Software Metrics Measurement Report

Juliette Waltregny

October 2025

1 Introduction

This report presents the results of measuring fundamental software metrics on three large software systems: the Linux kernel, the `java.base` module of the JDK, and the `Modules` directory of ITK. The aim is to estimate and analyze the software's size, complexity, and coupling using standard metrics: Physical Lines of Code (PLOC), Logical Lines of Code (LLOC), Cyclomatic Complexity (CC), Fan-in, and Fan-out. These metrics provide insight into maintainability, modularity, and potential refactoring needs.

2 Measurement Methodology

2.1 Physical LOC (PLOC)

PLOC was implemented by counting every non-empty line in the source code, including comments and documentation. This provides a raw estimate of project size and total code volume.

2.2 Logical LOC (LLOC)

LLOC was implemented by counting non-empty lines while excluding comments, preprocessor directives (e.g., `#include`, `#define`), and documentation lines. This provides a more accurate measure of lines contributing to the program's behavior.

2.3 Cyclomatic Complexity (CC)

Cyclomatic complexity was implemented using McCabe's formula:

$$CC = 1 + \text{number of decision points}$$

The tool scans each function, detects its boundaries (based on indentation in Python or braces in C/Java), and counts decision-making constructs such as `if`, `for`, `while`, `case`, `catch`, as well as logical operators like `&&`, `||`, and ternary `?`. Each function's CC starts at 1 and increases with every branching or decision point encountered. The algorithm ensures that nested functions or new definitions reset complexity counting appropriately.

2.4 Fan-in and Fan-out

The measurement of fan-in and fan-out was done in two passes:

1. **Collect function names:** All function definitions are extracted from the source files using regular expressions.
2. **Analyze calls:** For each function body, all function calls are extracted. Calls that match user-defined functions are considered valid.

Fan-in is computed as the number of distinct functions calling a given function, while fan-out is the number of distinct functions a given function calls. This approach avoids loading the entire codebase into memory, allowing scalability to millions of lines of code.

2.5 Testing Challenges and Scope Limitations

Testing the tool on Python code was challenging due to Python's reliance on indentation and dynamic typing, which complicates function boundary detection and decision counting. As a result, fan-in and fan-out analysis for Python was less reliable.

Due to these difficulties and time constraints, the main analysis focused on C and Java files from the test directories. Results were consistent and meaningful for these languages, while Python results showed inaccuracies.

2.6 Dataset Selection

To ensure meaningful yet computationally feasible results, the following subsets were analyzed:

- **Linux kernel:** `kernel/`, `fs/`, and `mm/` directories.
- **JDK:** `java.base` module.
- **ITK:** `Modules/` directory.

All selected subsets exceed 1 million PLOC, ensuring a comparable scale for analysis across large codebases. For the Linux kernel, only a subset comprising the core directories was analyzed due to time and computational constraints. While this does not cover the entire kernel, it includes its most central components, providing a representative view of overall metric trends and allowing cautious extrapolation of results to the broader codebase.

3 QME of Metrics

3.1 Physical LOC

Physical Lines of Code (PLOC) provides a fundamental measure of software size by counting all non-empty lines in the source code, including comments and documentation. This metric offers a quick, language-agnostic estimate of the overall scale and maintenance effort required for a software project. Its simplicity and broad applicability make it a baseline measure for comparing projects and understanding their evolution over time.

QME	Description
Information need	Estimate the size of a software project
Measurable concept	Size
Relevant entities	Source code module
Attributes	Programming language
Base measures	PLOC
Measurement method	Count any non-empty line in the source code module
Type of measurement method	Objective
Type of scale	Ratio
Unit of measurement	Line
Derived measure	N/A
Measurement function	N/A
Indicator	Product size
Model	N/A
Decision criteria	TBD by stakeholder

Table 1: Physical LOC QME

Logical Lines of Code (LLOC) refines the measurement of software size by focusing solely on lines that contribute directly to program behavior. By excluding comments, documentation, and preprocessor directives, LLOC offers a more accurate picture of the code's functional complexity. This metric is particularly valuable when comparing projects across languages or development styles, as it reflects actual programming effort and logical structure. The motivation behind including LLOC alongside PLOC is to reveal the density of meaningful logic in the codebase and help assess whether a system might be overly complex, under-documented, or growing beyond maintainable limits.

3.2 Logical LOC

QME	Description
Information need	Estimate the size of a software project
Measurable concept	Size
Relevant entities	Source code module
Attributes	Programming language statement
Base measures	LLOC
Measurement method	Count semicolons or logical statements in the source code module
Type of measurement method	Objective
Type of scale	Ratio
Unit of measurement	Line
Derived measure	N/A
Measurement function	N/A
Indicator	Product size
Model	N/A
Decision criteria	If the product size is larger than 1 MLOC, flag it for modularization

Table 2: Logical LOC QME

3.3 Cyclomatic Complexity

Cyclomatic complexity is a key measure of internal code complexity, capturing the number of independent paths through a program's control flow. By counting decision points such as conditionals, loops, and case statements, this metric reflects how difficult the code may be to understand, test, and maintain. Functions with higher complexity are more prone to defects and harder to refactor. The motivation for including CC is twofold: it provides actionable insight into code quality at the function level, and it supports decision-making for refactoring or testing prioritization by highlighting modules where complexity might become problematic.

QME	Description
Information need	Estimate the complexity of a software project
Measurable concept	Complexity
Relevant entities	Source code module
Attributes	Programming language statement
Base measures	Cyclomatic Complexity (CC)
Measurement method	Count number of logical indicators (decision points)
Type of measurement method	Objective
Type of scale	Ratio
Unit of measurement	Number of decision points
Derived measure	Average Cyclomatic Complexity per function/module
Measurement function	$CC = 1 +$ number of decision points
Indicator	High CC indicates higher complexity and lower maintainability
Model	Complexity trends per module; comparison against thresholds
Decision criteria	Review or refactor functions with CC $>$ 10, following McCabe's widely cited guideline for maintainability thresholds.

Table 3: Cyclomatic complexity QME

3.4 Fan-in and fan-out

While this metric is defined as "The fan-in of a module M is the number of local flows that terminate at M, plus the number of data structures from which information is retrieved by M. Similarly, the fan-out of a module M is the number of local flows that emanate from M, plus the number of data structures that are updated by M." by Fenton et al.[1], this implementation considers only control-flow interactions (function calls) when computing fan-in and fan-out. While the original definition also includes data flows, this simplification allows for scalable analysis and still provides meaningful insight into dependency structure.

3.4.1 Fan-in

Fan-in measures how many other modules or functions depend on a given function. High fan-in values indicate widely reused components that are critical to the system's behavior and stability. Monitoring fan-in helps identify key architectural elements whose modification could have far-reaching effects, guiding risk assessment and testing strategies. The motivation for including fan-in is to understand the dependency structure of the software from a reusability and stability

[1]Fenton, Norman, and James Bieman. Software metrics: a rigorous and practical approach. CRC Press, 2014.

perspective and to highlight components that require careful design consideration due to their central role in the system.

QME	Description
Information need	Assess how many other modules or functions depend on a given module/function
Measurable concept	Coupling / dependency strength
Relevant entities	Function or module
Attributes	Number of incoming calls
Base measures	Fan-in
Measurement method	Count how many distinct functions or modules call the entity
Type of measurement method	Objective
Type of scale	Ratio
Unit of measurement	Number of incoming calls
Derived measure	Average fan-in per function or per module
Measurement function	$FI(f) = \sum_{i=1}^{n} calls(f_i, f)$
Indicator	High fan-in indicates that the function is widely used and changes may have large impact
Model	Dependency network model showing coupling between components
Decision criteria	Review functions with fan-in above a threshold for potential design issues

Table 4: QME for Fan-in metric

3.4.2 Fan-out

Similarly to fan-in, fan-out quantifies how many distinct modules or functions a given function depends on. High fan-out can indicate strong coupling and reduced modularity, making code harder to maintain or test. However, it may also reflect the complexity of functionality or deliberate design choices in layered or object-oriented architectures. Including fan-out in the analysis helps reveal potential design smells such as excessive dependencies and informs refactoring decisions aimed at improving modularity and reducing maintenance overhead. When interpreted together with fan-in, it provides a comprehensive picture of the system's dependency dynamics.

QME	Description
Information need	Assess how many other modules or functions a given module/function depends on
Measurable concept	Coupling / dependency strength
Relevant entities	Function or module
Attributes	Number of outgoing calls
Base measures	Fan-out
Measurement method	Count how many distinct functions or modules are called by the entity
Type of measurement method	Objective
Type of scale	Ratio
Unit of measurement	Number of outgoing calls
Derived measure	Average fan-out per function or per module
Measurement function	$FO(f) = \sum_{j=1}^{m} calls(f, f_j)$
Indicator	High fan-out indicates strong coupling and potential maintenance complexity
Model	Dependency graph illustrating outgoing connections from each function
Decision criteria	Refactor or modularize functions with excessive fan-out

Table 5: QME for Fan-out metric

4 Results

4.1 Linux kernel

Metric	**Value**
Total PLOC	1,944,568
Total LLOC	1,089,825
Total Cyclomatic Complexity	172,845
Average Cyclomatic Complexity per function	2.55
Total Fan-out	1,209,442
Average Fan-out per function	17.85
Max Fan-out (most dependent function)	857
Total Fan-in	917,441
Average Fan-in per function	13.54
Max Fan-in (most reused function)	680

Table 6: Aggregate Metrics Summary for Linux kernel subset

4.2 JDK java.base

Metric	Value
Total PLOC	1,121,429
Total LLOC	419,743
Total Cyclomatic Complexity	105,742
Average Cyclomatic Complexity per function	2.32
Total Fan-out	6,758,367
Average Fan-out per function	148.29
Max Fan-out (most dependent function)	2,517
Total Fan-in	4,312,453
Average Fan-in per function	94.63
Max Fan-in (most reused function)	1530

Table 7: Aggregate Metrics Summary for JDK `java.base` module

4.3 ITK Modules

Metric	Value
Total PLOC	1,244,127
Total LLOC	535,102
Total Cyclomatic Complexity	81,569
Average Cyclomatic Complexity per function	2.32
Total Fan-out	1,308,253
Average Fan-out per function	37.26
Max Fan-out (most dependent function)	1,687
Total Fan-in	437,129
Average Fan-in per function	12.45
Max Fan-in (most reused function)	385

Table 8: Aggregate Metrics Summary for ITK `Modules`

5 Visualization

5.1 Comparison of Size (PLOC vs LLOC)

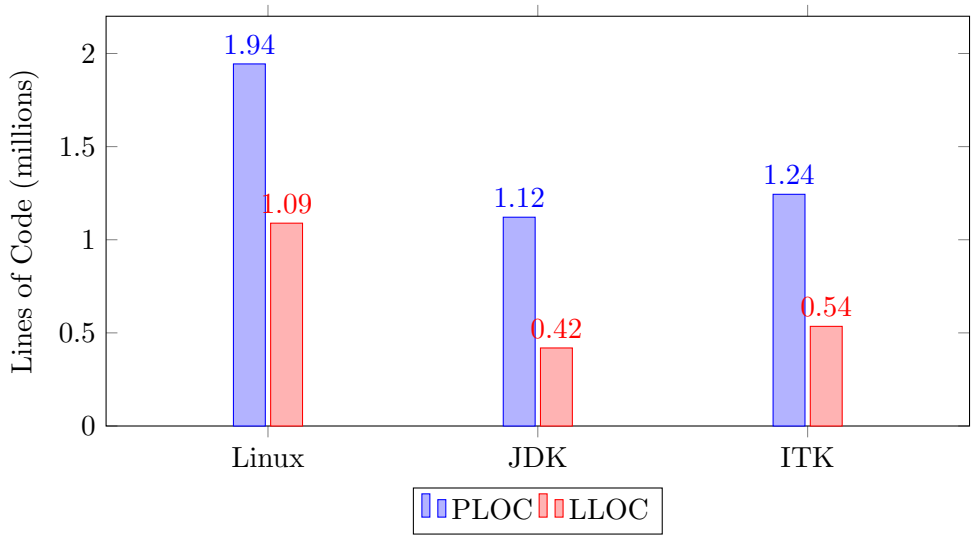

Figure 1: Comparison of total PLOC and LLOC across repositories

Figure 1 shows the comparison of physical lines of code (PLOC) and logical lines of code (LLOC) across the three analyzed repositories. Prior work shows typical comment density in OSS around 19% on average[2], with substantial variation by project and language. Since LLOC excludes comments while PLOC includes non-empty lines, the LLOC/PLOC ratio naturally varies and should not be treated as a universal constant.

The results for the analyzed repositories all fall above this expected range, with LLOC representing approximately 56%, 37% and 43% of the total PLOC, respectively to the graph.

This deviation across all three projects can be attributed to the fact that all three repositories are mature, production-grade systems where code readability and performance often take precedence over extensive inline documentation, resulting in a relatively lower proportion of comment lines and thus a higher LLOC share.

[2]Arafati and Riehle, "The Comment Density of Open Source Software Code," in *Proceedings of the 10th International Symposium on Open Collaboration (OpenSym '14)*, ACM, 2014

5.2 Comparison of Average Fan-in and Fan-out

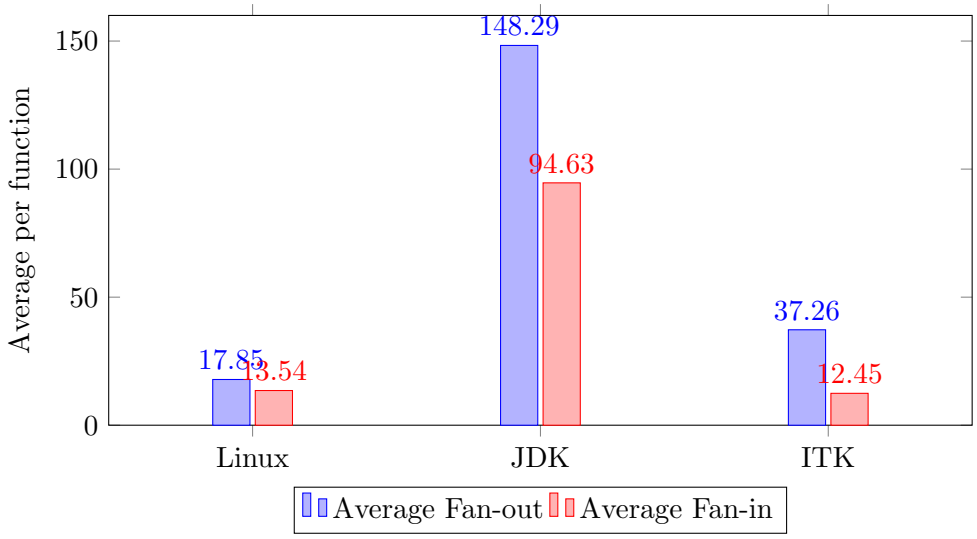

Figure 2: Comparison of average fan-in and fan-out across projects

Figure 2 shows a combined view of average fan-in and fan-out across the three analyzed codebases. The `java.base` module exhibits both significantly higher fan-out (148.29) and fan-in (94.63) compared to the Linux kernel and ITK projects. This reflects the highly interconnected nature of Java's object-oriented design, where functions frequently invoke many others and are themselves widely reused. While this increases functionality and promotes code reuse, it also implies higher coupling and a greater potential for ripple effects when changes are made.

In contrast, the Linux kernel and ITK demonstrate lower fan-in and fan-out values, suggesting simpler dependency structures and more localized interactions. This typically leads to easier maintenance and lower risk of unintended side effects. However, it may also indicate less reuse and fewer opportunities for polymorphic design compared to Java.

Taken together, these metrics highlight a key trade-off between modularity and maintainability: systems with high fan-in and fan-out provide rich functionality but require more careful dependency management, whereas those with lower coupling are easier to evolve but may involve more duplication or limited extensibility.

5.3 Comparison of Cyclomatic Complexity

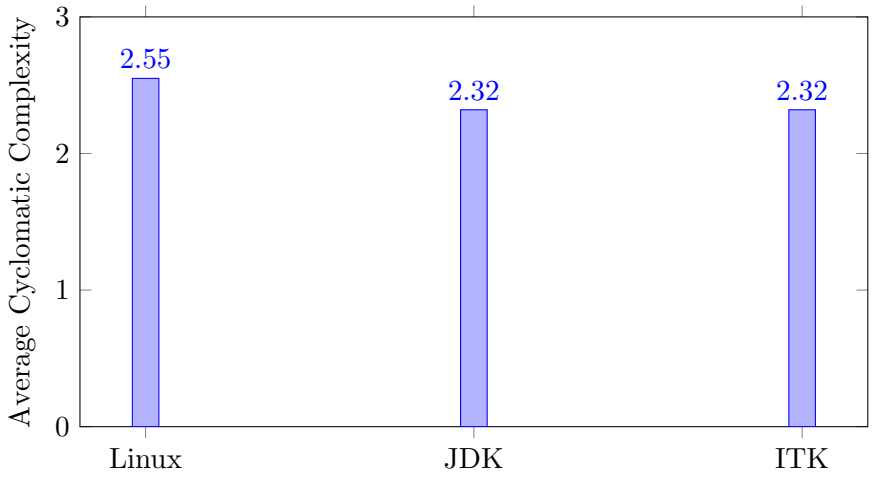

Figure 3: Comparison of average cyclomatic complexity per function

All three projects exhibit low average cyclomatic complexity, ranging from 2.32 to 2.55 per function. These values are well below the commonly cited threshold of 10, above which code is typically considered difficult to maintain or test. This suggests that, despite differences in size, structure, and language, all analyzed projects maintain functions that are relatively simple in terms of control flow.

Low average complexity is often associated with good modularization practices, effective decomposition of functionality, and ease of testing. The slightly higher value in the Linux kernel likely reflects the presence of more low-level logic and conditional branching, characteristic of system-level programming. Overall, the results indicate that all three repositories manage complexity effectively, contributing positively to maintainability and software quality.

6 Analysis and Discussion

The measurements across the three large-scale codebases reveal clear differences in size, complexity, and coupling, reflecting their respective design philosophies and implementation languages.

Size (PLOC and LLOC). All three systems are large, production-grade projects, each exceeding one million physical lines of code (PLOC). The Linux kernel and ITK have similar overall sizes, while the JDK `java.base` module is slightly smaller. The ratio of LLOC to PLOC is considerably higher than the comment density typically reported in open-source projects (15–25%). This suggests that these systems prioritize executable logic and performance over inline documentation and comments. The Linux kernel, in particular, shows a high proportion of LLOC relative to PLOC, reflecting the compact nature of C code, minimal use of comments, and the prevalence of preprocessor directives and low-level constructs.

Complexity (Cyclomatic Complexity). All three repositories show low average cyclomatic complexity (2.32–2.55), well below the commonly accepted threshold of 10. This indicates that most functions remain simple and maintainable despite the overall size and complexity of the projects. The slightly higher value in the Linux kernel likely results from the more intricate control flow logic required in system-level programming. Maintaining low complexity per function is crucial for testability and reduces the risk of introducing defects during evolution.

Coupling (Fan-in and Fan-out). The JDK `java.base` module exhibits a much higher average fan-out (148.29) and fan-in (94.63) than the Linux kernel and ITK. This reflects the object-oriented, highly modular nature of Java, where components frequently interact with many others. While this promotes reuse and rich functionality, it also increases coupling and maintenance complexity, as changes in one part of the system can have widespread effects. The lower fan-in and fan-out in the Linux kernel and ITK suggest more isolated components and simpler dependency structures, which improve maintainability but may limit flexibility and reuse.

Interpretation and Implications. These metrics indicate that all three projects are well-structured in terms of control flow and maintainability. However, the JDK's dense dependency network warrants careful management to avoid ripple effects during changes. According to the QME decision criteria, functions with a cyclomatic complexity greater than 10 or unusually high fan-in/out values should be prioritized for review and potential refactoring. Furthermore, the substantial size of all three codebases highlights the importance of modularization and incremental maintenance to manage long-term system evolution.

7 Conclusion

This project demonstrated how software metrics can provide valuable insights into size, complexity, and coupling in large-scale software systems. Despite limitations in Python analysis, the implemented measurement instruments performed effectively on C and Java code. The results show that while the Linux kernel and ITK are large and moderately coupled, the JDK exhibits high coupling, indicating potential maintenance challenges. Such metrics are valuable for guiding refactoring decisions, improving modularity, and understanding system evolution. Future work could extend this analysis to the full Linux kernel, include data-flow components in fan-in/fan-out measurement, and improve support for dynamically typed languages like Python.