arXiv:2603.18604v2 [cs.NI] 19 Nov 2026

xAppGen: LLM-driven Natural Language Programming for Agile xApp Development

Xin Li, Shiming Yu, Leming Shen, Jianing Zhang, Yuanqing Zheng, and Yaxiong Xie

***Abstract*—Traditional RAN systems are closed and monolithic, stifling innovation. The openness and programmability enabled by Open Radio Access Network (O-RAN) are envisioned to revolutionize cellular networks with control-plane applications–xApps. The development of xApps (typically by third-party developers), however, remains time-consuming and cumbersome, often requiring months of manual coding and integration, which hinders the roll-out of new functionalities in practice. To lower the barrier of xApp development for both developers and network operators, we present *xAppGen*, the first LLM-driven natural language programming framework for agile xApps that automates the entire xApp development pipeline. In a nutshell, *xAppGen* turns high-level user intents into swiftly deployable xApps within minutes, eliminating the need for manual coding or testing. To this end, *xAppGen* builds a fully automated xApp generation pipeline, which integrates multiple functional modules (from user requirement elicitation, AI/ML function design and validation, to xApp synthesis and deployment). We design, implement, and comprehensively evaluate *xAppGen* on representative xApp tasks. Results show *xAppGen*-generated xApps can achieve similar or even better performance than the best known manually developed baselines. *xAppGen* drastically accelerates the xApp development cycle (from user intent elicitation to roll-out), streamlining O-RAN innovation.**

***Index Terms*—O-RAN, xApp Generation, near-RT RIC, LLM-assisted Coding.**

I. Introduction

In recent years, Open Radio Access Network (O-RAN) [1]–[8] has emerged as a promising paradigm for more intelligent and flexible cellular networks. O-RAN disaggregates the traditional RAN into multiple modularized units, including the O-RAN Central Unit (O-CU), O-RAN Distributed Unit (O-DU), and O-RAN Radio Unit (O-RU), and standardizes the interfaces for interoperability. A key element of O-RAN is the RAN Intelligent Controllers (RICs), which act as the network's intelligence layer by hosting modular control-plane applications across different control timescales, including *xApps* on the near-real-time (near-RT) RIC and *rApps* on the non-real-time (non-RT) RIC. Specifically, xApps execute near-RT monitoring and control tasks at 10 ms to 1 s timescales, *e.g.*, anomaly detection [9]–[11] and traffic steering [12], [13] for self-optimizing networks. This open and programmable architecture unleashes innovation by allowing third-party developers to introduce new RAN functionalities via xApps. However, it also creates an urgent need for **fast and flexible xApp development:** network operators must be able to develop and update xApps at the pace of evolving service demands.

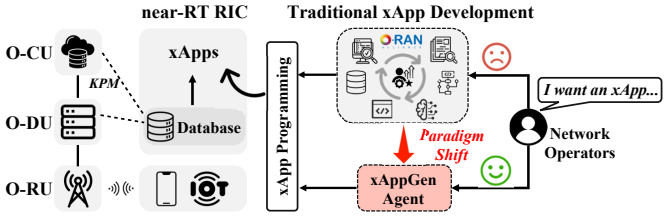


Fig. 1. Conventional xApp development hinders O-RAN innovation. *xAppGen* presents an automated agile xApp development framework.

Developing a new xApp today, however, remains *time-consuming and cumbersome*. It requires deep expertise in O-RAN architecture with complex interface specifications, protocols, and artificial intelligence/machine learning (AI/ML) algorithms for network control. Thus, third-party xApp developers often face tremendous challenges aligning with operators requirements, hindering the rollout of new functionalities. In practice, introducing a new service via xApps often requires substantial in-house development effort or reliance on external developers. In-house development demands substantial time and resources, while outsourcing raises concerns about data privacy and limited transparency. Fig. 1 illustrates the high-level gap between manual xApp development and the automated xApp generation paradigm enabled by *xAppGen*. Moreover, as networks scale and become more heterogeneous, operators may need to deploy hundreds of specialized xApps to cover diverse use cases [2], [14]–[17]. This manual development paradigm has become a major bottleneck to O-RAN innovation, calling for a radically more efficient approach to xApp creation.

To address these challenges, we introduce *xAppGen*, the first large language model (LLM)-driven natural language programming framework for agile xApp development. Rather than directly prompting an LLM to generate a complete xApp in one shot, *xAppGen* follows a staged pipeline that refines operator intents, retrieves task- and platform-specific O-RAN information, and validates both the core logic and interface integration before deployment. Thus, a network operator can describe a desired network functionality or policy in natural language, while *xAppGen* generates a corresponding deployable near-RT RIC xApp that satisfies the functional and

Xin Li, Shiming Yu, Leming Shen, Jianing Zhang, and Yuanqing Zheng are with the Department of Computing, The Hong Kong Polytechnic University, Hong Kong SAR, China (e-mail: cs-xin.li@connect.polyu.hk; shiming.yu@connect.polyu.hk; leming.shen@connect.polyu.hk; jianing98.zhang@connect.polyu.hk; csyqzheng@comp.polyu.edu.hk).

Yaxiong Xie is with the Department of Computer Science and Engineering, University at Buffalo, NY, USA (e-mail: yaxiongx@buffalo.edu).

This work has been submitted to the IEEE for possible publication. Copyright may be transferred without notice, after which this version may no longer be accessible.

interface requirements.

This approach is inspired by recent advances in large language models (LLMs) and agentic AI [18]–[21], which have exhibited the ability to interpret complex instructions and generate code from specifications. However, general-purpose LLMs (*e.g.*, GPT-4 [19]) may generate hallucinated APIs or interface logic inconsistent with required specifications [22]. Meanwhile, recent telecom- and O-RAN-specific LLMs have improved domain knowledge and code understanding [23], [24], while other LLM-based systems have explored intent translation and specification-aware testing in O-RAN [25], [26]. These studies demonstrate the value of incorporating O-RAN-specific knowledge and specification-aware validation into LLM-based O-RAN systems. However, these capabilities alone do not address the complete process of generating a deployable xApp from a high-level operator requirement. Such a process must identify the O-RAN information required by the requested function and target RIC/RAN platform, generate and validate the xApp function, synthesize the platform-specific interface layer, and validate the integrated xApp before deployment. Our key insight is to organize these requirements into a staged development pipeline that transforms high-level operator requirements into deployable near-RT RIC xApps.

Realizing this vision requires addressing several O-RAN-specific challenges that make naive LLM-based xApp generation unreliable. *xAppGen* addresses these challenges through a staged and domain-aware design.

❶ **Under-specified requirements:** Network operators specify high-level requirements for desired network functions or policies, while xApp developers translate these requirements into xApp implementations involving O-RAN interfaces, APIs, and service models. A natural-language request may not include all of the implementation details required for xApp development and can also be incomplete or ambiguous, causing an LLM to misinterpret the intended function [27]. To mitigate this issue, *xAppGen* uses a *requirement refinement and structuring* module that guides the user to specify key xApp details (*e.g.*, objectives, input metrics, policies). This structured template helps capture the intent and provides the LLM with a clear specification to work from.

❷ **Task- and platform-specific O-RAN knowledge requirements:** The O-RAN information required to generate a specific xApp varies with both the requested function and the target RIC/RAN platform. For example, different xApps may rely on different E2 service models, Key Performance Metric (KPM) semantics, platform APIs/SDKs, reporting scopes, and control interfaces. Having general O-RAN domain knowledge alone does not determine which of these specifications and platform capabilities should be used for a particular xApp, while missing or mismatched information can lead to incorrect API usage or interface logic [22]. *xAppGen* addresses this issue through an automated *knowledge retrieval* module, which retrieves relevant O-RAN specifications, platform information, and xApp development examples based on the refined user requirements and target deployment, and provides them as context to the LLM. This in-context knowledge injection [28] provides the subsequent function design and xApp synthesis stages with the information required for the requested task and platform.

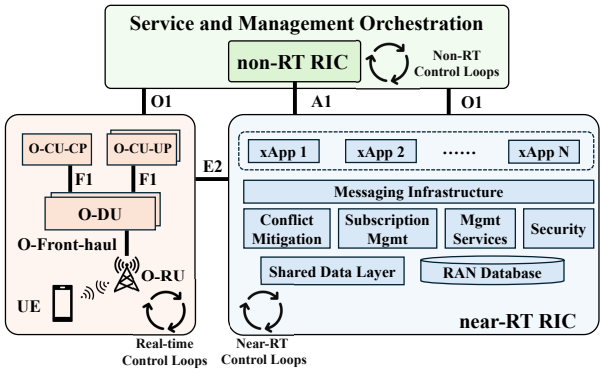


Fig. 2. O-RAN architecture.

❸ **Joint generation of xApp function logic and platform integration:** *xAppGen* separates xApp generation into task-specific function logic and platform integration. The function logic may be rule-based, optimization-based, or AI/ML-based depending on the requested monitoring or control task. Platform integration refers to the implementation required to connect the generated function to the target near-RT RIC/RAN environment, including RIC lifecycle and messaging, E2 subscription setup, KPM parsing, and control-message encoding according to the applicable service models [29], [30]. Generating the function logic and platform integration in a single LLM step is *error-prone*, since the LLM may mix task-specific algorithm requirements with O-RAN interface, service-model, and runtime requirements, causing functional or integration failures. *xAppGen* tackles this challenge with a *two-stage generation and validation process*: it first synthesizes and validates the standalone algorithmic *core function*, and then uses the validated core as context to integrate the platform-specific interface code. This staged approach separates the two generation tasks and enables *xAppGen* to produce a fully formed xApp that implements the requested function, satisfies the required O-RAN interfaces and target-platform constraints, and can be executed on the RIC without manual fixes.

We build *xAppGen* on a real-world 5G testbed using the open-source srsRAN stack [31]. We evaluate *xAppGen* on multiple representative xApp use cases, including anomaly detection, interference classification, and slice scheduling, which cover various network monitoring and control scenarios. Results demonstrate that *xAppGen* can generate functionally correct xApps that match or even exceed the effectiveness of the best manually developed ones. By eliminating most manual development effort, *xAppGen* dramatically accelerates the xApp development pipeline — a new xApp can be realized and rolled out in a matter of hours rather than weeks or months in the traditional paradigm. In summary, this paper makes the following key contributions:

- We present *xAppGen*, the first framework to automate xApp development using LLMs, transforming traditional manual processes into natural-language-driven workflows. This paradigm shift significantly reduces development effort and lowers the barrier for operators to create xApps.
- We propose a suite of xApp-specific techniques to adapt LLMs into xApp development agents, integrating task- and

TABLE I
TRADITIONAL RAN V.S. O-RAN

Aspects	Traditional RAN	O-RAN
Architecture	Monolithic, Integrated hardware/software	Disaggregated into modular components
Interface	Proprietary	Standardized
Vendor Ecosystem	Closed Single-vendor	Open Multi-vendor
Control Logic	Embedded in software stack	Realized through xApps
Upgrade Flexibility	Constrained by vendor-specific dependencies	High flexibility in upgrading components
Function Development	Black-box implementation	Software-defined Programmable

platform-specific O-RAN knowledge retrieval with structured generation and validation for complex RAN interfaces and control logic. These techniques bridge high-level operator requirements and platform-specific xApp implementations with standard O-RAN interfaces and runtime validation.

- We design, implement, and extensively evaluate *xAppGen* on a live O-RAN testbed. Across diverse use cases, the *xAppGen*-generated xApps demonstrate both strong functional performance and seamless deployment viability, achieving comparable or superior results to manually developed xApps while requiring minimal human effort.

II. BACKGROUND AND MOTIVATION

This section provides the background and motivation for *xAppGen*. We first introduce the basic architecture of O-RAN and explain how its open interfaces enable programmable RAN control. We then describe the role of the near-RT RIC and xApps, including how xApps interact with RAN nodes through standardized interfaces. Finally, we discuss why existing xApp development remains difficult and motivate automated, domain-aware xApp generation.

A. O-RAN v.s. Traditional RAN

Traditional RAN architectures are built on vertically integrated hardware and software stacks provided by a single vendor. While this approach simplifies system integration, it leads to limited flexibility, increased cost, and vendor lock-in. Thus, launching new features often necessitates hardware upgrade by vendors, which is costly and time-consuming.

Unlike traditional RANs that rely on monolithic, vendor-locked solutions, O-RAN promotes interoperability by disaggregating RAN into multiple units with distinct functionalities. In Fig. 2, the O-RU handles Radio Frequency (RF) signal processing and lower-PHY functions; the O-DU executes lower-layer protocol stack functions, including RLC, MAC, and higher-PHY functions; and the O-CU hosts higher-layer RAN protocol functions, including RRC and PDCP. Meanwhile, to enable agile and intelligent control, O-RAN separates RAN control intelligence into the near-RT RIC [32] and the non-RT RIC [33]. Operating at timescales ranging from 10 milliseconds to 1 second, the near-RT RIC enables dynamic control with xApps. Information exchange between the two RICs is supported via the A1 interface. By defining the E2 interface between the near-RT RIC and E2 Nodes [34] and other open interfaces between the units, O-RAN supports the interoperability of units from different vendors and offers network operators more unit options. TABLE I summarizes the key differences between traditional RAN and O-RAN.

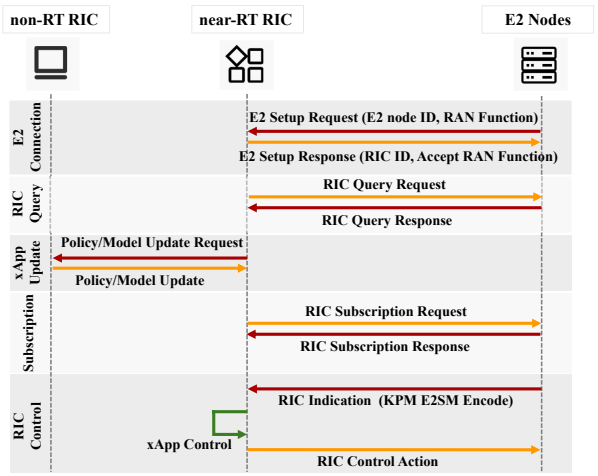


Fig. 3. xApp Workflow.

B. xApp Workflow

xApps enable fine-grained, near-RT control over O-RAN operations. These applications, hosted on the near-RT RIC, subscribe to specific telemetry streams (*e.g.*, network KPMs), execute control logic, and transmit decisions to underlying RAN units via the E2 interface. In this workflow, an E2 Node is a logical node that terminates the E2 interface and exposes RAN functions to the near-RT RIC. E2 Nodes include O-CU-CP, O-CU-UP, O-DU, and O-eNB. Each xApp follows a structured lifecycle including deployment, registration, optional RIC query, data subscription, decision execution, and control feedback. Specifically, upon deployment, an xApp registers its service capabilities and communication endpoints with the RIC. After E2 setup, the near-RT RIC may issue a RIC Query Request and receive a RIC Query Response to obtain RAN- or UE-related information from the E2 Node before or during subsequent subscription and control procedures. It then subscribes to measurements such as signal-to-noise ratio, handover events, or buffer occupancy, streamed in near-RT from E2 Nodes or other RAN nodes. The xApp processes the stream using control logic—typically rule-based policies or AI/ML models—and issues corresponding control actions, *e.g.*, adjusting scheduling priorities or triggering handovers. This closed-loop interaction enables responsive network control. In parallel, the non-RT RIC can influence xApp behavior via the A1 interface, delivering high-level policy directives, model updates, or intent-driven objectives. Fig. 3 shows the typical workflow of an anomaly detection xApp.

C. Current xApp Development Process

Unlike proprietary monolithic RANs, where control functions are implemented within vendor-specific network equip-

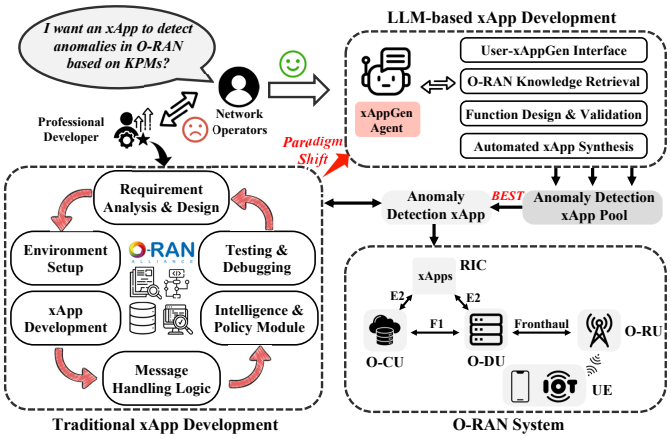


Fig. 4. Traditional xApp Development in O-RAN v.s. Automated xApp Development.

ment and access to these functions is limited by proprietary interfaces, O-RAN enables external xApps to program RAN behavior through open interfaces. This openness introduces a new development burden for xApp developers, who must navigate heterogeneous vendor units, constantly evolving specifications, RIC lifecycle management, O-RAN interfaces, service-model semantics, telemetry subscription, and control-message encoding. Developing a deployable xApp therefore requires more than designing the control algorithm itself. Developers must integrate the algorithm with the near-RT RIC runtime interfaces, configure E2 subscriptions, parse service-model-specific KPM reports, encode control messages according to the corresponding O-RAN service-model definitions, and validate the xApp against RIC lifecycle and latency constraints, all of which require deep domain expertise and familiarity with the O-RAN specifications (Fig. 4). As a result, practical xApp development is time-consuming (spanning weeks or months) even for experienced developers. For network operators, this not only delays innovation but also increases reliance on third-party developers, hindering agility in xApp development, raising concerns about user data, and increasing the deployment costs of new functionalities. These challenges highlight the pressing need for an agile xApp development paradigm.

D. Towards Automated xApp Development

Recent advances in LLMs (*e.g.*, GPT-4o [35], DeepSeek-Coder [36]) have substantially improved program synthesis and natural-language programming capabilities. By translating high-level user requirements into executable programs, LLMs offer an opportunity to reduce the effort required in traditional software development. This capability is particularly appealing for xApp development, where developers must bridge high-level network objectives and low-level RAN interfaces.

This motivates a natural question: can xApp development be shifted from manual function-level programming to intent-driven generation? Fig. 4 illustrates this transition from conventional xApp development to LLM-based xApp generation. In the traditional workflow, xApp development requires repeated effort in requirement analysis, environment setup, implementation, message-handling logic, intelligence and policy integration, and testing/debugging. In contrast, xApp developers or network operators may express their needs in natural language, and an AI agent can translate these high-level requirements into executable xApps. This shift from function-level programming (by professional xApp developers) to intent-driven xApp generation (initiated by either developers or network operators) has the potential to significantly streamline xApp development [37].

At the same time, recent studies have expanded the use of LLMs in telecom and O-RAN. Telecom-specific LLMs have been developed to improve domain knowledge and task performance [23], while other studies use LLMs to translate natural-language intents into O-RAN policies or management actions [25]. More recently, ORANSight-2.0 [24] develops O-RAN-specific foundation models and demonstrates the benefit of domain-specific knowledge for O-RAN knowledge understanding and code-related tasks. AI5GTest [26] further demonstrates that 3GPP and O-RAN specifications can be incorporated into LLM-based systems for automated testing and validation of O-RAN components.

These advances provide important evidence for automated xApp development. ORANSight-2.0 shows that O-RAN-specific knowledge improves an LLM's ability to handle O-RAN tasks, while AI5GTest shows the importance of specification-aware reasoning and validation when LLMs interact with O-RAN systems. They support two key requirements of *xAppGen*: the generation process should use O-RAN-specific information relevant to the target task, and the generated software should be validated against O-RAN and runtime requirements. However, automated xApp development requires these capabilities to be connected with additional steps, including operator requirement refinement, xApp function generation, platform-specific interface synthesis, deployment, and runtime execution.

Motivated by this gap, *xAppGen* integrates task- and platform-specific O-RAN knowledge retrieval, xApp function generation and validation, platform-specific interface synthesis, deployment, and runtime validation into an automated development pipeline that transforms a high-level operator requirement into a deployable near-RT RIC xApp. In the following, we present the design of *xAppGen* and its key technical components.

III. xAppGen Design

This section presents the design rationale and technical modules of *xAppGen*. Automatic xApp generation requires more than translating a natural-language prompt into code. It must transform an underspecified network intent into a deployable, platform-specific near-RT RIC xApp. To this end, *xAppGen* addresses four main challenges: incomplete user intents, task- and platform-specific O-RAN knowledge requirements, O-RAN-aware core-function generation and validation, and platform-specific xApp synthesis under deployment and runtime constraints. Accordingly, *xAppGen* is organized into four components, as shown in Fig. 5. The User-*xAppGen* Interface refines natural-language requests into structured xApp requirements; the O-RAN Knowledge Retrieval module retrieves O-RAN and platform information associated with the requested

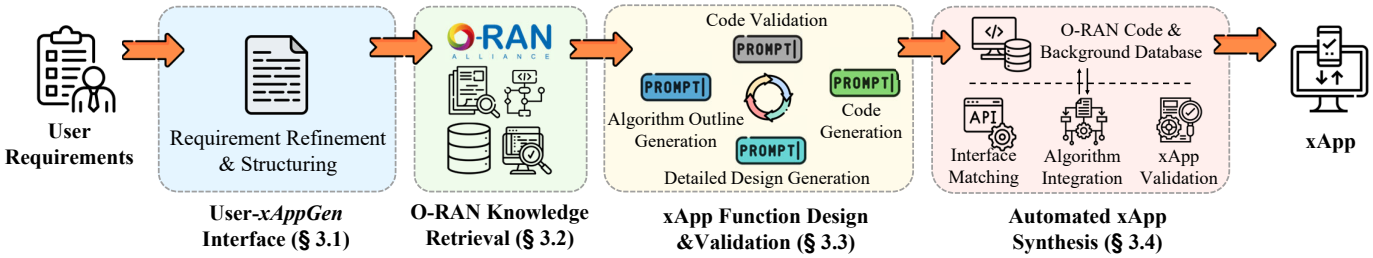


Fig. 5. *xAppGen* overview.

xApp and target deployment; the xApp Function Design and Validation module generates O-RAN-aware core functions and validates their algorithmic performance; and the Automated xApp Synthesis module performs interface matching, algorithm integration, and validation to produce deployable platform-specific xApps. This pipeline reduces manual xApp development effort and allows operators to specify new xApp functions at a higher level of abstraction.

A. User-xAppGen Interface

Traditional xApp development requires mastering detailed O-RAN interfaces, data streams, and APIs, posing a high entry barrier. *xAppGen* addresses this by providing a user-friendly interface where developers or operators specify desired functionalities in natural language. This design hides O-RAN complexity and enables non-experts to quickly develop and deploy xApps. Specifically, users first specify task requirements, operational objectives, and high-level control policies in natural language as "Design a Python-based xApp to detect anomalies in O-RAN based on KPM measurements". Such natural language specifications are intentionally designed to abstract away the details of implementation, allowing non-expert users to focus on expressing their intents.

Technical challenge: underspecified xApp intents. LLMs cannot generate high-quality code to meet user expectations when prompted with simple instructions [38]. The main reason is that user requirements are often unstructured and underspecified, and lack sufficient details for correct design and implementation [27], [39], [40]. Therefore, the corresponding design choice is to refine natural-language requests into structured xApp requirements before code generation. To tackle this challenge, we develop a *requirement refinement and structuring* module with a user requirement template specifically tailored for xApp development.

Requirement Refinement and Structuring. This module aims to progressively guide users to express structured requirements through multi-round interactions. *xAppGen* first parses the initial natural-language request using a lightweight intent-extraction module, which identifies key task components such as the functional objective (*e.g.*, anomaly detection or traffic classification), expected input modality (*e.g.*, structured telemetry or RF spectrograms), temporal resolution, and output format. The operator may optionally specify the target RIC/RAN platform or other available deployment information. When no such information is provided, *xAppGen* uses an O-RAN Software Community (OSC)-based reference profile [41]–[43] as the default generation target. The resulting xApp is generated for the selected reference profile and is not assumed to be portable across platforms.

If any fields are missing or under-specified, they are flagged as unresolved fields and trigger follow-up interactions. To handle ambiguous or incomplete input, *xAppGen* can engage users in follow-up dialogues, automatically generating targeted questions to elicit additional details—such as the specific type of anomalies, relevant data sources, required measurement granularity, or deployment information needed to resolve an interface ambiguity. To determine what information needs to be clarified, *xAppGen* maps the initial user input to a predefined *User Requirement Template* corresponding to the intended xApp type (*e.g.*, anomaly detection, interference classification, traffic classification). This template specifies essential fields such as the task objective, input modality, temporal resolution, and output format, ensuring that requirements are captured in a consistent and complete form.

B. O-RAN Knowledge Retrieval

To bridge the gap between user intents specified in user requirement templates and detailed algorithm design and implementation, *xAppGen* develops an *O-RAN knowledge retrieval* module, which gathers relevant knowledge from diverse sources (*e.g.*, O-RAN specifications) to facilitate accurate interpretation of user intents and xApp generation.

Technical challenge: selecting O-RAN knowledge for xApp generation. Automatically generating a deployable xApp requires concrete O-RAN knowledge, including service-model definitions, KPM meanings, RIC platform information, SDK usage, and existing xApp development examples. The key challenge is to identify the subset of this information that applies to the current xApp requirement and deployment environment. For example, the generated xApp must select KPMs available at the required reporting scope, use service models supported by the target RIC/RAN platform, and invoke the corresponding platform APIs and control interfaces. Incorrect selections may result in unsupported KPMs or service models, or interface logic inconsistent with the required O-RAN specifications or target-platform APIs. Therefore, *xAppGen* retrieves and selects task- and platform-specific O-RAN knowledge before algorithm design and xApp synthesis, and provides the selected information to the following generation stages.

Keyword Extraction. *xAppGen* first automatically identifies keywords from the standardized user requirements in the context of O-RAN. This is important for retrieving highly relevant domain knowledge. Naive methods suffer the risk of missing

essential contextual information or adding irrelevant noise [44]. For instance, extracting generic terms like "anomaly detection" without specifying the application domain may retrieve less relevant knowledge from other fields such as finance or healthcare. Conversely, extracting excessively fine-grained keywords may limit the flexibility and generalizability of the knowledge retrieval module. To address this challenge, *xAppGen* adopts a structured keyword extraction strategy that decomposes each keyword into two semantic fields: the *functional task* (*e.g.*, "anomaly detection", "traffic classification") and *target domain* (*e.g.*, "in O-RAN", "in near-RT RIC", "based on KPMs"). This two-part structure ensures that the extracted keywords reflect both the algorithmic intent and the deployment context. As shown in Prompt 1, the system explicitly instructs LLMs to balance specificity and generality by extracting phrases with optimal granularity—for example, preferring "anomaly detection in O-RAN" over either "anomaly detection" (too general) or "anomaly detection in O-RAN based on past hour KPMs" (which is too specific and can be revised to "anomaly detection in O-RAN" and "KPMs in O-RAN"). This trade-off ensures that the retrieved domain knowledge is both highly relevant and widely applicable across similar xApp tasks.

Prompt 1: Keyword Extraction

User Problem
<user input>...</user input>
Target
To understand the user problem, please identify key concepts that provide essential background knowledge.
Rules
Extract task-related and domain-related keyword phrases from the user problem. The keyword should contain two parts: the core task or function, and the domain context. Avoid generic or overly specific expressions.
Response Format
Term1, Term2, ...

Efficient Knowledge Storage and Retrieval. *xAppGen* first checks the local knowledge base for relevant technical repositories, O-RAN specifications, service-model definitions, SDK documents, and open-source xApp development libraries. If the required information has already been collected, *xAppGen* retrieves the corresponding documents from the local index. External web search is invoked only during initial knowledge-base construction or when the local repository lacks information for a new xApp requirement, target RIC platform, service model, or software version. To enhance efficiency and reliability, the search is scoped to a predefined list of selected authoritative sources (as in Prompt 2), ensuring that the retrieved content is both precise and relevant to the keyword and the development task.

The search results are then passed into the structuring and embedding pipeline for integration into local knowledge base. With various retrieved relevant information, *xAppGen* converts them into a structured format (*e.g.*, Markdown). Next, *xAppGen* uses an embedding model to encode available knowledge items into a dense vector space (*i.e.*, embeddings), enabling semantic-level retrieval beyond simple keyword matching. Finally, the embeddings are organized and stored in a local knowledge base, categorized according to content types such as O-RAN background and specifications, algorithm principles, performance optimization methods, and coding patterns. During the entire code generation process, *xAppGen* proactively retrieves relevant information from the knowledge base for reference. For instance, when a user requests for "anomaly detection in O-RAN", the knowledge base provides relevant information about common KPMs for the functionality in the O-RAN literature, best practices for AI/ML model design for xApps, and a variety of performance evaluation metrics.

Prompt 2: Knowledge Search

User Problem
<user input>{...}</user input>
Target
To enrich domain knowledge, please search for the fundamental definitions or background of given terms {...}.
Rules
Prioritize Wikipedia or official sites. Exclude implementation details. Filter out irrelevant content.
Response Format
URL1, URL2, ...

C. xApp Function Design and Validation

This module focuses on generating executable code for the xApp core function with automated validation. Given the structured user requirement and the retrieved O-RAN knowledge, *xAppGen* first translates the target O-RAN task into task-specific algorithmic logic, including telemetry feature selection, data preprocessing, model construction, decision logic, and evaluation metrics.

Technical challenge: O-RAN-aware core-function generation. The core function of an xApp must be consistent with the O-RAN task scenario and achieve reliable algorithmic performance. However, directly generating such a function is challenging because different xApp tasks rely on different telemetry modalities, KPM semantics, feature-processing steps, model choices, and optimization objectives. A generic algorithm may be executable, but it may use irrelevant inputs, miss important O-RAN-specific features, or perform poorly on the target task. Therefore, the corresponding design choice is to decompose the core-function design into explicit algorithmic steps and validate the generated function with task-relevant datasets.

This module is designed to produce high-quality, optimized algorithms tailored to fulfill user requirements, which is accomplished through a multi-stage prompting framework guided by Chain-of-Thought (CoT) reasoning strategies [45]. In particular, CoT reasoning is used to decompose the xApp core-function design into intermediate steps, including feature selection, preprocessing, model design, prediction or control logic, and evaluation. This decomposition helps *xAppGen* check whether the generated algorithm matches the O-RAN task context and provides checkpoints for iterative correction and performance improvement. The divide-and-conquer process has three key stages: algorithm outline generation, detailed design generation, and code generation with validation.

Prompt 3: Algorithm Outline Generation

User Problem
<user input>{...}</user input>
Target
To address the user problem effectively, please provide an algorithm outline step by step to solve the user problem based on the background information.
Rules
Provide a step-by-step algorithm design. Each step should have a clear goal and describe what actions will be performed. Focus on input/output relationship between steps.
Response Format
Step 1: [Title] ... ; Step 2: [Title] ... ; ...

Algorithm Outline Generation. Considering the high complexity of xApp algorithms, *xAppGen* is instructed to first generate a high-level outline of the target algorithm, decomposing the solution into a sequence of logical steps or functional modules. To ensure the relevance and clarity of the output, *xAppGen* employs a specially designed Prompt 3 that reiterates the user problem description, specifies the generation target, and enforces strict quality assurance rules for consistency and task alignment. For example, when a user specifies an anomaly detection task based on O-RAN KPMs, *xAppGen* generates an outline that begins with dataset loading and preprocessing, followed by feature selection, model construction, prediction logic, and evaluation using provided metrics (*e.g.*, accuracy, F1 score). Each step is designed to ensure input-output consistency, robustness to missing data, and compatibility with a standalone execution script.

Detailed Design Generation. Subsequently, *xAppGen* is further prompted to expand on each step in the outline by specifying concrete operations, data processing methods, feature selection strategies, and decision criteria. For example, in the context of anomaly detection, the LLM may recommend selecting E2SM-KPM measurements [46], such as PRB utilization, user throughput, or handover rates, as critical input features for training the detection model.

Prompt 4: Code Validation and Repair

User Problem
<user input>{...}</user input>
Target
To ensure the generated code is executable and robust, please analyze the provided output logs {...}, identify the problem, and modify the code to fix the errors if the compiler or interpreter cannot successfully run the code.
Rules
The following case is not allowed:
... (other functions remain unchanged)
... (same as before)

Code Generation with Validation. Once the detailed design is completed, *xAppGen* triggers the *Code Generation* module to translate each design specification into executable code segments. Then, *xAppGen* constructively integrates them into a comprehensive program (*i.e.*, the algorithm part of an xApp). In practice, however, the input data is collected from different O-RAN interfaces and protocols, exhibiting significant variability and lacking ground truth labels for model training. This limitation hinders *xAppGen* from automatically verifying the correctness of the algorithm and even improving its performance iteratively. To tackle this challenge, we propose to instruct *xAppGen* to first download public datasets corresponding to the user problem (*e.g.*, SpotLight [9] for anomaly detection). The dataset is then input into the generated algorithm for evaluation, and the error logs are recorded and fed back to *xAppGen* for iterative refinement, with the designed prompt shown in Prompt 4. Such a self-correcting loop produces stable and functional code with minimal human intervention. Additionally, *xAppGen* could generate multiple algorithm variants for the same task, each exploring different model architectures, feature engineering techniques, or parameter configurations. *xAppGen* could then select the one that achieves the best performance on the evaluation dataset across multiple metrics, which could be optimized to execute in parallel but comes at an increased xApp generation cost.

D. Automated xApp Synthesis

After validating the core algorithms and functional components with either public or local datasets, the final step is to integrate them into a deployable xApp. Unlike general-purpose program synthesis, O-RAN xApps must satisfy strict requirements on functional correctness, interface correctness, timing feasibility, and runtime stability. Conventional LLM-based code generation often stops once the code is syntactically correct or shows good offline accuracy, but *xAppGen* treats *validation* as a central objective and evaluates the correctness, robustness, timing behavior, and near-RT deployability of the generated control logic. Nevertheless, the complex nature of O-RAN platforms and dynamic working environments present significant practical challenges.

Technical challenge: deployable interface integration. One key challenge is *interface matching*, which requires xApps to use the applicable E2 interface procedures, E2AP message structures, and E2 service-model definitions (*e.g.*, E2SM-KPM for KPM reporting and E2SM-RC for RAN control) [47], and to encode control messages in corresponding formats. Any mismatch between the expected and available metrics–such as expecting per-UE throughput when only cell-level aggregates are reported–can result in functional failures. Moreover, *policy enforcement* presents another layer of complexity. xApps must respect dynamic control rules defined by the RIC, such as slicing constraints, control priorities, and resource allocation limits. Even if an anomaly is correctly detected, xApps must verify whether intended actions (*e.g.*, resource reallocation) are permissible under current policies. *Compatibility validation* is also essential to ensure that generated xApps satisfy runtime constraints such as configuration schemas, protocol versions, and execution-time bounds. For example, if the model inference of one xApp exceeds the 1-second near-RT latency requirement, it becomes inapplicable regardless of its correct logic or high accuracy. *These challenges collectively highlight the need for robust validation, adaptive interfacing, and policy-awareness mechanisms in the final deployment phase.*

Design choice: platform-specific synthesis and validation. To tackle these challenges, *xAppGen* constructs a platform-specific xApp template for the target RIC implementation, with OSC as the primary target. It does not assume that

one generated xApp can run unchanged across different RIC implementations. The OSC-specific xApp follows OSC runtime, messaging, subscription, and lifecycle conventions. For other RIC implementations, such as FlexRIC [48], *xAppGen* retrieves the corresponding SDK documents, service-model definitions, and sample xApps as workflow and interface references, and regenerates the platform-specific interface layer. The template contains the components required for runtime initialization, configuration parsing, E2 message handling, algorithm execution, optional A1 policy enforcement, and output delivery. The validated core algorithm is inserted into this template to process incoming KPMs or other measurements and produce monitoring or control outputs. When the selected RIC implementation supports the required control capability, the output is encoded and sent to the RAN through the corresponding platform interface. The final xApp is produced through three stages: interface matching, algorithm integration, and xApp validation.

Prompt 5: Interface Matching

User Problem
<user input>{...}</user input>
Target
To ensure the xApp communicates correctly with O-RAN components, please generate complete and specification-aligned interface functions according to the scaffold structure, retrieved domain knowledge, and user requirements.
Rules
- Refer to the knowledge base.
- The generated functions should follow the data formats and message structures defined in O-RAN specifications.
- All code must be complete and self-contained.

Interface Matching. This stage maps the input and output requirements of the generated algorithm to the measurements and control interfaces available in the target RIC/RAN deployment. Based on the selected or default RIC platform, *xAppGen* retrieves the relevant interface specifications and deployment information from the knowledge base. It then checks whether the measurements, reporting scope, service models, and control interfaces required by the generated algorithm are supported by the selected platform. For each algorithm input, *xAppGen* identifies the corresponding KPM, reporting granularity, and E2 service model. We use a few-shot in-context learning Prompt 5, which provides several representative interface-generation examples and instructs the LLM to generate interface-related code according to the retrieved knowledge and user requirements. *xAppGen* generates a platform-specific interface layer for the selected RIC platform, rather than producing one platform-independent xApp for all RIC implementations. When the available information is incomplete or ambiguous, *xAppGen* returns to requirement refinement and requests only the necessary clarification from the operator. If a required capability is unavailable, *xAppGen* uses an operator-approved alternative, when available, or reports that the requested functionality cannot be supported by the selected platform. These examples serve only as workflow and interface references for understanding registration, subscription, message flow, KPM reporting, and control-output delivery, while *xAppGen* regenerates the xApp implementation for each requirement rather than reusing or filling in an existing sample application.

Algorithm Integration. The algorithms and functional modules generated by *xAppGen* take the public or local datasets as input (§ III-C), rather than the real-world data streams reported via O-RAN interfaces. Thus, we need to modify and adapt the algorithms before integrating them into an xApp. Specifically, we design Prompt 6 that instructs *xAppGen* to replace the offline data loading and evaluation code with interface-driven input and output handling logic. This includes adapting input pipelines to consume KPMs from E2 messages, restructuring the output format to align with xApp control actions, and embedding the algorithm logic into a real-time loop.

Prompt 6: Algorithm Integration

User Problem
<user input>{...}</user input>
Target
To transform the algorithm into a deployable xApp component, please adapt the code to read real-time E2 KPM inputs, process them with the existing algorithm logic, and output actionable results to the RIC system using standardized control message formats.
Rules
- Refer to the knowledge base.
- Follow the previous code template {...}.
- Avoid any blocking operations.
- All code must be complete with no placeholders.

xApp Validation. To enhance the reliability of the integrated xApp, we further design an auxiliary function that supports essential tasks such as data parsing, logging, and configuration management. In addition to syntax checking, *xAppGen* uses SonarQube [49], a static code analysis tool, to inspect the generated source code before deployment. It detects code-level issues such as bugs, code smells, maintainability problems, security warnings, and risky API or data-type usage, and feeds the reported issues back to the repair loop. This process can effectively improve the reliability and maintainability of the generated xApp. However, it is important to note that the existing code analysis methods are limited to verifying code-level semantics and structural correctness. They do not assess the functional logic of the xApp. Specifically, these existing methods do not validate whether each interface is correctly implemented or real-time communication and control actions are performed as expected.

To address this limitation, we perform system-level validation in the experimental evaluation (§V-E), where generated xApps are deployed and executed on a real-world O-RAN testbed. This runtime validation checks whether the generated xApp can be successfully deployed and executed, whether the requested KPMs can be received and decoded at the expected scope, whether required control interfaces, if any, are available, and whether the execution latency satisfies the near-RT requirement. In addition, *xAppGen* performs runtime behavioral checks using known-input validation cases derived from collected KPM traces or task-specific test scenarios. The generated xApp is executed on these known inputs, and its outputs are checked against task-specific expectations, such as correctly detecting ground-truth anomalies while limiting false alarms and missed detections. If the validation exposes an un-

supported measurement, an unexpected reporting granularity, an interface mismatch, or inconsistent runtime behavior, the corresponding logs are fed back to the Interface Matching or Algorithm Design stage for correction. If the mismatch cannot be resolved without changing the intended functionality, *xAppGen* requests operator confirmation or reports that the functionality is unsupported in the selected deployment.

IV. Automated Deployment and Execution

We now present the xApp deployment and execution process in our main OSC-based implementation. The xApp execution module is tightly coupled with O-RAN, ensuring that the entire workflow from xApp generation to deployment can be validated end-to-end within the operational O-RAN environment.

Inter-unit Communication. The generated xApps are deployed on the near-RT RIC. These xApps communicate internally via RIC Message Router (RMR) [29] to handle inter-component messaging within the RIC software stack. The xApps communicate with other units in O-RAN via the standardized E2 interface and E2 Application Protocol (E2AP) [50]. In particular, the xApps receive real-time telemetry from base stations (*e.g.*, gNB) and transmit control messages to dynamically influence network behavior.

Automated xApp Deployment. The generated xApps are packaged as Docker containers following the OSC xApp deployment workflow and are deployed onto the RIC host. Deployment scripts include all necessary environment configurations to enable integration with platform services such as logging, status monitoring, and configuration management. Containerized xApps are then launched by the RIC orchestrator and automatically establish initial connections to the RMR, subscription manager, and internal databases.

Dynamic Registration Procedure. Once deployed, the xApps register their presence and functional capabilities with the E2 Service Manager of the RIC. This involves subscribing to specific metrics (*e.g.*, RLC buffer occupancy, PRB utilization, handover statistics) emitted by the base stations, depending on the functionalities of the generated xApps. For instance, in the case of an anomaly detection xApp, *xAppGen* ensures that only the relevant data streams are subscribed, minimizing telemetry transmission overhead.

Real-time Execution and Decision Logic. When the xApps operate in a closed-loop control mode, telemetry data collected via the E2 interface are continuously streamed into the xApps' processing engine. The core algorithms generated by *xAppGen* process these data streams in real time for various tasks such as anomaly detection, load balancing, and interference mitigation. Based on real-time processing results, control actions (*e.g.*, RAN parameter reconfiguration, traffic steering) are generated and sent back to the base stations via E2 to adapt network configurations.

Standard O-RAN Interface Integration. The generated xApps use standard O-RAN interfaces according to the relevant specifications and the capabilities supported by the selected RIC/RAN platform. To this end, *xAppGen* automatically generates task-specific E2 Service Model (E2SM) handlers, such as E2SM-KPM for KPM reporting and E2SM-RC for RAN control messaging. The generated interface logic follows the corresponding message formats, telemetry structures, and control-interface definitions. We further validate the resulting xApps through deployment and runtime communication checks on the evaluated platforms. These checks demonstrate interface and runtime compatibility under the tested configurations, but do not constitute formal O-RAN conformance testing or certification. This also does not assume source-level portability across different RIC implementations.

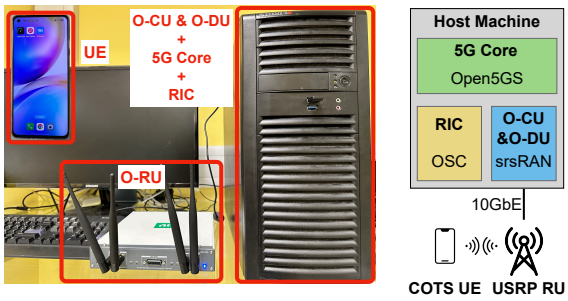


Fig. 6. Testbedding with SDR and COTS UE.

V. Implementation and Evaluation

In this section, we introduce the implementation and evaluation of *xAppGen*, aiming to answer the following key research questions: (1) Can *xAppGen* generate xApps that satisfy the specified task objectives and achieve task-level performance comparable to manually developed baselines? (2) Can *xAppGen*-generated xApps be automatically deployed and executed on real-world O-RAN platforms with near-RT control loop constraints? (3) How does each novel technical module contribute to the overall performance of *xAppGen*?

A. Experiment Setup

Hardware. We implement *xAppGen* on the software-defined srsRAN [31] stack. As shown in Fig. 6, we use a Ubuntu 22.04.1 LTS workstation with an Intel Xeon(R) Core E5-2620 v4 CPU, 32 GB RAM, and USRP X310 units to serve as the gNodeB. We run Open5GS [51] on the workstation as the core network. User equipments (UEs) include OnePlus 8T and Xiaomi 13 Pro smartphones with programmable SIM/USIM cards (sysmoISIM-SJA2 SIM cards [52]). For control-plane support, we use the OSC near-RT RIC as the main xApp execution platform to host generated xApps and manage near-RT RAN control. In this testbed, srsRAN provides the RAN-side gNB/E2-node functionality, while the generated xApps run on the OSC near-RT RIC and interact with the srsRAN-based RAN stack through KPM subscription, KPM reception/parsing, and monitoring or control execution.

Software. We use GPT-4o [19], [35] as the default LLM and LangChain [53] as the orchestration framework for prompt management, module integration, and tool wrapping. We implement all backend logic in Python 3.10 and utilize libraries such as OpenAI SDK, FAISS [54], and FastAPI [55]. Specifically, LangChain coordinates prompt templates, LLM calls, retrieval utilities, and validation tools across different generation modules; FAISS provides embedding-based similarity search over the local O-RAN knowledge base; and FastAPI

TABLE II
ACCURACY COMPARISON WITH SPOTLIGHT [9]

Method	Metric	MAC	NETWORK	PDCP	RADIO	MIXED
xAppGen	Precision	97.3%	98.9%	92.1%	78.8%	97.6%
	Recall	97.5%	98.9%	91.8%	81.5%	97.6%
SpotLight	Precision	93.6%	94%	100%	95%	95.5%
	Recall	100%	92%	93%	93%	94.5%
Z Score	Precision	74.4%	75%	69.2%	58.8%	65.4%
	Recall	82.4%	83.6%	69.4%	59.7%	79.2%
LSTM	Precision	74.5%	65.2%	73.5%	92.05%	69.5%
	Recall	6.3%	37.6%	7.1%	54.2%	10.5%

exposes lightweight REST APIs for user interaction, module invocation, and backend service integration. SonarQube [49] is used for static code analysis.

B. Open-Source Dataset Performance

We compare *xAppGen* with three representative baselines to evaluate whether *xAppGen*-developed xApps can satisfy different user requirements across input types and xApp tasks. The three baselines are organized around two evaluation settings.

Anomaly detection under different input pathways. SpotLight [9] and MobiWatch [10] share the same high-level objective of anomaly detection, but rely on different input pathways. SpotLight uses fine-grained RAN KPM telemetry, where the generated xApp selects representative measurements such as PRB utilization, throughput, latency, packet loss, and RSRP from the available E2SM-KPM telemetry according to the required reporting scope. In contrast, MobiWatch uses link-layer and session-layer signaling traces rather than E2SM-KPM counters. This setting evaluates whether *xAppGen* can generate functionally similar xApps under different developer requirements on input data.

Generalization to radio-layer interference classification. IC [56] represents a different radio-layer task, i.e., interference classification, and is used to examine whether *xAppGen* can generalize beyond anomaly detection to another xApp function and data modality. For the KPM-based IC setting, the generated xApp selects radio-link measurements such as SINR, BLER, MCS, throughput, and PRB utilization from the available E2SM-KPM telemetry according to the required reporting scope.

For fair comparisons, we use the same dataset (*i.e.*, the one that each baseline uses) for evaluation. For *xAppGen*-generated xApps, the datasets are used in the algorithm validation loop to evaluate and refine the generated algorithms, rather than to fine-tune the LLM. To ensure a fair comparison, we use the same fixed 4:6 train/test split for *xAppGen* and the baseline algorithms, and report all final results on the testing set.

1) Evaluation Metrics: We adopt two sets of metrics:

***xAppGen*-generated xApp Evaluation.** (1) *Precision*, *Recall Rate* and *F1 Score* are the primary indicators of model correctness. (2) *VRAM Usage* evaluates the runtime efficiency of the generated AI/ML algorithms.

***xAppGen* Evaluation.** (3) *Synthesis Time* quantifies the total duration from the moment a user requirement is input to the completion of xApp generation; (4) *Number of Bugs* assesses the code quality and the robustness of *xAppGen*s xApp generation. Each syntactic or semantic error caught during compilation or testing is counted as a bug; (5) *Iteration-to-Success Count* measures the correction cycles required to synthesize a fully functional program; (6) *One-Shot Success Rate* measures the proportion of xApp generation tasks that can be completed successfully in the first attempt, from the operator's high-level intent to a deployed and executable xApp, without additional repair or regeneration iterations.

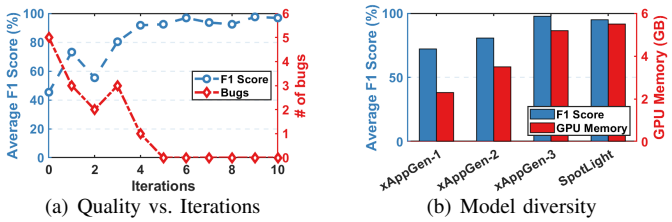
2) Baseline 1 - SpotLight for Anomaly Detection: SpotLight collects a comprehensive 5G O-RAN dataset, containing over 100 million KPM datapoints sampled at 100ms resolution. The dataset covers 600+ metrics across MAC, RLC, PDCP, Radio, and platform components, and captures both synthetic and real-world anomalies such as CPU contention and fronthaul congestion. SpotLight adopts a two-stage distribution learning method (*i.e.*, JVGAN+MRPI) to detect the anomalous behaviors of RAN KPMs. For fair comparison, we leverage *xAppGen* to synthesize an anomaly detection xApp based on the user requirements. We also report the performance of a typical Z-score detection algorithm and an LSTM-based autoencoder for comparison.

Results. Table II lists the precision and recall rate on five KPM subsets: MAC, Network, PDCP, Radio, and Mixed, corresponding to five different O-RAN anomalies. We observe that *xAppGen*-developed xApps achieve higher precision and recall rates across all subsets. Surprisingly, it surpasses SpotLight in several subsets, especially when using Mixed KPMs that aggregate multiple cases. Further analysis of the generated code reveals that *xAppGen* can automatically select and preprocess the most relevant KPMs, while discarding noisy or weakly correlated knowledge during prompt construction. *xAppGen* also applies task-specific filtering strategies during data preparation, leading to more robust features and better generalization across anomaly types.

To examine the model iteration process of *xAppGen*, we analyze both accuracy and code quality across multiple design and validation cycles. As shown in Fig. 7(a), the generated xApps achieve higher accuracy and fewer bugs after iterative refinement. On average, the initial implementation contains approximately five bugs per trial across 25 independent runs, mainly due to incomplete logic, improper data preprocessing, and incorrect control flow. Notably, after two to three iterations, the number of bugs decreases significantly, while the accuracy also increases. This shows that the code improvement module can use prior errors and system feedback to improve both functional correctness and task performance.

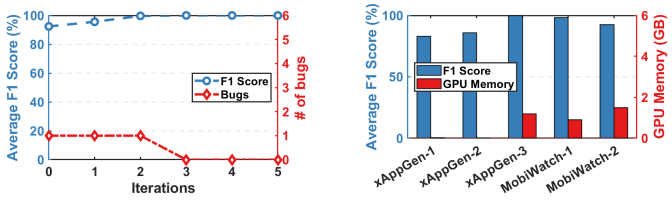
Beyond accuracy, *xAppGen* also supports interactive customization. As shown in Fig. 7(b), the three generated variants (*i.e.*, *xAppGen-1* to *xAppGen-3*) differ in feature selection, model architecture, accuracy, and VRAM usage. This allows operators to choose an xApp variant according to available hardware resources and performance requirements.

3) Baseline 2 - MobiWatch for Anomaly Detection: MobiWatch monitors link-layer and session-layer (RRC and NAS) messages using MobiFlow [10], a telemetry pipeline that transforms packet traces into structured signaling features. The dataset includes both benign and adversarial traces. Mobi-



(a) Quality vs. Iterations (b) Model diversity

Fig. 7. Performance of *xAppGen*-developed anomaly detection xApp on SpotLight dataset: (a) quality improvement across iterations; and (b) xApp variants.



(a) Quality vs. Iteration (b) Model diversity

Fig. 8. Performance of *xAppGen*-developed anomaly detection xApp on MobiWatch: (a) quality increment across iterations; and (b) diverse xApp variants.

Watch implements two versions of anomaly detection using Autoencoders [57] and LSTM [58], noted as MobiWatch-1 and MobiWatch-2, respectively.

Results. Table V reports the average accuracy, precision, and recall of both MobiWatch and *xAppGen*-developed xApps. Since the input feature space of MobiWatch is more structured and lower-dimensional than the heterogeneous KPM telemetry used in Baseline 1, the accuracy of *xAppGen* significantly outperforms MobiWatch on the same dataset. Fig. 8(a) illustrates the model iteration process. We observe that for simpler tasks, *xAppGen* can iterate more quickly to a satisfactory model with <3 iterations. In addition, *xAppGen* provides various xApp options as shown in Fig. 8(b). Users can choose simple models (*i.e.*, *xAppGen-2* without GPU usage) to achieve comparable performance.

4) Baseline 3 - IC for Interference Classification: This dataset was collected in [56] using a real srsRAN-based O-RAN testbed. It includes two modalities: 10,000 spectrograms (128×128 grayscale) recorded over the air from USRP-based receivers, and over 25,000 uplink KPM traces including SINR, BLER, MCS, and throughput metrics. Note that half of the samples correspond to clean transmission, while the other half represent continuous-wave jamming scenarios.

Results. We generated two versions of *xAppGen*-IC xApps using either the spectrogram data (InterClass-Spec) or KPM subsets (InterClass-KPM). As shown in Table III, *xAppGen* achieves detection accuracies of 92.9% and 98.8% on the two respective subsets, matching the performance of the CNN and DNN models proposed in the baseline paper [56]. However, we observed significant performance variance across different generated versions: some models achieve less than 80% accuracy, while others exceed 95%. The underlying reasons are twofold. Unlike time-series KPM data, spectrograms are high-dimensional images, which require more complex algorithms for effective and robust data preprocessing. In addition, inappropriate selections of AI/ML model architectures and hyperparameters from the vast configuration space can lead to unstable performance [59]. To tackle this issue, we may further integrate architecture-level reasoning and guide search strategies into the generation process. Overall, the experiment results demonstrate that *xAppGen* is capable of generating xApp functions for radio-layer tasks by adopting more complex image-based AI/ML models.

TABLE III
ACCURACY COMPARISON WITH IC [56]

Method	Type of xApps (Dataset)	Accuracy
xAppGen	InterClass-Spec	92.9%
	InterClass-KPM	98.8%
IC [56]	InterClass-Spec	98%
	InterClass-KPM	97.9%

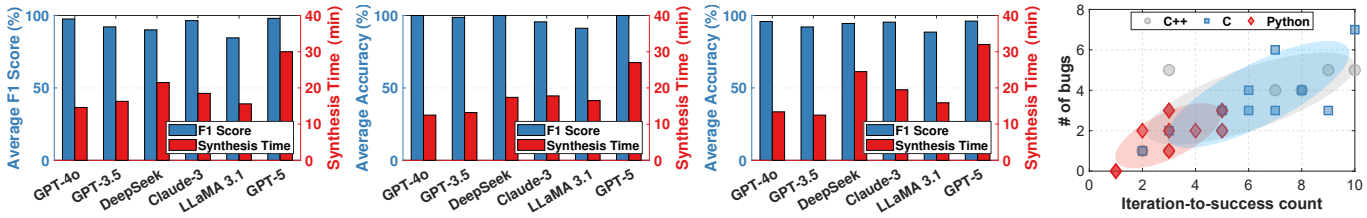
C. Key Impact Factors

Beyond task-level performance, we further examine key factors that affect the quality, efficiency, and deployability of *xAppGen*-generated xApps, including LLM choice, programming language, generation cost, and target RIC implementation. These analyses complement the task-level accuracy results and provide process-level evidence on the practical use of *xAppGen* for automated xApp development.

1) Impact of LLM Choice: We test six LLMs for comparison: GPT-4o [19], [35], GPT-3.5 [60], DeepSeek-R1 [61], Claude 3 [62], LLaMA 3.1 (70B) [63] and GPT-5 [64]. Each model was used to independently generate xApps, which were subsequently evaluated in terms of accuracy/F1 score and total synthesis time. As shown in Fig. 9(a), the xApp generated via GPT-4o consistently achieves strong performance across all datasets, while GPT-5 delivers nearly identical accuracy but with longer synthesis time. This superior performance stems from three key factors: (1) stronger chain-of-thought reasoning, leading to better decomposition of high-level xApp logic; (2) more semantically aligned code with fewer hallucinations or misinterpretations of O-RAN domain terms; and (3) more modular and syntactically complete outputs, reducing post-generation correction.

Moreover, GPT-4o-based *xAppGen* requires less than 20 minutes to generate most xApps, while GPT-5 achieves comparable accuracy with longer wall-clock synthesis time. Other models show greater variability, often producing verbose code, relying on overly generic patterns, or omitting low-level implementation details. As shown in Fig. 9, LLM performance also varies across tasks: structured KPM-based anomaly detection is generally easier to synthesize, whereas tasks involving fine-grained telemetry or high-dimensional spectrogram inputs are more challenging. For Baseline 2, GPT-3.5 and DeepSeek-based variants often misinterpret the input schema, leading to accuracy drops of 10–15% compared with GPT-4o and GPT-5. For Baseline 3, models that fail to generate valid spectrogram preprocessing stages perform substantially worse. These results show that xApp synthesis quality depends on both the input modality and the LLM's ability to align generated code with O-RAN-specific structures and constraints.

2) Impact of Programming Language: We explicitly prompted *xAppGen* to synthesize functionally equivalent



(a) *xAppGen*- Baseline 1 (b) *xAppGen*- Baseline 2 (c) *xAppGen*- Baseline 3 (d) Programming language

Fig. 9. Impact of different LLM and programming languages on *xAppGen*-generated xApps.

xApps in Python, C, and C++. Each version was tested across 10 independent trials using the same requirements and input dataset. We then recorded the cumulative bug count before the first successful run and the iteration-to-success count to assess the code reliability and structural difficulty. As shown in Fig. 9(d), Python xApps exhibit higher efficiency and concise structures, averaging only 1.8 bugs per version and requiring only 3 iterations to achieve a fully functional implementation. This advantage can be attributed to Python's high-level syntax, which resembles natural language compared to C and C++. Its concise, human-readable structure enables LLMs to more effectively translate high-level user requirements into functional code [65]. In contrast, C and C++ based xApps exhibit more bugs – often due to errors in memory management, pointer arithmetic, and data type handling – and require more code refinement iterations to converge.

3) Generation Cost: We also estimate the generation cost of *xAppGen* under the GPT-4o pricing [66] used in our experiments. Based on our multi-stage prompt workflow and the generation/repair process, generating one usable xApp consumes roughly 1.0M–2.5M tokens in total, corresponding to approximately $7–$15 per usable xApp. This cost varies with xApp complexity, since more complex xApps typically involve larger context inputs, longer generated code, more validation logs, and additional repair iterations.

4) Multi-target Generation and RIC Communication Validation: To clarify the portability boundary and runtime communication capability of *xAppGen*, we conduct a multi-target generation case study based on the same high-level requirement, which is to generate a KPM-based anomaly detection xApp. We instantiate this requirement for two target RIC implementations, OSC and FlexRIC[1], and evaluate whether the generated xApps can preserve the same algorithmic functionality while adapting to different RIC runtimes and interfaces. Both implementations use E2SM-KPM (RAN Function ID 2), with Report Styles 1 and 4 and a 1000-ms reporting period. The subscribed KPMs include `RRU.PrbUsedDl` and `RRU.PrbAvailDl`, which are used together to derive PRB utilization rather than directly using `RRU.PrbTotDl`, as well as `DRB.UEThpDl`, `DRB.RlcSduDelayDl`, and `DRB.RlcPacketDropRateDl`.

Table IV reports both algorithm-level performance and platform-level communication validation results. We do not claim formal O-RAN conformance certification. Instead, we perform interface and runtime validation checks to verify whether the generated xApps can complete the runtime workflow required by the target RIC implementation, including deployment, RIC-side registration, subscription setup, KPM reception, KPM parsing, alarm/control-output generation, end-to-end execution, and runtime trace inspection. Passing these checks demonstrates interface and runtime compatibility with the tested RIC implementations and configurations; it does not constitute formal O-RAN conformance testing or certification. Since the two xApps implement the same anomaly detection objective, they achieve the same F1 score, while their execution-time ranges and interface layers differ due to platform-specific runtime and messaging logic. Both generated xApps pass the corresponding communication and runtime validation checks. These results show that *xAppGen* maps the same high-level requirement to different platform-specific xApp implementations by regenerating the platform-dependent interface layer, rather than relying on source-level portability of a single xApp.

[1]The generated OSC-specific and FlexRIC-specific xApps are publicly available at https://anonymous.4open.science/r/xAppGen/.

TABLE IV
MULTI-TARGET GENERATION AND RIC COMMUNICATION VALIDATION RESULTS FROM THE SAME HIGH-LEVEL REQUIREMENT.

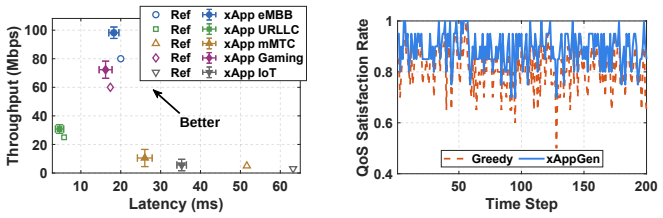
Metric	OSC-specific xApp	FlexRIC-specific xApp
Performance (F1 score)	97.6%	97.6%
Execution time	20–110 ms	18–95 ms
Deployment success	✓	✓
RIC-side registration	✓	✓
Subscription setup	✓	✓
KPM reception	✓	✓
KPM parsing correctness	✓	✓
Alarm/control-output generation	✓	✓
End-to-end execution	✓	✓
Runtime trace inspection	✓	✓

TABLE V
ACCURACY COMPARISON WITH MOBIWATCH [10]

Method	Dataset	Accuracy	Precision	Recall
xAppGen	Benign	100%	100%	N/A
	Attack	100%	100%	100%
MobiWatch-1	Benign	93.23%	93.23%	N/A
	Attack	100%	100%	100%
MobiWatch-2	Benign	91.15%	91.15%	N/A
	Attack	95%	88.68%	100%

D. Private & Domain-Specific Performance

To further assess the generalizability and practical utility of *xAppGen*, we evaluate a slice scheduling xApp, which requires real-time radio resource allocation under multi-slice QoS constraints. Since public datasets for operator-specific slice



(a) Latency vs. throughput per slice (b) QoS satisfaction over time

Fig. 10. Performance of *xAppGen*-generated slice scheduling xApp.

scheduling are unavailable, we construct a synthetic dataset that emulates per-slice telemetry and QoS policies based on standardized O-RAN interfaces. This also avoids potential memorization effects from public samples and provides a controlled setting for evaluating the generalization capability.

We extend standard service types defined by 3GPP [67] to include five representative slice categories, where each class embodies distinct QoS objectiveseMBB emphasizes sustained high throughput, URLLC targets ultra-low latency, mMTC prioritizes energy-efficient massive connectivity, Gaming demands low latency with stable jitter, and IoT focuses on reliable transmission under low data-rate conditions. These slice definitions are instantiated across the O-RAN stack, influencing A1 policy configurations, guiding E2 node telemetry reporting, and serving as templates for SMO-driven service orchestration. To support evaluation, we construct a synthetic dataset that closely replicates slice-level telemetry patterns observed in operational O-RAN deployments.

The telemetry generation process follows the standardized E2SM-KPM service model and captures a set of representative per-slice metrics, including Physical Resource Block (PRB) utilization, end-to-end latency, throughput, packet loss, Reference Signal Received Power (RSRP), channel quality, modulation and coding information, and buffer status. Slice-specific QoS targets and relative priorities are derived from A1 policy profiles, enabling the dataset to encode both observed performance states and policy-driven intents. For the generated slice scheduling xApp, *xAppGen* produces radio-resource allocation decisions that identify the target slice and specify its PRB allocation ratio or resource quota. The xApp therefore belongs to the radio-resource allocation control type. During Interface Matching, these control outputs are mapped to the E2SM-RC control style and attributes supported by the selected RIC/RAN platform [68], according to the corresponding service-model definitions and platform capabilities. E2SM-RC defines several Control Styles for different RAN functions, including Style 1 for Radio Bearer Control, Style 2 for Radio Resource Allocation Control, Style 3 for Connected Mode Mobility Control, Style 4 for Radio Access Control, and Style 5 for Dual Connectivity Control. The generated slice scheduling xApp uses Style 2 (Radio Resource Allocation Control), with Action ID 6, Control Header Format 1, and Control Message Format 1. Its control message identifies the target slice using PLMN and S-NSSAI (SST and SD) and specifies the minimum, maximum, and dedicated resource quotas. Hence, *xAppGen* does not hard-code a universal E2SM-RC style or attribute set across different platforms. To enhance realism, we calibrate value ranges, inter-metric correlations, and noise models using empirical trends reported in prior measurement studies. In particular, SpotLight provides insights into PRB allocation and interference patterns, while MobiWatch reports user mobility and throughput dynamics. These studies guide the construction of slice-level telemetry patterns for the synthetic dataset.

We evaluate the *xAppGen*-generated slice scheduling xApp by jointly examining its effect on per-slice throughput/latency trade-offs and on the temporal evolution of QoS satisfaction. As shown in Fig. 10(a), each marker pair represents the Requirement (Ref) and the *xAppGen*-generated xApp for the same slice type. Across all five slice categories, the xApp consistently shifts the operating point towards the top-left region, indicating simultaneous throughput gains and latency reductions. URLLC slices, in particular, exhibit the largest latency improvement, confirming that *xAppGen* can generate control logic that prioritizes delay-sensitive services while preserving throughput. Fig. 10(b) further presents the QoS satisfaction rate over time under two scheduling strategies: a greedy heuristic baseline and *xAppGen*. While the greedy approach achieves only moderate compliance and fluctuates heavily with traffic changes, the *xAppGen*-generated xApp maintains consistently higher satisfaction rates by dynamically reallocating PRBs in response to real-time slice performance feedback, ensuring robust QoS across heterogeneous and time-varying loads.

E. Real-World Execution Performance

To distinguish dataset-level algorithm validation from full-stack runtime testing, we further conduct integration and runtime experiments on the srsRAN-based O-RAN testbed. The public and synthetic datasets used in previous evaluations mainly assess the core algorithmic performance of generated xApps, such as anomaly detection accuracy and slice scheduling effectiveness. In contrast, this subsection evaluates whether the generated xApps can be onboarded into the OSC near-RT RIC, subscribe to KPM streams exposed by the srsRAN-based RAN stack, receive and parse KPM messages, and execute their monitoring or control logic under near-RT runtime constraints. For each representative function (two anomaly detection applications and one slice scheduling application), we automatically generate three distinct algorithmic variants of the xApp, denoted as *xAppGen*-1/2/3, and onboard them into the near-RT RIC platform via standard integration procedures. In addition to measuring execution latency, we also inspect whether the generated xApps produce expected runtime outputs under known input traces.

Execution Time. We measured the end-to-end execution time of the complete control loop, encompassing KPM stream reception, model inference, and control policy execution. As shown in Fig. 11(a,b), the two anomaly detection xApps exhibit execution times ranging from 20110 ms and 5590 ms, respectively. The slightly higher latency of the second baseline is attributed to its more intensive telemetry analysis. For slice scheduling (Fig. 11(c)), execution times span 250420 ms, reflecting the added complexity of multi-slice QoS optimization. These results demonstrate that *xAppGen*-generated xApps

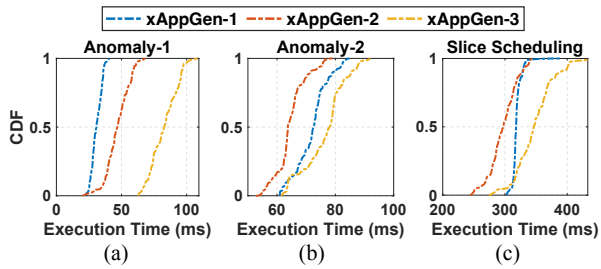


Fig. 11. Execution time of *xAppGen*-generated xApps on real-world testbed.

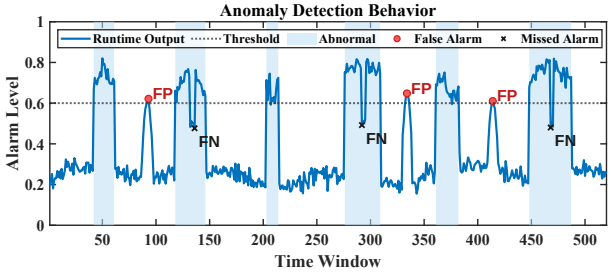


Fig. 12. Runtime behavior of the generated anomaly detection xApp.

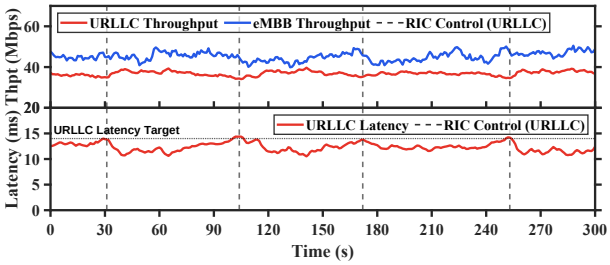


Fig. 13. Runtime behavior of the generated slice-scheduling xApp

TABLE VI
ABLATION STUDY (REQUIREMENT REFINEMENT AND STRUCTURING, KNOWLEDGE RETRIEVAL, FUNCTION DESIGN, AND VALIDATION.)

Method	One-Shot Success Rate		Iteration-to-Success Count	
	Baseline 1	Baseline 2	Baseline 1	Baseline 2
w/o R. (§ III-A)	0.88	0.84	8	3
w/o K. (§ III-B)	0.76	0.80	14	5
w/o F. (§ III-C)	0.90	0.80	8	3
w/o V. (§ III-C)	0.84	0.92	10	6
xAppGen	**0.92**	**0.96**	**5**	**3**

are deployable within the sub-second latency constraints of near-RT RIC control loops, even when accommodating more sophisticated control tasks.

Runtime Behavioral Check. Beyond execution time, we further examine the runtime behavior of the generated xApps through two representative tasks. For anomaly detection, we use collected KPM traces with ground-truth anomalies to examine whether the xApp correctly detects anomalies while keeping false alarms and missed alarms at a low level. As shown in Fig. 12, the generated xApp raises clear alarm outputs for most ground-truth anomalies, while the few false alarms and missed alarms are explicitly marked. For slice scheduling, we further evaluate the closed-loop control behavior of the generated xApp using live KPM measurements. Based on per-UE downlink throughput and configured slice requirements, the xApp dynamically determines PRB resource quotas and sends control requests to the E2 Node through E2SM-RC Style 2. As shown in Fig. 13, the vertical dashed lines indicate actual control requests, while the subsequent throughput and latency traces illustrate the resulting network behavior. These experiments demonstrate that the generated xApps can perform both runtime monitoring and network control.

F. Ablation Study

We conducted ablation studies to separately evaluate the contribution of each technical module. Specifically, we consider four key modules: *Requirement Refinement and Structuring*, *Background Knowledge Retrieval*, *xApp Function Design*, and *Code Validation*. In each ablation setting, we disabled one module while keeping the others intact, then generated 25 xApps per dataset and recorded two metrics: the one-shot success rate and the iteration-to-success count.

The results shown in Table VI demonstrate that the key modules collectively achieve the robust and reliable xApp synthesis with high performance and distinct contributions. The ablation of any single module results in a noticeable degradation in the overall performance. Specifically, a significant drop occurs when the *Background Knowledge Retrieval* module is excluded. Without this module, *xAppGen* cannot retrieve and select the task-relevant O-RAN information needed during generation, such as KPM definitions, optimization heuristics, and algorithm references, and instead relies on the knowledge already available to the LLM. As a result, the generated xApps tend to adopt simplistic algorithms with generic logic flow, leading to a one-shot success rate below 80%. We also observe a substantial increase in the number of code refinement iterations.

In addition, the absence of the *Requirement Refinement and Structuring* module also causes poor performance. Without it, user inputs are directly passed to LLMs in an ambiguous and unstructured form, hampering *xAppGen*'s ability to accurately interpret user intents. Consequently, the generated xApps often produce misaligned outputs with incorrect formats. Worse still, when the *xApp Function Design* module is disabled, *xAppGen* skips important intermediate reasoning steps without CoT prompting. This results in code that appears to be syntactically correct but often suffers from logical flaws and incomplete functionality implementations. Lastly, the ablation of *Code Validation* module would disable the automated syntax checking and static analysis before final output, which leads to more immediate runtime failures.

In summary, these findings emphasize that the complete pipeline architecture of *xAppGen* is essential to achieve consistent performance for generating functional xApps. Such a complementary combination of all technical modules allows *xAppGen* to reliably generate executable xApps across diverse O-RAN control scenarios.

VI. RELATED WORK

xApp Development. O-RAN promotes openness and modularity, enabling flexible integration of heterogeneous components across the RAN. Numerous studies [69]–[76] have explored various aspects of O-RAN, but xApp development [29] remains cumbersome due to system complexity and evolving

specifications [77]. To facilitate deployment efficiency and multi-service support, OREO [15] formulates xApp orchestration as a multi-service deployment problem and enables xApp sharing. Spotlight [9] focuses on functionality design and proposes a telemetry-driven xApp for explainable anomaly detection. In control intelligence, DRL-based xApps [78] adopt deep reinforcement learning to allocate O-RAN resources and meet traffic and slicing demands. Recent work [79] explores natural-language-based intent translation for O-RAN slice control, but it focuses on simple rule-based control and does not support code synthesis or validation. In contrast, *xAppGen presents an end-to-end automation framework for agile xApp development*, which allows developers or even network operators to quickly transform high-level business ideas and into readily deployable xApps and thereby drastically shorten the xApp development cycle.

LLMs for Code Generation. By exploiting the remarkable language understanding and processing capabilities of LLMs, numerous intelligent applications have been developed, such as mobile task automation [80], [81] and IoT data interpretation [82], [83]. LLM-based code generation [84]–[87] automatically translates natural-language user requirements into executable programs. A pioneering work MapCoder [86] decomposes the code generation process into four stages—retrieval, planning, coding, and debugging—each orchestrated by LLM prompts. Recent AIoT-oriented systems, such as AutoIoT [84] and GPIoT [85], further show that natural-language programming can be adapted to domain-specific IoT and AIoT application development through intent refinement, knowledge retrieval, code synthesis, and iterative validation. Recent studies have also explored LLMs specifically for O-RAN. ORANSight-2.0 [24] demonstrates that O-RAN-specific model training can improve domain knowledge and code-related capabilities. AI5GTest [26] shows that explicit 3GPP and O-RAN specifications can be incorporated into LLM-based systems for automated testing and validation. These results reinforce two observations that motivate *xAppGen*: O-RAN-specific knowledge is important when LLMs operate on O-RAN tasks, and specification-aware validation is necessary when generated outputs interact with O-RAN components.

Building on these observations and the natural-language programming paradigm demonstrated by prior AIoT systems, *xAppGen* addresses the next step of automating xApp development. O-RAN xApps must interact with the near-RT RIC, subscribe to E2/KPM telemetry, satisfy A1 policy and E2 interface constraints, and meet near-RT control-loop requirements. *xAppGen* therefore connects high-level operator requir ements with task- and platform-specific O-RAN knowledge retrieval, xApp function generation and validation, platform-specific interface synthesis, deployment, and runtime validation. This end-to-end process transforms the O-RAN knowledge and specification-aware capabilities demonstrated by recent LLM systems into a workflow for generating deployable near-RT RIC xApps.

VII. Conclusion

xAppGen marks the paradigm shift in xApp development, from current manual programming by experts to fully automated xApp generation and deployment. *xAppGen* is motivated to streamline xApp development with the latest advances in LLMs and agentic AI, and thus simplify xApp programming, shorten new feature launch time, and stimulate O-RAN innovation. To this end, *xAppGen* builds an end-to-end automated xApp generation pipeline integrating a set of novel techniques for user intent elicitation, knowledge retrieval, code generation, and deployment. Evaluations show *xAppGen* can generate performant xApps (on par with manually developed SOTA baselines) with little programming effort at a much shorter time. These encouraging results underscore the potential of *xAppGen* in accelerating O-RAN innovation.

References

[1] O-RAN Alliance, "https://www.o-ran.org", 2024.

[2] X. Foukas, B. Radunovic, M. Balkwill, and Z. Lai, "Taking 5g ran analytics and control to a new level," in *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*, 2023, pp. 1–16.

[3] J. Xing, J. Gong, X. Foukas, A. Kalia, D. Kim, and M. Kotaru, "Enabling resilience in virtualized rans with atlas," in *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*, 2023, pp. 1–15.

[4] X. Foukas and B. Radunovic, "Concordia: Teaching the 5g vran to share compute," in *Proceedings of the ACM SIGCOMM 2021 Conference*, 2021, pp. 580–596.

[5] N. Lazarev, T. Ji, A. Kalia, D. Kim, I. Marinos, F. Y. Yan, C. Delimitrou, Z. Zhang, and A. Akella, "Resilient baseband processing in virtualized rans with slingshot," in *Proceedings of the ACM SIGCOMM 2023 Conference*, 2023, pp. 654–667.

[6] X. Foukas, T. S. Ukyab, B. Radunovic, S. Ratnasamy, and S. Shenker, "Ranbooster: Democratizing advanced cellular connectivity through fronthaul middleboxes," in *Proceedings of the ACM SIGCOMM 2025 Conference*, 2025, pp. 742–757.

[7] X. Cheng, J. Zhang, N. Ding, N. Li, Y. Li, T. Wu, W. Xu, J. Zhang, and Q. Sun, "Integrated ai and communications: A two-way catalysis toward 6g and beyond," *Journal of Communications and Information Networks*, vol. 10, no. 3, pp. 191–200, 2025.

[8] Y. Chen, J. Guo, Y. Sun, H. Yao, Y. Liu, and Y. He, "Sensing resource scheduling in 5g vran: an elastic approach," *ACM Transactions on Internet of Things*, vol. 6, no. 3, pp. 1–24, 2025.

[9] C. Sun, U. Pawar, M. Khoja, X. Foukas, M. K. Marina, and B. Radunovic, "Spotlight: Accurate, explainable and efficient anomaly detection for open ran," in *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, 2024, pp. 923–937.

[10] H. Wen, P. Sharma, V. Yegneswaran, P. Porras, A. Gehani, and Z. Lin, "6g-xsec: Explainable edge security for emerging openran architectures," in *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks*, 2024, pp. 77–85.

[11] S.-E. Hong, J. Moon, and J. Na, "Design of interpretable and enhanced anomaly detection xapp for traffic steering in o-ran," in *IEEE 36th International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*, 2025, pp. 1–6.

[12] A. Lacava, M. Polese, R. Sivaraj, R. Soundrarajan, B. S. Bhati, T. Singh, T. Zugno, F. Cuomo, and T. Melodia, "Programmable and customized intelligence for traffic steering in 5g networks using open ran architectures," *IEEE Transactions on Mobile Computing*, vol. 23, no. 4, pp. 2882–2897, 2023.

[13] R. Ntassah, G. M. Dell'Aera, and F. Granelli, "xapp for traffic steering and load balancing in the o-ran architecture," in *IEEE International Conference on Communications*, 2023, pp. 5259–5264.

[14] M. Polese, L. Bonati, S. Doro, S. Basagni, and T. Melodia, "Understanding o-ran: Architecture, interfaces, algorithms, security, and research challenges," *IEEE Communications Surveys & Tutorials*, vol. 25, no. 2, pp. 1376–1411, 2023.

[15] F. Mungari, C. Puligheddu, A. Garcia-Saavedra, and C. F. Chiasserini, "O-ran intelligence orchestration framework for quality-driven xapp deployment and sharing," *IEEE Transactions on Mobile Computing*, vol. 24, no. 6, pp. 4811–4828, 2025.

[16] N. N. Sapavath, B. Kim, K. Chowdhury, and V. K. Shah, "Experimental study of adversarial attacks on ml-based xapps in o-ran," in *IEEE Global Communications Conference*, 2023, pp. 6352–6357.

[17] A. Scalingi, S. DOro, F. Restuccia, T. Melodia, and D. Giustiniano, "Detran: Data-driven cross-layer real-time attack detection in 5g open rans," in *IEEE Conference on Computer Communications*, 2024, pp. 41–50.

[18] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," *ACM Transactions on Software Engineering and Methodology*, vol. 35, no. 2, pp. 1–72, 2026.

[19] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[20] C. Fiandrino, L. Bonati, S. D'Oro, M. Polese, T. Melodia, and J. Widmer, "Explora: Ai/ml explainability for the open ran," *Proceedings of the ACM on Networking*, vol. 1, no. CoNEXT3, pp. 1–26, 2023.

[21] G. Aguzzi, N. Farabegoli, and M. Viroli, "A language-based approach to macroprogramming for iot systems through large language models," *ACM Transactions on Internet of Things*, vol. 6, no. 4, pp. 1–30, 2025.

[22] Y. Chen, M. Chen, C. Gao, Z. Jiang, Z. Li, and Y. Ma, "Towards mitigating api hallucination in code generated by llms with hierarchical dependency aware," in *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, 2025, pp. 468–479.

[23] H. Zou, Q. Zhao, Y. Tian, L. Bariah, F. Bader, T. Lestable, and M. Debbah, "Telecomgpt: A framework to build telecom-specific large language models," *IEEE Transactions on Machine Learning in Communications and Networking*, 2025.

[24] P. Gajjar and V. K. Shah, "Oransight-2.0: Foundational llms for o-ran," *IEEE Transactions on Machine Learning in Communications and Networking*, 2025.

[25] A. Al Haj, H. A. Alameddine, and C. Assi, "Aura-translation: Intent translation for autonomous security management in o-ran," in *2026 IFIP Networking Conference (IFIP Networking)*. IEEE, 2026, pp. 1–9.

[26] A. Ganiyu, P. Gajjar, and V. K. Shah, "Ai5gtest: Llm based automation for 5g o-ran testing," in *18th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2025, pp. 298–299.

[27] A. S. Jacobs, R. J. Pfitscher, R. H. Ribeiro, R. A. Ferreira, L. Z. Granville, W. Willinger, and S. G. Rao, "Hey, lumi! using natural language for intent-based network management," in *2021 usenix annual technical conference (usenix atc 21)*, 2021, pp. 625–639.

[28] Q. Dong, L. Li, D. Dai, C. Zheng, J. Ma, R. Li, H. Xia, J. Xu, Z. Wu, B. Chang *et al.*, "A survey on in-context learning," in *Proceedings of the conference on empirical methods in natural language processing*, 2024, pp. 1107–1128.

[29] J. F. Santos, A. Huff, D. Campos, K. V. Cardoso, C. B. Both, and L. A. DaSilva, "Managing o-ran networks: xapp development from zero to hero," *IEEE Communications Surveys & Tutorials*, vol. 28, pp. 800–840, 2025.

[30] A. Feraudo, S. Maxenti, A. Lacava, P. Bellavista, M. Polese, and T. Melodia, "xdevsm: Streamlining xapp development with a flexible framework for o-ran e2 service models," in *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, 2024, pp. 1954–1961.

[31] srsRAN: Open source 4g/5g software radio access network, "https://www.srsran.com/", 2023.

[32] O-RAN Working Group 3, "O-ran near-rt ric: Architecture 7.0," *O-RAN.WG3.TS.RICARCH-R004-v07.00 Technical Specification*, 2025.

[33] O-RAN Working Group 2, "O-ran non-rt ric: Architecture 6.0," *O-RAN.WG2.Non-RT-RIC-ARCH-R004-v06.00 Technical Specification*, 2024.

[34] O-RAN Working Group 3, "O-ran e2 general aspects and principles (e2gap) 8.0," *O-RAN.WG3.TS.E2GAP-R004-v08.00 Technical Specification*, 2025.

[35] A. Hurst, A. Lerer, A. P. Goucher, A. Perelman, A. Ramesh, A. Clark, A. Ostrow, A. Welihinda, A. Hayes, A. Radford *et al.*, "Gpt-4o system card," *arXiv preprint arXiv:2410.21276*, 2024.

[36] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[37] AI-RAN Alliance, "https://ai-ran.org/", 2024.

[38] A. A. Abbassi, L. Da Silva, A. Nikanjam, and F. Khomh, "Unveiling inefficiencies in llm-generated code: Toward a comprehensive taxonomy," *arXiv preprint arXiv:2503.06327*, 2025.

[39] H. Jia, R. Morris, H. Ye, F. Sarro, and S. Mechtaev, "Automated repair of ambiguous natural language requirements," *arXiv preprint*, 2025.

[40] D. Yang, X. Xie, X. Yang, M. Hu, Y. Huang, Y. Zhang, W. Miao, T. Su, C. Wan, and G. Pu, "Assessing the impact of requirement ambiguity on llm-based function-level code generation," *arXiv preprint*, 2026.

[41] O-RAN Software Community Documentation, "https://docs.o-ran-sc.org/en/i-release/", 2026.

[42] O-RAN SC Projects: Near Realtime RAN Intelligent Controller, "https://docs.o-ran-sc.org/en/latest/projects.html", 2026.

[43] O-RAN SC xApp Python Framework, "https://docs.o-ran-sc.org/projects/o-ran-sc-ric-plt-xapp-frame-py/en/latest/overview.html", 2026.

[44] Y. Zhu, H. Yuan, S. Wang, J. Liu, W. Liu, C. Deng, H. Chen, Z. Liu, Z. Dou, and J.-R. Wen, "Large language models for information retrieval: A survey," *ACM Transactions on Information Systems*, vol. 44, no. 1, pp. 1–54, 2025.

[45] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[46] O-RAN Working Group 3, "O-ran e2 service model (e2sm) kpm 8.0," *O-RAN.WG3.TS.E2SM-KPM-R005-v08.00 Technical Specification*, 2026.

[47] O-RAN Alliance Working Group 3, "O-ran e2 service model (e2sm) 7.0," *O-RAN.WG3.TS.E2SM-R004-v07.00 Technical Specification*, 2024.

[48] R. Schmidt, M. Irazabal, and N. Nikaein, "Flexric: An sdk for next-generation sd-rans," in *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*, 2021, pp. 411–425.

[49] SonarQube, "https://www.sonarsource.com/", 2024.

[50] O-RAN Working Group 3, "O-ran e2 application protocol (e2ap) 8.0," *O-RAN.WG3.TS.E2AP-R004-v08.00 Technical Specification*, 2024.

[51] Open5GS, "https://open5gs.org", 2024.

[52] sysmoISIM SJA2, "https://sysmocom.de/products/sim/sysmousim/", 2021.

[53] Langchain, "https://www.langchain.com/", 2024.

[54] M. Douze, A. Guzhva, C. Deng, J. Johnson, G. Szilvasy, P.-E. Mazaré, M. Lomeli, L. Hosseini, and H. Jégou, "The faiss library," *IEEE Transactions on Big Data*, vol. 12, pp. 346–361, 2025.

[55] S. Ramírez *et al.*, "Fastapi." [Online]. Available: https://fastapi.tiangolo.com

[56] A. Chiejina, B. Kim, K. Chowhdury, and V. K. Shah, "System-level analysis of adversarial attacks and defenses on intelligence in o-ran based cellular networks," in *Proceedings of the 17th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2024, pp. 237–247.

[57] D. E. Rumelhart, G. E. Hinton, R. J. Williams *et al.*, "Learning internal representations by error propagation," 1985.

[58] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[59] H. Li, P. Chaudhari, H. Yang, M. Lam, A. Ravichandran, R. Bhotika, and S. Soatto, "Rethinking the hyperparameters for fine-tuning," *arXiv preprint arXiv:2002.11770*, 2020.

[60] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[61] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan *et al.*, "Deepseek-v3 technical report," *arXiv preprint arXiv:2412.19437*, 2024.

[62] Claude Opus 4.1, "https://www.anthropic.com/news/claude-opus-4-1", 2025.

[63] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[64] GPT-5, "https://openai.com/index/introducing-gpt-5/", 2025.

[65] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[66] Open AI Model Pricing, "https://developers.openai.com/api/docs/pricing", 2026.

[67] 3GPP. (2024) 3rd generation partnership project. "https://www.3gpp.org/".

[68] O-RAN Working Group 3, "O-ran e2 service model (e2sm), ran control 10.0," *O-RAN.WG3.TS.E2SM-RC-R005-v10.00 Technical Specification*, 2026.

[69] B. Brik, H. Chergui, L. Zanzi, F. Devoti, A. Ksentini, M. S. Siddiqui, X. Costa-Pérez, and C. Verikoukis, "Explainable ai in 6g o-ran: A tutorial and survey on architecture, use cases, challenges, and future research," *IEEE Communications Surveys & Tutorials*, vol. 27, no. 5, pp. 2826–2859, 2024.

[70] B. Agarwal, R. Irmer, D. Lister, and G.-M. Muntean, "Open ran for 6g networks: Architecture, use cases and open issues," *IEEE Communications Surveys & Tutorials*, vol. 28, pp. 2881–2924, 2025.
[71] X. Foukas, N. Nikaein, M. M. Kassem, M. K. Marina, and K. Kontovasilis, "Flexran: A flexible and programmable platform for software-defined radio access networks," in *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, 2016, pp. 427–441.
[72] X. Foukas, G. Patounas, A. Elmokashfi, and M. K. Marina, "Network slicing in 5g: Survey and challenges," *IEEE communications magazine*, vol. 55, no. 5, pp. 94–100, 2017.
[73] J. Groen, S. DOro, U. Demir, L. Bonati, M. Polese, T. Melodia, and K. Chowdhury, "Implementing and evaluating security in o-ran: Interfaces, intelligence, and platforms," *IEEE Network*, vol. 39, no. 1, pp. 227–234, 2024.
[74] W.-H. Ko, U. Ghosh, U. Dinesha, R. Wu, S. Shakkottai, and D. Bharadia, "Edgeric: Empowering real-time intelligent optimization and control in nextg cellular networks," in *21st USENIX Symposium on Networked Systems Design and Implementation*, 2024, pp. 1315–1330.
[75] A. Calagna, S. Maxenti, L. Bonati, S. D'Oro, T. Melodia, and C. F. Chiasserini, "Cormo-ran: Lossless migration of xapps in o-ran," *arXiv preprint arXiv:2506.19760*, 2025.
[76] L. P. Rachakonda, M. Siddula, and V. Sathya, "A comprehensive study on iot privacy and security challenges with focus on spectrum sharing in next-generation networks (5g/6g/beyond)," *High-Confidence Computing*, vol. 4, no. 2, p. 100220, 2024.
[77] C. Kilinc, M. K. Marina, M. Usama, S. Ergut, J. Crowcroft, T. Gundogdu, and I. Akinci, "Jade: Data-driven automated jammer detection framework for operational mobile networks," in *IEEE Conference on Computer Communications*, 2022, pp. 1139–1148.
[78] M. Martínez-Morfa, C. R. De Mendoza, C. Cervelló-Pastor, and S. Sallent, "Drl-based xapps for dynamic ran and mec resource allocation and slicing in o-ran," in *15th International Conference on Network of the Future*, 2024, pp. 106–114.
[79] X. Wu, J. Farooq, Y. Wang, and J. Chen, "Llm-xapp: A large language model empowered radio resource management xapp for 5g o-ran," in *Symposium on Networks and Distributed Systems Security, Workshop on Security and Privacy of Next-Generation Networks (FutureG 2025), San Diego, CA*, 2025.
[80] H. Wen, Y. Li, G. Liu, S. Zhao, T. Yu, T. J.-J. Li, S. Jiang, Y. Liu, Y. Zhang, and Y. Liu, "Autodroid: Llm-powered task automation in android," in *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, 2024, pp. 543–557.
[81] S. Lee, J. Choi, J. Lee, M. H. Wasi, H. Choi, S. Ko, S. Oh, and I. Shin, "Mobilegpt: Augmenting llm with human-like app memory for mobile task automation," in *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, 2024, pp. 1119–1133.
[82] H. Xu, L. Han, Q. Yang, M. Li, and M. Srivastava, "Penetrative ai: Making llms comprehend the physical world," in *Proceedings of the 25th International Workshop on Mobile Computing Systems and Applications*, 2024, pp. 1–7.
[83] S. Ji, X. Zheng, and C. Wu, "Hargpt: Are llms zero-shot human activity recognizers?" in *IEEE International Workshop on Foundation Models for Cyber-Physical Systems & Internet of Things*, 2024, pp. 38–43.
[84] L. Shen, Q. Yang, Y. Zheng, and M. Li, "Autoiot: Llm-driven automated natural language programming for aiot applications," in *Proceedings of the 31st Annual International Conference on Mobile Computing and Networking*, 2025, pp. 468–482.
[85] L. Shen, Q. Yang, X. Huang, Z. Ma, and Y. Zheng, "Gpiot: Tailoring small language models for iot program synthesis and development," in *Proceedings of the 23rd ACM Conference on Embedded Networked Sensor Systems*, 2025, pp. 199–212.
[86] M. A. Islam, M. E. Ali, and M. R. Parvez, "Mapcoder: Multi-agent code generation for competitive problem solving," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2024, pp. 4912–4944.
[87] D. Huang, J. M. Zhang, M. Luck, Q. Bu, Y. Qing, and H. Cui, "Agentcoder: Multi-agent-based code generation with iterative testing and optimisation," *arXiv preprint arXiv:2312.13010*, 2023.