a set of hidden test cases against which learner submissions were evaluated. Test cases were designed to cover both typical and boundary-case inputs, ensuring that correct solutions were evaluated robustly and that partial solutions received appropriately calibrated partial credit.

Question Type Configuration

Questions were implemented using the following CodeRunner question types, selected based on the programming language and task category:

- **python3:** Used for the majority of programming tasks, including sensor data processing, conditional logic exercises, and IoT simulation scenarios.
- **Blockly-to-Python (custom template):** Utilized for visual programming tasks in which learners constructed programs using Blockly block interfaces that were translated to Python for automated execution and evaluation.

The practical training quiz, titled Coderunner – Practical Training, was configured as the primary vehicle for formative programming assessment and was designated as the trigger activity for the quiz submission XP award rule within the gamification framework. Upon each submission, five experience points were automatically allocated to the learner's XP total, reinforcing iterative engagement with programming exercises.

Following the description of the CodeRunner question design and configuration, it is important to illustrate how these elements were implemented in practice.

The following section presents representative examples of CodeRunner programming exercises used in the course, accompanied by figures of the corresponding code, problem statements, and feedback mechanisms. These examples aim to demonstrate the structure of the questions, the types of learner interactions supported, and the evaluation logic applied through automated test cases. By examining these practical instances, the alignment between the pedagogical objectives and the technical implementation of the assessment activities becomes more explicit.

6.3 Representative CodeRunner Exercise Walkthroughs

CodeRunner programming exercises formed the assessment backbone of the Application and Evaluation stages across all four course modules. This section presents a complete walkthrough of each of the three principal exercises, demonstrating the end-to-end operation of the Blockly-CodeRunner integration under actual pilot conditions: from visual program construction and code generation through automated transfer, execution, test validation, and error detection and correction.

The three exercises represent a deliberate pedagogical progression: Exercise 1 introduces fundamental arithmetic operations using standard Blockly blocks; Exercise 2 extends the toolbox with Arduino hardware control; Exercise 3 represents the most complex scenario, combining Bluetooth communication, sensor reading, and IoT control within the full IoT edition of the toolbox.

6.3.1 Exercise 1 — Advanced Basic Operations

The first practical exercise introduces learners to fundamental programming concepts, including variable definition, arithmetic operations, and output display.

This task was designed as an introductory activity to familiarize students with the CodeRunner environment and the process of writing and submitting executable code.

Learners were required to define two variables, number1 and number2, perform basic arithmetic operations (sum, difference, and product), and display the computed results. This exercise uses the standard Blockly toolbox with control, logic, math, text, lists, variables, procedures, input/output, and button block categories.

The problem statement guided students to implement a simple program structure while reinforcing core computational thinking skills.

System Architecture — Exercise 1

Here's a breakdown of the main sections of this Blockly CodeRunner exercises code:

1. HTML Structure & Toolbox

The page renders a visual block-based programming interface using the Blockly library (loaded from a CDN), shown in Figure 72. The <xml id="toolbox"> defines the palette of available block categories (see Figure 73.): Control, Logic, Math, Text, Lists, Variables, Procedures, Input/Output, and Buttons. Each <category> holds specific block types the user can drag into the workspace.

```
9     <!-- Blockly Scripts -->
10    <script src="https://unpkg.com/blockly@9.3.3/blockly.min.js"></script>
11    <script src="https://unpkg.com/blockly@9.3.3/python_compressed.js"></script>
```

Figure 72. Blockly Scripts

```
<!-- Fixed Toolbox with working blocks -->
    <xml id="toolbox" style="display: none">
        <!-- Control -->
        <category name="Control" colour="120">
            <block type="controls_if"></block>
            <block type="controls_if">
                <mutation else="1"></mutation>
            </block>
            <block type="controls_if">
                <mutation elseif="1" else="1"></mutation>
            </block>
            <block type="controls_repeat_ext"></block>
            <block type="controls_whileUntil"></block>
            <block type="controls_for"></block>
            <block type="controls_forEach"></block>
            <block type="controls_flow_statements"></block>
        </category>

        <!-- Logic -->
        <category name="Logic" colour="210">
            <block type="logic_compare"></block>
            <block type="logic_operation"></block>
            <block type="logic_negate"></block>
            <block type="logic_boolean"></block>
            <block type="logic_null"></block>
            <block type="logic_ternary"></block>
        </category>
```

```
    <!-- Math -->
    <category name="Math" colour="230">
        <block type="math_number"></block>
        <block type="math_arithmetic"></block>
        <block type="math_single"></block>
        <block type="math_trig"></block>
        <block type="math_constant"></block>
        <block type="math_round"></block>
        <block type="math_modulo"></block>
        <block type="math_constrain"></block>
        <block type="math_random_int"></block>
        <block type="math_random_float"></block>
    </category>

    <!-- Text -->
    <category name="Text" colour="160">
        <block type="text"></block>
        <block type="text_join"></block>
        <block type="text_append"></block>
        <block type="text_length"></block>
        <block type="text_isEmpty"></block>
        <block type="text_indexOf"></block>
        <block type="text_charAt"></block>
        <block type="text_getSubstring"></block>
        <block type="text_changeCase"></block>
        <block type="text_trim"></block>
        <block type="text_print"></block>
    </category>

    <!-- Lists -->
    <category name="Lists" colour="260">
        <block type="lists_create_empty"></block>
        <block type="lists_create_with"></block>
        <block type="lists_repeat"></block>
        <block type="lists_length"></block>
        <block type="lists_isEmpty"></block>
        <block type="lists_indexOf"></block>
        <block type="lists_getIndex"></block>
        <block type="lists_setIndex"></block>
        <block type="lists_getSublist"></block>
        <block type="lists_split"></block>
        <block type="lists_sort"></block>
        <block type="lists_reverse"></block>
    </category>

    <!-- Variables -->
    <category name="Variables" custom="VARIABLE" colour="330"></category>

    <!-- Procedures -->
    <category name="Procedures" custom="PROCEDURE" colour="290"></category>

    <!-- Input/Output -->
    <category name="Input/Output" colour="65">
        <block type="text_print"></block>
        <block type="input_number"></block>
        <block type="input_text"></block>
    </category>

    <!-- Button Events -->
    <category name="Buttons" colour="300">
        <block type="button_click"></block>
        <block type="button_press"></block>
        <block type="button_release"></block>
        <block type="button_when_clicked"></block>
    </category>
</xml>
```

Figure 73. Toolbox Area – Category Blocks

2. initBlockly() — Workspace Initialization

This is the entry point (see Figure 74.), called on window.load. It:

- Calls defineCustomBlocks() and definePythonGenerators() first
- Injects the Blockly editor into the with grid, zoom, and trashcan options
- Overrides Blockly.Python.finish to return code as-is (no extra wrapping)
- Patches Blockly.Python.init to correctly bind the variable name database to the workspace

```
function initBlockly() {
    try {
        console.log("🚀 Starting Blockly initialization...");

        // Define custom blocks first
        defineCustomBlocks();

        // Initialize workspace
        workspace = Blockly.inject('blocklyDiv', {
            toolbox: document.getElementById('toolbox'),
            grid: {
                spacing: 20,
                length: 3,
                colour: '#ccc',
                snap: true
            },
            trashcan: true,
            zoom: {
                controls: true,
                wheel: true,
                startScale: 1.0,
                maxScale: 3,
                minScale: 0.3,
                scaleSpeed: 1.2
            }
        });

        // Load and Transfer button
        document.getElementById("loadAndTransfer").addEventListener("click", function() {
            const input = document.createElement('input');
            input.type = 'file';
            input.accept = '.xml';
            input.onchange = function(event) {
                const file = event.target.files[0];
                if (file) {
                    const reader = new FileReader();
                    reader.onload = function(e) {
                        try {
                            console.log("📥 Loading workspace and transferring from file:", file.name);
                            console.log("File content length:", e.target.result.length);

                            // Ensure custom blocks are defined
                            defineCustomBlocks();
                            definePythonGenerators();
```

Figure 74. Code Segment of initBlockly()

3. defineCustomBlocks() — Custom Block Definitions

Registers blocks that don't exist in Blockly's standard library (see Figure 75.):

- **input_number / input_text**: Represent Python's int(input()) and input()
- **button_click, button_press, button_release**: Event-style blocks with a text field for button name
- **button_when_clicked**: A dropdown-based button event block (Button 1/2/3, OK, Cancel)

```
function defineCustomBlocks() {
    console.log("🔧 Defining custom blocks...");

    // Input number block
    Blockly.Blocks['input_number'] = {
        init: function() {
            this.appendDummyInput()
                .appendField("είσοδος αριθμού");
            this.setOutput(true, "Number");
            this.setColour(65);
            this.setTooltip("Διαβάζει έναν ακέραιο αριθμό από τον χρήστη");
        }
    };

    // Input text block
    Blockly.Blocks['input_text'] = {
        init: function() {
            this.appendDummyInput()
                .appendField("είσοδος κειμένου");
            this.setOutput(true, "String");
            this.setColour(65);
            this.setTooltip("Διαβάζει κείμενο από τον χρήστη");
        }
    };

    // Button Click Event
    Blockly.Blocks['button_click'] = {
        init: function() {
            this.appendDummyInput()
                .appendField("όταν πατηθεί κουμπί")
                .appendField(new Blockly.FieldTextInput("Button1"), "BUTTON_NAME");
            this.appendStatementInput("DO")
                .setCheck(null)
                .appendField("κάνε");
            this.setPreviousStatement(true, null);
            this.setNextStatement(true, null);
            this.setColour(300);
            this.setTooltip("Εκτελείται όταν πατηθεί το κουμπί");
        }
    };
```

Figure 75. Code Segment of defineCustomBlocks ()

4. definePythonGenerators() — Code Generation Rules

Maps each custom block type to its Python output (see Figure 76.):

- input_number → int(input())
- input_text → input()
- Button blocks → Python def on_<name>_click(): functions
- **variables_set** is overridden to intelligently detect if the value being assigned is an input_number or input_text block, and generates the correct Python directly (e.g. x = int(input())) instead of a generic assignment

```
function definePythonGenerators() {
    console.log("🔧 Defining Python generators...");

    // Input number generator
    Blockly.Python['input_number'] = function(block) {
        return ['int(input())', Blockly.Python.ORDER_FUNCTION_CALL];
    };

    // Input text generator
    Blockly.Python['input_text'] = function(block) {
        return ['input()', Blockly.Python.ORDER_FUNCTION_CALL];
    };

    // Button Click Event Generator
    Blockly.Python['button_click'] = function(block) {
        const buttonName = block.getFieldValue('BUTTON_NAME');
        const statements = Blockly.Python.statementToCode(block, 'DO');
        const code = `# Event: Button ${buttonName} clicked\ndef on_${buttonName.toLowerCase()}_click():\n${statements || '    pass'}\n\n`;
        return code;
    };

    // Button Press Generator
    Blockly.Python['button_press'] = function(block) {
        const buttonName = block.getFieldValue('BUTTON_NAME');
        const statements = Blockly.Python.statementToCode(block, 'DO');
        const code = `# Event: Button ${buttonName} pressed\ndef on_${buttonName.toLowerCase()}_press():\n${statements || '    pass'}\n\n`;
        return code;
    };

    // Button Release Generator
    Blockly.Python['button_release'] = function(block) {
        const buttonName = block.getFieldValue('BUTTON_NAME');
        const statements = Blockly.Python.statementToCode(block, 'DO');
        const code = `# Event: Button ${buttonName} released\ndef on_${buttonName.toLowerCase()}_release():\n${statements || '    pass'}\n\n`;
        return code;
    };

    // Button When Clicked Generator
    Blockly.Python['button_when_clicked'] = function(block) {
        const buttonType = block.getFieldValue('BUTTON_TYPE');
        const statements = Blockly.Python.statementToCode(block, 'DO');
        const code = `# Event: ${buttonType} button clicked\ndef on_${buttonType.toLowerCase()}_clicked():\n${statements || '    pass'}\n\n`;
        return code;
    };
```

Figure 76. Code Segment of definePythonGenerators()

5. setupEventListeners() — Button Wiring

Connects the four UI buttons (see Figure 77.):

- **Convert to Python Code (🚀)**: Calls generateAndTransferCode()
- **Clear (🗑️)**: Clears the workspace after a confirm dialog
- **Save (💾)**: Serializes the workspace to XML and downloads it as blockly_workspace.xml

- **Load & Transfer (📥)**: Opens a file picker, loads an XML file into the workspace, then auto-generates and transfers code, shown in Figure 78
- Also adds a **real-time listener** on the workspace so code regenerates on any block move, create, delete, or change, shown in Figure 79

```
<!-- Buttons -->
<div class="controls" bis_skin_checked="1">
    <button type="button" id="convertCode">🚀 Convert</button>
    <button type="button" id="clearWorkspace">🗑️ Clear</button>
    <button type="button" id="saveWorkspace">💾 Save</button>
    <button type="button" id="loadAndTransfer">📥 Load &amp;amp; Transfer </button>
</div>
```

Figure 77. Button Wiring

```
function generateAndTransferCode(event, isAutomatic = false) {
    if (event) {
        event.preventDefault();
    }

    if (!workspace) {
        if (!isAutomatic) alert("Blockly workspace not initialized!");
        return false;
    }

    try {
        console.log("Generating code from workspace...");

        Blockly.Python.init(workspace);
        let code = Blockly.Python.workspaceToCode(workspace);

        // Clean up code
        code = code.replace(/\b(\w+) = None\s+\1 =/g, "$1 =");

        if (!code.trim()) {
            if (!isAutomatic) alert("Δεν υπάρχει κώδικας για μεταφορά!");
            return false;
        }

        console.log("Generated code:", code);

        // ENHANCED search for Moodle/CodeRunner answer fields
        const possibleSelectors = [
            ".que.coderunner textarea[name*='answer']",
            ".que.coderunner textarea",
            "textarea[name='answer']",
            "textarea[name*='answer']",
            "textarea[id*='answer']",
            ".que.coderunner .answer textarea",
            "#id_answer",
            "[name^='q'][name*='answer']",
            "[id^='id_q'][id$='_answer']",
            ".answer textarea",
            "textarea.form-control",
            "textarea"
        ];
```

Figure 78. Code Segment of generateAndTransferCode ()

```
// Save workspace button
document.getElementById("saveWorkspace").addEventListener("click", function() {
    try {
        const xml = Blockly.Xml.workspaceToDom(workspace);
        const xmlText = Blockly.Xml.domToPrettyText(xml);
        const blob = new Blob([xmlText], {
            type: 'text/xml'
        });
        const url = URL.createObjectURL(blob);
        const a = document.createElement('a');
        a.href = url;
        a.download = 'blockly_workspace.xml';
        a.click();
        URL.revokeObjectURL(url);
        console.log("Workspace saved successfully");
    } catch (error) {
        alert("Storage error: " + error.message);
    }
});

// Real-time code generation
const workspaceChangeListener = function(event) {
    if (event.type === Blockly.Events.BLOCK_MOVE ||
        event.type === Blockly.Events.BLOCK_CREATE ||
        event.type === Blockly.Events.BLOCK_DELETE ||
        event.type === Blockly.Events.BLOCK_CHANGE) {
        generateCode();
    }
};

workspace.addChangeListener(workspaceChangeListener);
workspace.changeListener = workspaceChangeListener;

console.log("☑ Event listeners set up successfully");
```

Figure 79. Save workspace button & Real-time code generation

6. generateCode() — Live Code Preview

Calls Blockly.Python.workspaceToCode() (see Figure 80.), to translate all blocks to Python, applies a small regex cleanup to remove redundant var = None; var = patterns, and displays the result in #codeOutput. Returns the raw code string (see Figure 81.).

```
console.log("🔄 generateCode: Starting code generation...");

Blockly.Python.init(workspace);
let code = Blockly.Python.workspaceToCode(workspace);

// Clean up code
code = code.replace(/\b(\w+) = None\s+\1 =/g, "$1 =");

console.log("🔄 generateCode: Raw generated code:", code);
console.log("🔄 generateCode: Code length:", code.length);

const outputElement = document.getElementById('codeOutput');
```

Figure 80. Code Segment of Blockly.Python.workspaceToCode ()

```
<!-- Code results -->
<div class="output-header" bis_skin_checked="1">Generated Python Code:</div>
<div id="codeOutput" bis_skin_checked="1"># Empty workspace - add blocks to view code</div>
```

```
// Check if code was generated
const codeContent = document.getElementById('codeOutput').textContent;
console.log("Generated code content:", codeContent);
```

Figure 81. Code Segment of codeOutput element

7. generateAndTransferCode() — Transfer to Moodle/CodeRunner

This is the most complex function (see Figure 82.). After generating the Python code it:

1. Searches the DOM for a writable <textarea> answer field using a prioritized list of CSS selectors (targeting Moodle's CodeRunner question format)
2. If found: focuses it, clears it, sets the code, then fires a large set of events (input, change, keyup, etc.) and tries jQuery/MooTools triggers for Moodle compatibility
3. If not found: falls back to **clipboard copy** via navigator.clipboard, or opens a popup via showCodePopup()

```
// ENHANCED search for Moodle/CodeRunner answer fields
const possibleSelectors = [
    ".que.coderunner textarea[name*='answer']",
    ".que.coderunner textarea",
    "textarea[name='answer']",
    "textarea[name*='answer']",
    "textarea[id*='answer']",
    ".que.coderunner .answer textarea",
    "#id_answer",
    "[name^='q'][name*='answer']",
    "[id^='id_q'][id$='_answer']",
    ".answer textarea",
    "textarea.form-control",
    "textarea"
];

let answerField = null;
for (let selector of possibleSelectors) {
    const elements = document.querySelectorAll(selector);
    for (let element of elements) {
        if (element &&
            element.tagName.toLowerCase() === 'textarea' &&
            !element.disabled &&
            !element.readOnly &&
            element.style.display !== 'none') {
            answerField = element;
            console.log("Found answer field with selector:", selector);
            break;
        }
    }
    if (answerField) break;
}
```

Figure 82. Code Segment of generateAndTransferCode() – textarea tag

8. showCodePopup() — Fallback Display

Opens a new browser window with a read-only <textarea> containing the generated code and a copy button, used when no answer field is found and clipboard access fails, described in Figure 83..

```
function showCodePopup(code) {
    const popup = window.open('', '_blank', 'width=600,height=400');
    if (popup) {
        popup.document.write(`
            <!DOCTYPE html>


                Παραγόμενος Python Κώδικας


                <h2>📋 Copy the code below:</h2>
                <textarea readonly>${code}</textarea>
                <br>
                <button onclick="document.querySelector('textarea').select(); document.execCommand('copy'); alert('Copied!');">Copy</button>
                <button onclick="window.close();">Closure</button>


        `);
    } else {
        alert("Please copy the code from the preview below.");
    }
```

Figure 83. Code Segment of showCodePopup()

9) Answer Field

The Answer Field is the section in the Moodle CodeRunner question where the student's Python solution is submitted. In this exercise, the expected answer demonstrates how to read two integer inputs, perform arithmetic operations, and display the results.

Expected Python Code

The correct solution that should appear in the answer field, (see Figure 84.), is the following:

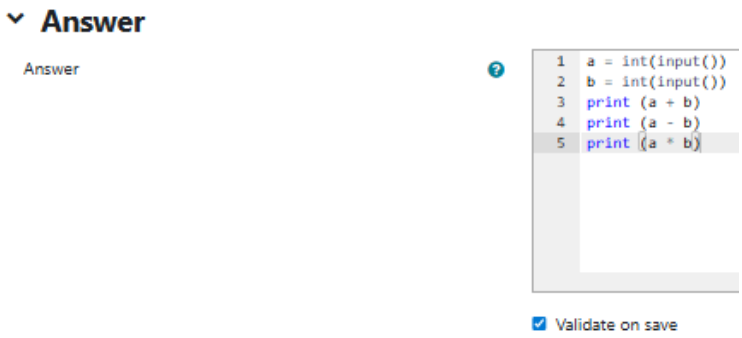


Figure 84. Answer Field Case 1

10) Test Cases

Test cases are used by Moodle CodeRunner to automatically evaluate the student's submitted code. Each test case defines a set of inputs and the expected output, along with display and grading properties (see Figure 85.).

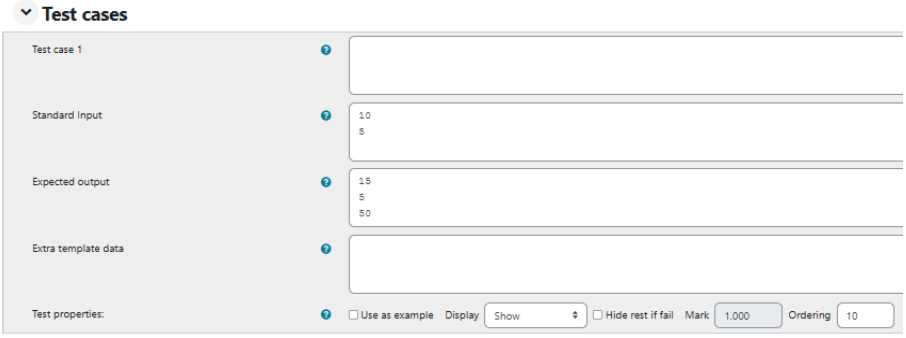


Figure 85. Test Case 1 – Configuration

Results and Output

This section presents the execution results of the developed CodeRunner exercise, along with the system behavior during code generation and transfer. The figures illustrate both the visual block-based implementation and the corresponding generated Python code.

Code Construction and Execution — Exercise 1

The CodeRunner environment allows users to construct programs using a block-based interface, which is then automatically translated into Python code.

As shown in Figure 86, the user defines two variables (a and b) using numeric input blocks. The program then performs three fundamental arithmetic operations: addition, subtraction, and multiplication. Each result is displayed using print statements.

The generated Python code, visible beneath the Blockly interface, is:

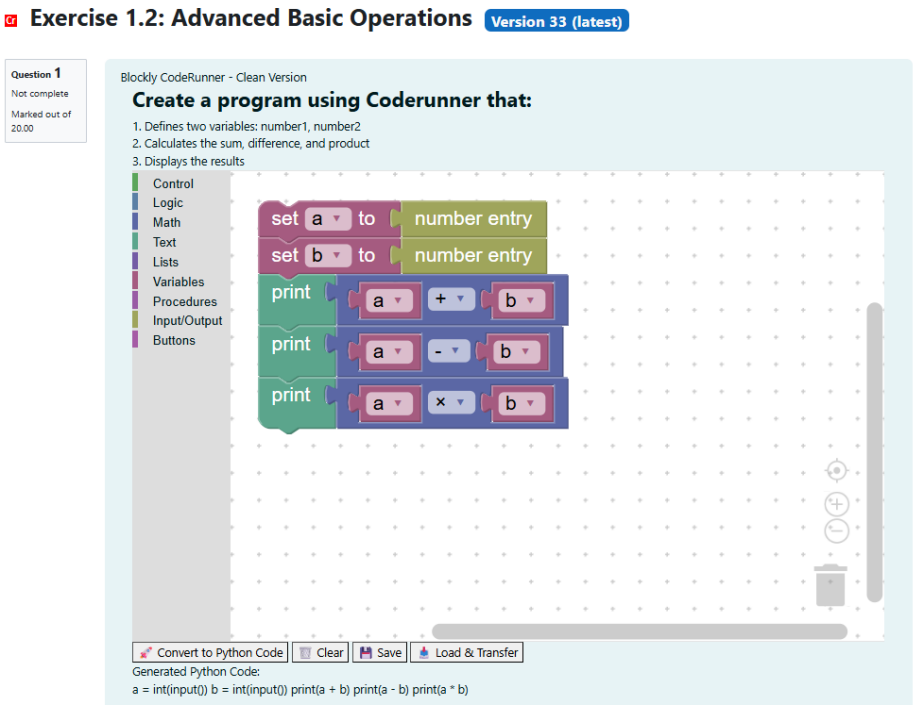


Figure 86. Case 1 Blockly code

Code Transfer and Confirmation

After constructing the program, the user selects the **"Load & Transfer"** option to submit the generated code to the answer field.

Figure 87, illustrates the system's confirmation message, indicating that the code has been successfully transferred. This feedback mechanism ensures that the user is aware of a successful action and reduces the likelihood of submission errors.

The message:

"The code was successfully transferred to the response field!"

demonstrates proper system responsiveness and usability within the CodeRunner environment.

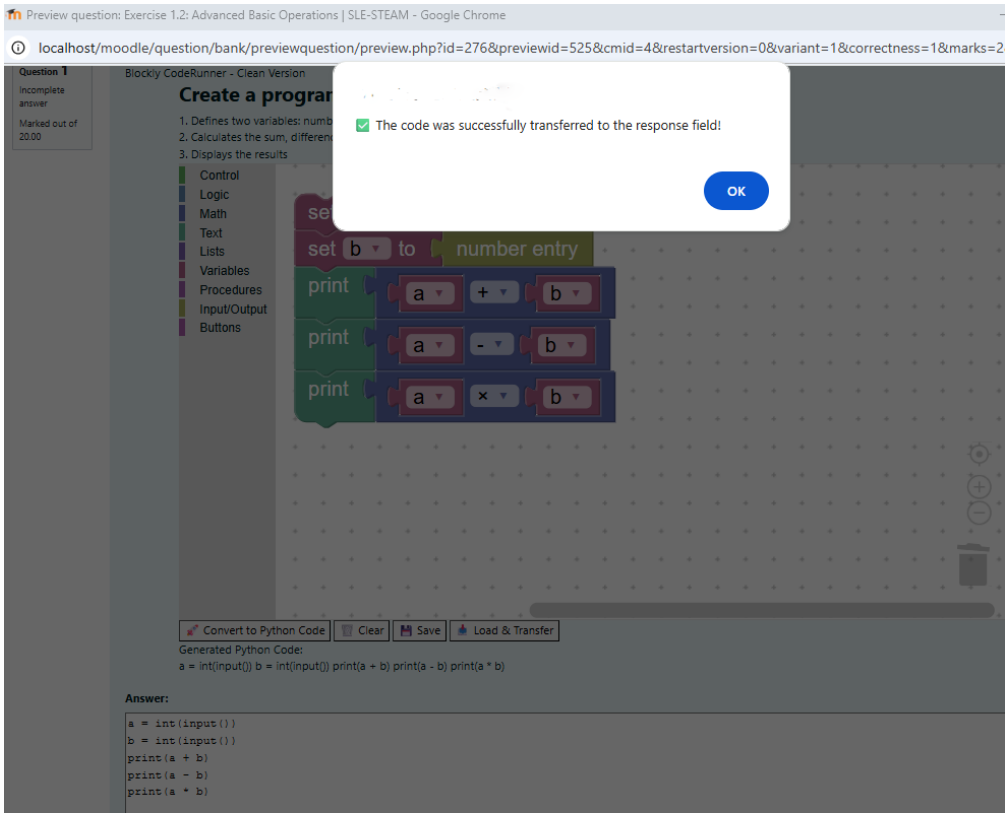


Figure 87. Confirmation message Case 1

Final Code Representation in Answer Field

Once the transfer process is completed, the generated Python code appears in the answer section, as shown in Figure 88. The displayed code maintains the same structure as the automatically generated version, ensuring consistency between the visual programming interface and the textual representation. This feature is particularly important in educational contexts, as it helps learners transition from block-based programming to text-based coding.

Additionally, the correctness of the implementation can be verified by executing the code with sample inputs.

The expected output consists of three values:

- The sum of the two input numbers
- Their difference
- Their product

This confirms that the program meets all the requirements defined in the exercise.

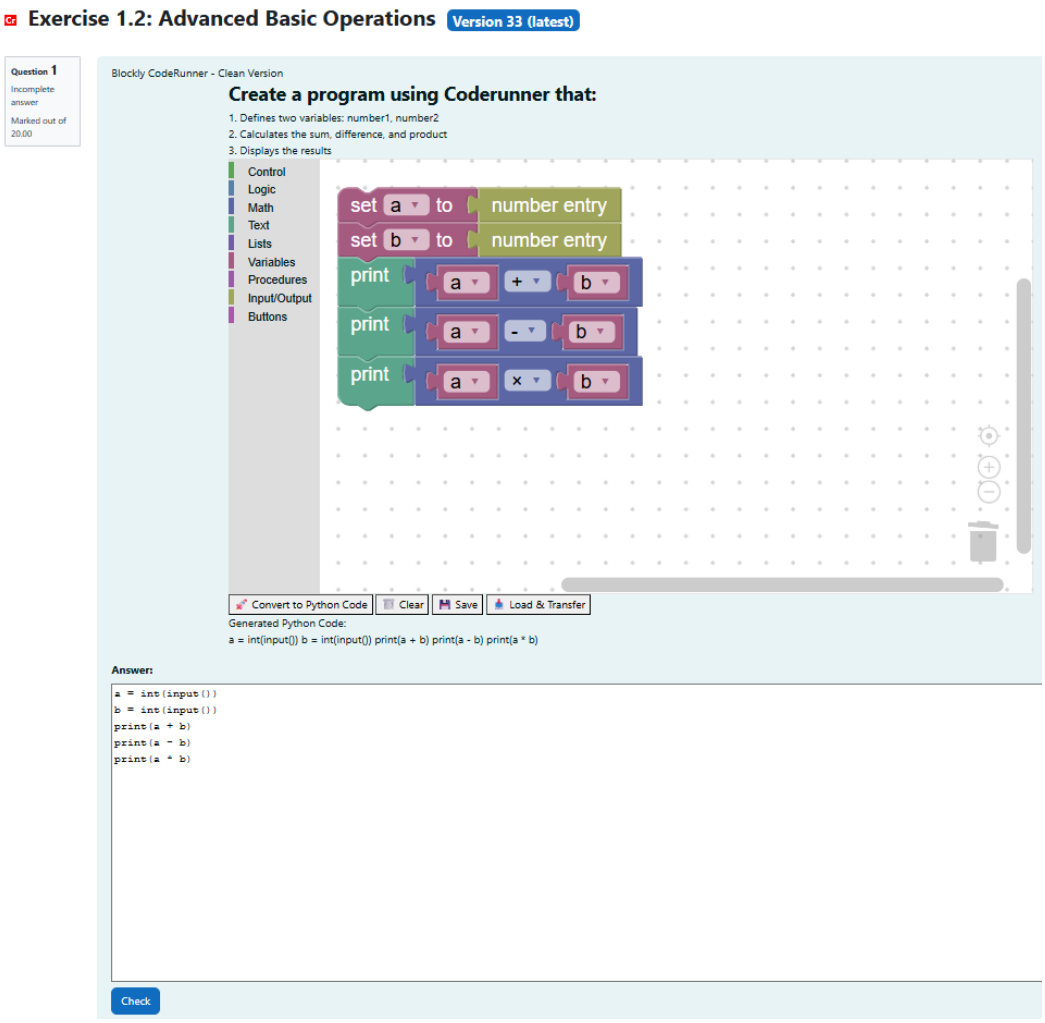


Figure 88. Converted code Case 1

Code Execution and Output Verification

After confirming that the final code is correctly displayed in the answer field, the next step involves executing the program using the built-in testing environment, as illustrated in Figure 89.

When the program is run, it prompts the user to enter two integer values. These inputs are then processed according to the implemented operations. The system evaluates the program by comparing the generated outputs against the expected results.

The output consists of three sequential values:

- The sum of the two input numbers
- The difference between the first and second numbers
- The product of the two numbers

The successful execution of the program demonstrates that the logic defined in both the block-based interface and the Python code has been correctly implemented. Furthermore, the alignment between expected and actual outputs confirms the functional accuracy of the solution.

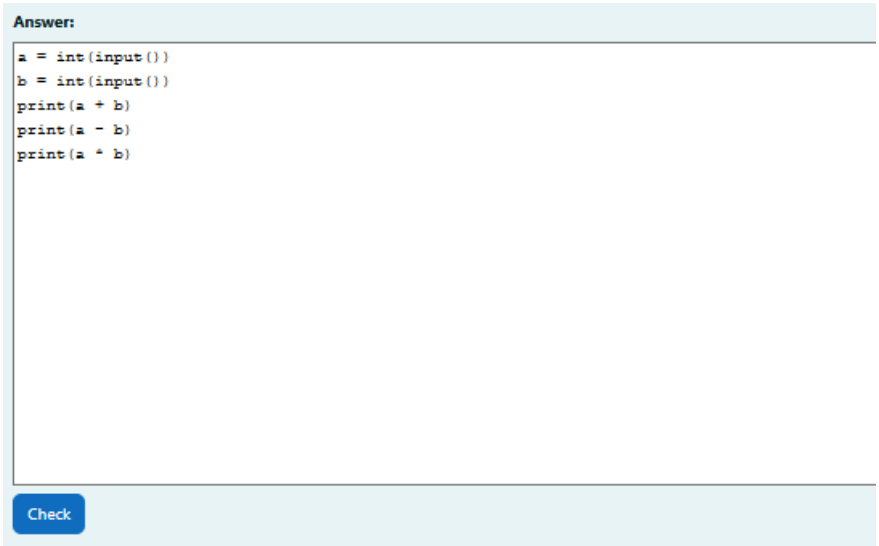


Figure 89. Python code for Case 1

Test Results and Validation — Exercise 1

The final stage, shown in Figure 90, presents the test results after running the program through the automated checking system.

The results panel indicates that all test cases have passed successfully. Each input is matched with its corresponding expected output, and the program's results are displayed alongside for comparison. The presence of validation indicators (such as check marks) confirms that the outputs are correct.

This validation step is essential as it ensures:

- The correctness of arithmetic operations
- Proper handling of user input
- Accurate display of results in the required format

The message indicating that all tests have passed serves as final confirmation that the program satisfies the exercise requirements. Therefore, the implementation is complete, correct, and ready for submission.

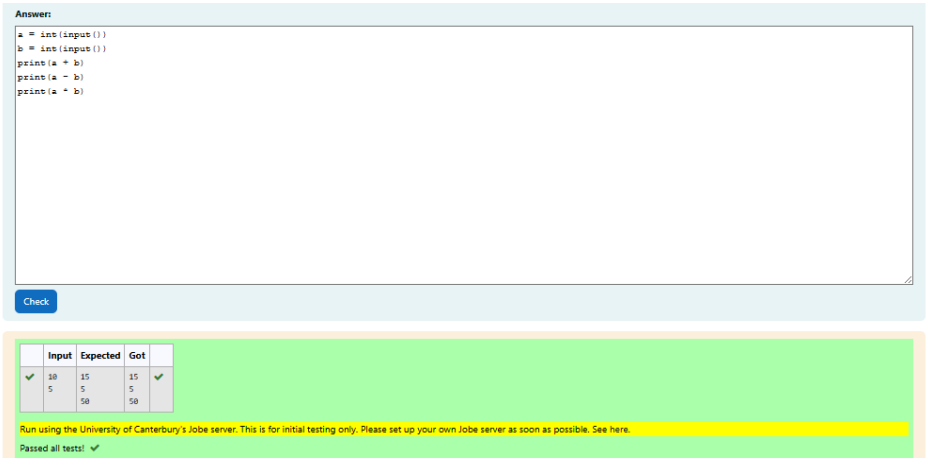


Figure 90. Test Results & Validation for Case 1

Error Detection, Debugging, and Final Validation

Error Handling and Debugging

Figure 91, illustrates the intermediate stage where the program encountered a syntax error during execution. The error message indicates that a parenthesis was not properly closed in one of the print statements. As a result, the program failed to compile and execute.

This step highlights the importance of careful debugging and syntax validation in programming.

The error message provided useful feedback, allowing the issue to be quickly identified and corrected.

The debugging process ensured:

- Identification of syntax errors in the code
- Correction of incomplete or improperly written statements
- Improved understanding of Python syntax rules

By resolving the error, the program was able to run successfully in subsequent attempts. This stage demonstrates the iterative nature of coding, where errors are detected, analyzed, and fixed to achieve a correct implementation.

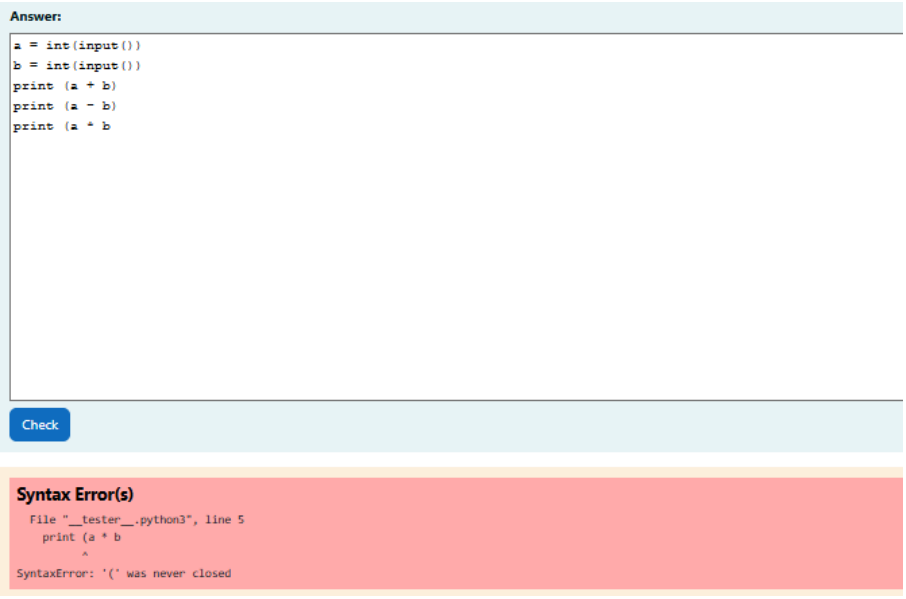


Figure 91. Syntax Case 1

Final Corrected Execution

Figure 92 presents the final corrected version of the program after resolving all syntax and logical errors. The code executes without any issues, and the outputs are displayed correctly for the given inputs.

The program successfully performs:

- Addition of the two input values
- Subtraction of the two input values
- Multiplication of the two input values

All outputs match the expected results, confirming that the corrections made during debugging were effective. The absence of errors and the successful execution indicate that the program is stable and functions as intended.This final stage confirms that the implementation process is complete, and the program meets all specified requirements.

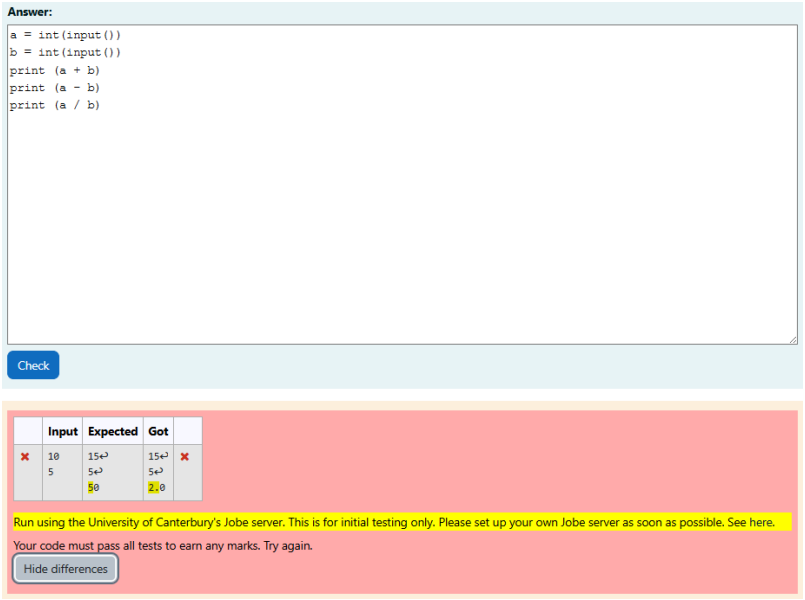


Figure 92. Test Results & Validation for Case 1

Figures 91 and 92 together demonstrate how CodeRunner supports both syntax checking and logical validation throughout the development process. In Figure 91., the system identifies a syntax error caused by an unclosed parenthesis, preventing the program from executing. This immediate feedback highlights how CodeRunner effectively detects structural issues in the code and guides the debugging process.

Once the syntax error is corrected, Figure 100 shows the successful execution of the program, where all test cases are evaluated. At this stage, CodeRunner compares the program's outputs with the expected results, enabling the detection of any logical errors. The matching outputs confirm that the program not only runs correctly but also produces accurate results.
This connection between syntax checking and test case validation ensures:

- Early detection and correction of syntax errors
- Systematic identification of logical mistakes through test cases
- Progressive improvement of code reliability and correctness

Together, these stages form a complete validation cycle, where CodeRunner first ensures the code is executable and then verifies its correctness, leading to a fully functional and accurate implementation.

6.3.2 Exercise 2 — Arduino LED Blink

Introduces Arduino hardware interaction, requiring students to blink an LED on pin 13 five times using a loop. This exercise uses the standard Blockly categories augmented with an Arduino category containing 13 specialised blocks covering digital I/O, analog I/O, timing, serial communication, servo motors, tone/sound, and pin/value constants. A simulation template defines Python equivalents of Arduino functions (pinMode, digitalWrite, delay) that produce timestamped formatted output messages, enabling automated grading without physical hardware.

System Architecture — Exercise 2

1. HTML Structure & Toolbox

The page renders a Blockly visual programming environment inside a Moodle CodeRunner question. The task displayed to students is to blink an LED on pin 13 five times. The <xml id="toolbox"> defines all available block categories: the standard ones (Control, Logic, Math, Text, Lists, Variables, Procedures, Input/Output, Buttons) **plus a new Arduino category** exclusive to this file, which contains 13 specialized blocks covering digital I/O, analog I/O, timing, serial communication, servo motors, tone/sound, and pin/value constants (see Figure 93.).

```
<!-- Arduino Blocks -->
<category name="Arduino" colour="120">
    <block type="arduino_pin_mode"></block>
    <block type="arduino_digital_write"></block>
    <block type="arduino_digital_read"></block>
    <block type="arduino_analog_read"></block>
    <block type="arduino_analog_write"></block>
    <block type="arduino_delay"></block>
    <block type="arduino_millis"></block>
    <block type="arduino_serial_print"></block>
    <block type="arduino_serial_println"></block>
    <block type="arduino_servo_attach"></block>
    <block type="arduino_servo_write"></block>
    <block type="arduino_tone"></block>
    <block type="arduino_constant"></block>
    <block type="arduino_pin_constant"></block>
</category>
```

Figure 93. Toolbox Area – Arduino Blocks

2. initBlockly() — Workspace Initialization

Identical in structure to the previous version: loads custom blocks, injects the Blockly editor into #blocklyDiv with grid/zoom/trashcan options, sets up the Load & Transfer file picker, overrides Blockly.Python.finish, patches the variable name database, then calls setupEventListeners() and the initial generateCode(), shown in the abobe Figure 94.

```
function initBlockly() {
    try {
        console.log("🚀 Starting Blockly initialization...");

        // Define custom blocks first
        defineCustomBlocks();

        // Initialize workspace
        workspace = Blockly.inject('blocklyDiv', {
            toolbox: document.getElementById('toolbox'),
            grid: {
                spacing: 20,
                length: 3,
                colour: '#ccc',
                snap: true
            },
            trashcan: true,
            zoom: {
                controls: true,
                wheel: true,
                startScale: 1.0,
                maxScale: 3,
                minScale: 0.3,
                scaleSpeed: 1.2
            }
        });

        // Load and Transfer button
        document.getElementById("loadAndTransfer").addEventListener("click", function() {
            const input = document.createElement('input');
            input.type = 'file';
            input.accept = '.xml';
            input.onchange = function(event) {
                const file = event.target.files[0];
                if (file) {
                    const reader = new FileReader();
                    reader.onload = function(e) {
                        try {
                            console.log("📥 Loading workspace and transferring from file:", file.name);
                            console.log("File content length:", e.target.result.length);

                            // Ensure custom blocks are defined
                            defineCustomBlocks();
                            definePythonGenerators();
```

Figure 94. Code Segment of initBlockly()

3. defineCustomBlocks() — Custom + Arduino Block Definitions

This is the key difference from the previous file. Beyond the standard custom blocks (input_number, input_text, button events), this version defines a full set of Arduino hardware blocks (see Figure 95.):

Digital I/O — arduino_pin_mode (sets a pin as INPUT/OUTPUT/INPUT_PULLUP), arduino_digital_write (writes HIGH or LOW to a pin), arduino_digital_read (reads the state of a digital pin).

Analog I/O — arduino_analog_read (reads a value 0–1023 from an analog pin), arduino_analog_write (PWM output, value 0–255).

Timing — arduino_delay (pauses execution for a given number of milliseconds), arduino_millis (returns elapsed time since startup in ms).

Serial Communication — arduino_serial_print and arduino_serial_println (send data to the Serial Monitor, the latter adding a newline).

Servo Motor — arduino_servo_attach (connects a servo to a pin), arduino_servo_write (rotates the servo to a given angle, 0–180°).

Tone/Sound — arduino_tone (plays sound at a given frequency / duration on a specified pin).

Constants — arduino_constant (HIGH/LOW dropdown), arduino_pin_constant (a dropdown listing all Arduino pins: 0–13 and A0–A5).

```
// ============ DIGITAL I/O ============

// pinMode Block
Blockly.Blocks['arduino_pin_mode'] = {
    init: function() {
        this.appendValueInput("PIN")
            .setCheck("Number")
            .appendField("pinMode pin");
        this.appendDummyInput()
            .appendField("as")
            .appendField(new Blockly.FieldDropdown([
                ["INPUT", "INPUT"],
                ["OUTPUT", "OUTPUT"],
                ["INPUT_PULLUP", "INPUT_PULLUP"]
            ]), "MODE");
        this.setPreviousStatement(true, null);
        this.setNextStatement(true, null);
        this.setColour(120);
        this.setTooltip("Sets the function of a pin");
    }
};

// digitalWrite Block
Blockly.Blocks['arduino_digital_write'] = {
    init: function() {
        this.appendValueInput("PIN")
            .setCheck("Number")
            .appendField("digitalWrite pin");
        this.appendValueInput("VALUE")
            .setCheck(["Boolean", "Number"])
            .appendField("τιμή");
        this.setPreviousStatement(true, null);
        this.setNextStatement(true, null);
        this.setColour(120);
        this.setTooltip("Writes HIGH (1) or LOW (0) to a digital pin");
    }
};
```

Figure 95. Code Segment of defineCustomBlocks ()

4. definePythonGenerators() — Code Generation Rules

Covers all the same standard generators as before (input, button events, variables_set), plus the Arduino Python generators which translate each Arduino block into Python-style function calls that Moodle CodeRunner can evaluate (see Figure 96.):

pinMode(pin, mode), digitalWrite(pin, value), digitalRead(pin), analogRead(pin), analogWrite(pin, value), delay(time), millis(), Serial.print(text), Serial.println(text), servo.write(angle), tone(pin, freq, duration), and the constants (HIGH, LOW, pin numbers) are returned as plain values.

```
// pinMode
Blockly.Python['arduino_pin_mode'] = function(block) {
    const pin = Blockly.Python.valueToCode(block, 'PIN', Blockly.Python.ORDER_ATOMIC);
    const mode = block.getFieldValue('MODE');
    return `pinMode(${pin}, ${mode})\n`;
};

// digitalWrite
Blockly.Python['arduino_digital_write'] = function(block) {
    const pin = Blockly.Python.valueToCode(block, 'PIN', Blockly.Python.ORDER_ATOMIC);
    const value = Blockly.Python.valueToCode(block, 'VALUE', Blockly.Python.ORDER_ATOMIC);
    return `digitalWrite(${pin}, ${value})\n`;
};

// digitalRead
Blockly.Python['arduino_digital_read'] = function(block) {
    const pin = Blockly.Python.valueToCode(block, 'PIN', Blockly.Python.ORDER_ATOMIC);
    return [`digitalRead(${pin})`, Blockly.Python.ORDER_FUNCTION_CALL];
};

// analogRead
Blockly.Python['arduino_analog_read'] = function(block) {
    const pin = Blockly.Python.valueToCode(block, 'PIN', Blockly.Python.ORDER_ATOMIC);
    return [`analogRead(${pin})`, Blockly.Python.ORDER_FUNCTION_CALL];
};

// analogWrite
Blockly.Python['arduino_analog_write'] = function(block) {
    const pin = Blockly.Python.valueToCode(block, 'PIN', Blockly.Python.ORDER_ATOMIC);
    const value = Blockly.Python.valueToCode(block, 'VALUE', Blockly.Python.ORDER_ATOMIC);
    return `analogWrite(${pin}, ${value})\n`;
};
```

Figure 96. Code Segment of definePythonGenerators()

5. setupEventListeners() — Button Wiring

Same four buttons as before: Convert (🚀), Clear (🗑️), Save (💾), Load & Transfer (📥), plus the real-time workspace change listener that regenerates code on every block move, create, delete, or change event, as shown in Figure 97.

```
<!-- Buttons -->
<div class="controls" bis_skin_checked="1">
    <button type="button" id="convertCode">🚀 Convert</button>
    <button type="button" id="clearWorkspace">🗑️ Clear</button>
    <button type="button" id="saveWorkspace">💾 Save</button>
    <button type="button" id="loadAndTransfer">📥 Load &amp;amp; Transfer </button>
</div>
```

Figure 97. Button Wiring

6. generateCode() — Live Code Preview

Identical to the previous version: calls Blockly.Python.workspaceToCode(), shown in Figure 98, cleans up redundant var = None assignments, and displays the result in #codeOutput.

```
console.log("🔄 generateCode: Starting code generation...");

Blockly.Python.init(workspace);
let code = Blockly.Python.workspaceToCode(workspace);

// Clean up code
code = code.replace(/\b(\w+) = None\s+\1 =/g, "$1 =");

console.log("🔄 generateCode: Raw generated code:", code);
console.log("🔄 generateCode: Code length:", code.length);

const outputElement = document.getElementById('codeOutput');
```

Figure 98. Code Segment of Blockly.Python.workspaceToCode ()

7. generateAndTransferCode() — Transfer to Moodle/CodeRunner

Identical to the previous version (see Figure 99): searches the DOM for a writable CodeRunner <textarea> using a prioritized selector list, injects the generated code into it, fires a comprehensive chain of DOM events (plus jQuery/MooTools triggers), and falls back to clipboard copy or a popup window if no answer field is found.

```
// ENHANCED search for Moodle/CodeRunner answer fields
const possibleSelectors = [
    ".que.coderunner textarea[name*='answer']",
    ".que.coderunner textarea",
    "textarea[name='answer']",
    "textarea[name*='answer']",
    "textarea[id*='answer']",
    ".que.coderunner .answer textarea",
    "#id_answer",
    "[name^='q'][name*='answer']",
    "[id^='id_q'][id$='_answer']",
    ".answer textarea",
    "textarea.form-control",
    "textarea"
];

let answerField = null;
for (let selector of possibleSelectors) {
    const elements = document.querySelectorAll(selector);
    for (let element of elements) {
        if (element &&
            element.tagName.toLowerCase() === 'textarea' &&
            !element.disabled &&
            !element.readOnly &&
            element.style.display !== 'none') {
            answerField = element;
            console.log("Found answer field with selector:", selector);
            break;
        }
    }
    if (answerField) break;
}
```

Figure 99. Code Segment of generateAndTransferCode() – textarea tag

8. showCodePopup() — Fallback Display

Identical to the previous version: (see Figure 100) opens a small popup window with a read-only textarea and a copy button when all other transfer methods fail.

```
function showCodePopup(code) {
    const popup = window.open('', '_blank', 'width=600,height=400');
    if (popup) {
        popup.document.write(`
            <!DOCTYPE html>


                Παραγόμενος Python Κώδικας


                <h2>📋 Copy the code below:</h2>
                <textarea readonly>${code}</textarea>
                <br>
                <button onclick="document.querySelector('textarea').select(); document.execCommand('copy'); alert('Copied!');">Copy</button>
                <button onclick="window.close();">Closure</button>


        `);
    } else {
        alert("Please copy the code from the preview below.");
    }
```

Figure 100. Code Segment of showCodePopup()

9) Customisation Field Description

The customisation field, shown in Figure 101, defines a simulated execution environment that emulates fundamental Arduino functionality, enabling the automated assessment of student code without the need for physical hardware.

Specifically, standard Arduino constants (OUTPUT, HIGH, and LOW) are represented as string values to facilitate readable output. A global variable, simulation_time, is introduced to model the temporal progression of execution. This variable is initialized at 100 ms and incremented by 100 ms following each simulated operation, thereby providing a consistent chronological trace of program behavior.

Core Arduino functions—pinMode, digitalWrite, and delay—are reimplemented to produce formatted output messages instead of interacting with hardware. Each function logs its invocation along with the current simulation timestamp and subsequently updates the simulation time. This approach ensures deterministic and interpretable output suitable for automated grading.

The placeholder {{ STUDENT_ANSWER }} is dynamically replaced with the student's submitted code, which is executed within the simulated environment. Test cases are then applied sequentially using a templating loop, with outputs separated by a predefined delimiter to ensure clarity.

Customisation

Template

```
# Arduino constants
OUTPUT = "OUTPUT"
HIGH = "HIGH"
LOW = "LOW"

# Simulated time starting at 100ms and increasing by 100ms
simulation_time = 100

def pinMode(pin, mode):
    global simulation_time
    print(f"[{simulation_time}ms] pinMode({pin}, {mode})")
    simulation_time += 100  # +100ms after each action

def digitalWrite(pin, value):
    global simulation_time
    print(f"[{simulation_time}ms] digitalWrite({pin}, {value}) -> OK")
    simulation_time += 100  # +100ms after each action

def delay(ms):
    global simulation_time
    print(f"[{simulation_time}ms] delay({ms})")
    simulation_time += 100  # +100ms after each action

# Student's code
{{ STUDENT_ANSWER }}

# Test cases
{% for TEST in TESTCASES %}
{{ TEST.testcode }}
{% if not loop.last %}
print(SEPARATOR)
{% endif %}
{% endfor %}
```

Figure 101. Customisation Field Case 2

10) Answer Field Case 2

The Answer Field, shown in Figure 102, is the area in the Moodle CodeRunner question where students submit their Python code. In this task, the expected response shows how to take two integer inputs, carry out arithmetic operations, and present the results.

The correct solution that should be shown in the answer field is as follows:

Answer

Answer

```
pinMode(13, OUTPUT)
for count in range(5):
    digitalWrite(13, HIGH)
    delay(1000)
    digitalWrite(13, LOW)
    delay(1000)
```

Validate on save

Figure 102. Answer Field Case 2

11) Test Cases

Test cases are utilized by Moodle CodeRunner to automatically assess the code submitted by the student. Each test case specifies a set of inputs and the corresponding expected output, along with settings for display and grading, as shown in Figure 103.

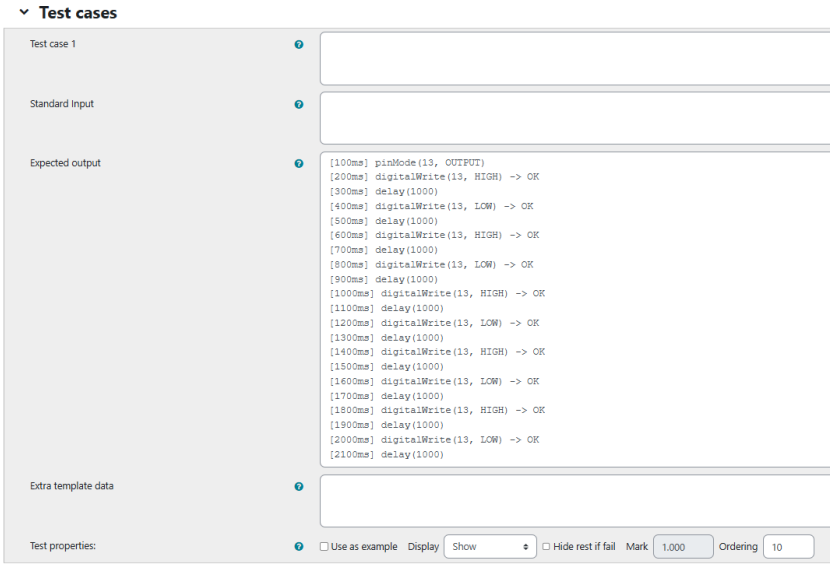


Figure 103. Answer Field Case 2

Results and Output

This section displays the execution outcomes of the developed CodeRunner exercise, as well as the system's behavior during code generation and transfer. The Figure 106, show both the visual block-based implementation and the equivalent generated Python code.

Code Construction and Execution — Exercise 2

In this stage, the user constructs the program visually using Blockly blocks. Each block represents a programming command (e.g., pinMode, digitalWrite, delay, and loops).

The blocks are snapped together to define the program logic:

- The **pinMode block** initializes pin 13 as an output.
- A **repeat block** is used to execute actions 5 times.
- Inside the loop, **digitalWrite blocks** control the LED state (HIGH/LOW).
- **Delay blocks** introduce a 1-second pause between actions.

Once the blocks are assembled , (see Figure 104), the system automatically translates this visual structure into equivalent Python code. This helps beginners understand programming logic without needing to write syntax manually.

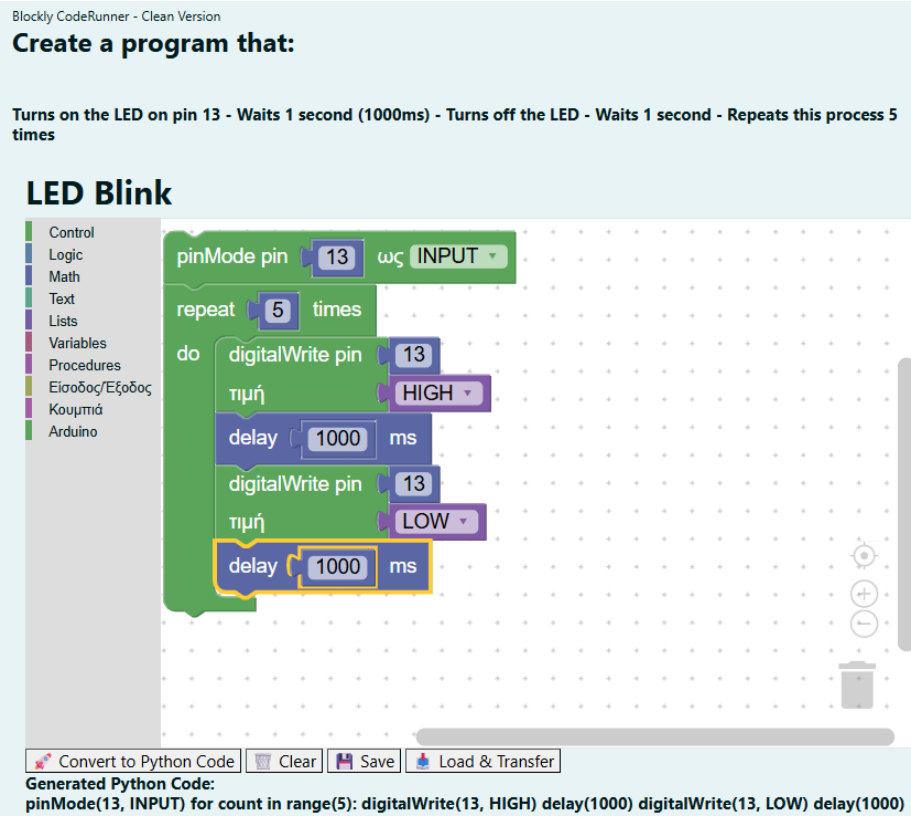


Figure 104. Case 2 Blockly code

Code Transfer and System Feedback

After generating the code, the user clicks the **"Load & Transfer"** button. At this point:

- The generated Python code is automatically copied into the **answer field** in Moodle CodeRunner.
- The system provides immediate feedback through a confirmation message (e.g., *"The code was successfully transferred to the response field!"*).
- This step ensures that the student's visual program is correctly converted and submitted in textual form.

This automated transfer (see Figure 105),reduces errors from manual copying and improves the workflow between Blockly and CodeRunner.

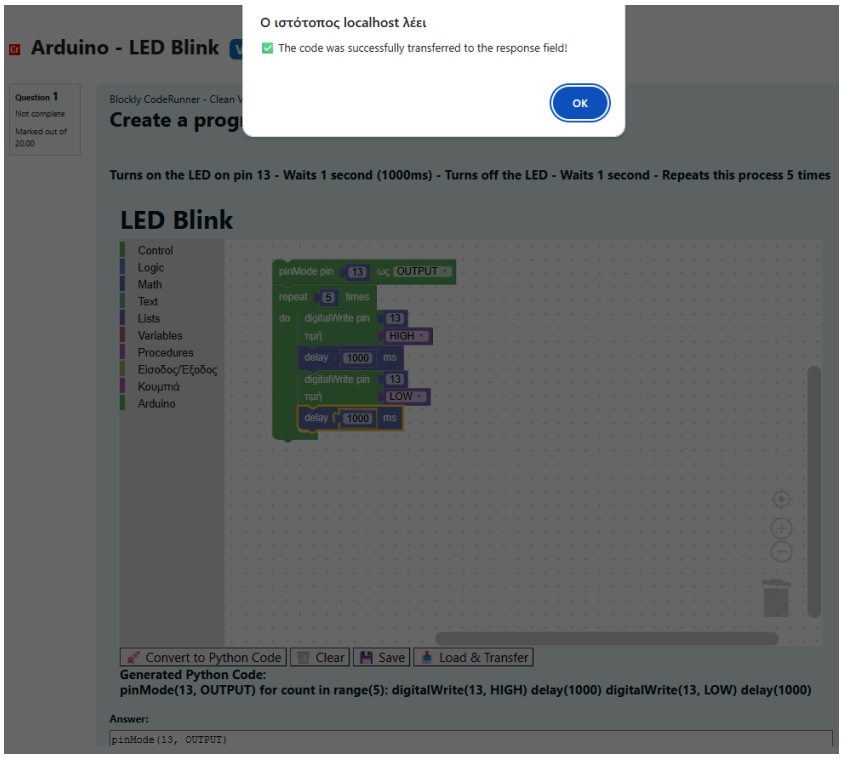


Figure 105. Confirmation message Case 2

Final Code Representation in Answer Field

In the final stage, the translated Python code appears in the answer box, ready for submission and grading. The code reflects the exact logic created in Blockly:

- Proper structure and indentation (e.g., the for loop).
- Correct sequence of commands for blinking the LED.
- Readable and executable format for the CodeRunner system.

Students can review and, if necessary, edit the code before submission. This stage reinforces the connection between visual programming and traditional text-based coding, helping learners transition to writing code independently.

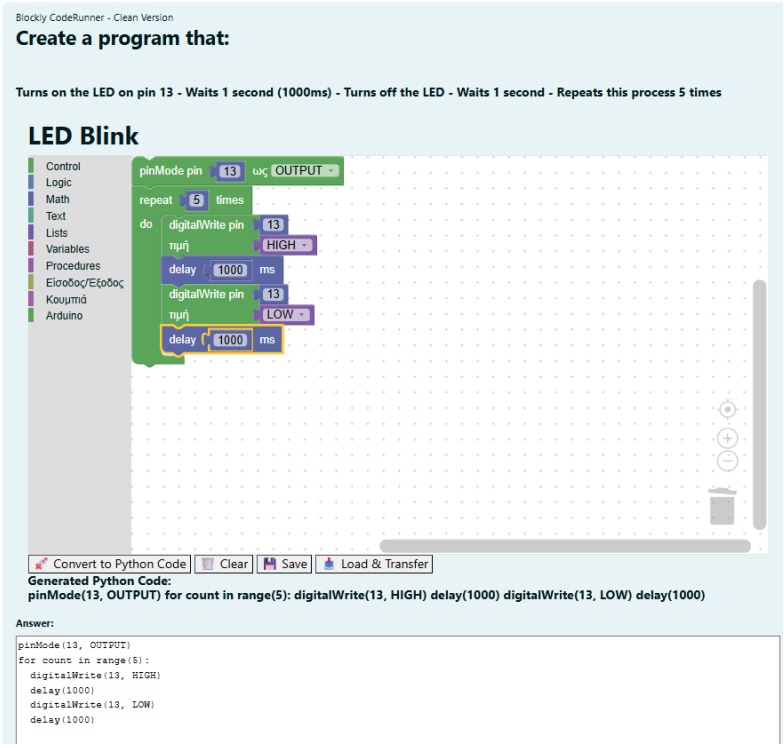


Figure 106. Converted code Case 2

Code Execution and Output Verification

Once it is verified that the final code appears correctly in the answer field, as shown in the Figure 107, the next step is to run the program using the integrated testing environment, as shown in Figure 117.

This code simulates controlling an LED connected to a digital pin, following a pattern similar to Arduino-style programming. First, the instruction pinMode(13, OUTPUT) sets pin 13 as an output pin. This means it can be used to send signals (such as turning an LED on or off).

Next, the for loop runs five times using range(5). This controls how many times the LED will blink. Inside the loop:

- digitalWrite(13, HIGH) turns the LED on by sending a high signal to pin 13.
- delay(1000) pauses the program for 1000 milliseconds (1 second), keeping the LED on during that time.
- digitalWrite(13, LOW) turns the LED off by sending a low signal.
- Another delay(1000) pauses the program for 1 second, keeping the LED off.

Overall, the code makes the LED blink on and off every second, repeating this cycle five times.



Figure 107. Python code for Case 2

Test Results and Validation

The final stage, shown in Figure 108, presents the results of the program's execution within the automated evaluation system. The results panel indicates that all test cases were successfully completed, displaying each input alongside its expected and actual output for verification.

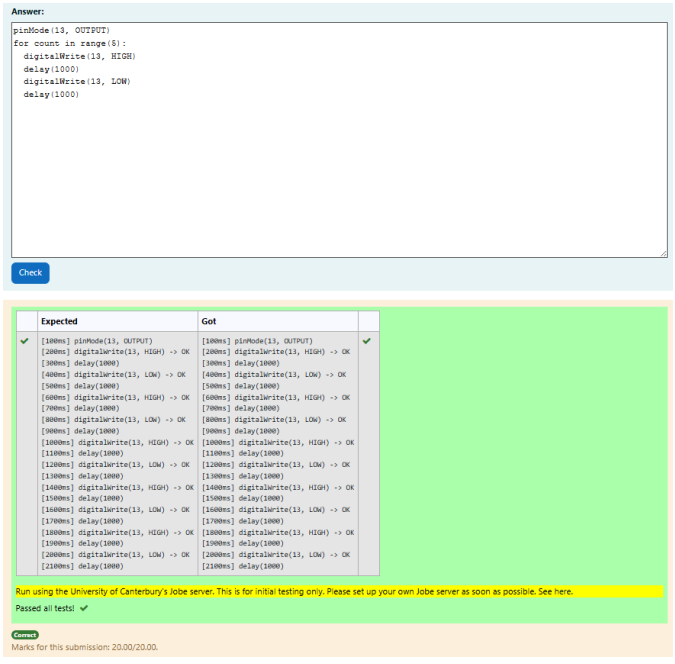


Figure 108. Test Results & Validation for Case 2

Error Detection and Debugging — Exercise 2

Error Handling and Debugging

Figure 109 shows an intermediate step in which the program ran into a syntax error during execution. The error message reveals that a parenthesis was left unclosed in one of the print statements, causing the program to fail to compile and run.

The first figure highlights a syntax error encountered during code execution. The error message indicates that a parenthesis in the statement pinMode(13, OUTPUT was not properly closed. This prevented the program from compiling and running. Such errors are typically identified by the interpreter before execution, and resolving them requires correcting the code structure—in this case, by adding the missing closing parenthesis.

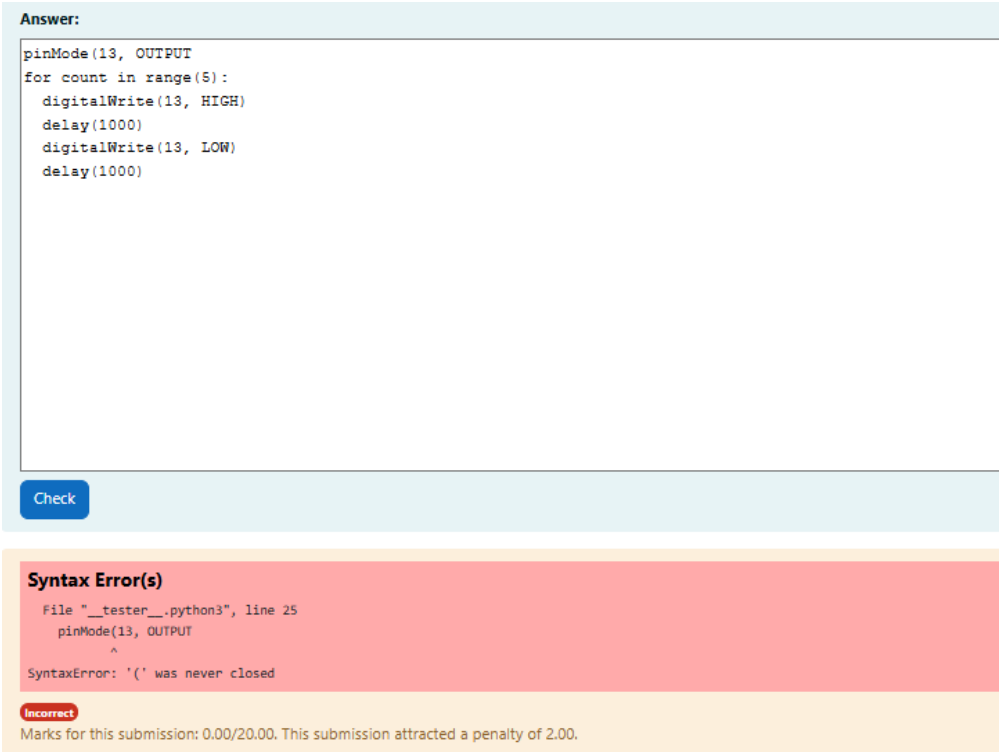


Figure 109. Syntax Error Case 2

Final Corrected Execution

The second Figure 110, shows the program after fixing the syntax issue, allowing it to execute. However, a logical error remains in the sequence of operations. The expected output requires the LED to switch LOW → HIGH, but the implemented code performs HIGH → LOW, resulting in a mismatch between expected and actual behavior.

Although the program runs successfully, the incorrect order of instructions leads to unintended output, demonstrating the importance of verifying logic in addition to syntax.

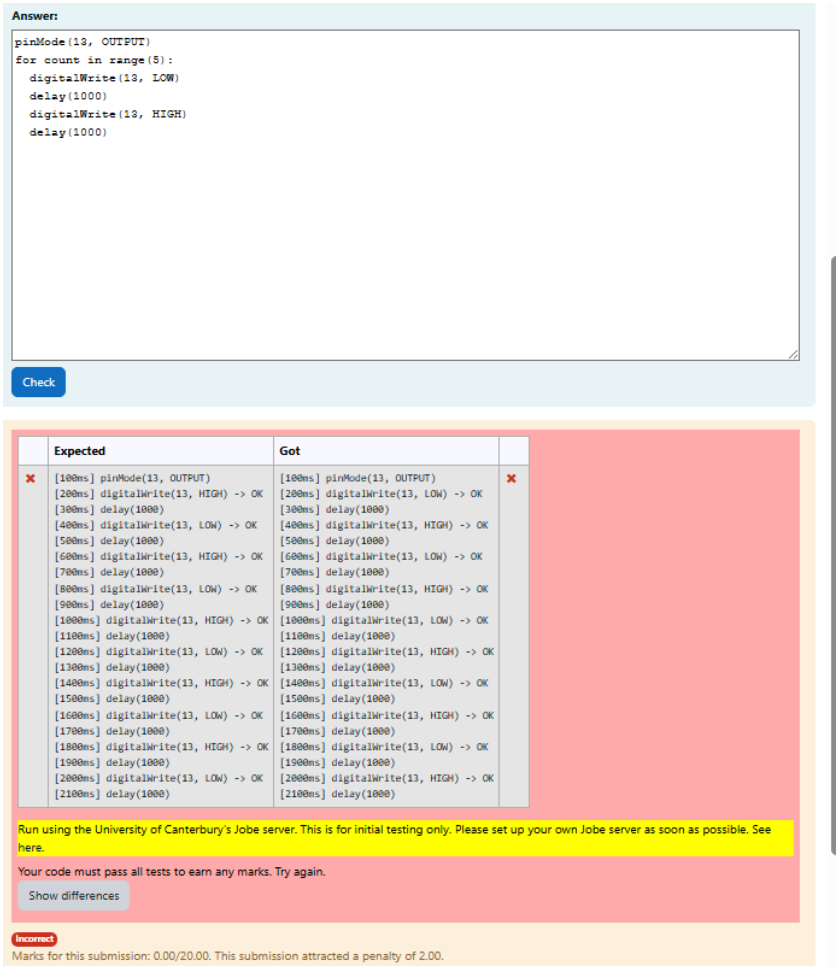


Figure 110. Logical Error Case 2

6.3.3 Exercise 3 — Arduino IoT with Bluetooth

The third exercise represents the most advanced scenario: a complete Arduino IoT setup using Bluetooth communication with an HC-05 module. The program initializes serial communication, configures input and output pins, and tests an LED by turning it on and off while sending its status via Bluetooth. It also reads data from a temperature sensor and a light sensor, printing the values to the serial monitor and transmitting them wirelessly.

The exercise uses the IoT Edition of the Blockly toolbox, containing 14 categories including App Inventor, Connectivity, Sensors, Arduino, and IoT Smart Home blocks and shows how to combine hardware control, sensor data collection, and Bluetooth communication in a basic IoT application.

Expanded Block Architecture — Exercise 3

1. HTML Structure & Toolbox

This is the IoT Edition — the most complete of the three files. It builds on the previous two and adds three entirely new block categories to the toolbox. In total the toolbar now contains: Control, Logic, Math, Text, Lists, Variables, Procedures, Input/Output, Button, App Inventor, Connectivity, Sensors, Arduino, and IoT Smart Home — 14 categories in all, as shown in Figure 111.

```
<!-- App Inventor Components -->
<category name="App Inventor" colour="200">
    <!-- Label Components -->
    <block type="app_label_set_text"></block>
    <block type="app_label_get_text"></block>
    <block type="app_label_set_color"></block>
    <block type="app_label_set_size"></block>

    <!-- Notifier Components -->
    <block type="app_notifier_show_message"></block>
    <block type="app_notifier_show_alert"></block>
    <block type="app_notifier_show_choose_dialog"></block>

    <!-- Bluetooth Components -->
    <block type="app_bluetooth_send_text"></block>
    <block type="app_bluetooth_receive_text"></block>
    <block type="app_bluetooth_is_connected"></block>

    <!-- Button Events -->
    <block type="app_button_when_clicked"></block>

    <!-- App Inventor Specific Buttons -->
    <block type="app_led_on_button_click"></block>
    <block type="app_led_off_button_click"></block>
    <block type="app_temp_button_click"></block>
    <block type="app_light_button_click"></block>
</category>
```

```
<!-- ============ CONNECTIVITY ============ -->
<category name="Connectivity" colour="45">
    <!-- Serial Communication -->
    <block type="serial_begin"></block>
    <block type="serial_available"></block>
    <block type="serial_read"></block>
    <block type="serial_write"></block>
    <block type="serial_print"></block>
    <block type="serial_println"></block>

    <!-- Bluetooth Client -->
    <block type="bluetooth_client_connect"></block>
    <block type="bluetooth_client_send"></block>
    <block type="bluetooth_client_receive"></block>
    <block type="bluetooth_client_disconnect"></block>
    <block type="bluetooth_client_connected"></block>

    <!-- Web/HTTP -->
    <block type="web_http_get"></block>
    <block type="web_http_post"></block>
</category>

<!-- ============ SENSORS - ΑΙΣΘΗΤΗΡΕΣ ============ -->
<category name="Sensors" colour="95">
    <!-- Proximity Sensor -->
    <block type="sensor_proximity_read"></block>
    <block type="sensor_proximity_detect"></block>
    <block type="sensor_proximity_distance"></block>

    <!-- Light Sensor -->
    <block type="sensor_light_read"></block>
    <block type="sensor_light_level"></block>

    <!-- Location/GPS -->
    <block type="sensor_location_get"></block>
    <block type="sensor_location_latitude"></block>
    <block type="sensor_location_longitude"></block>

    <!-- Accelerometer -->
    <block type="sensor_accelerometer_read"></block>
    <block type="sensor_accelerometer_x"></block>
    <block type="sensor_accelerometer_y"></block>
    <block type="sensor_accelerometer_z"></block>

    <!-- Temperature -->
    <block type="sensor_temperature_read"></block>

    <!-- Motion/PIR -->
    <block type="sensor_motion_detect"></block>
</category>
```

```
<!-- IoT Smart Home Blocks -->
<category name="IoT Smart Home" colour="50">
    <block type="iot_led_control"></block>
    <block type="iot_temperature_read"></block>
    <block type="iot_light_read"></block>
    <block type="iot_bluetooth_send"></block>
    <block type="iot_bluetooth_receive"></block>
    <block type="iot_smart_home_setup"></block>
    <block type="iot_main_loop"></block>
</category>
```

Figure 111. App Invento–Connectivity–Sensors- IoT Smart Home Components

2. initBlockly() — Workspace Initialization

One notable structural difference from the previous versions: here the workspace is injected first, and defineCustomBlocks() is called afterwards (not before). This avoids a potential race condition where block definitions are registered before the Blockly engine is fully ready. Everything else is the same: override Blockly.Python.finish, patch the variable name database, call setupEventListeners(), and run the initial generateCode(), see Figure 112.

```
function initBlockly() {
    try {
        console.log("🚀 Starting Blockly initialization...");

        // Initialize workspace FIRST
        workspace = Blockly.inject('blocklyDiv', {
            toolbox: document.getElementById('toolbox'),
            grid: {
                spacing: 20,
                length: 3,
                colour: '#ccc',
                snap: true
            },
            trashcan: true,
            zoom: {
                controls: true,
                wheel: true,
                startScale: 1.0,
                maxScale: 3,
                minScale: 0.3,
                scaleSpeed: 1.2
            }
        });

        if (!workspace) {
            throw new Error("Failed to create Blockly workspace");
        }

        console.log("☑ Workspace created successfully!");

        // Define custom blocks AFTER workspace creation
        defineCustomBlocks();
```

Figure 112. Code Segment of initBlockly()

3. defineCustomBlocks() — All Block Definitions

This is the most expanded version (see Figure 113.), with five distinct block groups:
Basic custom blocks (same as before): input_number, input_text, all four button event blocks.
Arduino blocks (same as Exercise 2): arduino_pin_mode, arduino_digital_write, arduino_digital_read, arduino_analog_read, arduino_analog_write, arduino_delay, arduino_millis, arduino_serial_print, arduino_serial_println, arduino_servo_attach, arduino_servo_write, arduino_tone, arduino_constant, arduino_pin_constant.
Connectivity blocks (new in this file): Serial communication (serial_begin, serial_available, serial_read, serial_write, serial_print, serial_println), Bluetooth client operations (bluetooth_client_connect, bluetooth_client_send, bluetooth_client_receive, bluetooth_client_disconnect, bluetooth_client_connected), and Web/HTTP (web_http_get, web_http_post).
App Inventor blocks (new in this file): Label components (app_label_set_text, app_label_get_text, app_label_set_color with a colour dropdown, app_label_set_size), Notifier components (app_notifier_show_message, app_notifier_show_alert, app_notifier_show_choose_dialog), Bluetooth components (app_bluetooth_send_text, app_bluetooth_receive_text, app_bluetooth_is_connected), a generic app_button_when_clicked, and four specific IoT app buttons: LED ON, LED OFF, Temperature, and Brightness.
IoT Smart Home blocks (new in this file): iot_led_control (ON/OFF dropdown), iot_temperature_read, iot_light_read, iot_bluetooth_send, iot_bluetooth_receive, iot_smart_home_setup (setup function container), and iot_main_loop (loop function container).
Sensor blocks (new in this file): Proximity (sensor_proximity_read, sensor_proximity_detect, sensor_proximity_distance), Light (sensor_light_read, sensor_light_level), GPS (sensor_location_get, sensor_location_latitude, sensor_location_longitude), Accelerometer (sensor_accelerometer_read, plus X/Y/Z axis blocks), Temperature (sensor_temperature_read), and Motion/PIR (sensor_motion_detect).

```
// ============ APP INVENTOR BLUETOOTH COMPONENTS ============

// Bluetooth Send Text
Blockly.Blocks['app_bluetooth_send_text'] = {
    init: function() {
        this.appendValueInput("MESSAGE")
            .setCheck("String")
            .appendField("Send Bluetooth text");
        this.setPreviousStatement(true, null);
        this.setNextStatement(true, null);
        this.setColour(200);
        this.setTooltip("Sends text via Bluetooth");
    }
};

// Bluetooth Receive Text
Blockly.Blocks['app_bluetooth_receive_text'] = {
    init: function() {
        this.appendDummyInput()
            .appendField("Bluetooth text reception");
        this.setOutput(true, "String");
        this.setColour(200);
        this.setTooltip("Receives text from Bluetooth");
    }
};

// Bluetooth Is Connected
Blockly.Blocks['app_bluetooth_is_connected'] = {
    init: function() {
        this.appendDummyInput()
            .appendField("Bluetooth is connected");
        this.setOutput(true, "Boolean");
        this.setColour(200);
        this.setTooltip("Checks if Bluetooth is connected");
    }
};
```

Figure 113. Code Segment of defineCustomBlocks ()

4. definePythonGenerators() — Code Generation Rules

Covers all the same generator groups as before, plus three new ones specific to this file:

Connectivity generators (see Figure 114.): serial functions map to serial_print(), serial_begin(), serial_read(), etc.; Bluetooth maps to bluetooth_connect(), bluetooth_send(), bluetooth_receive(), bluetooth_disconnect(), bluetooth_connected(); HTTP maps to http_get(url) and http_post(url, data).

App Inventor generators: Label blocks produce app_label_set_text(), app_label_get_text(), etc.; Notifier blocks produce app_notifier_show_message(), app_notifier_show_alert(), app_notifier_show_choose_dialog(); Bluetooth blocks produce app_bluetooth_send_text(), app_bluetooth_receive_text(); the four specific buttons generate named Python functions: LED_ON_BUTTON_Click(), LED_OFF_BUTTON_Click(), TEMP_BUTTON_Click(), LIGHT_BUTTON_Click().

IoT Smart Home generators: iot_led_control maps directly to digitalWrite(13, ON/OFF) (hardcoded pin 13); iot_temperature_read → temperature_read(); iot_light_read → light_sensor_read(); iot_smart_home_setup generates a def arduino_setup(): function; iot_main_loop generates a def arduino_loop(): function.

Sensor generators: Each sensor maps to a clean Python function call — proximity_read(), light_sensor_read(), gps_get_location(), gps_latitude(), gps_longitude(), accelerometer_x/y/z(), temperature_read(), motion_detect().

After registering all generators, the code runs a verification loop that checks 10 key block types via console.log to confirm they are correctly registered before the user starts working.

```
// ============ IOT SMART HOME GENERATORS ============

Blockly.Python['iot_led_control'] = function(block) {
    const state = block.getFieldValue('STATE');
    return `digitalWrite(13, ${state})\n`;
};

Blockly.Python['iot_temperature_read'] = function(block) {
    return ['temperature_read()', Blockly.Python.ORDER_FUNCTION_CALL];
};

Blockly.Python['iot_light_read'] = function(block) {
    return ['light_sensor_read()', Blockly.Python.ORDER_FUNCTION_CALL];
};

Blockly.Python['iot_bluetooth_send'] = function(block) {
    const message = Blockly.Python.valueToCode(block, 'MESSAGE', Blockly.Python.ORDER_ATOMIC) || '""';
    return `bluetooth_send(${message})\n`;
};

Blockly.Python['iot_bluetooth_receive'] = function(block) {
    return ['bluetooth_receive()', Blockly.Python.ORDER_FUNCTION_CALL];
};

Blockly.Python['iot_smart_home_setup'] = function(block) {
    const setup = Blockly.Python.statementToCode(block, 'SETUP');
    const indentedSetup = Blockly.Python.prefixLines(setup || 'pass', Blockly.Python.INDENT);
    const code = `def arduino_setup():\n${indentedSetup}\n\n`;
    return code;
};

Blockly.Python['iot_main_loop'] = function(block) {
    const loop = Blockly.Python.statementToCode(block, 'LOOP');
    const indentedLoop = Blockly.Python.prefixLines(loop || 'pass', Blockly.Python.INDENT);
    const code = `def arduino_loop():\n${indentedLoop}\n\n`;
    return code;
};
```

Figure 114. Code Segment of definePythonGenerators()

5. setupEventListeners() — Button Wiring

Same four buttons (Convert, Clear, Save, Load & Transfer), but the Load & Transfer logic here is significantly simplified compared to the previous two files, illustrated in Figure 115 — it loads the XML, clears the workspace, renders the blocks, calls generateCode(), and then fires generateAndTransferCode(null, true), as shown in Figure 116, after a 500ms timeout. No intermediate error pop-ups or step-by-step console logging. The real-time listener also only

tracks BLOCK_MOVE, BLOCK_CREATE, and BLOCK_DELETE (not BLOCK_CHANGE like the other two).

```
<!-- Buttons -->
<div class="controls" bis_skin_checked="1">
    <button type="button" id="convertCode">🚀 Convert</button>
    <button type="button" id="clearWorkspace">🗑️ Clear</button>
    <button type="button" id="saveWorkspace">💾 Save</button>
    <button type="button" id="loadAndTransfer">📥 Load &amp;amp; Transfer </button>
</div>
```

Figure 115. Button Wiring

```
function generateAndTransferCode(event, isAutomatic = false) {
    if (event) {
        event.preventDefault();
        event.stopPropagation();
    }

    if (!workspace) {
        console.error('❌ No workspace available');
        if (!isAutomatic) alert('Blockly workspace not initialized!');
        return false;
    }

    try {
        console.log('🚀 Starting code transfer process...');

        // Generate Python code
        let code = generateCode();

        if (!code || !code.trim()) {
            console.warn('⚠️ No code generated');
            if (!isAutomatic) alert('Δεν υπάρχει κώδικας για μεταφορά!');
            return false;
        }

        console.log('📝 Generated code:', code);

        // 🔍 Improved answer field search
        const findAnswerField = () => {
            console.log('🔍 Starting advanced answer field search...');

            const selectors = [
                // CodeRunner specific selectors
                "textarea[name*='answer']",
                "textarea[name='answer']",
                "textarea[id*='answer']",
                "textarea[id*='answertext']",
                "#id_answer",
                "#id_answertext",
```

Figure 116. Code Segment of generateAndTransferCode ()

6. generateCode() — Live Code Preview

Same logic as before with one improvement: on error it now also logs the full stack trace to the #codeOutput div, making it much easier to debug generator failures during development, as shown in Figures 117 and 118.

```
function generateCode() {
    if (!workspace) {
        console.error('❌ Workspace not available');
        return '';
    }

    try {
        console.log('🔄 Starting code generation...');

        // Initialize Python generator
        Blockly.Python.init(workspace);

        // Generate code
        let code = Blockly.Python.workspaceToCode(workspace);

        console.log('📝 Raw generated code:', code);

        // Clean up code
        code = code.replace(/\b(\w+) = None\s+\1 =/g, "$1 =");

        const outputElement = document.getElementById('codeOutput');
```

Figure 117. Code Segment of Blockly.Python.workspaceToCode ()

```
// Clean up code
code = code.replace(/\b(\w+) = None\s+\1 =/g, "$1 =");

const outputElement = document.getElementById('codeOutput');
if (code && code.trim()) {
    outputElement.textContent = code;
    console.log('☑ Code generation successful');
} else {
    outputElement.textContent = '# Κενό workspace - προσθέστε blocks';
    console.log('ℹ Empty workspace');
}

return code;
```

Figure 118. Code Segment of codeOutput element

7. generateAndTransferCode() — Transfer to Moodle/CodeRunner

This version has the most robust answer field search of all three files, using three separate search strategies in sequence, as shown in Figure 119. First it tries a comprehensive list of CSS selectors (including .formulation textarea, .ablock textarea, textarea[id*='answertext'], etc.) and also validates that the found element has non-zero pixel dimensions on screen. Second, if nothing is found, it searches inside all <iframe> elements on the page — useful when Moodle embeds the question inside an iframe.

Third, if still nothing is found, it performs content analysis — checking parent element HTML for keywords like "answer", "question", or "coderunner" and checking the textarea's placeholder, name, and id attributes. After transfer it fires only three events (input, change, blur) rather than the ten used in the previous files.

```
for (let element of elements) {
    if (element &&
        element.tagName === 'TEXTAREA' &&
        !element.disabled &&
        !element.readOnly &&
        element.offsetParent !== null &&
        element.style.display !== 'none' &&
        element.getBoundingClientRect().width > 0 &&
        element.getBoundingClientRect().height > 0) {

        console.log('☑ Candidate found:', element);
        console.log('📏 Element dimensions:', element.getBoundingClientRect());
        console.log('🏷 Element attributes:', {
            id: element.id,
            name: element.name,
            className: element.className,
            placeholder: element.placeholder
        });
```

Figure 119. Code Segment of generateAndTransferCode() – textarea tag

8. manualCodeTransfer() and showCodePopup() — Fallback Chain

A new dedicated manualCodeTransfer() function is introduced, shown in Figure 120, as a clean fallback layer. It first tries navigator.clipboard.writeText(), and only if that fails does it call showCodePopup(). The popup itself is slightly improved with a larger window (700×500 instead of 600×400) and a proper copyCode() function using document.execCommand('copy').

```
// Auxiliary function for manual transfer
function manualCodeTransfer(code, isAutomatic = false) {
    console.log('📋 Using manual code transfer');

    // Try clipboard first
    if (navigator.clipboard && navigator.clipboard.writeText) {
        navigator.clipboard.writeText(code).then(() => {
            const message = isAutomatic ?
                "📋 Code copied to clipboard! Paste with Ctrl+V" :
                "📋 Code copied to clipboard! ";
            alert(message);
        }).catch((err) => {
            console.error('Clipboard error:', err);
            showCodePopup(code);
        });
    } else {
        showCodePopup(code);
    }
}

// Function to display code in popup
function showCodePopup(code) {
    const popup = window.open('', '_blank', 'width=700,height=500,scrollbars=yes');
```

Figure 120. Code Segment of manualCodeTransfer () & showCodePopup()

9) Customisation Field Description

The customisation field, as shown in Figure 121, is responsible for **starting, running, and finalizing the IoT simulation environment** in which the student's code is executed.

It begins by printing a startup message:

print(" 🚀 STARTING IOT SIMULATION")

This indicates that the simulation environment has been initialized and is ready to run.

Next, the placeholder:

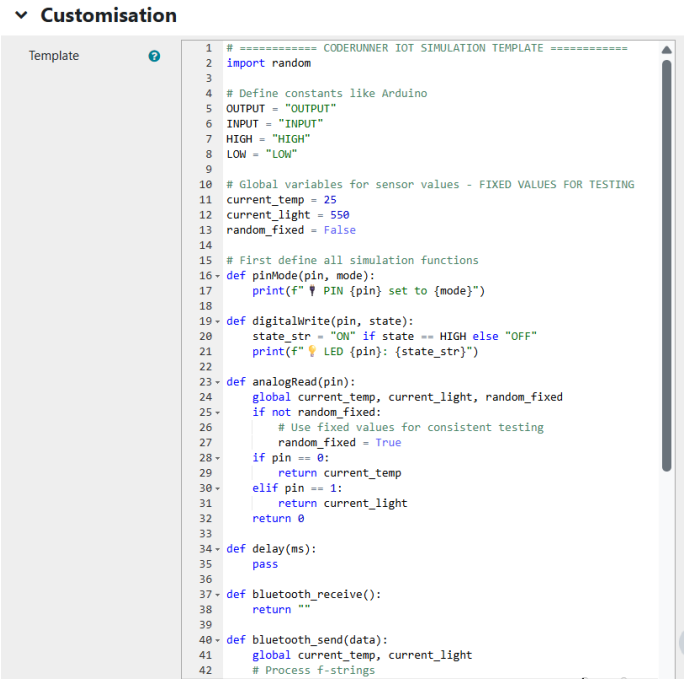
{{ STUDENT_ANSWER }}

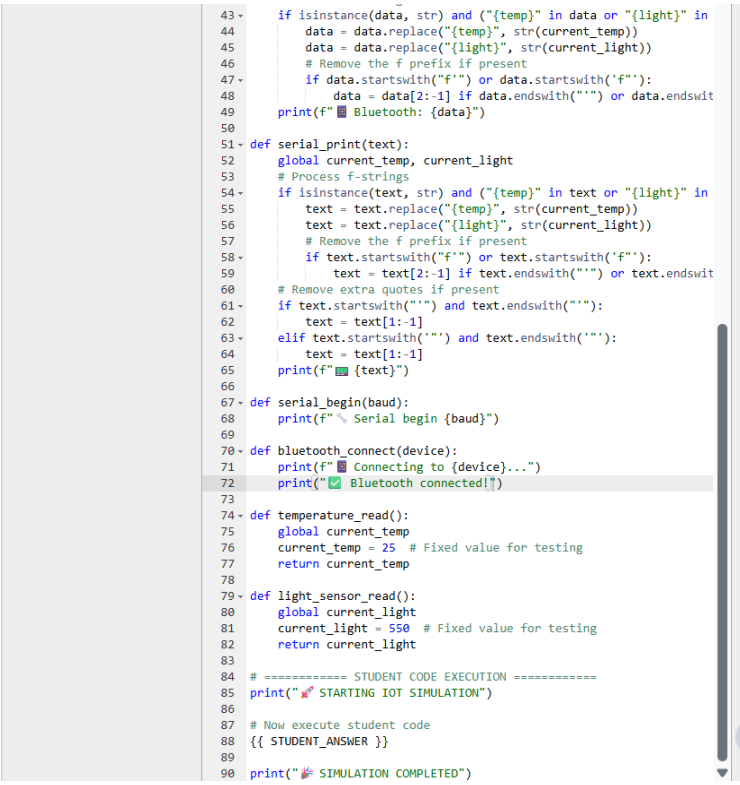
is the **core execution point**. In the CodeRunner plugin in Moodle, this placeholder is dynamically replaced with the code submitted by the student. When the program runs, the student's code is executed within the predefined simulation environment, meaning it can directly use the provided functions such as pinMode, digitalWrite, analogRead, serial_print, and others. This allows students to write Arduino-like IoT programs while the system simulates hardware behavior (e.g., LEDs, sensors, Bluetooth communication).

Finally, after the student's code has finished executing, the template prints:

print(" SIMULATION COMPLETED")

This marks the end of execution and provides a clear visual indication that the simulation has finished.





```
    if isinstance(data, str) and ("{temp}" in data or "{light}" in
        data = data.replace("{temp}", str(current_temp))
        data = data.replace("{light}", str(current_light))
        # Remove the f prefix if present
        if data.startswith("f'") or data.startswith('f"'):
            data = data[2:-1] if data.endswith("'") or data.endswit
    print(f" Bluetooth: {data}")

def serial_print(text):
    global current_temp, current_light
    # Process f-strings
    if isinstance(text, str) and ("{temp}" in text or "{light}" in
        text = text.replace("{temp}", str(current_temp))
        text = text.replace("{light}", str(current_light))
        # Remove the f prefix if present
        if text.startswith("f'") or text.startswith('f"'):
            text = text[2:-1] if text.endswith("'") or text.endswit
    # Remove extra quotes if present
    if text.startswith("'") and text.endswith("'"):
        text = text[1:-1]
    elif text.startswith('"') and text.endswith('"'):
        text = text[1:-1]
    print(f" {text}")

def serial_begin(baud):
    print(f" Serial begin {baud}")

def bluetooth_connect(device):
    print(f" Connecting to {device}...")
    print(" Bluetooth connected!")

def temperature_read():
    global current_temp
    current_temp = 25  # Fixed value for testing
    return current_temp

def light_sensor_read():
    global current_light
    current_light = 550  # Fixed value for testing
    return current_light

# ============ STUDENT CODE EXECUTION ============
print(" STARTING IOT SIMULATION")

# Now execute student code
{{ STUDENT_ANSWER }}

print(" SIMULATION COMPLETED")
```

Figure 121. Customisation Field Case 3

10) Answer Field Case 3

The Answer Field, as shown in Figure 122, refers to the section within a Moodle CodeRunner question where learners input their Python program. In this specific exercise, the required response demonstrates the process of receiving two integer values as input, performing basic arithmetic computations, and displaying the resulting outputs.

The model solution that is expected to appear in the answer field is presented below:

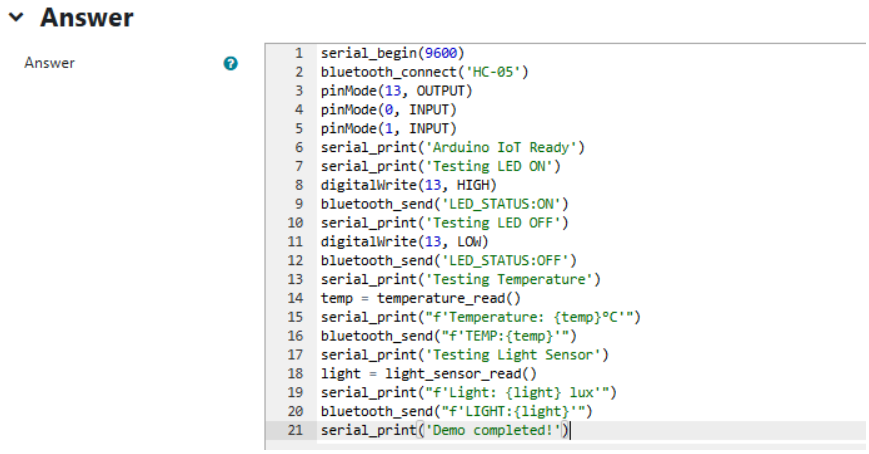


Figure 122. Answer Field Case 3

11) Test Cases

Test cases, as shown in Figure 123, are employed within Moodle CodeRunner to automatically evaluate the code provided by students. Each test case defines specific input values alongside the expected output, as well as configuration parameters that control how results are displayed and how marks are assigned.

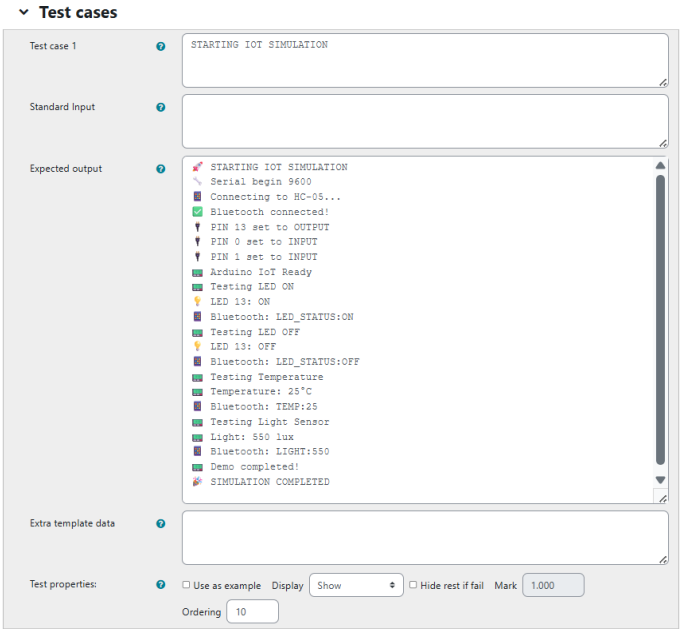


Figure 123. Test Case 3 – Configuration

Results and Output

This section presents the results obtained from executing the developed CodeRunner activity, along with the system's operation during the code generation and transfer processes.

The figures illustrate both the graphical block-based representation and the corresponding Python code produced from it.

Code Construction and Execution — Exercise 3

This Figure 124, shows the use of a Blockly-based interface to visually create an Arduino IoT program. Users assemble logic by dragging and connecting blocks for serial communication, Bluetooth connectivity (HC-05), LED control, and sensor readings (temperature and light). Once the program is complete, it is automatically converted into Python-style code, allowing users to see the textual equivalent of their visual design. This approach helps beginners understand programming concepts while bridging the gap between graphical and text-based coding.

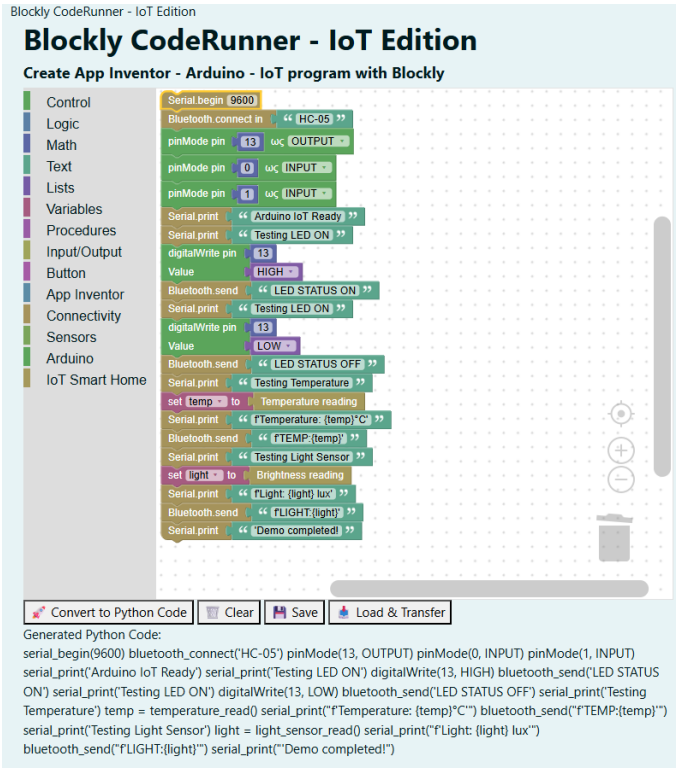


Figure 124. Test Case 3 – Configuration

Code Transfer and System Feedback

After the code is generated, the user selects the “Load & Transfer” button. At this stage:

- The Python code produced is automatically inserted into the Moodle CodeRunner answer field.
- The system instantly confirms the action with a message (e.g., “The code was successfully transferred to the response field!”), as shown in Figure 125.
- This process guarantees that the visual program is accurately converted and submitted as text.

By automating the transfer, the system minimizes mistakes from manual copying and streamlines the interaction between Blockly and CodeRunner.

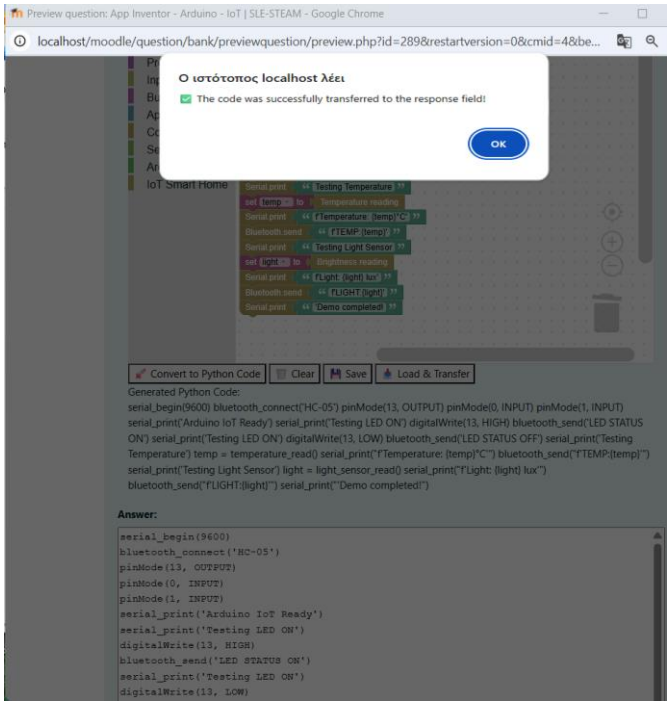


Figure 125. Confirmation message Case 3

Final Code Representation in Answer Field

After the transfer is completed, the generated Python code is displayed in the answer section, as illustrated in Figure 126. The code preserves the same structure as the automatically generated version, ensuring consistency between the visual programming environment and its textual form.

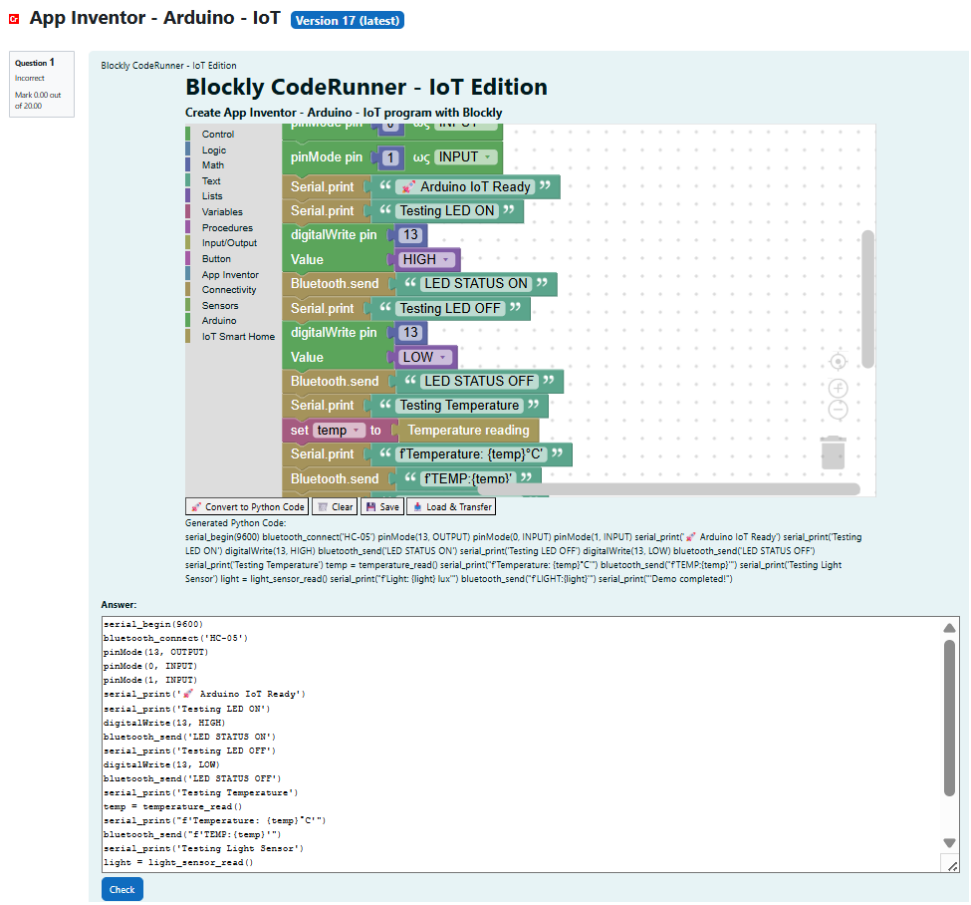


Figure 126. Converted code Case 3

Code Execution and Output Verification

After confirming that the final code is properly displayed in the answer field, the next step is to execute it within the integrated testing environment, as illustrated in Figure 127. The presented code demonstrates a simulated IoT application within the Moodle CodeRunner environment, combining Blockly-based programming with automatically generated Python code.

It emulates an Arduino system that connects to a Bluetooth module (HC-05), controls an LED, and reads data from temperature and light sensors. The program initializes communication, configures input/output pins, and performs test actions such as turning the LED on and off while sending status updates via serial and Bluetooth. It also collects and transmits sensor data, showcasing the integration of sensing, processing, and communication. Overall, the example highlights the transition from visual programming to text-based code and represents a complete IoT workflow in an educational setting.



Figure 127. Python code for Case 3

Test Results and Validation

The final stage, illustrated in Figure 128, displays the results obtained after executing the program within the automated evaluation system.

The results panel shows that all test cases were completed successfully. Each input is paired with its expected output, and the program's actual results are presented alongside for verification.



Figure 128. Test Results & Validation for Case 3

Error Detection and Debugging — Exercise 3

Error Handling and Debugging

Figure 129 illustrates an intermediate stage where the program encountered a syntax error during execution. The error message shows that a parenthesis was not properly closed in one of the statements, which caused the program to fail to compile and run. The figure highlights this issue, specifically in the command pinMode(13, OUTPUT, where the closing parenthesis is missing. Errors of this type are usually detected by the interpreter before execution begins, and fixing them requires correcting the code syntax—in this case, by adding the missing parenthesis.



Figure 129. Syntax Error Case 3

Final Corrected Execution

The second Figure 130, presents the program after correcting the syntax error, which allows it to run successfully. However, logical errors are still present in the sequence of operations. The expected behavior requires digitalWrite on pin 13 to be set to HIGH when the LED is ON and to LOW when the LED is OFF, but the implemented code does the opposite, leading to a discrepancy between the expected and actual results.

Although the program executes without errors, the incorrect instruction order produces unintended output, highlighting the importance of checking program logic in addition to syntax correctness.



Figure 130. Logical Error Case 3

Synthesis: The Role of Exercise Walkthroughs in System Evaluation

The three exercise walkthroughs collectively validate the end-to-end operation of the Blockly-CodeRunner integration across the full range of programming complexity targeted by the SLE. Each walkthrough demonstrates the system fulfilling its core evaluation functions: automated generation of syntactically correct Python from visual blocks; reliable transfer to the CodeRunner assessment engine; deterministic test case evaluation with immediate structured feedback; and differentiated error diagnosis distinguishing syntax errors from logical faults.

6.4 Summary

This chapter has presented the evaluation of the Smart Learning Ecosystem for interactive STEAM education, combining technical documentation of the Blockly-CodeRunner integration with operational validation through representative exercise walkthroughs.

Section 6.1 documented the Blockly visual programming environment as the primary student-facing programming interface, describing the two-panel workspace design, the progressive toolbox structure across course modules, and the four student-facing controls that support code construction, transfer, persistence, and reuse. Section 6.1.1 described the CodeRunner assessment interface, detailing the all-or-nothing grading model, the penalty regime for multiple attempts, and the differentiated feedback architecture that distinguishes syntax errors from incorrect output.

Section 6.2 provided the full technical architecture of the CodeRunner programming exercises, documenting the question configuration, the Blockly integration system — including block definitions, Python generators, event handling, and the DOM-based code transfer mechanism — and the test case configuration framework that enables deterministic automated grading.

Section 6.3 presented complete walkthroughs of the three representative exercises. Exercise 1 introduced fundamental arithmetic operations using the standard Blockly toolbox. Exercise 2 extended this to Arduino hardware control with LED blinking logic and a simulation template enabling grading without physical hardware. Exercise 3 demonstrated the most complex scenario — a full IoT application combining Bluetooth communication, sensor reading, and smart home control using all fourteen block categories of the IoT edition toolbox. Each walkthrough validated the end-to-end Blockly-CodeRunner pipeline under real pilot conditions, demonstrating successful code construction, automated transfer, test case evaluation, structured error feedback, and differentiation between syntax and logical errors.

Chapter 7 – Gamification and Analytics Reporting in our SLE

This chapter presents the gamification and analytics reporting components of the Smart Learning Ecosystem (SLE), with particular emphasis on their design, implementation, and integration within the Moodle-based platform. It examines how gamification mechanisms—including XP accumulation, level progression, badges, competency recognition, and conditional content unlocking—have been incorporated to enhance learner engagement, motivation, and personalized progression. Furthermore, the chapter describes the underlying data collection, assessment, and analytics infrastructure that supports monitoring and evaluation within the ecosystem, as well as the different assessment modalities implemented in the SLE. Special attention is also given to the Analytics Graphs Plugin, which extends the core analytics capabilities through advanced visual reporting features. Through these sections, the chapter provides a comprehensive overview of both the pedagogical and technical implementation of the gamification and analytics reporting framework within the Smart Learning Ecosystem.

7.1 Gamification Elements: XP, Levels, and Badges

The gamification layer of the SLE complements formal assessment by providing continuous motivational feedback through a progression-based reward system. Integrated into the course interface via the Level Up XP block, it displays each student's XP total current level shown in Figure 131, (from Novice Explorer at Level 1 through to SLE Champion at Level 10 – shown in Figure 132), and progress toward the next level. XP is awarded automatically for engagement in learning activities, including five points for each CodeRunner quiz submission and two points for each H5P Game Map attempt.

The leaderboard component, as shown in Figure 133, provides students with a social comparison mechanism, displaying the XP totals and level designations of all enrolled peers. The leaderboard was configured to display level badges prominently alongside XP totals, supporting both competitive motivation (for students focused on ranking) and mastery motivation (for students focused on personal level advancement).

Badges represent a second gamification mechanism operating alongside the XP system, awarded automatically upon the attainment of specific competency thresholds and visible in the student's Moodle profile and the course badge gallery.

The interaction between gamification and content access represents a third gamification element from the student perspective. As students accumulate XP and advance through levels, specific enrichment content becomes conditionally available based on level thresholds: advanced Blockly challenge activities unlock at Level 3, and early access to Capstone Project materials becomes available from Level 5. This creates a differentiated engagement pathway in which sustained, effortful participation in formative activities generates access to additional learning opportunities, reinforcing the motivational alignment between engagement and progression.

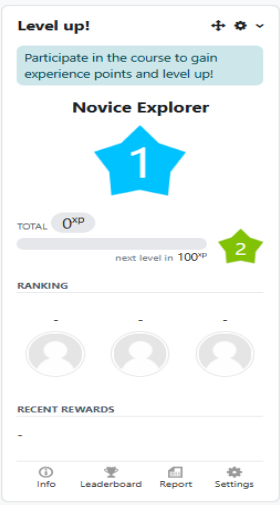
7.2 Gamification Implementation in Practice

The gamification framework configured during the development phase described in Chapter 4 was operationalized throughout the course as an embedded motivational scaffold. This section describes how the gamification components — the Level Up XP system, the badge framework, and the conditional unlocking mechanisms — functioned in practice during the pilot study, and how learner engagement with the gamification layer was observed and monitored.

7.2.1 XP Accumulation and Level Progression

Experience points were automatically allocated to learners throughout the course in response to their interaction with designated activity types. The two primary XP award rules configured during development — the Quiz Submission Rule awarding 5 XP per CodeRunner quiz attempt and the Interactive Activity Completion Rule awarding 2 XP per H5P Game Map attempt — generated a continuous stream of XP events as learners progressed through the course modules.

The visible XP block, shown in Figure 131, displayed within the course interface provided learners with a persistent progress indicator showing their current XP total, their current level designation, and a progress bar indicating the XP required to reach the next level. This transparency of progression was intended to support goal-setting and self-regulated learning behaviors, consistent with the motivational design principles underlying the gamification framework.



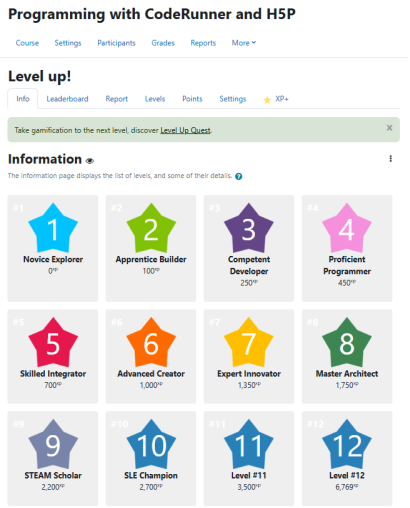


Figure 131. Level up - block-full-seed **Figure 132.** Level up – Point levels

The ten-level, shown in the above Figure 132, progression structure defined during configuration was observed to function as intended throughout the pilot study. The majority of actively engaged learners progressed through the early levels relatively rapidly during the initial modules, with progression rate slowing in later modules as XP thresholds increased. This snowballing threshold design was consistent with the intended motivational dynamic, maintaining challenge and providing visible milestones for high-engagement learners.

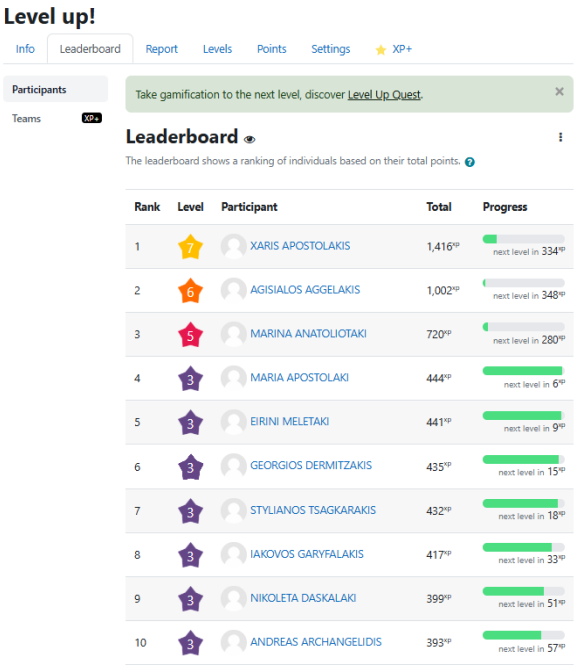


Figure 133. Level up – Leaderboard

7.2.2 Badge Issuance and Competency Recognition

Automated badge issuance was active throughout the pilot study, with badges awarded upon satisfaction of the predefined competency criteria described in Chapter 4. The five custom badge types — Loop Master, Sensor Sage, Debugging Detective, Peer Mentor, and Integration Innovator — were tracked through Moodle's event-based badge criteria system, and awarded badges appeared in learners' Moodle profile pages and the course badge gallery, illustrated in Figure 134.

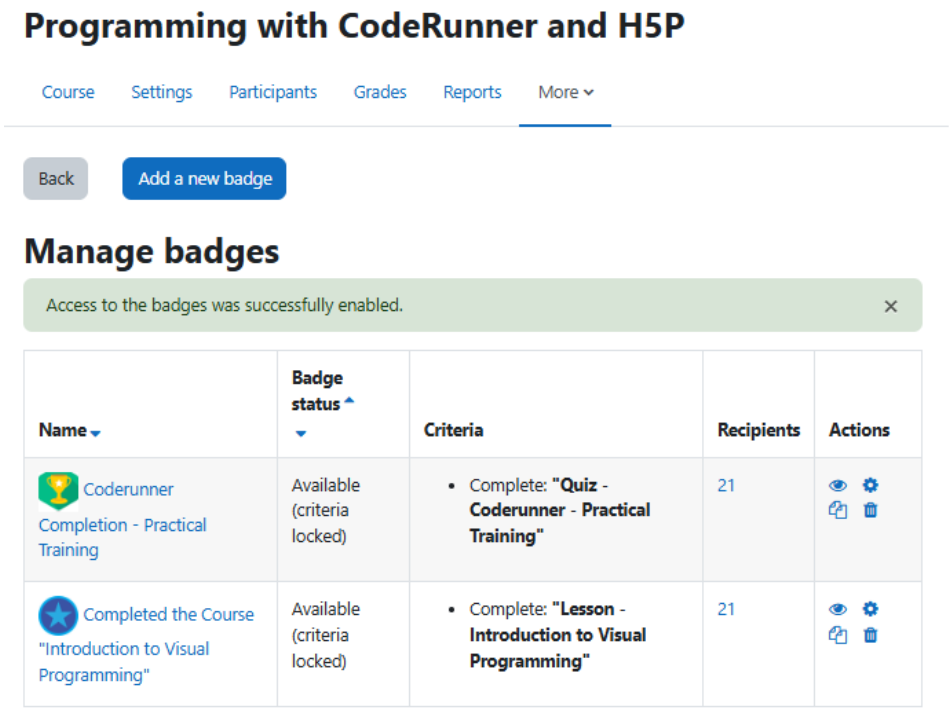


Figure 134. Badges management

7.2.3 Conditional Content Unlocking

In addition to the XP-based progression mechanics, conditional content unlocking was applied at key transition points within the course to reinforce mastery-based progression. Specifically, the advanced Blockly challenge activities — unlocked at Level 3 — and the early access to Capstone Projects — available from Level 5 — were implemented as conditional sections within the Moodle course, accessible only to learners whose Level Up XP level satisfied the defined threshold.

This conditional unlocking mechanism created a differentiated engagement pathway within the course: learners who engaged intensively with formative activities and accumulated XP rapidly gained access to enrichment content ahead of the standard course schedule, while learners progressing at a standard pace encountered the same content in the prescribed sequential order. This design supported differentiated learning within a single course environment without requiring administrative segmentation of the learner cohort.

7.3 Data Collection, Assessment, and Analytics Infrastructure

This section documents the complete assessment and analytics infrastructure embedded within the SLE, organized across two complementary layers: the core reporting and analytics capabilities built into Moodle, and the extended visualization capabilities provided by the Analytics Graphs plugin.

A central feature of the SLE is its ability to furnish instructors not only with a delivery medium, but also with an organized, multi-tiered framework enabling systematic monitoring of learner progress and data-informed instructional adaptation.

7.3.1 Assessment Modalities in the Smart Learning Ecosystem

Assessment within the SLE was conceptualised as a multi-functional process simultaneously serving three purposes: providing formative feedback to learners during learning, generating summative evidence of competency attainment at module boundaries, and producing the empirical datasets required for the study's research evaluation. These purposes are not mutually exclusive; rather, the assessment architecture was deliberately designed so that instruments serving pedagogical functions simultaneously generate the data required for research analysis.

This integrated design, shown in Table 19 necessitated the deployment of four complementary assessment modalities, each aligned with specific learning outcomes, pedagogical stages, and data collection objectives:

- **Rule-Based Automated Assessment:** Implemented through CodeRunner, this modality evaluates learner-submitted code against predefined test cases using objective, deterministic grading logic. A submission either satisfies a test case or it does not; partial credit is awarded proportionally to the number of passing tests.

This approach ensures consistency, scalability, and immediacy of feedback, eliminating assessor subjectivity and enabling high-frequency formative evaluation without instructor grading overhead. Attempt logs, test-case pass/fail sequences, and score trajectories constitute the primary data outputs of this modality.

- **Competency-Based Progression Assessment:** Realised through the conditional activity release mechanism and the Level Up XP framework, this modality evaluates whether learners have demonstrated sufficient mastery of prerequisite competencies before advancing to subsequent content.

  Rather than measuring performance at a single point in time, it monitors cumulative evidence of skill acquisition across multiple activity completions, quiz scores, and XP thresholds, gating access to advanced content based on demonstrated mastery. This operationalises a progression-oriented model of evaluation aligned with the mastery-based design of the SLE.

- **Formative Interactive Assessment:** H5P interactive content activities — including embedded quizzes, drag-and-drop exercises, and branching scenarios — serve as low-stakes formative instruments. Scores are recorded in the gradebook but are primarily intended to provide immediate conceptual feedback and to signal readiness for subsequent programming tasks. Attempt limits and automatic score recording ensure that engagement with formative activities is tracked without penalising iterative learning.

- **Reflective and Peer Assessment:** Forum discussion activities provide a qualitative assessment dimension, enabling learners to articulate their problem-solving processes, respond to peer contributions, and receive instructor feedback on their reflective engagement. This modality contributes to the Peer Mentor badge criteria and forum participation XP allocation, integrating reflective assessment within the gamification layer of the SLE.

Table 19. Assessment Modalities, Instruments, and Data Outputs in the SLE

Assessment Modality	Primary Instrument	Pedagogical Stage	Data Generated
Rule-Based Automated	CodeRunner Quizzes	Application, Evaluation	Attempt logs, test-case pass/fail, score trajectories
Competency-Based Progression	Conditional Release + Level Up XP	All stages	Completion events, XP totals, level milestones
Formative Interactive	H5P Activities	Conceptualization	Attempt scores, completion status, access frequency
Reflective & Peer	Forum Discussions	Reflection	Post counts, peer interaction records, badge awards

7.3.1 Moodle Core Analytics and Reporting

Moodle provides a comprehensive suite of built-in analytics and reporting tools forming the foundational layer of the SLE's monitoring infrastructure. These core tools collect data from all user interactions within the platform and make it available in a range of report formats.

Activity Logs, Reports, and Predictive Analytics

Moodle's logging subsystem records a detailed, timestamped event stream for all user interactions including logins, resource views, activity submissions, quiz attempts, forum posts, and file uploads, as shown in Figure 135.

The Live Logs view provides a real-time stream enabling immediate situational awareness during supervised sessions. The Activity Report provides a summary of access frequency and completion status, while the Course Participation Report enables aggregated comparison of engagement levels across all enrolled learners.

Beyond descriptive reporting, Moodle incorporates predictive analytics through its Learning Analytics API. The Students at Risk of Dropping Out predictor generates probabilistic risk indicators based on interaction pattern trajectories, enabling proactive identification of students requiring early intervention. The Moodle gradebook serves as the formal aggregation point for all assessment scores, configured with grading scales and aggregation rules aligned with the diverse activity types deployed in the SLE.

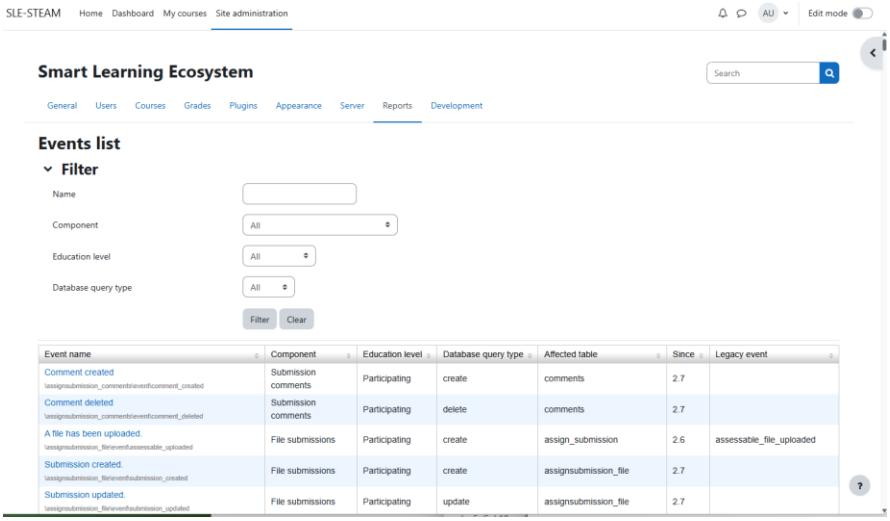


Figure 135. Moodle Core Logs Report — Event Stream Filtered by Activity

Activity Reports and Course Participation

The Activity Report, in Figure 136 provides a summary of access frequency and completion status for each activity within the course, broken down by individual learner. This report gives the instructor a rapid overview of which activities have been engaged with, by whom, and to what extent — supporting identification of under-engaged learners and frequently revisited content without requiring navigation through the raw event log.

The Course Participation Report extends this view by presenting an aggregated comparison of engagement levels across all enrolled learners, enabling the instructor to identify patterns of differential engagement between individual students and between the cohort subgroups defined in the study design. Within the SLE, this report was used to monitor whether engagement levels were consistent across the two learner subgroups and to flag any systematic differences requiring intervention.

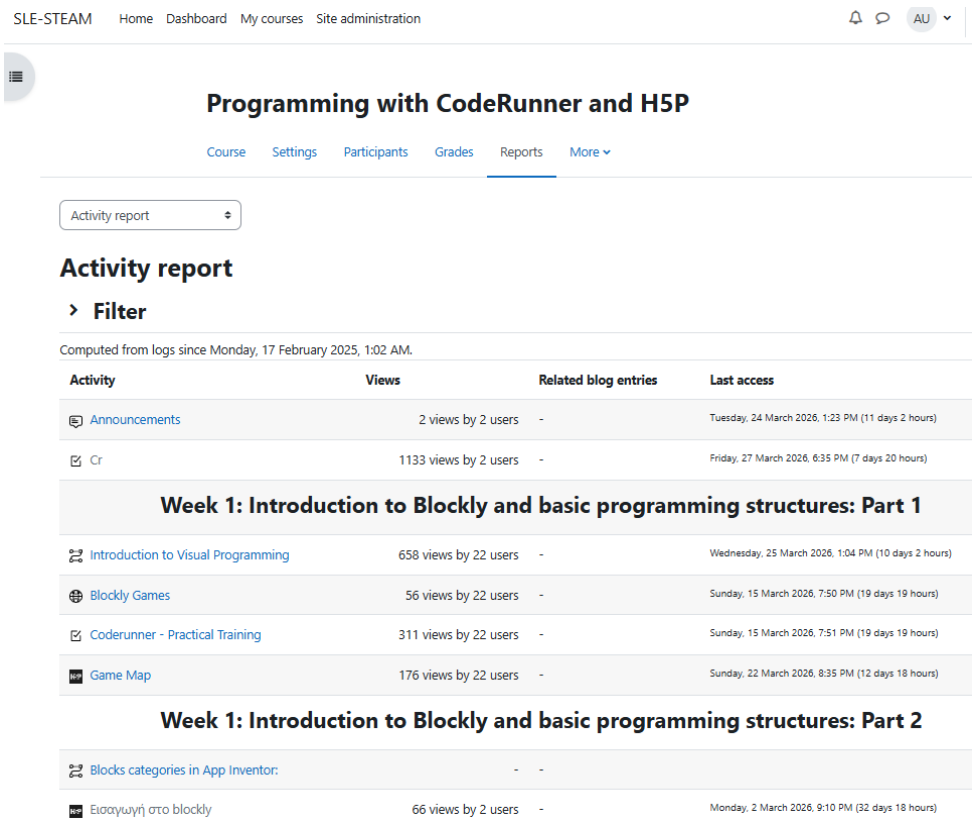


Figure 136. Moodle Core Logs Report — Event Stream Filtered by Activity

Activity Reports and Course Participation

Beyond descriptive reporting, Moodle incorporates a predictive analytics capability through its Learning Analytics API. A notable built-in model is the "Students at Risk of Dropping Out" predictor, which applies machine learning techniques to interaction data to identify learners who may require early intervention.

The analytics workflow supporting this model comprises four stages: data collection from user interactions, model training and evaluation against historical data, prediction generation for the current cohort, and the surfacing of intervention recommendations to the instructor.

Within the context of the SLE pilot study, the predictive model served as a complementary early-warning mechanism alongside the Activity Completion and Course Participation reports. Where these descriptive reports identify current engagement deficits, the predictive model surfaces probabilistic risk indicators based on interaction pattern trajectories, enabling the instructor to act proactively rather than reactively.

The combination of descriptive and predictive analytics, in Figure 137, within the core platform thus supports a data-informed intervention cycle aligned with the monitoring objectives of the research.

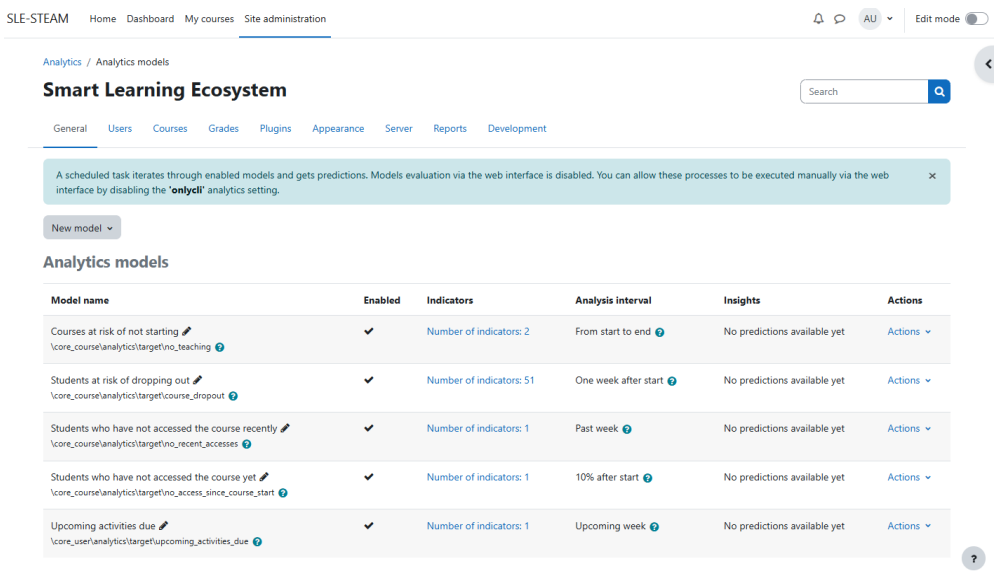


Figure 137. Moodle Predictive Analytics — Students at Risk Indicator Dashboard

Gradebook Reports

The Moodle gradebook, as shown in Figure 138, serves as the formal aggregation point for all assessment scores generated across the course. Configured during the development phase with grading scales and aggregation rules aligned with the diverse activity types deployed in the SLE — H5P interactive content, CodeRunner programming exercises, and summative quizzes — the gradebook combines individual activity scores into coherent module and course-level grade totals accessible to both instructors and learners.

Gradebook Reports provide performance tracking through grades, score distributions, averages, and feedback records. From the instructor's perspective, the gradebook offers a complementary view to the engagement-level data provided by activity logs: where logs reveal how learners are interacting with the course, the gradebook reveals how well they are performing. The combination of these two data sources enables the instructor to distinguish between learners who are engaging actively but struggling — potentially indicating a need for additional

conceptual scaffolding — and learners who are disengaged — potentially indicating motivational or accessibility barriers requiring a different form of intervention.

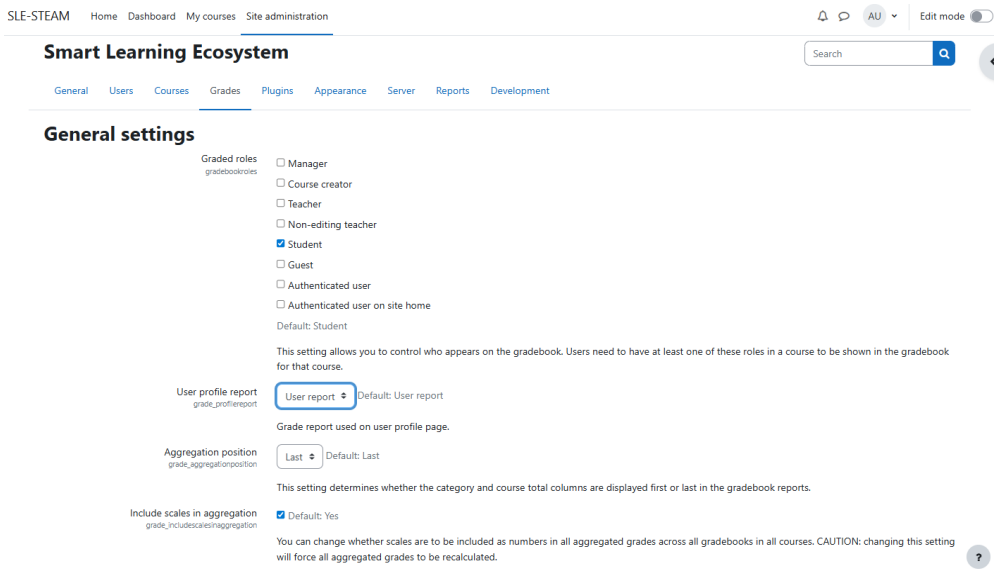


Figure 138. Moodle Gradebook — Module Grade Overview General Settings

Limitations of Core Moodle Reporting

Despite their comprehensiveness, Moodle's core analytics tools present limitations that constrain their usability for non-technical instructors in a fast-paced teaching context. Reports are predominantly text-heavy and tabular, lacking the intuitive graphical representations that enable rapid visual identification of engagement trends and anomalies. Extracting meaningful insights frequently requires navigation across multiple report types, and advanced customisation of report outputs is restricted without additional development.

These limitations reduce the accessibility of core analytics for routine instructional use, particularly in research contexts requiring frequent monitoring of multiple cohort metrics simultaneously. They constitute the primary motivation for the deployment of the Analytics Graphs plugin as a visual extension of the core reporting layer.

7.3.2 The Analytics Graphs Plugin: Visual Extension of Core Analytics

The Analytics Graphs plugin was installed to address the visualisation gap identified in Moodle's core reporting capabilities. Rather than replacing the core analytics layer, it extends it

by transforming the raw log, grade, and participation data already collected by Moodle into a set of intuitive graphical reports accessible directly within the course interface.

This integration makes the Analytics Graphs plugin not an alternative to core analytics but its visual counterpart — a lens through which the same underlying data becomes more rapidly interpretable and pedagogically actionable.

Accessible through the course Reports menu, the plugin was configured with instructor-level access permissions, cohort-based filtering capability, and data export functionality supporting downstream statistical analysis. These configuration choices reflected the dual purpose of the analytics infrastructure: supporting real-time pedagogical decision-making during the pilot study, and generating exportable datasets for the empirical evaluation of the SLE's research hypotheses.

Report Types and Visualisation Capabilities

The Analytics Graphs plugin, shown in Figure 139, provides the following visual report types, each rendering a specific dimension of learner engagement or performance data in graphical form:

- **Grades Chart:** A graphical distribution of grades across the cohort for selected gradable activities, rendered as a bar chart. This visualisation enables the instructor to assess score distributions at a glance, identify ceiling and floor effects, and evaluate the difficulty calibration of assessment items more efficiently than the tabular gradebook view alone permits.
- **Content Accesses:** A bar chart quantifying the number of times each content resource has been accessed by learners, enabling rapid identification of frequently referenced materials — potentially indicating conceptual difficulty — and under-utilised resources that may warrant redesign or promotion.
- **Number of Active Students:** A time-series line graph displaying the number of learners actively engaging with the course on each day of the study period. This visualisation provides the instructor with a macro-level view of engagement trends over time, enabling identification of participation declines requiring intervention and confirmation that engagement patterns align with the intended delivery schedule.

- **Hits Distribution:** A frequency distribution chart of learner interactions across the course timeline, enabling identification of peak engagement periods, pre-deadline activity spikes, and module-level disengagement patterns.

  Within the SLE pilot, this report revealed consistent engagement spikes in the 24–48 hours preceding summative quiz deadlines, informing the interpretation of activity log data in the empirical analysis phase.

- **Activity Report (Graphical):** A visual rendering of per-activity access and completion data, complementing the tabular Activity Report available through Moodle's core tools. Bar chart representation enables rapid cross-activity comparison of engagement levels.

- **Course Participation (Graphical):** A graphical summary of learner participation aggregated across course components, enabling visual comparison of engagement levels between individual students. When used with cohort-based filtering, this report directly supports the subgroup comparative analysis required by the study's research design.

- **Activity Completion (Matrix View):** A matrix display showing completion status for each activity and each enrolled learner, presented as a colour-coded grid. Incomplete activity chains are immediately visible without requiring row-by-row inspection, enabling the instructor to identify learners who have stalled at specific points in the conditional activity sequence.

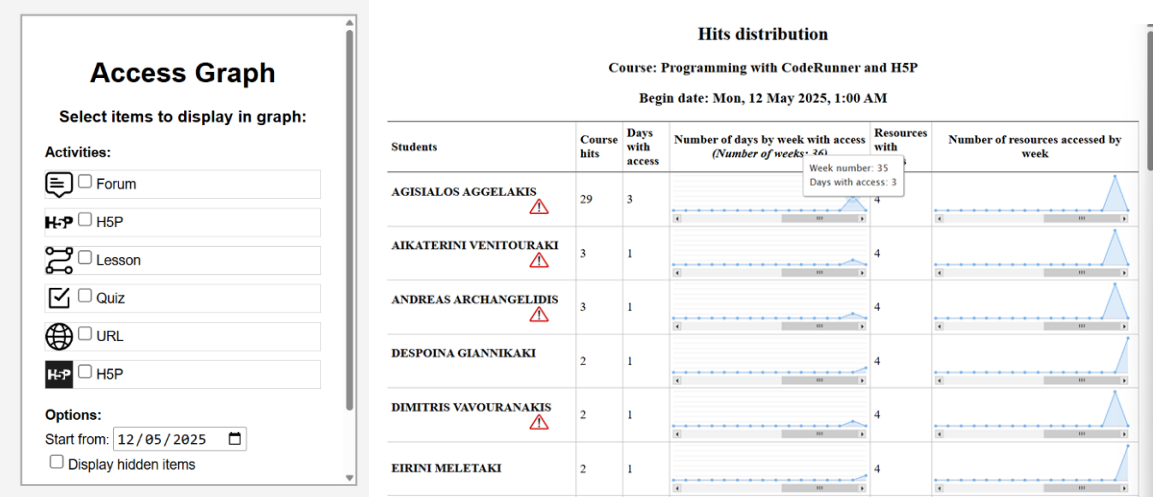


Figure 139. Analytics Graphs — Access Graph and Hits Distribution

Cohort-Based Filtering and Badges Management

A key configuration feature of the Analytics Graphs plugin within the SLE was the activation of cohort-based filtering, which enabled the instructor to generate all reports separately defined during user provisioning.

Within the Moodle Smart Learning Ecosystem, cohort-based filtering was configured through the use of system-level cohorts to enable structured subgroup analysis. Specifically, a manually created system cohort (Dakorateachers) was established during the user provisioning phase, allowing designated participants to be grouped under a common classification.

This cohort, associated with global grading visibility permissions, functioned as a filtering mechanism within the Analytics Graphs plugin, as shown in Figure 140, enabling the instructor to isolate and generate reports based on predefined user groupings.

By leveraging this configuration, the instructor was able to segment analytical outputs according to cohort membership, thereby facilitating targeted examination of engagement and performance indicators.

The availability of system-wide cohorts ensured consistency across courses and supported controlled access to aggregated data, particularly for users with extended oversight roles. Consequently, cohort-based filtering reduced the complexity of comparative analysis, allowing for efficient monitoring of subgroup dynamics and reinforcing the capacity for data-informed instructional decision-making within the learning environment.

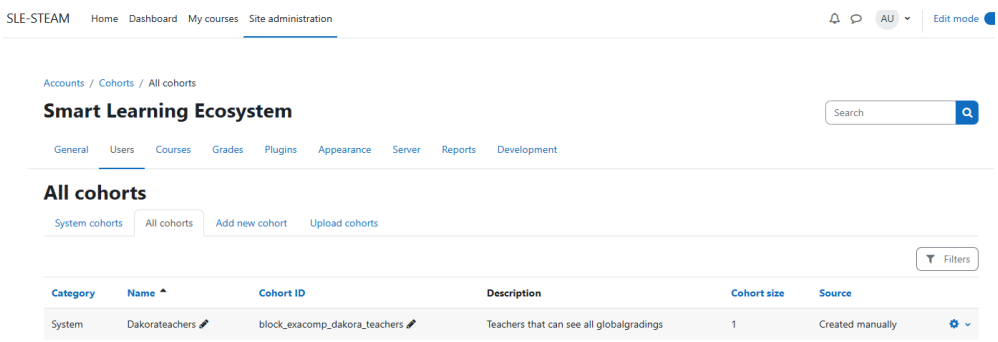


Figure 140. Analytics Graphs — Cohort Filtering Interface for Subgroup Comparison

7.3.3 Data Streams for Research Evaluation

The SLE's analytics infrastructure was designed from the outset to support not only real-time pedagogical monitoring but also the export of structured datasets for external statistical analysis.

Three complementary data streams constitute the primary empirical inputs:

- **Engagement Logs (Moodle Core + Analytics Graphs):** Activity participation frequency, content access counts, and hits distribution data, exportable in CSV format from both the core Logs report and the Analytics Graphs plugin. These provide a behavioural engagement dataset indexed by learner, activity, and date.
- **Performance Data (Moodle Gradebook):** Module and course-level grades exported from the Moodle gradebook, including individual activity scores for H5P, CodeRunner, and summative quiz components — providing a quantitative performance dataset for each enrolled learner.
- **Gamification Metrics (Level Up XP):** XP totals and level progression records exported from the Level Up XP plugin, providing a proxy measure of cumulative behavioural engagement intensity that complements the activity-level engagement data from the core logs and Analytics Graphs reports.

These three datasets were exported at the conclusion of the pilot study and imported into SPSS for statistical analysis.

Their combination enables investigation of the relationships between engagement intensity, gamification participation, and learning outcomes. The integrity of this dataset is directly dependent on the configuration decisions made during development, including cohort provisioning, gradebook aggregation rules, and XP event rule definitions — all described in Chapters 4 and 5.

Table 20. Integrated Analytics Architecture of the SLE

Analytics Layer	Tool / Report	Data Type	Primary Use in SLE
Core — Descriptive	Logs / Live Logs	Event stream	Individual interaction audit, real-time monitoring
Core — Descriptive	Activity Report	Access & completion	Per-activity engagement overview
Core — Descriptive	Course Participation	Engagement aggregates	Cohort-level comparison
Core — Descriptive	Gradebook Reports	Performance scores	Grade distribution, module averages
Core — Predictive	Students at Risk Model	ML risk scores	Early identification of at-risk learners
Visual Extension	Analytics Graphs Plugin	Charts & matrices	Trend monitoring, subgroup comparison, export

This layered architecture, shown in the above Table 20, positions the instructor not as a passive recipient of aggregated end-of-course statistics, but as an active participant in an ongoing monitoring cycle informed by continuous data generated through normal learner interaction with the course. The descriptive layer reveals what is happening; the predictive layer surfaces what may happen; and the visual extension layer makes both accessible and interpretable without requiring statistical expertise or external tooling.

7.4 Proof-of-Concept Development and Functional Validation

The Preliminary Experimental Assessment was conducted with a cohort of thirty secondary education learners enrolled in the Smart Learning Ecosystem course. The study period spanned the duration of four course modules, during which participants engaged with the full sequence of course activities as described in chapters 5 and 6. This section provides an overview of the pilot execution, including participant provisioning, study timeline, and instructor facilitation.

7.4.1 Participant Provisioning and Onboarding

Thirty student accounts were provisioned in bulk using Moodle's CSV upload functionality prior to the commencement of the pilot study, as described in Chapter 4. Account credentials were distributed to participants, and an onboarding session was conducted to familiarize learners with the Moodle interface, the course navigation structure, the CodeRunner quiz environment, and the App Inventor and mBlock external tools.

During the onboarding session, participants were guided through the process of logging into the Moodle platform, navigating the course structure, completing a practice H5P activity, and submitting a trial CodeRunner question.

App Inventor account provisioning using the Revisit Code mechanism and mLink installation for mBlock hardware connectivity were also addressed during onboarding.

7.4.2 Study Delivery and Facilitation

The pilot study was delivered across the four course modules, with instructor facilitation combining proactive monitoring — using the Live Logs view to maintain real-time situational awareness during supervised sessions — with reactive support through Moodle's messaging system when engagement deficits were identified in the Activity Completion matrix or Course Participation reports. In the below figures, Figure 141 presents, A course activity report, showing the number of views for each activity and resource, while Figure 142, shows how each user is accessing the course and its resources in the first course week and Figure 143, refers how assessments contribute to the final course grade.

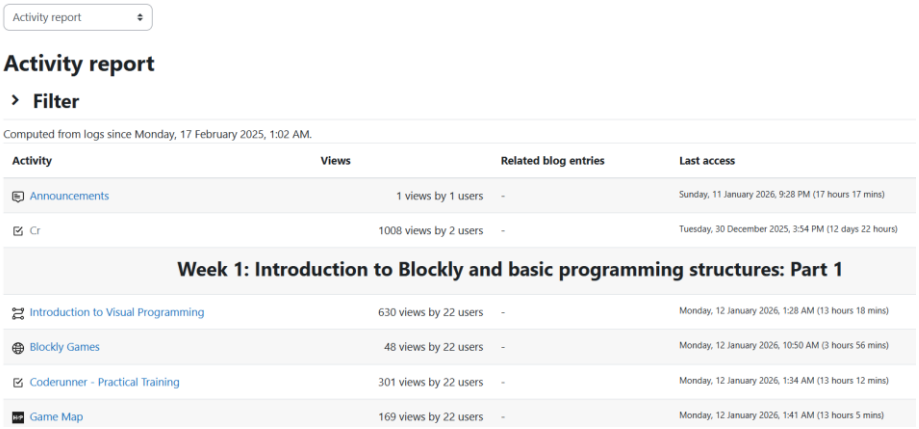


Figure 141. Activity Report

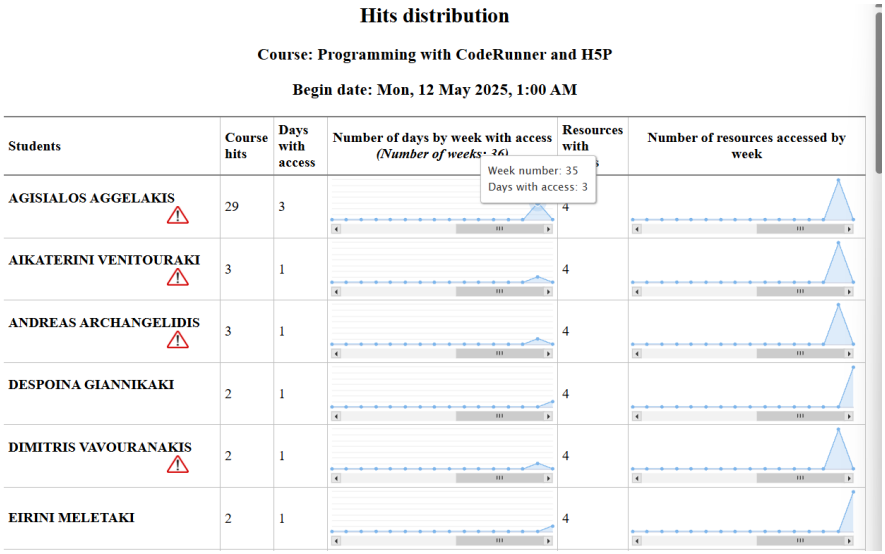


Figure 142. Hits Distribution

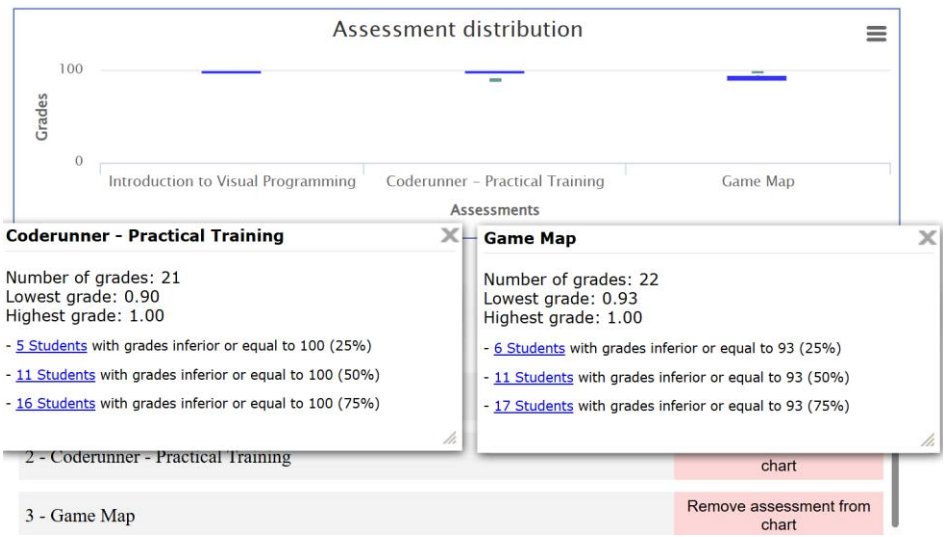


Figure 143. Assesment Distribution

The gamification framework operated continuously throughout the study period, with XP award events generated automatically in response to qualifying learner interactions. There is a ranking of students based on their members' points, as shown in Figure 133.

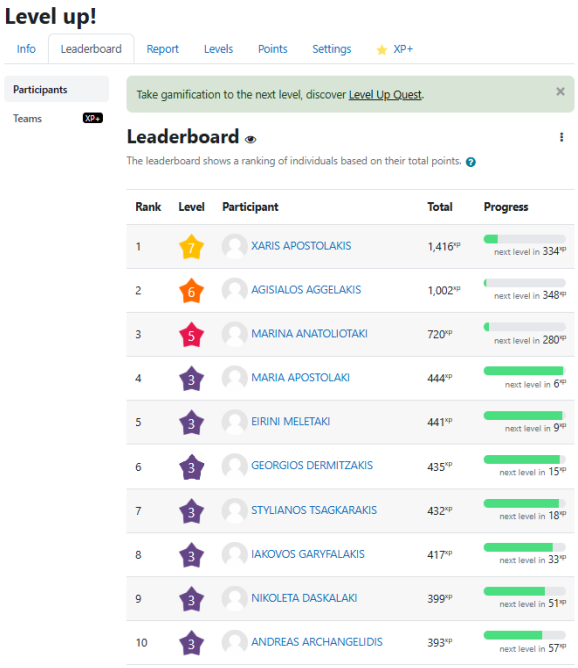


Figure 144. Level up – Leaderboard

The ten-level progression structure and automated badge issuance functioned as intended from the first day of the pilot, with the majority of actively engaged learners progressing through early levels during initial modules.

Chapter 8 - Discussion and conclusions

This final chapter synthesizes the work presented in this thesis, reflecting on the journey from the initial conception of a Smart Learning Ecosystem (SLE) to its validation as a functional and pedagogically coherent platform.
It summarizes the key achievements, discusses the findings in relation to the original research questions, explores the theoretical and practical implications of the work, acknowledges its limitations, and concludes with recommendations for future research and development.

8.1 Summary of the Thesis and Key Achievements

This thesis set out to address a significant and underexplored challenge in contemporary educational technology: the design, development, and validation of a Smart Learning Ecosystem (SLE) capable of delivering interactive, progressive, and adaptive STEAM education through the integration of gamification, Internet of Things (IoT) technologies, and automated assessment within an asynchronous learning framework.

Motivated by the imperatives of the digital industrial revolution and accelerated by the digital disruption of the COVID-19 pandemic, the research identified a clear gap between the theoretical potential of intelligent educational technologies and the practical realities of what most learners actually encounter in asynchronous digital learning environments.The thesis pursued this goal through six interconnected chapters. Chapter 1 established the intellectual context of the research, presenting the motivation for the study, defining the central research question, and formulating five supporting research sub-questions. These questions guided the investigation into how an integrated Smart Learning Ecosystem could effectively combine pedagogical design, technological infrastructure, and automated assessment to support asynchronous STEAM education.

Chapter 2 conducted a systematic literature review using the PRISMA methodology [70]. A total of 26 empirical studies from major academic databases—including Google Scholar, ERIC, Scopus, and IEEE Xplore—were analyzed to map the current state of research in Smart Learning Ecosystems, Intelligent Tutoring Systems, gamification in education, learning analytics, and IoT-enhanced learning environments.

The review identified several critical research gaps: the underrepresentation of block-based programming within adaptive STEAM ecosystems, the lack of structured pathways supporting the transition from visual to text-based programming, the limited integration of gamification at the programming interaction level, and the weak coupling between IoT-based physical computing and visual programming environments.

Building upon these findings, Chapter 3 proposed the architectural blueprint of the Smart Learning Ecosystem. The design followed a hub-and-spoke architecture in which Moodle functions as the central orchestration platform responsible for managing learning workflows, assessment processes, and learner data. External development environments—such as MIT App Inventor for mobile application development and Arduino programming through mBlock—were integrated as specialized practice environments, enabling hands-on experimentation while maintaining centralized control of assessment, analytics, and progress tracking. The architecture was guided by six core design principles: pedagogical coherence, progressive scaffolding, integration across technological boundaries, analytics-enhanced formative assessment, asynchronous-first design with embedded gamification, and open extensibility.

These principles were operationalized through a five-stage pedagogical workflow guiding learners from conceptual introduction and interactive content exploration through practical programming tasks, IoT experimentation, automated programming assessment, and ultimately to knowledge consolidation and advanced challenges in Python programming.

Chapter 4 established the technical backbone of the SLE. The system was built on a hub-and-spoke architecture, with Moodle 4.1 LTS as the central hub, deployed locally using XAMPP on a Windows 11 workstation. The platform was configured with a structured course category architecture, role-based access control, and critical security measures like HTTPS enforcement and strict password policies. The ecosystem's functionality was extended through four key plugins. H5P was integrated for creating interactive content, while CodeRunner, in conjunction with the Jobe server, enabled automated programming assessment with secure code execution and immediate feedback.

Analytics Graphs was installed to monitor learner engagement and performance, and Level Up XP implemented a gamification framework featuring experience points, a ten-level progression system, and competency-based badges.

Finally, external programming environments were prepared, including MIT App Inventor for mobile app development and mBlock for Arduino-based robotics and IoT projects, to support hands-on learning activities.

Chapter 5 operationalized the pedagogical design within the completed infrastructure. It began by structuring the Moodle course into four thematic modules, each following a consistent, progression-gated sequence based on the five-stage pedagogical model: Exploration, Conceptualization, Application, Evaluation, and Reflection. A total of ten H5P activities — including interactive videos, branching lessons, games, and quizzes — were deployed to support conceptual learning across all modules.

Chapter 6 provides a detailed presentation of the Blockly visual programming environment and the CodeRunner assessment interface, highlighting their role within the Smart Learning Ecosystem and their contribution to the learning and assessment process. It explains how the visual programming functionalities of Blockly support learners in developing computational thinking and programming skills through an interactive and user-friendly environment, while the CodeRunner interface enables automated assessment and immediate feedback. In addition, the chapter includes representative walkthroughs of three carefully designed exercises that demonstrate the practical use of these tools across progressively increasing levels of complexity. Through these examples, the chapter illustrates both the pedagogical structure of the activities and the technical implementation of the programming and assessment workflow.

Chapter 7 explores the gamification and analytics reporting features integrated into the Smart Learning Ecosystem (SLE), placing particular focus on their architecture, practical implementation, and incorporation within the Moodle-based platform. It discusses the ways in which gamification functionalities—such as experience point (XP) accumulation, level advancement, badge allocation, competency recognition, and conditional access to learning content—have been designed to strengthen learner motivation, engagement, and individualized learning pathways.

In addition, the chapter presents the data collection, assessment, and analytics infrastructure that enables the monitoring, evaluation, and interpretation of learner activity throughout the ecosystem. The various assessment approaches supported within the SLE are also examined, demonstrating how they contribute to both formative and summative evaluation processes. Particular emphasis is placed on the Analytics Graphs Plugin, which enhances the platform's native analytics capabilities by providing advanced visualization and reporting functionalities. Overall, the chapter offers a thorough presentation of the pedagogical principles and technical mechanisms underpinning the gamification and analytics reporting framework implemented within the Smart Learning Ecosystem.

This concluding chapter brings together and synthesizes the research and development work presented throughout this thesis, providing a comprehensive reflection on the progression from the initial design and conceptualization of the Smart Learning Ecosystem (SLE) to its successful implementation and validation as a functional, technically robust, and pedagogically consistent platform. This chapter revisits the primary contributions of the study, critically examines the findings in light of the original research objectives, and highlights both the theoretical significance and practical applications of the results. It also provides a transparent discussion of the study's limitations, framing them as opportunities for refinement, and concludes by outlining informed directions for future research and continued development in the field.

The key achievements of this thesis can be summarized as follows. First, the research successfully designed and implemented a fully functional Smart Learning Ecosystem that integrates interactive learning content, visual programming practice, IoT experimentation, automated assessment, gamification, and learning analytics within a coherent architecture. Second, it demonstrated the technical feasibility of a Blockly-to-Python automated assessment bridge, enabling block-based programming solutions to be translated into executable Python code and evaluated automatically within the Moodle environment.

Third, the thesis operationalized a structured five-stage pedagogical workflow that connects conceptual learning, practical experimentation, and formal assessment in a progressive learning pathway.

Fourth, the study demonstrated the practical integration of gamification mechanisms—such as experience points, levels, and achievement badges—directly within programming activities, supporting sustained engagement in asynchronous learning contexts. Fifth, the ecosystem implemented analytics-driven monitoring and adaptive feedback mechanisms capable of tracking learner interactions and triggering rule-based interventions to support learner progress. Finally, the validation process established that the proposed Smart Learning Ecosystem has progressed beyond a conceptual design to a state of operational readiness.

The platform demonstrated technical stability, pedagogical coherence, and reliable system functionality, making it suitable for deployment in authentic educational settings and for future experimental research evaluating its impact on learning outcomes.

Overall, this thesis demonstrates that the integration of visual programming environments, IoT experimentation, automated programming assessment, gamified progression, and learning analytics within a unified ecosystem can provide a scalable and pedagogically coherent approach to supporting progressive STEAM education in asynchronous digital learning environments.

8.2 Discussion of Findings in Relation to Research Questions

This section revisits each of the six research sub-questions and the overarching central research question, discussing what the design, implementation, and validation of the Smart Learning Ecosystem reveal in response to each. It is important to note at the outset that the six sub-questions are not equivalent in terms of the depth of evidence available to answer them. RQ1 through RQ5 are addressed by a robust body of empirical literature and are further supported by the functional validation findings reported in Chapter 5. RQ6, concerning the structured Blockly-to-Python learning pathway, stands in a different position: as established in Chapter 2, the systematic review found no study that directly addresses this configuration, and this thesis's contribution is therefore primarily one of problem identification, system design, and functional demonstration rather than empirical validation.

RQ1: How does the integration of block-based programming, IoT technologies, and gamification in a Smart Learning Ecosystem contribute to adaptive STEAM education?

The central research question asks how the integration of block-based programming, IoT technologies, and gamification in a Smart Learning Ecosystem can contribute to adaptive STEAM education. The literature review, particularly the contributions of Sie and Lin (2025), Fitriasih et al. (2025), Ishaq et al. (2024), and Kassenkhan et al. (2025), establishes that each of these three elements independently enhances aspects of STEAM learning. The distinctive claim of this thesis is that their integration produces effects that are synergistic rather than merely additive. The findings across all six sub-questions converge on a coherent answer: integration is itself the mechanism through which each component acquires its adaptive potential.

Block-based programming, when embedded within a structured assessment framework that includes automated translation, analytics-driven feedback, and gamified progression, becomes more than an introductory tool. It becomes a data-rich, assessable, and pedagogically rich modality capable of supporting adaptive learning pathways from novice engagement toward Python proficiency. IoT technologies, when coupled with visual programming through the hub-and-spoke model, extend the STEAM learning context from abstract algorithmic problems to tangible physical systems without imposing hardware dependencies that undermine equity of access.

Gamification, when tied directly to programming task completion and assessment outcomes rather than generic platform interactions, functions as a motivational scaffold that reinforces skill mastery rather than superficial engagement.

The collective contribution of this research is therefore not any single technological innovation but the demonstration that these three elements, when designed with coherent pedagogical intent and validated through systematic testing, constitute a synergistic and adaptive learning ecosystem. Each component enhances the effectiveness of the others, and the hub-and-spoke architecture provides the integration layer that makes this synergy possible within a scalable, open-source, asynchronous learning framework.

RQ2: : What Are the Essential Architectural Components Required for an Effective SLE Supporting Asynchronous STEAM Education?

The literature review established that existing Smart Learning Ecosystems tend to be fragmented, tool-centric, or theoretically oriented, lacking coherent integration between physical computing, visual programming, and automated assessment. Studies by Kerimbayev et al. (2025), Edula et al. (2023), Ihichr et al. (2024), and Silva et al. (2025) converge on an architectural blueprint comprising five essential layers: content management, assessment and feedback, gamification and motivation, physical computing and IoT, and learning analytics.
The architecture developed in this thesis responds to that evidence by proposing a hub-and-spoke model in which Moodle functions as the single authoritative orchestration platform, while MIT App Inventor, Arduino via mBlock, and the Blockly-CodeRunner environment operate as loosely coupled practice spokes.
The validation findings confirm that this architectural choice is both technically viable and pedagogically effective.

By centralizing all assessment, progression tracking, analytics, and gamification within Moodle while allowing learners to experiment freely in external tools, the ecosystem achieves what the literature describes as pedagogical coherence without technical rigidity. The four logical layers identified in the architectural design—the interaction layer, the orchestration layer, the assessment and feedback layer, and the data and analytics layer—each fulfilled their designated functions during validation without critical failure.

Critically, the research demonstrates that the effectiveness of an SLE for asynchronous STEAM education depends not on the sophistication of any individual component but on the quality of integration between components. Conditional activity release, mastery-based progression thresholds, and centralized gradebook aggregation collectively created a learning environment in which students could engage at their own pace without sacrificing the structured progression that is essential for developing computational thinking skills.

RQ3: How Does Gamified Progression Affect Learner Motivation and Persistence in Asynchronous STEAM Activities?

The literature consistently reports that gamification contributes positively to learner motivation and persistence in asynchronous learning contexts (Kassenkhan et al., 2025; Ishaq et al., 2024; Son, 2025; Navas Bonilla et al., 2025).

However, the review identified that gamification in existing systems is predominantly applied at the platform level through generic badge and points systems, rather than being embedded within programming activities themselves. This thesis directly addresses that gap. The gamification engine implemented in the SLE goes beyond surface-level reward mechanisms. Experience points are tied directly to CodeRunner test case outcomes, H5P formative check completion, and forum contributions, meaning that every point reflects a demonstrable learning behavior rather than merely a system interaction.

The ten-level progression system, with snowballing XP thresholds mapped to pedagogically meaningful milestones, is designed to sustain motivation across the full learning trajectory rather than generating a brief spike of early engagement followed by diminishing returns. Koukaras et al. (2026) further reinforce the importance of equity-aware gamification design—an aspect addressed in the SLE through the accessibility of early achievement milestones.

The pilot validation provided initial evidence that learners responded positively to visible progress indicators, including the XP block, progress bars, and badge notifications, describing these as contributing to their sense of progress and engagement. While the sample size precludes statistical inference, these observations are consistent with the theoretical predictions of Self-Determination Theory, which posits that visible competence indicators and structured goal-setting contribute to intrinsic motivation.

The achievement badges, designed to recognize specific computational competencies such as loop mastery, sensor integration, and peer mentoring, suggest that gamification can function as a micro-credentialing framework acknowledging discrete skill development in ways that conventional grades do not.

The finding that gamification thresholds required recalibration following the pilot—with early achievements made more attainable to build engagement—illustrates a broader design principle: that effective gamification in educational contexts requires empirical tuning rather than theoretical prescription alone. The iterative refinement process demonstrated that the boundary between motivating challenge and frustrating difficulty must be calibrated to the specific learner population and task type.

RQ4: Can Integrated Analytics and Automated Assessment Provide Timely Feedback That Improves Learning Performance?

The literature identifies integrated analytics and automated assessment as among the most impactful features of adaptive learning systems, with LLM-based and AI-driven assessment frameworks demonstrating strong potential (Lee et al., 2026; Ihichr et al., 2024; Rodríguez-Ortiz et al., 2025). However, the review also revealed that existing assessment models are overwhelmingly text-code-centric, with limited attention to the analytics derivable from block-based programming behaviors.

The SLE addresses this gap through two mechanisms. First, the multi-layered assessment toolkit—combining H5P for formative conceptual checks, Moodle Quiz for knowledge consolidation, CodeRunner for automated programming evaluation, and Moodle Assignments for project-based evidence collection—ensures that assessment is continuous, automated, and distributed across conceptual, applied, and integrative dimensions.

All assessment data feeds into a unified Moodle gradebook, enabling stage-by-stage performance analysis and correlation between conceptual understanding and programming success. This aligns with the finding of Munar-Garau et al. (2025) that embedded, workflow-integrated feedback produces stronger learning outcomes than separately administered assessment.

Second, the custom Blockly workspace telemetry system represents a novel contribution to the field. By capturing block addition, deletion, movement, and configuration events at millisecond precision, the ecosystem generates a rich behavioral dataset that extends beyond final solution correctness to encompass problem-solving strategies, debugging patterns, and block-to-text transition readiness.

This enables the kind of fine-grained learning analytics that the literature identifies as missing from current block-based programming research—and provides the raw data that future AI-driven adaptive models could exploit. The validation confirmed that the analytics dashboards accurately reflected real-time learner activity and that the rule-based adaptive intervention system correctly identified patterns of repeated failure, prolonged inactivity, and progression stagnation, triggering remedial content release and instructor notifications accordingly.
These findings validate the operational functionality of the analytics layer and provide a foundation for future research into predictive modeling of block-based programming behaviors.

RQ5: How Effectively Does IoT-Based Physical Computing Enhance Learners' Understanding of Computational Concepts?

Research by Fitriasih et al. (2025) and Sie and Lin (2025) demonstrates that IoT-enhanced learning environments effectively contextualize computational thinking and support real-world problem solving. Van Mechelen et al. (2022), Weng et al. (2024), and Ng et al. (2022) further confirm that physical-digital integration is a consistent predictor of deeper computational concept acquisition, particularly for learners who have previously struggled with purely abstract programming instruction. However, the literature review revealed that connections between IoT-based physical computing and block-based programming environments are weakly explored, creating an accessibility barrier for novice learners who must simultaneously manage hardware complexity and programming syntax.

The SLE addresses this gap by positioning mBlock-based Arduino practice alongside MIT App Inventor as complementary external experimentation environments that are pedagogically coupled with, but technically decoupled from, the formal Moodle assessment layer. This dual-environment approach is pedagogically significant, as it allows learners to engage with IoT systems through both physical computing (via Arduino and sensors) and mobile application development (via App Inventor), thereby broadening the spectrum of interaction with computational concepts. Learners develop intuitive understanding of IoT processes through direct observation—for instance, witnessing an LED respond to a light sensor reading—while also designing user-facing mobile interfaces that can control or monitor such systems.

This multimodal engagement strengthens the connection between abstract logic and real-world application contexts.

Subsequently, learners formalize their understanding through Blockly and Python within the controlled assessment environment. This sequence embodies the concrete-to-abstract progression advocated by cognitive load theory and constructionism, while also extending it to include user-centered design principles through mobile app development.
The hub-and-spoke architecture also resolves a practical challenge that limits IoT education at scale: hardware dependency.
By separating hands-on Arduino experimentation and mobile app prototyping from formal assessment, the ecosystem ensures that learners without home access to hardware can still complete the full assessment pathway within Moodle, using CodeRunner to solve IoT-scenario problems in simulation. This design principle—prioritizing equitable assessment while preserving authentic hands-on and application-driven experience—represents a meaningful contribution to the practical implementation of IoT education in secondary contexts.

RQ6: How Can a Structured Blockly-to-Python Learning Pathway Be Designed Within a Smart Learning Ecosystem?

As established in Chapter 2, RQ6 occupies a distinct position within this study.
The systematic literature review found no study that directly addresses a Moodle/CodeRunner/Blockly configuration for structured visual-to-textual programming transitions, and the evidence base is therefore theoretical and analogical rather than directly empirical.

The contribution of this thesis with respect to RQ6 is accordingly one of identification and first instantiation rather than resolution: it makes the problem visible and precisely defined, proposes a theoretically grounded design response, and delivers a functional technical implementation—while acknowledging that the necessary empirical validation remains ahead. The Blockly-to-Python assessment bridge implemented through the custom CodeRunner question template constitutes the most technically novel component of the ecosystem.

By developing a JavaScript-based translation engine that converts Blockly XML into executable Python and evaluates it against predefined test cases, the system provides a first operational demonstration of automated visual-to-textual programming assessment within a Moodle environment.

The five-stage workflow draws on the design principles supported by the literature—dual representation (Ishaq et al., 2024), scaffolded fading (Dutta et al., 2024), mastery-gated progression (Son, 2025), and explicit pathway signposting (Navas Bonilla et al., 2025)—to sequence the transition from block-based to Python programming in a theoretically coherent way.

The functional validation of Stage 3 confirmed that correct block-based solutions were consistently translated into executable Python without errors, and that logical mistakes triggered targeted feedback. The translation maintained syntax consistency in tested cases, including edge cases involving incorrect loop structures, variable misassignments, and missing conditions. These results are encouraging as a proof of technical concept, but they do not constitute evidence that the pathway achieves its intended pedagogical purpose of reducing barriers to text-based programming. That question—whether learners who engage with this pathway develop stronger Python proficiency than those using alternative approaches—remains open and constitutes the primary empirical priority for future work on this research question. In sum, this thesis takes a meaningful first step toward addressing RQ6 by making the problem technically tractable and providing a functional platform for future investigation. Significant empirical and design work remains to be undertaken before confident conclusions about the effectiveness of the Blockly-to-Python pathway can be drawn.

8.3 Theoretical and Practical Implications

8.3.1 Theoretical Implications

The findings of this thesis contribute to the fields of educational technology, computational thinking education, and learning design by advancing the theoretical understanding of Smart Learning Ecosystems (SLEs) for STEAM education.

First, the research extends the conceptualization of Smart Learning Ecosystems by demonstrating how multiple educational technologies—such as block-based programming, IoT learning environments, learning analytics, and gamification—can be integrated into a coherent architectural and pedagogical model.

Rather than functioning solely as platforms for personalized content delivery, SLEs can support structured cognitive progression, guiding learners from concrete exploration to abstract computational thinking through sequenced learning activities.

Second, the study challenges the dominant AI-centric perspective in recent SLE literature. While many approaches emphasize machine learning and AI-driven personalization, this research demonstrates that meaningful adaptive behavior can also emerge from rule-based mechanisms grounded in rich behavioral data, such as Blockly workspace telemetry and programming execution logs.
This finding suggests that adaptive educational systems do not necessarily require complex AI models when well-designed data collection and conditional logic can provide pedagogically relevant adaptive responses.
Third, the thesis contributes to constructionist learning theory by showing how digital environments can capture not only the artifacts produced by learners but also the processes through which these artifacts are constructed. By recording detailed interaction data from Blockly environments and code execution systems, the proposed ecosystem creates analyzable traces of constructionist learning, enabling new forms of formative feedback and adaptive intervention.

Fourth, the proposed Blockly-to-Python transition framework addresses a gap in programming education research by providing a concrete architectural model for supporting progression from visual to textual programming. The translation mechanism between Blockly constructs and Python code offers a practical implementation of scaffolding aligned with Vygotsky's Zone of Proximal Development, allowing learners to operate at the boundary between visual programming competence and textual programming proficiency. Finally, the research contributes to learning design by operationalizing principles of cognitive load theory in STEAM education.

The proposed multi-stage learning workflow separates conceptual, programming, and hardware complexities, thereby reducing extraneous cognitive load. In addition, gamification is reconceptualized as an integrated component of assessment and progression, where rewards are directly linked to demonstrated competencies rather than functioning solely as external motivational elements.

Overall, the thesis contributes a theoretically grounded model for designing integrated Smart Learning Ecosystems that support progressive development of computational thinking and programming skills.

8.3.2 Practical Implications

The findings of this research have practical implications for educational technologists, curriculum designers, educators, and institutions seeking to implement technology-enhanced STEAM education.

First, the proposed ecosystem offers a replicable architectural model for integrating specialized learning tools within a central Learning Management System. The hub-and-spoke architecture allows external environments—such as block-based programming platforms and IoT experimentation tools—to function as practice spaces while assessment, tracking, and analytics remain centralized in the LMS. This design simplifies system integration, reduces data fragmentation, and enables scalable automated assessment. Because the ecosystem relies on open-source technologies, it can be adopted and adapted by institutions without significant financial investment.

Second, the study provides a practical framework for progressive programming education. The five-stage learning workflow offers a structured model for sequencing computational learning activities, gradually moving learners from conceptual understanding to visual programming, IoT experimentation, and finally textual programming assessment. This staged approach supports flexible asynchronous or blended learning environments while maintaining opportunities for hands-on STEAM activities.

Third, the ecosystem demonstrates how automated assessment and learning analytics can support scalable programming education. Tools such as automated code evaluation and centralized analytics dashboards enable instructors to monitor learner progress, identify common error patterns, and detect students at risk of disengagement. This reduces instructor workload while supporting data-informed instructional decisions. Fourth, the gamification framework provides a practical model for integrating motivational mechanisms into programming education.
The use of progression levels, competency-based badges, and experience points aligned with activity complexity embeds motivation within the learning process without compromising assessment integrity.

Finally, the development and pilot testing of the ecosystem highlight the importance of iterative and participatory design in educational technology development.
Usability improvements and calibration of assessment and gamification mechanisms emerged through observation of learners interacting with the system, demonstrating the value of pilot implementations for aligning system architecture with real user experience. Overall, the ecosystem developed in this thesis provides a scalable and adaptable blueprint for institutions seeking to implement integrated, data-informed STEAM learning environments.

8.4 Limitations of the Work

A rigorous assessment of the research requires honest acknowledgment of its limitations. These limitations relate to sample size and representativeness, scope of validation, technological constraints, and the generalizability of findings.

8.4.1 Sample Size and Representativeness

The pilot study involved only thirty secondary education learners aged 15 to 16, which is insufficient for drawing statistically reliable conclusions about the ecosystem's effectiveness across the full target population. While the sample was deliberately designed to include learners with diverse levels of prior programming experience—ranging from no background through Scratch experience to text-based programming familiarity—it cannot be considered representative of the broader diversity of secondary learners in terms of cultural background, socioeconomic context, learning disability, or neurological diversity.

The broader validation phase in Chapter 5, while systematic in its technical testing, similarly did not involve a large-scale learner cohort. Consequently, all learner experience findings must be treated as indicative rather than conclusive.

8.4.2 Absence of Longitudinal Learning Outcome Measurement

The validation framework was explicitly designed to confirm functional completeness, technical correctness, pedagogical workflow integrity, data flow accuracy, and usability, rather than to measure learning outcomes.

As a consequence, the thesis does not provide empirical evidence of statistically significant improvements in computational thinking, programming proficiency, or STEAM knowledge as a result of using the ecosystem.

The absence of pre- and post-intervention assessments of learning outcomes, retention, and skill transfer represents a significant limitation of the current work.

Whether the ecosystem is more effective than alternative approaches to STEAM education, or more effective than non-gamified or non-IoT integrated alternatives, remains an open empirical question. This limitation is particularly acute in relation to RQ6: the functional validation confirms that the Blockly-to-Python translation works technically, but it cannot confirm that learners who use the pathway develop stronger Python proficiency than they would through alternative instructional approaches.

8.4.3 Rule-Based Rather Than AI-Driven Adaptivity

The adaptive mechanisms implemented in the SLE are rule-based, relying on predefined conditional logic within Moodle rather than dynamic machine learning algorithms. While this approach proved effective within the tested parameters, it lacks the predictive capacity and personalization depth that AI-driven adaptive systems could theoretically offer.

The rule-based system can identify learners who have failed repeatedly or who have been inactive for a defined period, but it cannot dynamically model individual learner knowledge states, predict future performance trajectories with high confidence, or automatically adjust task difficulty in real time based on learner-specific behavioral patterns. The analytics models enabled through Moodle's PHP predictions processor provide some predictive capability, but these are generic engagement models rather than domain-specific computational thinking models tailored to the SLE's curriculum.

8.4.4 Incomplete Data Capture from External Tools

The hub-and-spoke architecture deliberately avoids tight technical integration between Moodle and external practice environments, accepting some loss of behavioral data in exchange for architectural simplicity and robustness. As a result, learner interactions with MIT App Inventor and mBlock-based Arduino environments are not automatically captured in the analytics pipeline.

Instructors have no visibility into how much time learners spend in App Inventor, what block structures they attempt during practice, how many debugging iterations they undergo before transitioning to formal assessment, or what hardware-related challenges they encounter with physical computing tasks.This data gap limits the completeness of the behavioral analytics available to instructors and constrains the system's ability to identify and support learners who are struggling in the practice environments.

8.4.5 Local Development Environment and Scalability Testing

The development and validation of the SLE were conducted on a local XAMPP-based environment running on a single Windows 11 workstation, rather than a production-grade cloud server. While this configuration was appropriate for the development and pilot phases, it limits the conclusions that can be drawn about system scalability, performance under high concurrent user loads, and long-term stability in production deployment. The performance characteristics observed during pilot testing with thirty learners cannot be assumed to generalize to classroom deployments involving more simultaneous users, and cloud deployment would require additional configuration for HTTPS enforcement, database replication, and backup management.

8.4.6 Scope and Empirical Status of the Blockly-to-Python Translation Engine

The Blockly-to-Python translation engine, while technically validated for correctness and syntax consistency within the tested range of constructs, represents a first implementation rather than a mature, comprehensively validated system.

It supports a defined subset of Blockly constructs mapped to their Python equivalents; complex or unconventional configurations—particularly those combining multiple control structures in non-standard ways or using custom block definitions—may produce unexpected translation outputs.

More fundamentally, as noted throughout this thesis, no study in the reviewed literature has empirically validated a Blockly-to-Python transition pathway of this kind. The translation engine therefore operates in largely uncharted educational territory. Its technical correctness has been confirmed, but whether it produces the intended pedagogical outcomes—specifically, a reduction in the barriers to text-based programming and a meaningful improvement in Python learning—remains to be demonstrated through rigorous empirical research.

8.5 Conclusions and Future Work

8.5.1 Conclusions

This thesis has designed, implemented, and validated a Smart Learning Ecosystem that integrates gamification, IoT technologies, visual-to-textual programming pathways, and automated assessment within an asynchronous STEAM learning framework.

The work was motivated by the convergence of two imperatives: the demands of a Industry 4.0 economy for adaptable, computationally literate, and interdisciplinary thinkers, and the digital learning imperative revealed by the COVID-19 pandemic for high-quality, scalable, and engaging asynchronous education. The central contribution of the thesis is the demonstration that block-based programming need not be treated as a transitional afterthought in the progression toward professional programming competence. When embedded within a coherent ecosystem that provides automated assessment, behavioral analytics, IoT contextualization, and gamified progression, block-based programming becomes a foundational pedagogical environment capable of supporting adaptive, engaging, and transferable STEAM learning. The Blockly-to-Python bridge, in particular, represents a technically functional first response to one of the most significant gaps identified in the literature—a gap that this thesis identifies, frames, and provides a starting point for addressing, while acknowledging that its pedagogical resolution will require the substantial empirical research programme outlined in Section 7.7.2.
The hub-and-spoke architecture centered on Moodle provides a replicable, open-source, and institutionally accessible model for implementing complex multi-tool STEAM learning environments without requiring proprietary platforms, advanced technical infrastructure, or tight system coupling.

The five-stage pedagogical workflow offers a detailed and theoretically grounded model of progressive STEAM education that can be adapted across secondary education contexts.
The systematic functional validation confirms that the ecosystem operates reliably, maintains architectural consistency with the hub-and-spoke model, supports a progressive STEAM learning pathway, delivers dependable automated assessment, and generates meaningful analytics and adaptive instructional responses.

The pilot testing and iterative refinement process demonstrates that the system can be made usable by secondary learners with diverse prior experience levels, and that the gap between theoretical design and practical usability can be effectively narrowed through structured observation and evidence-based refinement. In sum, this thesis establishes that Smart Learning Ecosystems for STEAM education can be designed, implemented, and validated without reliance on advanced artificial intelligence, proprietary platforms, or extensive institutional resources, provided that the integration of simpler technologies is guided by coherent pedagogical principles, grounded in the empirical literature, and subjected to systematic validation and iterative refinement.

The ecosystem developed here is operationally ready for broader classroom deployment and provides a stable, functional platform for the rigorous experimental investigation of its pedagogical impact—investigation that the current work has made possible but has not yet conducted.

8.5.2 Future Work

The limitations identified in Section 7.6 suggest several priority directions for future research that would extend, validate, and deepen the contributions of this thesis. These directions span three broad themes: empirical validation of the ecosystem's pedagogical effectiveness (Directions 1 and 2), technical enhancement of the system's adaptive and analytical capabilities (Directions 3, 4, and 5), and design expansion to address accessibility, collaboration, and broader applicability (Directions 6, 7, 8, and 9).

The most urgent priority, and the one most directly demanded by the findings of this thesis, is an empirical study of the Blockly-to-Python pathway.

As established throughout this chapter and in Chapter 2, the literature currently provides no empirical evidence for the effectiveness of a structured visual-to-textual programming transition of the kind implemented here. A controlled study comparing learners who use the five-stage SLE pathway against a matched group using alternative approaches—such as direct Python instruction or unstructured block-to-text transition—would provide the foundational evidence needed to evaluate RQ6 with confidence.

Pre- and post-intervention measures of Python proficiency, programming self-efficacy, and computational thinking, combined with behavioral data from the Blockly workspace telemetry, would yield a dataset capable of addressing not only whether the pathway works but how and for whom it works. This is the single most important direction for future research arising from this thesis.

The second priority is a full-scale experimental study measuring learning outcomes across a statistically adequate and diverse participant sample for the ecosystem as a whole. A randomized or quasi-experimental design comparing learners using the SLE against a matched control group receiving conventional STEAM instruction would provide the empirical evidence base necessary to make confident claims about the ecosystem's overall pedagogical effectiveness.

Pre- and post-intervention measures of computational thinking, programming proficiency, and STEAM concept knowledge, combined with follow-up retention measures, would address the absence of longitudinal outcome data that is the most significant limitation of the current work.

A third direction is the integration of AI-driven adaptive mechanisms to supplement the current rule-based system. The rich behavioral dataset generated by Blockly workspace telemetry, CodeRunner execution logs, and H5P interaction data provides an ideal foundation for training machine learning models capable of predicting individual learner performance trajectories, diagnosing specific misconceptions, and dynamically adjusting task difficulty and scaffolding in real time. Future work could explore the application of knowledge tracing models—such as Bayesian Knowledge Tracing or Deep Knowledge Tracing—to the block-based programming domain, using the SLE's behavioral data as training input.

A fourth direction involves extending the Blockly-to-Python translation engine to support a wider range of block configurations, including custom block types, nested function definitions, and object-oriented programming constructs.
This extension would support a more comprehensive progression pathway from basic sequential programming through procedural and object-oriented paradigms, and would enable the SLE to serve learners across a wider range of educational levels.

A fifth direction is the development of automated data capture from external practice environments. Lightweight browser extensions or embedded logging modules for MIT App Inventor and mBlock could enable passive capture of learner interaction data during practice phases, closing the data gap identified in Section 6.4.4 without requiring tight technical integration with Moodle. This data would significantly enrich the analytics available to instructors and the adaptive mechanisms available to the system.

A sixth direction is a comprehensive accessibility audit and redesign of the ecosystem for learners with visual, motor, or cognitive disabilities. The current implementation was designed and validated with a general secondary education population, and the usability findings from the pilot do not address accessibility needs specific to learners with disabilities. Future work should engage accessibility specialists and learners with diverse needs in co-designing adaptations to the interface, gamification mechanics, and assessment modalities.
A seventh direction involves the investigation of peer learning and collaborative dimensions within the SLE. The current ecosystem is primarily designed for individual asynchronous learning, with peer interaction limited to forum activities.

Future iterations could incorporate collaborative Blockly workspaces, peer assessment mechanisms, or group IoT project challenges that leverage the social dimensions of learning identified in Vygotsky's sociocultural theory. The gamification system's mentor status and peer moderation privileges, introduced at Level 9, offer a starting point for this direction.
An eighth direction concerns the development of genuinely adaptive learning paths within the ecosystem.

The current implementation simulates adaptivity through Moodle's Activity Completion and Restrict Access mechanisms, which, while effective for progression gating, require substantial upfront design effort and offer no runtime intelligence. As the Moodle community itself acknowledges, true adaptive learning is not native to the platform: when a student fails repeatedly, the system returns them to the same module without modification, content adjustment, or diagnostic differentiation. During the development of this thesis, three Moodle plugins were investigated as candidates for extending adaptive capability beyond these limitations.

The Adaptive Quiz plugin implements Computer-Adaptive Testing (CAT) based on Item Response Theory, dynamically selecting questions according to the student's estimated ability level—presenting harder items following correct responses and easier items following errors—until a statistically sufficient estimate of ability is achieved.

The ALiSe CAT Quiz similarly applies a full IRT-based CAT model, restricting item presentation to those matching the learner's current estimated proficiency. Both tools address the "same questions for everyone" problem at the assessment level but do not extend adaptivity to content sequencing or learning pathway modification.

The Exabis Competencies plugin was the most promising candidate explored, offering competency-based assessment grids, dynamic learning pathways, and dashboards designed to give students visibility over their own progress and enable instructors to provide targeted support to struggling learners. Experimental integration of Exabis within the SLE revealed, however, that the plugin could not be combined with Moodle's native competency framework in a way that enabled automated pathway adjustment; the tool supports human-mediated intervention rather than algorithmic adaptivity, and its competency structures operate in parallel to, rather than in integration with, the Moodle competency system.

As a consequence, none of the three plugins could be incorporated into the current ecosystem without sacrificing automation potential or architectural coherence.

Future work should investigate whether emerging developments in these plugins, or alternative adaptive learning frameworks, can be integrated with Moodle's competency infrastructure to enable genuine runtime adaptation of content sequencing based on individual learner performance data—a capability that would substantially enhance the ecosystem's responsiveness to diverse learner needs.

Finally as a ninth direction, the research raises an important question about the transferability of the hub-and-spoke model to other educational contexts beyond STEAM. The principles underlying the architecture—centralizing assessment and analytics within an LMS while allowing specialist tools to operate as practice environments—are potentially applicable to language learning, creative arts education, vocational training, and professional development. Comparative studies across these contexts would test the generalizability of the model and contribute to a broader theory of integration in educational technology.

Together, these directions constitute a substantial research agenda that builds on the foundations established in this thesis. Each direction is motivated by a genuine gap in the current work and has the potential to generate knowledge of both theoretical and practical significance.

The Smart Learning Ecosystem developed here is not an endpoint but a point of departure: a validated, operational platform on which the next generation of adaptive STEAM education research—and in particular the rigorous empirical investigation of the Blockly-to-Python learning pathway—can be built.

References

[1] Selitskiy, S., & Inoue, C. , "Education Paradigm Shift To Maintain Human Competitive Advantage Over AI," 2024.

[2] Zou, Y., Kuek, F., Feng, W., & Cheng, X., "Digital learning in the 21st century: Trends, challenges, and innovations in technology integration. Frontiers in Education, 10(10).," 2025.

[3] Benita, F., Virupaksha, D., Wilhelm, E., & Tunçer, B., "A smart learning ecosystem design for delivering Data-driven Thinking in STEM education," 2021,.

[4] Helbawy, M. E., & bogatin, eric. , "Advancing Engagement and Equity in Asynchronous Online Education.," 2025.

[5] Ndebele, M., & Mazhindu, A. N. , "Design and Implementation of an Android GIS Data Collection Application for Students with Non-Programming Experience Using the MIT App Inventor: A Case Study on a Tick Data Collection APP," 2023.

[6] Moraes, E. B., Kipper, L. M., Hackenhaar Kellermann, A. C., Austria, L., Leivas, P., Moraes, J. A. R., & Witczak, M., "Integration of Industry 4.0 technologies with Education 4.0: advantages for improvements in learning. Interactive Technology and Smart Education.," 2022.

[7] Bianca Ifeoma Chigbu, Viwe Ngwevu, & Jojo, A., "The effectiveness of innovative pedagogy in the industry 4.0: Educational ecosystem perspective.," 2023.

[8] Mahmoud Elkhodr, & Ergun Gide., "AI leads, humans lead, or collaborate? Empirical findings and the SAGE roadmap for embedding GenAI in systems analysis and design education," 2026.

[9] Mishra, L., Gupta, T., & Shree, A. , "Online Teaching-Learning in Higher Education during Lockdown Period of COVID-19 Pandemic," 2020.

[10] Raj, A., Nkutia, J. S., Jesús, M., Saman Ange-Michel Gougou, & Thurab-Nkhosi, D., "Smart Learning Ecosystems in Transition: A Systematic Review on the Adoption of ICT and IoT as Emerging Technologies in Education," 2025.

[11] Lansonia, A., Austin, M., & Beldiq, E. A., "Study of Student Satisfaction in Using the Moodle E-Learning System: PLS-SEM Approach," 2024.

[12] Jantakoon, T., Jantakun, T., & Jantakun, K., "A Scientometrics Analysis and Systematic Review of STEAM Education with Gamification.," 2024.

[13] Rossetti, L., Francisco, R., Tavares, R., & Luis, J. , "Intelligent environments and assistive technologies for assisting visually impaired people: a systematic literature review," 2024.

[14] Suharto, A. , "Design and Build IoT Smart Home System with Blocks Based App Inventor Programming.," 2025.

[15] Elatawy, S. , "IoT-based Smart Lab System in Schools using Arduino and Bluetooth based Android Smartphone," 2020.

[16] osé-Antonio Marín-Marín, Pedro Antonio García-Tudela, & Duo-Terrón, P., "Computational thinking and programming with Arduino in education: A systematic review for secondary education," 2024.

[17] Zadorozhnii, V. M. , "Using Arduino to develop research competencies of students in school physics education," 2024.

[18] Asif Ali Laghari, Li, H., Khan, A. A., Yin Shoulin, Karim, S., & Muhammad, "Internet of Things (IoT) applications security trends and challenges.," 2024.

[19] Saleem, A., Shah, S., Iftikhar, H., Justyna Zywiołek, & Olayan Albalawi, "A Comprehensive Systematic Survey of IoT Protocols: Implications for Data Quality and Performance.," 2024.

[20] R. Lobb and J. Harlow, "Coderunner: A tool for assessing computer programming skills," 2016.

[21] Goodfellow, M., Abel, A., Konstantinos Liaskos, & Levine, J. , "Automated Marking in Undergraduate Programming Classes," 2023.

[22] Ping, K. Y. , "ENHANCING STUDENT ENGAGEMENT AND LEARNING OUTCOMES IN HIGHER EDUCATION USING H5P INTERACTIVE LEARNING TOOLS: A SYSTEMATIC LITERATURE REVIEW," 2025.

[23] Khan, W., Rehman, M., Zangoti, H., Afzal, M., Armi, N., & Salah, K. , "Industrial Internet of Things Recent Advances, Enabling Technologies and Open Challenges.," 2020.

[24] Amanuel, Y., & Krugel, J. , "From Play to Proficiency: A Systematic Literature Review on Learning Modalities and Transition Strategies in CS and AI Education," 2025.

[25] Page, M. J., McKenzie, J. E., Bossuyt, P. M., Boutron, I., Hoffmann, T. C., Mulrow, C. D., Shamseer, L., Tetzlaff, J. M., Akl, E. A., Brennan, S. E., Chou, R., Glanville, J., Grimshaw, J. M., Hróbjartsson, A., Lalu, M. M., Li, T., Loder, E. W., Mayo-Wilso, "The PRISMA 2020 statement: An updated guideline for reporting systematic reviews.," 2021.

[26] Kolaski, K., Lynne Romeiser Logan, & John P.A. Ioannidis., "Guidance to Best Tools and Practices for Systematic Reviews.," 2023.

[27] Sie, Y. J., & Lin, K. Y., "Effects of Psychological Capital and Cognition on STEM Learning in IoT Smart Energy-Saving Projects. Journal of Baltic Science Education, 24(2), 340-359," 2025.

[28] Fitriasih, F., Handayani, T., & Isfaeni, H., "Smart biotech learning media: A STEAM-IoT based to improve students' problem-solving abilities and collaboration skills. JPBI (Jurnal Pendidikan Biologi Indonesia), 11(2), 450–459," 2025.

[29] Vinutha, D. C., Kavyashree, S., Vijay, C. P., & Raju, G. T., "Innovative Practices in Education Systems Using Artificial Intelligence for Advanced Society. The New Advanced Society, 351–372," 2022.

[30] Lee, Y. O., Bang, B., Lee, J., & Oh, S., "Personalized Auto-Grading and Feedback System for Constructive Geometry Tasks Using Large Language Models on an Online Math Platform. IEEE Access, 1–1," 2026.

[31] Patil, M., Pham, M., & Jia, B., "he role of smart education platforms in enhancing community engagement: a systematic," 2025.

[32] Kassenkhan, A. M., Moldagulova, A. N., & Serbin, V. V., "Gamification and Artificial Intelligence in Education: A Review of Innovative Approaches to Fostering Critical Thinking. IEEE Access, 13, 98699–98728," 2025.

[33] Khritish Swargiary, "Emerging Technologies in Education. ResearchGate," 2024.

[34] Kerimbayev, N., Adamova, K., Shadiev, R., & Altinay, Z., "Intelligent educational technologies in individual learning: a systematic literature review. Smart Learning Environments, 12(1)," 2025.

[35] Dutta, S., Ranjan, S., Mishra, S., Sharma, V., Pradeep Hewage, & Iwendi, C., "Enhancing Educational Adaptability: A Review and Analysis of AI-Driven Adaptive Learning Platforms. Enhancing Educational Adaptability: A Review and Analysis of AI-Driven Adaptive Learning Platforms," 2024.

[36] Ishaq, K., Alvi, A., Muhammad, Fadhilah Rosdi, Choudhry, A. N., Anjum, A., & Khan, F. A., "Level up your coding: a systematic review of personalized, cognitive, and gamified learning in programming education. PeerJ Computer Science, 10, e2310–e2310," 2024.

[37] Edula, R., Shengcheng, L., Sasaki, H., Viladiro, R., & Roy, D., Personalized learning systems in smart education: A review of the literature & pilot feasibility discussion. In AIP Conference Proceedings (Vol. 2909, No. 1, p. 060004). AIP Publishing LLC, 2023.

[38] Gurusamy, P., "Smart Classroom Technology in Artificial Intelligence: A Review Paper. Eduac Multidisciplinary Research Journal, 1(01), 40-46. Retrieved from," 2025.

[39] Son, N. D., "Personalized Learning in STEAM. Integrating Personalized Learning Methods into STEAM Education, 1–30," 2025.

[40] Ruberg, P., Ellervee, P., Heinsar, R., Selg, H., & Eek, A., "From Structured Labs to Agile Teams: A Curriculum-Based Approach to Active Learning in First-Year Computer Engineering. 2025 IEEE Frontiers in Education Conference (FIE), 1–9," 2025.

[41] Nevena Kostadinova Dicheva, Ikram Ur Rehman, Anwar, A., Nasralla, M. M., Laden Husamaldin, & Sama Aleshaiker., "Digital Transformation in Nursing Education: A Systematic Review on Computer-Aided Nursing Education Pedagogies, Recent Advancements and Outlook on the Post-COVID-19 Era. IEEE Access, 11, 135659–135695," 2023.

[42] Al Battashi, M. A., Adnan, M. A., Jamil, A. I. B., & Al-Battashi, M. A., "Mapping Research Trends in AI-Driven Personalized Learning Pathways: A Scoping Review. Generators, Bots, and Tutors: Creative Approaches to Human-AI Synergy in Classroom Instruction, 1-30," 2026.

[43] Ihichr, A., Oustous, O., Idrissi, Y. E. B. E., & Lahcen, A. A., "A Systematic Review on Assessment in Adaptive Learning: Theories, Algorithms and Techniques. International Journal of Advanced Computer Science and Applications, 15(7)," 2024.

[44] Rodríguez-Ortiz, M. Á., Santana-Mancilla, P. C., & Anido-Rifón, L. E., "Machine Learning and Generative AI in Learning Analytics for Higher Education: A Systematic Review of Models, Trends, and Challenges. Applied Sciences, 15(15), 8679," 2025.

[45] Munar-Garau, J., De-Benito-Crosetti, B., & Salinas, J., "Trends in Publications on AI Tools and Applications in Learning Design to Personalization of Learning—A Scoping Review. Information, 16(12), 1065," 2025.

[46] Silva, Lotfi, A., Isibor Kennedy Ihianle, Golnaz Shahtahmassebi, & Bird, J. J., "Generative Artificial Intelligence and the Future of Education: A Systematic Review of Trends, Trustworthiness and Technological Evaluation," 2025.

[47] Van Mechelen, M., Smith, R. C., Schaper, M.-M., Tamashiro, M. A., Bilstrup, K.-E. K., Lunding, M. S., Petersen, M. G., & Iversen, O. S., "Emerging Technologies in K–12 Education: A Future HCI Research Agenda. ACM Transactions on Computer-Human Interaction, 30(3)," 2022.

[48] Christos Koukaras, Stavrinides, S. G., Euripides Hatzikraniotis, Mitsiaki, M., Paraskevas Koukaras, & Christos Tjortjis, "Navigating the Future of Education: A Review on Telecommunications and AI Technologies, Ethical Implications, and Equity Challenges. Telecom, 7(1), 2–2," 2026.

[49] Weng, X., Ye, H., Dai, Y., & Ng, O., "Integrating Artificial Intelligence and Computational Thinking in Educational Contexts: A Systematic Review of Instructional Design and Student Learning Outcomes. Journal of Educational Computing Research, 62(6)," 2024.

[50] Ng, K., Min Goo Lee, Jun, R., Hu, X., J. Stephen Downie, & Kai, S., "A review of AI teaching and learning from 2000 to 2020. Education and Information Technologies, 28," 2022.

[51] Navas Bonilla, C. d. R., Viñan Carrasco, L. M., Gaibor Pupiales, J. C., & Murillo Noriega, D. E., "The Future of Education: A Systematic Literature Review of Self-Directed Learning with AI. Future Internet, 17(8), 366," 2025.

[52] Revenaya, M., Bambang Subali, & Ellianawati, E., "Exploring the Integration of Artificial Intelligence in Primary Education: A Systematic Literature Review (2020–2024). STEAM Journal for Elementary School Education, 1(02), 116–136," 2025.

[53] Yoga, I. P. S. R., Meliana, S., & Nurjanah, D. , "Adaptive Practice with Recommender System in Moodle Learning Management System for Introductory Programming Course," 2025.

[54] Rajesh, S., Rao, V. V., & Thushara, M. G. , "Comprehensive investigation of code assessment tools in programming courses," 2024.

[55] Pechbooranin, A., Chaisirithungnaklang, T., Tantrairatn, S., & Chamniprasart, K., "Multi-Classes Object Detection and Manipulation for Industrial Robots Using Deep Learning," 2025.

[56] Морзе, Наталія Вікторівна, Варченко-Троценко, Лілія Олександрівна, & Терлецька, Тетяна Сергіївна. , "Comprehensive framework for adaptive learning implementation in Moodle LMS: technical, pedagogical, and administrative perspectives," 2025.

[57] Abgaz, Y. , "An Integrated Framework for Interactive and Inclusive Asynchronous Online Learning at Scale: Data Literacy in Higher Education," 2026.

[58] Riva, G. , "The Architecture of Cognitive Amplification: Enhanced Cognitive Scaffolding as a Resolution to the Comfort-Growth Paradox in Human-AI Cognitive Integration," 2025.

[59] CORDIS, cordis.europa.eu. , "Breaking Educational Barriers with Contextualised, Pervasive and Gameful Learning," 2025.

[60] Naseer, F., & Khawaja, S. , "Mitigating Conceptual Learning Gaps in Mixed-Ability Classrooms: A Learning Analytics-Based Evaluation of AI-Driven Adaptive Feedback for Struggling Learners," 2025.

[61] Moon, J., McNeill, L., Christopher Thomas Edmonds, Seyyed Kazem Banihashem, & Omid Noroozi., "Using learning analytics to explore peer learning patterns in asynchronous gamified environments.," 2024.

[62] Papazian, P., & Pardos, Z. , "Frankenstein Curricula: Stitching Together Open Educational Resources to Generate Dynamic Learning Paths With AI.," 2025.

[63] Curasma, R. P., Huayta-Meza, F. T., Jara, N. J., Illesca-Cangalaya, O. F., Porras-Ccancce, L. E., Romero-Gabino, A. J., López-Robles, Z. M., & Llamoca, R. Q., "Arduino and mBlock Projects to Enhance Computational Thinking in First-Year Engineering Students.," 2025.

[64] Sweller, J. , "Cognitive Load during Problem Solving: Effects on Learning.," 1988.

[65] Bloom, B. S. , "Learning for Mastery. Instruction and Curriculum. Regional Education Laboratory for the Carolinas and Virginia, Topical Papers and Reprints, Number 1," 1968.

[66] Papert, S. , "Mindstorms: Children, Computers, and Powerful Ideas.," 1980.

[67] Cano, E., " Mastering XAMPP: A Comprehensive Guide to Managing a Local Server.," 2024.

[68] Wild, I., "oodle 4 Security: Enhance security, regulation, and compliance within your Moodle infrastructure," 2024.

[69] Al-Fraihat, D., Alshahrani, A. M., Alzaidi, M., Shaikh, A. A., Al-Obeidallah, M., & Al-Okaily, M. , "Exploring students' perceptions of the design and use of the Moodle learning management system," 2025.

[70] Srinivasan, B., Nehru, M., Parthasarathi, R., Mukherjee, S., & Thankachan, J. A., "Automated Computer Program Evaluation and Projects -- Our Experiences," 2024.