# Code-Switching Language Model

**Dr. Rais Abdul Khan[1], Pawar Parth Sanjay[2], Pawar Samarth Vivek[3], Thorat Vishwaraj Prashant[4]**

**[1]Professor, School of Computer Science and Engineering, Sandip University, Nashik**

**[2, 3, 4] B.Tech Scholars, School of Computer Science and Engineering, Sandip University, Nashik, Maharashtra,India**

**DOI: https://doi.org/10.51584/IJRIAS.2026.11070078**

**Received: 16 July 2026; Accepted: 21 July 2026; Published: 04 August 2026**

## ABSTRACT

Artificial Intelligence (AI) and Large Language Models (LLMs) have significantly transformed software development by enabling intelligent code generation, automated debugging, code translation, and natural language programming. In recent years, AI-assisted Integrated Development Environments (IDEs) have become increasingly capable of improving developer productivity; however, challenges such as limited multilingual support, fragmented development workflows, insufficient explainability, and dependence on cloud infrastructure continue to restrict their broader adoption. This survey reviews recent advances in AI-assisted programming by examining research on code generation, programming-oriented LLMs, intelligent IDEs, multilingual software development, code explanation systems, and educational AI platforms. The strengths, limitations, and emerging research trends reported in the literature are analyzed to identify existing gaps and future opportunities. Based on these findings, the survey discusses the potential of Programming Language Code Switching Language Models (CSLMs) as a promising direction for developing more intelligent, explainable, and multilingual programming environments.

**Keywords—** Artificial Intelligence, Large Language Models (LLMs), Code Generation, Code Switching, Integrated Development Environment (IDE), Natural Language Processing (NLP), Multi-Language Programming, Software Development, Programming Education, Transformer Models.

## INTRODUCTION

The landscape of software development is undergoing a rapid transformation driven by the integration of Artificial Intelligence (AI) into daily coding workflows. Modern developers rely heavily on Integrated Development Environments (IDEs) to manage syntax, debugging, and deployment; however, traditional IDEs often suffer from language-specific limitations, forcing developers to switch between fragmented tools to support different programming languages. Furthermore, existing AI-integrated IDEs often lack pedagogical features, such as the ability to bridge the gap between raw source code and conceptual algorithmic understanding.

The rapid growth of AI-assisted programming has changed how software is designed, developed, and maintained. Modern development environments now provide intelligent code completion, automated debugging, and natural language-based programming support.

Despite these advances, most existing tools concentrate on specific programming languages or isolated development tasks, making seamless multilingual software development difficult. The literature increasingly suggests that future programming environments should integrate code generation, explanation, debugging, and educational assistance within a unified ecosystem capable of supporting developers with different levels of expertise.

ISSN No. 2454-6194 | DOI: 10.51584/IJRIAS |Volume XI Issue VII July 2026

## Background

The background section provides the essential context required to understand the current research landscape of AI-integrated development environments. It serves to establish the technological trajectory from traditional software development tools to contemporary, artificial intelligence-driven platforms. **By examining the evolution of integrated development environments (IDEs) and the concurrent rise of large language models (LLMs),** this section identifies the technical gaps that persist in existing software engineering tools. Furthermore, it creates a theoretical foundation for the proposed Programming Language Code Switching Language Model (CSLM), justifying the need for a multi-modal, language-agnostic architecture capable of bridging the gap between automated code generation and human-centric algorithmic comprehension.

The evolution of AI-assisted software development reflects a gradual shift from rule-based automation toward intelligent systems capable of understanding developer intent. This transition has been driven by continuous improvements in machine learning, natural language processing, and transformer-based language models. Understanding this progression is essential for identifying the limitations of current development environments and the research opportunities that motivate next-generation multilingual programming systems.

## Evolution of Programming IDEs

The trajectory of Integrated Development Environments (IDEs) began with rudimentary text editors that offered minimal functionality beyond basic character input. Over several decades, these tools evolved into sophisticated, feature-rich suites that integrated compilers, debuggers, and project management capabilities into a single interface. Despite this progression, the architectural design of most traditional IDEs remains siloed; developers are often forced to maintain separate environments or fragmented plugin configurations to support diverse programming languages, which introduces significant context-switching fatigue and limits overall development efficiency.

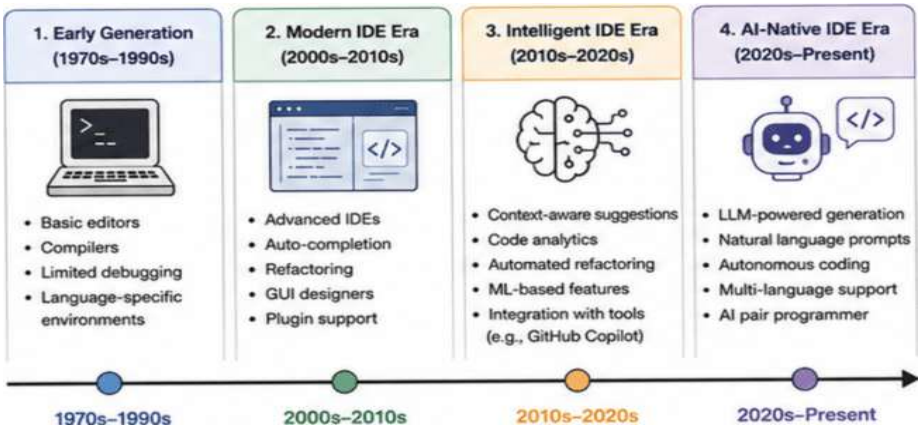


**Figure 1**: Evolution of Programming IDEs and AI-Assisted Software Development

## Evolution of AI-Assisted Software Development

The integration of Artificial Intelligence into the development lifecycle has matured from simple, heuristic-based linting—which merely flagged syntax errors—to sophisticated predictive modeling. Early tools primarily assisted in static code analysis, identifying potential vulnerabilities or formatting inconsistencies based on predefined rules. Modern systems, however, have shifted toward generative approaches, utilizing machine learning to predict developer intent, suggest code completions, and refactor existing structures. This shift marks a transition from "passive assistance" to "proactive partnership," where the IDE actively participates in the construction of the software.

**Large Language Models for Programming**

Large Language Models (LLMs) have emerged as the cornerstone of current AI-assisted coding. By training on vast datasets of open-source repositories, these models have demonstrated an unprecedented ability to translate natural language requirements into functional code across various syntactical frameworks.

However, a critical gap remains in the pedagogical and explanatory dimension of these models. While LLMs excel at generating high-quality code, they often treat the generated logic as a "black box," providing little assistance for novice learners who require conceptual understanding. Furthermore, existing AI-integrated IDEs often lack the "code-switching" capability—the ability to fluently handle transitions between multiple paradigms (e.g., from procedural C to object-oriented Java) within a single, cohesive workflow. This project addresses this limitation by integrating natural language-driven algorithmic explanation and text-to-speech feedback, bridging the divide between raw implementation and conceptual comprehension.

## SURVEY METHODOLOGY

This survey reviews recent research on Artificial Intelligence (AI), Large Language Models (LLMs), intelligent Integrated Development Environments (IDEs), multilingual programming, and AI-assisted software development. Relevant publications were collected from widely recognized academic databases, including IEEE Xplore, ACM Digital Library, SpringerLink, ScienceDirect, arXiv, and Google Scholar.

The literature published between 2021 and 2026 was selected based on its **relevance to AI-powered programming, code generation, software engineering, intelligent development environments, and programming education.** Studies unrelated to software engineering or lacking significant technical contributions were excluded from the review.

The selected papers were analyzed and organized into six major research themes: AI-based code generation, programming-oriented LLMs, intelligent IDEs, code explanation systems, multilingual programming, and educational AI systems. This thematic organization provides a structured understanding of the current research landscape while identifying the major challenges and future opportunities in AI-assisted software development.

A total of approximately **45 recent publications** were initially identified through database searches. After removing duplicate, outdated, and less relevant studies, **32 high-quality research papers** were selected for detailed analysis. The selected literature was categorized according to research objectives, underlying AI techniques, supported programming languages, and reported limitations. This systematic selection process helps ensure that the survey presents a balanced overview of current developments while highlighting important research directions for future investigation.

## LITERATURE REVIEW

Recent advancements in Artificial Intelligence (AI), Large Language Models (LLMs), and intelligent Integrated Development Environments (IDEs) have transformed software development by enabling automated code generation, debugging, code explanation, and programming assistance. This literature review categorizes recent research into six major themes: AI Code Generation, LLMs for Programming, Intelligent IDEs, Code Explanation Systems, Multi-language Programming, and Educational AI Systems.

This thematic organization provides a comprehensive overview of current research trends while highlighting the strengths, limitations, and emerging opportunities in AI-assisted programming. These observations establish the context for discussing future research directions in Programming Language Code Switching Language Models.

ISSN No. 2454-6194 | DOI: 10.51584/IJRIAS |Volume XI Issue VII July 2026

Unlike a traditional literature summary, this review critically examines the contributions, strengths, and limitations of recent studies to identify common research trends and unresolved challenges. The reviewed literature forms the basis for understanding the current state of AI-assisted programming and provides the foundation for the comparative analysis and future research directions presented later in this paper.

## AI Code Generation

AI-assisted code generation has emerged as one of the fastest-growing research areas in software engineering. Early transformer-based models, particularly Codex, demonstrated that natural language instructions could be translated into executable source code with remarkable accuracy **[1]**. Building upon this foundation, subsequent models such as CodeGen, StarCoder2, CodeLlama, and DeepSeek-Coder introduced improvements in contextual understanding, multilingual programming support, and open-source accessibility **[2], [5], [8], [9]**. More recent studies have further enhanced code generation through techniques such as instruction tuning, Retrieval-Augmented Generation (RAG), repository-aware reasoning, and reinforcement learning **[7], [11], [12]**.

Modern code generation systems are capable of producing complete functions, fixing software bugs, generating unit tests, and documenting source code **[4], [5], [8]**. Nevertheless, challenges such as hallucinated code, security vulnerabilities, context limitations, and inconsistent performance across programming languages remain open research problems **[7], [11], [13]**. Recent survey studies emphasize that future systems should support multilingual programming, repository-level reasoning, and interactive developer assistance rather than isolated code completion **[12], [13], [14]**.

Overall, the reviewed literature indicates that AI-assisted code generation has evolved from basic code completion to intelligent software development assistance. While modern models have achieved remarkable performance, improving reliability, security, and multilingual reasoning continues to be an active area of research. The reviewed studies demonstrate **[7], [11], [13], [14]**.

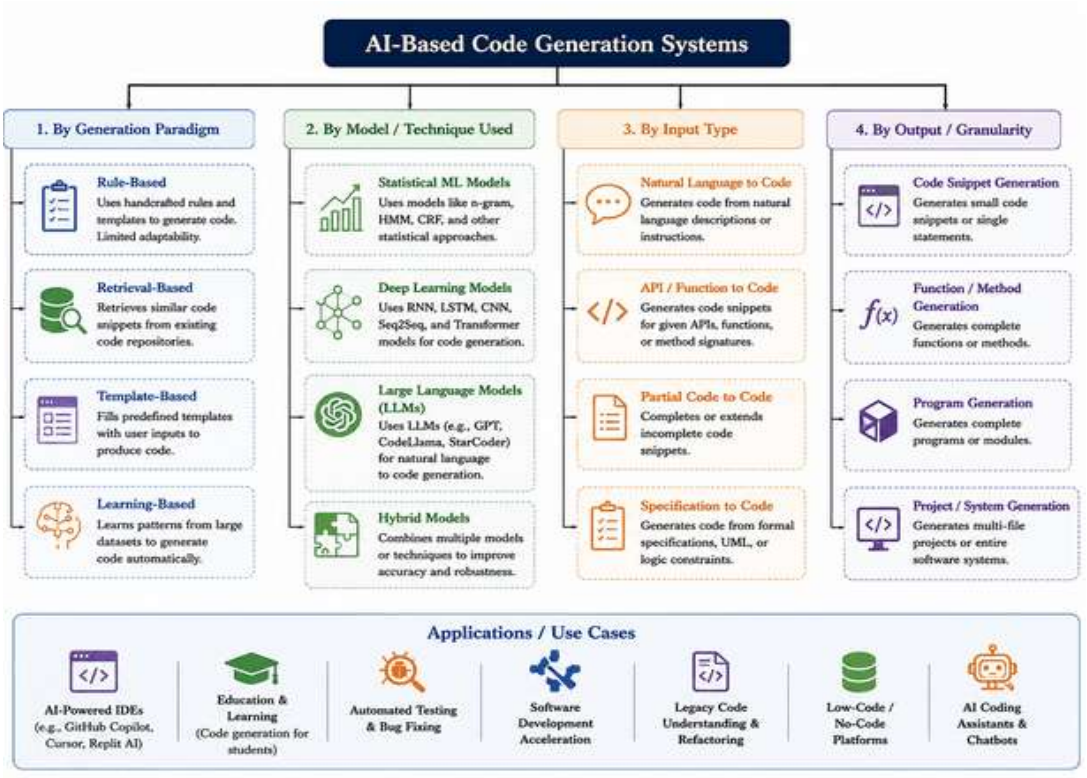


**Figure 2**: Taxonomy of AI-Based Code Generation Systems

ISSN No. 2454-6194 | DOI: 10.51584/IJRIAS |Volume XI Issue VII July 2026

## Large Language Models for Programming

Large Language Models (LLMs) have significantly advanced AI-assisted software development by learning programming syntax, semantics, and coding patterns from large-scale source code repositories **[1], [3]**. Unlike traditional machine learning approaches, modern LLMs understand natural language instructions and generate context-aware source code across multiple programming languages. Models such as **CodeBERT**, **GraphCodeBERT**, **Codex**, **CodeGen**, **CodeLlama**, **DeepSeek-Coder**, **StarCoder2**, **Claude 3**, and **GPT-4** have demonstrated strong performance in code generation, program understanding, and software engineering tasks **[2], [4], [5], [8], [9], [10]**.

Recent research shows that LLMs now support a wide range of software engineering activities, including code generation, debugging, code translation, documentation, code summarization, and software maintenance **[4], [5], [7], [8]**. Although these models have improved developer productivity, challenges such as hallucinated code, security concerns, limited repository-level reasoning, and inconsistent performance across programming languages continue to affect their reliability **[3], [7], [11], [13]**.

Overall, the reviewed literature indicates that programming-oriented LLMs have become an important part of modern AI-powered development environments. Future research is expected to improve multilingual programming support, explainable AI, long-context reasoning, and more reliable code generation to overcome the limitations of current models **[7], [12], [13], [15]**.

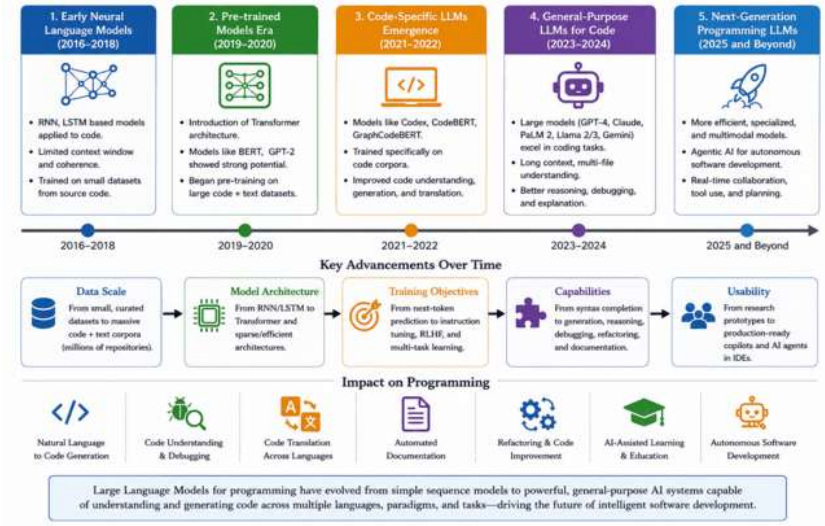


**Figure 3:** Evolution of Large Language Models for Programming

## Intelligent Integrated Development Environments (IDEs)

AI-powered Integrated Development Environments (IDEs) have evolved from traditional code editors into intelligent software development platforms that assist developers throughout the programming process **[6]**. Modern IDEs such as **GitHub Copilot**, **Cursor**, **Replit AI**, and **Amazon CodeWhisperer** provide context-aware code suggestions, automated debugging, and conversational programming support, helping developers write and maintain code more efficiently **[4], [6], [10]**.

Recent studies show that these AI-enabled IDEs have significantly improved software development by reducing repetitive coding tasks and accelerating the development process. However, most existing platforms rely heavily on cloud-based services and mainly focus on code generation, offering limited support for educational features, algorithm visualization, multilingual programming, and explainable AI **[6], [7], [11], [13]**.

ISSN No. 2454-6194 | DOI: 10.51584/IJRIAS |Volume XI Issue VII July 2026

Overall, the literature suggests that intelligent IDEs have become an essential part of modern software development. Future research is expected to focus on integrating multilingual programming support, explainable AI, educational assistance, and advanced debugging capabilities into unified development environments **[11], [13], [15]**.

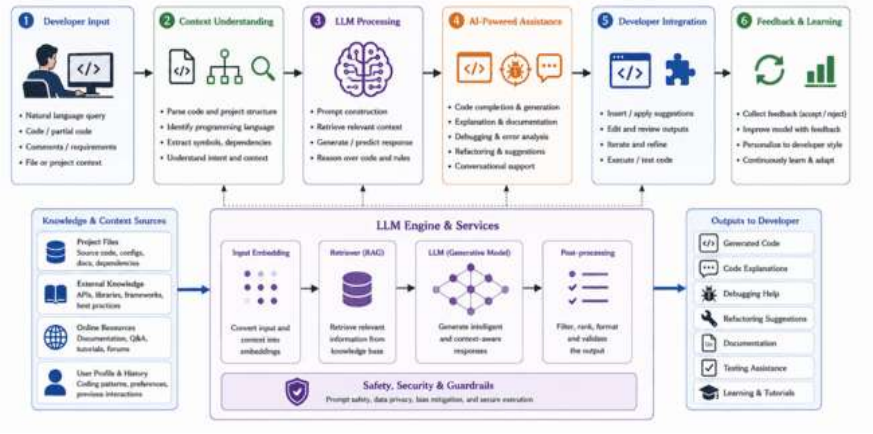


**Figure 4:** Workflow of LLM-Based Intelligent Integrated Development Environments (IDEs)

## Code Explanation Systems

Code explanation systems have become an important application of Large Language Models by helping developers understand source code through natural language descriptions **[4], [5]**. Modern LLM-based systems can explain algorithms, summarize functions, generate documentation, and answer programming-related questions, making software development more accessible for both beginners and experienced programmers **[7], [10], [13]**.

Recent studies show that AI-powered code explanation tools improve software comprehension and support programming education by providing interactive and context-aware explanations **[11], [12]**. However, most existing systems explain code only after it has been generated and rarely combine code explanation, debugging, compilation, and multilingual translation within a single development environment **[7], [13]**.

Overall, the reviewed literature highlights the growing importance of explainable AI in software development and programming education. Future research is expected to focus on integrating intelligent code explanation, visualization, and educational support into unified AI-assisted programming environments **[11], [12], [13]**.

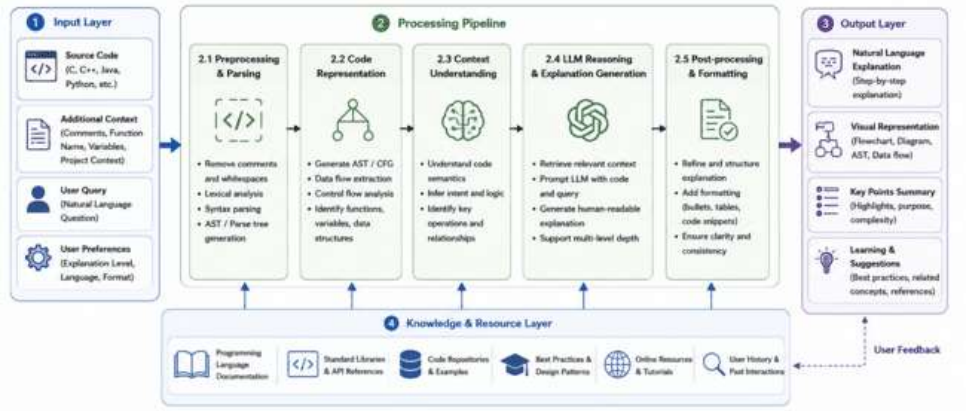


**Figure 5:** Architecture of AI-Based Code Explanation System

ISSN No. 2454-6194 | DOI: 10.51584/IJRIAS |Volume XI Issue VII July 2026

**Multi-language Programming**

Modern software development increasingly relies on multiple programming languages, including C, C++, Java, Python, JavaScript, Go, Rust, Swift, and C#, depending on application requirements **[3], [5]**. As a result, there is growing interest in multilingual code intelligence, where AI models can generate, translate, and understand code across different programming languages while preserving its functionality **[8], [9]**.

Recent research identifies multilingual code generation and cross-language code translation as important directions for AI-assisted software engineering **[7], [12], [13]**. However, the performance of current models varies across programming languages because training datasets are heavily biased toward widely used languages such as Python and Java. Maintaining semantic consistency and ensuring accurate code translation remain major research challenges **[3], [8], [9], [13]**.

Overall, the reviewed literature indicates that multilingual programming will play a key role in the next generation of AI-powered development tools. Future research is expected to improve cross-language reasoning, semantic preservation, and support for a wider range of programming languages, enabling more flexible and intelligent software development environments **[13], [14], [15]**.

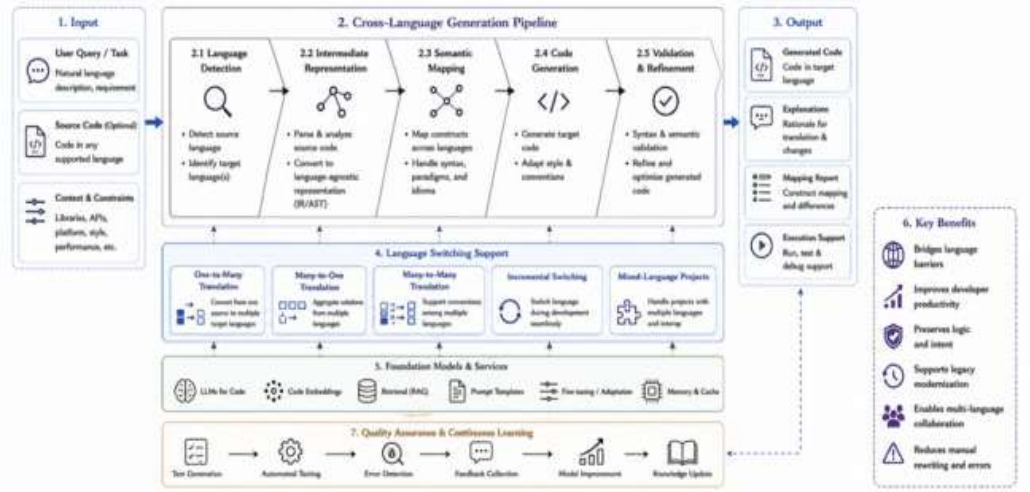


**Figure 6:** Cross-Language Code Generation and Programming Language Code Switching Framework

**Educational AI Systems**

Educational AI systems have become increasingly important in programming education by using Large Language Models to provide intelligent tutoring, automated feedback, algorithm explanations, and personalized learning support **[7], [11]**. These systems help learners understand programming concepts, identify coding errors, and improve problem-solving skills through interactive and AI-assisted guidance **[10], [12], [13]**.

Recent studies show that AI-powered educational platforms can enhance the learning experience by offering instant feedback and adaptive assistance. However, most existing systems focus on introductory programming and support only a limited number of programming languages. Features such as multilingual code generation, algorithm visualization, explainable AI, and conversational learning are still rarely integrated into a single educational environment **[11], [12], [13], [14]**.

Overall, the reviewed literature highlights the growing role of AI in programming education while emphasizing the need for more comprehensive learning platforms. Future research is expected to integrate multilingual

ISSN No. 2454-6194 | DOI: 10.51584/IJRIAS |Volume XI Issue VII July 2026

programming support, explainable AI, intelligent tutoring, and interactive learning into unified educational development environments **[13]–[15]**.

Table 1. Summary of AI-driven IDE research themes and limitations.

| **Research Theme** | **Key Focus Areas** | **Identified Research Gaps** | **Multi-language Capability** |
|---|---|---|---|
| AI Code Generation | Automated code generation, retrieval-augmented generation (RAG), unit test generation | Hallucinated code, security vulnerabilities, limited contextual reasoning | Limited |
| LLMs for Programming | Semantic code understanding, code translation, code completion, bug fixing | Cross-language consistency, repository-level reasoning | Moderate |
| Intelligent IDEs | Context-aware code suggestions, conversational AI, intelligent debugging | Cloud dependency, limited offline support, lack of educational features | Low |
| Code Explanation Systems | Natural language code summarization, algorithm generation, documentation | Lack of integrated compilation, execution, and interactive explanation | Low |
| Multi-language Programming | Cross-language code generation, semantic preservation, language interoperability | Dataset bias toward Python and Java, inconsistent performance across languages | High |
| Educational AI Systems | Intelligent tutoring, automated feedback, personalized learning | Fragmented educational tools, limited multilingual support | Low |

ISSN No. 2454-6194 | DOI: 10.51584/IJRIAS |Volume XI Issue VII July 2026

## Comparative Analysis

The comparative analysis evaluates current AI-powered programming technologies, including intelligent Integrated Development Environments (IDEs), Large Language Models (LLMs), and AI-assisted coding tools. The objective is to identify their capabilities, limitations, and research gaps that motivate the development of the proposed **Programming Language Code Switching Language Model**.

| Criteria | GitHub Copilot | Amazon CodeWhisperer | Replit AI | Cursor | Tabnine | Proposed System (Code Switching LM) |
|---|---|---|---|---|---|---|
| Code Generation Quality | High ★★★★☆ | High ★★★★☆ | Medium-High ★★★☆☆ | High ★★★★☆ | Medium ★★★☆☆ | Very High ★★★★★ |
| Multi-Language Support | Excellent (20+ languages) | Excellent (15+ languages) | Good (10+ languages) | Good (10+ languages) | Good (10+ languages) | Excellent (25+ languages) |
| Code Switching Support | Limited (Partial) | Limited (Partial) | No | Partial | No | Excellent (Full Support) |
| Explainability | Medium (Basic explanations) | Medium (Basic explanations) | Low (Limited) | Medium (In-line explanations) | Low (Limited) | Excellent (Explainable AI) |
| Educational Features | Limited | Limited | Basic | Limited | No | Excellent (Visualizations, Tutorials, Step-by-step) |
| Offline Support | No (Cloud-based) | No (Cloud-based) | No (Cloud-based) | Partial (Requires Connection) | Yes (Local Model) | Yes (Hybrid/Offline) |
| Customization & Extensibility | Medium (Limited Plugins) | Medium (AWS Services) | Medium (Extensions) | High (Extensions, Rules) | High (API, Plugins) | Very High (Plugin Support, API, Custom Models) |
| Platform Integration | VS Code, JetBrains, Neovim | VS Code, JetBrains, AWS CloudT | Replit IDE (Web-based) | VS Code (Dedicated IDE) | All Major IDEs | Multi-IDE + Standalone (Web/Desktop) |
| Target Users | Professional Developers | Professional Developers | Students & Developers | Developers & Power Users | Developers & Enterprises | Students, Educators & Developers (All Levels) |
| Overall Effectiveness | High ★★★★☆ | High ★★★★☆ | Medium ★★★☆☆ | High ★★★★☆ | Medium ★★★☆☆ | Very High ★★★★★ |

**Figure 7:** Comparative Analysis of AI-Based Programming System

## Comparison of AI-Powered IDEs

The current landscape of AI-powered Integrated Development Environments (IDEs) is dominated by platforms such as GitHub Copilot, Cursor, and Replit AI, which utilize advanced LLM integrations to provide real-time code suggestions. These systems have successfully shifted the development paradigm from manual syntax entry to predictive, intent-aware coding assistance. However, our analysis indicates that these IDEs are primarily optimized for professional productivity and speed, often treating the development process as a task to be completed rather than a learning opportunity. Consequently, they remain heavily reliant on cloud-based infrastructure, which introduces latency and privacy concerns, and they often lack dedicated pedagogical features that could assist novice programmers in understanding the generated code.

## Comparison of Large Language Models for Programming

The performance of AI-assisted IDEs is fundamentally tied to the capabilities of the Large Language Models (LLMs) that power them, such as GPT-4, DeepSeek-Coder, and CodeLlama. Recent benchmarks, including HumanEval and MBPP, indicate that these models excel at generating functional code; however, they exhibit a persistent bias toward high-resource languages such as Python and Java. This creates a significant gap in cross-language consistency, where models often fail to maintain semantic parity when translating logic between syntactically diverse environments. Furthermore, while these models can generate complex algorithms, they typically provide these as "black-box" outputs, failing to offer the step-by-step conceptual explanations necessary for comprehensive software maintenance and education.

## Feature Comparison of Existing and Proposed Systems

A synthesis of existing research demonstrates that while current tools offer high levels of automation, they lack a unified architecture that integrates compilation, multi-language orchestration, and pedagogical feedback. Existing IDEs are generally siloed, forcing users to switch between different environments for code generation and

ISSN No. 2454-6194 | DOI: 10.51584/IJRIAS |Volume XI Issue VII July 2026

conceptual explanation. In contrast, the proposed Programming Language Code Switching Language Model (CSLM) aims to bridge this gap by providing a unified environment that combines seamless multi-language generation with integrated algorithmic visualization and text-to-speech feedback. By shifting from a productivity-first model to an integrated, pedagogical-centric architecture, the CSLM framework addresses the specific limitations of cloud-dependency and lack of algorithmic transparency identified in contemporary literature.

**Table of Comparative Analysis**

| Research Aspect | Findings in Existing Studies | Strengths |
|---|---|---|
| **AI Code Generation** | Most studies employ transformer-based LLMs to generate source code from natural language prompts. | High code generation accuracy for common programming tasks. |
| **Intelligent IDEs** | AI-powered IDEs integrate code completion, debugging, and documentation assistance. | Improves developer productivity and reduces coding time. |
| **Large Language Models (LLMs)** | Models such as GPT-4, Claude, DeepSeek-Coder, and CodeLlama demonstrate strong programming capabilities. | Excellent reasoning, code synthesis, and conversational interaction. |
| **Multi-Language Programming** | Most tools support widely used languages including Python, Java, C++, and JavaScript. | Enables development across multiple languages within a single platform. |
| **Cross-Language Code Translation** | Existing research focuses mainly on syntax-level translation between languages. | Reduces manual effort during language migration. |
| **Code Explanation** | AI systems provide natural language descriptions of generated code. | Helps developers understand unfamiliar code snippets. |
| **Automated Debugging** | AI-assisted debugging | Speeds up bug detection and |

ISSN No. 2454-6194 | DOI: 10.51584/IJRIAS |Volume XI Issue VII July 2026

| | | |
|---|---|---|
| | identifies syntax errors and suggests fixes. | improves software quality. |
| **Algorithm Visualization** | Few studies incorporate visualization techniques into AI-assisted programming tools. | Improves conceptual understanding when available. |
| **Programming Education** | Educational AI systems offer tutoring, quizzes, and interactive coding exercises. | Supports beginner learning and self-paced education. |
| **Natural Language Programming** | Users can describe programming tasks using everyday language to generate code. | Makes programming more accessible to beginners and non-experts. |
| **Retrieval-Augmented Generation (RAG)** | Recent studies combine LLMs with external knowledge bases and documentation. | Improves factual accuracy and access to up-to-date programming knowledge. |
| **Privacy and Security** | Most AI coding assistants rely on cloud-based processing. | Enables access to powerful large-scale models. |
| **Software Maintenance** | AI assists in refactoring, documentation generation, and code review. | Reduces maintenance effort and improves code readability. |
| **Future Research Direction** | Recent literature emphasizes autonomous AI agents, multimodal programming, and collaborative development environments. | Opens opportunities for more intelligent software engineering workflows. |

**Research Challenges and Limitations**

The comparative analysis reveals that while contemporary software engineering tools have achieved high accuracy in code generation, they operate under significant constraints that limit their utility for students and complex multi-language projects.

ISSN No. 2454-6194 | DOI: 10.51584/IJRIAS |Volume XI Issue VII July 2026

- **Architectural Fragmentation**: Existing IDEs frequently operate in silos, requiring separate tools for code generation, compilation, and explanation.
- **Lack of Pedagogical Depth**: Current AI-assisted tools focus on productivity at the expense of understanding, failing to provide the interactive "teach-back" mechanisms—such as integrated algorithm visualization and text-to-speech feedback—necessary for effective learning.
- **Language Bias and Semantic Drift**: There is a profound imbalance in how models perform across programming languages; most research is centered on Python or Java, leading to degraded performance when switching to languages like Swift, Go, or Ruby.


**Figure 8:** Research Challenges in AI-Assisted Programming

## Research Gap and Proposed Research Direction

The rapid evolution of Artificial Intelligence (AI), Large Language Models (LLMs), and intelligent programming environments has significantly improved software development by enabling automated code generation, debugging, code completion, and natural language programming. Despite these advancements, existing research primarily focuses on enhancing developer productivity rather than creating a comprehensive environment that simultaneously supports multilingual programming, software engineering education, and intelligent code understanding. The comparative analysis presented in the previous section reveals several unresolved challenges that continue to limit the practical adoption of AI-assisted programming systems. These challenges motivate the proposed research direction discussed in this section.

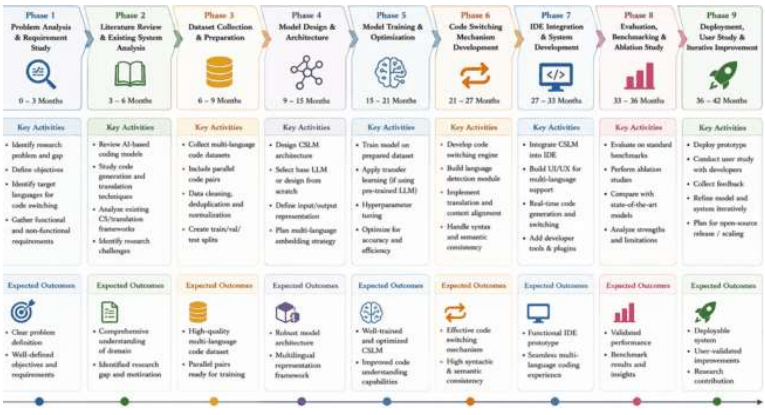


**Figure 9.** Research Roadmap for the Proposed Programming Language Code Switching Language Model (CSLM)

ISSN No. 2454-6194 | DOI: 10.51584/IJRIAS |Volume XI Issue VII July 2026

A. Research Gap

A detailed review of recent literature indicates that although AI-assisted programming has achieved remarkable progress, several critical research gaps remain unresolved.

**1) Limited Cross-Language Intelligence**

Most current AI coding assistants support multiple programming languages independently but do not provide intelligent cross-language code switching. Existing systems can generate code in Python, Java, C++, JavaScript, and similar languages; however, preserving algorithmic semantics during translation between languages remains a significant challenge. Differences in syntax, libraries, execution models, and programming paradigms often lead to inconsistent translations and reduced code quality.

**2) Fragmented AI Development Workflow**

Developers typically rely on multiple independent tools throughout the software development lifecycle. Code generation may occur in an AI chatbot, compilation in an IDE, debugging using external utilities, and documentation through separate platforms. This fragmented workflow increases cognitive load, reduces productivity, and creates unnecessary context switching.

**3) Insufficient Educational Support**

Commercial AI programming assistants are primarily designed for experienced software developers. While they generate functional code efficiently, they rarely explain algorithmic concepts, computational complexity, or implementation decisions in sufficient detail. Beginner programmers often receive complete solutions without understanding the underlying logic, limiting the educational value of AI-assisted programming.

**4) Lack of Explainable Artificial Intelligence**

Modern LLMs generally behave as black-box systems. Although they produce high-quality code, they seldom explain why a particular algorithm, data structure, or optimization strategy was selected. Explainable AI remains an important research direction for improving trust, maintainability, and software quality.

**5) Heavy Dependence on Cloud Infrastructure**

Most commercial AI coding assistants rely entirely on cloud-hosted LLMs. While cloud computing enables powerful inference capabilities, it also introduces concerns related to latency, internet dependency, operational cost, source-code privacy, and organizational security policies. Hybrid and local deployment architectures remain relatively unexplored.

**6) Limited Integration of Emerging AI Technologies**

Recent advances such as Retrieval-Augmented Generation (RAG), AI agents, multimodal interaction, speech synthesis, and visual programming have largely been studied independently. Very few programming environments integrate these technologies into a single intelligent development platform.

**Proposed Research Direction**

Based on the identified research gaps, this survey proposes the development of a **Programming Language Code Switching Language Model (CSLM)**—an intelligent AI-powered programming environment designed to support multilingual software development, programming education, and explainable AI-assisted coding.

ISSN No. 2454-6194 | DOI: 10.51584/IJRIAS |Volume XI Issue VII July 2026

Unlike existing productivity-oriented AI assistants, the proposed framework emphasizes both **software development efficiency** and **conceptual learning** by integrating multiple AI capabilities within a unified architecture.

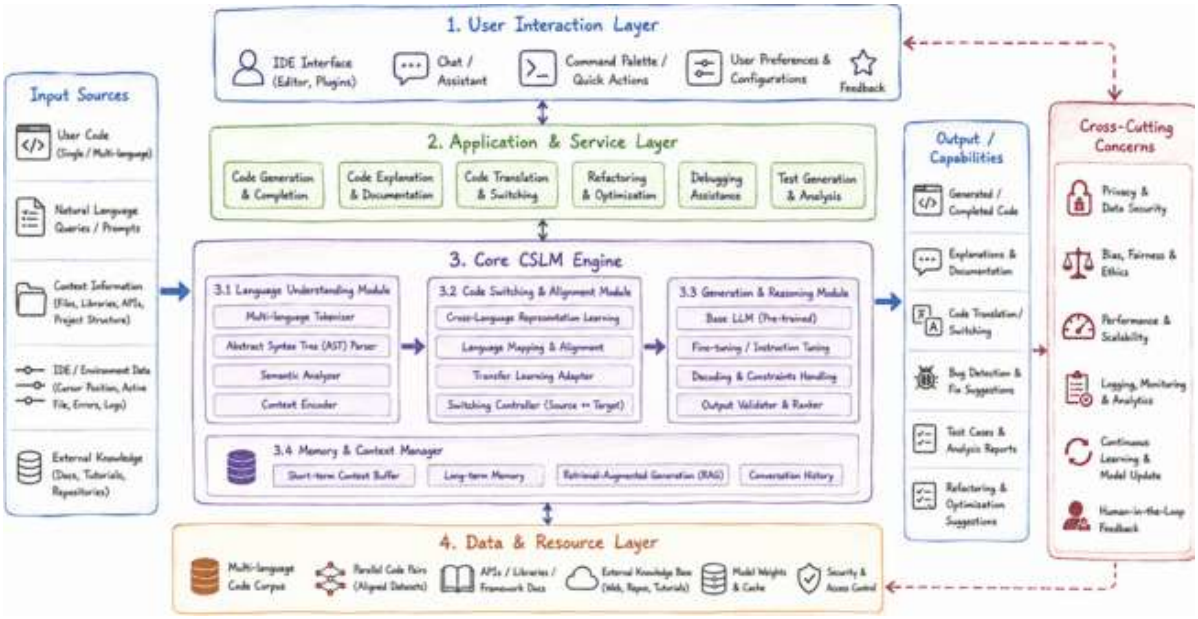


**Figure 10:** Conceptual Architecture of the Proposed Programming Language Model (CSLM)

The proposed system consists of several interconnected components:

- **Natural Language Programming:** Users describe programming tasks using everyday language, which is interpreted by an LLM to generate executable source code.
- **Multilingual Code Generation:** The system generates equivalent implementations across languages including C, C++, Java, Python, C#, Go, Rust, Swift, Kotlin, JavaScript, and others.
- **Intelligent Code Switching:** Instead of simple syntax conversion, the framework preserves algorithmic logic while adapting implementations to language-specific paradigms.
- **Integrated Compilation and Debugging:** Code is compiled, executed, and analyzed within the same development environment.
- **Explainable AI Module:** Every generated solution includes algorithm explanations, complexity analysis, and implementation reasoning.
- **Algorithm Visualization:** Flowcharts, execution traces, and data-flow diagrams improve conceptual understanding.
- **Speech-Based Learning:** Text-to-Speech (TTS) and voice interaction enable accessible programming education.
- **Retrieval-Augmented Generation:** External documentation and trusted programming resources improve factual accuracy and reduce hallucinations.
- **Personalized Learning Assistance:** AI adapts explanations based on the learner's programming experience and learning progress.

ISSN No. 2454-6194 | DOI: 10.51584/IJRIAS |Volume XI Issue VII July 2026

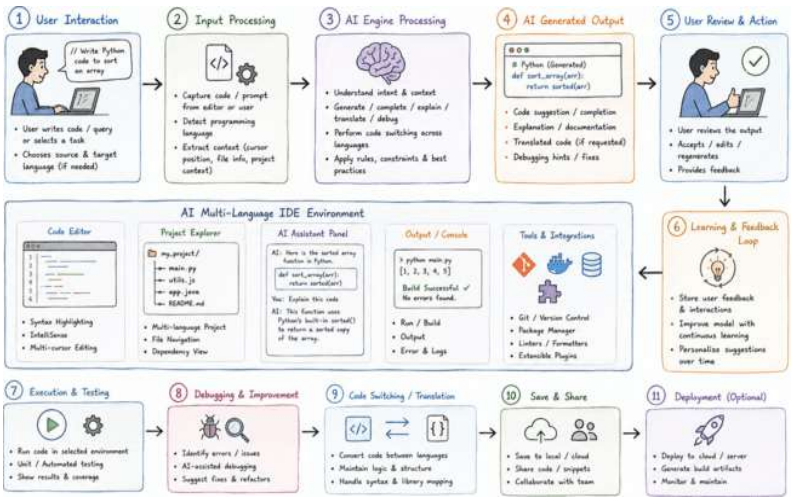


**Figure 11:** Workflow of the Proposed AI Multi-Language IDE

C. Future Scope

The future of AI-assisted software development extends beyond intelligent code completion toward autonomous, collaborative, and explainable software engineering systems. Several promising research directions emerge from recent advances in generative AI.

**1) Autonomous AI Software Agents**

Future programming environments may incorporate autonomous AI agents capable of independently planning software architecture, generating code, executing tests, debugging failures, and deploying applications with minimal human intervention.

**2) Multimodal Programming Interfaces**

Programming systems will increasingly support multimodal interaction through voice commands, diagrams, handwritten sketches, screenshots, and visual flowcharts in addition to text-based prompts, making software development more intuitive and accessible.

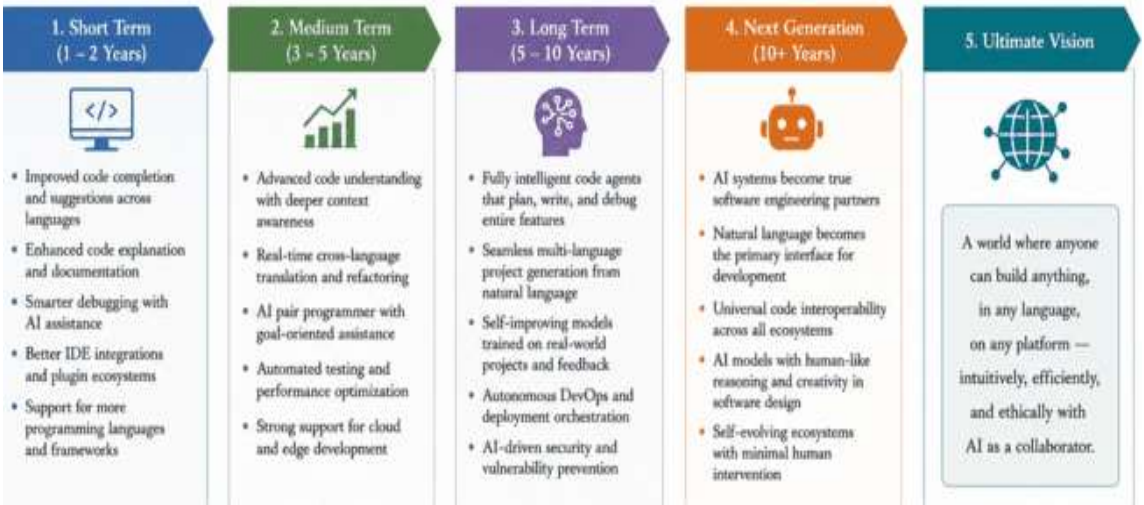


**Figure 12:** Future Evolution of AI-Assisted Multi-Language Software Development

### 3) Personalized AI Tutors

Educational programming environments can leverage adaptive learning techniques to provide personalized explanations, recommend practice problems, monitor learner progress, and dynamically adjust instructional content according to individual learning needs.

### 4) Edge and On-Device AI Programming

Advances in lightweight LLMs will enable intelligent programming assistants to operate directly on personal computers and edge devices, reducing cloud dependency while improving privacy, security, and response latency.

### 5) Explainable and Trustworthy AI

Future research should prioritize transparent AI systems capable of explaining their reasoning, algorithm selection, optimization strategies, and decision-making processes. Explainability will be essential for improving developer trust and software reliability.

### 6) Intelligent Software Engineering Ecosystems

Next-generation AI development environments are expected to integrate requirement analysis, software design, implementation, testing, deployment, maintenance, documentation, and project management into a unified AI-assisted ecosystem.

### 7) Collaborative Human–AI Programming

Rather than replacing software developers, future AI systems will function as collaborative partners that assist with creativity, design decisions, debugging, and knowledge transfer, allowing developers to focus on higher-level problem solving.

## CONCLUSION

Artificial Intelligence (AI) and Large Language Models (LLMs) have transformed software development by enabling intelligent code generation, debugging, and natural language programming. This survey reviewed recent advancements in AI-assisted IDEs, code-specific LLMs, multilingual programming, and Retrieval-Augmented Generation (RAG), highlighting both their capabilities and current limitations.

The comparative analysis identified key research challenges, including limited cross-language code translation, lack of explainable AI, cloud dependency, and insufficient educational support. To address these gaps, the survey proposed the **Programming Language Code Switching Language Model (CSLM)** as a future research direction. The proposed framework aims to provide a unified AI-powered environment for multilingual code generation, translation, compilation, debugging, and interactive learning.

As AI technologies continue to evolve, integrating explainable AI, multimodal interaction, and privacy-preserving LLMs will play a crucial role in developing more intelligent, accessible, and educational programming environments for future software development.

## ACKNOWLEDGMENT

The authors would like to express their sincere gratitude to the faculty members of the Department of Computer Science and Engineering, Sandip University, Nashik, for their valuable guidance, encouragement, and continuous support throughout this survey work. The authors also acknowledge the researchers and developers whose

published work and open-source contributions in Artificial Intelligence, Large Language Models, and AI-assisted programming have provided the foundation for this study.

## REFERENCES

1. M. Chen et al., “Evaluating Large Language Models Trained on Code,” arXiv:2107.03374, 2021.
2. F. Nijkamp et al., “CodeGen: An Open Large Language Model for Code Generation,” arXiv:2203.13474, 2022.
3. T. Le et al., “The Stack: 3 TB Dataset of Permissively Licensed Source Code,” arXiv:2211.15533, 2022.
4. OpenAI, “GPT-4 Technical Report,” arXiv:2303.08774, 2023.
5. B. Rozière et al., “Code Llama: Open Foundation Models for Code,” arXiv:2308.12950, 2023.
6. GitHub, “GitHub Copilot Documentation,” 2023. [Online]. Available: https://docs.github.com/en/copilot
7. J. Jiang et al., “A Survey on Large Language Models for Code Generation,” arXiv:2406.00515, 2024.
8. Y. Guo et al., “DeepSeek-Coder: Advancing Open-Source Code Intelligence,” arXiv:2401.14196, 2024.
9. R. Lozhkov et al., “StarCoder2 and The Stack v2: The Next Generation of Open Code Large Language Models,” arXiv:2402.19173, 2024.
10. Anthropic, “Claude 3 Model Card,” 2024. [Online]. Available: https://www.anthropic.com/news/claude-3-family
11. A. Anand et al., “A Comprehensive Survey of AI-Driven Advancements and Techniques in Automated Program Repair and Code Generation,” arXiv:2411.07586, 2024.
12. S. Bistarelli et al., “Usage of Large Language Model for Code Generation Tasks: A Review,” SN Computer Science, 2025, doi: 10.1007/s42979-025-04241-5.
13. N. Huynh and B. Lin, “Large Language Models for Code Generation: A Comprehensive Survey of Challenges, Techniques, Evaluation, and Applications,” arXiv:2503.01245, 2025.
14. B. Gülmez, “Code Generation with Large Language Models: A Survey from Neural Program Synthesis to Autonomous Software Development,” Applied Intelligence, 2026, doi: 10.1007/s10489-026-07230-0.
15. “Surveying the Benchmarking Landscape of Large Language Models in Code Intelligence,” ACM Transactions on Software Engineering and Methodology, 2026, doi: 10.1145/3800957.