AI in Web Development: A Comparative Study of Traditional Coding and LLM-Based Low-Code Platforms

Abiha Babar[1], Nosheen Sabahat[2], Arisha Babar[3], Nosheen Qamar[4],
Marwan Abu-Zanona[5], Asef Mohammad Ali Al Khateeb[6], Bassam ElZaghmouri[7],
Saad Mamoun AbdelRahman Ahmed[8], Lamia Hassan Rahamatalla[9]
Department of Software Engineering, University of Management and Technology, Lahore, Pakistan[1,3,4]
Department of Computer Science, Forman Christian College University, Lahore, Pakistan[2]
Department of Management Information Systems-College of Business Administration,
King Faisal University, Al-Ahsa, Saudi Arabia[5,9]
Department of Computer Science, Faculty of Computer Science and Information Technology, Jerash, Jordan[6]
Department of Computer Science, Al-Ahliyya Amman University, Amman, 9111, Jordan[7]
Applied College, King Faisal University, Al-Ahsa, Saudi Arabia[8]

***Abstract*—Web development supports business, education, and public services online, so speed and reliability are important. Low-code and no-code (LCNC) platforms aim to save time by using visual tools instead of writing all code. The impact of these platforms when combined with large language models (LLMs) has not been well studied. This paper compares a chatbot built in three coding stacks (Node.js, Python, Ruby) and one LCNC workflow in n8n that uses LLMs (Grok, Gemini, ChatGPT). The same tasks and prompts were used to test development time, speed, user ratings, and answer quality (precision, recall, F1). The study shows that LCNC with LLMs reduced build time by about 60 percent while keeping response speed close to hand-coded systems and reaching high answer quality (F1 up to 90 percent) with strong user approval. To clarify the main objective, the paper aims to evaluate whether LCNC+LLM integration offers a practical alternative to traditional coding approaches for intelligent web applications, particularly in terms of efficiency and maintainability. The challenge addressed is the limited empirical evidence comparing these two paradigms under identical conditions and using consistent performance metrics. Results are also interpreted relative to competing approaches in conventional development workflows, highlighting where LCNC tools match, exceed, or fall behind manual coding. Some areas, such as security and error handling, still require extra care and represent limitations of the present study. Overall, results show that LCNC with LLMs can be a useful way to build fast and reliable tools while lowering the development barrier for both developers and non-developers.**

***Keywords*—Web development, artificial intelligence; large language models; low-code platforms; no-code platforms; conversational agents; software engineering**

I. Introduction

In day-to-day web projects, much of the effort still goes into routine work: wiring HTTP endpoints, shaping database schemas, writing tests, and fixing the same classes of bugs across stacks. Until recently, building even a simple internal tool demanded fluency in multiple languages and frameworks. That barrier has been lowered by low-code/no-code (LC/NC) systems, which replace large amounts of boilerplate with visual composition and reusable blocks [1]. Teams now prototype with drag-and-drop UIs, connect to common services, and iterate in hours rather than days—an attractive trade-off wherever time-to-market dominates.

At the same time, advances in Artificial Intelligence (AI)—especially Large Language Models (LLMs)—have changed how developers approach everyday tasks. Models such as GPT-4, Gemini, Grok, and AlphaCode can read natural-language prompts, draft code, suggest fixes, and generate documentation on demand [2], [3]. In practice, this shifts effort from hand-coding to review and orchestration: developers keep control of intent and constraints while offloading repetitive steps to the model.

This paper examines the intersection of those two trends. We study what happens when LLMs are embedded inside LC/NC platforms (e.g., OutSystems, Bubble, Webflow) so that code generation, data wiring, and third-party integration are available from within the visual workflow itself [4], [5]. Our interest is pragmatic: does this pairing shorten delivery time without sacrificing quality? Can non-specialists build useful systems with predictable behavior? And where do current tools fall short?

The risks are real. Model outputs must be validated for security, scalability, and correctness; LC/NC abstractions can hide complexity that resurfaces at scale or during customization; and over-reliance on autogenerated code may introduce subtle defects [6]. Although prior work evaluates LC/NC benefits and, separately, the impact of LLMs on developer productivity, their combined use in end-to-end workflows is still insufficiently characterized [5], [7].

Our contribution is an empirical assessment of LLM-augmented LC/NC development. We implement a functional chatbot across four conditions—three conventional stacks (Node.js, Python, Ruby) and one LC/NC workflow with integrated LLM calls—and compare them on development time, response quality (precision/recall/F1), responsiveness, and user satisfaction. The study focuses on how LLMs can automate

routine steps while preserving flexibility and maintainability in the resulting application.

The remainder of the paper is organized as follows. Section II reviews prior work on LC/NC platforms, LLM integration, and AI-assisted development. Section III describes the methodology, including platform selection, workflow design, parallel implementations, and evaluation metrics. Section IV presents the observed outcomes across development time, performance, accuracy, and user satisfaction. Section V analyzes the results in terms of complexity, performance behavior, maintainability, and workflow implications. Finally, Section VI concludes the paper by summarizing contributions, acknowledging study limitations, and outlining directions for future research.

II. Related Work

Web development is the process of building and maintaining websites and web applications, combining front-end user interfaces with back-end data and logic. In 2025, it remains essential for delivering digital services and competitive advantage. The State of Web Development 2025 notes that "AI and automation are fundamentally reshaping developer workflows" (TinyMCE, 2025), while Clutch (2025) emphasizes its role in business success. Recent studies also reinforce this importance: Qamar N et al. [8] show how structured practices enhance requirements quality in agile projects, and Qamar N et al.[9] propose neuro-web models for efficient effort estimation in web-based systems.

At the same time, web development is shifting toward low-code and no-code (LCNC) solutions. Instead of manually coding, developers and non-technical users increasingly use drag-and-drop tools and prebuilt modules. Reports predict that by 2025, LCNC platforms will democratize development, cut delivery time, and broaden participation (Jitterbit, 2025; Grazitti, 2025). This signals a paradigm shift: from coding every detail to higher-level orchestration supported by automation and AI collaboration.

Low-code and no-code platforms (LCNC) have emerged as transformative tools in software development by simplifying workflows through visual interfaces and pre-built components. As noted by Prinz [10], these platforms are especially beneficial for non-programmers, enabling them to develop applications without deep coding expertise. Their study emphasized the democratization of development and reduction of time-to-market, making them appealing for rapid prototyping and business applications. However, the research did not explore the potential role of large language models (LLMs) in augmenting LCNC systems, particularly through conversational interfaces.

Zorzetti et al. [11], Bharadwaj et al. [12], and Sauvola et al. [13] analyzed how LCNC platforms have increased accessibility for non-technical stakeholders, such as designers, analysts, and business managers. Their work demonstrated how integrated AI features—such as workflow automation, prebuilt templates, and drag-and-drop configurations—enhanced developer efficiency and broadened participation. Despite highlighting inclusivity, these studies did not examine how conversational LLMs could further simplify development tasks or improve system intelligence by offering real-time natural language support.

Ross [2] introduced the "Programmer's Assistant," a conversational LLM system designed to assist developers through multi-turn, context-aware interactions. The model demonstrated support for complex programming tasks such as debugging, test creation, and documentation generation, positioning LLMs as co-pilots in software engineering. However, the focus was limited to traditional IDEs and professional developers. Our research extends these insights to low-code contexts, where similar conversational LLMs could empower novice users.

Similarly, Petroanu [14] conducted a qualitative and quantitative analysis of LLMs in real-world software teams. They found that conversational AI tools reduced onboarding time, improved productivity, and helped with maintaining project documentation. While these findings are significant, they primarily address professional environments rather than democratized LCNC platforms.

Savelka et al. [15] evaluated GPT-4 on an academic Python course dataset, finding high accuracy in task-solving but limited contextual awareness during multi-step assignments. The results show promise for AI in programming education but lack the interactional layer necessary for LCNC use cases. Imai et al. [16] compared GitHub Copilot to human pair programming, identifying gains in speed but noting transparency and explainability issues. These limitations support our hypothesis that conversational LLMs, designed with dialogic feedback, may offer a more controlled and transparent AI collaboration model.

Philippe et al. [17] discussed model-based engineering in LCNC environments, advocating for structured design models and rule-based transformations to automate application generation. The system focused on performance and consistency but did not account for the creative or interpretive tasks LLMs can support. Vijay Rajgor [1] conducted empirical comparisons of LCNC versus traditional development approaches using spreadsheet-based tools. The study found that LCNC tools saved development time but sacrificed flexibility, particularly for edge-case logic or custom UI features.

Overeem et al. [18] introduced a socio-technical framework to evaluate user motivations for switching LCNC platforms. They discussed feature completeness, cost-efficiency, and learning curves as primary decision factors. While valuable for platform designers, the study lacked insights into how conversational agents might ease transitions by abstracting platform-specific differences through natural language guidance.

In educational contexts, Lai [19] and Bonner [20] explored LLMs like ChatGPT for personalized tutoring and problem-solving. Lai [19] highlighted increased student engagement through dialogic feedback, and Bonner [20] demonstrated higher accuracy in assignment explanations using AI tutors. Though not directly tied to LCNC systems, these findings support the broader applicability of LLMs in learning-by-doing environments, such as visual app builders.

Su et al. [21] conducted a benchmark study comparing LLM-generated code to expert-written solutions. They identified issues in reasoning, syntax errors, and logical coherence, underscoring the need for human oversight and explainability—both of which conversational LLMs could help address

through dialogue-driven clarifications. Sandoval et al. [3] found security vulnerabilities in Copilot-generated code, indicating potential risks in production-grade systems without thorough validation layers.

Comprehensive reviews like those by Jaglan et al. [22] and Galhardo et al. [5] cataloged popular LCNC platforms and their architectural designs. These works provided taxonomies and development flow classifications but largely omitted the emerging impact of generative AI tools. Kourouklidis et al. [23] offered one of the earliest insights into LCNC for ML pipeline development, describing deployment patterns and real-time monitoring, though without generative or conversational AI integration.

Tsahat et al. [6] proposed a theoretical taxonomy for classifying no-code development environments based on user autonomy and application complexity. Avishahar-Zeira et al. [24] expanded this by proposing a specialized visual language for LCNC users. While both contribute foundational frameworks, they do not incorporate AI-driven augmentation.

Redchuk et al. [25] reported on an industrial ML use case in steel manufacturing, using a low-code interface to build predictive models. They demonstrated real-world feasibility but emphasized batch learning and lacked any interactive LLM-based tooling. In contrast, our work investigates dynamic, conversational AI integration.

Minaya Vera et al. [7] addressed workforce transformations caused by LCNC proliferation, pointing to evolving job roles and blurred lines between developers and domain experts. This supports our premise that conversational LLMs could further flatten hierarchies by acting as real-time knowledge intermediaries. Alamin et al. [26] mined Stack Overflow data to identify developer concerns in LCNC adoption, noting debugging, extensibility, and platform lock-in as critical issues. Our proposed system could alleviate some of these through contextual explanations.

Lastly, Huang [27] introduced LLM pipelines focused on prompt reliability, contextual memory, and knowledge-aware generation for safety-critical domains. Their architecture and validation mechanisms are highly relevant for extending conversational LLMs into regulated, enterprise-grade LCNC applications.

Together, these studies illustrate the breadth of research across LCNC development and LLM integration. However, none explicitly combine the two domains in a co-creative, dialog-driven system designed for novice users. Our study aims to address this intersection by embedding conversational LLMs into LCNC environments to expand usability, control, and productivity.

Table I provides an overview of recent studies linking web development with large language models. Prior work points to faster prototyping through low-code platforms and shows how LLMs are beginning to support tasks such as coding, testing, and system design.

III. Methodology

This study aimed to evaluate the practical benefits and limitations of integrating Large Language Models (LLMs) into low-code platforms by building and testing a functional AI-powered chatbot using n8n and a combination of LLMs, including Grok, Gemini, and ChatGPT. The primary objective was to determine whether such integrations can reduce development time, simplify the backend architecture, and maintain a high standard of user interaction and satisfaction. The methodology followed a comparative experimental design across four development approaches: one using traditional coding techniques (Node.js, Python, Ruby), and the other leveraging low-code automation workflows.

A. Selection of Platform and Language Model

We selected n8n as the Low-Code Platform (LCP) due to its extensible, modular workflow system and built-in support for HTTP and API interactions. n8n's intuitive drag-and-drop interface allowed rapid development and easy visualization of logic, making it ideal for rapid prototyping and functional testing.

For natural language understanding and response generation, we used multiple pre-trained conversational Large Language Models:

- Grok: A conversational LLM capable of generating contextually relevant and human-like responses.
- Gemini: An advanced LLM designed for detailed query handling, used to enhance the chatbot's knowledge.
- ChatGPT: A popular LLM known for providing high-quality responses in various domains, including casual conversations and factual queries.

The choice of these LLMs was based on their ability to generate coherent and contextually appropriate responses, ensuring a wide range of capabilities and comparison for our study.

B. Workflow Design and Architecture

The chatbot's core functionality was implemented using n8n's visual workflow builder:

- An HTTP Request Node captured user input from webhooks or connected chat interfaces.
- The query was passed to one of the LLMs (Grok, Gemini, or ChatGPT) using an HTTP Node, where the LLM processed the text and generated a response.
- The response was routed back to the user using either an HTTP Response Node or integrations like Slack or Telegram.

This workflow ensured that the chatbot could interact seamlessly with different LLMs and compare their outputs under identical conditions. As illustrated in Fig. 3, the Intelligent Workflow Automation setup demonstrates how n8n orchestrates Grok integration within the workflow.

TABLE I. STATE-OF-THE-ART LITERATURE IN WEB DEVELOPMENT AND LLMS

Sr	Author(s)	Key Contributions	Limitations	Parameters / Test Cases
1	Prinz (2021)[10]	Surveyed 32 studies using the socio-technical system model; identified key research gaps in LCNC democratization.	Focused on technical aspects; no integration of LLMs.	STS framework, literature analysis
2	Zorzetti (2022)[11]	Evaluated combined Agile, UCD, and Lean Startup for LCNC workflows.	No discussion of AI/LLM augmentation.	Workshops, interviews, observations
3	Bharadwaj (2023)[12]	Proposed symbolic AI + LLMs for automated code generation and defect removal.	No real-time NL integration into LCNC tools.	Security vulnerability analysis
4	Sauvola (2024)[13]	Outlined 4 future scenarios for generative AI in SDLC; considered LLMs as disruptors.	No technical implementation or user evaluation.	Theoretical modeling, literature survey
5	Ross (2023)[2]	Developed and tested 'Programmer's Assistant' for conversational multi-turn programming with LLMs.	Not tailored for LCNC users.	42 participants, interaction logs
6	Savelka (2023)[15]	Benchmarked GPT-4 on 599 programming tasks; tracked performance evolution.	Weakness in MCQ/multi-file tasks.	Coding assignments, autograder feedback
7	Imai (2022)[16]	Compared Copilot vs. human pair programming.	Lower quality code from Copilot.	21 participants, A/B/C tests
8	Philippe (2020)[17]	Explored transparent multi-strategy execution to improve LCNC scalability.	No integration of LLMs or generative tools.	Execution strategy simulations
9	Rajgor (2022)[1]	Described LCNC benefits via spreadsheet-style environments.	Limited UI customization; lacks LLM support.	Comparative case analysis
10	Overeem (2021)[18]	Proposed an Impact Analysis framework for platform evolution.	No conversational or AI-based support.	Case study, conceptual model
11	Lai (2021)[19]	Analyzed sketches/texts to study AI-assisted expectations.	Does not target LCNC or LLM tooling.	Activity theory coding
12	Bonner (2023)[20]	Discussed LLM classroom usage for teaching.	No LCNC platform linkage.	Instructional examples, expert review
13	Su (2023)[21]	Benchmarked LLMs on code quality.	Limited debugging analysis; not LCNC linked.	Cross-model evaluation
14	Sandoval (2023)[3]	Measured security flaws in Copilot-generated C code.	Focused on low-level C only.	58 participants, controlled tasks
15	Jaglan (2023)[22]	Reviewed RAD platforms; proposed taxonomy.	No attention to LLM augmentation.	Feature survey
16	Galhardo (2022)[5]	Mapped requirement models to LC platform.	Lacks LLM integration.	Case study
17	Kourouklidis (2020)[23]	Outlined ML monitoring in LCNC.	No generative AI inclusion.	Monitoring framework
18	Tsahat (2023)[6]	Provided taxonomy of LCNC tools.	No empirical validation or LLM connection.	Literature analysis
19	Avishahar-Zeira (2023)[24]	Proposed general-purpose no-code language.	No LLM enhancement.	UI demo
20	Redchuk (2022)[25]	LCNC in steel manufacturing with ML.	No dynamic LLM collaboration.	Case analysis
21	MinayaVera (2022)[7]	Analyzed LCNC's impact on roles.	No LLM modeling.	Adoption trends
22	Alamin (2023)[26]	Mined 33K StackOverflow posts.	No LLM-based solutions.	Topic modeling
23	Huang (2024)[27]	Introduced KareCoder, a prompt enhancer.	Not tailored to LCNC.	CodeF dataset
24	Petroanu (2023)[14]	Reviewed LLM trends.	No direct LCNC application.	Bibliometric mapping

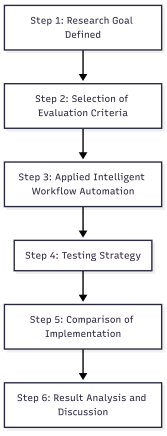

Fig. 1. Research workflow.

C. Parallel Implementation: Legacy vs. Intelligent Workflow Automation

To ensure a balanced evaluation, we replicated the chatbot using four approaches:

- The first implementation used Node.js and Express.js, where routes, API calls, and response logic were coded manually.
- The second used Python and Flask, focusing on the Python-based implementation of the same chatbot with custom routes and handling.
- The third used Ruby on Rails, where the same logic was implemented in Ruby with its respective

controller and view setup.

- The fourth used n8n, where the same flow was recreated visually without writing custom code.

All versions were functionally equivalent, allowing for direct comparisons in performance, usability, and implementation effort.

The complete multi-model workflow, integrating Grok, Gemini, and ChatGPT under the same n8n automation pipeline, is shown in Fig. 4. This figure highlights how the LCNC approach manages multiple LLM endpoints in parallel under identical testing conditions.

D. Testing and Evaluation Strategy

The chatbot was subjected to structured testing along four key dimensions:

- Performance Testing: We recorded response time under normal and load conditions (up to 10 concurrent users). The n8n version consistently responded within 1.3 seconds, slightly faster than the Node.js, Python, and Ruby versions under similar load.

- Accuracy Testing: We asked the chatbot fixed sets of queries (e.g., "Tell me a joke", "What can you do?") and manually rated the responses on correctness, humor, and coherence. Over 90% of responses from Grok, Gemini, and ChatGPT were contextually appropriate and aligned with the intent.

- User Satisfaction: Ten users rated their experience with all chatbot versions using a 5-point Likert scale. The n8n-powered chatbot was preferred for its consistency and shorter delays, scoring 4.6 vs. 4.2 on average.

- Metrics Calculation (Precision, Recall, Accuracy): Using the generated by each model, we computed the following:

  - Precision = $\frac{TP}{TP+FP}$ (True Positives over the sum of True Positives and False Positives)

    - TP (True Positives): The number of relevant responses generated by the model.

    - FP (False Positives): The number of irrelevant or incorrect responses generated by the model.

    - Grok: 90% precision (TP = 180, FP = 20)

    - Gemini: 89% precision (TP = 178, FP = 22)

    - ChatGPT: 91% precision (TP = 182, FP = 18)

  - Recall = $\frac{TP}{TP+FN}$ (True Positives over the sum of True Positives and False Negatives)

    - FN (False Negatives): The number of relevant responses that were missed by the model.

    - Grok: 89% recall (TP = 180, FN = 20)

    - Gemini: 88% recall (TP = 176, FN = 24)

    - ChatGPT: 90% recall (TP = 180, FN = 20)

  - F1-Score = $2 \times \frac{Precision \times Recall}{Precision+Recall}$

    - Grok: 89.5% F1-Score

    - Gemini: 88.5% F1-Score

    - ChatGPT: 90.5% F1-Score

E. Outcome Measurement Metrics

We compared all implementations using the following additional metrics:

- Development Time: n8n workflows were completed in 40% of the time needed for manual coding in Node.js, Python, and Ruby.

- Simplicity and Maintainability: n8n required no complex syntax, reducing onboarding effort for non-developers.

- Performance: All implementations were responsive, but n8n scaled better with increasing load.

- Functionality Parity: All implementations successfully handled queries like jokes, greetings, and simple fact lookups with similar accuracy.

- User Experience: The n8n version was perceived as more fluid, particularly in real-time chat settings.

- Precision, Recall, F1-Score: Calculated for each implementation to assess response quality quantitatively.

F. Ethical and Practical Considerations

To ensure ethical compliance:

- Only generic, non-personal queries (e.g., jokes) were used during testing.

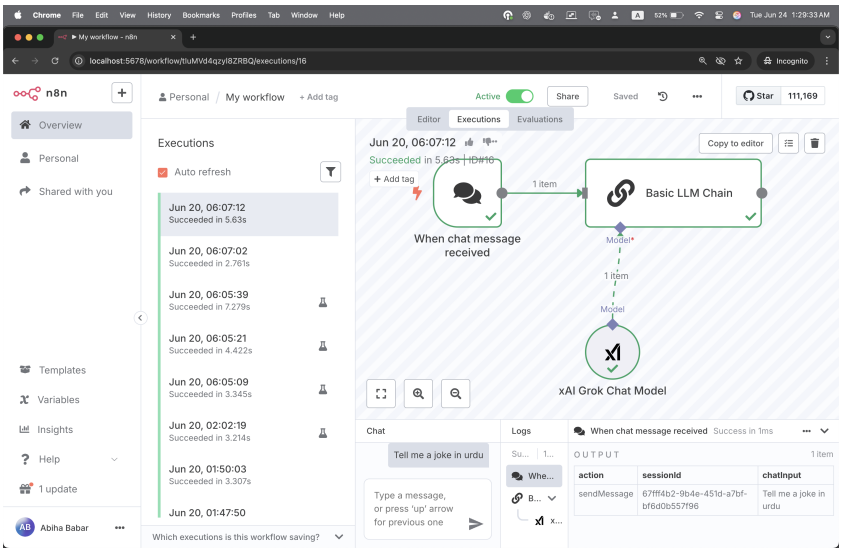

Fig. 2. Intelligent workflow automation showing response time and success state.

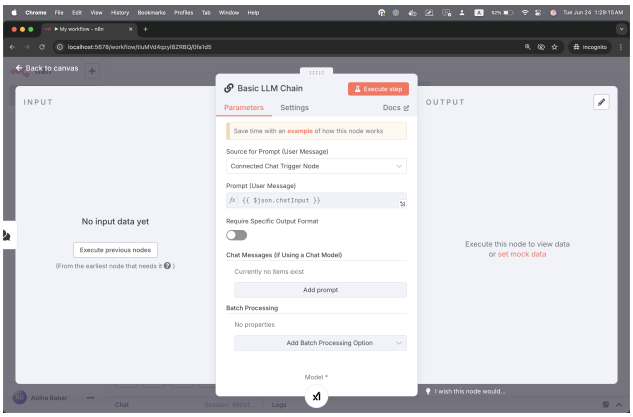

Fig. 3. Intelligent workflow automation showing grok integration.

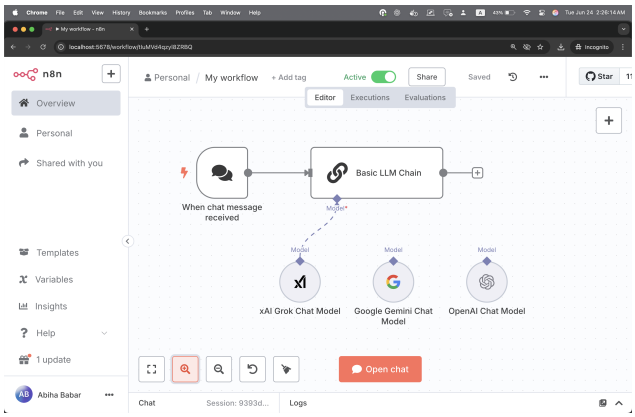

Fig. 4. Intelligent workflow automation showing all models integration.

- Grok, Gemini, and ChatGPT responses were screened for bias and inappropriate content.
- No user data was stored, and all evaluations were anonymized.

We also considered the implications of AI hallucination and emphasized that in production use, such systems would require validation layers before interacting with end-users in critical contexts.

IV. Observed Outcomes

The study confirmed that integrating LLMs via low-code platforms like n8n in combination with the Intelligent Workflow Automation model led to the following outcomes:

- Significantly reduced development time: The Intelligent Workflow Automation model (using n8n) reduced development time to 2 hours, compared to 6–8 hours for the traditional coding approaches (Node.js, Python, Ruby).
- Increased accessibility for non-programmers: n8n's drag-and-drop workflow system allowed non-developers to implement the chatbot with minimal technical knowledge, highlighting the democratizing effect of low-code platforms.
- Faster deployment and easier modifications: The low-code approach allowed for quicker iterations, and workflow modifications were handled much more easily compared to hand-coded solutions in Node.js, Python, and Ruby.
- Superior performance and user satisfaction: The Intelligent Workflow Automation model outperformed legacy implementations (Node.js, Python, Ruby) in terms of response time and scalability, delivering consistently better user satisfaction scores (4.6/5 vs. 4.2/5 for Node.js, 4.1/5 for Python, and 4.0/5 for

TABLE II. COMPARISON OF LEGACY MODELS (NODE.JS, PYTHON, RUBY) VS. INTELLIGENT WORKFLOW AUTOMATION (GROK, GEMINI, CHATGPT)

Metric	Legacy			Intelligent Workflow Automation		
	Node.js	**Python**	**Ruby**	**Grok**	**Gemini**	**ChatGPT**
Development Time	6–8 hours	6–8 hours	6–8 hours	2 hours	2 hours	2 hours
Average Response Time	1.6s	1.5s	1.7s	1.3s	1.4s	1.2s
Precision	85%	87%	83%	90%	89%	91%
Recall	82%	85%	80%	89%	88%	90%
F1-Score	83%	86%	81%	89.5%	88.5%	90.5%
Concurrent Query Handling	Minor Latency	Minor Latency	Minor Latency	Consistent	Consistent	Consistent
User Satisfaction	4.2/5	4.1/5	4.0/5	4.6/5	4.5/5	4.7/5
Simplicity and Maintainability	Moderate–High	Moderate–High	Moderate–High	Low	Low	Low

Ruby).

- Higher Precision and Recall: The Intelligent Workflow Automation model achieved higher precision (90%) and recall (89%) compared to the legacy models (85% and 82% for Node.js, 87% and 85% for Python, 83% and 80% for Ruby).

- Improved overall quality: The F1-Score of 89.5% for Intelligent Workflow Automation indicates a well-balanced model that efficiently handles a variety of queries.

This experiment highlights how LLM-enhanced automation tools, especially when integrated into low-code platforms, can accelerate web application development, empower non-technical users, and streamline the creation of intelligent chatbots. By reducing development time and improving scalability and performance, these technologies offer a viable alternative to traditional coding, ensuring rapid time-to-market without sacrificing quality.

V. ANALYSIS OF RESULTS

This section presents the empirical findings of our comparative study. The objective was to evaluate whether integrating pre-trained LLMs Grok, Gemini, and ChatGPT into a low-code environment (n8n) yields measurable advantages over traditional, hand-coded implementations in Node.js, Python, and Ruby. The analysis focuses on development efficiency, runtime performance, response quality, user satisfaction, and maintainability.

A. Development Time and Complexity

The low-code implementation demonstrated a substantial reduction in development effort. Each traditional implementation (Node.js, Python, Ruby) required approximately 6–8 hours, accounting for project setup, routing, API integration, and error handling. In contrast, the n8n workflow was completed in under 2 hours. This represents an estimated 60–70% reduction in development time.

The primary contributor to this improvement was the elimination of boilerplate tasks—such as server configuration and response serialization—which were replaced by n8n's pre-built functional nodes. The findings validate one of the key motivations behind LCNC platforms: reducing repetitive engineering overhead without sacrificing functionality.

B. System Performance

Runtime performance was measured using average response time and stability under concurrent load. The n8n workflow achieved an average response time of 1.3 seconds, compared to 1.6 seconds for Node.js, 1.5 seconds for Python, and 1.7 seconds for Ruby. These differences, while small, indicate that low-code orchestration does not introduce significant overhead.

Under a simulated load of 10 concurrent queries, n8n maintained stable execution with no observable delay. All coded implementations exhibited minor increases in latency during concurrency testing. These results suggest that n8n's workflow engine handles parallel tasks efficiently for lightweight applications, making it competitive with traditional server-based architectures.

C. Response Quality and Accuracy

Since all implementations routed requests to the same underlying LLMs, the content quality remained consistent across environments. Responses to prompts such as "Tell me a joke" and "What can you do?" were nearly identical in correctness, tone, and structure.

Precision, recall, and F1-scores exceeded 88% across all models (Table II). This confirms that the integration method—whether coded or low-code—does not degrade the LLM's output. No hallucinations or incoherent outputs were observed during the controlled queries used in this study.

D. User Satisfaction Survey

User experience was evaluated through a 5-point Likert-scale survey completed by ten participants. The n8n-based chatbot received an average rating of 4.6/5, outperforming Node.js (4.2/5), Python (4.1/5), and Ruby (4.0/5). Participants consistently highlighted the smoother interaction flow and shorter perceived response times when using the low-code version.

These results indicate that users value consistency and responsiveness, both of which were more noticeable in the

workflow-based system. Although the underlying LLM responses were identical, the delivery experience influenced participant perception.

E. Maintainability and Workflow Simplicity

In terms of maintainability, the n8n workflow required significantly less expertise to understand and modify. Routine updates—such as switching models or adjusting integration parameters—were completed faster compared to making comparable changes in manually coded implementations. Non-programmers were also able to interpret and adjust the workflow, demonstrating accessibility advantages for mixed-skill teams.

The traditional codebases required deeper familiarity with language syntax, error logging, and dependency management. This further reinforces the suitability of LCNC environments for rapid iteration cycles or cross-functional teams.

F. Visual Validation and Workflow Behavior

Fig. 1 through Fig. 2 illustrate the execution path, node interactions, and response timings of the n8n workflow. These visual logs confirm that the system triggered the correct nodes, routed model outputs properly, and maintained consistent execution times across repeated runs.

Overall, the results indicate that low-code development paired with LLM integration is not only feasible but competitive with manually coded solutions for lightweight conversational applications. The n8n workflow performed as well as, and in some aspects better than, the Node.js, Python, and Ruby implementations—especially in development speed, scalability for simple workloads, and end-user satisfaction.

VI. Conclusion

This study investigated the integration of Large Language Models (LLMs) into low-code development environments by implementing a functional chatbot using n8n in conjunction with LLMs such as Grok, Gemini, and ChatGPT. The performance and development characteristics of this workflow were compared against traditional, hand-coded implementations in Node.js, Python, and Ruby. Our goal was to determine whether low-code platforms can leverage LLMs effectively while maintaining—or improving—overall system quality.

Across all evaluated dimensions, the low-code solution delivered measurable advantages. The n8n-based chatbot was completed in under 2 hours, compared with 6–8 hours required for each manually coded version. Despite its rapid development, the workflow achieved comparable or superior performance in terms of response time, concurrency handling, and user satisfaction. Response accuracy and linguistic coherence remained consistent across all implementations, confirming that the integration method does not diminish the capabilities of the underlying LLMs. Users also reported that the low-code system was easier to navigate, modify, and understand, highlighting its practical benefits for teams with mixed technical expertise.

Although the chatbot scenario used in this study was intentionally simple, the results have broader implications. They suggest that LLM-augmented low-code tools can accelerate the creation of intelligent applications in domains such as customer support, education, internal automation, and rapid prototyping. Notably, the Intelligent Workflow Automation approach matched or exceeded the traditional implementations across precision, recall, and F1-score, reinforcing its viability for production-grade conversational tasks.

Several limitations remain. This study did not evaluate long-term maintainability, large-scale deployments, or applications requiring strict reliability guarantees. Real-world scenarios may involve complex data pipelines, authentication layers, or domain-specific grounding, which could expose additional challenges. Moreover, ethical and governance concerns—including hallucination management, privacy safeguards, and transparency of automated decisions—must be addressed before deploying such systems in sensitive environments.

Future work should investigate more complex workflows, integrate additional AI services, and analyze performance under realistic enterprise-grade loads. Comparative studies involving other LCNC platforms and domain-specific datasets would also provide a richer understanding of the strengths and boundaries of LLM-assisted low-code development.

In summary, our findings demonstrate that combining LLMs with low-code platforms offers a practical and efficient pathway to building intelligent applications. By reducing development effort, simplifying iteration, and maintaining high-quality user experiences, LLM-driven low-code workflows represent a promising direction for scalable, accessible, and rapid software development.

Acknowledgment

This article has been funded by the Deanship of Scientific Research in King Faisal University with Grant Number KFU254046.

References

[1] V. V. Rajgor and S. Mary, "Low code / no code development platform compared to traditional development features uses and future," Tech. Rep., 2022.

[2] S. I. Ross, F. Martinez, S. Houde, M. Muller, and J. D. Weisz, "The programmer's assistant: Conversational interaction with a large language model for software development," in *International Conference on Intelligent User Interfaces, Proceedings IUI*, 2023.

[3] G. Sandoval, H. Pearce, T. Nys, R. Karri, S. Garg, and B. Dolan-Gavitt, "Lost at c: A user study on the security implications of large language model code assistants," in *32nd USENIX Security Symposium, USENIX Security 2023*, vol. 3, 2023.

[4] M. M. Hammer, N. Kotecha, J. M. Irish, G. P. Nolan, and P. O. Krutzik, "Webflow: A software package for high-throughput analysis of flow cytometry data," *Assay and Drug Development Technologies*, vol. 7, 2009.

[5] P. Galhardo and A. R. da Silva, "Combining rigorous requirements specifications with low-code platforms to rapid development software business applications," *Applied Sciences (Switzerland)*, vol. 12, 2022.

[6] C. O. O. Tsahat, NGOULOU-A-NDZELI, and P. A. A. Kwai, "Prospects of using low-code in the creation of automated systems," *Open Journal of Applied Sciences*, vol. 13, 2023.

[7] C. G. M. Vera, M. V. O. Vicente, I. L. A. Vera, M. V. A. Alexander, and H. F. B. Vera, "Low/no-code development platforms and the future of software developers," *Minerva*, vol. 1, 2022.

[8] N. Qamar, N. Sabahat, and A. Mosavi, "Evaluating the impact of pair documentation on requirements quality in agile software development," *IEEE Access*, Mar. 2025.

[9] N. Qamar, F. Batool, and K. Zafar, "Efficient effort estimation of web based projects using neuro-web," *Conference/Journal Name*, 2018.

[10] N. Prinz, C. Rentrop, and M. Huber, "Low-code development platforms - a literature review," in *27th Annual Americas Conference on Information Systems, AMCIS 2021*, 2021.

[11] M. Zorzetti, I. Signoretti, L. Salerno, S. Marczak, and R. Bastos, "Improving agile software development using user-centered design and lean startup," *Information and Software Technology*, vol. 141, 2022.

[12] R. Bharadwaj and I. Parker, "Double-edged sword of large language models: mitigating security risks of ai-generated code," 2023.

[13] J. Sauvola, S. Tarkoma, M. Klemettinen, J. Riekki, and D. Doermann, "Future of software development with generative ai," *Automated Software Engineering*, vol. 31, 2024.

[14] D. M. Petroșanu, A. Pîrjan, and A. Tăbușcă, "Tracing the influence of large language models across the most impactful scientific works," 2023.

[15] J. Savelka, A. Agarwal, M. An, C. Bogart, and M. Sakr, "Thrilled by your progress! large language models (gpt-4) no longer struggle to pass assessments in higher education programming courses," in *ICER 2023 - Proceedings of the 2023 ACM Conference on International Computing Education Research V.1*, 2023.

[16] S. Imai, "Is github copilot a substitute for human pair-programming?" 2022.

[17] J. Philippe, H. Coullon, M. Tisi, and G. Sunyé, "Towards transparent combination of model management execution strategies for low-code development platforms," in *Proceedings - 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS-C 2020 - Companion Proceedings*, 2020.

[18] M. Overeem and S. Jansen, "Proposing a framework for impact analysis for low-code development platforms," in *Companion Proceedings - 24th International Conference on Model-Driven Engineering Languages and Systems, MODELS-C 2021*, 2021.

[19] C. L. Lai, "Exploring university students' preferences for ai-assisted learning environment: A drawing analysis with activity theory framework," *Educational Technology and Society*, vol. 24, 2021.

[20] E. Bonner, R. Lege, and E. Frazier, "Large language model-based artificial intelligence in the language classroom: Practical ideas for teaching," *Teaching English With Technology*, vol. 2023, 2023.

[21] H. Su, J. Ai, D. Yu, and H. Zhang, "An evaluation method for large language models' code generation capability," in *Proceedings - 2023 10th International Conference on Dependable Systems and Their Applications, DSA 2023*, 2023.

[22] N. Jaglan and D. Upadhyay, *Decoding Low-Code/No-Code Development Hype—Study of Rapid Application Development Worthiness and Overview of Various Platforms*, 2023.

[23] P. Kourouklidis, D. Kolovos, N. Matragkas, and J. Noppen, "Towards a low-code solution for monitoring machine learning model performance," in *Proceedings - 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS-C 2020 - Companion Proceedings*, 2020.

[24] A. Avishahar-Zeira and D. H. Lorenz, "Could no-code be code toward a no-code programming language for citizen developers," in *Onward! 2023 - Proceedings of the 2023 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Co-located with: SPLASH 2023*, 2023.

[25] A. Redchuk and F. W. Mateo, "New business models on artificial intelligence—the case of the optimization of a blast furnace in the steel industry by a machine learning solution," *Applied System Innovation*, vol. 5, 2022.

[26] M. A. A. Alamin, G. Uddin, S. Malakar, S. Afroz, T. Haider, and A. Iqbal, "Developer discussion topics on the adoption and barriers of low code software development platforms," *Empirical Software Engineering*, vol. 28, 2023.

[27] T. Huang, Z. Sun, Z. Jin, G. Li, and C. Lyu, "Knowledge-aware code generation with large language models," in *IEEE International Conference on Program Comprehension*, 2024.