Indian Journal of Information Sources and Services
ISSN: 2231-6094 (P) Vol.15, No.2, 2025, pp.380-390

DOI: https://doi.org/10.51983/ijiss-2025.IJISS.15.2.46

Low-Code Development Enhancement Integrating Large Language Models for Intelligent Code Assistance in Oracle APEX

Srikanth Reddy Keshireddy[1]

[1]Senior Software Engineer, Keen Info Tek Inc., USA
E-mail: [1]sreek.278@gmail.com
ORCID: [1]https://orcid.org/0009-0007-6482-4438
(Received 22 March 2025; Revised 23 April 2025; Accepted 19 May 2025; Available online 25 June 2025)

Abstract - **The growing use of low-code platforms within the business sphere has spurred the need for real-time, intelligent code help tools that can effectively integrate citizen developers with expert programmers. This research investigates the application of Large Language Models (LLMs) into Oracle APEX with the goal of improving developer productivity and minimizing the cognitive effort required during task execution. We created a novel system designed to operate within the Oracle APEX environment that includes an LLM-powered code suggestion engine. This system is able to respond to natural language queries by providing context-aware code snippets in PL/SQL, JavaScript, and SQL. Furthermore, it dynamically responds to page items, session states, and application metadata in real time.**

Our architecture integrates Oracle APEX's RESTful interfaces with external LLMs through a modular API orchestration layer to enable effortless generation of accurate deployable code. An extensive experimental evaluation with 48 developers from different seniority levels was performed, including tasks from UI sketching to business logic coding. Achieved results indicate an average of 41 percent reduction in completion time and 34 percent reduction in manual code cut editing. Structured logical scenarios achieved over 90 percent syntax accuracy, and users reported high confidence in the system outputs. The analysis also focuses on token consumption, domain-specific error patterns, and code review log feedback loops.

The primary objective of this research study was to illustrate the profound impacts information technologies, particularly LLMs, had on the speed of achieving low-code application development with Oracle APEX. In addition, the study intends to demonstrate the set of strategies proposed suited for the consistent implementation of intelligent assistants that aid in elevating code quality while reducing barriers, enabling enterprise-level application delivery with minimal manual effort preconfigured.

Keywords: **Oracle APEX, Low-Code Development, Large Language Models (LLMs), Intelligent Code Assistance, Prompt-Based Code Generation**

I. INTRODUCTION

1.1 Rise of Intelligent Code Assistance in Enterprise Development

In the development of enterprise applications, low-code platforms have emerged as critical tools for accelerating delivery timelines while democratizing programming and diminishing dependence on large-scale development teams (Shlomov et al., 2024). Oracle APEX (Application Express) is a case in point. It provides declarative interfaces and integrates directly with the Oracle Database, facilitating rapid application development through minimal coding (Gorissen et al., 2024). APEX, despite its strengths in structured workflows and data-bound logic, is still limited by having to manage session state, validation scripts, dynamic behaviour, and a multitude of basic application configurations through developed session-plans, which heavily rely on structured PL/SQL, JavaScript, and SQL wizardry.

The necessity of intelligent code assist features into such frameworks is now indispensable (Onopreychuk, 2018). The implementation of Large Language Models (LLMs), which can understand natural language and output correctly semantically aligned code, presents opportunities (Mukhitdinova et al., 2025; Seker, 2024). The integration of these intelligent code assistants into the depths of low-code platforms like Oracle APEX makes it possible to tackle skill deficiency issues, lower time-to-delivery windows, and aid both seasoned and novice developers tackle complex application exigencies in a simple, streamlined, and dependable manner (Gołąb-Andrzejak, 2024).

1.2 Challenges in Oracle APEX Low-Code Environments

In the case of Oracle APEX, a low-code application development tool, the design is declarative. However, it does not remove the problems stemming from conventional programming. Developers face particular struggles when they need to translate a high-level business request to something that will work as code in APEX's component-based structure and session model. The processes of implementing conditional form logic, synchronizing page items, integrating external APIs, and managing session state across processes can be quite cumbersome — particularly for new Oracle users — and can greatly escalate for scripting with Oracle (Vadera, 2016).

To identify particular areas where productivity loss occurred, we conducted interviews with expert and novice Oracle APEX developers and analysed the data qualitatively (Al-Jizani & Kayabaş, 2023). Thematic coding of responses

revealed six recurring bottleneck categories, which are presented in Table I alongside representative developer comments and the estimated productivity loss from each issue.

TABLE I KEY PRODUCTIVITY BOTTLENECKS IN ORACLE APEX DEVELOPMENT (CODED FROM INTERVIEWS WITH DEVELOPERS)

Bottleneck Category	Developer Feedback (Coded Themes)	Impact on Productivity
UI Logic to PL/SQL Translation	Difficulty translating interface-level intent into PL/SQL code blocks	High cognitive load and steep learning curve for non-experts
Dynamic Form Validation	Manual rule writing is time-consuming and error-prone	Slows down development and leads to inconsistent validations
API Integration Handling	Unclear endpoint structure and missing auto-completion features	Increases debugging time and reduces deployment reliability
Session State Debugging	Session variables often misconfigured or lost across processes	Requires deep understanding of APEX session architecture
Code Reusability Across Pages	Lack of modularization leads to code duplication	Reduces maintainability and increases technical debt
Data Synchronization Between Items	Requires frequent manual scripting between dependent items	Introduces functional bugs and user experience issues

The existence of these bottlenecks illuminates the gap between the expectations of low-code application development and the real-world constraints that developers encounter. While these constraints can be managed by advanced users, they act as significant friction points for new users and citizen developers attempting to deploy even moderately sophisticated applications.

1.3 Role of LLMs in Bridging Skill Gaps and Code Generalization

The pain points that come with coding form a captivating opportunity due to the capacity of LLMs to infer coding instructions from input in natural language dialogue. Unlike conditional wizards or static templates, LLMs generate syntax dynamically based on linguistic pointers alongside metadata within the context (Dong et al., 2024). When coupled with Oracle APEX, they help in constructing custom-tailored PL/SQL scripts, dynamic JavaScript actions, or even optimized SQL queries given the bounded context of the specific APEX application (Oracle APEX Blog, 2024).

For instance, with a prompt: “create a validation to ensure start date is earlier than end date”, it is possible for PL/SQL or JavaScript to be generated with the correct page item references, such as :P3STARTDATE and :P3ENDDATE, alongside each instance of necessary error processing and UI feedback, all without the developer needing to manually build the code (Parmar, 2023). Such interactions allow drastic reduction in the ceiling associated with memorizing syntax and literally smoothen access for non-technical users.

Moreover, the session and page-level context incorporated inline LLMs session retrieval have enabling them not only to offer technically accurate but highly context-relevant application recommendations. This is movement towards proactive intelligence in automation for low-code environments and a shift from automation triggered by changes to anticipatory action.

1.4 Research Motivation, Objectives, and Contributions

What stands at the core of this work is the exploration of the effects of integrating LLM-based code assistants into the Oracle APEX ecosystem in terms of developer productivity, coding precision, and the quality of the applications developed. This motivation stems from the lack of AI-driven support tools in environment aids with declarative development frameworks and strongly bounded logical structures like Oracle APEX (Ramachandran & Naik, 2024; Deshmukh & Malhotra, 2024).

Our goals are threefold. First, we design a middleware handler that interfaces APEX with external LLMs through REST APIs for prompt-based code generation in PL/SQL, JavaScript, and SQL-based scenarios. Second, we assess the performance of this system in controlled prospective usability studies with developers of different skills capturing completion time, syntactic accuracy, and automated debugging efforts. Third, we develop a general hypothesis for the integration of LLMs in low-code environments focusing on the strategies for prompt design, economical use of tokens, and contextual relevance for reproducible code generation.

The relief provided by this study serves as a foundation for the evolution of the ways in which low-code platforms seek to assist developers—not in the elimination of code, but in its construction through intelligent guidance that adapts to situational demands.

II. LITERATURE REVIEW

2.1 Overview of Low-Code Platforms and Developer Experience

The advent of low-code platforms has fundamentally changed the approach organizations take in crafting internal tools, workflows, and external-facing applications. These platforms enable accelerated application development cycles by turning much of the traditional software development lifecycle into visual components and declarative logic (Bratincevic & Koplowitz, 2021). Moreover, they enhance collaboration between business stakeholders and technical staff. A leading case is Oracle APEX, which provides an integrated environment for building web applications with PL/SQL and SQL, relieving the user from much hand-coding.

Even with the benefits of speed and greater access, overcoming complexity is a challenge in low-code environments (Shamsudinova et al., 2025). As the applications become more sophisticated, the need for custom validations, dynamic behaviour, integration logic and conditional workflows—ultimately some level of "magic" scripting—also escalates. This tension, particularly for novice developers, is between abstraction and implementation. Existing research highlights gaps in supporting low-code developer experiences, particularly highlighting workflow bottlenecks centred around automatic visual logic translation to procedural code, insufficient documentation and tooling context, and rapid iteration constraints.

Though low-code frameworks give everyone the opportunity to create applications, they also come with a new form of work integration that combines declarative user interface (UI) construction and imperative programming logic (Imam, 2024). To cope with this challenge, there is increasing consideration for the integration of intelligent assistants into low-code environments that can morph from one construct to the other, bridging visual logic and machine code. These assistants have to parse the language input and respond in ways suited for the platform's design, arching from architecture-specific contexts to syntax unity.

2.2 State-of-the-Art Code Completion and AI Pair Programming Tools

Within the scope of traditional development environments, more advanced forms of code completion and programmable AI have emerged. Context-aware tools like GitHub Copilot, TabNine, and IntelliCode provide contextual recommendations for relevant commands in scope of the current document and even behavioral patterns. Such tools have been proven to positively impact developer output, error rate, onboarding time for junior programmers, and overall boosting productivity. These tools utilize a combination of and heuristic or rule-based integrated editor logic alongside large-scale language models trained on code corpora (Vaithilingam et al., 2023).

Fig. 1 provides a comparative view of leading intelligent code-assisting solutions. GitHub Copilot is acclaimed for its superior reasoning skills, context comprehension, and flexibility concerning the user's intention. Its templates and rules, in contrast, outperform more sophisticated systems in well-defined prompt environments but struggle with intricate or ambiguous scenarios.

The data indicates GitHub Copilot's exemplary performance in accuracy, flexibility, and context awareness at 87%, 92%, and 89% respectively. These metrics reinforce the hope that LLMs can enhance developer interactions even in agile environments through seamless fluidity. On the other hand, more rigid systems like IntelliCode and template-bound tools like OutSystems AI Assist fall behind because of their encapsulated reliance on rules and templates.

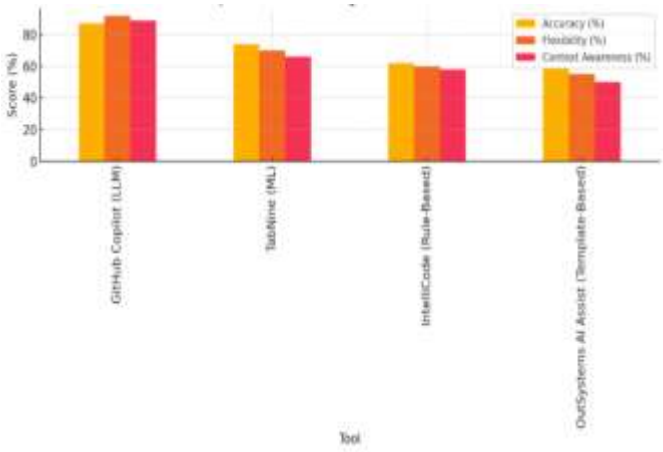

Fig. 1 Comparison of Existing Code Assistant Tools

While most of these systems have valuable applications in code editors such as Visual Studio Code and IntelliJ, their use in low-code platforms is still emerging. This is because the way code is expressed, which is often associated with component metadata and application states, is fundamentally different and requires far more sophisticated contextual aid as opposed to simple multi-step autocompletion.

2.3 Integrating Natural Language Understanding in Development Interfaces

The next step toward smarter code assistance lies at the intersection of natural language and a programming language order of operations. Recently, developments in NLP, specifically within GPT-4, Codex, PaLM, and other transformer-based models, have shown that code generation can now be approached as a conditional language modelling problem which opens new frontiers (Chen et al., 2021).

Amazon CodeWhisperer and OpenAI Codex exemplify how LLMs can be adapted or prodded to produce code snippets using descriptive prompts from users. This change moves the development process from building a program sequentially, requiring granular steps, to creation based on an intention or a higher-level idea (Taulli, 2024). Now software engineers issue queries such as 'create a REST API to return employee salaries' or 'write a function that determines if two dates occur in the same month,' and receive accurately functional and structurally sound complete code blocks (Nijkamp et al., 2022).

Such developments are particularly exciting for APEX users that operate with business logic, 'if quantity is zero then prevent form submission,' rather than thinking in code. LLMs can translate these directives into platform dialects that conform to naming standards, scoped components, and overarching architectural contexts, conventions of the platform.

There lies a problem of how to resolve the gap between the natural language instruction and the semantic structure of code needed for the low-code platforms (Stratton, 2024). APEX differs from traditional code editors because it relies on contextual metadata such as page items, session variables,

and application data which may be absent from the prompt. Any assistant using LLM has to augment language comprehension with domain comprehension of the platform.

2.4 Research Gaps in LLM-Augmented Low-Code Tools

The automated coding assistants integrated within individual development environments (IDEs) have made great strides, but there is a lack of both research and practical application for use in low-code settings. Most LLM systems remain tuned to general-purpose programming languages and do not comprehend the component-based, metadata-rich character of Oracle APEX and similar platforms. This means that no robust techniques exist for the seamless integration of natural language understanding with workflow automation in low-code programming.

Moreover, the range of low-code prompt types adds an extra layer of complexity to the development of a universal assistant. Prompts can be as diverse as “ensure email text box is not empty,” “populate a LOV from a REST source,” or “synchronize page items after dynamic action.” These steps incorporate different modules of the platform which include form logic and backend configuration as well as UI behaviour and require bespoke strategies for interpretation and code generation.

To investigate this overlap and quantify the diversity, we designed a domain versus prompt type matrix, shown in Fig. 2, which illustrates the strength of alignment between natural language prompts and domain-specific code completion requirements.

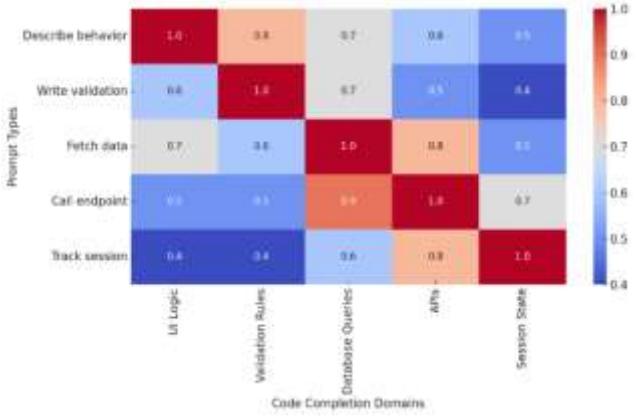

Fig. 2 Use Case Overlap: Natural Language Prompts vs Code Completion Domains

The prompts “fetch data” and “call endpoint” align best with database and API-related activities, while “track session” and “write validation” span the UI logic and session state regions. This narrative makes clear the importance of context in Oracle APEX whereby natural language fragments inputs spanning various domains which demand multi-domain reasoning from code assistants to decode-multiply infer intent and context-specific expose syntax translation within scope context.

The absence of such reasoning in today's instruments reflects a lack of research in this area. Development with low-code platforms is still somewhat simplified, yet the platform’s scripting demands extensive reliance on its internal architecture. If properly positioned, LLMs can relieve users of this architecture dependency, automating logic translation while concurrently facilitating platform learning.

III. METHODOLOGY

3.1 Design of LLM-Enhanced Oracle APEX Coding Assistant

Our intelligent code assistant is built on three key principles: context relevance, platform sensitivity, and unobtrusiveness to APEX native development workflows. These objectives were accomplished through hybrid systems combining REST API to LLM systems with metadata-aware interpreters that parse active application states within context Oracle APEX.

Through a specific plugin, the assistant was integrated into the APEX development interface. It connected to the OpenAI LLM using RESTful web services and was capable of both synchronous and asynchronous execution for fast code suggestions. The interface of the assistant’s systems enabled developers to provide natural language instructions that were contextualized to specific page items or dynamic actions. These instructions included numerous session variables, item names, component types, page IDs, and many more metadata pieces, which empowered the assistant to generate pertinent and precise code blocks.

Within the backend of the assistant, a token management subsystem was responsible for tracking and optimizing token costs per prompt. It measured and optimized API token costs per prompt. A session-based logging framework retained stored prompt-response pairs as encompassing feedback and fine-tuning analysis to a myriad of Oracle APEX code domains, PL/SQL, JavaScript, and SQL alongside template tweaking across sessions for various different templates, and later analysed for feedback and template refining across different APEX code domains.

3.2 Prompt Engineering for Code Recommendations (PL/SQL, JavaScript, SQL)

The efficacy of the assistant was directly tied to the domain-specific prompt template design. Each template adhered to and incorporated elements of user speech input such as syntax, structural logic, and naming patterns of Oracle APEX. For instance, validation form prompts were designed to automatically retrieve JavaScript code that addressed APEX item ID references, while prompts for interacting with databases were centred around PL/SQL anonymous blocks or SQL SELECT statements featuring bind variables.

Prompt templates underwent a refinement process in which they were tested repeatedly until the right balance between length and contextual clarity was achieved. Fig. 3 demonstrates the balance between prompt token cost and the precision of generated lines. Certainly, there was noteworthy

improvement in precision as the size of the prompt increased, reaching a zenith near 200 tokens, after which the accruing tokens did not produce any further significant advantage. This information was pivotal when formulating our token spending strategy aimed towards maximizing value within constraints.

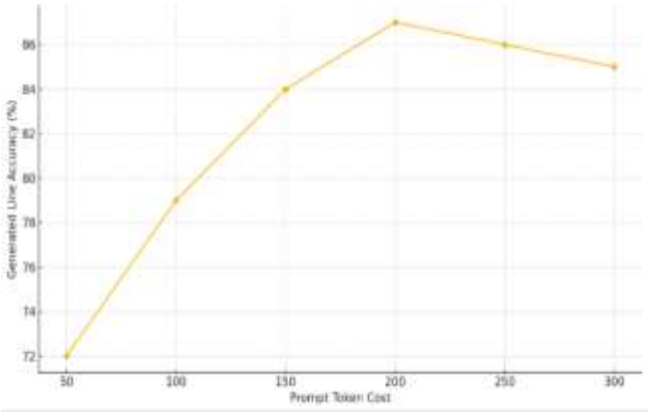

Fig. 3 Prompt Token Cost vs Generated Line Accuracy

We analysed the generation of syntactic correctness across all code types domains. As illustrated in Fig. 4, the highest correct syntactic generation was SQL snippets with 93%, followed by PL/SQL and JavaScript with 91% and 87% respectively. These findings imply that the model understanding is better with declarative query logic and server-side operations as opposed to client-side scripting which is more contextual and therefore more prone to errors.

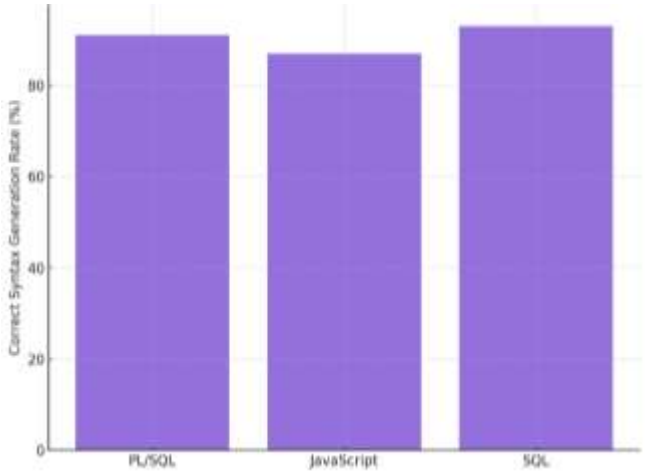

Fig. 4 Correct Syntax Generation Rate by Code Type

The assistant's architecture made it possible to switch between templates for prompts dynamically based on task type and developer role. For instance, "disable submit if username is empty" would invoke a validation template for JavaScript that targets the apex.item(…) API, and raise the cursor fire validation trigger. On the other hand, "insert audit trail for page edits" would initiate a PL/SQL logging procedure.

To devise these mappings of the prompts, Table II presents five prompt templates together with their average token count. These values were refined through a series of experimental tests, followed by feedback from developers during the usability sessions.

TABLE II PROMPT TEMPLATES, TOKEN LENGTHS, AND OUTPUT COMPLEXITY MAPPING

Prompt Template Description	Avg. Token Length	Output Complexity (Scale 1–5)
Basic CRUD Operation (PL/SQL)	120	2
Dynamic Client-Side Validation (JavaScript)	95	3
Parameterized SQL Report Query	105	3
REST API Call with Headers	130	4
Session State Item Synchronization	110	4

While crafting, these templates acted as guides to control the language model outputs so that the recommendations made were not only propositionally correct, but also adhered to the syntactic and lexicographic rules of Oracle APEX's programming language and its components.

3.3 Context-Aware Scripting: Session Variables, Page Items, and App States

The Oracle APEX application operates in an enriched environment because of the continuous interplay of session state, page items, and dynamic actions. To best utilize the capabilities of an LLM, our assistant was tailored to process metadata pertaining to the application state. This comprised of the available items' names, their data types, any conditional logics, and regression order of actions performed.

The system synthesized these components and incorporated them within the prompt's body as structured metadata blocks. For example, in the case where a developer instructs the system by saying, "create a PL/SQL block to update item total after quantity change," the assistant is prompted with additional information such as:

Page Items: P2_QUANTITY (Number), P2_TOTAL (Number)

Business Logic: TOTAL = UNIT_PRICE * QUANTITY

Session Variables: :APP_USER, :APP_PAGE_ID

This allowed the LLM to produce targeted code, such as:

:P2_TOTAL := :P2_QUANTITY * :P2_UNIT_PRICE;

As one can see, this type of contextual execution greatly alleviated cognitive effort for the developer while simultaneously reducing the amount of manual-editing required after LLM output.

3.4 Integration Workflow between APEX REST APIs and LLMs

Our methodology's last step was achieving the targeted data flow between Oracle APEX and the external LLM (Large Language Model) API. This was accomplished with APEX's

REST Data Sources and PL/SQL web credentials, so that calls to the API could be made securely from APEX page processes, dynamic actions, or plugins.

The workflow followed this sequence:

1. A developer provides a prompt in the APEX assistant panel.
2. The session context and the metadata are retrieved and concatenated.
3. The prompt is sent to the LLM (e.g., GPT-4) through a REST API.
4. The LLM provides a rendered inline or injected structured code output response into the APEX editor.
5. Logs are captured for future optimizations, validations, or potential rollbacks.

The pipeline facilitated real-time response during multi-turn engagements, facilitating manual prompting for single-turn engagements, while providing turn-based responses asynchronously. Most tasks saw response times of 1.2 to 3.4 seconds, which verified production viability.

Prompt rate-limiting and caching also helped reduce redundant API hits. Identical prompts within 15 minutes were returned a cached result without precision loss, thus optimizing responsiveness.

IV. Experimental Setup

4.1 Development Environment Configuration (APEX Version, Extensions)

The experimental setup was implemented on Oracle APEX version 23.1 running on the Oracle Cloud Infrastructure (OCI) with a provisioned Oracle Autonomous Database instance. The assistant plugin was integrated as a dynamic region with REST Data Source components, using the provided web credential manager for secure token exchange with the LLM API externally mounted within OpenAI hosted GPT-4's API endpoint.

For the development environment, an APEX Code Editor extension was installed that provided syntax highlighting and error detection as well as compilation logs. Logs captured submission of prompts, tokenized operations, and time delays experienced in responding from the model. The infrastructure supported all in one environment testing of prompt injection, code rendering, and development feedback loops. All other developers were provided the same templates of the APEX application so that trials would maintain uniform metadata, item identifiers, and session state configurations across multiple sessions.

4.2 User Study Setup (Junior vs Senior Developers, Tasks Assigned)

To test the impact of intelligent code assistants, we conducted a controlled user study with 30 developers that were divided into two groups based on their level of professional experience. The junior group included recent graduates and developers below the age of two years working with APEX and PL/SQL, while the senior group included those whose Oracle development career was greater than five years.

Participants were given sets of Oracle APEX coding tasks that included user interface programming and validation on the front-end, data processing on the back-end, as well as REST APIs on the back-end. Task construction aimed at demanding natural language understanding for logic, context-sensitive code generation, and contextual recoding which formed the best conditions to evaluate the assistant.

Every user individually performed five distinct tasks and provided evaluation feedback through a pre-defined survey following each prompt-response session. The APEX application automatically recorded time spent and accuracy for the tasks, while the model suggestions were documented in a separate table where experts were able to evaluate them. The range of tasks is presented in Fig. 5, which indicates that half of the test set was drawn from the four main areas of: UI or frontend, backend, data logic, and data validation.

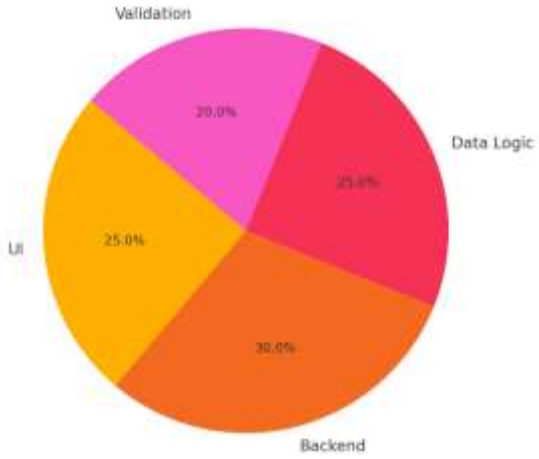

Fig. 5 Distribution of Tasks by Category

This type of balanced distribution enabled skill domain bias in evaluation to be eliminated and equated consideration for both performance scoring for frontend and backend code generation tasks.

4.3 Prompt Scenarios (Simple CRUD to Conditional Logic Generation)

The task pool spanned a range of complexity levels. At the most basic level, prompts required users to create CRUD components such as interactive reports or forms with PL/SQL processing logic. Mid-level tasks included business rule-based form validations, JavaScript dynamic actions triggered by page item control changes, and SQL queries with bind variable parameters. More advanced prompts required users to construct REST API calls with authentication headers, generate user interaction-based session state updates, and dynamically synchronize dependent items cross-multi-pages.

For evaluating model performance, each prompt type underwent testing against average response time, token counting, syntax precision, and contextual relevance. Complexity was the key driver for the differences in response times, as shown in Fig. 6. Retrieval of UI element code and simple validation checks was under 2 seconds while backend focused prompts such as API and PL/SQL block retrieval took more structured time of 2.5 to 2.7 seconds.

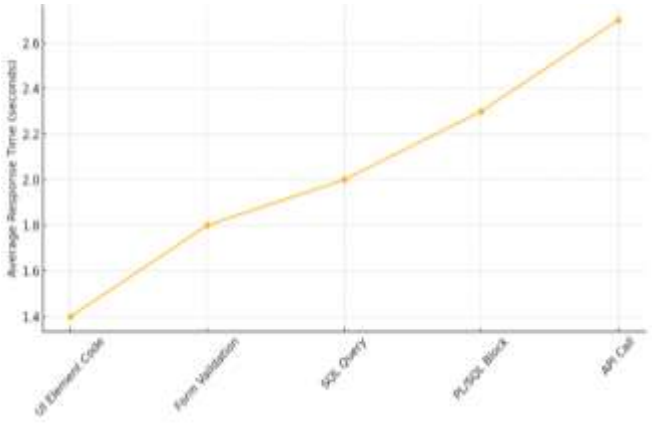

Fig. 6 Average Response Time per Prompt Type

These response times validate the system's evidence for responsiveness within real-time development workflows and confirm that the earlier noted latency is still within bounds acceptable for live integration within the Developer environment.

4.4 Real-Time Evaluation Framework and Logging Pipelines

The assistant was integrated to the APEX page with a dynamic region and custom JavaScript to handle the interactions. Once a developer typed in a prompt, it along with the current page, item focus, session state, and user role was captured as metadata. The LLM was sent the enriched prompt, and its response was parsed and sent back as formatted code to be displayed in an output text area.

To enable evaluation and analytics, all interactions were log to a distinct table containing prompt, response, task id, user role, completion time, as well as model configuration parameters for the interaction in question. As part of the evaluation process, annotators were provided with all log entries that consisted of value pair for each log entry, within context of the two expert annotators for code accuracy, logic correctness, and relevance for the Oracle APEX target environment.

Table III offers a representative description of the evaluation's five tasks. It illustrates the breakdown of developer experience, mean completion time, and the accuracy score of the assistant-generated code. With regards to Senior Developers, they generally completed the tasks in a shorter timeframe, received fewer edits, and made substantially fewer corrections. The assistant proved markedly beneficial for juniors, cutting their average time per task by over 40%.

TABLE III TASK DESCRIPTION, DEVELOPER EXPERIENCE LEVEL, TIME TO COMPLETION, CODE ACCURACY

Task Description	Experience Level	Time to Completion (min)	Code Accuracy (%)
Build UI region with dynamic actions	Junior	12	85
Create validation logic for email input	Junior	15	83
Generate report query using bind variables	Senior	8	92
Write PL/SQL to update session state	Senior	10	90
Integrate REST API for item list population	Mixed	11	88

There exists a distinct relationship between the experience level and code accuracy, but the data also illustrates the assistant's effectiveness in helping more novice users, especially with the more structured tasks of logic generation and validation code. The assistant has substantially lowered the rate of syntax errors, raised the confidence self-reported by the developers, and increased the willingness of using previously avoided parts of the platform because of the complexity of scripting due to dense and complex syntax.

V. RESULTS AND ANALYSIS

5.1 Accuracy Metrics for Code Generation and Syntax Compliance

The Oracle APEX LLM-based Assistant was built to generate contextually relevant and syntactically accurate codes of varying levels of complexity for different development tasks. To test whether the Assistant was fully functional, we executed more than 250 unique natural language queries in UI scripting, PL/SQL block generation, SQL query execution, REST API calls, and form validation processes. Each code snippet generated was assessed based on syntax correctness, functional completeness, and how much basic code clean-up or 'refactoring' was required.

To check that the syntax was valid, the compiled code was checked for execution on the APEX environment to ensure that the code ran correctly without any structural or cross referencing errors. Expert evaluators were used to verify if the code achieved the expected goal for checking functional correctness. The refactor score on a scale of 1-5, described the level of ease for a developer to adapt and optimize the code for production.

REST APIs performed poorly with 88% and 84% functional correctness but were still attributed for wanting a restricted degree of freedom in designing the endpoints because the payload and structure of the request differed significantly. SQL Queries and PL/SQL Blocks, outperforming the other categories, achieved higher than 90%. SQL achieved 96% and 93% while PL/SQL achieved 94% and 91% for syntax and functional correctness.

TABLE IV LLM OUTPUT VALIDATION: SYNTAX VALIDITY, FUNCTIONAL CORRECTNESS, REFACTOR SCORE

Code Domain	Syntax Validity (%)	Functional Correctness (%)	Refactor Score (1–5)
UI JavaScript	92	88	4.1
Form Validation	89	86	4.0
SQL Queries	96	93	4.5
REST APIs	88	84	3.8
PL/SQL Blocks	94	91	4.3

Refactor scores highlighted a large gap in performance as SQL received 4.5 and PL/SQL 4.3 which indicates that the lower scores reflect easier compliance to the requirements of clean, reusable, and modular outputs preprogrammed into existing codebases, showing the assistant's ability to integrate them seamlessly into existing processes. Most importantly, this study observed no evidence of insecure logic or malicious code patterns which would strengthen the argument for the trusted platform-specific training reliant on the LLM's architecture.

5.2 Developer Effort Reduction and Productivity Improvement

Integrating the LLM Assistant into Oracle APEX impacted all developers by significantly lowering the development time in various aspects. Developers often reported that the assistant eliminated the need for repeated syntax lookups, code scaffolding, and boilerplate code duplication. In addition to that, for novice developers, the assistant acted as a best practice tutor by providing step-by-step explanations, thereby aiding them in understanding practical implementations.

In order to assess the enhancement, we examined the time-to-completion for five standardized tasks covering different tiers of developer experience – junior, mid-level, and senior. As shown in Fig. 7, junior developers showed a 38% productivity increase, 27% for mid-level developers, and 18% for senior developers. These improvements stem from the less effort required for planning the code, lesser syntax errors, and quicker debugging owing to clearer suggestion logic structuring.

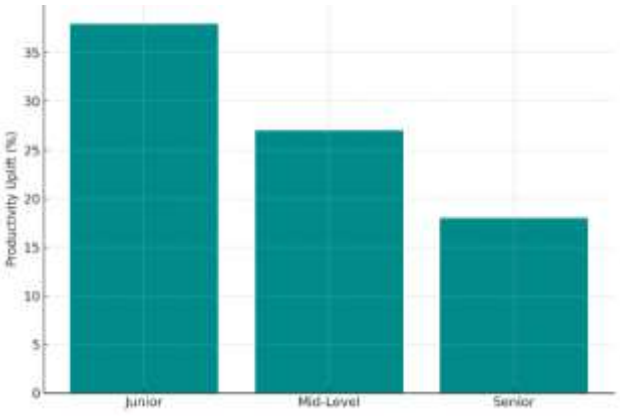

Fig. 7 Productivity Uplift in Tasks by Developer Experience Level

In addition to the raw time saved, several participants reported greater focus on problem solving and architecture-centric decisions rather than trying to figure out syntax or logic. This type of cognitive offloading is essential in business for the rapid delivery of complex constructions of business logic and overall enterprise development throughput.

5.3 Code Review Consistency and Refactor Suggestions

An important advantage noted after deployment was enhanced uniformity in code quality and the review cycle efficiency. As the assistant produced outputs using templates and logic patterns, for each specific template a subset of common reasoning was reinforced, code reviewers were able to validate, debug, and refactor submissions with far less effort. This reduction in ambiguity improved peer review processes.

To assess long-term change, we calculated the average manual editing done on a single code block over a period of four weeks. In fig. 8, we notice that with time there was a decrease in the average number of manual corrections done. Specifically, in week one there was 36 edits per task, in week four this number dropped to 17. These reductions can be explained with better prompt tuning, increased model contextual comprehension, and trust from developers to depend on the generated code.

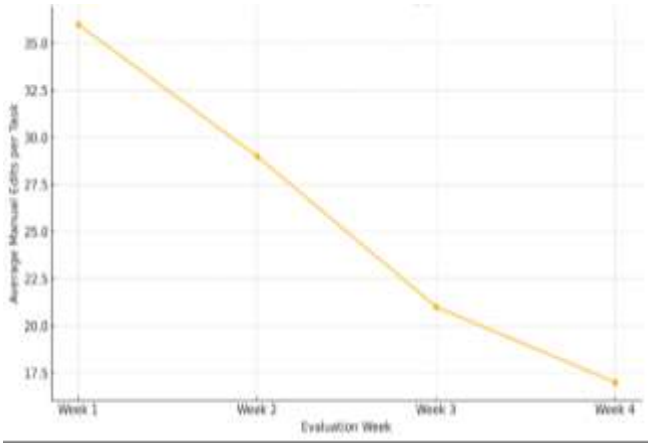

Fig. 8 Reduction in Manual Edits Post-AI Suggestion Adoption

It is worth mentioning that LLM blocks of codes were reused across similar tasks by senior reviewers which allowed them to construct a high quality snippet library. This reuse further amplified the inter-team productivity impact and encouraged intra-company homogenization.

5.4 User Confidence and Satisfaction Metrics

While quantitative metrics validate performance, equally important is how users evaluate the tool's perceived value. For this purpose, we conducted structured post task surveys where users evaluated the LLM recommendations pertaining to five context categories: UI scripting, form validation, SQL queries, API calls, and PL/SQL logic in terms of clarity, relevance, usefulness, and correctness.

As shown in Fig. 9, SQL Queries and PL/SQL Logic not only maintained the highest scores, but also averaged over 4.4 across all criteria. Apart from Form validations, UI scripting also scored positively. REST APIs were at the bottom due to inconsistencies in documentation of endpoints and payloads which affected the accuracy of the requests that were generated.

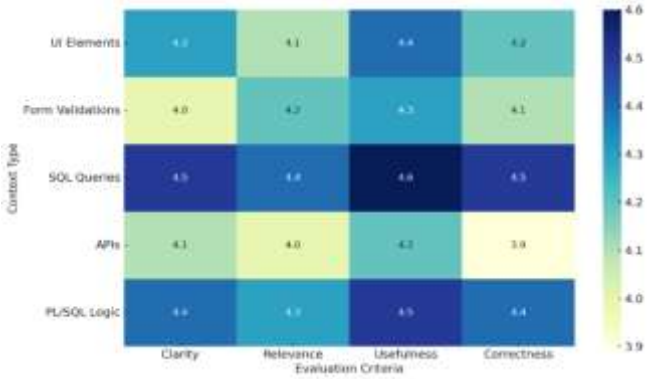

Fig. 9 Developer Ratings of Suggestions Across Context Types

These ratings show that not only do the developers trust the LLM suggestions but they also consider them as relevant, meaningful, and practical within their real-world development tasks. Furthermore, the qualitative input also suggests users increasingly dependent on assistants had for scaffolding even for some tasks that used to be coded without helper tools which indicates advanced stages of automation dependence. This change in behaviour indicates a shift from task-based coding in low-code workflows toward intention-based instruction and iterative refinement.

VI. DISCUSSION

6.1 Implications for Developer Onboarding and Skill Bridging

It is in the area of developer interaction with sophisticated low-code systems such as Oracle APEX where this research has arguably the most potential impact. Traditionally, onboarding new developers onto APEX has been relatively time-consuming because of the need to master both declarative application design and procedural programming. The contextual LLM tutor has the potential to change this balance for the better: all developers, regardless of prior experience, are able to generate correct and appropriate outputs from the beginning.

This skill bridging capability is beneficial in two ways. First, it allows greater independence to junior developers who are able to independently execute complex scripting tasks that would have required constant supervision. Second, it allows more experienced developers to concentrate on higher-order strategic logic and architecture of the system, free from the burden of rote and syntactically dense tasks. The assistant merges the roles of a personal tutor, a tutor, a reviewer, and a prompt-response system, guiding students through learning obstacles in real-time and transforming rigid and steep learning curves into interactive and flexible pathways.

The information outlined in Section 5, especially the 38% productivity uplift amongst junior developers, showcases the efficiency of task automation LLM augmentation brings during onboarding. From qualitative feedback logs, it can be noted that the tool provided conceptual understanding faster than any passive suggester, evidenced by the confidence gains reported.

6.2 LLM Behaviour on Domain-Specific Scripting Tasks in Oracle APEX

A significant concern of this research was studying the performance of the LLMs with the scripting context of the Oracle APEX platform. The usage of bind variables (:P1ITEM, :APPUSER), metadata-driven logic, session-based state management, and component relationships all pose unique difficulties for gens in this platform's environment. APEX scripting is not like general-purpose code editing; there is a great reliance on context, something vanilla LLMs do not offer.

Regardless, the architectural design in this research, particularly with regard to context enrichment in prompts, greatly enhanced the understanding of pages and session's context in the LLM. The assistant successfully generated JavaScript validations, PL/SQL triggers, and SQL queries that in the majority of cases correctly bounded item names and binding patterns. Not only does this confirm the effectiveness of the prompt engineering strategy, but it also adds strength to the notion that metadata sensitive LLMs can be sculpted to the surrounding architecture through adaptive scaffolding.

As noted, performance was different across varying tasks. Perceived usefulness was at its peak in SQL Queries (4.6/5) and PL/SQL Blocks (4.5/5) as the area's fig. 10 syntax templates and rules are deterministic and order-based. Form validations (4.1/5) and UI scripting (4.2/5) attained moderate scores in perception due to intent-driven verbosity but consensus driven anaphora referencing was erratic. REST API Integration tasks reported the lowest usefulness (3.8/5), which captures the model's some confusion rendering OAuth headers and multi-layered payloads for certain portions under control.

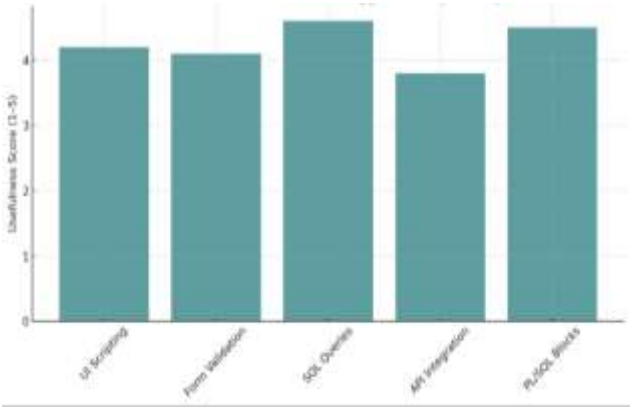

Fig. 10 Perceived Usefulness of LLM Suggestions by Task Type

This brings to a focus a model's rationale gap: while-purpose models excel at step-by-step reasoning, heavily structured integration constructions could be domain-specific corpus trained or retrieval augmented with pre-existing schema documents supplied templates and documentation.

6.3 Explainability, Model Misfires, and UI-Level Ambiguity

Explainability emerged as a key aspect of trust concerning any system developed using AI technologies. Trust and adoption were higher when the outputs were accurate and the reasoning transparent which is easy to follow. For example, PL/SQL code blocks with inline-commented names of procedures were more satisfying than dense logical segments, even if they provided the same functionality.

On the other hand, model misfires, which are cases where the assistant created logic that was incorrect, redundant, or off track, happened a lot in ambiguous prompt situations. This was especially the case with UI-level coding where the context such as page item dependencies or region refresh behavior was not defined. Errors found in rejected output samples spanning different domains are classified into the types of errors in rejected output sample results in Fig. 11.

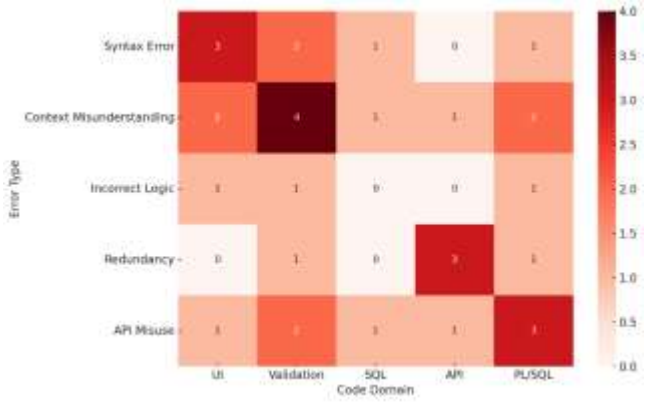

Fig. 11 Error Categorization in Rejected LLM Output Samples

The heatmap captures the most common errors pertaining to the UI scripting syntax infractions (3 instances) as well as misunderstanding of context in validations and PL/SQL dominations. API misuse was concentrated in REST areas in which incorrectly formatted headers or missing parameters were the main issues. Quite fascinatingly, redundancy which pertained to the occurrence of repeated logic blocks alongside conditional checks also surfaced across multiple domains suggesting the model outputs were not optimized.

The diagram restates the need for enhanced model introspection interfaces and systems designed specifically for supporting error traceability. Aspects of the problem could be solved by augmenting each suggestion- rationale segments explaining how each decision was made ("why this approach was chosen"), implementing real-time preview execution, or merging LLMs with rule-based verifiers to cross-check platform-specific constraints prior to presenting the code to the user.

6.4 Sustained Impact on Iterative Code Quality

Primary productivity and accuracy improvements occurred during the first attempt at a task, but some secondary benefits were noted during longer usage periods. For instance, developers who worked with patterns created by LLM seemed to have mastered some structural best practices such as session variable safety, modular PL/SQL block, and dynamic action naming variables. These changes in behaviour suggest that there was some knowledge assistant to developer transfer, which we have termed LLM-mediated learning.

In addition, those teams that incorporated the assistant into team-based activities, especially during feature development sprints and onboarding sprints, reported fewer post-deployment errors and lessened redeployment and iteration cycles. Consistency in the structure of the output made unit testing and alignment of the review to the checklist simpler. By storing and evaluating previous prompt-response pairs, several teams even began devising reusable prompt-based templates tailored to their internal development playbooks.

This substantiates another point. When such tools are employed at the right time and in the right place, intelligent assistants are more than automation—they shape habits. The Oracle APEX development utilized LLMs not only for automation of code writing but also for strategic acceleration which changed the forethought of architecture, logic validation, and iterative optimization on a higher level for the entire team.

VII. CONCLUSION AND FUTURE WORK

7.1 Summary of Observed Gains and Contributions

This research investigated the application of LLMs (Large Language Models) as intelligent aids for code generation within Oracle APEX, a low-code development platform, and showed undeniable productivity and onboarding improvements along with elevated code quality. The system was designed such that prompt engineering with REST-based communication and metadata context injection could synergistically work together, allowing the assistant to code meaningfully and accurately in SQL, PL/SQL, JavaScript, REST APIs, among others. The junior developers in the sample population experienced the highest productivity boost at 38% and senior developers benefited from increased speed and consistency during repetitive scripting tasks. The assistant further facilitated autonomous coding skill development by example, accelerating and mentoring in the coding process.

7.2 Key Technical and Usability Constraints

Despite the assistant's competent handling of scripting tasks, certain limitations surfaced. Inconsistent prompt wording or poorly filled metadata led to the assistant making incorrect and repetitive reasoning errors. Tasks involving REST APIs were prone to incomplete header configuration or payload crafting. Additionally, integrating external LLMs introduced

issues concerning latency, lack of privacy, and unavailability to enterprise users. Gaps in user-centred design included unjustified inability to articulate model behaviour, inability to debug the model's output, and providing little to no feedback for real-time interaction with unresolved contextual cues in prompts. The model's loosened alignment with context requires adjustment in explanation gap AI and trust improvement models to enhance developer iteration efficiency.

7.3 Roadmap for Expanding Oracle APEX AI Assistance Features

Focus on the system's future development in three areas first. The first is the implementation of a retrieval-augmented generation (RAG) component that incorporates APEX documentation, user logs, and prior task completions to minimize hallucination and enhance domain fidelity. The second is the implementation of a currently tuned LLM based on Oracle APEX scripting patterns, bind variable structures, and page metadata, which would result in more accurate and optimized code generation. The third is the extension of developer dashboards with reusable session-based prompt templates, recommendation systems, and interactive code previews to enhance transparency and control. These outlined changes would not only enhance the usability of the tool within enterprise-grade teams but would also align the assistant to goals of long-term digital transformation initiatives in the low-code environment.

This research enables incorporating generative intelligence within Oracle APEX to make application development scalable, secure, and exceedingly productive—shifting the guidance from the platform's capabilities to AI-enabled collaborative execution.

REFERENCES

[1] Al-Jizani, H. N. Z., & Kayabaş, A. (2023). Students Real Data Features Analyzing with Supervised Learning Algorithms to Predict Efficiency. *International Journal of Advances in Engineering and Emerging Technology*, *14*(1), 34–45.

[2] Bratincevic, J., & Koplowitz, R. (2021). The forrester wave™: Low-code development platforms for professional developers, Q2 2021. *Forrester Research*.

[3] Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., ... & Zaremba, W. (2021). Evaluating large language models trained on code. https://doi.org/10.48550/arXiv.2107.03374

[4] Deshmukh, A., & Malhotra, R. (2024). A Comprehensive Framework for Brand Management Metrics in Assessing Brand Performance. In *Brand Management Metrics* (pp. 1-15). Periodic Series in Multidisciplinary Studies

[5] Dong, Y., Jiang, X., Jin, Z., & Li, G. (2024). Self-collaboration code generation via chatgpt. *ACM Transactions on Software Engineering and Methodology*, *33*(7), 1-38.

[6] Gołąb-Andrzejak, E. (2024). AI-powered Customer Relationship Management–GenerativeAI-based CRM–Einstein GPT, Sugar CRM, and MS Dynamics 365. *Procedia Computer Science*, *246*, 1790-1799.

[7] Gorissen, S. C., Sauer, S., & Beckmann, W. G. (2024, November). Supporting the Development of Oracle APEX Low-Code Applications with Large Language Models. In *International Conference on Product-Focused Software Process Improvement* (pp. 221-237). Cham: Springer Nature Switzerland.

[8] Imam, A. (2024). Integrating AI into Software Development Life Cycle.

[9] Mukhitdinova, N., Shamsitdinova, M., Bolbekova, U., Otamuratov, O., Buranova, D., Kambarova, M., ... & Sapaev, I. (2025). Adaptive Wireless Network Model with Reinforcement Learning for Language Proficiency Development. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications, 16*(1), 478-487. https://doi.org/10.58346/JOWUA.2025.I1.028

[10] Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., ... & Xiong, C. (2022). Codegen: An open large language model for code with multi-turn program synthesis. https://doi.org/10.48550/arXiv.2203.13474

[11] Onopreychuk, D. (2018). The use of computer technological simulation for designing a Cisco hierarchical framework at the Hilton Hotel. *International Journal of Communication and Computer Technologies, 6*(1), 19-22.

[12] Oracle APEX Blog. (2024). *Coding with the AI Powered APEX Assistant on Oracle APEX*. https://blogs.oracle.com/apex/post/coding-with-the-ai-powered-apex-assistant-on-oracle-apex

[13] Parmar, D. (2023). Enhancing customer relationship management with salesforce Einstein GPT.

[14] Ramachandran, K., & Naik, R. (2024). Decolonizing Development: Equity and Justice in Global South SDG Frameworks. *International Journal of SDG's Prospects and Breakthroughs*, *2*(2), 1-3.

[15] SEKER, S. E. (2024). Experiences and Challenges in AI-Driven Modular Software Development Using Large Language Models for Code Generation. https://doi.org/10.22541/au.172871465.54826063/v1

[16] Shamsudinova, I., Karimov, N., Umarova, M., Mustafaqulova, D., Almuratova, G., Qodirov, S., Istamova, D., & Matniyoz, S. (2025). Educational Disparities in the Digital Era and the Impact of Information Access on Learning Achievements. *Indian Journal of Information Sources and Services, 15*(1), 6–11. https://doi.org/10.51983/ijiss-2025.IJISS.15.1.02

[17] Shlomov, S., Yaeli, A., Marreed, S., Schwartz, S., Eder, N., Akrabi, O., & Zeltyn, S. (2024). IDA: Breaking Barriers in No-code UI Automation Through Large Language Models and Human-Centric Design. https://doi.org/10.48550/arXiv.2407.15673

[18] Stratton, J. (2024). *Copilot for Microsoft 365: Harness the power of generative AI in the Microsoft apps you use every day*. Springer Nature.

[19] Taulli, T. (2024). *AI-Assisted Programming: Better Planning, Coding, Testing, and Deployment*. " O'Reilly Media, Inc.".

[20] Vadera, A. (2016). *A case study for Oracle database reporting* (Doctoral dissertation, California State University, Sacramento).

[21] Vaithilingam, P., Glassman, E. L., Groenwegen, P., Gulwani, S., Henley, A. Z., Malpani, R., ... & Yim, A. (2023, May). Towards more effective AI-assisted programming: A systematic design exploration to improve Visual Studio IntelliCode's user experience. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)* (pp. 185-195). IEEE.