Designing AI-Resilient Embedded Programming Assignments Using a Hardware-in-the-Loop Inspired Emulator

Priit Ruberg[1,*], Eke-Martin Paas[1], Risto Heinsar[1] and Peeter Ellervee[1]

[1]Tallinn University of Technology, Ehitajate tee 5, 19086 Tallinn, Estonia

Abstract

The widespread availability of generative AI tools is challenging programming education: large language models can produce functionally correct solutions to many standard tasks without any understanding of the underlying system. This paper presents a protocol-compatible robot emulator designed to support AI-resilient embedded programming education in a 3 ECTS Python course for electronics and computer engineering students. Inspired by hardware-in-the-loop testing practices, the emulator acts as a drop-in replacement for a physical Arduino-based robot platform, communicating over the same custom serial protocol. Three behavioural modes — synthetic data generation, log replay, and fault injection — are unlocked progressively according to a server-side timestamp schedule, tying emulator functionality to the course calendar and limiting the window available for AI-assisted shortcuts. Students develop a PC-side application incrementally, progressing from a command-line interface through a graphical user interface to fault-handling extensions. The course concludes with a mandatory in-person assessment in which students connect their application to the physical robot and defend their implementation under questioning. Fault injection scenarios require runtime observation and cross-message reasoning that current language models do not reliably produce without hands-on system interaction, and the oral defence component provides a secondary verification layer that is resistant to AI-generated preparation.

Keywords

Embedded systems education, hardware-in-the-loop, fault injection, AI-resilient assessment, Python programming

CEUR-WS.org/Vol-4211/paper08.pdf

1. Introduction

Embedded systems programming courses traditionally rely on physical hardware platforms, which creates several practical challenges for teaching and assessment. Students must often configure hardware environments before meaningful software development can begin, and debugging interactions between software and hardware can be slow and difficult to reproduce. As a result, early development stages are often dominated by hardware setup issues rather than programming tasks.

At the same time, the rapid adoption of generative AI tools is transforming programming education. Modern large language models can quickly generate solutions for many conventional programming assignments, including communication clients, graphical user interface skeletons, protocol parsers, and even task-specific control programs for educational robots such as line-following or maze-solving applications. This development challenges traditional assignment design, as tasks that primarily involve code generation may no longer guarantee meaningful learning.

To address these challenges, this paper introduces a protocol-compatible hardware emulator designed to support embedded programming education. The emulator implements the same communication protocol as the physical robot platform and therefore acts as a drop-in replacement during early software development. Inspired by hardware-in-the-loop (HIL) testing practices, the system decouples software development from physical hardware while preserving authentic system interactions. In addition, the emulator supports staged behavioural scenarios and fault injection, enabling the design of programming assignments that emphasize runtime analysis and debugging rather than static code generation. This

Baltic DB&IS 2026 Conference Forum and Doctoral Consortium, 28 June - 1 July 2026, Tartu, Estonia
*Corresponding author.
priit.ruberg@taltech.ee (P. Ruberg); ekpaas@taltech.ee (E. Paas); risto.heinsar@taltech.ee (R. Heinsar); peeter.ellervee@taltech.ee (P. Ellervee)
0000-0003-3546-131X (P. Ruberg); 0009-0005-4794-6797 (R. Heinsar); 0000-0002-0745-6743 (P. Ellervee)
© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2026-06-09

approach provides a foundation for designing programming assignments that remain pedagogically effective in the presence of AI-assisted coding tools.

To structure the contribution of this work, we formulate three research questions that guide the system design and educational evaluation:

1. How can a protocol-compatible hardware emulator act as a drop-in replacement for physical robots in embedded systems programming education?
2. How can protocol-level behavioural scenarios and fault injection support systematic debugging and robustness testing in student-developed embedded applications?
3. How can staged emulator functionality contribute to AI-resilient assignment design in programming education?

This paper contributes more than a course description. It introduces:

- a protocol-compatible, hardware-in-the-loop inspired emulator architecture that functions as a drop-in replacement for a physical robot platform in embedded systems education;
- a temporally gated behavioural unlocking mechanism that couples system complexity to the course calendar;
- a fault-injection-based assessment design that shifts evaluation from static artefacts to documented runtime reasoning;
- an integrity-oriented emulator infrastructure incorporating binary distribution, runtime randomisation, and session-specific verification tokens.

2. Background and related work

Three research directions are particularly relevant for the present work: robotics-based programming education, simulation environments for hardware-oriented learning, and the emerging impact of generative AI tools on programming assessment.

2.1. Robotics and hardware-based programming education

Educational robots are widely used to connect programming concepts with observable physical behaviour in engineering curricula. Hands-on robotics activities have been shown to improve engagement and support experiential learning in STEM-oriented courses [1]. Such platforms allow students to directly observe the consequences of software decisions on sensors, actuators, and control systems.

In embedded systems education these approaches are often described as *hardware-close programming*. Earlier work in our department introduced micro-controller- and FPGA-based robot kits that enable students to connect introductory programming concepts with real-world hardware tasks [2]. The platform was later extended with a flexible controller architecture supporting wireless communication and modular hardware expansion [3]. Related work has also explored software analysis techniques for embedded systems development on such platforms [4].

While these approaches provide valuable experience with embedded devices, they typically require continuous access to physical hardware during the development process. This limitation motivates the use of simulation and emulation techniques that can provide similar system interaction in a controlled environment.

2.2. Simulation and emulation in engineering education

Simulation environments and remote laboratories are widely used to overcome practical constraints associated with hardware-based courses, such as limited equipment availability and laboratory access. Systematic studies of remote and virtual laboratories show that simulator-based environments can effectively support STEM and STEAM learning while improving scalability of laboratory teaching [5].

During the COVID-19 pandemic, similar approaches became essential for maintaining laboratory-based courses. Remote laboratories and emulator-based experiments enabled engineering courses to continue without physical access to hardware systems [6]. These experiences demonstrated that software-based experimentation environments can preserve key learning outcomes even when physical laboratory resources are unavailable.

The present work builds on these ideas but positions the emulator not merely as a contingency solution for restricted laboratory access, but as the primary development environment during the software design phase of the course.

2.3. Generative AI in programming education

The rapid emergence of large language models (LLMs) such as ChatGPT has introduced new challenges for programming education. Generative AI systems are capable of producing syntactically correct and often functionally adequate code for many conventional programming tasks, including protocol parsers, communication clients, and graphical interface templates.

Recent research highlights both opportunities and risks associated with the use of generative AI in STEM education. While AI tools can support learning by providing explanations and debugging assistance, they also raise concerns regarding assessment integrity and student over-reliance on automatically generated solutions [7]. Bibliometric studies show a rapidly expanding body of research on the use of large language models in programming education, particularly in areas such as code generation, tutoring, and automated feedback [8].

These developments have motivated new approaches to programming assessment that emphasise runtime reasoning, debugging, and system interaction rather than static code production. Our previous work explored scalable digital assessment mechanisms and integrity-preserving evaluation strategies in large engineering courses [9]. In parallel, curriculum design efforts have focused on introducing active learning and project-based structures in early engineering education [10].

While prior work has examined robotics in programming education, simulation environments in engineering laboratories, and the educational implications of generative AI tools, these strands are rarely combined within a single learning architecture. The present work integrates these perspectives by introducing a protocol-compatible robot emulator that functions as a controlled software environment for developing host-side applications interacting with embedded systems.

3. System architecture

The system consists of three main components: the student-developed PC-side application, the physical robot platform with its wireless gateway, and the robot emulator. Fig. 1 presents an overview of the platform architecture. The PC-side application communicates with either the physical robot or the robot emulator through the same serial interface, using a custom communication protocol described in Section 3.2. This protocol-level interchangeability is the foundational property that allows the emulator to serve as a drop-in replacement for the physical hardware.

3.1. Robot platform

The physical robot used in this course is a customised Parallax Boe-Bot [11], shown in Fig. 2. The main customisation is a topmost black PCB that simplifies adding hardware modules. The board hosts a Nordic Semiconductor NRF24L01+ [12] radio module and an HC-SR04 ultrasound module [13]. In addition, the PCB provides two user-controllable LEDs and a push button. The servo motors and QTI infrared line sensors connect via pin headers, allowing the hardware configuration to be changed between tasks.

The robot platform is also used in a first-year hardware-close programming course in which students implement line-following and maze-navigation behaviours directly on the microcontroller. These tasks

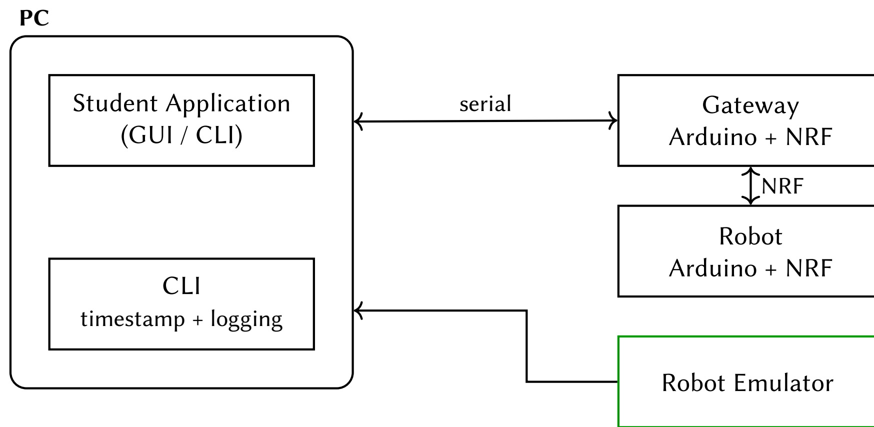
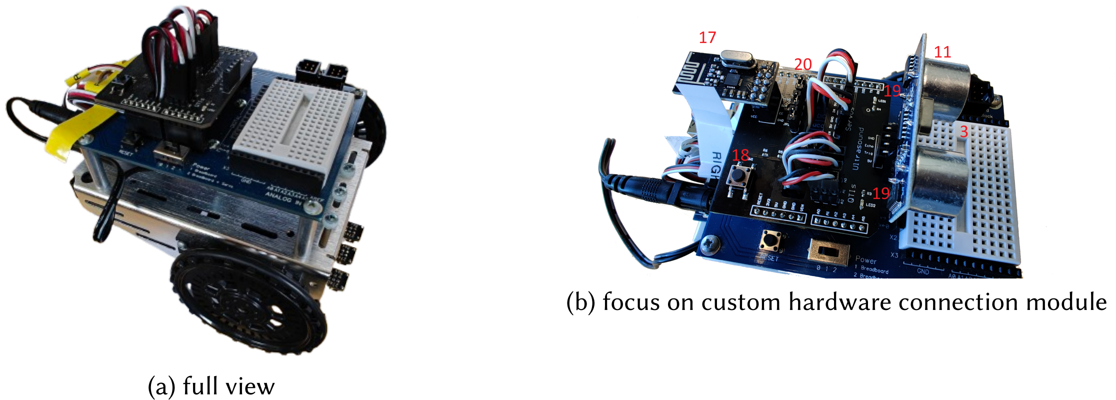


Figure 1: Platform architecture overview.



(a) full view

(b) focus on custom hardware connection module

Figure 2: Customised Parallax Boe-Bot platform with NRF24 radio module and custom PCB.

provide the hardware context for the Python course described in this paper, but the course itself focuses exclusively on host-side software that communicates with the robot via the defined protocol.

3.2. Communication protocol

The communication protocol is independent of any specific programming language or communication hardware. Also the protocol is designed in a way that it takes into account the functionality of our educational robot but on the other hand is extendable by different apps that the robot can execute.

Table 1 summarises the message types defined in the protocol. There are five messages that use incremental coding in the payload that can be grouped to four groups: `periodic`; `app based`; `command associated` and `faults`. **Status** - `periodic` status message from robot to host; one-sided heartbeat. **Event** - `app based` message to indicate events during app execution. **Command** - `command` from host to robot that always needs respose. **Fault** - `fault` message as response to command or some other internal issue.

In Fig. 3 is presented the sequence diagram of the protocol messages shown in Table 1. There are four main message sequences: `beacon`, `event`, `command` / `response` and `fault`. Also for each signal, originating from robot the robot_id value is required, however for brevity it is not shown in the figure.

Status message is the only periodic message that is meant to function as a beacon message. In our case the beacon is one-sided, meaning that the host does not respond to it. Whenever the robot has power it starts to transmit the status message with 1 second interval. The message length is 6 bytes and consists of minimal possible required data about the robot and it's current program.

Table 1
Protocol messages overview

Message	Code	Direction	Frequency	Feedback
status	0	Robot → Host	Regular (1 s)	No
event	1	Robot → Host	Event-based	No
command	2	Host → Robot	Event-based	Yes
response	3	Robot → Host	Event-based	No
fault	4	Robot → Host	Event-based	No

All messages start with robot_id field, followed by payload.

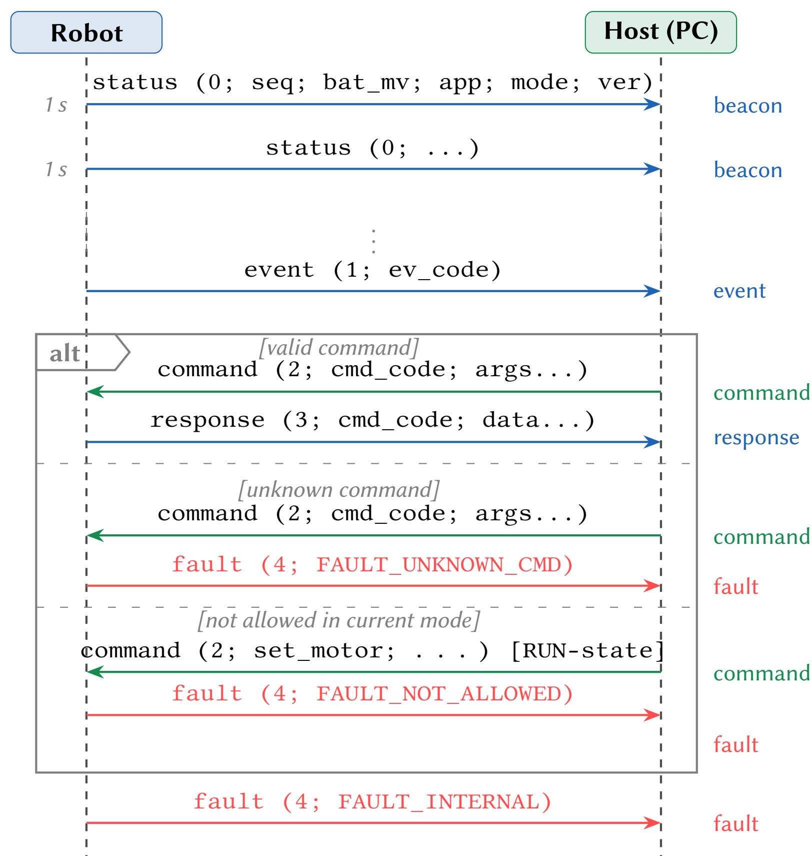


Figure 3: Protocol message flow with status beacons, events, command/response pairs, and fault responses.

Event message is an app-based message that the robot sends whenever a significant change occurs during application execution. Unlike the status message, events are not periodic but triggered by specific conditions such as a detected line, a wall, or a mode change. The event message carries a single event code that identifies the occurrence.

Command message is sent from the host to the robot to request an action or configuration change. Every command must be answered by the robot with either a response or a fault message. Commands are divided into core commands, which are supported by all applications, and app-specific commands that depend on the active application.

Response message is sent by the robot as a direct reply to every received command. The response echoes the command code so that the host can match the reply to the original request, followed by the requested data.

Fault message is sent when a command cannot be executed or when an internal error is detected. A

fault represents a protocol-level response to an invalid or disallowed request and does not necessarily indicate a system failure.

3.3. Robot emulator

The robot emulator (Fig. 1) is a software component that implements the same communication protocol as the physical robot platform. From the perspective of the student application, the emulator behaves as the physical robot: it receives commands over a serial connection, responds according to the protocol, and emits periodic status messages. This protocol-level transparency establishes a hardware-in-the-loop inspired abstraction layer in which the physical plant is replaced by a behaviourally equivalent software component while preserving the interaction contract. Unlike conventional simulators that model only nominal behaviour, the emulator deliberately exposes controlled deviations from expected runtime behaviour.

The emulator supports three behavioural modes, shown in Fig. 4. In *replay mode*, the emulator plays back a previously recorded log file, reproducing the exact message sequence of a real robot run. Fig. 5 shows an example of such a log, captured from a physical robot transmitting periodic status messages. In *synthetic mode*, the emulator generates protocol-compliant messages autonomously without requiring a real robot log, allowing consistent and reproducible test environments to be created independently of physical hardware. In *fault injection mode*, the emulator deliberately introduces anomalous behaviour into the message stream, such as out-of-range sensor values, malformed responses, or unexpected fault codes.

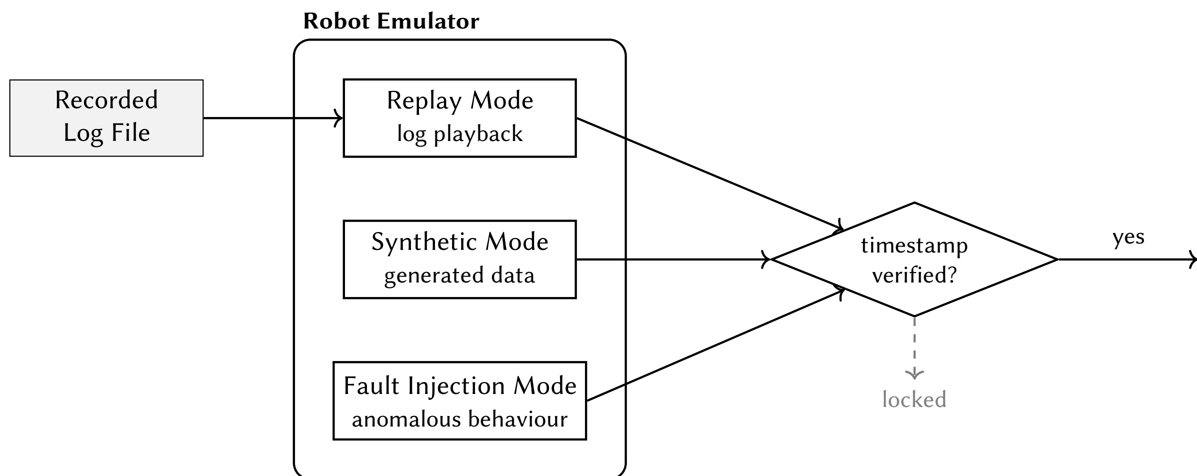


Figure 4: Robot emulator behavioural modes. Access to fault injection mode is gated by a server-side timestamp verification.

Access to fault injection mode is gated by a server-side timestamp retrieved at runtime. The emulator executable verifies the current date against a predefined schedule before unlocking each mode. This mechanism ensures that students cannot access fault injection scenarios before the corresponding course material has been covered, and prevents the entire assignment from being completed in a single session.

The robot firmware and protocol implementation are provided as stable and documented infrastructure. Students do not modify the embedded C code during the Python course. This deliberate separation ensures that the educational focus remains on host-side system integration rather than low-level firmware debugging.

4. Educational design

In contrast to the first-year hardware-close programming course, students in the Python course do not modify the robot firmware and do not write any C code. The robot-side implementation, including

```
21:15:43  [BEACON] 1;0;97;11421;2;0;1
21:15:44  [BEACON] 1;0;98;11417;1;0;1
21:15:45  [BEACON] 1;0;99;11410;3;0;1
21:15:46  [BEACON] 1;0;100;11400;3;0;1
21:15:47  [BEACON] 1;0;101;11398;1;0;1
```

Figure 5: Example of status messages from the physical robot, used as input for the emulator in replay mode.

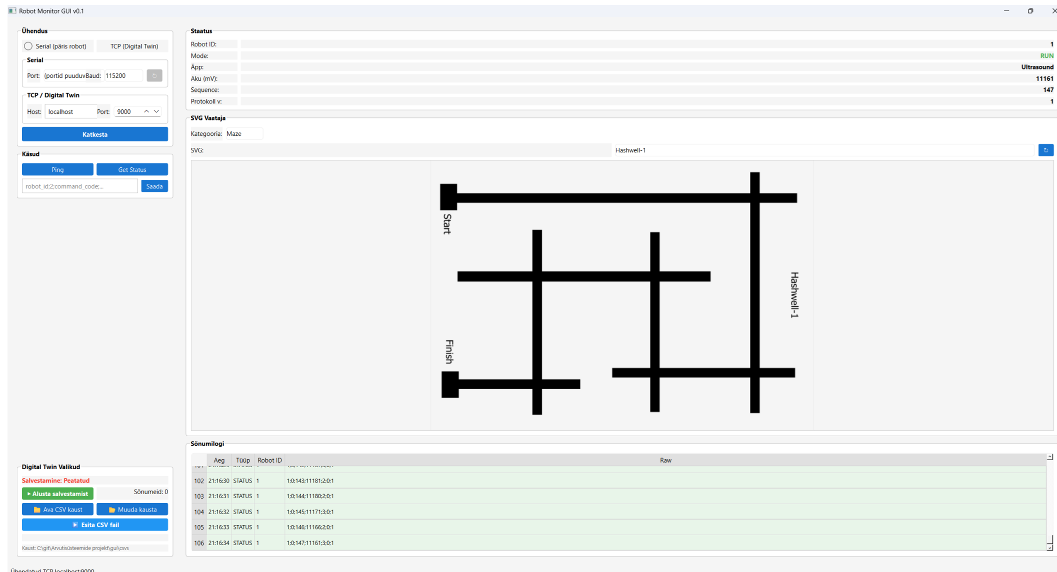


Figure 6: Example of the GUI showing robot status fields, a maze track visualisation loaded from an SVG file, and the incoming message log. The application communicates with either the physical robot or the emulator via the same protocol interface.

the communication protocol and embedded control logic, is provided as a fixed and documented platform. This separation allows students to focus exclusively on developing the PC-side application in Python, without being distracted by hardware configuration or low-level embedded debugging. The emulator therefore acts as a controlled systems interface rather than a simplified substitute for firmware development.

The educational design builds on the system architecture described in Section 3. Rather than serving as a convenience tool, the emulator acts as the primary learning environment: students develop the PC-side application shown in Fig. 1 from scratch, progressing through milestones aligned with the staged behavioural modes of the emulator. Fig. 6 shows an example of a student-developed GUI that visualises robot state and displays the incoming message log.

4.1. Staged interaction and progressive emulator behaviour

The course is structured as a 16-week programme. The first six weeks introduce Python fundamentals independently of the robot platform. From week seven onward, students work with the robot emulator in weekly team-based tasks, each of which must be demonstrated to the course instructor. The emulator mode is visible to the student at all times, so that understanding the current behavioural context is part of the task rather than an additional puzzle.

In weeks 7–10, students implement the CLI component of the PC-side application. The emulator operates in either synthetic or replay mode at the student's choice. In synthetic mode the emulator generates protocol-compliant messages autonomously, providing a predictable environment for initial development. In replay mode it reproduces a recorded log from a physical robot run, exposing students

Table 2
Course stages and emulator configuration

Weeks	Stage	Emulator mode	Deliverable
1–6	Python fundamentals	—	Weekly exercises
7–10	CLI development	Synthetic / Replay	Working CLI
11–13	GUI development	Synthetic / Replay	Working GUI
13–14	Fault handling	Fault injection (unlocked)	Extended GUI
14–16	In-person assessment	Physical robot	Oral defence

to realistic message timing and content variation. Both modes behave correctly with respect to the protocol: all valid commands receive well-formed responses, and status messages are emitted at regular intervals. This stage requires students to implement protocol parsing, command construction, connection handling, and basic error detection, but does not yet involve unexpected runtime behaviour.

In weeks 11–14, students extend the CLI with a graphical user interface. The protocol layer implemented in the previous stage requires no modification, which reinforces the value of modular design and protocol abstraction. At approximately week 13, fault injection mode becomes available. From this point the emulator may introduce anomalous behaviour into the message stream: out-of-range sensor values, responses that contradict the subsequent status message, or unexpected fault codes. Students must extend their application to detect, classify, and respond to these anomalies. Because the emulator mode is visible, students know when fault injection is active, but they do not know which fault will occur or when it will appear.

In weeks 14–16, students attend a mandatory in-person assessment in which they connect their application to the physical robot and demonstrate its functionality. The physical robot does not replicate the fault injection scenarios, but its behaviour differs from the emulator in timing, radio latency, and sensor noise. An application developed exclusively against a predictable emulator and never run against real hardware is likely to fail at this stage. The in-person format also requires students to explain observed behaviour on demand, providing a secondary verification layer that is independent of the automated tests.

Table 2 summarises the course structure and the corresponding emulator configuration at each stage. The alignment between course milestones and emulator modes is intentional: each newly unlocked behavioural capability corresponds to a previously introduced conceptual topic in lectures. This temporal coupling between theory and system behaviour forms the backbone of the AI-resilient assignment design.

4.2. Assignment structure

The assignment is decomposed into three milestones that correspond to the staged emulator behaviour described above.

1. **Protocol client.** Implement a CLI that sends core commands to the emulator, receives and parses responses, and displays the incoming status stream. The emulator operates in synthetic mode.
2. **Graphical interface.** Extend the CLI with a GUI that visualises robot state and allows interactive command submission. No changes to the protocol layer are required.
3. **Fault handling.** Extend the application to handle fault injection scenarios introduced by the emulator. Students must identify the fault type from the protocol message, log the occurrence, and implement a recovery or reporting strategy.

Each milestone is delivered as a Git tag, and progress is assessed through a combination of automated protocol-level tests and a short oral defence.

4.3. Fault injection and debugging tasks

Fault injection scenarios are designed to require runtime observation and reasoning rather than code generation. Two representative scenarios are described here.

In the *LED command anomaly* scenario, the emulator accepts a `set_led` command and returns a syntactically valid response, but subsequently emits a status message in which the reported LED state does not match the value that was set. A student application that does not cross-validate command responses against the status stream will silently accept the inconsistency. Detecting this fault requires the student to implement state tracking across multiple message types, a task that current large language models do not reliably produce without explicit prompting and observable runtime feedback.

In the *out-of-range value* scenario, the emulator inserts a sensor reading that exceeds the maximum value defined in the protocol, for example a battery voltage of `0xFFFF` when the defined operational range is considerably lower. Students must identify the anomalous value, classify it as a protocol violation rather than a hardware fault, and decide whether to discard, flag, or escalate the reading. This scenario exercises protocol awareness and defensive programming rather than algorithmic problem solving.

Both scenarios are introduced only after the corresponding protocol sections have been covered in lectures, enforced by the timestamp gating mechanism described in Section 3.

The combination of staged unlocking, visible mode indication, and in-person assessment addresses AI-assisted completion at three levels. First, fault injection scenarios cannot be solved by generating code from the protocol specification alone, because the correct response depends on runtime observations that vary between emulator sessions. A student who submits AI-generated fault-handling code without having run it against the emulator is unlikely to produce a correct written fault report, since the report must reference specific message sequences observed during execution. Second, the timestamp gating limits the window in which fault injection is available, reducing the time available for iterative AI-assisted trial and error. Third, the in-person assessment requires the student to operate their application against the physical robot and to explain its behaviour under questioning. This stage cannot be prepared for by code generation alone, as it depends on having developed a working understanding of the protocol interaction through repeated hands-on use of the emulator.

From a pedagogical perspective, these scenarios operationalise defensive programming principles. Students are required to treat the communication partner not as a perfectly reliable component but as a potentially inconsistent system, thereby internalising robustness as a design requirement rather than an afterthought.

Weekly in-person demonstrations further reduce the feasibility of delegating the assignment entirely to AI tools. Students must explain their implementation decisions, reproduce observed faults on demand, and justify their recovery logic. This continuous verification shifts the emphasis from artefact submission to process understanding.

4.4. Integrity-oriented design considerations

The emulator is not distributed in source form. It is packaged as a compiled executable and distributed in binary form to reduce direct inspection of its internal fault injection logic. While reverse engineering cannot be completely prevented, the goal is to increase the effort required to predict fault behaviour without executing the system.

Fault injection scenarios are partially randomised at runtime. For example, anomaly positions and injected values may vary between sessions within defined protocol constraints. This design prevents students from relying on fixed expected outputs or sharing static solution templates.

To support assessment integrity, the emulator can optionally emit a verification token derived from the injected fault sequence. The student application is required to include this token, together with a structured fault report, as part of the milestone submission. The instructor can decode the token server-side to verify whether the reported anomalies correspond to the actual injected events. This mechanism shifts the assessment from code inspection to behavioural verification.

5. Example scenario

To illustrate the fault injection mechanism in a concrete course context, this section presents the LED command anomaly scenario in detail.

The `set_led` command (code 3) accepts three arguments: the LED identifier, the mode, and the desired state. The student application sends the following command to turn on LED 1:

```
robot_id;2;3;1;1;1
```

The emulator acknowledges the command with a well-formed response indicating success:

```
robot_id;3;3;0
```

However, the subsequent status message reports the LED state as off:

```
robot_id;0;12;11800;1;0;1
```

A naive implementation that trusts the response without monitoring the status stream will consider the command successful. The fault becomes observable only when the student application correlates the response with the LED state field in the status message.

Students are required to produce a written fault report as part of the milestone submission. The report must identify the message sequence that revealed the fault, state which protocol field contained the anomalous value, and describe the recovery action taken by their application. This requirement shifts the assessment focus from working code to documented reasoning about system behaviour, a task that is significantly more difficult to delegate to an AI tool than code generation alone.

In the extended implementation used for assessment, the emulator additionally produces a session-specific verification hash derived from the injected anomaly sequence. Students must submit this hash together with their fault report. Because the hash is computed server-side from the actual runtime behaviour, it cannot be reconstructed from the protocol specification alone. This mechanism provides an additional integrity layer without requiring invasive code inspection.

6. Discussion

The following discussion examines the scalability of the proposed approach, its effectiveness against AI-assisted code generation, and the limitations of the current implementation."

The emulator architecture is not specific to the Boe-Bot platform. Any robot or embedded device that communicates via a defined serial protocol can in principle be emulated using the same approach. Extending the emulator to a new platform requires implementing the target protocol and providing a set of synthetic or recorded scenarios; the timestamp gating and fault injection mechanisms are platform-independent.

The staged unlocking mechanism scales naturally to courses of varying length. The number of milestones, the complexity of fault injection scenarios, and the unlock schedule can all be adjusted without modifying the student-facing protocol interface.

6.1. AI resilience

In this context, AI-resilience does not imply that assignments are immune to AI assistance. Rather, it denotes that successful completion requires sustained interaction with a live system, iterative debugging, and contextual reasoning about runtime behaviour. Large language models can assist in code generation, but they cannot replace the experiential component of observing and interpreting system responses. The emulator-based design therefore shifts the locus of assessment from static artefacts to dynamic interaction. Our approach addresses AI-assisted completion at three levels that correspond to the three research questions posed in the introduction.

At the *protocol level* (RQ1), the emulator is a drop-in replacement for the physical robot, but it is not a static target. In synthetic mode it generates messages autonomously; in replay mode it reproduces recorded hardware behaviour; in fault injection mode it introduces anomalies that are not predictable from the protocol specification. A student who prompts a large language model with the protocol specification and asks for a complete solution will obtain code that handles the nominal case correctly,

but is unlikely to handle fault injection scenarios without observable runtime feedback. The emulator therefore raises the bar for AI-assisted completion beyond what can be achieved through specification-only code generation.

At the *assessment level* (RQ2), the written fault report requires students to document specific message sequences observed during execution. This requirement cannot be satisfied by a student who has not run their application against the emulator, because the fault sequences are not deterministic across sessions and cannot be fabricated from the protocol specification alone. The oral defence component at the in-person assessment extends this requirement to live explanation of observed behaviour, which is difficult to prepare for without genuine hands-on experience.

At the *temporal level* (RQ3), the staged unlocking mechanism ties emulator functionality to the course schedule. Fault injection mode becomes available only at week 13, limiting the time window for AI-assisted trial-and-error to the final weeks before the in-person assessment. The in-person assessment itself requires physical presence and real hardware, neither of which can be replaced by an AI.

We acknowledge that a sufficiently capable AI system with access to both the protocol specification and a set of recorded emulator logs could in principle generate fault-handling code that satisfies the automated tests. The oral defence and the physical robot assessment are designed to detect this case. The long-term robustness of the approach depends on continued revision of fault injection scenarios as AI capabilities develop, which the modular emulator architecture supports without changes to the student-facing protocol interface.

The objective is not to detect AI-generated code but to ensure that successful completion requires genuine interaction with the system. Even if AI tools assist in implementation, the student must execute, observe, interpret, and explain runtime behaviour that is not statically derivable from the protocol description.

6.2. Limitations

The current implementation does not yet support multi-robot scenarios, which limits its applicability to the VIP robot project task described in Section 3.1. Extending the emulator to simulate two communicating robots is planned as future work. In addition, the fault injection scenario library is currently small; expanding it to cover a wider range of protocol violations and timing anomalies would increase the robustness of the assessment design.

At the time of submission, the emulator and assignment infrastructure have been functionally implemented and internally tested during course preparation. The first full cohort deployment will take place in the upcoming academic term. A structured evaluation of learning outcomes is planned as future work.

7. Conclusion

This paper presented a protocol-compatible robot emulator designed to support AI-resilient embedded programming education. The emulator acts as a drop-in replacement for a physical robot platform, supports synthetic and replay-based operation, and introduces fault injection scenarios that require runtime reasoning rather than static code generation. A staged unlocking mechanism ties emulator functionality to the course schedule, limiting the opportunity for AI-assisted solutions without genuine engagement with the system.

The educational design structures student work around incremental protocol client development, progressing from a CLI through a GUI to fault-handling extensions. Assessment combines automated protocol-level testing with oral defence, targeting skills that are not directly delegated to current AI tools.

Future work will extend the emulator to multi-robot scenarios, expand the fault injection library, and evaluate the approach with student cohorts to measure its effectiveness as an AI-resilience measure in practice.

Declaration on Generative AI

During the preparation of this work, the authors used ChatGPT-EDU in order to: Grammar, spelling check and generate figure 3 structure. After using these tool(s)/service(s), the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

Acknowledgments

The authors would like to thank Edmond Aru, Kristo Kangur, Jörgen Varjas and Karl Michal for their contribution during the early development discussions of the communication protocol.

References

[1] G. Georgiev, G. Hristov, P. Zahariev, D. Kinaneva, V. Kolev, R. Stewart, Enhancing engineering education with robotics: From theory to practice, in: Proc. European Association for Education in Electrical and Information Engineering Annual Conference (EAEEIE), 2025, pp. 1–7. doi:10.1109/EAEEIE65428.2025.11136346.

[2] H. Kruus, M. Brik, M. Kruus, P. Ruberg, V. Viies, P. Ellervee, Hardware close programming for freshmen, in: Proceedings of the 10th European Workshop on Microelectronics Education (EWME), Tallinn, Estonia, 2014, pp. 93–96.

[3] P. Ruberg, A. Guitar, P. Ellervee, Flexible controller for educational robot kit, in: Proceedings of the IEEE International Conference on Microelectronics Systems Education (MSE), 2015, pp. 17–20. doi:10.1109/MSE.2015.7223367.

[4] P. Ruberg, K. Lass, P. Ellervee, Microcontroller energy consumption estimation based on software analysis for embedded systems, in: Proceedings of the 2015 Nordic Circuits and Systems Conference (NORCAS), 2015, pp. 1–4. doi:10.1109/NORCHIP.2015.7364397.

[5] N. M. Mamani, F. J. García-Peñalvo, M. Á. Conde, J. Gonçalves, A systematic mapping about simulators and remote laboratories using hardware-in-the-loop and robotics for developing stem/steam skills in pre-university education, in: Proc. Int. Symp. on Computers in Education (SIIE), 2021, pp. 1–6. doi:10.1109/SIIE53363.2021.9583622.

[6] P. Ruberg, P. Ellervee, K. Tammemäe, U. Reinsalu, A. Rähni, T. Robal, Surviving the unforeseen: Teaching IT and engineering students during COVID-19 outbreak, in: Proceedings of the IEEE Frontiers in Education Conference (FIE), Uppsala, Sweden, 2022, pp. 1–9. doi:10.1109/FIE56618.2022.9962383.

[7] K. Ru, G. Li, Challenges of generative ai in stem education, in: Proc. Int. Conf. on Computer Science and Technologies in Education (CSTE), 2025, pp. 543–547. doi:10.1109/CSTE64638.2025.11092081.

[8] J. P. Amos, O. A. Amodu, R. A. R. Mahmood, A. B. Abdulqudus, A. F. Zakaria, A. R. Iyanda, U. A. Bukar, Z. M. Hanapi, A bibliometric exposition and review on leveraging llms for programming education, IEEE Access 13 (2025) 58364–58393. doi:10.1109/ACCESS.2025.3554627.

[9] P. Ruberg, H. Lensen, E. Orasson, Transforming assessment for 250+ students per year: From paper exams to secure, scalable, and continuous digital evaluation, in: Proceedings of the 28th International Conference on Interactive Collaborative Learning (ICL2025), Volume 1, 2026, pp. 612 – 620. doi:10.1007/978-3-032-18888-5.

[10] P. Ruberg, P. Ellervee, R. Heinsar, H. Selg, A. Eek, From structured labs to agile teams: A curriculum-based approach to active learning in first-year computer engineering, in: Proc. IEEE Frontiers in Education Conference (FIE), 2025, pp. 1–9. doi:10.1109/FIE63693.2025.11328686.

[11] Parallax Inc., Boe-bot robot, https://www.parallax.com/boe-bot-robot/, June 6, 2026.

[12] Nordic Semiconductor ASA, nrf24 series, https://www.nordicsemi.com, June 6, 2026.

[13] SparkFun Electronics, Hc-sr04 ultrasonic sensor datasheet, https://cdn.sparkfun.com, June 6, 2026.