Phase 01

Code Summarization and Generation

Laiba Akram

42943

BSCS-7A

Theory Of programming Laguage

Title

Code Generation and Summarization Across Programming Languages: The Role of Syntax Complexity and PL Features

Abstract

Large Language Models (LLMs) such as GPT-4 and specialized code models (e.g., CodeT5, CodeBERT, StarCoder) are increasingly used for automatic code generation and source code summarization in modern software development.ACM Digital Library+1 Existing work evaluates these models mainly in terms of functional correctness and natural-language similarity (e.g., BLEU, ROUGE), often across several programming languages.ACL Anthology+1 However, there is limited understanding of how programming language (PL) features and syntactic complexity of code interact with model performance. Recent surveys on code summarization and LLMs for code highlight open questions about language diversity, structural metrics, and maintainability-oriented evaluation.MDPI+2SciTePress+2

This project proposes to empirically investigate whether PL features such as typing discipline (static vs dynamic), paradigm (imperative vs functional), and memory safety systematically influence LLM performance in code generation and summarization when measured through syntax-aware metrics such as Abstract Syntax Tree (AST) depth, branching factor, and traditional complexity measures (e.g., McCabe and Halstead).arXiv+2Codacy Blog+2 The central hypothesis is that languages with richer static structure and stricter type systems will exhibit distinct patterns of syntactic complexity in generated code, and that these patterns will correlate with summarization quality. The findings could guide both PL-aware evaluation benchmarks and the design of more language-sensitive code assistants, complementing existing code generation and summarization systems.ACM Digital Library+1

1. Introduction

Recent advances in AI-assisted development have produced systems that can **generate code from natural language and summarize existing source code into human-readable documentation**, as described in your Code Generation and Summarization System concept.

Code_Generation_Summarization

These tools aim to reduce boilerplate, improve understanding of legacy systems, and accelerate onboarding for developers and students. At the same time, programming languages differ significantly in their syntax and semantic features, including:

- Typing discipline – *Static typing* (e.g., Java, C++, Rust) vs *dynamic typing* (e.g., Python, JavaScript).
- Paradigm – *Imperative / object-oriented* (Java, C#), *functional* (Haskell, Erlang), *logic* (Prolog), or *multi-paradigm* languages.Medium+1
- Memory safety and management – Languages with built-in memory safety guarantees (e.g., Rust, Go) vs manual or less restrictive memory models (e.g., C, C++).
- Concurrency models – Threads with shared memory (Java, C++), actor-based systems (Erlang), async/await and event loops (JavaScript), etc.Medium

From a Programming Languages (PL) perspective, these features affect the structure of Abstract Syntax Trees (ASTs), control-flow graphs, and other internal representations used by compilers and static analysis tools. AST-based metrics such as tree depth, branching factor, and number of distinct node types capture syntax complexity, i.e., "how structurally complex a program's syntactic form is."arXiv+1 Classic software metrics like McCabe's cyclomatic complexity and Halstead's metrics also quantify control-flow and lexical complexity.Scispace+3GeeksforGeeks+3verifysoft.com+3

While LLMs are now evaluated on multi-language benchmarks for code generation and summarization, most studies focus on accuracy and correctness, not on how PL features and syntax complexity shape model behaviour.ACM Digital Library+2ACL Anthology+2 This creates a promising research opportunity at the intersection of Programming Languages, software metrics, and AI-for-code.

2. Literature Review

2.1 LLMs for Code Generation

Several recent works systematically evaluate LLMs for code generation. Chen et al. and subsequent studies investigate LLMs on benchmark suites such as HumanEval-X, APPS, and BigCodeBench, assessing functional correctness via unit tests and semantic evaluation frameworks like CodeJudge.MIT Press Direct+3ACM Digital Library+3arXiv+3 These evaluations often cover multiple languages (e.g., Python, Java, C++, JavaScript, Go) but treat language mostly as a dataset dimension, not as a first-class PL object with specific features.

A recent comparative study evaluates several state-of-the-art LLMs on Python, Java, and Swift, highlighting performance differences but not deeply linking them to language design choices (typing, memory safety, paradigm).ResearchGate Another line of work looks at LLMs' ability to generate code clones across C++, Java, and Python, again focusing on dependability and correctness rather than syntactic structure.ScienceDirect

Overall, these studies show that language matters for performance, but they do not explain *why* through PL-level properties or syntax complexity metrics.

2.2 Code Summarization and Multi-Language Evaluation

Code summarization aims to generate brief natural language descriptions of code snippets to aid comprehension and maintenance. Surveys by Zhang et al. and more recent systematic reviews map the evolution from sequence-to-sequence models to Transformer-based architectures and LLM-driven summarization.MDPI+1

Sun et al. (2024) explicitly evaluate LLM code summarization across multiple programming paradigms, constructing datasets for languages such as Erlang and Haskell (functional) and Prolog (logic). Their results suggest that language paradigm influences summarization difficulty, but they stop short of quantifying syntax complexity or tying results to concrete PL metrics.arXiv

Recent work on project-specific code summarization with in-context learning shows that context and project conventions affect summarization quality, yet again does not systematically layer PL features or structural metrics into the analysis.ScienceDirect

2.3 Syntax Complexity and PL-Level Metrics

In programming languages and software engineering, syntax and structural complexity are well-studied:

- AST-based metrics: several works compare AST representations across tools (ANTLR, Tree-sitter, srcML) using metrics like tree size, depth, branching factor, and token diversity to characterise syntactic structure across languages.arXiv+2ResearchGate+2
- Control-flow and lexical metrics: McCabe's cyclomatic complexity measures the number of independent control paths in a program, while Halstead's metrics use distinct operators/operands to estimate volume, difficulty, and effort.Scispace+4GeeksforGeeks+4verifysoft.com+4
- Software metrics tooling: industrial tools (e.g., McCabe IQ, SonarQube) apply these metrics at scale to identify complex, error-prone modules.mccabe.com+1

These metrics provide a precise way to define "syntax complexity":

For this project, *syntax complexity* will refer to structurally quantifiable properties of code such as AST depth, branching factor, number of distinct node types, cyclomatic complexity, and Halstead volume for function- or method-level code units.

However, very few LLM-for-code papers explicitly incorporate these metrics when evaluating generated or summarized code. The PL dimension (typing, paradigm, memory model) is rarely analysed alongside such structural measures.

2.4 Research Gap and Hypothesis

Gap 1 – Lack of PL-aware evaluation:

Existing evaluations of code-generation and summarization models compare languages mainly by aggregate accuracy (pass@k, BLEU, ROUGE, etc.) without systematically linking differences to **PL features** like typing discipline, paradigm, or memory safety.MDPI+4ACM Digital Library+4ACL Anthology+4

Gap 2 – Missing connection between syntax complexity and model performance:

AST-based and classic complexity metrics are well established in PL and software metrics research, but they are seldom used to explain why LLMs may perform better on some languages or tasks.Meng Yan's Home Page+3arXiv+3ResearchGate+3

Gap 3 – Limited integration into practical AI coding tools:

Industrial and academic systems for code generation/summarization, including the architecture described in your project (frontend, backend API, AI model layer), typically do not adapt behaviour based on language-specific syntax complexity or PL properties.

Proposed Research Question

RQ: How do programming language features (typing discipline, paradigm, memory safety) and syntax complexity metrics (AST-based and traditional) influence LLM performance in code generation and code summarization?

Hypothesis

H1: For function-level tasks of equivalent functionality, LLM-generated code in **statically typed, memory-safe languages** (e.g., Rust, Go, Java) will exhibit **higher syntax complexity** (e.g., deeper ASTs, higher cyclomatic complexity) than generated code in **dynamically typed languages** (e.g., Python, JavaScript), and these complexity differences will correlate with **differences in summarization quality** across languages.

This hypothesis directly connects **PL concepts** (static vs dynamic typing, memory safety, paradigm) with **measurable syntax complexity** and **LLM performance**, satisfying the rubric's emphasis on accurate PL feature definitions and a clearly identified research gap in the current state of the art.