



An Agentic AI-Driven End-to-End IoT Engineering Platform: A Proposed Workflow from Natural Language to IoT Prototype

Satish Neval¹, Mandar Patil², Omkar Shinde³, Prof. Hrushikesh Ingole⁴

¹Department of Artificial Intelligence & Machine Learning, ISBM College of Engineering, Pune, India, Email => nevalsatish37@gmail.com

²Department of Artificial Intelligence & Machine Learning, ISBM College of Engineering, Pune, India, Email => mandarpatil13579@gmail.com

³Department of Artificial Intelligence & Machine Learning, ISBM College of Engineering, Pune, India, Email => odshinde1234@gmail.com

⁴Department of Artificial Intelligence & Machine Learning, ISBM College of Engineering, Pune, India, Email => hrushikesh.ingole@isbmcoe.org

ABSTRACT :

Developing an Internet of Things (IoT) prototype requires coordinated decisions about requirements, hardware, circuit connections, embedded firmware, and validation. These tasks can be challenging for beginners because an error in one stage may prevent the complete system from working. This paper presents the proposed design of an Agentic AI-Driven End-to-End IoT Engineering Platform that aims to translate natural-language descriptions into structured IoT prototype plans. The proposed workflow includes requirement analysis, ESP32-oriented component and pin planning, context-aware firmware generation, validation support, user approval, and deployment guidance. The current Circuitra user-interface prototype shows workspace areas for projects, components, code, simulation, AI, hardware, and console. However, the supplied screenshot shows an empty circuit canvas, no selected board, and an idle simulation. The screenshot is therefore treated as evidence of the frontend layout only; backend integration, generated circuits, firmware execution, and experimental performance have not yet been verified. This paper documents the problem, proposed architecture, workflow, and planned evaluation rather than claiming completed implementation results.

Keywords: Agentic AI; Internet of Things; ESP32; Natural Language Processing; Firmware Generation; IoT Engineering

Introduction

The Internet of Things (IoT) connects sensors, actuators, embedded controllers, and software services to support monitoring and automation. Building an IoT prototype commonly requires a developer to understand requirements, select compatible components, map pins, create firmware, and test the result. Beginners may find these stages difficult because information is distributed across board documentation, component datasheets, and programming examples.

Large language models can interpret natural-language requests and generate code, but generated outputs are not automatically correct for a particular board or circuit. A complete workflow should connect requirement interpretation with hardware-aware planning, code generation, validation, and human approval. This paper proposes a beginner-oriented platform that organizes these stages, initially focusing on ESP32-based prototypes.

Problem Statement

A beginner may describe an IoT idea in ordinary language but may not know which components to use, how to connect them to a controller, or how to produce and validate suitable firmware. A single generated code answer does not necessarily provide a consistent prototype plan. The problem addressed here is the absence of a unified guided workflow that links requirement interpretation, component and GPIO planning, firmware generation, validation status, and deployment instructions while keeping the user involved in important decisions.

Objectives

The proposed platform aims to: (i) translate natural-language IoT ideas into structured requirements; (ii) recommend ESP32-oriented components and GPIO/interface connections using technical references; (iii) prepare firmware with assumptions and dependencies documented; (iv) show validation status only for checks actually executed; and (v) keep the user's approval before deployment and provide deployment guidance. These are project objectives, not claims that every feature has been implemented.

Literature Review

Peng and Gao et al. introduced IoTPIlot, an LLM-driven tool for programming embedded IoT applications from natural language. Their work identifies conflicts between model-generated code and operating-system-specific APIs and uses external documentation to improve code generation across embedded OS environments [1]. Yang et al. presented AutoEmbed, which connects component-aware library resolution, code generation, compilation, flashing, and hardware execution. Its evaluation covers multiple development platforms and IoT tasks, demonstrating a broad automation pipeline rather than code-only assistance [2]. Luo et al. proposed IoTGen for natural-language-driven schematic and PCB generation, combining component retrieval, schematic creation, and PCB layout; its emphasis is hardware design and PCB output [3].

Li et al. studied skills-based agents for embedded and IoT development and evaluated them with hardware-validated tasks, emphasizing that expert knowledge and real-device checks matter beyond code generation [4]. SPELL addresses secure SoC design by translating security requirements into vulnerability checks and hardware assertions [5]. SafeCodeRL explores security-constrained multi-agent code generation for IoT/CPS software [6]. Together, these studies cover natural-language embedded programming, end-to-end deployment, PCB generation, hardware-validated agent workflows, and secure code/hardware design. They provide relevant baselines, but their scopes and evaluation settings differ; the comparison below is qualitative and does not imply a direct performance ranking.

Research Gap

The reviewed systems provide substantial capabilities in individual or closely coupled areas: embedded code generation and deployment [1,2], schematic/PCB generation [3], hardware-validated agent skills [4], and security-focused design or code generation [5,6]. The gap motivating this project is a beginner-oriented, reviewable workflow that brings requirement clarification, ESP32-focused component/GPIO planning, firmware preparation, visible validation status, and explicit user approval together in one interface. The intended novelty is this integration and its staged interaction design—not a claim that the project is the first IoT agent, nor that it outperforms existing systems. The current evidence supports a designstage contribution only; comparative effectiveness remains to be tested.

Table 1. Qualitative comparison of relevant approaches and the proposed framework.

Approach	Main scope	Evidence / strength reported	Relation to this proposed framework
IoTPIlot (2025) [1]	Natural-language embedded IoT programming	Uses external OS/API documentation to improve generated code	Code-generation focus; proposed framework adds a staged beginner workflow and approval/status presentation
AutoEmbed (2026) [2]	Embedded code generation through compile/flash/runtime	Evaluated across boards, modules, and IoT tasks	Broad end-to-end automation; this project is narrower and currently designstage, targeting ESP32-oriented planning
IoTGen (2026) [3]	Schematic and PCB generation	Component retrieval, schematic generation, PCB layout workflow	Hardware/PCB design focus; proposed framework also organizes requirements, firmware, and user review
Skilled agents (2026) [4]	Agent skills for embedded/IoT tasks	Hardware-validated benchmark and experiments	Highlights value of technical knowledge and physical checks; informs the proposed validation plan
SPELL / SafeCodeRL (2024/2026) [5,6]	Secure SoC verification / security-constrained code generation	Security-oriented checks and constrained generation	Security is complementary; proposed framework should include security review but has no verified security module yet
Proposed Circuitra workflow	Requirements → ESP32 plan → firmware → validation → approval	Frontend workspace screenshot only; no verified end-to-end results	Proposed integration of these stages; implementation and effectiveness still to be evaluated

Proposed System Architecture

The proposed architecture is divided into modular stages. The user interface accepts an IoT description and displays project, component, code, simulation, AI, hardware, and console workspaces. A requirement-analysis module extracts functions, inputs, outputs, constraints, and missing information. A hardware-planning module proposes compatible components and GPIO mappings using verified board and component references. A code-generation module prepares firmware for the selected board and components.

A validation module is intended to run available static checks and compilation checks through configured embedded-development tools. It must report only checks that were actually executed and must not mark unexecuted checks as passed. An approval and deployment module presents the proposed plan and code for human review and then provides deployment or flashing instructions. A curated retrieval-augmented generation (RAG) knowledge base may supply board documentation, datasheets, and approved code examples. The exact model and integrations should be documented only after they are implemented and verified.



Fig. 1. Proposed system architecture from Project Review III.

The intended workflow is: natural-language request → requirement analysis → hardware selection → GPIO/circuit plan → firmware generation → validation → user approval → deployment guidance → monitoring and improvement. Each stage should pass relevant outputs to the next stage and preserve assumptions and unresolved questions for the user.

Methodology

The proposed workflow is organized as six stages. (1) Input and clarification: the user enters an IoT idea in natural language. The requirement-analysis stage extracts the goal, sensor inputs, actuator outputs, communication needs, power constraints, and expected behaviour; ambiguous or missing information is returned to the user as a clarification question. (2) Context record: confirmed requirements, assumptions, selected board, component choices, pin map, code version, and validation messages are kept together so later stages can trace decisions. These are planned data fields, not evidence of a completed backend.

(3) Hardware planning: for the selected ESP32 board, the system is intended to suggest components and GPIO/interface mappings using board documentation and component datasheets. It should check pin restrictions, digital/analog or I2C/SPI/UART/PWM needs, voltage compatibility, and power requirements; uncertain decisions must be marked for human review. (4) Firmware preparation: after the user reviews the hardware proposal, a codegeneration stage is intended to create firmware aligned with the confirmed requirements, target board, pin definitions, and library dependencies. Each generated item should retain assumptions and unresolved issues.

(5) Validation and approval: the planned validation layer should distinguish source review, compilation, supported simulation, and physical hardware testing. It must report only checks actually run, with tool/version and diagnostics, and use statuses such as passed, failed, skipped, or unavailable. Compilation alone is not proof of circuit correctness. The user should inspect the component list, circuit plan, code, and validation report and explicitly approve before any deployment/flash action. (6) Deployment and feedback: only after approval should the system provide deployment instructions; test observations and corrections should be stored against the project version. Backend, AI-agent execution, generated circuit/firmware, and deployment are not verified in the supplied prototype evidence.

Current Prototype Status

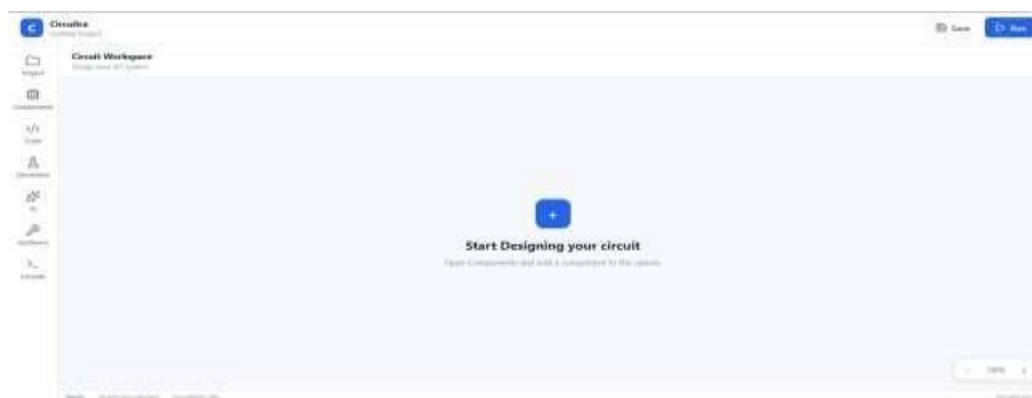


Fig. 2. Current Circuitra frontend workspace; board not selected and simulation idle.

The supplied Circuitra screenshot documents the frontend navigation/workspace only: Project, Components, Code, Simulation, AI, Hardware, and Console are visible, while the circuit canvas is empty, no board is selected, and simulation is idle. This screenshot does not establish that AI agents, backend services, automatic circuit generation, firmware generation, compilation, simulation, or hardware deployment are working.

Accordingly, this manuscript labels those functions as proposed or planned. The implementation status should be updated only when reproducible evidence—such as an actual generated circuit/code, compiler log, simulation output, or physical test record—is available.

Planned Evaluation

After implementation, evaluation should use a pre-defined set of ESP32 tasks covering sensor input, actuator control, and multi-component behaviour. For each task, an evaluator should prepare expected requirements, components, GPIO/interface mapping, and observable behaviour before running the platform. Record the exact user prompt, board model, libraries/tool versions, generated artefacts, manual changes, and whether a physical test was performed.

Report requirement coverage, component/pin-plan correctness, compile success (among attempts actually compiled), issues caught by validation, manual corrections, task completion, and time-to-result. Separate simulated, compiled, and physical outcomes; mark unavailable hardware tests as “not tested.” Include failures and clarification requests, not only successful examples. No numerical performance results are claimed in this design-stage paper.

Illustrative Evaluation Cases (Planned)

Case A—Environmental monitoring: request a prototype that reads temperature and humidity and displays values on the serial console. The evaluator should check whether the platform asks for missing sensor/board details, identifies a compatible sensor interface, proposes board-specific pins with a datasheet source, and prepares firmware with library dependencies. A successful compilation and a physical sensor-reading check must be recorded separately; if hardware is unavailable, the physical outcome is “not tested.”

Case B—Threshold indicator: request an LED that turns on when a sensor reading exceeds a user-defined threshold. The evaluator should check that the threshold and desired behaviour are clarified, the output circuit includes suitable current limiting, the GPIO choice is reviewed for the selected board, and the generated code uses the confirmed pin and threshold. Deployment should remain blocked until user approval. These cases are proposed test examples, not completed runs or measured results.

Expected Outcomes and Limitations

The expected outcome is a guided interface that helps users move from a plain-language IoT idea toward a structured prototype plan and firmware. Potential benefits include a clearer development workflow, less manual switching between tools, and more visible validation steps. These benefits remain expected until measured experimentally.

The platform may still produce incorrect or incomplete requirements, pin mappings, or code. Results depend on the quality and freshness of technical references, model behaviour, board-specific constraints, and availability of development tools. Electrical safety and hardware compatibility must be considered. Human approval and physical testing remain necessary, and a successful compilation must not be presented as proof of correct hardware operation.

System Requirements and Proposed Workflow

Functional requirements (proposed): FR1—accept a natural-language IoT description; FR2—extract goals, inputs, outputs, and constraints and ask for missing details; FR3—propose compatible components and ESP32 GPIO/interface assignments with supporting references; FR4—prepare firmware aligned with confirmed hardware; FR5—display validation results and diagnostics; FR6—require explicit user approval before deployment and provide deployment instructions. These are requirements from the project design/SRS, not verified completed features.

Table 2. Proposed functional requirements and planned acceptance checks (not yet executed).

ID	Proposed requirement	Planned acceptance check
FR1	Accept an IoT idea in natural language	Confirm the original user request is retained and parsed into a project record
FR2	Extract goals/constraints and ask for missing details	Compare extracted fields with a task specification; verify a clarification is asked for ambiguous inputs
FR3	Recommend components and ESP32 GPIO/interfaces	Review component and pin proposals against board documentation and datasheets; record incompatibilities
FR4	Prepare firmware aligned with approved hardware	Check board, pins, libraries, and required behaviour; compile only when toolchain is configured
FR5	Show validation status and diagnostics	Check that each status names the actual test/tool and never marks an unrun check as passed
FR6	Require user approval before deployment	Verify deployment/flash action is blocked until an explicit approval is recorded

Non-functional requirements (proposed): the interface should be understandable to beginners, preserve project versions and traceable assumptions, protect credentials and project data, present clear error/validation status, and avoid silently treating uncertain hardware choices as verified. The workflow is: user

request → requirement clarification → context record → component/GPIO plan → user review → firmware preparation → available validation → explicit approval → deployment guidance. A component or requirement change should trigger review of dependent pin mappings, libraries, and code.

Conclusion

This paper defines a proposed agentic workflow for guiding IoT prototype development from natural-language requirements through ESP32-oriented hardware planning, firmware preparation, validation, user approval, and deployment guidance. Its intended contribution is the integration of these reviewable stages in a beginner-oriented interface. The current evidence is limited to a frontend workspace screenshot; backend execution, generated circuits/firmware, and experimental performance have not been verified. Future research should implement the workflow, validate hardware compatibility and security, and evaluate it with reproducible tasks and transparent reporting of successes and failures.

REFERENCES

1. Y. Peng, Y. Gao, et al., “Programming Embedded IoT Applications in Natural Language with IoTPIlot,” in Proc. 23rd Annual International Conference on Mobile Systems, Applications and Services (MobiSys ’25), pp. 70–82, 2025, doi: 10.1145/3711875.3729136.
2. H. Yang, M. Li, M. Han, Z. Li, and W. Xu, “AutoEmbed: LLM-driven Automated Software Development for Generic Embedded IoT Systems,” in Proc. ACM/IEEE International Conference on Embedded Artificial Intelligence and Sensing Systems (SenSys ’26), pp. 362–376, 2026, doi: 10.1145/3774906.3800466.
3. Q. Luo, R. Ma, X. Zhang, and L. Qiu, “IoTGen: Towards LLM-Driven IoT Hardware PCB Generation,” in Proc. 24th Annual International Conference on Mobile Systems, Applications and Services (MobiSys ’26), 2026, doi: 10.1145/3745756.3809239.
4. Y. Li, Y. Cheng, M. Ma, Y. Zou, N. Yang, W. Cheng, H. Li, Y. Chen, and T. Chen, “Skilled AI Agents for Embedded and IoT Systems Development,” in Proc. ACM Conference on AI and Agentic Systems (CAIS ’26), pp. 1248–1252, 2026, doi: 10.1145/3786335.3813205.
5. S. Paria, A. Dasgupta, and S. Bhunia, “SPELL: An End-to-End Tool Flow for LLM-Guided Secure SoC Design for Embedded Systems,” IEEE Embedded Systems Letters, vol. 16, no. 4, pp. 365–368, 2024, doi: 10.1109/LES.2024.3447691.
6. Z. Wang, J. Chen, Z. Wei, L. Lin, and G. Tong, “SafeCodeRL: Security-Constrained Multi-Agent Reinforcement Learning for Trustworthy LLM-Generated IoT/CPS Software,” Sensors, vol. 26, no. 11, art. 3502, 2026, doi: 10.3390/s26113502.