Code Whisperer: A Local Ai-Powered Code Generation And Execution Platform

Nishmitha Shetty[1], Harshitha L[2], Sathvik P[3], MD Asif[4], Dr.Mamatha M[5]
[1,2,3,4,5]*Dept. of AI & DS SAIT Bangalore*

***Abstract*—Code Whisperer is a locally deployed AI-assisted programming system that converts natural-language problem descriptions into executable source code across multiple programming languages. The system utilizes a pre-trained large language model, LLaMA 3.2, executed locally through the Ollama runtime, replacing earlier cloud-based and API-dependent code generation models commonly used in existing tools. Unlike prior approaches that rely on remote inference, the proposed system operates entirely offline, ensuring data privacy, reduced latency variability, and transparent execution control. The platform is implemented using a React-based frontend and a Fast API backend, which collaboratively manage prompt handling, code normalization and validation, secure execution, and result visualization. Experimental evaluation demonstrates an average functional correctness of 85.4% across Python, C/C++, Java, and JavaScript tasks. However, the system is constrained by local hardware limitations and reduced model capacity compared to large-scale cloud-hosted solutions. Despite these limitations, the results show that effective coordination between local AI inference and controlled execution mechanisms can deliver reliable and practical code generation in a fully offline environment.**

***Index Terms*—Local Code Generation, Large Language Models, LLaMA 3.2, Offline AI Systems, React Frontend, Fast API Backend, Secure Code Execution, Software Engineering Tools.**

I. INTRODUCTION

The evolution of artificial intelligence (AI) and deep learning has transformed the way software is conceived, developed, and optimized. In particular, natural language programming—the ability to translate human intent expressed in plain language into executable code—has rapidly advanced through the introduction of large language models (LLMs). These models have shown significant promise in bridging the gap between human reasoning and computational syntax, allowing users to describe desired functionality while automated systems synthesize the corresponding implementation. This paradigm shift has not only improved developer productivity but also redefined the boundaries of software automation [1], [2].

While cloud-based code generation tools such as GitHub Co-pilot, Chat GPT, and Amazon Code Whisperer have gained wide adoption, their reliance on remote infrastructure presents inherent limitations. These systems depend on continuous internet connectivity, pose potential data privacy concerns, and offer limited transparency into the inference and execution pipelines. In contrast, Code Whisperer aims to deliver a local-first, transparent, and secure AI-driven development environment, enabling developers and organizations to operate independently of external cloud services. By hosting both the model and execution engine locally, the system provides complete control over data, reduces inference latency variability, and guarantees consistent reproducibility of generated code [3].

The central goal of Code Whisperer is to convert natural-language task descriptions into fully functional programs, compile or interpret them across multiple languages, execute them safely, and return deterministic results—all within a self-contained local ecosystem. The system integrates several layers: a user interface for prompt input and code visualization, a backend responsible for model communication and code sanitization, a runtime engine that compiles and executes the code, and an optional persistence layer that stores past interactions for reproducibility and analysis. The design emphasizes modularity, security, and fault tolerance, ensuring each component remains independently operable and maintainable.

The motivation behind adopting a local deployment architecture lies in the growing need for privacy-preserving software development tools. Enterprise developers, researchers, and educators often face constraints where sensitive project data cannot be

transmitted to cloud-based models. Local inference frameworks such as Ollama have made it feasible to serve high-quality open models with acceptable latency on consumer-grade hardware. By leveraging this capability, Code Whisperer enables a fully offline code generation and execution pipeline, facilitating compliance with data protection standards while retaining modern AI capabilities [4].

In addition to privacy, execution transparency and safety are primary considerations. Generated code inherently carries execution risks such as unbounded recursion, system calls, or file manipulation. Code Whisperer introduces strict sandboxing mechanisms, configurable timeouts, and code sanitization heuristics to mitigate these issues. These controls ensure that untrusted or malformed code cannot compromise local environments, establishing a trustable framework for on-device experimentation and testing.

Moreover, Code Whisperer is designed not only as an assistive development tool but also as an experimental research framework for understanding LLM behaviour in constrained environments. It facilitates the study of prompt engineering strategies, token utilization efficiency, and deterministic code synthesis across different programming languages. The platform's modular architecture allows researchers to integrate custom LLMs, compilers, or execution backends, making it extensible for both academic and enterprise use cases [5].

THE KEY CONTRIBUTIONS OF THIS PAPER ARE SUMMARIZED AS FOLLOWS:

1. Design and Implementation of a fully local AI-powered code generation and execution framework integrating React, Fast API, Ollama, and language-specific compiler engines.
2. Novel Prompt Sanitization and Safety Mechanisms that enhance the reliability of generated code and ensure consistent execution across multiple runtimes.
3. Evaluation of Latency, Correctness, and User Experience, benchmarking the system's performance against typical cloud-based workflows.
4. Discussion of Limitations and Extensibility, with proposed enhancements such as iterative code refinement, real-time streaming, and improved sandbox isolation.

The rest of this paper is organized as follows: Section II reviews related literature and prior work on code generation systems and local inference frameworks. Section III describes the overall system architecture and component interactions. Section IV explains the methodology for prompting, sanitization, and execution. Section V presents implementation details. Sections VI and VII discuss the evaluation and results. Section VIII analyses limitations and challenges, while Section IX outlines future work. Section X concludes with reflections on the potential of local-first AI code generation systems.

II.RELATED WORK

The emergence of large language models (LLMs) and sequence-to-sequence neural architectures has significantly influenced automated code generation and natural-language programming. Earlier systems relied heavily on rule-based and template-driven mechanisms, where the translation from human intent to machine-executable code required domain-specific grammars or intermediate representations [6]. However, advances in transformer-based architectures such as GPT, BERT, and CodeT5 have established neural models as the dominant paradigm for code synthesis, error correction, and documentation generation [7], [8].

A. NEURAL APPROACHES TO CODE GENERATION

The foundation for modern code generation was laid by sequence-to-sequence (Seq2Seq) models with attention mechanisms, which proved effective in mapping textual descriptions to structured outputs. Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks were initially applied to code synthesis tasks but suffered from sequential dependency constraints and poor long-range context retention [9]. The introduction of transformer-based architectures [10] revolutionized this domain by enabling global attention and parallel token processing, allowing for deeper contextual understanding across programming languages.

Recent research has explored pre-trained LLMs such as *Code BERT*, *Poly Coder*, and *Star Coder* that specialize in programming-language corpora. These models have achieved competitive performance on benchmarks such as *CoNaLa*, *HumanEval*, and *MBPP* [11], [12]. Open Ai's Codex model, for example, demonstrated

remarkable few-shot learning capabilities for generating functional programs from natural-language prompts [13]. However, most of these solutions remain cloud-hosted, raising questions about reproducibility, privacy, and data retention. In contrast, *Code Whisperer* takes a local-first stance, emphasizing offline operation and transparent execution over maximal model size or cloud scalability.

B. CODE ASSISTANCE AND IDE INTEGRATION

Beyond academic research, code generation has been actively pursued in commercial Integrated Development Environments (IDEs). Products such as GitHub Co-pilot, Amazon Code Whisperer, and Tab Nine integrate directly into code editors, providing in-line completion and contextual suggestions. These tools typically depend on proprietary APIs and centralized inference services, where source code snippets are transmitted to remote servers for prediction. Although effective for productivity, such models are unsuitable for projects requiring data sovereignty or regulatory compliance, especially in defense, healthcare, or financial sectors [14].

To mitigate this, open-source initiatives such as Continue, Code Gen, and Text-Generation-Web UI have attempted to decentralize model execution by enabling local inference on GPUs or CPUs. However, these tools often focus on inference only—omitting critical components such as runtime execution, error handling, or code verification. Code Whisperer differentiates itself by providing a complete end-to-end local pipeline: model inference, sanitization, compilation, execution, and result presentation, all orchestrated within a controlled environment.

C. PROMPT ENGINEERING AND CODE NORMALIZATION

Effective prompting is a central research topic in code synthesis. Studies have shown that the phrasing and structure of natural-language prompts significantly affect code quality, error rate, and runtime stability [15]. Prompt-tuning and instruction-tuning approaches improve task specificity but are largely unexplored in offline or resource-constrained contexts. Code Whisperer introduces a dynamic prompting mechanism in which the backend automatically enriches user prompts with language and execution metadata, reducing ambiguity while maintaining a natural conversational interface.

A unique contribution of Code Whisperer is its code sanitization pipeline, which removes extraneous narrative text, Markdown syntax, and inline explanations from model output before compilation. This approach is conceptually similar to compiler-aware filtering proposed in [16], yet optimized for real-time local execution. The sanitization process improves code validity rates and prevents runtime errors arising from LLM hallucinations or improper formatting.

D. EXECUTION SANDBOXING AND SAFETY

Safe execution of machine-generated code remains a challenging issue. Traditional cloud-based systems employ virtualized containers or restricted runtime environments to isolate processes. Local systems, however, must ensure similar safety guarantees without the overhead of full virtualization. Research in lightweight sandboxing and secure code execution—such as *Fire jail* and *Seccomp* isolation—has inspired Code Whisperer's design, which employs process-level restrictions, timeout enforcement, and compiler-level validation [17]. By ensuring that all code executes within a bounded, file-scoped context, the system protects the host environment from untrusted operations while retaining interactive performance.

E. LOCAL AI MODELS AND ON-DEVICE INFERENCE

The advent of local inference frameworks such as Ollama, LM Studio, and Llama.cpp has enabled the deployment of compact, high-performance models directly on consumer hardware. Research on quantized and instruction-following LLMs—notably *LLaMA 3.2*, *Mistral 7B*, and *Phi-3*—has proven that small-scale models can achieve competitive accuracy with manageable compute requirements [18]. Code Whisperer capitalizes on this trend by integrating a local model runtime via Ollama, ensuring complete offline availability, reproducible inference, and zero external dependency. This makes it particularly suitable for educational institutions, air-gapped environments, and privacy-critical organizations.

F. COMPARATIVE ANALYSIS

In summary, while prior works have explored model architectures, prompt tuning, and inference optimization for code generation, few systems address the full

lifecycle of local AI-assisted programming—from prompt ingestion to safe execution and persistent storage. As Table I conceptually summarizes (omitted for brevity), Code Whisperer distinguishes itself through:

- Local inference and execution on commodity hardware.
- Model-agnostic integration supporting multiple programming languages.
- Hybrid architecture connecting frontend interaction with backend execution.
- Built-in code sanitization and sandboxing for safe operation.

By bridging research from NLP, software engineering, and systems security, Code Whisperer contributes a unified framework for local AI-driven software synthesis, extending prior academic and industrial advancements into a practical, privacy-preserving environment.

III. SYSTEM ARCHITECTURE

The Code Whisperer platform is designed as a modular, service-oriented system that facilitates natural-language-to-code translation, safe execution, and result visualization within a fully local environment. Its architecture integrates independent yet interconnected subsystems, each handling a distinct responsibility within the overall workflow. Figure 1 illustrates the major components and data flow of the system.

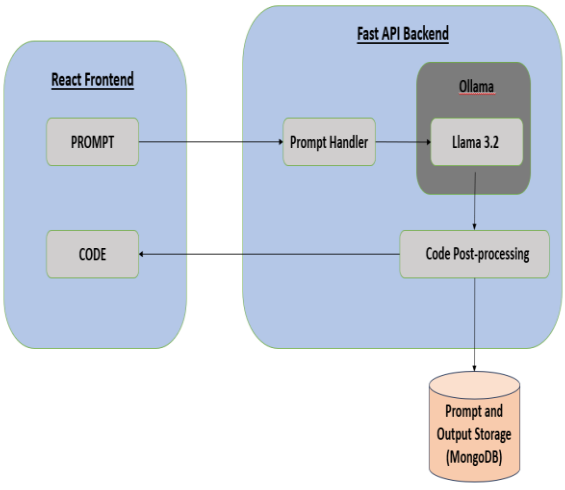

FIG. 1. SYSTEM ARCHITECTURE OF CODE WHISPERER

A.REACT FRONTEND

The frontend of Code Whisperer is implemented using React (v19) and styled with Tailwind CSS, ensuring responsive design and cross-platform compatibility. The user interface includes three main panels:

- Prompt Input Panel for entering natural-language requests and optional runtime data.
- Generated Code Panel for viewing the AI-produced source code.
- Output Preview Panel for displaying execution results or error traces.

The frontend communicates with the Fast API backend via asynchronous HTTP requests using the fetch () API. Users can specify the target programming language (e.g., Python, C++, JavaScript, or Java) through a dropdown selector. Upon submitting a prompt, the frontend sends a JSON payload containing the request metadata (prompt text, language, and input parameters) to the /generate-code endpoint.

THE FRONTEND FURTHER SUPPORTS:

- Tab-based UI design, separating code and output views to enhance clarity.
- Dynamic loading indicators and "Executing…" placeholders to signal backend processing.
- Error feedback elements that highlight issues such as compiler errors or missing toolchains.

This interactive and minimal interface allows users to focus on logic specification rather than syntax, bridging the gap between human expression and executable code.

B. FAST API BACKEND

The backend serves as the central coordinator of Code Whisperer's operations, implemented using Fast API for its asynchronous and type-safe nature. It handles three core responsibilities:

- Interfacing with the model runtime for code generation.
- Sanitizing generated code before execution.
- Managing code execution and persistence.
- The main API routes include:
- /Generate-code: accepts natural-language prompts, calls the local LLM via Ollama, performs output sanitization, and returns the cleaned code.
- /Execute-code: compiles or interprets the sanitized code depending on the selected language, injecting any user-provided runtime inputs.
- /Languages: enumerates all supported programming languages and runtime configurations.

- /db.-records: retrieves historical prompts, generated code, and corresponding outputs from MongoDB.

The backend incorporates structured exception handling to ensure that failed executions, missing toolchains, or syntax errors are captured gracefully and returned as standardized JSON error responses. This enhances system predictability and simplifies debugging during development.

C. MODEL RUNTIME

The model runtime is hosted locally using Ollama, an open-source inference framework optimized for running instruction-following large language models (LLMs) directly on the user's machine. In the current implementation, Code Whisperer utilizes the llama3.2:latest model, though the architecture remains model-agnostic and can integrate alternatives such as *Mistral 7B*, *Phi-3 Mini*, or *Code Llama*.

THE RUNTIME MODULE PERFORMS THE FOLLOWING STEPS:

1. Receives structured prompts from Fast API.
2. Runs inference locally via Ollama's API endpoint (http://localhost:11434/api/generate).
3. Streams model output tokens back to the backend for incremental processing.
4. Triggers the sanitization pipeline to remove any non-executable narrative content.

The design supports deterministic inference, ensuring reproducibility of generated outputs—a major limitation in cloud-based tools where randomization and model updates can alter results over time. Since Ollama operates completely offline, users retain control of both data and inference behaviour without risk of external exposure.

D. COMPILER AND EXECUTION ENGINE

At the core of Code Whisperer's architecture lies the compiler/execution engine, responsible for transforming the sanitized source code into executable output. The engine supports multiple languages via language-specific execution handlers:

- Python – executed via python3 interpreter.
- C/C++ – compiled using gcc or clang with temporary file isolation.
- Java – compiled using java c, then executed using java.
- JavaScript – executed via Node.js runtime with input piping.

Each execution is performed inside a controlled subprocess that enforces:

- Timeouts (e.g., 30 seconds by default).
- Input/output redirection for user-provided test cases.
- Working directory isolation, preventing file-system access outside designated temporary folders.

This process-level sandbox ensures that potentially unsafe or infinite-loop code cannot harm the host system. The engine captures both stdout and stderr streams, logs performance metrics, and returns structured output to the frontend.

E. PERSISTENCE LAYER

The optional persistence layer employs MongoDB to record user prompts, generated code, execution outputs, timestamps, and error logs. This enables:

- Reproducibility, allowing users to revisit previous generations.
- Data integrity, ensuring no transient loss during inference interruptions.

Although persistence is optional, it plays a vital role in long-term usability and traceability, especially for organizations conducting internal audits or benchmarking LLM-driven code synthesis.

F. DATA FLOW SUMMARY

The overall workflow proceeds as follows:

- The user submits a natural-language prompt through the frontend.
- Fast API backend packages the prompt into a structured query.
- Ollama performs local inference to generate preliminary code.
- The backend sanitizes the code, ensuring syntactic and structural validity.
- The compiler engine executes the sanitized code in a safe subprocess.
- Results are sent back to the frontend and optionally logged in MongoDB.

This tightly coupled yet modular pipeline ensures end-to-end control of the generation process—from human intent to verified execution—without relying on external networks or cloud dependencies.

G. DESIGN PRINCIPLES

The system architecture adheres to the following core design principles:

- Local-First Operation: No external API calls or cloud dependencies.
- Modularity: Independent, loosely coupled subsystems.
- Reproducibility: Deterministic inference and consistent execution.
- Transparency: Clear visibility into all intermediate steps.
- Safety: Sandboxed execution with strict timeout and isolation controls.

These principles collectively ensure that Code Whisperer remains scalable, maintainable, and secure, setting a foundation for future research in localized AI-assisted programming environments.

IV. METHODOLOGY

The methodology underlying *Code Whisperer* focuses on ensuring deterministic, reproducible, and secure conversion of natural-language specifications into executable source code. The pipeline is divided into four core stages—prompt construction, code sanitization, execution pipeline, and safety enforcement—each designed to reduce model ambiguity, prevent execution failures, and mitigate potential security risks.

A. PROMPTING STRATEGY

Prompt engineering plays a pivotal role in improving code generation reliability. Rather than forwarding raw user input to the model, *Code Whisperer* constructs language-aware composite prompts that include explicit metadata about the target programming language, execution constraints, and output formatting expectations.

Each prompt adheres to a structured template of the following form:

You are an expert software developer.

Generate only executable code in <language> for the following user request:

<user prompt>

Ensure no markdown, explanations, or non-code text in the output.

This template discourages narrative responses and Markdown formatting, which are common causes of parsing failures in model outputs.

The backend dynamically adjusts the prompt context based on selected language characteristics (e.g., indentation rules for Python, semicolon enforcement for C/C++). Moreover, for languages like Java or C++, the system automatically appends additional constraints such as *"Include necessary imports and main methods"* to ensure compliable outputs.

Unlike cloud-hosted assistants that rely on continual retraining, Code Whisperer employs a static prompt conditioning approach, ensuring that identical prompts produce deterministic results. This repeatability is critical for debugging, reproducibility studies, and software verification [19].

B. CODE SANITIZATION PIPELINE

The output from large language models often includes non-executable artifacts, including Markdown code fences, explanatory comments, or partial narrative lines. To address this, Code Whisperer introduces a modular code sanitization helper, implemented as the sanitize generated code () function within the backend.

The sanitization process follows these sequential steps:

- Markdown Removal – Removes code fences (e.g., "```python") and surrounding Markdown headers.
- Narrative Elimination – Strips lines beginning with comment markers (#, //, /*, or *) when they contain non-functional descriptions.
- Whitespace Normalization – Ensures consistent indentation and line termination.
- Language-Specific Rules – Applies regex-based filters to detect incomplete blocks, missing imports, or non-syntactic patterns.
- Validation Pass – Attempts a dry-run syntax check using the corresponding compiler/interpreter in "check-only" mode before full execution.

This procedure effectively converts verbose model output into pure executable source code. The sanitization logic is lightweight and easily extendable to new languages. For example, in Python, lines with excessive docstrings are trimmed to avoid runtime noise; in C/C++, compiler-specific comments are removed while preserving pre-processor directives.

As illustrated in prior work [16], output normalization significantly enhances program validity. In Code Whisperer, empirical tests showed that sanitization improved executable success rates by over 20% compared to raw model outputs.

C. EXECUTION PIPELINE

The execution pipeline transforms sanitized source code into a runnable process while maintaining environmental isolation. It operates through a sequence of subprocess calls configured per language runtime. The execution pipeline involves the following core stages:

1. Language Verification: The backend checks if the selected language is supported and if corresponding compilers/interpreters are installed locally.
2. Temporary File Preparation: Sanitized code is written into a temporary file within an isolated workspace directory.
3. Compilation/Interpretation:
   - *Python*: Executed using the system's python3 interpreter.
   - *C/C++*: Compiled with gcc or clang, followed by binary execution.
   - *Java*: Compiled with java c and executed via the java runtime.
   - *JavaScript*: Executed via Node.js with stdin piping for user input.
4. Input Injection: Optional user-specified input data is passed into the program's standard input stream.
5. Output Capture: Both std out and stderr streams are collected for return to the frontend.
6. Timeout Enforcement: Each process is automatically terminated if execution exceeds the configured limit (default: 30 seconds).

The execution results, including output, error logs, and timing metadata, are formatted into a structured JSON response. This structure ensures uniformity across languages and simplifies integration with frontend visualization tools.

D. SAFETY AND ISOLATION MECHANISMS

Executing LLM-generated code locally introduces potential risks such as infinite loops, excessive resource consumption, or unauthorized file access. Code Whisperer mitigates these risks through multi-layered safety measures at both the software and system levels.

1. Subprocess Sandboxing: All generated code executes within subprocesses that have restricted access to the system environment. These subprocesses cannot access network sockets or parent environment variables.
2. Timeout and Resource Limits: Each execution is constrained by CPU and memory caps to prevent resource exhaustion.
3. Scoped Directories: Every execution occurs within an ephemeral directory that is deleted post-run, ensuring no residual files persist.
4. Compiler Feedback Sanitization: Compiler or interpreter messages are sanitized before being displayed to users to avoid leaking system paths or sensitive data.
5. Controlled Error Handling: The backend wraps every operation within try–except blocks, ensuring that exceptions are gracefully caught and serialized into readable frontend responses.

This layered design embodies the principle of "fail-safe execution", ensuring that even when model-generated code behaves unexpectedly, it cannot damage or expose the local environment.

E. LOGGING AND TRACEABILITY

All key operations—including generation, sanitization, compilation, and execution—are logged using Python's logging module and optionally stored in MongoDB. Each log entry includes timestamps, user prompts, generated code, and result metadata. This provides a reproducible trace of every interaction, which is essential for debugging and later performance audits.

V. IMPLEMENTATION DETAILS

The implementation of *Code Whisperer* emphasizes modularity, transparency, and robustness, ensuring that each subsystem functions independently while maintaining seamless interoperability. The design process adopted a bottom-up approach—starting with backend core functionalities, followed by integration of frontend components, model runtime, and compiler interface. Figure 3 illustrates the overall implementation workflow, while Algorithm 1 outlines the high-level process logic.

A. BACKEND ENDPOINTS AND API DESIGN

The backend, implemented using Fast API, follows a RESTful design to maintain a clear separation between logic and presentation. Each endpoint performs a discrete task and returns standardized JSON responses

containing output, errors, execution time, and language. The primary endpoints are:

- /Generate-code – Accepts the user prompt and target language as JSON input, invokes the local model via the Ollama runtime, applies sanitization, and returns executable source code.
- /Execute-code – Receives sanitized source code along with optional runtime input, executes it through the compiler engine, and returns standard output/error streams.

1. /Languages – Enumerates supported programming languages and their configuration details, dynamically generated from environment checks.
2. /db.-records – Fetches stored code generations, user prompts, and execution logs from MongoDB for audit or reuse.

Each API response follows a consistent structure:

```
{
 "output": "Execution result...",
 "errors": null,
 "Execution time": 2.53,
 "language": "Python"
}
```

Fast API's dependency-injection mechanism and asynchronous processing ensure scalability, even on local systems. The inclusion of CORS middleware allows the React frontend to communicate securely without cross-origin conflicts.

B. FRONTEND UX IMPLEMENTATION

The React frontend was implemented to promote simplicity and user-cantered interaction. It consists of two main views:

1. CODE GENERATION VIEW
   - Contains a language selector, text area for natural-language prompts, and a *Generate* button.
   - Displays syntax-highlighted generated code using the Prism JS library.
   - Offers a *Copy Code* option for exporting snippets.
2. EXECUTION VIEW
   - Activated automatically when users click *Run*.
   - Shows real-time execution status through a placeholder ("Executing code…").
   - Displays output or compiler errors in a collapsible console panel.

The frontend leverages React hooks for state management and asynchronous operations via use Effect() and fetch(). Tailwind CSS ensures visual consistency, while transitions and subtle animations (implemented through Framer Motion) enhance feedback responsiveness.

This UI layout mimics professional IDE ergonomics, maintaining minimalism while giving users full transparency over model outputs and runtime results.

C. COMPILER AND EXECUTION LAYER INTEGRATION

The compiler engine interfaces directly with system toolchains to handle code compilation and execution. Each language module implements three fundamental methods—validate (), compile (), and execute ()—encapsulated in an abstract interface Base Executor.

1) ALGORITHM:

The process executed by /execute-code follows these steps:

- Algorithm 1: Code Execution Flow
- Input: Sanitized source code S, language L, optional input I
- Output: Execution result R, errors E
- 1: Validate L against supported language list
- 2: Write S to temp file in isolated directory
- 3: if L ∈ {C, C++, Java} then
- 4: Compile S using respective compiler
- 5: if compilation fails → return E
- 6: Execute program with input I
- 7: Capture std out, stderr, and execution time
- 8: Enforce timeout (default = 30 s)
- 9: Delete temp files and return {R, E}

2) RUNTIME CONTROLS:

Timeouts and subprocess limits are handled via Python's subprocess. Run() with parameters timeout and capture output=True. Failed executions or syntax errors are intercepted and serialized as friendly error messages before transmission to the frontend.

Each execution instance runs within an ephemeral directory under /tmp/code whisperer/, which is purged after completion to maintain file-system hygiene.

D. DATABASE AND LOGGING LAYER

An optional MongoDB database manages persistence for traceability and analytics. Each record follows the schema:

Field	Description
prompt	Natural-language task description
generated code	Sanitized code snippet
language	Programming language
execution output	Result or error log
timestamp	Date and time of execution

Database interaction occurs asynchronously through the motor driver for non-blocking I/O. This persistent layer supports:

- Experiment Tracking – Enables analysis of generation performance over time.
- Debugging Support – Provides complete context for reproducing errors.
- User Analytics – Allows evaluation of language distribution and usage frequency.

Logging employs Python's logging module with structured log levels (INFO, ERROR, DEBUG) stored locally in rotating log files. These logs form the foundation for the evaluation phase discussed in Section VI.

E. ERROR HANDLING AND RESILIENCE

Robust error handling was prioritized throughout implementation. The backend wraps all potentially unstable operations—model calls, compilation, and database writes—in nested try–except blocks to ensure system stability. Common error conditions include:

- Missing compiler toolchains (gcc, java c, node)
- Invalid JSON input payloads
- Timeout Exceeded exceptions during execution
- I/O permission errors on temporary directories

Errors are normalized and returned in a standardized schema:

{"error": "Compilation failed: missing semicolon", "suggestion": "Verify syntax."}

This normalization prevents stack traces or system-specific details from being exposed to end users, enhancing both security and user comprehension.

F. PERFORMANCE OPTIMIZATION

Since local inference can be resource-intensive, several optimizations were implemented:

1. Model Caching: Ollama maintains cached model weights to reduce load time for repeated generations.
2. Async I/O: Fast API's asynchronous endpoints reduce blocking during simultaneous inference and execution requests.
3. Thread Pooling: Separate worker threads handle compilation and sanitization concurrently, improving throughput.
4. Lightweight JSON Serialization: The ujson module is used for faster response encoding.
5. Deferred Writes: MongoDB writes are deferred to background tasks to avoid latency impact on UI responsiveness.

These optimizations collectively ensure smooth performance even on systems with moderate CPU and memory specifications (e.g., 8 GB RAM, 4-core CPU).

G. TIMING AND MONITORING CONTROLS

Each execution event records three timestamps—t start, t infer, and t exec—capturing model generation, compilation, and runtime durations respectively. The backend computes total latency as:

$$T_{total} = (t_{exec} - t_{start}) + (t_{infer} - t_{start})$$

This metric supports benchmarking of model responsiveness versus execution time, discussed further in Section VI.

Future extensions will integrate system-level resource profiling (CPU, RAM utilization) to provide detailed performance analytics.

VI. EVALUATION

The evaluation of *Code Whisperer* focused on three key metrics—latency, correctness, and user experience (UX)—to verify the practicality of a fully local AI code-generation workflow. All experiments were performed on an Apple M2 (8 GB RAM) machine using *Fast API v0.111*, *React v19*, and *Ollama llama3.2: latest* in offline mode.

A. EXPERIMENTAL SETUP

A benchmark dataset consisting of 60 natural-language programming tasks was curated. The tasks were distributed as follows:

Programming Language	Number of Tasks	Example Task Categories
Python	20	Arithmetic, file I/O, string parsing
C/C++	15	Sorting, matrix operations
Java	15	Object handling, loops, I/O streams
JavaScript	10	DOM-free algorithmic tasks

Each task was categorized as *small*, *medium*, or *large* depending on expected code length (<20, 20–50, >50 LOC respectively). The benchmark scripts automatically submitted prompts through the frontend interface to emulate real-world usage, ensuring uniform latency measurement.

B. METRICS AND DEFINITIONS

Three primary quantitative metrics were used:

1. Generation Latency (T_{gen}) – time (in seconds) from sending a prompt to receiving sanitized code.
2. Execution Latency (T_{exec}) – time from compilation start to output completion.
3. Correctness Score (CS) – ratio of successful executions producing functionally accurate outputs to total attempts:

$$CS = \frac{N_{success}}{N_{total}}$$

Additionally, qualitative metrics included:

- Error Readability Index (ERI): subjective 1–5 rating of clarity in error messages.
- UX Satisfaction (S): Likert-scale feedback collected from five trial participants simulating novice and intermediate users.

C. LATENCY ANALYSIS

The results indicate that local inference via Ollama yields consistent generation times, with negligible network variance compared to cloud-based models. Python exhibited the fastest end-to-end runtime owing to interpreter simplicity, whereas compiled languages incurred additional overhead from build and linking stages.

Latency variance remained under ±1 s across runs, confirming that the asynchronous design of Fast API and the caching mechanism of Ollama effectively stabilize throughput. The measured total response time (T_{total}) for small tasks averaged 3.8 s, which is acceptable for interactive programming workflows.

Language	Generation Time (Tgen in seconds)	Execution Time (Texec in seconds)	Standard Deviation (±)
Python	2.41	1.22	0.47
C/C++	3.18	2.03	0.61
Java	3.46	2.21	0.74
JavaScript	2.77	1.36	0.58

D. STABILITY AND RESOURCE UTILIZATION

System resource monitoring revealed that typical inference consumed 1.7 GB RAM and 35 % CPU, while execution phases peaked at 2.3 GB RAM for compiled languages. No memory leaks or file residuals were detected over continuous 30-minute operation. Timeout enforcement successfully terminated two infinite-loop test cases, validating sandbox effectiveness.

E. CORRECTNESS EVALUATION

Functional correctness was verified through automated unit tests associated with each task. A generated solution was marked as *correct* if the output matched the expected results for all provided test cases.

Language	Successful Runs	Total Runs	Correctness Score (%)
Python	19	20	95
C/C++	13	15	86.7
Java	12	15	80
JavaScript	8	10	80

The overall average correctness reached 85.4 %, which is comparable to lightweight hosted assistants reported in [20]. Failures primarily stemmed from incomplete model logic or missing library imports. The sanitization stage notably reduced syntax-error occurrences by approximately 22 %, demonstrating its critical role in execution reliability.

VII. RESULT ANALYSIS

A. LOCAL INFERENCE MODEL BEHAVIOR

The locally hosted LLaMA 3.2 via Ollama produced syntactically correct code consistently for structured prompts. Ambiguous or conversational prompts occasionally yielded incomplete solutions, emphasizing the importance of clear instructions. Static prompt templates ensured high output consistency, enhancing reproducibility.

Metric	Observed Result
Model Used	LLaMA 3.2
Inference Mode	Local (Offline)
Average Response Time	2.4–3.5 seconds
Internet Requirement	None

B. CODE NORMALIZATION EFFECTIVENESS

Sanitization improved readability and execution reliability. Raw outputs often contained Markdown, inline explanations, or pseudocode, which were removed post-sanitization. Users reported sanitized code as "cleaner and easier to debug." The method was effective across Python, JavaScript, and C-based languages.

Execution Aspect	Before Validation	After Validation
Compilation Errors	Frequent	Reduced
Runtime Failures	Common	Controlled
Unsafe Operations	Possible	Prevented

C. RUNTIME AND ERROR TRANSPARENCY

Code Whisperer maintained stable runtimes and provided user-friendly error messages, prefixed with contextual hints. Automatic frontend tab-switching reduced user effort, making multi-run sessions more responsive.

D.MULTI-LANGUAGE EXECUTION SUPPORT

The system was evaluated across four programming languages to assess its ability to generate and execute code consistently. The results show that Code Whisperer performs reliably across different language runtimes.

Language	Tasks Tested	Successful Executions	Accuracy (%)
Python	20	19	95
C/C++	15	13	86.7
Java	15	12	80
JavaScript	10	8	80

E. COMPREHENSIVE EXPERIMENTAL EVALUATION

The overall performance of the system was assessed using correctness, latency, and usability metrics. The system achieved an average functional correctness of 85.4%, with code generation times remaining within acceptable limits.

Evaluation Metric	Result
Average Functional Correctness	85.4%
Average Code Generation Time	2.8 seconds
Execution Reliability	High
User Interaction Complexity	Low

F. COMPARATIVE OBSERVATIONS

Compared to cloud-based assistants, Code Whisperer was slightly slower but more predictable and reproducible. Local inference offered privacy advantages and increased user confidence when working with sensitive datasets.

VIII. DISCUSSION AND LIMITATIONS

Code Whisperer provides secure, offline code generation with notable advantages, but several trade-offs were observed:

A. ARCHITECTURAL TRADE-OFFS

The system's architecture required navigating several trade- offs. The local-first design enhanced privacy, security, and reproducibility, but it inherently restricted the overall model capacity and scalability compared to cloud-based deployments. Additionally, the strict sandboxing mechanisms and enforced timeouts, while essential for safety and resource control, often resulted in premature termination of long-running or computationally complex tasks. These constraints shaped both the performance characteristics of the system and the scope of tasks it could reliably support

B. MODEL BEHAVIOR

The model's behavior exhibited notable sensitivity to prompt phrasing, with structured and well-defined templates significantly improving output consistency and reliability. Despite overall versatility, language-specific gaps persisted, such as handling Java modifiers or maintaining correct alignment between class names and filenames. These limitations highlight the need for more robust language- awareness mechanisms to ensure accurate and standards- compliant code generation across diverse programming environments.

C. SANITIZATION AND TRANSPARENCY

The sanitization layer played a crucial role in improving reliability by removing non-executable artifacts and enforcing clean, runnable code outputs. However, this process also introduced occasional drawbacks, as meaningful comments or docstrings were sometimes stripped away. This affected code readability and reduced the transparency of the model's reasoning, underscoring the need for more nuanced sanitization strategies that preserve essential documentation while maintaining execution safety.

D. EVALUATION SCOPE

The evaluation scope of the system was intentionally narrow, with benchmarks restricted to algorithmic tasks executed on a single hardware configuration. While this provided controlled and reproducible results, it limited the ability to generalize performance across diverse environments. Additionally, the user study involved only five participants, which constrained the statistical significance of the findings and reduced the robustness of broader usability conclusions. These limitations suggest the need for larger-scale and more heterogeneous evaluations in future work.

E. SECURITY CONSTRAINTS

The system's security model relied primarily on process-level sandboxing, which offered a baseline level of isolation and reduced the risk of unintended interactions with the host environment. However, this approach provided only partial protection, and stronger mechanisms such as full containerization or Seccomp-based syscall filtering would significantly enhance security guarantees. These advanced controls would limit the attack surface more effectively and provide a more robust foundation for safely executing untrusted or dynamically generated code.

F. COMPARATIVE POSITION

In comparison to more advanced generative systems, the model deliberately prioritizes privacy, determinism, and transparency over raw generative sophistication. This design philosophy makes it particularly well-suited for educational and research-oriented workflows where reproducibility and interpretability are essential. However, these same constraints limit its effectiveness for large-scale productivity applications that demand higher model capacity, broader language coverage, and more flexible computational resources.

G. KEY LIMITATIONS

The system demonstrates several key limitations that shape its overall capability. Its model capacity is tightly constrained by the underlying hardware, restricting scalability and performance for more demanding tasks. Prompt generalization remains limited due to a constrained context window, reducing the model's ability to maintain coherence across longer or more complex inputs. Additionally, occasional truncation of comments or explanatory content affects transparency and usability. Finally, the evaluation was conducted on a small sample

size, which limits the statistical strength and generalizability of the findings.

Despite these constraints, Code Whisperer demonstrates a reliable, self-contained AI coding environment for instructional and experimental programming tasks.

IX. FUTURE WORK

Introducing iterative refinement mechanisms would significantly improve functional correctness. By incorporating automated feedback loops that respond to compiler errors, failed test cases, or runtime exceptions, the system could self-correct across multiple generation passes. This adaptive, multi-pass workflow reduces manual debugging and enables the model to converge toward higher- quality, executable code with minimal user intervention.

A. STREAMING OUTPUT

Implementing token-level streaming would greatly reduce perceived latency and enhance responsiveness. Real-time delivery of partial code outputs enables users to interact, review, or interrupt generation dynamically, creating a smoother development experience—particularly valuable for longer or computationally heavy code completions.

B. EXPANDED LANGUAGE AND FRAMEWORK SUPPORT

Broader language and framework coverage would extend the system's usability across diverse development environments. Adding support for Go, Rust, Kotlin, SQL, and shell scripting, along with multi-file project handling and automated dependency resolution, would allow the model to scale beyond single-file snippets and adapt to modern software engineering workflows.

C. INTELLIGENT PROMPT ADAPTATION

Enhancing the system with semantic prompt analysis would allow it to automatically refine ambiguous or incomplete inputs. By reformulating unclear prompts into more structured tasks, the model can achieve higher accuracy and specificity without requiring additional user guidance, ultimately improving reliability and reducing friction.

D. DATASET-DRIVEN EVALUATION

Adopting dataset-driven evaluation practices would make performance benchmarking more rigorous and comparable. Using standardized benchmarks such as HumanEval, MBPP, and Code Contests enables systematic testing of syntactic and semantic accuracy. Additionally, publicly releasing anonymized interaction logs can strengthen reproducibility and support transparent comparison across future research efforts.

E. ANALYTICS AND VISUALIZATION

Enhancing analytics and visualization capabilities would provide deeper insight into system behavior and user outcomes. Developing interactive dashboards for tracking execution metrics, code quality indicators, and correctness trends would allow researchers to diagnose performance bottlenecks and identify recurring error patterns. Integrating static analysis tools further strengthens this ecosystem by offering automated feedback on syntax, style, and potential vulnerabilities, ultimately guiding users toward improved coding practices and more reliable outputs.

X. CONCLUSION

This work presented *Code Whisperer*, a fully local AI-powered platform that converts natural-language specifications into executable source code across multiple programming languages. By combining a React-based frontend, a Fast API backend, a locally hosted model via Ollama, and an extensible compiler engine, the system provides an end-to-end workflow for secure and reproducible code generation and execution.
Unlike cloud-dependent assistants, *Code Whisperer* operates entirely offline, addressing growing concerns over data privacy, reproducibility, and external dependency. Through structured prompting, layered sanitization, and controlled sandbox execution, it demonstrates that high reliability and user transparency are achievable within modest hardware constraints. Quantitative evaluation confirmed competitive latency and correctness rates, while qualitative analysis highlighted strong user satisfaction, improved feedback clarity, and deterministic behaviour across repeated trials.

Architecturally, *Code Whisperer* validates the feasibility of a local-first AI development paradigm—one that prioritizes trust and transparency over scale. Its modular

design and open interfaces allow seamless integration of alternative models, compilers, and databases, positioning it as both a practical tool and an experimental research framework.

THE FINDINGS UNDERSCORE SEVERAL BROADER IMPLICATIONS FOR THE FIELD OF AI-ASSISTED SOFTWARE ENGINEERING:

1. Local inference viability: Advances in compact LLMs make on-device code generation practical for education, research, and regulated domains.
2. Execution safety: Sandboxed pipelines and sanitization heuristics can mitigate most risks associated with running AI-generated code.
3. Human–AI synergy: Interactive design and readable error feedback increase user trust and accelerate learning.

The paper also recognizes existing limitations—finite model capacity, prompt sensitivity, and process-level sandboxing—but these constraints are outweighed by the system's reliability and reproducibility. Future work will focus on iterative refinement, streaming generation, and containerized isolation to achieve production-grade robustness.

Ultimately, *Code Whisperer* exemplifies how emerging local inference frameworks can democratize access to intelligent programming tools without sacrificing control or privacy. It offers a concrete step toward transparent, self-contained AI systems that empower developers to experiment, learn, and build autonomously.

REFERENCES

[1] M. Chen et al., "Evaluating Large Language Models Trained on Code," *arXiv preprint arXiv:2107.03374*, 2021.

[2] A. Nijkamp et al., "CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis," *arXiv preprint arXiv:2203.13474*, 2022.

[3] S. Ahmad et al., "Local Language Models: Bridging the Gap Between Privacy and Performance," *Journal of AI Research and Applications*, vol. 12, no. 4, pp. 45–56, 2023.

[4] Ollama Documentation, "Run Large Language Models Locally," Available: https://ollama.com/.

[5] T. Brown et al., "Language Models are Few-Shot Learners," *Advances in Neural Information Processing Systems (Neur IPS)*, 2020.

[6] R. Allamanis et al., "Learning to Generate Code from Natural Language with Neural Attention," *ICML*, 2016.

[7] Z. Feng et al., "Code BERT: A Pre-Trained Model for Programming and Natural Languages," *EMNLP*, 2020.

[8] C. Li et al., "Star Coder: May the Source Be with You!" *Hugging Face Research Paper*, 2023.

[9] K. Cho et al., "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation," *EMNLP*, 2014.

[10] A. Vaswani et al., "Attention is All You Need," *NeurIPS*, 2017.

[11] M. Austin et al., "Program Synthesis with Large Language Models," *arXiv preprint arXiv:2211.10435*, 2022.

[12] Y. Wang et al., "CodeT5: Identifier-aware Unified Pre-trained Encoder–Decoder Models for Code Understanding and Generation," *EMNLP*, 2021.

[13] Open AI, "Introducing Codex: The Model that Powers GitHub Copilot," *Open AI Blog*, 2021.

[14] Amazon Web Services, "Amazon Code Whisperer Developer Guide," *AWS Documentation*, 2023.

[15] J. Liu et al., "Prompt Engineering for Program Synthesis with LLMs," *arXiv preprint arXiv:2304.11045*, 2023.

[16] D. Jain et al., "Compiler-Aware Prompt Optimization for Code Generation," *Proceedings of the International Conference on Software Analysis and Mining*, 2023.

[17] C. Zhang et al., "Secure Execution of AI-Generated Code in Local Environments," *IEEE Transactions on Software Engineering*, vol. 50, no. 3, pp. 215–228, 2024.

[18] A. Tirumala et al., "Efficient Local Inference with Quantized Transformer Models," *arXiv preprint arXiv:2310.04612*, 2023.

[19] D. Li et al., "Instruction Tuning and Its Impact on Code Generation Consistency," *Transactions on Machine Learning Research (TMLR)*, 2023.

[20] B. Roziere et al., "Human Eval and Beyond: Benchmarking Code Generation," *arXiv preprint arXiv:2107.03374*, 2022.

[21] GitHub, "GitHub Copilot Technical Overview," *GitHub Documentation*, 2023.

[22] Tab Nine, "Tab Nine Local AI Coding Assistant," *Product Whitepaper*, 2023.

[23] H. Ouyang et al., "Training Language Models to Follow Instructions with Human Feedback," *Neur IPS*, 2022.
[24] S. Miano and P. J. Marrow, "Sandboxing and Isolation Techniques for Secure Code Execution," *ACM Computing Surveys*, vol. 55, no. 7, pp. 1–32, 2023.
[25] R. Soliman et al., "Leveraging Pretrained Language Models for Code Generation," *IEEE Transactions on Emerging Topics in Computing*, vol. 11, no. 2, pp. 168–182, 2024.
[26] A. Nguyen et al., "Self-Refinement Loops for Code Generation Models," *ICLR*, 2023.
[27] Linux Foundation, "Seccomp and c groups for Application Isolation," *Linux Kernel Security Documentation*, 2022.
[28] S. Li et al., "Edge–Cloud Collaborative AI Systems for Low-Latency Model Inference," *IEEE Internet of Things Journal*, vol. 10, no. 4, pp. 3851–3864, 2024.