

AI-Native Programming Languages: Reconsidering Programming Language Selection in the Era of Generative AI and Autonomous Coding

Swati Risil Patel^{1,*},

Abstract

The role of programming languages is undergoing a significant transformation as generative artificial intelligence becomes integrated into everyday software development. Programming is gradually moving from an activity dominated by manual source code production toward a collaborative environment in which developers work with large language models, intelligent coding assistants, automated testing systems, and increasingly autonomous software agents. This transformation introduces a new dimension to programming language selection. Historically, programming languages have primarily been compared using criteria such as execution performance, syntax, portability, ecosystem maturity, developer popularity, and learning complexity. These criteria remain important, but they do not fully capture the requirements of AI-supported software engineering. This paper examines recent developments in programming languages from the perspective of AI-assisted and agentic software development. It proposes the AI-Native Programming Language Readiness Framework (AIPLRF), a conceptual framework for assessing programming languages using six dimensions: AI generation compatibility, machine-verifiable structure, agent interaction efficiency, execution reliability, ecosystem maturity, and human verification effort. In addition, two research concepts are introduced: language-agent affinity (LAA) and verification debt (VD). LAA describes the degree to which an AI coding agent can effectively work with a particular programming language, whereas VD represents the human effort required to validate and correct AI-produced software. The paper argues that programming language evaluation should move beyond conventional popularity rankings and consider the complete human-AI software development lifecycle. Python, TypeScript, Rust, Java, Go, C++, and other languages may have different advantages depending on the computational layer and development objective. The paper further proposes an empirical methodology for evaluating programming languages using correctness, token consumption, compilation success, testing performance, security defects, debugging iterations, and human intervention. The proposed framework provides a foundation for future research into AI-native software engineering, programming education, secure code generation, and autonomous development systems.

Keywords: Programming languages, generative artificial intelligence, AI coding agents, AI-assisted software development, python, typescript, rust, software engineering, language-agent affinity, verification debt

*Author for Correspondence

Swati Risil Patel
E-mail: Swatidhamecha456@gmail.com

Assistant Professor, Sardar Patel College of Administration and Management, Sardar Patel Education Campus, Bakrol, Gujarat, India

Received Date: August 22, 2026
Accepted Date: September 05, 2026
Published Date: September 28, 2026

Citation: Swati Risil Patel. AI-Native Programming Languages: Reconsidering Programming Language Selection in the Era of Generative AI and Autonomous Coding. Recent Trends in Programming Languages. 2026; 13(2): 1–16p.

INTRODUCTION

Programming languages have always reflected changes in computing technology. The development of procedural programming, object-oriented programming, scripting languages, web programming, functional programming, and systems programming can be understood as responses to changing computational requirements.

The current transformation is fundamentally different because artificial intelligence is becoming an active participant in software development.

Modern AI systems can generate program fragments, explain source code, suggest alternative implementations, create test cases, identify possible defects, modify existing files, and assist with debugging. More advanced coding agents can operate across repositories and perform multiple development activities with comparatively limited direct intervention.

Consequently, programming is gradually changing from a predominantly human-to-machine activity into a human-AI-machine interaction process.

A simplified traditional development process can be represented as:

Requirement → Design → Coding → Testing → Debugging → Deployment

An AI-assisted process becomes:

Requirement → AI-Assisted Planning → Code Generation → Human Review → Testing → Refinement → Deployment

An agentic process can extend this further:

Goal → Agent Planning → Code Generation → Tool Execution → Testing → Error Analysis → Repair → Re-testing

This transformation creates an important research question:

Are existing programming languages equally suitable for software development in which artificial intelligence participates in code generation and verification?

This question is particularly relevant because current language trends show significant changes. TypeScript became the most-used language on the platform by contributor count in August 2025, surpassing Python and JavaScript, while Python continued to dominate AI-focused development [1]. Therefore, the issue is not simply identifying the most popular programming language.

The more important question is:

Which characteristics make a programming language effective when humans and AI systems jointly develop software?

This study addresses this question by proposing a framework for evaluating programming languages in the context of generative AI and autonomous coding.

CHANGING NATURE OF SOFTWARE DEVELOPMENT

From Manual Coding to AI-Assisted Development

Traditional software engineering assumes that developers are primarily responsible for converting requirements into executable programs.

AI-assisted development introduces a second participant into this process.

The developer may provide:

- Requirements;
- Architectural decisions;
- Constraints;
- Examples;
- Test conditions;
- Prompts.

The AI system may then provide:

- Implementation suggestions;
- Source code generation;
- Documentation;
- Test generation;
- Debugging suggestions;
- Refactoring alternatives.

The developer remains responsible for determining whether the resulting implementation is appropriate. This distinction is important because code generation and software correctness are not equivalent.

An AI system can produce code that appears syntactically correct while containing:

- Incorrect assumptions;
- Logical errors;
- Inappropriate APIs;
- Inefficient algorithms;
- Security weaknesses;
- Compatibility problems.

Recent survey data provide evidence of this trust challenge: although 84% of respondents reported using or planning to use AI tools, 46% said they did not trust the accuracy of AI-generated output [2, 3]. Therefore, the future of programming is not simply about generating more code. It is increasingly about generating code that can be efficiently verified.

RECENT TRENDS IN PROGRAMMING LANGUAGES

The programming language landscape is changing rapidly because of the interaction between AI, cloud computing, web development, systems programming, and developer tooling.

Type Script

TypeScript represents one of the strongest recent language trends. TypeScript became the most-used language on GitHub by contributor count in August 2025, adding more than one million contributors year-over-year and recording approximately 66% growth [4]. One reason for this growth is its type system.

When AI generates code, explicit type information can provide additional constraints that help identify inconsistencies. This growth has been associated with AI-assisted development and the need for stronger guardrails around generated code [5, 6].

This does not imply that TypeScript is universally superior. Rather, it indicates that type information may become increasingly valuable in AI-mediated programming.

Python

Python continues to occupy a central position in artificial intelligence, machine learning, data science, automation, and backend development.

Python accounted for nearly half of the new AI-focused repositories in a recent analysis, with approximately 582,196 AI repositories and 50.7% year-over-year growth [7]. Python's strength is based on several factors:

- Relatively accessible syntax;
- Extensive libraries;
- Mature machine learning ecosystem;
- Strong data science support;

- Large developer community;
- Rapid prototyping;
- Integration with AI frameworks.

A seven-percentage-point increase in Python adoption between 2024 and 2025 has also been reported [8]. Therefore, Python's current position demonstrates that ecosystem strength can be more important than language novelty.

Rust

Rust represents another important direction because it combines system-level performance with strong safety mechanisms. Its characteristics make it relevant to:

- Cloud infrastructure;
- Networking;
- Embedded systems;
- Security-sensitive applications;
- High-performance services;
- Developer tools;
- Systems software.

The importance of Rust demonstrates that AI-era programming is not solely about rapid code generation. In many environments, developers require code that is:

- Predictable;
- Efficient;
- Secure;
- Memory-safe;
- Maintainable.

Therefore, programming languages that provide strong compile-time guarantees may become increasingly important as AI-generated code becomes more common.

Java, Go, C++, and Other Languages

The AI era does not eliminate established programming languages.

- Java remains important for enterprise software.
- Go is widely associated with cloud and distributed systems.
- C++ continues to be relevant in high-performance computing, AI infrastructure, graphics, and hardware-oriented systems.
- JavaScript remains a major web development technology.
- C# remains important across enterprise, desktop, cloud, and game-development environments.

Consequently, the future is unlikely to be characterized by the disappearance of existing programming languages. Instead, it is more likely to involve greater specialization and cooperation among languages.

FROM LANGUAGE POPULARITY TO LANGUAGE SUITABILITY

Conventional programming language rankings commonly measure:

- Number of developers;
- Repository activity;
- Job-market demand;
- Developer preference;
- Ecosystem size;
- Learning interest.

These measures are useful but incomplete. AI-assisted development introduces additional questions:

- Can AI generate correct code in the language?
- How easily can generated code be checked?
- How many corrections are required?
- How many tokens are consumed?
- How efficiently can an AI agent debug the code?
- How much human supervision is necessary?
- How easy is it to detect security defects?

This leads to a fundamental shift:

Traditional Evaluation

Popularity → Productivity → Adoption

AI-Era Evaluation

Generation → Verification → Reliability → Human Oversight

This paper proposes that future programming language research should consider both models.

AI-NATIVE PROGRAMMING LANGUAGE

This paper introduces the concept of an AI-Native Programming Language. The term does not mean that the language itself must have been designed specifically for artificial intelligence. Instead, an AI-native programming language is defined as: a programming language whose syntax, type information, tooling, ecosystem, verification mechanisms, and development environment enable effective collaboration between human developers and AI-based software development systems. Four fundamental capabilities are proposed.

Generation

AI systems should be able to generate syntactically valid and semantically meaningful programs.

Verification

Generated code should be easy to inspect and validate using compilers, type systems, testing frameworks, static analyzers, and related mechanisms.

Agent Interaction

AI agents should be able to navigate projects, modify source files, execute tools, run tests, and recover from errors.

Maintenance

Human developers should be able to understand, modify, debug, and maintain AI-generated software.

These characteristics provide the foundation for the proposed framework.

AI-NATIVE PROGRAMMING LANGUAGE READINESS FRAMEWORK

The proposed AI-Native Programming Language Readiness Framework (AIPLRF) evaluates programming languages using six dimensions.

AI Generation Compatibility

This dimension measures how effectively AI systems can generate usable programs.

Possible indicators include:

- Compilation success;
- Functional correctness;

- API correctness;
- Unit-test success;
- Regeneration frequency;
- Prompt complexity.

A language in which AI-generated solutions frequently require extensive repair may have lower generation compatibility.

Machine-Verifiable Structure

AI-generated code requires mechanisms that can quickly identify mistakes. Relevant mechanisms include:

- Static typing;
- Compiler diagnostics;
- Type inference;
- Ownership systems;
- Contracts;
- Static analysis;
- Formal verification;
- Linting.

This dimension is particularly important because generated code can appear reasonable even when it contains subtle defects.

Agent Interaction Efficiency

AI coding agents do not simply generate code once. They may repeatedly:

1. Inspect files;
2. Formulate an implementation;
3. Modify code;
4. Compile;
5. Test;
6. Inspect errors;
7. Repair the implementation.

The efficiency of this loop can be measured using:

- Token consumption;
- Number of tool calls;
- Compilation failures;
- Test failures;
- Repair iterations;
- Execution time.

Recent research indicates that coding agent behavior can vary across programming languages, demonstrating the value of language-specific agent evaluation [8, 9].

Execution Reliability

A programming language should also be evaluated according to the reliability of the software it produces. Possible indicators include:

- Runtime errors;
- Test failures;
- Memory-related defects;
- Concurrency problems;
- Performance;
- Security vulnerabilities.

This dimension prevents language evaluation from focusing exclusively on generation speed.

Ecosystem Maturity

A strong ecosystem provides:

- Libraries;
- Frameworks;
- Documentation;
- Development tools;
- Testing frameworks;
- Package managers;
- Community support.

This is particularly important for AI because large and well-documented ecosystems provide more available examples and established development patterns.

Recent contributor and repository data illustrate this relationship: Python remains particularly strong for AI because of its extensive AI and data science ecosystem, while TypeScript benefits from modern application frameworks and typed development [10].

Human Verification Effort

The final dimension measures how much human work is required after AI generates code. This may include:

- Reading code;
- Reviewing architecture;
- Debugging;
- Testing;
- Security inspection;
- Correcting generated assumptions;
- Verifying dependencies.

This dimension leads directly to the concept of VD.

LANGUAGE-AGENT AFFINITY

This paper proposes language-agent affinity (LAA) as a measurable concept. LAA represents the effectiveness with which an AI coding agent can operate within a particular programming language. A basic representation is:

$$\text{LAA} = \text{Successful AI Tasks} / \text{Total AI Tasks}$$

A more comprehensive model could include five variables:

$$\text{LAA} = f(\text{C}, \text{T}, \text{E}, \text{H}, \text{R})$$

Where

- C, correctness;
- T, token efficiency;
- E, execution efficiency;
- H, human intervention;
- R, reliability.

A high LAA indicates that an AI agent can generate and maintain software in the language with comparatively little correction. A low LAA would suggest that the language creates greater difficulty for the agent. Importantly, LAA should be treated as task-dependent.

For example, a language may have a high LAA for web development but a lower LAA for systems programming. Therefore, there may be no universal language-agent affinity score. Instead, LAA should be measured according to application domain.

VERIFICATION DEBT

The second proposed concept is verification debt (VD). In traditional development, much of a developer's effort is invested in producing source code. AI changes this distribution. AI may reduce the amount of manual coding required while increasing the amount of code that needs to be inspected and validated. Therefore,

Verification debt = human validation effort generated by AI-produced software

Verification debt may include:

- Code review;
- Testing;
- Debugging;
- Security analysis;
- Dependency verification;
- Architecture validation;
- Correction of AI assumptions.

This leads to an important proposition:

Higher AI code generation productivity does not automatically result in lower overall software development costs. An AI system may generate 1,000 lines of code quickly; however, if a developer spends substantial time validating those lines, the total productivity gain may be smaller than expected. This is one of the most important research opportunities identified by the framework.

COMPARATIVE VIEW OF PROGRAMMING LANGUAGES

Table 1 summarizes this comparison across languages; it should not be interpreted as a universal ranking. Instead, it supports the argument that programming language selection should be context dependent.

POLYGLOT AI ARCHITECTURE

The emergence of AI-enabled systems is likely to strengthen rather than reduce multi-language development.

A hypothetical AI application could use:

- *Layer 1:* AI/ML: Python
- *Layer 2:* Application/API: TypeScript
- *Layer 3:* Performance-sensitive components: Rust or C++
- *Layer 4:* Cloud services: Go
- *Layer 5:* Enterprise integration: Java or C#

Table 1. Comparative overview of programming languages across AI-era considerations.

Language	Primary strength	AI-era opportunity	Major consideration
Python	AI, ML, data science	Model development and AI services	Runtime performance
Type Script	Typed application development	AI-enabled web and full-stack systems	Ecosystem complexity
Rust	Safety and performance	Infrastructure and secure systems	Learning curve
Java	Enterprise maturity	Large-scale AI integration	Development complexity
Go	Simplicity and concurrency	Cloud and distributed AI services	Ecosystem specialization
C++	High performance	AI infrastructure and hardware-oriented systems	Complexity
JavaScript	Web ecosystem	Rapid application development	Dynamic typing

This can be described as a polyglot AI architecture. The important question, therefore, becomes: Which language provides the best characteristics for each layer of an AI-enabled system? This is more meaningful than asking which single programming language will dominate the future.

AUTONOMOUS CODING AGENTS

AI coding assistants are evolving toward agentic systems. Instead of only suggesting code, an agent can potentially:

1. Understand a development objective;
2. Inspect the repository;
3. Identify relevant source files;
4. Formulate a solution;
5. Modify the implementation;
6. Execute tests;
7. Inspect errors;
8. Revise the implementation;
9. Repeat the process.

The software development loop therefore becomes increasingly iterative.

Conventional Model

Human → Code → Compiler → Human

AI-Assisted Model

Human → AI → Code → Compiler → Human

Agentic Model

Human → Agent → Code → Tools → Tests → Agent → Revised Code → Human

This change makes programming language characteristics increasingly relevant to AI agents.

PROGRAMMING LANGUAGE × AI AGENT INTERACTION

Traditional programming language benchmarks generally emphasize characteristics such as:

- Execution time;
- Memory consumption;
- Compilation speed;
- Language expressiveness.

Future AI-oriented benchmarks should additionally measure:

- Code generation accuracy;
- Token usage;
- Debugging efficiency;
- Tool interaction;
- Compilation success;
- Test completion;
- Human correction;
- Security.

This creates a new research domain:

- Programming Language × AI Agent Interaction
- The proposed AIPLRF can serve as a foundation for developing such benchmarks.

AI AND PROGRAMMING EDUCATION

The transformation is not limited to professional software development. Programming education must also respond. Traditional programming courses commonly assess whether students can manually

develop programs. AI-native programming education should additionally examine whether students can:

- Understand generated programs;
- Evaluate AI-generated solutions;
- Detect incorrect APIs;
- Identify security vulnerabilities;
- Test generated code;
- Optimize algorithms;
- Compare alternative implementations;
- Explain why a solution works.

A five-stage educational model is proposed.

Stage 1: Create—Students independently develop a solution.

Stage 2: Collaborate—Students use AI to generate an alternative solution.

Stage 3: Compare—Students analyze the differences between human and AI solutions.

Stage 4: Verify—Students execute tests and identify errors.

Stage 5: Improve—Students produce a final validated solution. The objective is not to prevent students from using AI. The objective is to teach them to use AI critically.

SECURITY IMPLICATIONS OF AI-GENERATED CODE

AI-assisted programming also creates security concerns. Potential weaknesses include:

- Unsafe API usage;
- Insecure authentication;
- Improper input validation;
- Vulnerable dependencies;
- Inadequate error handling;
- Exposed credentials;
- Excessive permissions;
- Obsolete libraries.

Therefore, an AI-native programming language should not be evaluated solely according to how easily an AI system can generate code. The effectiveness of the generated code in terms of verification and security should also be evaluated.

AI CODE SECURITY DENSITY

This paper extends the proposed framework with another possible metric.

AI Code Security Density (ACSD)

A simple representation is:

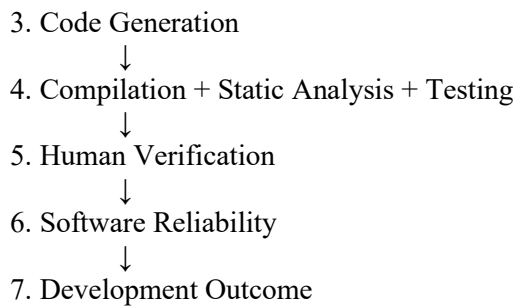
$$\text{ACSD} = \text{Security-Relevant Defects} / \text{AI-Generated Code Units}$$

For example, experiments could compare the number of security-related defects generated by AI agents when implementing equivalent applications in Python, TypeScript, Java, Rust, and other languages. Such a study could investigate whether stronger language-level constraints are associated with lower security-defect rates. This should remain an empirical question rather than an assumed conclusion.

PROPOSED RESEARCH MODEL

The proposed research model consists of the following stages:

1. Programming Language Characteristics
↓
2. AI Agent Interaction
↓



Three major outputs can be measured.

Output 1: AI Productivity—How quickly does the AI system generate a functional implementation?

Output 2: Verification Cost—How much human effort is required to validate and correct the result?

Output 3: Software Reliability—How correct, secure, maintainable, and efficient is the final implementation?

This model transforms programming language evaluation from a static comparison into a development process evaluation.

PROPOSED EXPERIMENTAL METHODOLOGY

The proposed framework can be validated through a controlled experiment.

Programming Languages

The initial experiment could evaluate:

- Python;
- TypeScript;
- Java;
- C#;
- Go;
- Rust;
- C++.

Benchmark Tasks

A benchmark containing approximately 100 tasks could be constructed. Tasks may cover:

- Algorithms;
- Data processing;
- REST APIs;
- Database operations;
- Authentication;
- File management;
- Networking;
- Concurrency;
- Error handling;
- Security-sensitive programming.

Each task should have equivalent functional requirements across languages.

EVALUATION METRICS

As shown in Table 2, these measurements can be combined to calculate an overall AI-native readiness score.

Table 2. Proposed evaluation metrics and measurement methods.

Metric	Measurement method
Functional correctness	Percentage of tests passed
Generation efficiency	Time to obtain first correct implementation
Token efficiency	Total tokens consumed
Compilation reliability	Successful compilation percentage
Debugging effort	Number of repair cycles
Human verification	Time spent reviewing/correcting
Security	Number of identified vulnerabilities
Maintainability	Structured code-quality assessment
Reliability	Failure rate across repeated executions

PROPOSED AI-NATIVE READINESS SCORE

A future empirical study could define:

$$\text{AIRS} = w_1G + w_2V + w_3A + w_4E + w_5S + w_6H$$

Where G, generation compatibility; V, verification strength; A, agent interaction efficiency; E, execution reliability; S, ecosystem strength; H, human oversight efficiency.

The values of the weights should not be arbitrarily assigned. They should be determined through:

- Expert surveys;
- Statistical analysis;
- Multi-criteria decision-making;
- Experimental results.

This makes the framework suitable for future empirical validation.

RESEARCH HYPOTHESES

The framework supports the following hypotheses.

- *H1*: Programming language characteristics significantly influence the functional correctness of AI-generated programs.
- *H2*: Languages providing stronger machine-verification mechanisms reduce human verification effort.
- *H3*: AI-agent token consumption differs significantly across programming languages.
- *H4*: Programming language type systems and compiler feedback are associated with lower verification debt.
- *H5*: Language-agent affinity varies according to programming task.
- *H6*: No single programming language provides optimal AI-agent performance across all software development domains.
- *H7*: Human-AI collaborative development produces significantly different productivity and reliability outcomes compared with human-only development.
- *H8*: AI-generated code requires measurable human verification effort even when functional tests are successfully passed.

These hypotheses provide a clear path for converting the conceptual paper into an experimental research project.

RESEARCH NOVELTY

The proposed research differs from conventional programming language comparison studies in three important ways.

AI-Native Language Readiness

Programming languages are evaluated according to their suitability for human-AI software development rather than their popularity alone.

Language-Agent Affinity

The framework treats the interaction between an AI coding agent and a programming language as a measurable research variable.

Verification Debt

The framework recognizes that reducing manual coding may shift development effort toward code validation.

The resulting research perspective is:
Generation → Verification → Reliability

rather than simply:
Generation → Productivity

This distinction is particularly relevant because current developer surveys show widespread AI use alongside substantial concerns about output accuracy.

POTENTIAL RESEARCH APPLICATIONS

The framework can support research in several areas.

AI Programming Benchmarks

A multilingual benchmark could compare coding agent performance across languages.

Programming Education

Researchers could investigate whether AI-supported programming improves learning outcomes.

Secure Software Development

Experiments could compare vulnerabilities in AI-generated implementations.

Energy-Efficient AI Coding

Researchers could measure computational resources and token consumption during AI-assisted programming.

Autonomous Software Engineering

Long-running coding agents could be evaluated across different language ecosystems.

Formal Verification

Researchers could investigate whether stronger language-level verification mechanisms reduce defects in AI-generated software.

FUTURE RESEARCH DIRECTION

The proposed framework can be extended into a complete empirical research platform. A future study could develop an AI-Native Programming Language Index (AIPLI). The index could combine:

- correctness;
- security;
- token efficiency;
- compilation success;
- debugging iterations;

- human verification;
- maintainability;
- execution performance.

An experimental platform can then automatically generate equivalent software tasks in multiple programming languages and provide the tasks to different AI coding agents. The system could collect the complete development trajectory:

Prompt → Generated Code → Compilation → Error → Agent Repair → Test → Final Code

This would allow researchers to study not only the final output but also how the AI reached the final solution. Such trajectory-based evaluation could become an important research direction in autonomous software engineering.

LIMITATIONS

The proposed framework is conceptual and requires empirical validation. Several limitations should therefore be recognized.

- First, AI performance changes rapidly as models and coding agents evolve. The results obtained using one model may not remain valid for future systems.
- Second, programming language performance may depend strongly on the task. A language that performs well for web development may not perform similarly for systems programming.
- Third, the benchmark design can influence the results. Public coding tasks may already be represented in AI training data.
- Fourth, token consumption alone cannot represent software quality.
- Finally, human verification is difficult to measure consistently because developers have different levels of experience.

Therefore, future experiments should use controlled tasks, multiple developers, multiple AI systems, repeated trials, and clearly defined evaluation criteria.

DISCUSSION

The current programming language ecosystem suggests that the future is unlikely to be determined by a single universal language. Recent contributor and repository data provide a useful illustration [3, 4]. TypeScript became the leading language by contributor count, while Python remained dominant in AI-focused repositories. These observations should not be interpreted as a competition in which one language must replace another.

- Instead, they indicate functional specialization.
- Python is strongly positioned in AI and data science.
- TypeScript is increasingly important for typed application development.
- Rust is positioned for performance and safety.
- Go remains useful for cloud infrastructure.
- Java and C# continue to serve large enterprise environments.
- C++ remains important in performance-critical computing.

This suggests that the future programming environment will likely be polyglot and AI-assisted.

The most successful developers may therefore be those who can determine:

1. Which language is appropriate;
2. Which AI tool is appropriate;
3. Which parts should be delegated;
4. Which parts require human implementation;
5. How generated software should be verified.

Programming expertise may consequently shift from writing every line toward making reliable engineering decisions.

CONCLUSION

The emergence of generative AI and autonomous coding agents is fundamentally changing the relationship between developers, programming languages, and software development tools. Current evidence demonstrates that programming language trends are already changing. TypeScript became the most-used language on GitHub by contributor count in August 2025, while Python remained highly dominant in AI-focused repositories. At the same time, widespread AI adoption has been accompanied by significant developer concerns regarding the reliability of generated output.

These developments suggest that traditional programming language rankings are no longer sufficient to understand language selection in AI-mediated development. This study proposes the AIPLRF. The framework evaluates programming languages through:

1. AI generation compatibility;
2. machine-verifiable structure;
3. agent interaction efficiency;
4. execution reliability;
5. ecosystem maturity;
6. human verification effort.

Two additional concepts were proposed:

1. LAA, which represents the effectiveness of an AI coding agent when working with a particular programming language; and
2. VD, which represents the human effort required to validate AI-generated software.

The framework emphasizes that software-generation speed alone is not an adequate measure of AI-assisted programming success. A system that produces code quickly but requires extensive human correction may not provide a meaningful overall productivity advantage. Therefore, the future of programming should be evaluated through:

Generation + Verification + Reliability + Human Judgment rather than generation speed alone.

The future is also unlikely to be characterized by one dominant programming language. Instead, different languages will probably serve different computational layers within AI-enabled systems. The emerging development paradigm can therefore be represented as:

Human Intent → AI Planning → Code Generation → Language-Level Constraints → Automated Verification → Human Judgment → Reliable Software

The central contribution of this paper is the proposition that programming language research should begin treating AI-agent compatibility and verification effort as first-class language-selection criteria. This creates a new research field at the intersection of:

Programming Languages + Generative AI + Software Engineering + Autonomous Agents provides several opportunities for future empirical research.

REFERENCES

1. Stack Overflow. (2021). Stack Overflow Developer Survey 2021 [Online]. Stack Overflow. Available from: <https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-integrated-development-environment>

2. Stack Overflow. (2025). Stack Overflow's 2025 Developer Survey reveals trust in AI at an all-time low [Online]. Stack Overflow. Available from: <https://stackoverflow.co/company/press/archive/stackoverflow-2025-developer-survey/>.
3. GitHub. (2025). A new developer joins GitHub every second as AI leads TypeScript to #1 [Online]. GitHub Blog. Available from: <https://github.blog/news-insights/octoverse/octoverse-a-new-developer-joins-github-every-second-as-ai-leads-typescript-to-1/>.
4. Lietzke A. (2025). TypeScript, Python, and the AI feedback loop changing software development [Online]. GitHub Blog. Available from: <https://github.blog/news-insights/octoverse/typescript-python-and-the-ai-feedback-loop-changing-software-development/>.
5. Birtolo D. (2026). What the fastest-growing tools reveal about how software is being built [Online]. GitHub Blog. Available from: <https://github.blog/news-insights/octoverse/what-the-fastest-growing-tools-reveal-about-how-software-is-being-built/>.
6. Siam MK, Gu H, Cheng JQ. Programming with AI: evaluating ChatGPT, Gemini, AlphaCode, and GitHub Copilot for programmers. In: Proceedings of the 3rd International Conference on Computing Advancements (ICCA 2024). Dhaka, Bangladesh: Association for Computing Machinery; 2024. p. 346-354. doi:10.1145/3723178.3723224.
7. Wang ZZ, Yang J, Lieret K, Tartaglini A, Chen V, Wei Y, et al. Humans are missing from AI coding agent research [Preprint]. 2026. arXiv:2608.12355.
8. Jimenez CE, Yang J, Wettig A, Yao S, Pei K, Press O, et al. SWE-bench: Can language models resolve real-world GitHub issues? [Preprint]. 2023. arXiv:2310.06770. doi:10.48550/arXiv.2310.06770.
9. Chen M, Tworek J, Jun H, Yuan Q, Pinto HPO, Kaplan J, et al. Evaluating large language models trained on code [Preprint]. 2021. arXiv:2107.03374.
10. Peng S, Kalliamvakou E, Cihon P, Demirer M. The impact of AI on developer productivity: evidence from GitHub Copilot [Preprint]. 2023. arXiv:2302.06590.