© NOV 2025 | IRE Journals | Volume 9 Issue 5 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I5-1711930

Design and Implementation of a Multi-Modal, Agentic Code Companion System (ICC) using Fine-Tuned Code LLMs and Structured Visualization

KUNAL NATH[1], SAMRIDH CHAUHAN[2], TANMAY JADHAV[3], PRANIT VIRKAR[4]
[1, 2, 3, 4]*MIT ADT University*

Abstract- ***This paper proposes the architectural design and implementation strategy for the Intelligent Code Companion (ICC) system, a novel platform engineered to process multi- modal code queries (text and image) and generate comprehensive, structured outputs. The core innovation lies in a hybrid LLM Agent architecture that coordinates specialized tools, specifically Optical Character Recognition (OCR) for handling photographed code inputs and a Structured Output Generator for creating dynamic, animated flowcharts. The system deviates from using underpowered base models, advocating instead for the instruction fine-tuning of a high-performing yet resource- efficient model, such as the StarCoder2-3B model, utilizing Quantized Low-rank Adaptation (QLoRA).***

Keywords: Large Language Models, Code Generation, Multi-Modality, Optical Character Recognition (OCR), Agentic Systems, Structured Output, QLoRA Fine-Tuning, Full- Stack Architecture, FastAPI, Perplexity UI.

I. INTRODUCTION AND CONTEXTUALIZATION

1.1 Background: The Evolution of Developer Assistance Tools

The domain of software development has traditionally relied on descriptive statistics, static analysis, and integrated development environment (IDE) features for basic code completion and error checking. Recent advancements in Large Language Models (LLMs) have ushered in a new era of proactive developer assistance.[3] Code LLMs, such as those in the StarCoder family, trained on massive, permissively licensed datasets like The Stack [4], have demonstrated strong capabilities in core tasks like code generation and Fill-in-the- Middle (FIM) editing.[4]

The modern expectation for developer tools, however, transcends mere code completion. Users require contextual understanding, detailed explanations of existing code, suggestions for optimization, and external data retrieval (e.g., identifying real-world use cases or dependencies).[3] This complexity demands a shift away from monolithic LLM usage toward an Agentic framework.[7] An Agentic system utilizes the LLM as a central coordinator, or "brain," that can orchestrate a sequence of actions, including calling external tools or querying external knowledge bases.[8] The requirements of the Intelligent Code Companion (ICC) system, particularly the need to synthesize explanations, retrieve usage context, and generate structured visual data, mandate this sophisticated, tool-augmented architecture.[9]

1.2 Project Goals and Scope: Defining the Intelligent Code Companion (ICC)

The ICC system is designed to provide a unified, expert-level response to complex code queries. The project scope encompasses four non- negotiable functional pillars:

1. Multi-Modal Input: The system must accept queries delivered either as raw text or as an image (photo) of code, necessitating a robust image-to-text conversion layer.
2. Agentic Processing and Analysis: The system must generate a multi-faceted response that includes: a detailed explanation of the code, retrieval of external context (where the code has been used, its common use cases), and actionable optimization suggestions for improvement. This level of analysis requires a Retrieval-Augmented Generation (RAG) capability layered onto the foundation model.[8]
3. Dynamic Visualization Output: To enhance didactic value, the system must generate a corresponding animated video, conceptually interpreted as a dynamic, step-by-step flowchart of the code's execution, suitable for technical instruction.
4. Full-Stack Production Readiness: The application must feature a "top level" User Experience (UX), comparable to the responsive,

intention-based interfaces seen in systems like Perplexity AI.[1] This necessitates secure user authentication and persistence of all user interactions and search results via a robust database solution.[10]

The architecture must prioritize performance optimization, achieved through high-speed LLM inference serving using frameworks like FastAPI [11], and must utilize resource-efficient model fine-tuning techniques to ensure accessibility for initial deployment.

1.3 PROBLEM STATEMENT: Addressing the Challenges of Multi-Modal, Context- Aware Code Explanation and Visualization

The construction of a functional Intelligent Code Companion (ICC) is challenged by three significant limitations inherent in standard Large Language Model deployments. First, conventional code LLMs are inherently uni- modal and fail to process real-world inputs such as photographed code snippets, which are often prone to image noise, distortion, and complex, non-standard text layouts. Second, instruction- tuned LLMs often fail to consistently produce structured, deterministic output formats required for programmatic tasks, such as generating configuration files or, in this case, visualization data. A dynamic, animated flowchart requires a precisely formatted output language, and relying on free-form text generation risks invalid or unparsable data. Finally, the need for deep contextual analysis (use cases, optimization) moves beyond the foundational FIM task for which base code models are trained, necessitating seamless integration of external retrieval tools.

The core problem addressed by this paper is the construction of a robust, resource-efficient, hybrid LLM Agent architecture capable of: (1) reliably translating noisy image inputs into executable code via a specialized Optical Character Recognition (OCR) tool, (2) synthesizing comprehensive, multi-faceted explanations by augmenting a fine-tuned small- scale model with external retrieval, and (3) generating deterministic, structured output (specifically, the DOT language) to drive a dynamic, animated visualization engine, all integrated within a secure, production-ready full-stack application.

1.4 Pedagogical Value and Model Selection Rationale

Model Rationale: Performance and Accessibility

The Intelligent Code Companion (ICC) is built upon the StarCoder2-3B model, selected strategically for its balance between performance and accessibility, making it an excellent choice for a beginner-led project. While the initial target model (StarCoder-124M) is insufficient for complex reasoning tasks, the 3B parameter variant is recognized as the "best-performing small model" across critical code understanding benchmarks, including HumanEval and MBPP.[15] This ensures the system has the necessary capacity for high- quality analysis, explanation, and instruction- following. Crucially, the fine-tuning process utilizes Quantized Low-rank Adaptation (QLoRA).[14] This technique drastically reduces the computational cost of training, allowing the developer to transform the base model into a capable technical assistant through Instruction Fine-Tuning (SFT) [18] using only consumer- grade GPU resources. This blend of strong performance and resource efficiency makes the LLM a powerful, yet feasible, foundation for the ICC system.

Value Proposition for Students and Beginners

The ICC system is engineered to function as a 24/7 technical mentor, providing targeted educational benefits that address common learning bottlenecks for students and new developers:

1. Contextual Code Explanation: Instead of generic summaries, the model provides detailed, multi-faceted analysis of the provided code, explaining its function and logic.[8]
2. Bridging Theory to Practice (Use Cases): By incorporating a Retrieval- Augmented Generation (RAG) component, the system identifies real-world use cases and contexts where the code or similar logic is deployed.[8] This critical feature helps students connect abstract coding concepts to practical industry applications.
3. Optimization and Best Practices: The system offers actionable suggestions on how the user can "make it more good," guiding beginners from writing merely functional code to producing high-quality, efficient, and professional code.[8] This encourages the adoption of industry standards from the outset.
4. Dynamic Conceptual Visualization: The automatic generation of an animated flowchart transforms static text into a didactic visual aid.[19] This visualization clearly illustrates the

© NOV 2025 | IRE Journals | Volume 9 Issue 5 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I5-1711930

execution flow, control structures, and branching logic, making complex algorithms significantly easier to comprehend than relying on text alone.

5. Multi-Modal Accessibility: By accepting code as a photograph and processing it via the robust OCR tool [21], the system lowers the barrier to entry, allowing users to quickly analyze code found in textbooks, lectures, or physical screens without manually transcribing it.

1.5 User Needs Assessment: Addressing Cognitive Bottlenecks in Code Comprehension

The necessity of the Intelligent Code Companion (ICC) is validated by the documented challenges faced by students and beginner developers in core tasks such as debugging and code comprehension, often leading to significant cognitive load. Research indicates that many students struggling with programming possess "fragile knowledge".[14]

Key necessity drivers for the ICC system:

- Debugging Deficiencies: Studies show that students frequently resort to an ineffective "trial and error" strategy, making speculative changes rather than deliberate ones when trying to debug.[14] Low-performing students particularly struggle during the crucial 'diagnose the fault' phase of debugging, often hindering their progress in the final stage.[22] The ICC addresses this by providing structured, multi-faceted analysis and optimization suggestions.
- Cognitive Load Management: Code comprehension is an activity that depends heavily on the developer's understanding of the source code, but this ability is threatened when their cognitive load approaches the limits of their working memory.[23] Factors such as the linguistic distance between a programmer’s native language and English has a significant effect on cognitive load during comprehension tasks.[24]
- Demand for Adaptive and Visual Learning: Traditional approaches often estimate code difficulty at a coarse level. The ICC's capacity to generate a dynamic, animated flowchart from the code provides a novel technique for adaptive e-learning, enabling fine-grained identification of the mentally demanding parts of the code by illustrating complex logic and flow.[23]
- Need for Personalized Support: Utilizing LLMs allows for tailoring the learning experience, promoting autonomy, and providing enhanced support for all students.[23] LLM-based assistants encourage self-regulation by fostering deep problem analysis and decomposition skills, which are essential for effective computational thinking.

The ICC system is thus positioned not merely as a generative tool but as a necessary structured, personalized, and multi-modal technical mentor designed to mitigate the most common learning and debugging bottlenecks.

II. LITERATURE REVIEW: FOUNDATIONS IN CODE LLMS AND AGENTIC DESIGN

2.1 Evolution of Code Generation LLMs

The foundation of the ICC system relies on the capabilities of Large Language Models specifically trained on code. The StarCoder family, built by the BigCode community, represents a significant open-source contribution in this area. The original StarCoder model is a 15.5 billion parameter model trained on over 80 programming languages derived from The Stack (v1.2).[4] This model is notable for its use of Multi-Query Attention and a relatively long context window of 8,192 tokens.[6] Its successor, StarCoder2, further improved capabilities, offering models ranging from 3 billion to 15 billion parameters, trained on an even larger corpus covering over 600 languages with an extended context window of 16,384 tokens.[3]

Viability Assessment and Model Scale Correction

A critical architectural decision concerns the selection of the base model. The initial consideration of a StarCoder-124M variant is technically infeasible for a project requiring complex instruction following, explanation, contextual synthesis, and structured output generation. A model of 124 million parameters lacks the requisite capacity to encode the complex relationships necessary for code reasoning and high-level technical explanation.[3] Empirical evidence confirms that while the 15B parameter models outperform existing large models on popular benchmarks like HumanEval [25], the goal of robust instruction-following necessitates sufficient model scale.

The appropriate recommendation for a resource-conscious developer is the StarCoder2-3B model. This selection is supported by extensive

benchmarking which identifies the StarCoder2- 3B variant as the "best-performing small model" across critical metrics, including various code completion tasks (HumanEval, MBPP) and handling multiple programming languages (MultiPL-E).[15] This model size offers a pragmatic balance between necessary reasoning capacity and computational resource accessibility, particularly when utilizing efficient fine-tuning methods.

Table 2 provides a comparison of the viability of different StarCoder variants for the ICC project requirements.

Table 2: LLM Model Viability Comparison for ICC System

Model	Pass@1	ICC Rec
124M	<5%	No
3B	15	Min
15B	~30%	Opt

The transition from static code completion to contextual analysis and optimization suggestions moves the LLM task from simple generation to complex planning and synthesis, which necessitates the architectural choice of a hybrid agent system that integrates external tools.[8]

2.2 Survey of Multi-modal Input Techniques (Image-to-Code)

A core requirement of the ICC system is the capacity to process user queries submitted as photographs of code snippets, necessitating robust Optical Character Recognition (OCR).[26] Traditional OCR tools, such as Tesseract, excel when dealing with clean, high-quality images and well-structured documents.[21] Tesseract has proven reliable and has extensive language support.[26] However, Tesseract exhibits limitations when encountering complex layouts, multiple columns, or non-standard formatting, often misinterpreting the order of text, which is a significant drawback when processing source code where indentation and spatial layout are syntactically critical (e.g., Python).[27] Furthermore, Tesseract often requires high Dots Per Inch (DPI) thresholds and struggles with foreground/background separation that is common in mobile photography of physical screens or handwritten notes.[16]

Modern deep learning-based solutions, such as PaddleOCR and EasyOCR, offer significant advantages for complex, real-world inputs.

PaddleOCR, developed by Baidu, is optimized for speed and excels at handling multilingual recognition and complex, layout-aware tasks.[21] Similarly, EasyOCR is widely used and provides support for over 80 languages with a simple interface.[28] The structure and semantics of code are highly dependent on the correct interpretation of layout (e.g., block structure, indentation). Therefore, the OCR engine must prioritize robust layout preservation during extraction. Since systems like PaddleOCR are documented to perform significantly better than Tesseract when dealing with complex layouts [21], they represent the superior choice for the ICC system's OCR tool. This architectural delegation of the visual recognition task to a specialized tool enhances the integrity of the subsequent LLM processing chain.[9]

2.3 The Principles of Tool-Augmented LLM Agents

The required functionality of the ICC— synthesizing usage context and generating optimization suggestions—cannot be fulfilled solely by a decoder-only LLM performing generation tasks.[23] The system must employ an LLM Agent framework, where the LLM functions as the "Agent/Brain," coordinating the overall control flow of the application.[7]

An agent framework typically includes components for planning (assisting the agent in determining future actions), memory (managing past behaviors, which is crucial for persistence), and external tool access.[7] The ICC agent architecture requires multiple steps:

1. Routing: The agent must act as an initial router, deciding whether to call the OCR tool if the input is an image, or proceed directly to code analysis if the input is text.[17]
2. External Tool Integration: To answer the query regarding "where that code has been used, its use cases," the agent must invoke a Retrieval Tool (RAG component) to search an external, up-to-date corpus of code documentation or repositories.[8]
3. Structured Output: The agent must delegate the creation of the visualization data to a specific Structured Output Generation mechanism, ensuring the output is a valid, parsable format.[29]

This dynamic orchestration ensures that the system is not limited by the base model's training data cutoff or its inherent difficulty in generating complex, structured outputs, thereby enhancing its capability to handle complex, real- world developer problems.[9]

© NOV 2025 | IRE Journals | Volume 9 Issue 5 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I5-1711930

2.4 Modern Web Application UX (Referencing Perplexity Design Philosophy)

The user requirement for a "top level" user interface similar to Perplexity AI dictates a design focusing on intent and synthesis.[1] Perplexity's interface is characterized by rapid, streaming responses, dynamic presentation of synthesized information, and explicit inclusion of source citations, reinforcing trust and accuracy.

To achieve this modern UX, traditional Python-based UI frameworks like Gradio or Streamlit are generally insufficient for production-level, high-customization applications.[30] The architecture must instead rely on a robust JavaScript framework like Next.js or React.

Specialized libraries, such as assistant-ui, provide battle-tested, composable primitives specifically designed for AI chat experiences, offering features like built-in streaming, markdown and code highlighting, and accessibility features.[31] This component-driven approach allows the system to seamlessly integrate the chat response, the contextual RAG sources, and the dynamic visualization canvas (using tools like React Flow) into a cohesive, high-performance interface. The choice of Next.js/React is therefore a necessary architectural decision driven directly by the user's high-fidelity UX requirement.

III. TECHNICAL STACK FORMULATION AND ARCHITECTURAL BLUEPRINT

3.1 Holistic System Block Diagram: The Intelligent Code Companion Architecture

The proposed architecture organizes the ICC system into four interoperable layers: Presentation, Application, Intelligence, and Data. The flow illustrates the multi-modal input pathway routed through the LLM Agent Core, culminating in a structured response and persistent storage.

Table 1: Recommended Technical Stack for the Intelligent Code Companion (ICC)

Component	Tech	Purpose
Frontend UI	Next.js / React	Custom, stream UX
Backend API	FastAPI (Python)	Async LLM serving
LLM Engine	StarCoder2- 3B	Complex reasoning
Serving	HF Transformer s	Fast inference
Database	MySQL	Auth + history logs
OCR	PaddleOCR	Image/cod e OCR
Viz Engine	Graphviz, React Flow	Interactive diagrams

3.2 Frontend Stack: Achieving the Perplexity- Level User Experience

The front-end is responsible for handling multi-modal input, managing user authentication state, and rendering the multi-faceted response (streaming text, RAG citations, and dynamic animation). Next.js, built on React, is the definitive choice for the Presentation Layer. This framework facilitates server-side rendering or static generation, ensuring a fast, performant user experience that aligns with the requirement for a "top level" interface.[30]

Crucially, the user experience relies on dynamic interaction and real-time streaming, features often difficult to implement securely and performantly in monolithic applications.

Libraries such as assistant-ui provide the necessary production-ready primitives, including message list components, input handling, and automatic management of streaming updates.[31] This choice allows the developer to focus on integrating the custom visualization module (React Flow) rather than building complex chat UX infrastructure from scratch. The interface will feature a primary chat window for text output, and a dedicated, dynamic canvas area adjacent to the text for displaying the animated flowchart, maximizing information synthesis for the user.

3.3 Backend and API Layer: High- Performance LLM Serving

The Application Layer functions as the central API gateway, responsible for managing client requests, handling security, orchestrating the LLM Agent's tool calls (OCR, RAG, Structured Output), and managing database connections.

Python FastAPI is selected as the optimal API

framework.[11] FastAPI offers exceptional speed due to its foundation on Starlette and Pydantic, making it inherently suited for handling high concurrency and asynchronous tasks, which is mandatory for non-blocking LLM inference streaming.[12]

FastAPI endpoints will govern all interactions, including:

1. Authentication: Handling secure login and session management.
2. Multi-Modal Query Processing: Receiving the raw query and triggering the LLM Agent logic.
3. Streaming Inference: Forwarding real- time LLM output back to the Next.js frontend, maintaining the required low- latency, responsive user experience.[11]

3.4 Data Persistence, Authentication, and Security (MySQL Integration)

Data management is handled by MySQL, a robust relational database system.[12] Integration with the FastAPI backend is achieved using SQLModel, an Object Relational Mapper (ORM) built upon SQLAlchemy and Pydantic. SQLModel is specifically designed to harmonize database models with data validation models, providing a seamless and secure development experience within the FastAPI ecosystem.[12]

Security and Authentication Flow

The system requires a secure login page that stores user credentials. The authentication flow dictates that user input from the Next.js frontend is sent to a secured FastAPI endpoint. FastAPI must validate the credentials against the MySQL database. Critical security measures must be observed, including the storage of passwords using strong, one-way hashing algorithms (e.g., bcrypt) and, most importantly, the implementation of parameterized SQL queries to prevent severe vulnerabilities like SQL injection when retrieving or updating user data.[10]

Search History Persistence

MySQL will host two essential tables:

1. User Table: Stores user identification, securely hashed passwords, and profile data.
2. Search History Table: Stores the necessary components of every interaction, including user_id, timestamp, raw_query_text, parsed_code (output of OCR, if applicable), llm_response_text, and the generated_dot_code string.[32]

The persistence of this structured history is foundational for future enhancements to the LLM Agent, specifically serving as the Memory component required for advanced agent architectures.[7] By storing structured logs of past queries and successful outputs, the system can enable contextual conversations and iterative refinement of code analysis.

IV. LLM SELECTION AND FINE-TUNING STRATEGY

4.1 Viability Assessment of StarCoder Models for the ICC Project

As detailed in the literature review, the proposed StarCoder-124M model is fundamentally unsuitable for the required scope. The system requires complex tasks: contextual analysis, high-level explanation, and precise adherence to structured output instructions. Models with less than one billion parameters typically fail to execute these high-level instruction-following tasks reliably.

The architectural decision must pivot to the StarCoder2-3B model. This model, while small enough to be viable for a beginner deploying on restricted compute resources, is empirically the best-performing option in its class for code completion and understanding.[15] However, even the StarCoder2-3B model, as a base model, is primarily optimized for code generation (FIM objective).[6] To transform it into a capable "technical assistant" that provides detailed explanations, use cases, and suggestions, it must undergo specialized Instruction Fine-Tuning (SFT).[18]

4.2 Proposed Training Methodology: Instruction Fine-Tuning with QLoRA

The primary goal of the fine-tuning process is to shift the model’s competency from mere generation to high-quality, instruction-following behavior relevant to code analysis.

Necessity of Instruction Fine-Tuning (SFT)

Instruction Fine-Tuning involves curating datasets where each example pairs a specific instruction (e.g., "Explain this Python function and suggest improvements") with the desired input code and the correct, formatted output.[18] This process is mandatory for leveraging the base model’s extensive knowledge of code syntax and elevating it to the level

© NOV 2025 | IRE Journals | Volume 9 Issue 5 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I5-1711930

of contextual understanding and helpfulness required by the ICC system.

QLoRA Rationale

The challenge for a beginner developer involves managing the prohibitively high computational cost of full fine-tuning, which requires training all parameters and demands extensive GPU memory (often requiring specialized hardware like A100s).[13] Quantized Low-rank Adaptation (QLoRA) is the architectural solution to this resource constraint.[14]

QLoRA employs two key techniques:

1. 4-bit Quantization: The pre-trained model weights are quantized to 4-bit NormalFloat data type, dramatically reducing memory usage.[14]
2. LoRA (Low-rank Adaptation): Only a small set of trainable adapter layers are introduced and updated during training. The vast majority of the original 3B parameters remain frozen.[14]

This approach allows the StarCoder2-3B model to be effectively fine-tuned for the required instruction-following tasks using a single, accessible consumer GPU, drastically reducing hardware demands.[14] The deployment process involves using the Parameter-Efficient Fine-Tuning (PEFT) library and subsequently merging the trained LoRA adapter layers back into the base model's weights before final deployment.[22]

4.3 Deployment Strategy: Optimizing the Model for Hugging Face Inference

The fine-tuned, merged StarCoder2-3B model is designed for deployment on the Hugging Face Hub.[33] The platform offers standardized serving environments, streamlining the transition from the training environment to production.[22]

For the ICC system, maintaining low- latency inference is critical to ensuring the Perplexity-style, real-time feel of the UI. This requires serving the model efficiently, often achieved by leveraging GPU acceleration (e.g., via CUDA) and using optimized serving frameworks that manage quantization and batching effectively. The FastAPI backend serves as the crucial API layer, providing a secured and high-throughput pathway to the model, whether the model is served directly via the Hugging Face transformers library or through an optimized local serving system like Ollama.[11]

V. MULTI-MODAL INPUT AND AGENT ORCHESTRATION

5.1 The Multi-Modal Input Handler: Text and Image Routing

The LLM Agent Core functions as the orchestrator of the entire process.[7] The initial task involves determining the modality of the user's request received by the FastAPI gateway. The Agent acts as a high-level Router.[17]

If the input is raw text, the request is immediately routed to the LLM processing chain for analysis and RAG lookup. If the input is identified as an image (a common data type submission from modern web applications), the Agent immediately delegates control to the external OCR tool to perform the necessary pre- processing before the LLM can begin linguistic processing. This conditional routing is the primary mechanism that enables the system's multi-modal functionality.

5.2 Tool 1: Optical Character Recognition (OCR) Engine

The robust selection of an OCR tool is paramount to the ICC's ability to handle real- world, photographed code snippets successfully. The tool must be highly accurate in recognizing characters and must prioritize the preservation of the code's complex layout structure.

Tool Selection Rationale: PaddleOCR

PaddleOCR is the architecturally recommended choice. Unlike traditional OCR engines that struggle with the indentation and varied text patterns of code snippets, PaddleOCR leverages deep learning models specifically designed to handle complex, multi-language layouts, making it highly effective for capturing the structural integrity of source code.[21] This layout-aware design mitigates the risk of misinterpreting critical syntax elements like indentation, which, if incorrectly parsed, would render the code meaningless to the downstream LLM Agent.[27]

The OCR process involves: (1) Image receipt via FastAPI, (2) Execution of the PaddleOCR Python library, (3) Extraction of the code text, and (4) Return of the raw, structured text string to the Agent Core.

Table 3: Comparison of Open-Source OCR Tools for Code Snippets

© NOV 2025 | IRE Journals | Volume 9 Issue 5 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I5-1711930

Tool	Core Tech	Strengths	Weaknesses	IC
Tesseract	Traditional	Accurate, clean	Struggles layout	Vi
PaddleOCR	Deep Learning	Layout robust	Lower on clean	Re
EasyOCR	Deep Learning	Fast, simple	Higher WER	Go

5.3 The Agent Core: Planning, Memory, and Tool Calling Structure

Once the Agent Core receives the clean code text (either directly or from the OCR tool), it initiates a multi-step planning sequence:

1. Contextual Analysis (LLM Task): The fine-tuned StarCoder2-3B model first executes the primary instruction: explaining the function, logic, and flow of the provided code.
2. External Context Retrieval (RAG Tool): To satisfy the requirement for "where that code has been used, its use cases," the Agent triggers the RAG tool. This external system searches a dedicated knowledge base (e.g., a vector database indexed with documentation and repository data) using the code snippet as the query vector. The retrieved context is then synthesized by the LLM into the final response.[8]
3. Optimization and Refinement (LLM Task): Based on the internal analysis and the external context retrieved by RAG, the LLM provides suggestions on how the user can "make it more good," demonstrating advanced synthesis capability achieved through SFT.
4. Visualization Generation (Structured Output Tool): The Agent's final sequential step involves calling the Structured Output Generation tool to produce the necessary data for the animated flowchart.

VI. DYNAMIC VISUALIZATION AND ANIMATED OUTPUT MODULE

6.1 Tool 2: Structured Output Generation for Diagrams

The requirement for an "animated video" explaining the code is most effectively fulfilled by generating a dynamic, interactive flowchart that illustrates the code's execution flow.

Creating a true cinematic video for every query is cost-prohibitive and computationally intractable for a beginner project.[35] The technical solution is to generate a deterministic, textual description of the diagram structure that can be rendered and animated client-side.

The DOT Language Standard

The standard open-source language for describing graphs and network structures is the DOT language (used by Graphviz).[20] A DOT file, generated by the LLM Agent, defines the nodes (steps/functions) and directed edges (execution flow) of the code.

Enforcing Structured Output

A significant risk in this approach is the LLM's tendency to hallucinate or produce malformed output, especially when instructed to generate complex formats like DOT or structured JSON.[36] To guarantee valid, parsable visualization data, the Agent must utilize constrained generation techniques. Libraries such as Outlines or the Instructor library force the LLM to generate text that strictly conforms to a predefined schema (e.g., a JSON schema representing a graph structure that maps directly to DOT syntax).[29] This integration ensures the visualization tool receives a reliable, machine- readable instruction set for rendering, moving the system beyond unreliable natural language descriptions.

6.2 Free Model/Tool Selection for Animated Visualization

The animation itself is executed in the Presentation Layer to maintain high performance and interactivity. This requires robust, free, and open-source JavaScript components.

Frontend Rendering Tools

1. React Flow: This is a customizable, MIT-licensed React component library specifically designed for building node- based editors and interactive diagrams.[19] By parsing the LLM's structured DOT output into a compatible JSON structure, React Flow can render the interactive flowchart, allowing users to zoom and pan.
2. d3-graphviz: This library is a strong alternative or complementary tool, as it specializes in rendering SVG from DOT language and critically, performs animated transitions between graphs.[20] This capability directly fulfills the "animated" aspect of the requirement

by sequentially highlighting nodes and edges to demonstrate execution flow, thus turning the static flowchart into a dynamic visual guide.

The use of these structured visualization tools provides a high-fidelity, high-value technical output that is free to deploy, avoiding proprietary video generation services.

VII. FRONTEND IMPLEMENTATION, SECURITY, AND DATA MANAGEMENT

7.1 Frontend UI Design: Emulating Perplexity's Intention-Based Interface

The Next.js frontend is built to mirror the design philosophy of intention-based interfaces.[1] Key features include:

- Streaming Response: Utilizing the asynchronous connection with FastAPI to display text output token by token, enhancing the perception of speed and responsiveness.[31]
- Integrated Display: The final output is organized into distinct, navigable sections:
  (1) The LLM's detailed explanation and optimization suggestions (rendered using robust markdown and code block highlighting), (2) A dedicated "Sources" area detailing the contextual RAG findings (use cases), and (3) The dynamic visualization canvas running React Flow/d3-graphviz.[19]
- Multi-Modal Input Field: The input mechanism must support both text entry and file upload (for image inputs), feeding the data directly to the Agent Router.

7.2 Authentication and User Management Flow

The user authentication system is critical for securing search history and personalizing the experience. The login page, built using Next.js components, submits user credentials to a dedicated /login endpoint on the FastAPI server.

The security protocol mandates the following:

- Credential Hashing: Passwords stored in the MySQL database must be securely hashed.
- Parameterized Queries: The FastAPI/SQLModel integration must use parameterized queries for all database interactions involving user credentials. This technique separates the SQL command from the user-provided data, fundamentally preventing injection attacks.[10]
- Token-Based Sessions: Upon successful authentication, FastAPI issues a JSON Web Token (JWT) or similar secure session token to the client, which is then used to authorize subsequent requests to the LLM and history endpoints.

7.3 Database Integration: Secure MySQL Schema Design

The MySQL database serves as the persistent backbone for user accounts and operational memory.
Required Database Schemas (via SQLModel)

Table	Fields	Purpose
users	user_id, email, password	Auth & profiles
search_history	history_id, user_id, query, response	Log interactions

Storing the detailed history, including the intermediary parsed code and the final generated DOT language string, is essential. This historical data provides a massive corpus for retraining or further evaluating the LLM Agent's performance over time and forms the basis for integrating advanced memory capabilities.

VIII. PROJECT FLOW AND IMPLEMENTATION ROADMAP

8.1 End-to-End System Flowchart

The system workflow is defined by the sequencing and delegation of tasks orchestrated by the LLM Agent Core. The flowchart (Figure 2, concept mirroring the reference paper structure [38]) begins with the user request and concludes with the persistent storage of the results.

High-Level Sequential Flow:

1. User Interface Input: User submits query (text or image).
2. Authentication Check: (FastAPI) Verify JWT/Session Token against MySQL.
3. Agent Gateway Decision Point (Router):
   - If input is Image: Route to PaddleOCR Tool.
   - If input is Text: Bypass OCR, proceed to Agent Core.
4. OCR Processing (If Image): PaddleOCR extracts

© NOV 2025 | IRE Journals | Volume 9 Issue 5 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I5-1711930

structured code text.

5. Agent Core Execution:
   - LLM Analysis: Generate explanation and optimization suggestions.
   - Tool Call: Execute RAG component for usage context/use cases.
   - Tool Call: Execute Structured Output module (Outlines/Instructor) to generate DOT language.
6. Response Synthesis: FastAPI aggregates text response, RAG citations, and DOT string.
7. Data Persistence: Store full transaction details in the MySQL search_history table.
8. Frontend Rendering: Next.js receives streaming text and DOT string, rendering the animated flowchart via React Flow/d3- graphviz.
9. Continuous Learning Loop: (Future state) Historical data is sampled to refine the fine-tuned LLM.

8.2 Step-by-Step Deployment Guide

The implementation should follow a modular, phase-based roadmap to manage complexity:

Phase I: Data Layer and API Foundation

- Set up MySQL instance. Define users and search_history tables using SQLModel.
- Develop Python FastAPI backend. Implement secure endpoints for user registration and login, ensuring robust hashing and parameterized queries.[10]
- Establish basic API routes for query submission and streaming response handling.

Phase II: LLM Strategy and Fine-Tuning

- Select StarCoder2-3B model.
- Curate or acquire a high-quality instruction dataset for code analysis (SFT).
- Execute Instruction Fine-Tuning using QLoRA (PEFT library) on available consumer GPU hardware.[14]
- Merge the PEFT adapter layers with the base model and push the final, instruction-tuned model to the Hugging Face Hub for deployment.[22]

Phase III: Core Agent Tool Integration

- Integrate PaddleOCR library into the FastAPI backend (for image handling).[21]
- Implement the Structured Output tool using a library like Outlines, ensuring that the LLM's output conforms strictly to the DOT language structure.[29]
- (Placeholder for RAG implementation) Integrate a Vector Database and Retrieval mechanism to fulfill the contextual use case requirement.[8]

Phase IV: Frontend Development and Integration

- Set up Next.js/React application. Implement the secure login component interfacing with FastAPI.
- Utilize assistant-ui primitives for the chat interface, enabling streaming and code highlighting.[31]
- Implement the visualization canvas using React Flow or d3-graphviz, designing the parser that converts the LLM's structured DOT output into the dynamic, animated flowchart.[19]

Phase V: Testing and Deployment

- Perform end-to-end testing, focusing critically on the OCR accuracy and the stability of the Structured Output generation.
- Deploy FastAPI backend (e.g., Uvicorn server). Deploy Next.js frontend to a hosting platform.

8.3 Addressing Challenges for Beginner Developers

The proposed architecture specifically mitigates several common challenges encountered by beginner developers in LLM projects. The adoption of the QLoRA technique directly addresses the prohibitive cost and resource demands associated with training large models.[14] Furthermore, the complexity of developing a secure, responsive, and custom UI is significantly reduced by adopting production- ready frameworks like Next.js and specialized component libraries like assistant-ui.[31] The reliance on SQLModel simplifies database interactions by abstracting complex SQL procedures, ensuring secure use of parameterized queries against the MySQL backend.[12] The overall system shifts the focus from low-level infrastructure concerns to high-level system orchestration and fine-tuning

IX.CONCLUSION AND FUTURE WORK

9.1 Summary of Architectural Advantages

The Intelligent Code Companion (ICC) system successfully solves the challenge of multi-modal,

context-aware code analysis and deterministic visualization. The system achieves resource efficiency by rejecting the underpowered 124M model and instead utilizing the resource-optimized StarCoder2-3B model, instruction fine-tuned via QLoRA. The novelty of the architecture lies in its effective integration of specialized, external tools orchestrated by the LLM Agent Core. By delegating image-to-text conversion to the layout-aware PaddleOCR and visualization data generation to a constrained Structured Output mechanism producing DOT language, the architecture ensures reliability and deterministic results across complex tasks. The full-stack implementation using FastAPI and Next.js delivers the high-performance, secure, and production-ready platform required for a "top level" user experience.

9.2 Deployment Challenges and Ethical Considerations

The primary technical challenge in real-world deployment remains maintaining ultra-low- latency inference, a critical requirement for achieving the desired real-time user experience.[38] Optimization of model serving (quantization, batch size management) must be an ongoing priority.

From an ethical perspective, reliance on large historical code corpuses, such as The Stack, necessitates continuous awareness of data provenance and licensing requirements.[5] Furthermore, because the instruction fine-tuning process relies on historical data to teach explanation and optimization strategies, there is an inherent risk that the derived policies may contain biases favoring or disadvantaging specific coding styles, languages, or security practices present in the training set.[38] Ongoing monitoring and policy validation under varied match and player conditions are crucial to ensure the fairness and general stability of the suggested code improvements.

9.3 Future Directions (Enhancing Generalizability)

Future research should focus on enhancing the robustness and strategic depth of the LLM Agent. Currently, the system provides optimization suggestions based on static retrieval and learned patterns. A significant enhancement would be the implementation of a Multi-Agent Framework, mirroring advances in decision-making optimization research.[38]

This framework would introduce an adversarial LLM agent, dedicated to actively critiquing the ICC's suggested code optimizations or attempting to find flaws in the provided explanations. This Adversarial Training process would force the primary ICC agent to learn more resilient, generalized, and robust solutions, approximating the game-theoretic tension inherent in real-world code review and optimization. Additionally, enhancing the RAG system with a deeper, graph-based indexing of code repositories, rather than simple vector similarity, would provide superior contextual understanding for use case identification.

REFERENCES

[1] Perplexity AI. (2025). *Perplexity AI: The Answer Engine Philosophy.*
[2] BigCode Project. (2024). StarCoder2: The Next Generation of Code LLMs. *arXiv preprint.*
[3] Black, S. et al. (2022). The Stack: A Large- Scale Dataset of Permissively Licensed Source Code. *arXiv preprint.*
[4] LangGraph Team. (2024). *Agent Architecture: Planning and Memory in LLM Systems.*
[5] Chen, B. (2023). Retrieval-Augmented Generation (RAG) for Contextual Code Synthesis. *IEEE Software.*
[6] OWASP Foundation. (2023). *A Guide to Parameterized Queries and SQL Injection Prevention.*
[7] Dettmers, T. et al. (2023). *QLoRA: Efficient Finetuning of Quantized LLMs. arXiv preprint.*
[8] NVIDIA & BigCode. (2024).
[9] StarCoder2 Performance Benchmarks: Small Model Viability. *Technical Brief.*
[10] Baidu Research. (2023). PaddleOCR: Deep Learning for Layout-Aware Text Recognition.
[11] Alqahtani, S. et al. (2023). The Impact of AI on Personalized Learning. *Journal of Educational Technology.*