Media Engineering and Technology Faculty
German University in Cairo

AI-Powered COSMIC Functional Size Measurement (FSM)

Bachelor Thesis

Author: Ahmed Mohamed El-Gohary
Supervisors: Dr. Milad Ghantous
Dr. Hassan Soubra

Submission Date: 29 May, 2025

Media Engineering and Technology Faculty
German University in Cairo

AI-Powered COSMIC Functional Size Measurement (FSM)

Bachelor Thesis

Author: Ahmed Mohamed El-Gohary
Supervisors: Dr. Milad Ghantous
Dr. Hassan Soubra

Submission Date: 29 May, 2025

This is to certify that:

(i) the thesis comprises only my original work toward the Bachelor Degree

(ii) due acknowlegement has been made in the text to all other material used

---

Ahmed Mohamed El-Gohary
29 May, 2025

Acknowledgments

I sincerely thank my supervisors, Dr. Milad and Dr. Hassan, for their guidance and support throughout this research. Their helpful suggestions and time spent in our meetings encouraged me to develop COSMIC-AI with confidence and independence. I am grateful for their mentorship. I also thank my parents for their constant support and encouragement, which helped me through challenging moments. Their belief in me was invaluable.

Abstract

Accurate software size estimation remains a critical challenge in modern software engineering, with traditional manual Common Software Measurement International Consortium (COSMIC) Functional Size Measurement (FSM) proving time-intensive and error-prone. This research presents an automated approach to COSMIC FSM using transformer-based deep learning, specifically fine-tuning the CodeBERT model for line-level prediction of functional movements across multiple programming languages. The study develops a comprehensive dataset encompassing Java, Python, C, and Arduino code with detailed COSMIC annotations, enabling multi-output regression prediction of Entry (E), Exit (X), Read (R), and Write (W) movements. The fine-tuned CodeBERT model achieves strong performance with a Mean Squared Error (MSE) of 0.095 for total COSMIC Function Points (CFP) and an R^2 of 0.887, explaining approximately 88.7% of variance in functional size predictions. Validation against published expert measurements demonstrates 96% accuracy on C programs and 88% accuracy on Arduino code. The research culminates in COSMIC-AI, a web-based application that provides practitioners with accessible, automated COSMIC measurement capabilities. This work establishes the viability of transformer-based automation for functional size measurement, offering significant improvements in efficiency while maintaining measurement accuracy comparable to expert manual analysis.

Contents

Chapter 1

Introduction

1.1 Understanding Functional Size in Software

The concept of functional size in software refers to the measurable amount of functionality that a system delivers to its users. It is a critical aspect in software engineering, offering a reliable foundation for activities like project planning, effort estimation, and performance evaluation. By capturing and quantifying functional requirements, it helps teams to assess the scope and scale of software applications with precision [25].

1.2 Functional Size Measurement (FSM)

Functional Size Measurement (FSM) is the systematic process of determining the size of a software system based on its functional components. FSM provides a standardized framework to evaluate software by measuring user-recognizable functionalities, such as inputs, outputs, data movements, and interactions. This process is not dependent on technical implementation details, making it adaptable across various technologies and software types. FSM serves as a foundation for reliable cost estimation, progress monitoring, and quality assurance in software projects. Its consistent application helps organizations achieve greater predictability and control in development activities [28, 1].

1.3 Established Methods for Functional Size Measurement

Over the years, several functional size measurement methods have emerged to meet the evolving needs of software engineering. Presented below in the order of their original development, these approaches reflect the progression from early transactional models to more refined, process-based methodologies:

- **IFPUG (International Function Point Users Group)** [16]: Originating from Albrecht's work in 1979 and institutionalized by IFPUG in 1986, this method provided the first systematic approach to measuring functional size. It remains one of the most influential frameworks, focusing on transactions, external interfaces, and logical data structures.
- **Mark II Function Point Analysis** [15]: Developed by Charles Symons in the late 1980s, Mark II extended the function point approach with greater flexibility and structured rules, emphasizing the measurement of user-driven data processes and information movements.
- **NESMA (Netherlands Software Measurement Association)** [19]: NESMA emerged in the early 1990s, adapting IFPUG principles to local industry requirements. It introduces tailored measurement guidelines that offer clarity for organizations operating within European and global contexts.
- **FiSMA (Finland Software Measurement Association)** [17]: Developed towards the end of the 1990s, FiSMA introduced national adaptations that align functional size measurement with Finnish development practices while preserving core measurement integrity.
- **COSMIC (Common Software Measurement International Consortium)** [18, 7]: Introduced in 1999, COSMIC represents a second-generation method that departs from transactional models by focusing on the measurement of functional processes. It evaluates data movements—E, X, R, and W—and is applicable to modern domains such as embedded systems, web applications, mobile platforms, and service-oriented architectures.

By reviewing these methods in their order of creation, one can observe how each addresses the measurement challenges of its era, collectively contributing to the evolution of flexible, precise, and domain-independent functional size measurement approaches.

1.4 The COSMIC Method: Modern Foundations and Unique Advantages

The COSMIC functional size measurement method is recognized as a second-generation standard, developed to address the limitations of earlier methods such as IFPUG, MkII, NESMA, and FiSMA [28, 7]. It measures software functionality by analyzing functional processes, each defined by four types of data movements: E, X, R, and W [7, 1].

This approach makes COSMIC applicable across diverse domains, including business, real-time, and embedded systems. Its technology-agnostic design allows it to adapt to complex, evolving architectures [25]. By focusing on user-recognizable functionality, it supports accurate estimation and aligns measurement with business needs [28].

Maintained and updated by the COSMIC Measurement Practices Committee, the method continues to evolve with industry developments [7], ensuring its relevance and reliability in functional size measurement.

1.5 Research Motivation

The growing complexity of modern software systems, driven by architectures such as microservices, distributed environments, and cloud platforms, exposes the limitations of traditional functional size measurement methods [28]. Manual measurement techniques are increasingly impractical due to their time-consuming nature and susceptibility to human error.

This research is motivated by the need for automated measurement solutions based on the COSMIC method, capable of adapting to diverse technologies and development practices. Such tools are essential for improving estimation accuracy and supporting real-time decision-making in agile and continuous delivery environments [9].

1.6 Research Objective

The primary objective of this research is to develop an automated system for COSMIC FSM that enhances efficiency and accuracy in software size estimation while maintaining accessibility for users with varying levels of technical expertise. By fine-tuning the CodeBERT transformer model [10], the study aims to automate the prediction of COSMIC functional movements—E, X, R, and W—at the line level across multiple programming languages. Additionally, the research seeks to implement a user-friendly interface through the COSMIC-Artificial Intelligence (AI) web application to deliver these predictions in a practical, scalable, and intuitive manner.

1.7 Structure of the Thesis

The rest of this thesis is organized as follows: Chapter 2 provides background information on the COSMIC method and reviews related work in automated functional size measurement. Chapter 3 presents the methodology, including dataset preparation, the CodeBERT fine-tuning process, and the development of the COSMIC-AI web application. Chapter 4 presents the experimental results and validation against real-world case studies. Finally, Chapter 5 concludes the research and discusses future directions.

Chapter 2

Background and Related Work

2.1 Overview of the COSMIC Method

The COSMIC (Common Software Measurement International Consortium) method is one of the most widely accepted FSM standards. It measures software functional size by analyzing functional processes and decomposing them into four types of data movements: Entry (E), Exit (X), Read (R), and Write (W).

2.1.1 Fundamental Concepts

The COSMIC method is built on several key conceptual foundations:

- **Functional User Requirements (FUR):** At the core of the COSMIC method are the Functional User Requirements, which represent the *essential* functionality that users expect from the software system. These requirements describe what the software must do, not how it does it [1].
- **Cosmic Function Points (CFP):** The method quantifies software size using Cosmic Function Points, a standardized unit of measurement. One CFP represents the effort required to process a *data movement*, providing a consistent way to compare software across different domains and technologies [7].

2.1.2 COSMIC Measurement Process and Functional Size Principles

The COSMIC method adopts a systematic three-stage process to quantify the functional size of software systems. This process includes defining a measurement approach, aligning software features with standardized components, and performing the size calculation [1]. Figure 2.1 illustrates this comprehensive three-step measurement process.

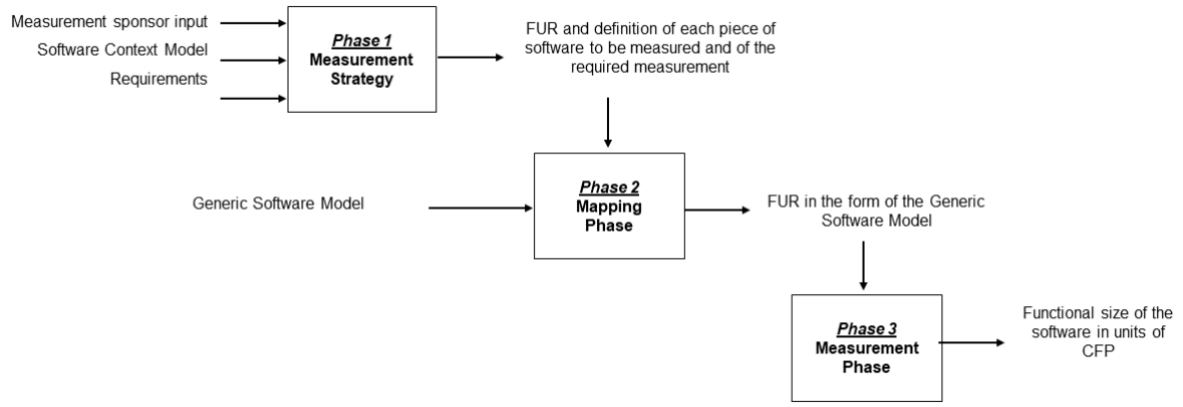

Figure 2.1: The three-step COSMIC measurement process [1].

- **Defining the Measurement Strategy:** The first stage involves clarifying the purpose of the measurement and specifying what will and will not be measured. This includes setting the scope, determining which software components and user interactions are to be considered, and outlining any constraints. This planning step results in a contextual model that outlines how the software interacts with users and persistent storage [1].

- **Mapping Functional Requirements:** Once the scope is clear, functional requirements are translated into a set of standard processes described in the COSMIC model. These processes are broken down into distinct interactions that represent data exchanges or movements. Each of these movements is classified into one of four categories, ensuring consistency in how systems of various types and complexities are measured [1].

- **Measuring Functional Size:** In the final phase, each identified data interaction is assigned a uniform measurement unit called a COSMIC Function Point (CFP). The size of the entire software system is calculated by summing up these points across all processes and data exchanges. This allows for objective comparisons between projects and provides a foundation for estimation and benchmarking [1].

2.1.3 Classification of Data Movements

The COSMIC approach organizes all data movements into four fundamental actions, which together describe how the software system communicates internally and with external entities. These four types of data movements are depicted in Figure 2.2.

- **Entry (E):** Input of data from outside the system into a process.

- **Exit (X):** Output of information from a process to an external user or system.

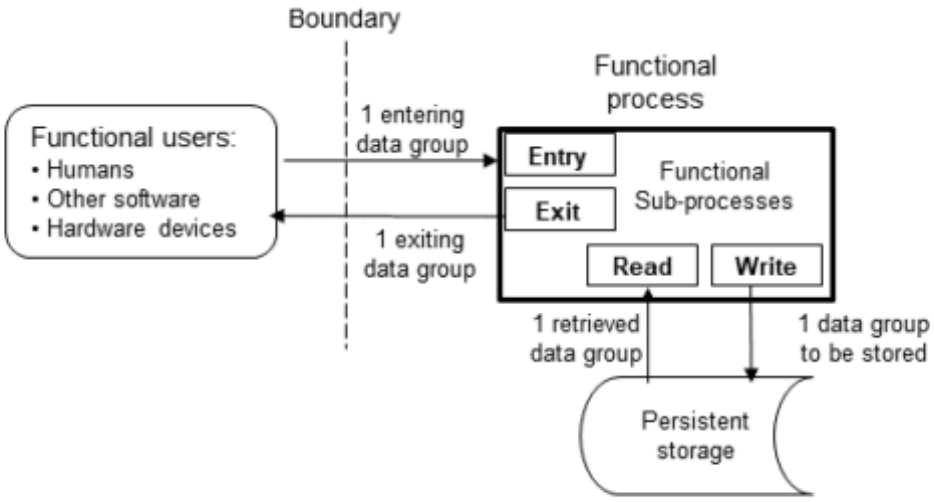

Figure 2.2: The four data movement types defined in COSMIC [1].

- **Read (R):** Retrieving stored data from persistent storage for use by a process.
- **Write (W):** Saving or updating data in persistent storage [1].

2.1.4 Core Measurement Principles

In line with software engineering fundamentals, COSMIC emphasizes the following principles when measuring functional size:

- Software functions are separated into distinct processes, each consisting of smaller activities representing data movements or transformations.
- A functional process is initiated when it receives external input (E) and is completed when an output (X) is provided.
- Internal manipulations of data are not counted unless they result in one of the four recognized movement types.
- The functional size of a system is determined by tallying all identified Entries, Exits, Reads, and Writes, with each action contributing one CFP to the total measurement [1].

2.2 Related Work

Functional Size Measurement (FSM) has been a critical area of research in software engineering, with numerous attempts to automate the COSMIC (Common Software Measurement International Consortium) method across various programming paradigms and application types [29].

2.2.1 Automation Approaches Across Programming Languages

Researchers have developed automation strategies for multiple programming languages and frameworks. Koulla et al. [20] presented an automated COSMIC functional size measurement approach for C programs using regular expressions. Similarly, Sahab [23] developed the CFP4J library for automating COSMIC measurement in Java web applications using the Spring MVC framework, achieving over 90% accuracy compared to manual measurement.

Expanding the scope of language-agnostic measurement, Soubra et al. [27] proposed a 'universal' tool based on the COSMIC method and MIPS instruction set architecture to automate measurement across different programming languages. Darwish [8] extended this approach to ARM assembly programs, highlighting the importance of having COSMIC measurement rules for low-level languages.

2.2.2 Domain-Specific Measurement Techniques

Researchers have explored specialized approaches for different software domains. Haoues et al. [11] proposed a measurement procedure specifically for web and mobile applications using COSMIC FSM method, Functional User Requirements (FUR) from UML Use Case Diagrams. Chamkha et al. [6] developed JavaCFP, a tool for automatically measuring the COSMIC functional size of Java Swing applications throughout their development lifecycle.

2.2.3 Machine Learning and Natural Language Processing Approaches

Recent studies have leveraged advanced computational techniques for functional size estimation. Tenekeci et al. [30] introduced a deep-learning-based NLP model (CodeBERT) that achieved 84.5% accuracy in predicting COSMIC functional size directly from code. Ochodek et al. [22] proposed a deep learning model capable of approximating COSMIC functional size based solely on use-case names, outperforming baseline techniques by approximately 20% in standardized accuracy.

2.2.4 Ontology and Requirements-Based Approaches

Bagriyanik [4] proposed an automated method using a requirements engineering ontology to measure functional size, eliminating the subjectivity of manual measurement. Hussain et al. [14] developed a methodology to approximate COSMIC functional size from informally written textual requirements, supporting early effort estimation in Agile development processes.

2.2.5 Emerging Conceptual Innovations

Attallah [3] proposed an innovative approach by introducing a new programming language compiler based on COSMIC FSM, allowing developers to express software requirements directly using COSMIC vocabulary and automatically obtain functional size measurements.

Chapter 3

Methodology

3.1 Development Process Overview

This research presents COSMIC-AI, an automated software functional size measurement system that leverages transformer-based deep learning to predict COSMIC Functional Size Measurement (FSM) at the line level. The system addresses the limitations of traditional manual COSMIC measurement by providing accurate, efficient, and scalable automation across multiple programming languages.

The development of COSMIC-AI follows a four-phase approach, as illustrated in Figure 3.1:

Phase 1 - Dataset Construction and Annotation: A comprehensive multi-language dataset was constructed containing Java, Python, C, and Arduino source code with detailed COSMIC annotations. For C and Arduino programs, an automated tool (`COSMICAnalyzer`) was developed to generate measurements using regex-based pattern matching, while Java and Python code underwent manual annotation following COSMIC measurement standards. Each code line was systematically labeled with its corresponding Entry (E), Exit (X), Read (R), and Write (W) movement counts to create a robust training foundation.

Phase 2 - Model Development and Training: The pre-trained CodeBERT transformer model was fine-tuned on the annotated dataset using a multi-output regression approach. Rather than treating COSMIC measurement as a classification problem, the model was trained to predict continuous values for each of the four movement types simultaneously. This methodology captures the nuanced combinations of functional movements that can occur within individual code lines and enables more accurate predictions.

Phase 3 - Model Validation and Testing: The trained model underwent comprehensive evaluation to assess its prediction accuracy across different programming languages and code patterns. Performance metrics were established to validate that the automated predictions achieve accuracy comparable to expert manual analysis, ensuring the reliability of the final system.

Phase 4 - Web Application Development: The validated model was integrated into a web-based application (COSMIC-AI) that provides practitioners with an accessible interface for automated COSMIC measurement. The application processes uploaded source code files by tokenizing each line, generating predictions for all four COSMIC movement types, and computing total COSMIC Function Points (CFP) for both individual lines and aggregate functional size estimates.

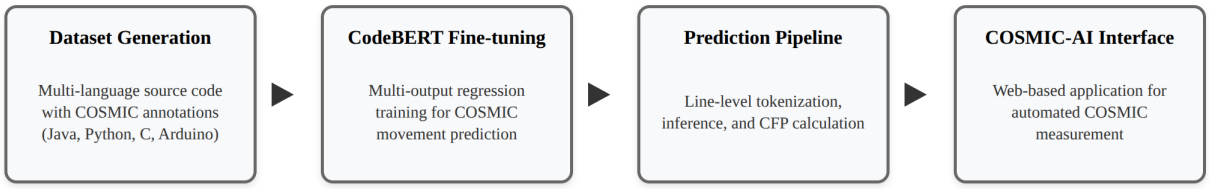

Figure 3.1: Development Process for COSMIC-AI

3.2 Dataset Preparation

The dataset in this research consists of a combination of small to medium open-source projects and manually generated codes.

3.2.1 Dataset Structure and Format

The dataset used is structured specifically to support line-level COSMIC functional size measurement (FSM). Each row in the dataset represents a single line of source code and is annotated with its corresponding functional movements: Entry (E), Exit (X), Read (R), and Write (W), along with the computed total CFP. Table 3.1 demonstrates this structured format with sample code lines and their corresponding movement annotations.

Table 3.1: Sample of the labeled dataset format

Code	E	X	R	W	CFP
`int sum(int x, int y) {`	2	1	0	0	3
`int z = 0;`	0	0	0	1	1
`z = x + y;`	0	0	1	1	2
`return z;`	0	0	1	0	1
`void checkEvenOrOdd(int x) {`	1	0	0	0	1
`if (x % 2 == 0)`	0	0	1	0	1

This structured format serves several critical purposes:

- **Fine-Grained Measurement:** Each line of code is assessed independently to capture every functional movement. This level of granularity is necessary because individual lines may contain multiple COSMIC operations (e.g., a line could R and W at once), which a pure classification approach might fail to detect.

- **Regression over Classification:** Rather than using classification (e.g., predicting a single label per line), this format supports *multi-output regression* where the model learns to predict continuous values (typically in the range 0–10) for each of the E, X, R, and W dimensions. This ensures that the nuanced combinations of movements are preserved.

- **Derived CFP Calculation:** The `CFP` column is derived as the sum of the four movement values:

$$\texttt{CFP} = E + X + R + W$$

  This simplifies total function point computation and allows easy verification of model predictions.

- **Training Readiness:** The tabular structure, with clearly defined columns, makes the dataset suitable for direct input into machine learning pipelines (e.g., using `Pandas` [21] and `NumPy` [12]) and for fine-tuning transformer models like CodeBERT.

3.2.2 Programming Languages Represented

To ensure broad coverage of programming paradigms and application domains, the dataset includes source code from four distinct languages: Java, Python, C, and Arduino (a C/C++-based variant for embedded systems). Table 3.2 provides an overview of the key characteristics and typical usage patterns for each programming language included in the dataset.

Table 3.2: Overview of Programming Languages in the Dataset

Language	**Paradigm**	**Notes / Typical Usage**
Java	Object-oriented, class-based, static typing	Widely used in enterprise applications, Android development, and teaching. Verbose syntax with explicit class and method definitions facilitates clear E and X movements.
Python	Multi-paradigm (procedural, object-oriented, functional), dynamic typing	Popular for scripting, data science, and rapid prototyping. Indentation-based blocks and concise syntax yield varied R and W patterns per line.
C	Procedural, static typing	Low-level systems programming and performance-critical code. Manual memory management and pointer usage influence multiple functional movements on single statements.
Arduino	Embedded C/C++ variant, static typing	Controls microcontrollers for IoT and robotics. Includes hardware-specific I/O calls (e.g., `digitalRead`, `analogWrite`) that map directly to COSMIC R and W movements.

The distribution of code lines across the four programming languages in our dataset is illustrated in Figure 3.2.

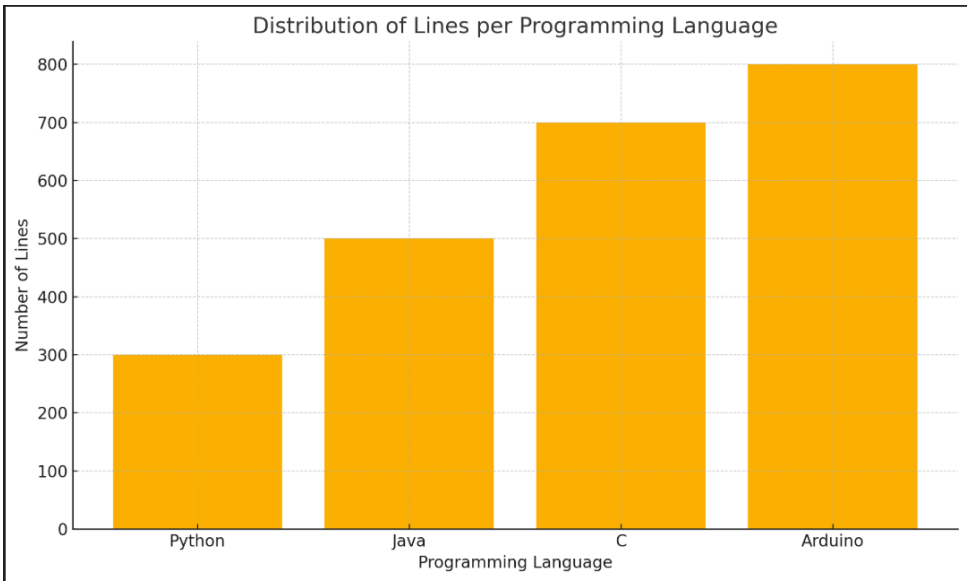

Figure 3.2: Approximate Distribution of lines per programming language

3.2.3 Automated Sizing for Arduino/C Programs

I developed an automated tool, `COSMICAnalyzer`, to measure the functional size of Arduino and C programs using the COSMIC method, facilitating efficient dataset generation. The tool builds on:

- Soubra & Abran's IoT mapping framework for Arduino [26].
- Koulla et al.'s regex-based COSMIC measurement for C [20].

The tool's implementation was verified using examples from the referenced papers, with calculated functional sizes matching the reported values at 100% accuracy.

Processing Pipeline The automated tool follows a systematic six-stage workflow as depicted in Figure 3.3.

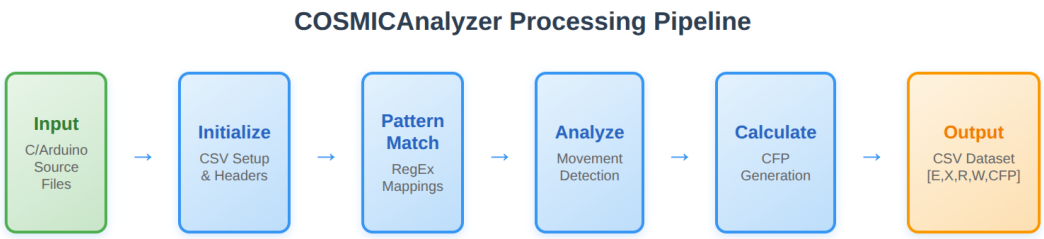

Figure 3.3: `COSMICAnalyzer` Processing Pipeline

Regex Mapping Samples Figures 3.4 and 3.5 illustrate portions of the two mapping tables.

```
C_MAPPINGS = [
    (r'\b(?:int|float|double|char)\s+([^;]+);', 'var_init'),
    (r'\b(\w+)\s+\w+\s*\(([^)]*)\)\s*\{', 'func_header'),
    (r'\breturn\b\s+([^;]+);', 'return_stmt'),
    (r'\bscanf\s*\(([^)]+)\)\s*;', 'scanf'),
    (r'\bprintf\s*\(([^)]+)\)\s*;', 'W'),
    (r'\b\w+\s*=\s*[0-9]+(?:\.[0-9]+)?\s*;', 'W'),
    (r'\b\w+\s*=\s*\w+\s*;', 'RW'),
    (r'\b\w+\s*=\s*[^=]+\s*[+\-*/]\s*[^;]+;', 'RW'),
    (r'\b\w+\s*(?:\+=|-=|\+\+|--)\s*[^;]*;', 'RW'),
    (r'\b\w+\s*(?:==|!=|>|<|>=|<=)\s*[0-9]+(?:\.[0-9]+)?', 'R'),
    (r'\b\w+\s*(?:==|!=|>|<|>=|<=)\s*\w+', 'RR'),
]
```

Figure 3.4: Sample of C regex mappings

```
ARDUINO_MAPPINGS = OrderedDict([
    (r'\bdigitalRead\s*\(', 'E'), (r'\banalogRead\s*\(', 'E'),
    (r'\bdigitalWrite\s*\(', 'X'), (r'\banalogWrite\s*\(', 'X'),
    (r'\bdelay\s*\(', 'X'),
    (r'\bSerial\.begin\s*\(', 'X'),
    (r'\bSerial\.read(?:String)?\s*\(', 'E'),
    (r'\bSerial\.available\s*\(', 'E'),
    (r'\bSerial\.print(?:ln)?\s*\(', 'X'),
    (r'\bSerial\.write\s*\(', 'X'),
    (r'\bSoftwareSerial\.begin\s*\(', 'X'),
    (r'\bSoftwareSerial\.read(?:String)?\s*\(', 'E'),
    (r'\bSoftwareSerial\.available\s*\(', 'E'),
    (r'\bSoftwareSerial\.print(?:ln)?\s*\(', 'X'),
    (r'\bSoftwareSerial\.write\s*\(', 'X'),
    (r'\blcd\.begin\s*\(', 'X'),
    (r'\blcd\.(?:print|println|setCursor|clear|home)\s*\(', 'X'),
    (r'\bu8g2\.begin\s*\(', 'X'),
    (r'\bu8g2\.(?:print|printStr|draw[A-Za-z]*)\s*\(', 'X'),
    (r'\bWiFi\.begin\s*\(', 'X'),
    (r'\bWiFiClient\.read\s*\(', 'E'),
    (r'\bWiFiClient\.available\s*\(', 'E'),
```

Figure 3.5: Sample of Arduino regex mappings

Sample of Tool Output The automated analysis produces structured output in CSV format, as demonstrated in Table 3.3, which shows six representative lines with their calculated movement counts.

Table 3.3: Sample Output from `COSMICAnalyzer`

Code	E	X	R	W	CFP
`int sum(int x, int y) {`	2	1	0	0	3
`int z = 0;`	0	0	0	1	1
`z = x + y;`	0	0	1	1	2
`return z;`	0	0	1	0	1
`void checkEvenOrOdd(int x) {`	1	0	0	0	1
`if (x % 2 == 0)`	0	0	1	0	1

This automated approach ensures consistent, reproducible COSMIC sizing across different codebases and directly produces the dataset's format.

3.2.4 Manual Sizing for Java and Python Programs

Manual COSMIC sizing for high-level languages like Java and Python follows the rules in the COSMIC Measurement Manual [18, 7] and the official online documentation [1]. Figure 3.6 presents Besada's [5] comprehensive mapping framework for object-oriented languages, which guided the manual annotation process. I adopted these guidelines to hand-measure every line in my samples.

COSMIC	Object Oriented Languages	Examples
Entry data group movements	create a function get a user input	public static void main() function add() int x = scanner.nextInt();
Exit data group movements	print data return data	print(); System.out.println(); return ;
Read data group movements	read data from a file	line = reader.readLine()
Write data group movements	writing data to a file	writer.write(data);

Figure 3.6: Besada's [5] mapping for Object-Oriented Languages

Java Samples Table 3.4 presents excerpts from the manually annotated Java code samples, demonstrating how object-oriented constructs and method calls are mapped to COSMIC movements.

Table 3.4: Manual COSMIC sizing for Java (excerpt)

Code	E	X	R	W	CFP
`int idx = digit - '0' - 1;`	0	0	1	1	2
`row[r].set(idx);`	0	0	1	1	2
`col[c].set(idx);`	0	0	1	1	2
`int currentSubgrid = getSubgrid(r, c);`	0	0	0	1	1
`subgrid[currentSubgrid].set(idx);`	0	0	1	1	2
`private static boolean solve(...)`	6	1	0	0	7
`public static void solveSudoku(...)`	1	0	0	0	1

Python Samples Similarly, Table 3.5 shows representative Python code lines with their corresponding COSMIC annotations, illustrating the measurement approach for dynamic language constructs and library function calls.

Table 3.5: Manual COSMIC sizing for Python (excerpt)

Code	E	X	R	W	CFP
`if is_prime[num]:`	0	0	2	0	2
`pygame.draw.circle(screen,'white',(x_prime,y_prime))`	0	0	2	1	3
`if steps == step_size:`	0	0	2	0	2
`horizontal = not horizontal`	0	0	1	1	2
`switch_counter += 1`	0	0	1	1	2
`step_size += 1`	0	0	1	1	2
`inc = 8 if step_size%2 == 1 else -8`	0	0	1	1	2

3.2.5 Dataset Loading and Preprocessing

The dataset intended for fine-tuning is loaded using Pandas' `read_csv` function, which reads `cosmic_dataset.csv` containing the columns `code`, `E`, `X`, `R`, `W` and the pre-computed total `CFP`. To prevent any target leakage, the CFP column is immediately removed. The cleaned DataFrame is then converted into a Hugging Face `Dataset` using Dataset.from_pandas(), and shuffled with a fixed seed to eliminate any ordering bias—ensuring that lines from different languages or projects are uniformly mixed. Finally, an 80/20 split allocates the majority of examples to training and reserves the remainder for evaluation. Figure 3.7 illustrates the complete data loading, cleaning, shuffling, and splitting pipeline.

```
# 1. Load CSV into a Hugging Face Dataset
df = pd.read_csv("cosmic_dataset.csv")  # expects columns: code, E, X, R, W, CFP
# Drop the CFP column since we predict E, X, R, W and recompute CFP
if "CFP" in df.columns:
    df = df.drop(columns=["CFP"])
dataset = Dataset.from_pandas(df)

# 2. Split 80/20 for train/test
dataset = dataset.train_test_split(test_size=0.2, seed=42)
train_ds = dataset["train"]
eval_ds  = dataset["test"]
```

Figure 3.7: Loading, cleaning, shuffling, and splitting the dataset

3.2.6 Dataset Utilization for Model Training

Upon loading and preprocessing the dataset, it is divided into two distinct subsets: the training dataset comprises 80% of the data, while the testing dataset consists of the remaining 20%. This partitioning ensures that the model is exposed to the majority

of the data during the learning phase while retaining a fair portion of unseen data for evaluation.

The training dataset acts as the foundational input for the model's fine-tuning process. It allows the CodeBERT model to analyze thousands of lines of code and learn the relationships between programming statements and their corresponding COSMIC movement counts (Entry, Exit, Read, and Write). Through iterative training, the model internalizes these mappings, refining its ability to make accurate multi-output regression predictions.

In contrast, the testing dataset remains entirely isolated from the training loop. It includes previously unseen code lines that serve as a benchmark for assessing the model's generalization capability. By evaluating the model's predicted movement counts on this fresh subset and comparing them against the true values, we obtain a reliable measure of the model's performance outside the training distribution.

The fundamental distinction between the training and testing datasets lies in their purpose: the training dataset is utilized to train and optimize the model's parameters, while the testing dataset is employed to evaluate the effectiveness and reliability of the trained model on novel data.

3.3 CodeBERT Fine-Tuning Process

3.3.1 Introduction to CodeBERT

CodeBERT is a pre-trained transformer model developed by Microsoft to support both programming and natural languages [10]. Based on the RoBERTa architecture, CodeBERT extends the capabilities of traditional language models by learning from bimodal data pairs—natural language comments and their corresponding code implementations. This makes it especially suitable for tasks such as code search, summarization, classification, and, as explored in this thesis, regression-based software size estimation.

In this research, CodeBERT is fine-tuned to predict COSMIC Functional Size Measurement (FSM) components—Entry (E), Exit (X), Read (R), and Write (W)—at the line level. Instead of a classification problem, the task is formulated as a multi-output regression problem, where the model learns to assign a real-valued estimate for each of the four movement types per code line.

Why CodeBERT? Traditional FSM automation tools often rely on rule-based matching, custom parsers, or language-specific frameworks [24, 23]. These approaches are difficult to scale across different languages and offer limited generalization. In contrast, CodeBERT has demonstrated strong transfer learning capabilities and high accuracy in FSM-related tasks, as shown in recent studies [30]. Tenekeci et al. used CodeBERT to predict COSMIC data movements directly from Python and C# functions, achieving up to 84.5% accuracy and normalized mean absolute error (NMAE) as low as 0.13.

3.3.2 Fine-Tuning Process

Fine-tuning adapts the pre-trained CodeBERT model to our specific task by training it on our labeled dataset. This process involves initializing the model and tokenizer, preprocessing the data, defining evaluation metrics, configuring training parameters, training the model, and evaluating its performance. Each step is designed to ensure accurate and automated functional size measurement across diverse programming languages.

Model and Tokenizer Initialization

I initialize the CodeBERT model using the `microsoft/codebert-base` checkpoint, configured for sequence classification with four regression outputs corresponding to the movement types E, X, R, and W. This setup employs mean squared error (MSE) as the loss function to optimize predictions for continuous movement counts, typically ranging from 0 to 3. The tokenizer, also loaded from the same checkpoint, converts code lines into token sequences compatible with the model. This configuration leverages CodeBERT's pre-trained understanding of code syntax and semantics, as described in [10].

```
# 3. Initialize CodeBERT tokenizer & model (regression head)
MODEL_NAME = "microsoft/codebert-base"
tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)
model = AutoModelForSequenceClassification.from_pretrained(
    MODEL_NAME,
    num_labels=4,
    problem_type="regression"
)
```

Figure 3.8: CodeBERT model with regression head for predicting COSMIC movement counts

Figure 3.8 illustrates the CodeBERT architecture, where the transformer encoder processes input tokens, and a regression head outputs four real-valued predictions for the COSMIC movements.

Preprocessing the Data

Each code line is tokenized using the CodeBERT tokenizer, which converts it into a sequence of token IDs suitable for the model. Truncation is applied to ensure sequences do not exceed the model's maximum input length (typically 512 tokens). The movement counts (E, X, R, W) are organized into a 2D array of 32-bit floating-point numbers,

serving as labels for the multi-output regression task. This preprocessing step ensures that the data is in a format the model can process effectively. To handle sequences of varying lengths, we use `DataCollatorWithPadding` from the Hugging Face Transformers library [13]. This collator dynamically pads tokenized sequences to the longest length within each batch, ensuring efficient processing during training and evaluation without requiring fixed-length padding across the entire dataset. Figure 3.9 demonstrates how a code line is tokenized and paired with its corresponding movement labels.

```python
# 4. Preprocessing: tokenize and attach float32 labels
def preprocess(examples):  2 usages
    toks = tokenizer(examples["code"], truncation=True, padding=False)
    import numpy as np
    labels = np.vstack([
        examples["E"],
        examples["X"],
        examples["R"],
        examples["W"],
    ]).T.astype(np.float32)
    toks["labels"] = labels.tolist()
    return toks

train_ds = train_ds.map(preprocess, batched=True)
eval_ds  = eval_ds.map(preprocess, batched=True)

# 5. Dynamic padding collator
data_collator = DataCollatorWithPadding(tokenizer)
```

Figure 3.9: Example of tokenization and label preparation for a code line

Defining Evaluation Metrics

To assess the model's performance, we define a comprehensive set of evaluation metrics: Mean Squared Error (MSE), Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), R-squared (R^2). These metrics are calculated for each movement type (E, X, R, W) and for the total COSMIC Function Points (CFP), which is the sum of the four movements. Table 3.6 provides detailed descriptions of each evaluation metric used to assess model performance. The implementation of these evaluation metrics is shown in Figure 3.10.

```python
def compute_metrics(pred):  1 usage
    preds, labels = pred
    # preds and labels: shape (n_samples, 4)

    # Per-dimension MSE, RMSE, MAE, R2, Pearson r
    mse_dims  = mean_squared_error(labels, preds, multioutput="raw_values")
    rmse_dims = np.sqrt(mse_dims)
    mae_dims  = mean_absolute_error(labels, preds, multioutput="raw_values")
    r2_dims   = [r2_score(labels[:, i], preds[:, i]) for i in range(4)]
    corr_dims = [pearsonr(labels[:, i], preds[:, i])[0] for i in range(4)]

    # Summed CFP metrics
    sum_preds  = np.sum(preds, axis=1)
    sum_labels = np.sum(labels, axis=1)
    mse_sum    = mean_squared_error(sum_labels, sum_preds)
    rmse_sum   = np.sqrt(mse_sum)
    mae_sum    = mean_absolute_error(sum_labels, sum_preds)
    r2_sum     = r2_score(sum_labels, sum_preds)
    corr_sum   = pearsonr(sum_labels, sum_preds)[0]

    return {
        # Entry
        "mse_E":   mse_dims[0], "rmse_E": rmse_dims[0], "mae_E": mae_dims[0],
        "r2_E":    r2_dims[0],  "corr_E": corr_dims[0],
        # Exit
        "mse_X":   mse_dims[1], "rmse_X": rmse_dims[1], "mae_X": mae_dims[1],
        "r2_X":    r2_dims[1],  "corr_X": corr_dims[1],
        # Read
        "mse_R":   mse_dims[2], "rmse_R": rmse_dims[2], "mae_R": mae_dims[2],
        "r2_R":    r2_dims[2],  "corr_R": corr_dims[2],
        # Write
        "mse_W":   mse_dims[3], "rmse_W": rmse_dims[3], "mae_W": mae_dims[3],
        "r2_W":    r2_dims[3],  "corr_W": corr_dims[3],
        # Total CFP
        "mse_CFP":  mse_sum,    "rmse_CFP": rmse_sum,  "mae_CFP": mae_sum,
        "r2_CFP":   r2_sum,     "corr_CFP": corr_sum
    }
```

Figure 3.10: Code snippet for defining evaluation metrics

Table 3.6: Evaluation Metrics and Their Descriptions

Metric	Description
Mean Squared Error (MSE)	Average of squared differences between predicted and actual values.
Root Mean Squared Error (RMSE)	Square root of MSE, indicating error magnitude in original units.
Mean Absolute Error (MAE)	Average of absolute differences between predicted and actual values.
R-squared (R^2)	Proportion of variance in the dependent variable explained by the model.

Setting Up Training Arguments

The training process is configured using `TrainingArguments` from the Hugging Face Transformers library [13]. Key parameters include the output directory (`codebert-cfp`), evaluation strategy (per epoch), batch sizes (8 for both training and evaluation), number of epochs (5), learning rate (5×10^{-5}), save strategy (per epoch), and the metric for selecting the best model (MSE of total CFP). The learning rate is chosen based on standard practices for fine-tuning transformer models, balancing convergence speed and stability. The batch size of 8 optimizes memory usage and training efficiency, while five epochs allow sufficient learning without overfitting, based on typical fine-tuning protocols. Figure 3.11 shows the complete configuration of these training arguments.

Training the Model

The `Trainer` class from the Hugging Face Transformers library manages the training and evaluation process, utilizing the AdamW optimizer (default for regression tasks) to update model parameters. The model is trained over five epochs, with periodic evaluation on the validation set to monitor performance. The best-performing model, determined by the lowest MSE for total CFP on the validation set, is saved to the directory `codebert-cfp/best-model`. This approach ensures that the model learns to accurately predict COSMIC movements while avoiding overfitting. Figure 3.12 demonstrates the initialization and execution of the training process.

3.4 Graphical User Interface

The Graphical User Interface (GUI) for this research is implemented as a web application named COSMIC-AI. It provides an accessible and efficient platform for analyzing source code using the fine-tuned CodeBERT model described in Section 3.3. Developed using the Flask web framework [2], COSMIC-AI enables users to upload code files and receive

```
training_args = TrainingArguments(
    output_dir="codebert-cfp",
    eval_strategy="epoch",
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    num_train_epochs=5,
    learning_rate=5e-5,
    save_strategy="epoch",
    save_total_limit=2,
    load_best_model_at_end=True,
    metric_for_best_model="mse_CFP",
    greater_is_better=False
)
```

Figure 3.11: Code snippet for configuring training arguments

```
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_ds,
    eval_dataset=eval_ds,
    tokenizer=tokenizer,
    data_collator=data_collator,
    compute_metrics=compute_metrics
)

train_result = trainer.train()
trainer.save_model("codebert-cfp/best-model")
```

Figure 3.12: Code snippet for initializing and running the Trainer

COSMIC Functional Size Measurement (FSM) predictions, along with summary statistics. This section details the interface's purpose, architecture, features, user workflow, and implementation.

3.4.1 Purpose and Design

COSMIC-AI addresses the complexity of manual FSM analysis [7] by providing a user-friendly system that leverages CodeBERT's capabilities to predict software functional movements (Entry, Exit, Read, Write) line by line. The web-based interface ensures platform independence and removes the need for local setup or machine learning expertise.

Key design principles include:

- **Input Simplicity**: Users can upload code files.
- **Clear Output**: Predictions are displayed in a results table with summary statistics (total lines, total CFP, average CFP, maximum CFP).
- **Error Resilience**: Invalid inputs and processing errors are caught and reported clearly.

3.4.2 System Architecture

COSMIC-AI follows a client-server architecture using Flask [2]. The backend is written in Python and performs code analysis by integrating the fine-tuned CodeBERT model via the Hugging Face Transformers library [13]. The front-end uses HTML templates, CSS, and JavaScript for layout and asynchronous communication via RESTful Application Programming Interface (API) endpoints. Figure 3.13 shows the homepage of the COSMIC-AI web application.

Figure 3.13: Homepage the COSMIC-AI web app

The user interface provides an intuitive file upload mechanism and displays prediction results in a structured format. Figure 3.14 illustrates the main interface of the COSMIC-AI web application, highlighting the file upload functionality and results display area.

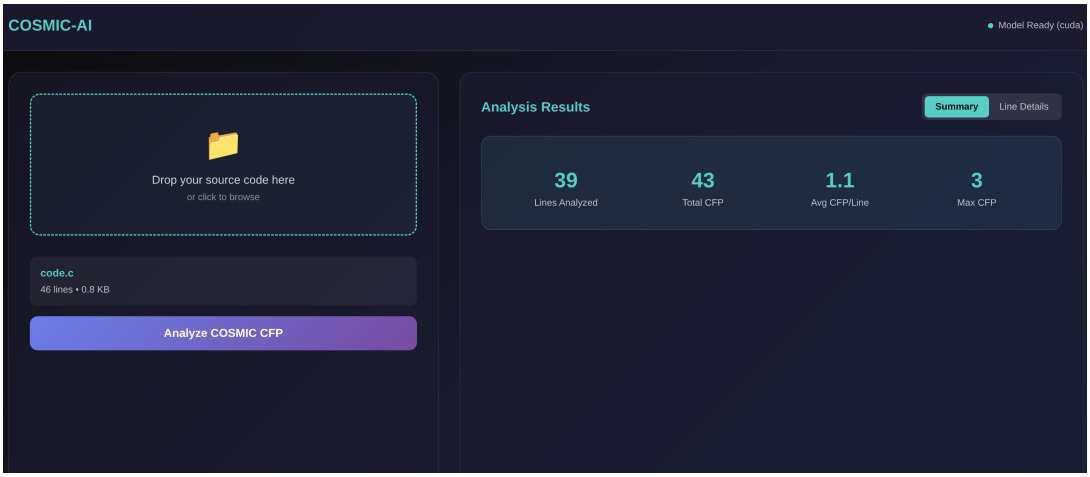

Figure 3.14: Interface of the COSMIC-AI web app

The application provides detailed line-by-line analysis results, showing individual COSMIC movement predictions for each code line. Figure 3.15 presents the detailed line-by-line COSMIC size breakdown that users receive after uploading their code.

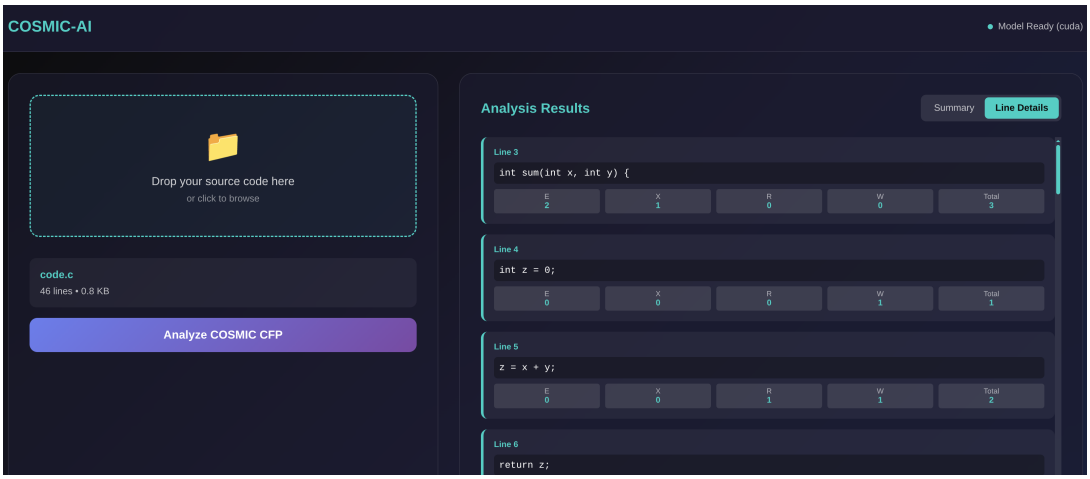

Figure 3.15: Line-by-line COSMIC size details

The backend operations include:

- **Model Initialization**: Loads the fine-tuned CodeBERT model and tokenizer from `codebert-cfp/best-model` using PyTorch and Hugging Face Transformers.
- **Code Filtering**: Filters comments and empty lines before processing.
- **Prediction Pipeline**: Tokenizes each code line, runs it through the CodeBERT model to get movement scores (Entry, Exit, Read, Write), rounds each predicted score to the nearest integer, then sums them to compute the line's total CFP.
- **Error Logging**: Implements robust logging and exception handling for traceability.

3.4.3 Key Functionalities

COSMIC-AI provides several REST API endpoints that manage core interactions, as detailed in Table 3.7.

Table 3.7: COSMIC-AI API Endpoints

Endpoint	Description
`GET /`	Serves the main HTML page.
`GET /health`	Returns model and device status.
`POST /analyze`	Accepts uploaded code and returns line-level predictions with summary.
`POST /model/reload`	Reloads the model (for updates or development).

The core endpoint, `/analyze`, handles user-submitted code by splitting input into lines, filtering comments and blank lines, tokenizing valid lines, generating movement

predictions, and returning structured results with error handling. Figure 3.16 shows the implementation of the analyze endpoint that processes user-submitted code.

```python
@app.route( rule: '/analyze', methods=['POST'])
def analyze():
    # Analyze code and return COSMIC CFP predictions
    try:
        if model is None:
            return jsonify({'success': False,'error':'Model not loaded.'}), 500
        data = request.get_json() or {}
        code = data.get('code','').strip()
        if not code:
            return jsonify({'success': False,'error':'No code provided.'}), 400

        return jsonify(process_code(code))
    except Exception as e:
        logger.error(f"analyze error: {e}\n{traceback.format_exc()}")
        return jsonify({'success': False,'error':f'Server error: {e}'}), 500
```

Figure 3.16: The analyze endpoint

3.4.4 User Interaction Flow

The typical user workflow is as follows:

1. Access the interface at `/`.
2. Upload a source code file.
3. Submit for analysis via the `/analyze` endpoint.
4. View results in the table and summary statistics.
5. Address errors via on-screen messages if applicable.

This streamlined process supports users with minimal FSM or AI expertise.

3.4.5 Implementation Details

The backend uses Flask [2] for routing and API services, PyTorch for inference, NumPy for calculations, and the Transformers library [13] for model and tokenizer integration. Logging is implemented using Python's standard `logging` module for tracking issues. The

system initializes via the `initialize_app` function, which loads the model and device state on startup.

The application runs on port 5000 by default and supports environment-variable configuration for deployment. The `/model/reload` endpoint enables hot-swapping the model during development without restarting the server. Figure 3.17 demonstrates the initialize_app function that loads the model and sets up the application state on startup.

```
def initialize_app():  1 usage
    logger.info("Initializing CosmicScope application...")

    # Try to load the model
    model_dir = os.getenv('MODEL_DIR', 'codebert-cfp/best-model')
    success = load_model(model_dir)

    if not success:
        logger.warning("Failed to load model on startup. The /model/reload endpoint can be used to load it later.")

    logger.info("Application initialized")
```

Figure 3.17: The intialize_app function

Chapter 4

Results

This chapter presents the results obtained from fine-tuning a CodeBERT model to predict COSMIC Function Points (CFP) directly from code snippets. The primary goal of this study was to estimate the data movements defined in the COSMIC methodology — Entry (E), Exit (X), Read (R), and Write (W) — and evaluate the model based on its ability to reconstruct the total CFP count. To this end, the model was trained with a regression head, optimizing for the lowest mean squared error of the total CFP (`mse_CFP`).

4.1 Evaluation on the Test Set

The fine-tuned CodeBERT model was evaluated on a held-out test set to assess its performance in predicting COSMIC movements and total CFP. The evaluation metrics include Mean Squared Error (MSE), Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), and R-squared (R^2), calculated for each movement type (E, X, R, W) and the total CFP, which is the sum of the movements. These metrics were chosen to quantify prediction accuracy and the proportion of variance explained by the model, with a focus on `mse_CFP` as the primary optimization target during training.

Table 4.1 summarizes the key evaluation metrics for the test set after five epochs of training with a batch size of 8, optimized to minimize `mse_CFP`.

Table 4.1: Test Set Evaluation Metrics for CodeBERT Model

Metric	**E**	**X**	**R**	**W**	**CFP**
MSE	0.019	0.012	0.029	0.019	0.095
RMSE	0.138	0.110	0.171	0.140	0.309
MAE	0.034	0.048	0.069	0.056	0.138
R^2	0.866	0.915	0.917	0.895	0.887

The model achieved a low `mse_CFP` of 0.095, indicating accurate prediction of total CFP with minimal squared error. The corresponding `rmse_CFP` of 0.309 suggests that, on

average, the model's CFP predictions deviate by approximately 0.31 function points from the actual values, which is acceptable given the typical range of CFP values (0 to 10 per line). The `mae_CFP` of 0.138 further confirms that absolute errors are small, enhancing the model's reliability for practical software sizing.

Notably, both MAE and RMSE for CFP predictions are well below 0.5, which has important practical implications for the COSMIC sizing process. Since CFP values are rounded to the nearest integer for final reporting, prediction errors of less than 0.5 function points generally result in correct rounded values. This means that for the majority of predictions (as indicated by the MAE), the model's continuous output will round to the correct integer CFP value, effectively achieving perfect accuracy in the final sizing result despite minor numerical deviations. While the RMSE being higher than MAE suggests some predictions have larger errors that may affect rounding accuracy, the overall performance indicates robust practical utility for automated COSMIC measurement.

An R^2 value of 0.887 for CFP indicates that the model explains approximately 88.7% of the variance in total CFP, demonstrating strong predictive power.

For individual movements, MSE values range from 0.012 (X) to 0.029 (R), with RMSE values between 0.110 (X) and 0.171 (R), reflecting precise predictions for each movement type. The lower MSE for Exit (X) suggests the model is particularly effective at predicting this movement, possibly due to distinct syntactic patterns in the dataset. MAE values, ranging from 0.034 (E) to 0.069 (R), indicate small absolute errors across movements, while R^2 values above 0.866 for all movements confirm the model's ability to capture most of the variance in individual predictions.

These results validate the effectiveness of optimizing for `mse_CFP`, as the low error metrics and high R^2 values demonstrate robust performance across both total CFP and individual movements. The evaluation runtime of 6.81 seconds for 446 samples (65.45 samples per second) indicates efficient inference, suitable for integration into the COSMIC-AI web application (Section 3.4). Overall, the model's performance supports its practical utility for automated COSMIC FSM in software engineering applications.

4.2 Functional Sizing Measurement Performance

To validate the practical performance of the CodeBERT-based COSMIC measurement system, I evaluated it against two real-world code examples that were previously measured manually using the COSMIC ISO 19761 method in published research: one C program and one Arduino code. This evaluation provides insight into how the automated system performs compared to expert human measurement on complete programs rather than isolated code snippets.

4.2.1 Case Study 1: C Program COSMIC Measurement

The first evaluation case comes from Koulla et al. [20], who presented a C program with comprehensive manual COSMIC measurement. The manual measurement was performed

by three independent evaluators on the code shown in figure 4.1, including one certified COSMIC measurer, ensuring high reliability of the ground truth. The consensus manual measurement yielded a total of 45 COSMIC Function Points (CFP) for the complete program.

```c
#include <stdio.h>

int sum(int x, int y) {
    int z = 0;
    z = x + y;
    return z;
}
void checkEvenOrOdd(int x) {
    if (x % 2 == 0)
        printf("%d is Even\n", x);
    else
        printf("%d is odd\n", x);
}
int square(int x) {
    return x * x;
}
int maximum(int x, int y) {
    int max = 0;

    if (x < y)
        max = y;
    else
        max = x;

    return max;
}

int main() {
    int x = 0, y = 0;
    printf("Give two integers numbers\n");
    scanf("%d%d", &x, &y);
    int z = 0, result;
    z = x + 2;
    result = z;
    int tmp = square(z);
    int su = sum(x, y);
    int max = maximum(x, y);
    result += 2;
    checkEvenOrOdd(result);
    printf("The sum of %d and %d is : %d\n", x, y, su);
    printf("The square of %d is : %d\n", z, tmp);
    printf("The maximum of %d and %d is : %d\n", x, y, max);

    return 0;
}
```

Figure 4.1: C program from Koulla et al. (2022) used for COSMIC measurement validation

Figure 4.2 shows the original sizing result from the study, while Figure 4.3 demonstrates the output from the COSMIC-AI web application for the same code.

CFP manual measurement results

N° of lines of code	COSMIC data movements	Value in CFP
3	1X, 2E	3
4	1W	1
5	2R, 1W	3
6	1R	1
8	1E	1
9	1R	1
10	1W	1
12	1W	1
14	1E, 1X	2
15	2R	2
17	1X, 2E	3
18	1W	1
20	2R	2
21	1R, 1W	2
23	1R, 1W (optional)	2
25	1R	1
28	1X	1
29	2W	2
30	1W	1
31	1R	1
32	1W	1
33	1R, 1W	2
34	1R, 1W	2
35	1W	1
36	1W	1
37	1W	1
38	1R, 1W	2
40	1W	1
41	1W	1
42	1W	1
30	**Total entries: 06; Total exits: 04; Total reads:15; Total writes: 20**	**Total CFP: 45**

Figure 4.2: Sizing of C program from Koulla et al. (2022)

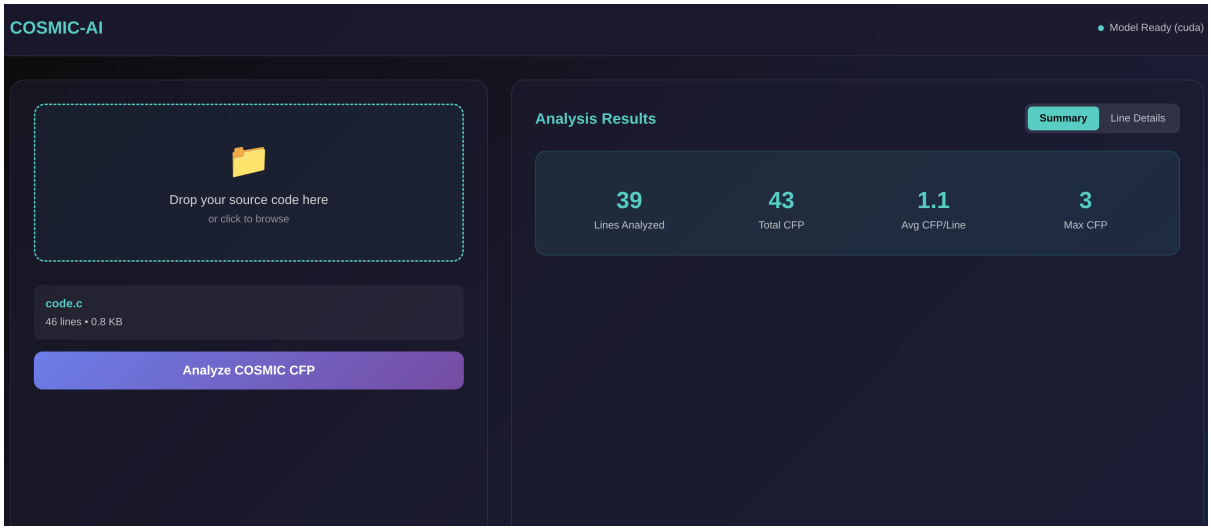

Figure 4.3: COSMIC-AI web application output for the Koulla et al. program

My automated system predicted 43 CFP for this program, achieving approximately 96% accuracy compared to the manual measurement. This represents a deviation of only 2 function points from the expert consensus, demonstrating strong alignment between automated and manual COSMIC measurement for complex, real-world code.

4.2.2 Case Study 2: Arduino Code Functional Size Measurement

The second validation case is derived from Soubra and Abran [26], who presented an Internet of Things (IoT) application example using the Arduino open-source platform. This study provided another independently verified COSMIC measurement for Arduino code shown in figure 4.4, focusing on embedded systems functionality. The manual measurement established a total of 17 CFP for the Arduino program.

Figure 4.6 shows the original sizing result from the study, while Figure 4.5 demonstrates the output from the COSMIC-AI web application for the same code.

```
#include <LiquidCrystal.h>

LiquidCrystal lcd(12, 11, 5, 4, 3, 2);

int sensorVal=0;

void setup() {
  Serial.begin(9600);
  lcd.begin(16, 2);
}

void loop() {
  int sensorVal = analogRead(A0);

  Serial.print("sensor Value: ");
  Serial.print(sensorVal);

  float voltage = (sensorVal / 1024.0) * 5.0;

  Serial.print(", Volts: ");
  Serial.print(voltage);

  float temperature = (voltage - .5) * 100;

  Serial.print(", degrees C: ");
  Serial.println(temperature);

  lcd.setCursor(0, 0);
  lcd.print("Hello");

  lcd.setCursor(0, 1);
  lcd.print("Paris Temp: ");
  lcd.print(temperature);

  delay(1000);
}
```

Figure 4.4: Arduino code from Soubra and Abran (2017) used for COSMIC measurement validation

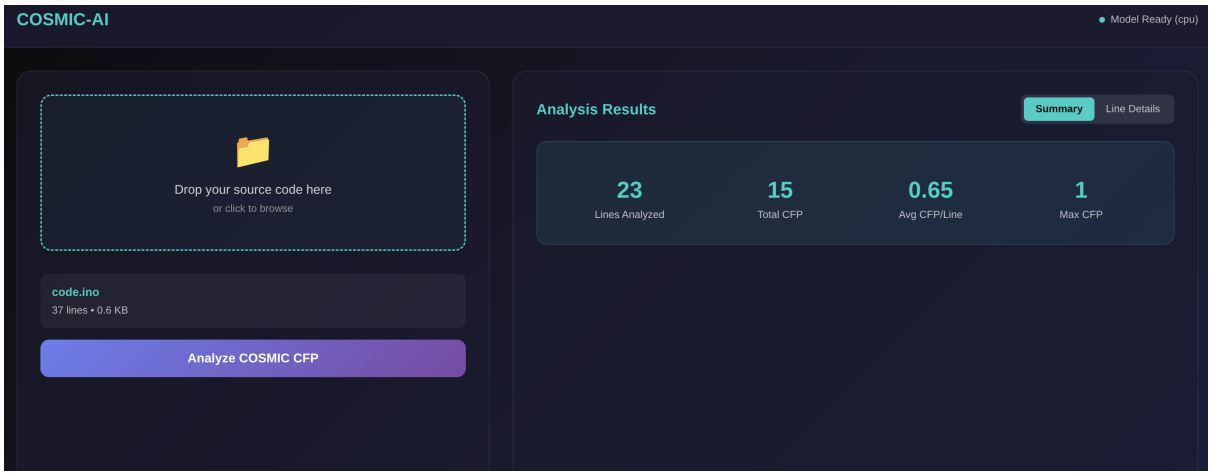

Figure 4.5: COSMIC-AI web application output for the Arduino program

Functional process	Data group movements identified	Size
setup()	• 1E: Trigger (power on) • 1X: serial.begin(9600); • 1X: *lcd.begin(16, 2);*	3 CFP
loop()	• 1E: Trigger (setup complete) • 1E: *analogRead(A0);* • 1X: *Serial.print("sensor Value: ");* • 1X: *Serial.print(sensorVal);* • 1X: *Serial.print(", Volts: ");* • 1X: *Serial.print(voltage);* • 1X: *Serial.print(", degrees C: ");* • 1X: *Serial.println(temperature);* • 1X: *lcd.setCursor(0, 0);* • 1X: *lcd.print("Hello");* • 1X: *lcd.setCursor(0, 1);* • 1X: *lcd.print("Paris Temp: ");* • 1X: *lcd.print(temperature);* • 1X: *delay(1000);*	14 CFP

Figure 4.6: Sizing Arduino program from Soubra and Abran (2017)

My automated system predicted 15 CFP for this Arduino code, achieving approximately 88% accuracy compared to the manual measurement. This represents a deviation of 2 function points from the expert measurement, indicating good performance on embedded systems code despite the different programming context.

4.2.3 Performance Analysis

The validation results demonstrate promising performance across different code types and complexity levels. The C program case study achieved 96% accuracy with only a 2 CFP deviation from expert consensus, while the Arduino code case study achieved 88% accuracy with the same absolute deviation. These results suggest that the automated system performs consistently well across different programming contexts, from traditional C programs to embedded IoT applications.

The slightly lower accuracy for the Arduino code may reflect the different programming patterns and hardware interaction patterns typical in embedded systems, which may be less represented in the training data. Nevertheless, both case studies demonstrate that the automated COSMIC measurement system produces results that are practically useful and closely aligned with expert human measurement, validating its potential for real-world software sizing applications.

Chapter 5

Conclusion and Future Work

5.1 Summary of Contributions

This research has successfully demonstrated the feasibility of automating COSMIC Functional Size Measurement using deep learning approaches, specifically through fine-tuning the CodeBERT transformer model. The primary contributions of this work include:

- **Automated COSMIC FSM System:** Development of a CodeBERT-based regression model capable of predicting COSMIC data movements (E, X, R, W) at the line level across multiple programming languages, achieving low prediction errors with MSE of 0.095 for total CFP and R^2 of 0.887.
- **Multi-language Dataset Creation:** Construction of a comprehensive dataset encompassing Java, Python, C, and Arduino code with line-level COSMIC annotations, supporting cross-language functional size measurement research.
- **COSMICAnalyzer Tool:** Integration and enhancement of existing regex-based measurement approaches from Koulla et al. [20] and Soubra & Abran [26] into a unified, automated tool for C and Arduino code analysis, facilitating consistent dataset generation.
- **COSMIC-AI Web Application:** Implementation of a user-friendly web interface that makes automated COSMIC measurement accessible to practitioners without machine learning expertise, demonstrating practical deployment of the research outcomes.
- **Validation Against Expert Measurements:** Empirical validation showing 96% accuracy on C programs and 88% accuracy on Arduino code when compared to published expert measurements, confirming the practical utility of the automated approach.

The results demonstrate that machine learning-based automation can achieve accuracy levels comparable to expert manual measurement while significantly reducing the time and effort required for COSMIC FSM.

5.2 Limitations and Threats to Validity

While the research demonstrates promising results, several limitations must be acknowledged:

- **Ground Truth Reliability:** The manual COSMIC sizing performed for Java and Python programs, while following established guidelines, is inherently subjective and prone to human error. This represents a potential threat to the validity of the training data and, consequently, the model's learned representations.
- **Dataset Size and Diversity:** The current dataset, while spanning four programming languages, is relatively limited in scale compared to large-scale software engineering datasets. This may affect the model's generalization capability to more complex or domain-specific applications.
- **Language Coverage:** The focus on four programming languages (Java, Python, C, Arduino) may not fully represent the diversity of modern software development, particularly emerging languages and frameworks.
- **Evaluation Scope:** Validation was conducted on relatively small programs, and the performance on large-scale enterprise applications remains to be demonstrated.

5.3 Future Work

Several research directions emerge from this work that could enhance the automation and reliability of COSMIC FSM:

- **Reliable Multi-language Dataset Development:** Future research should prioritize the creation of a large-scale, expertly validated dataset covering a broader range of programming languages and software domains. This dataset should involve multiple certified COSMIC measurers to establish more reliable ground truth annotations.
- **COSMICAnalyzer Extension:** The COSMICAnalyzer tool developed in this research could be extended to support additional programming languages by implementing language-specific regex patterns and measurement rules. Making this tool open-source would benefit the research community and facilitate consistent automated measurement across diverse codebases.
- **Enterprise-Scale Validation:** Conducting validation studies on large-scale commercial software systems would provide stronger evidence of the approach's practical applicability and identify potential scalability challenges.

5.4 Final Remarks

This research represents a significant step toward practical automation of COSMIC Functional Size Measurement, addressing long-standing challenges in software size estimation. The demonstrated accuracy and efficiency of the CodeBERT-based approach, combined with the accessible COSMIC-AI web interface, provide a foundation for broader adoption of automated FSM in software engineering practice.

The convergence of transformer-based language models and software engineering measurement opens new possibilities for intelligent software analysis tools. As the software industry continues to embrace automated development practices, the integration of automated functional size measurement becomes increasingly valuable for project planning, effort estimation, and quality assurance.

While challenges remain, particularly in establishing reliable ground truth and extending coverage to additional languages and domains, this work establishes a promising foundation for future research and development in automated software measurement. The open availability of tools and methodologies developed in this research will hopefully contribute to the continued evolution of evidence-based software engineering practices.

Appendix

Appendix A

Lists

List of Abbreviations

AI Artificial Intelligence

API Application Programming Interface

CFP COSMIC Function Points

COSMIC Common Software Measurement International Consortium

E Entry

FSM Functional Size Measurement

FUR Functional User Requirements

IDE Integrated Development Environment

MAE Mean Absolute Error

ML Machine Learning

MSE Mean Squared Error

NLP Natural Language Processing

R Read

RMSE Root Mean Squared Error

W Write

X Exit

List of Figures

List of Tables

Bibliography

[1] COSMIC Functional Size Measurement Method – Official Documentation. Available at: https://www.cosmic-sizing.org/.

[2] Armin Ronacher et al. Flask: Web development, one drop at a time. https://flask.palletsprojects.com/, 2024. Accessed: 2025-05-26.

[3] Youssef Attallah. Towards a cosmic fsm programming language compiler. In *Proceedings of the International Workshop on Software Measurement (IWSM-Mensura)*, 2022.

[4] Selami Bagriyanik. Automated cosmic function point measurement using a requirements engineering ontology. *Information and Software Technology*, 2016.

[5] Mark Mahrous Besada. Measurement of functional size across programming languages: A machine learning approach. Master's thesis, Bachelor's Thesis, [Your University], May 2024. Supervisors: Dr. Milad Ghantous and Dr. Hassan Soubra.

[6] Nadia Chamkha et al. Automated cosmic measurement of java swing applications throughout their development life cycle. In *Proceedings of the International Workshop on Software Measurement (IWSM-Mensura)*, 2018.

[7] COSMIC Measurement Practices Committee. *COSMIC Measurement Manual for ISO 19761, Version 5.0*, 2021.

[8] Ahmed Darwish. Cosmic functional size of arm assembly programs. In *Proceedings of the International Workshop on Software Measurement (IWSM-Mensura)*, 2020.

[9] C. Dekkers and A. Abran. Functional size measurement in agile and continuous delivery environments: Challenges and opportunities. In *Proceedings of the International Workshop on Software Measurement (IWSM-Mensura)*, pages 45–54, 2018.

[10] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Ming Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Jianfeng Bao, et al. Codebert: A pre-trained model for programming and natural languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings*, pages 1536–1547, 2020.

[11] Mariem Haoues et al. A rapid measurement procedure for sizing web and mobile applications based on cosmic fsm method. In *Proceedings of the International Workshop on Software Measurement (IWSM-Mensura)*, 2017.

[12] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Travis Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with numpy. *Nature*, 585:357–362, 2020.

[13] Hugging Face. Transformers documentation, n.d. Accessed: 2025-05-26.

[14] Ishrar Hussain et al. Approximation of cosmic functional size to support early effort estimation in agile. *Data & Knowledge Engineering*, 2013.

[15] International Organization for Standardization (ISO). *Software Engineering - Mk II Function Point Analysis - Counting Practices Manual*. ISO, Geneva, Switzerland, 1st edition, December 2002.

[16] International Organization for Standardization (ISO). *Software and Systems Engineering - Software Measurement - IFPUG Functional Size Measurement Method*. ISO, Geneva, Switzerland, 2nd edition, 2009.

[17] International Organization for Standardization (ISO). *Information Technology - Systems and Software Engineering - FiSMA 1.1 Functional Size Measurement Method*. ISO, Geneva, Switzerland, 1st edition, 2010.

[18] International Organization for Standardization (ISO). *Software Engineering - COSMIC: A Functional Size Measurement Method*. ISO, Geneva, Switzerland, 2nd edition, 2011. Reviewed and confirmed in 2019.

[19] International Organization for Standardization (ISO). *Software Engineering - NESMA Functional Size Measurement Method - Definitions and Counting Guidelines for the Application of Function Point Analysis*. ISO, Geneva, Switzerland, 2nd edition, 2018.

[20] Donatien Koulla et al. Automated cosmic function points measurement for c program using regular expressions. In *Proceedings of the International Workshop on Software Measurement (IWSM-Mensura)*, 2022.

[21] Wes McKinney. Data structures for statistical computing in python. *Proceedings of the 9th Python in Science Conference*, pages 56–61, 2010.

[22] Aw Ochodek et al. Deep learning model for end-to-end approximation of cosmic functional size based on use-case names. *Information and Software Technology*, 2020.

[23] Abdelaziz Sahab. Cosmic functional size automation of java web applications using the spring mvc framework. In *Proceedings of the International Workshop on Software Measurement (IWSM-Mensura)*, 2020.

[24] M. A. Sağ. Cosmic solver: Automated functional size measurement of java business applications. In *Proceedings of the International Workshop on Software Measurement*, 2020.

[25] H. Sneed. Purpose of software measurement. *Software Measurement News*, 26(1), March 2021.

[26] Hassan Soubra and Alain Abran. Functional size measurement for the internet of things (iot): An example using cosmic and the arduino open-source platform. In *Proceedings of the International Workshop on Software Measurement (IWSM-Mensura)*, Montigny-le-Bretonneux, France & Montreal, Canada, October 2017. ESTACA and ETS.

[27] Hassan Soubra et al. Towards universal cosmic size measurement automation. In *Proceedings of the International Workshop on Software Measurement (IWSM-Mensura)*, 2020.

[28] C. Symons. Why COSMIC functional measurement? *Measurement News*, 25(2), September 2020.

[29] Ayça Tarhan et al. A proposal on requirements for cosmic fsm automation from source code. In *Joint Conference of International Workshop on Software Measurement and International Conference on Software Process and Product Measurement*, 2016.

[30] Samet Tenekeci et al. Predicting software size and effort from code using natural language processing. In *Proceedings of the International Workshop on Software Measurement (IWSM-Mensura)*, 2024.