Homework - 2

Software Engineering (COL 740)

Proma Mondal, Yogesh R

`2024MCS2470, 2024MCS2457`

November 14, 2025

Contents

1 Choice of Codebases

1.1 Repository 1: `JsonCpp`

Repository URL: `https://github.com/open-source-parsers/jsoncpp`

Project Description. jsoncpp is a lightweight, open-source C++ library designed for parsing, serializing, and manipulating JSON data. It provides a clean abstraction over JSON types, supports robust error handling, and is widely integrated into configuration systems, networking tools, and general-purpose C++ applications. The project is relatively small, well-structured, and organized into a clear set of source files, making it suitable for fine-grained metric analysis.

Reason for Selection. jsoncpp is small and modular, allowing all source files to be evaluated thoroughly. Its clean structure makes it ideal for illustrating fundamental code complexity metrics such as LOC, Cyclomatic Complexity, Halstead measures, and basic data-flow analysis. Its manageable size also leads to clear and interpretable COCOMO estimations.

1.2 Repository 2: `OGRE`

Repository URL: `https://github.com/OGRECave/ogre`

Project Description. OGRE (Object-Oriented Graphics Rendering Engine) is a large, open-source 3D rendering engine written in C++. It provides an extensive abstraction over graphics APIs such as OpenGL and Direct3D, and is used in simulation, visualization, and real-time rendering applications. The project contains a substantial codebase with multiple modules, including core rendering, scene management, resource handling, and render system implementations. Its architectural complexity and diverse subsystems make it a strong candidate for detailed structural and code-level analysis.

Reason for Selection. OGRE represents a medium-to-large scale industrial C++ project, allowing evaluation of complexity metrics on a realistic, multi-module system. Its size and subsystem diversity make LOC, Cyclomatic Complexity, Halstead metrics, and data-flow characteristics more pronounced, enabling meaningful comparison with smaller projects. The project scale also provides a richer context for COCOMO effort estimation.

1.3 Repository 3: `SQLite`

Repository URL: `https://github.com/sqlite/sqlite`

Project Description. SQLite is a widely-used, lightweight, serverless relational database engine written primarily in C. Unlike traditional database systems, SQLite is delivered as a single library that can be embedded directly into applications. Its core functionality, including query parsing, optimization, storage management, and transaction handling, is implemented in a dense and highly optimized C codebaseoften consolidated into the `sqlite3.c` amalgamation file. The project is known for its focus on reliability, performance, and portability, and is used across operating systems, mobile devices, and embedded systems.

Reason for Selection. SQLite provides a strong contrast to the C++ repositories due to its procedural, single-file-heavy architecture. Its compact yet dense implementation makes it well suited for evaluating LOC, Cyclomatic Complexity, Halstead metrics, and data-flow characteristics within a monolithic design. Its substantial code size also enables realistic COCOMO effort estimation despite being highly optimized and handcrafted.

1.4 Repository 4: `tmux`

Repository URL: `https://github.com/tmux/tmux`

Project Description. tmux is a terminal multiplexer written in C that enables users to create, manage, and control multiple terminal sessions within a single window. It provides features such as session persistence, window and pane management, scripting support, and remote workflows. The codebase is modular, consisting of various command handlers, server components, client interfaces, and utility modules. tmux emphasizes portability and interaction with UNIX-like environments, resulting in a clean and well-organized C code structure.

Reason for Selection. tmux offers a medium-sized, modular C codebase that complements both the small (jsoncpp) and large (OGRE/SQLite) repositories in this study. Its clear separation of modules makes it suitable for analyzing LOC, Cyclomatic Complexity, Halstead metrics, and basic data-flow relationships. The project size and structure also make it appropriate for interpretable COCOMO effort and development-time estimation.

2 Choice of Metrics

2.1 Lines of Code (LOC / SLOC)

Definition. LOC measures the total number of lines in the source files, while SLOC (Source Lines of Code) counts only the executable, non-comment, non-blank lines. SLOC provides a more accurate estimate of actual implementation effort.

Purpose. Used to quantify project size, compare codebases, and serve as the primary input for COCOMO effort and development time estimation.

2.2 Cyclomatic Complexity (CC)

Definition. Cyclomatic Complexity (McCabes $V(G)$) measures the number of linearly independent execution paths in a program. It is computed from the control-flow graph (CFG) of a procedure using the classical formula:

$$V(G) = E - N + 2p,$$

where E is the number of edges, N is the number of nodes, and p is the number of connected components (typically $p = 1$ for a single procedure reachable from `main`).

Purpose. CC indicates the logical complexity of functions and directly correlates with testing effort, since at least $V(G)$ test cases are required to achieve basis path coverage. Industry guidelines recommend $V(G) < 10$, with values above 10–20 often considered challenging and harder to maintain.

2.3 Halstead Metrics and Effort

Definition. Halsteads Software Science quantifies program complexity based on operators and operands. Let N_1 and N_2 be the total number of operators and operands, and η_1 and η_2 the number of unique operators and operands:

$$\eta = \eta_1 + \eta_2 \quad \text{(Program Vocabulary)}, \qquad N = N_1 + N_2 \quad \text{(Program Length)}.$$

Using these, the key Halstead measures are:

$$V = N \times \log_2(\eta) \quad \text{(Program Volume)}$$
$$D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2} \quad \text{(Program Difficulty)}$$
$$E = D \times V \quad \text{(Program Effort)}$$

The estimated number of delivered bugs is:

$$B = \frac{V}{3000}.$$

Purpose. Halsteads metrics model the cognitive effort required to implement or understand code. Volume reflects size, Difficulty captures structural complexity, and Effort (E) estimates the mental work neededhigher values imply greater error-proneness and maintenance cost.

2.4 Data-Flow Complexity

Definition. Oviedos Data-Flow Complexity (DFC) measures the complexity arising from the flow of variable definitions and uses across basic blocks in a program. For each block i, the metric counts all *prior reaching definitions* of variables that are locally exposed in that block. The overall program-level complexity is:

$$DFC = \sum_{i=1}^{n} DF(i),$$

where $DF(i)$ is the number of reaching definitions for block i and n is the total number of basic blocks.

Purpose. DFC captures the interblock data dependencies that influence testing complexity. Higher values imply more intricate definitionuse relationships, greater potential for fault propagation, and increased effort for achieving all-uses test coverage.

2.5 COCOMO Effort and Development Time

Definition. The Constructive Cost Model (COCOMO), proposed by Boehm, estimates the effort and development time required for a software project based solely on its size in thousands of source lines of code (KLOC). In the Basic COCOMO model, effort and time are computed as:

$$E = a_b \times (\text{KLOC})^{b_b}, \qquad D = c_b \times (E)^{d_b},$$

where E is the effort in person-months, D is the development time in months, and the coefficients a_b, b_b, c_b, d_b depend on the project mode (Organic, Semi-detached, Embedded).

Project Modes. COCOMO classifies software development into three modes:

- **Organic:** Small, simple, and well-understood projects developed by experienced teams.
- **Semi-detached:** Medium-sized projects with mixed team experience and moderate complexity.
- **Embedded:** Highly complex, tightly constrained systems often involving hardware, real-time, or safety-critical requirements.

Purpose. COCOMO provides a size-driven estimation of the human effort and calendar time needed to develop a system. It allows comparison of development cost across projects and offers a quantitative basis for staffing and scheduling decisions.

3 Results

3.1 Lines of Code (LOC) and Source Lines of Code (SLOC)

Tabulated Results.

Table 1.1 summarises the total LOC and SLOC for all four projects. LOC includes blank lines and comments, whereas SLOC counts only executable, non-comment lines. SLOC therefore provides a closer approximation to the actual implementation size.

Table 1.1: LOC and SLOC for all projects

Project	**LOC**	**SLOC**	**SLOC/LOC**
JsonCpp	13,740	10,760	0.78
OGRE	199,327	100,530	0.50
SQLite	216,989	107,068	0.49
tmux	61,067	30,115	0.49

LOC vs SLOC Plot.

Figure 1.1 compares LOC and SLOC values for all four projects. JsonCpp is significantly smaller than the other codebases, whereas OGRE and SQLite are the two largest projects in terms of raw source size. SQLite is nearly as large as OGRE, while tmux falls into a smaller intermediate range.

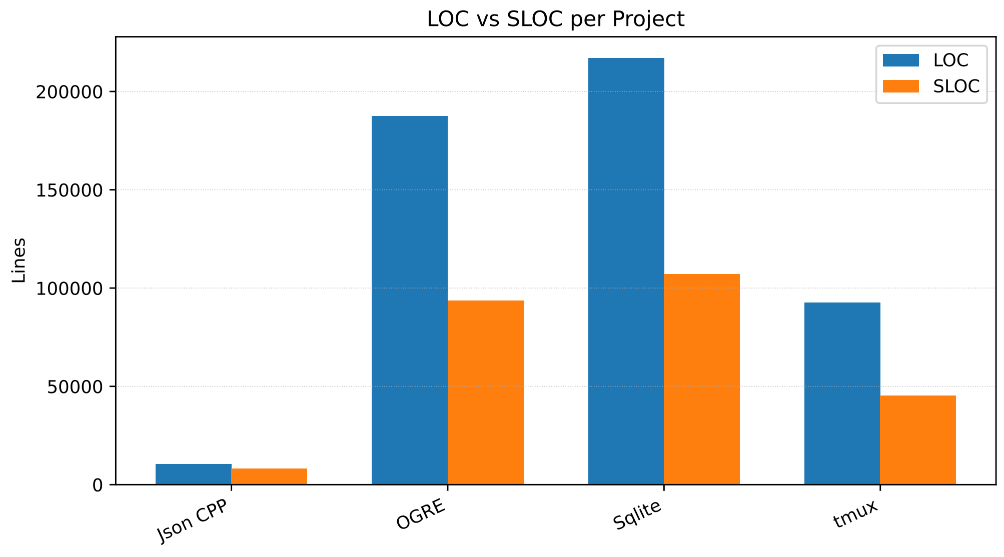


Figure 1.1: LOC vs SLOC per project.

Source Code Density.

Figure 1.2 shows the ratio SLOC/LOC, which reflects the proportion of actual executable code relative to overall file content. JsonCpp has the highest density (0.78), indicating minimal commenting and whitespace, whereas OGRE, SQLite, and tmux all cluster around 0.49–0.50, suggesting richer use of comments, documentation, and structural spacing within the source code.

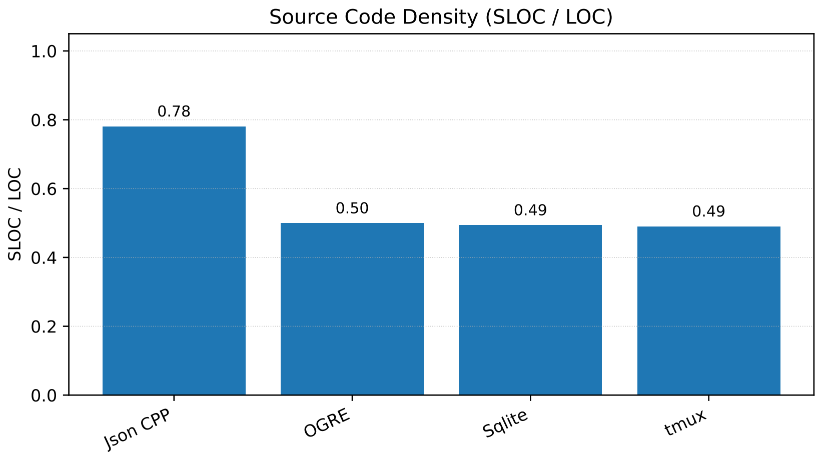


Figure 1.2: Source code density (SLOC/LOC ratio) for all projects.

Interpretation.

The LOC and SLOC measurements reveal three key observations:

- **Project size differences:** OGRE and SQLite are the two largest codebases, considerably larger than JsonCpp and tmux. This difference strongly impacts subsequent complexity metrics (Cyclomatic, Halstead, and Data-Flow Complexity).
- **Coding style variation:** JsonCpps higher SLOC/LOC ratio indicates denser code with fewer comments or blank lines, whereas OGRE, SQLite, and tmux exhibit more extensive commenting and structural spacing. A lower SLOC/LOC ratio typically suggests the presence of license headers, inline documentation, descriptive comments, and deliberate whitespace for readability. While this increases the total LOC, it often reflects a well-documented codebase that is easier to navigate and maintain, albeit with a larger non-executable footprint.
- **Impact on COCOMO:** Since COCOMO uses KLOC as its primary input, the large sizes of OGRE and SQLite lead to significantly higher predicted effort and development time in later estimation results.

3.2 Cyclomatic Complexity (CC)

Tabulated Complexity Statistics.

Table 1.2 reports the detailed CC statistics for all four projects, including the number of functions analysed, mean and median CC, standard deviation, and the minimum and maximum values. JsonCpp is the simplest project, whereas SQLite and tmux exhibit significantly higher variance and extreme maximum CC values, indicating complex control flow and deeply nested logic.

Table 1.2: Cyclomatic Complexity Statistics for All Projects

Project	Functions	Mean CC	Median CC	Min CC	Max CC	Std. Dev
JsonCpp	93	3.23	2.0	1	20	3.67
OGRE	4120	2.95	1.0	1	208	7.22
SQLite	3884	4.89	2.0	1	947	20.06
tmux	2263	5.80	3.0	1	705	17.13

Percentile-Based Distribution Analysis.

Table 1.3 reports the key percentiles (P75, P90, P95, P99), which describe how cyclomatic complexity values are distributed across each codebase. A percentile P_x indicates the CC value below which $x\%$ of all functions lie. For example, the P90 value is the CC threshold that 90% of functions do not exceed.

Percentiles are especially useful for understanding the "shape" of a distribution. While the mean and median summarise the centre of the data, the upper percentiles (P90–P99) capture the behaviour of the *heaviest and most complex* functions. High P95 and P99 values, as seen in OGRE, SQLite, and tmux, indicate a heavy-tailed distribution: most functions are simple, but a small number have extremely high complexity. These high-CC outliers are typically responsible for disproportionate maintenance cost and testing effort, and they serve as potential refactoring candidates.

Table 1.3: CC Distribution Percentiles and Chosen Caps

Project	Chosen Cap	P75	P90	P95	P99
JsonCpp	20	4.0	8.0	9.0	19.08
OGRE	25	3.0	6.0	9.0	25.00
SQLite	42	4.0	9.0	15.0	42.00
tmux	40	6.0	12.0	18.0	39.38

Average and Maximum CC.

Figures 1.3 and 1.4 show the average and maximum cyclomatic complexity across projects. JsonCpp maintains consistently low complexity, while SQLite reaches extreme maximum CC values due to large monolithic routines in its SQL parsing and execution engine. OGRE and tmux also contain several high-CC functions associated with rendering pipelines and terminal state management.

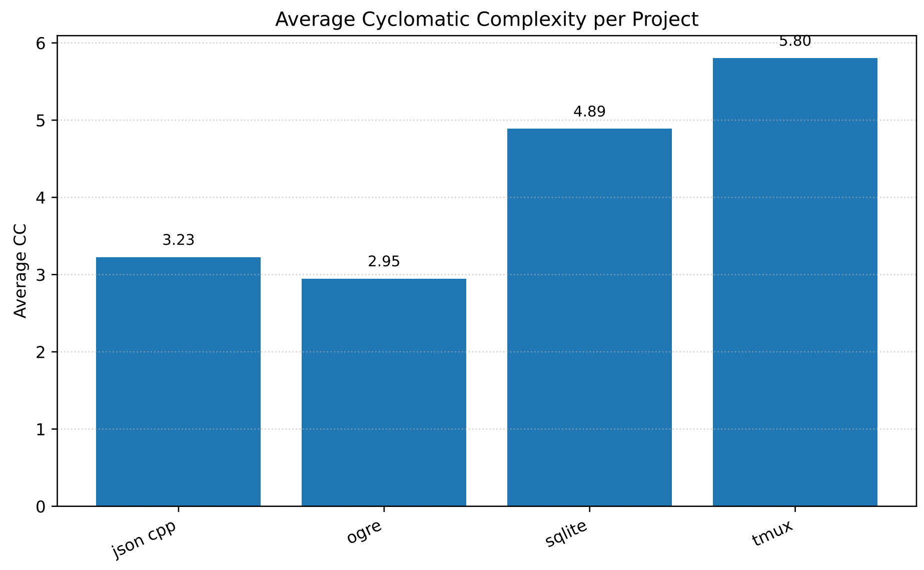


Figure 1.3: Average CC per project.

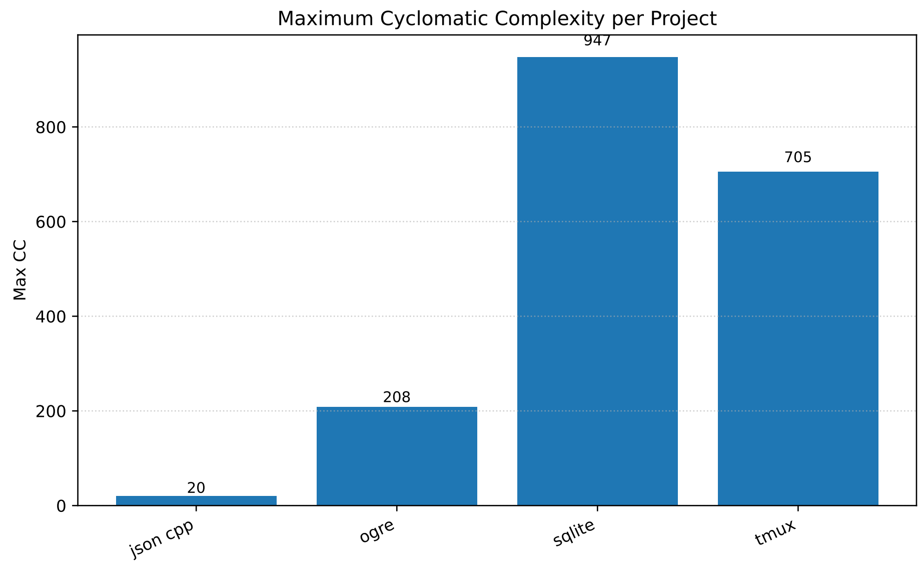


Figure 1.4: Maximum CC per project.

Distribution of Function-level CC.

Figures 1.5, 1.6, 1.7, and 1.8 show the CC distribution for each project with dynamic capping. JsonCpp and tmux exhibit compact distributions (mostly CC 15), whereas OGRE and SQLite display heavy-tailed patterns with a significant number of functions above CC 10.

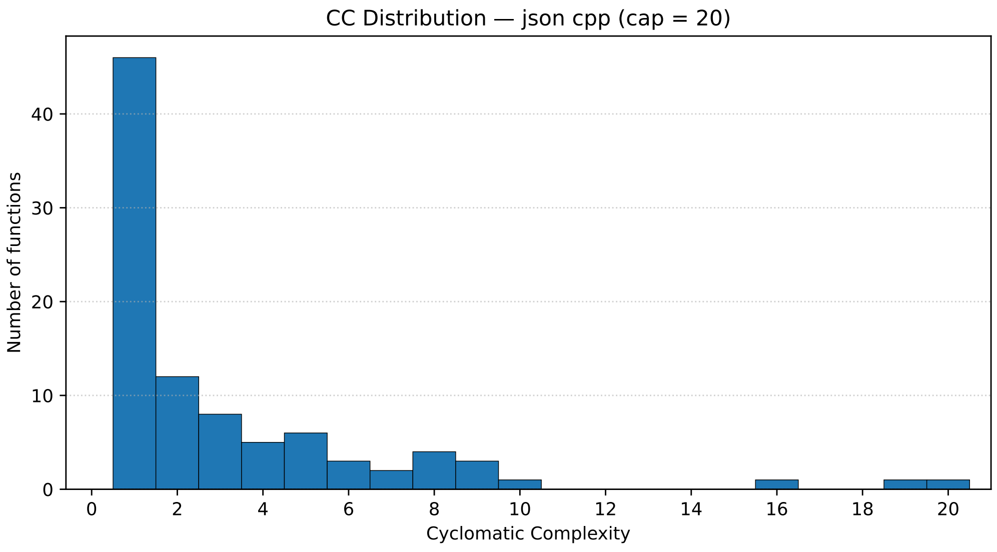


Figure 1.5: CC Distribution JsonCpp

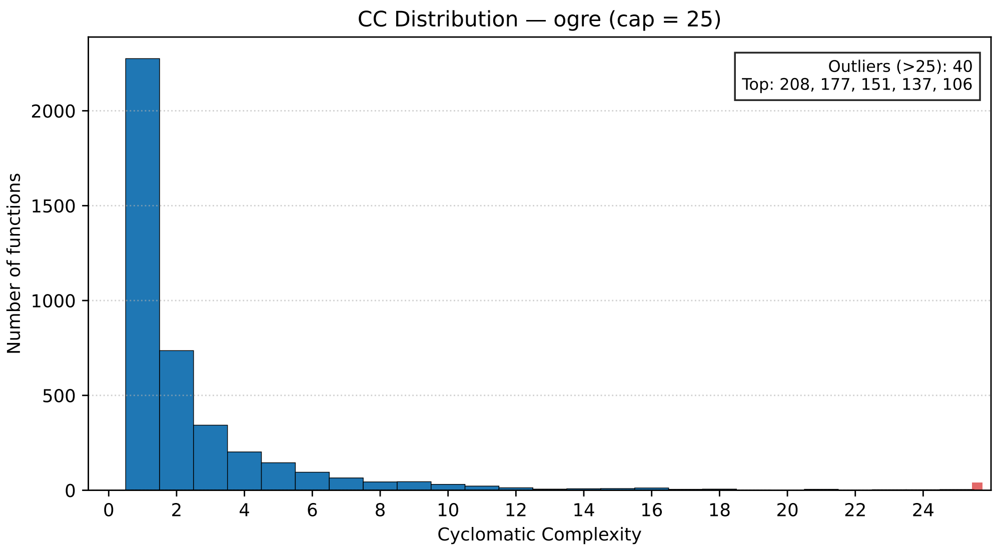


Figure 1.6: CC Distribution OGRE

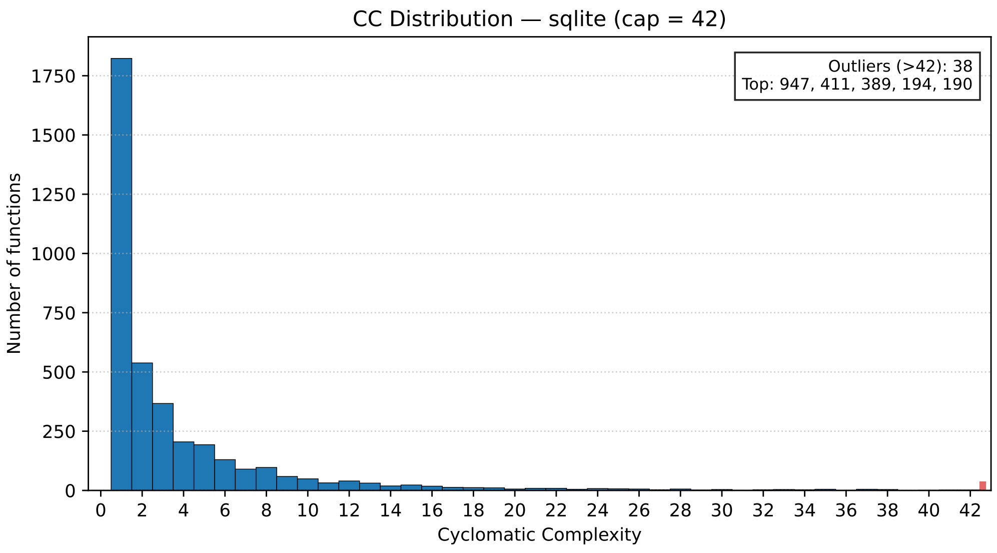


Figure 1.7: CC Distribution SQLite

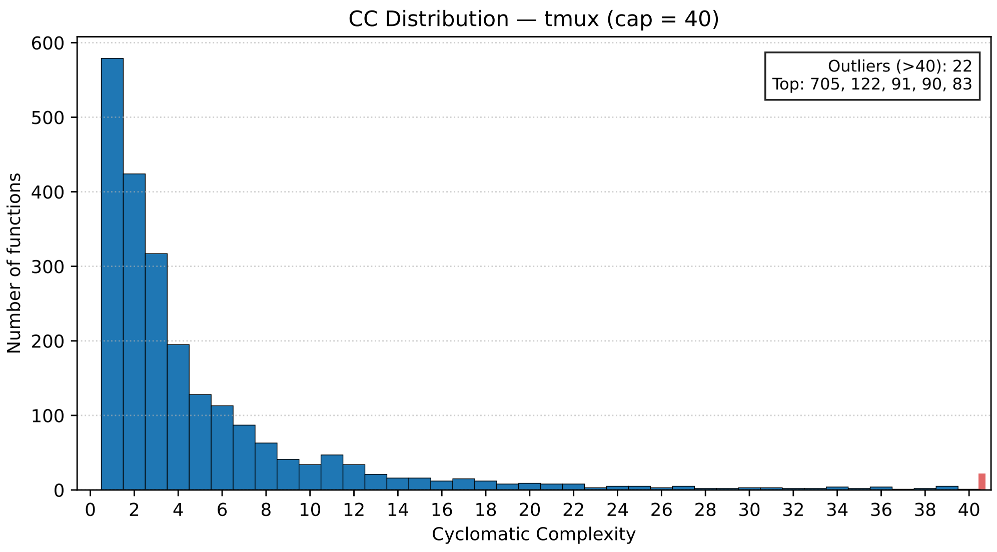


Figure 1.8: CC Distribution tmux

High-Complexity Function Counts.

Figure 1.9 summarises how many functions exceed CC thresholds of 5, 10, and 20. JsonCpp has almost no high-CC functions, while OGRE and SQLite show large volumes of functions above CC 10. This reinforces the presence of nested control logic spread throughout their codebases.

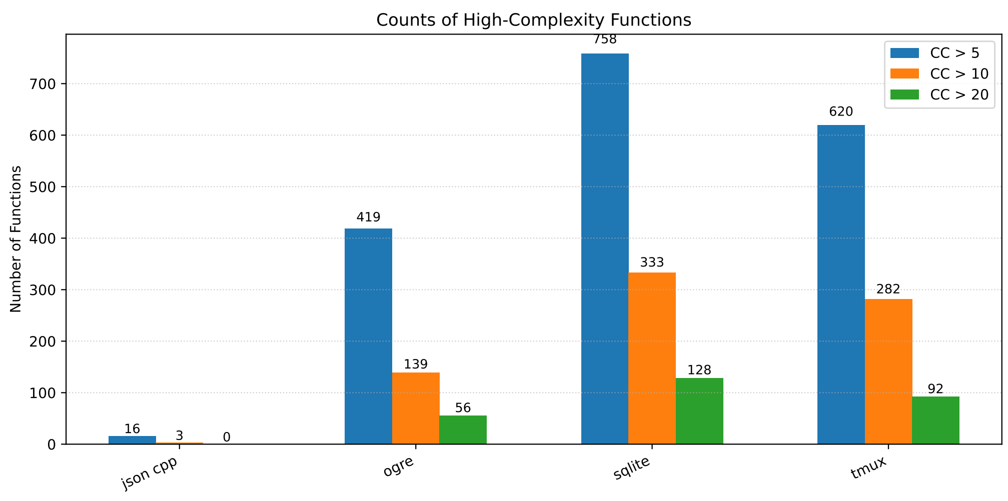


Figure 1.9: Counts of high-complexity functions (CC > 5, 10, 20).

Outlier Analysis.

Table 1.4 lists the number of functions that exceed the chosen cap and their top five CC values. SQLite and tmux contain extreme outliers CC 947 and CC 705 respectivelywhich indicate large, monolithic functions with substantial branching depth.

Table 1.4: Outlier Metrics and Top CC Values

Project	Cap	Outliers	Top 1	Top 2	Top 3	Top 4	Top 5
JsonCpp	20	0	–	–	–	–	–
OGRE	25	40	208	177	151	137	106
SQLite	42	38	947	411	389	194	190
tmux	40	22	705	122	91	91	91

Interpretation.

The cyclomatic complexity results highlight substantial variation in structural complexity across the four projects, revealing not just differences in average function complexity but also fundamentally different distributional behaviours.

- **JsonCpp exhibits highly uniform and predictable complexity.** Its low mean (3.23), low median (2), and low standard deviation (3.67) show a tightly controlled codebase with consistently simple control flow. The absence of any outliers beyond the chosen cap indicates that JsonCpp follows a highly disciplined modular style where no single function grows disproportionately complex. This predictability reduces cognitive load for developers and simplifies maintenance.
- **OGRE demonstrates broad dispersion with many moderately complex functions.** Although the mean CC is relatively low (2.95), the distribution is wide

(std. 7.22) and includes 40 outlier functions with CC values as high as 208. This discrepancy between the central tendency and the extreme values reveals a codebase that is largely simple at the per-function level, but periodically contains subsystemsparticularly rendering and resource pipelinesthat accumulate deeply nested control structures. This "mixed" profile is typical of large graphical engines where boilerplate code coexists with computation-heavy kernels.

- **SQLite shows the most pronounced heavy-tailed distribution.** With the highest standard deviation (20.06) and extreme outliers reaching CC = 947, SQLites complexity profile is dominated by a small number of exceptionally large functions. These pathological values are characteristic of systems implementing parsers, interpreters, and execution engines, where the accumulation of conditional branches, state transitions, and error paths produces deeply tangled control flow. The stark contrast between median CC 2 and maximum CC 947 reflects exceptionally high skewness, requiring disproportionate testing and representing clear long-term maintenance risks.
- **tmux combines moderate typical complexity with severe outliers.** While its mean (5.8) and median (3) suggest moderately complex everyday functions, the standard deviation (17.13) and extreme maximum CC 705 indicate the presence of a small number of highly intricate routines. These are characteristic of terminal state machines, mode-switching logic, and multiplexing paths. The structure resembles SQLites but on a smaller scale: broadly manageable complexity punctuated by large, monolithic functions.
- **Percentile behaviour confirms high skewness in OGRE, SQLite, and tmux.** The gap between P50 (medians 13) and P99 (19 to 42) demonstrates that the overwhelming majority of functions are simple, yet each of these projects contains a long tail of highly complex routines. This long-tailed behaviour is a well-known driver of software defects, as empirical studies consistently show that a small fraction of functions account for a large fraction of bugs and maintenance time.
- **Practical implications for maintenance, testing, and refactoring.** The extreme outliers (208, 705, 947) represent high-risk, high-cost regions of the codebase. Functions with CC above 20 already require extensive branch coverage to test

3.3 Halstead Metrics and Effort

Definition. Halstead's software science quantifies software complexity using the lexical properties of the programnamely the number of operators, operands, program vocabulary, length, volume, difficulty, effort and estimated bugs delivered. These metrics approximate the cognitive load required to understand and modify the system.

Tabulated Results. Table 1.5 summarizes the aggregated Halstead metrics for all projects. These values represent whole repository measurements and therefore capture the high-level lexical and cognitive characteristics of each system.

Table 1.5: Aggregated Halstead Metrics for All Projects

Project	**Vocabulary**	**Length**	**Volume**	**Difficulty**	**Effort**	**Time (s)**	**Estimated Bugs**
JsonCpp	735	22,353	2.12×10^5	274.32	5.84×10^7	3.24×10^6	70.95
OGRE	15,068	533,563	7.41×10^6	321.16	2.38×10^9	1.32×10^8	2,468.48
SQLite	12,999	724,515	9.90×10^6	505.48	5.00×10^9	2.78×10^8	3,300.43
tmux	7,847	452,814	5.86×10^6	494.54	2.90×10^9	1.61×10^8	1,952.82

Effort vs Difficulty

Purpose. Figure 1.10 examines the relationship between Halstead Difficulty and Halstead Effort, two metrics that jointly describe the cognitive demands of a software system. In this context, *Difficulty* captures the structural intricacy of the codespecifically, how complex the relationships between operators and operands are within the program. A higher Difficulty value indicates that individual expressions are harder for a developer to reason about.

Effort, by contrast, estimates the total mental workload required to understand, implement, or modify the system as a whole. It grows with both the logical intricacy of the code (Difficulty) and the sheer volume of lexical content (Volume).

By plotting these two measures together, the figure reveals whether a project's cognitive burden arises primarily from dense, tightly coupled logic, from large amounts of code, or from a combination of both. This provides a high-level view of where the primary sources of developer effort originate within each codebase.

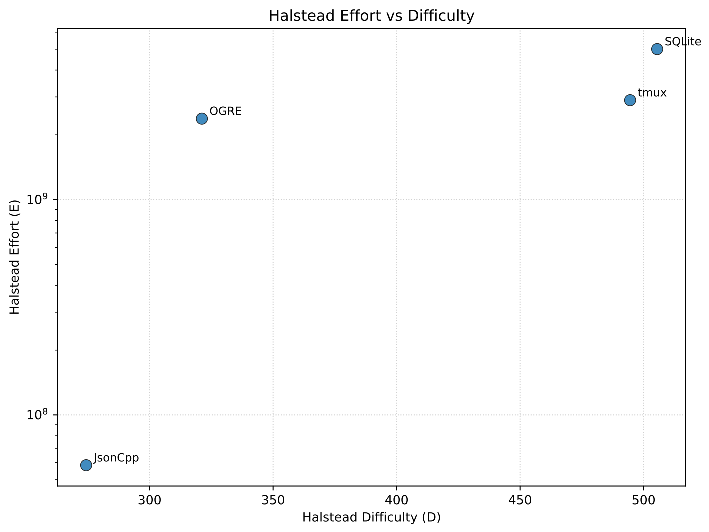


Figure 1.10: Halstead Effort vs Difficulty across all projects (Effort plotted on a logarithmic scale).

Difficulty-Volume Quadrant Analysis

Purpose. To characterize the structural profile of each repository, Figure 1.11 plots Halstead Difficulty against Volume, with median split-lines forming four analytical quadrants. Difficulty captures the logical intricacy of the code, while Volume reflects its overall lexical size. This view of the "quadrant" makes it easy to see whether the complexity of a project is primarily driven by large amounts of code, by dense and intricate logic or by a combination of the two.

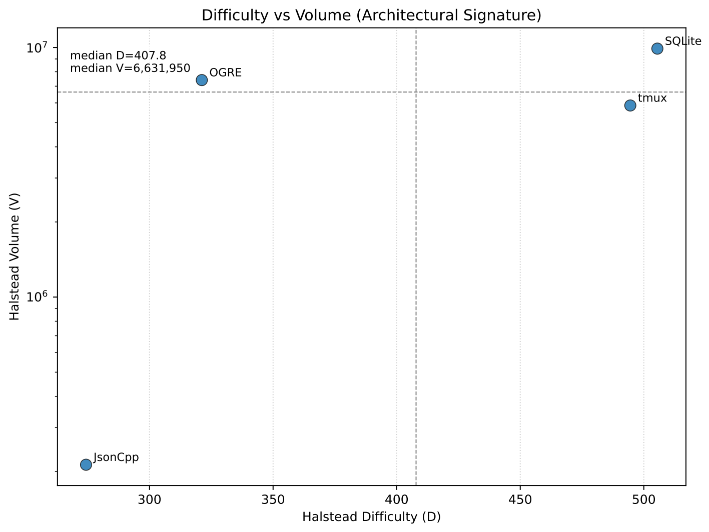


Figure 1.11: Difficulty vs Volume quadrant plot showing structural complexity signatures.

Normalized Effort per Function

Purpose. Raw Halstead Effort increases rapidly with project size, which can obscure how cognitively demanding individual parts of the system actually are. To obtain a more maintainability-focused perspective, Figure 1.12 reports a normalized measure of effort per-function (or per vocabulary unit where function counts are not available). This metric reflects the average mental workload associated with understanding or modifying a typical functional unit in the code, allowing for clearer comparison of maintainability across projects of different sizes.

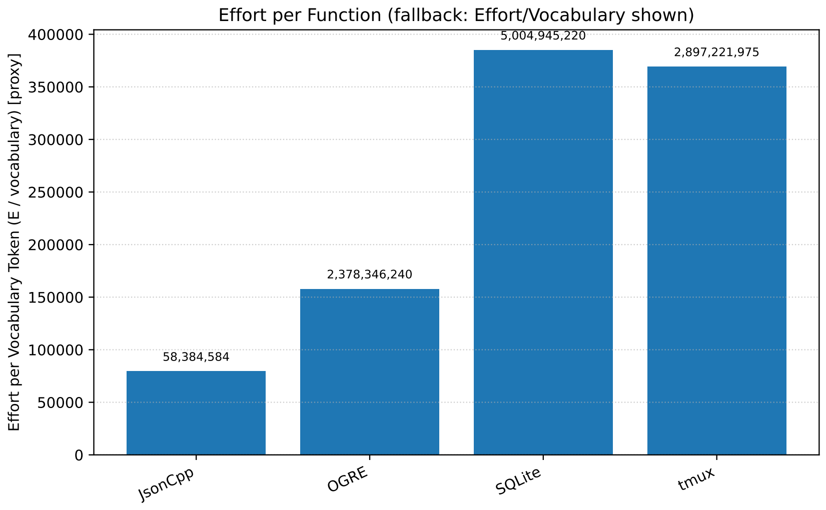


Figure 1.12: Normalized Halstead effort per function (or per vocabulary unit).

Interpretation

The Halstead results highlight distinct lexical and cognitive profiles across the four software systems:

- **JsonCpp is compact and cognitively lightweight.** With small vocabulary, short program length, and low Effort, JsonCpp exhibits minimal lexical complexity and is therefore highly maintainable, consistent with its role as a simple utility library.
- **OGRE's cognitive cost arises from scale rather than density.** Despite only moderate Difficulty, OGRE's enormous Volume drives its high Effort. This indicates a broad but not deeply nested codebase, aligning with its large rendering engine architecture.
- **SQLite combines high lexical complexity with high logical density.** Its

very large Volume and extremely high Difficulty yield the highest Effort of all four systems. These results reflect the inherent complexity of query parsing, execution planning, and virtual machine interpretation.

- **tmux is compact but cognitively dense.** Although much smaller than SQLite or OGRE, tmux displays high Difficulty, indicating intricate state transitions and control-flow interactions typical of a terminal multiplexer.
- **Effort-Difficulty analysis distinguishes complexity sources.** OGRE's effort derives from lexical scale; tmux's from logical intricacy; SQLite's from both; JsonCpp's from neither. This separation is meaningful for forecasting defect-proneness and maintenance cost.
- **The DifficultyVolume quadrant acts as a structural fingerprint.** JsonCpp resides in the "simple" quadrant, OGRE in "large", tmux in "dense", and SQLite in "dense + large". Such classification is valuable when guiding refactoring priorities, code reviews, and testing strategies.

Overall, the Halstead metrics and visualisations complement the earlier Cyclomatic and Data-Flow analyses by offering a lexical perspective on software complexity. They reveal not only how much code exists, but how cognitively demanding it is, and why certain systems impose significantly higher maintenance burdens.

3.4 Data-Flow Complexity (DFC)

Definition. Oviedo's Data-Flow Complexity (DFC) measures the number of inter-block definition-use relationships in a function. Higher DFC values indicate longer and more intricate data-flow chains, which increase testing difficulty, cognitive load, and maintenance risk.

Repository-level Summary. Table 1.6 gives a compact repository-level summary: the number of source files analysed, the number of functions discovered, and the total DFC aggregated for each codebase. These values give a quick, high-level view of how concentrated data-flow complexity is within each repository.

Table 1.6: Repository-level summary: number of files, number of functions, and total DFC

Project	#Files	#Functions	Total DFC
JsonCpp	4	350	2,125
OGRE	216	5,397	39,495
SQLite	102	3,207	101,412
tmux	193	2,370	51,570

Brief analysis of the repository summary. To interpret the table quantitatively we compute two helpful derived measures:

- **Average DFC per function** (Total DFC œ #Functions), which shows how much inter-block data-flow complexity an *average* function carries.
- **Functions per file** (#Functions œ #Files), which indicates how concentrated code logic is inside individual source files.

Using the values in Table 1.6 we obtain:

- **JsonCpp:** Avg DFC per function $\approx$ 6.07, functions per file = 87.50. Small total DFC but high functions-per-file (few files hosting many small functions).
- **OGRE:** Avg DFC per function $\approx$ 7.32, functions per file $\approx$ 24.99. Moderate per-function DFC but large scale (many functions and files).
- **SQLite:** Avg DFC per function $\approx$ 31.62, functions per file $\approx$ 31.44. Very high per-function DFC each function, on average, has deep data-flow chains.
- **tmux:** Avg DFC per function $\approx$ 21.76, functions per file $\approx$ 12.28. High per-function DFC concentrated in fewer functions per file compared to OGRE/SQLite.

DFC Distribution Across Projects

Purpose. The distribution of Data-Flow Complexity provides an initial, high-level view of how inter-block data dependencies are spread across each codebase. Examining the full shape of the distribution rather than only summary statistics helps identify whether a project consists mostly of simple, low-DFC functions or whether it contains substantial pockets of highly coupled, definition use dense logic. A consolidated 2x2 grid of histograms allows direct visual comparison of skewness, tail behaviour, and the prevalence of extreme values across all repositories.

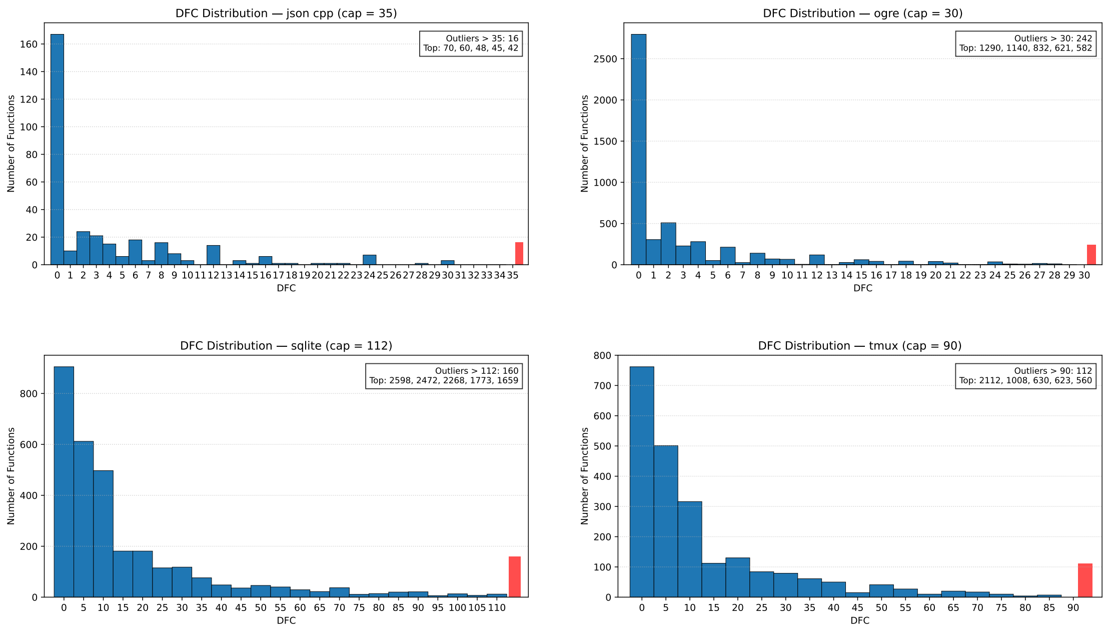


Figure 1.13: DFC distribution across JsonCpp, OGRE, SQLite, and `tmux`.

Across the four projects, JsonCpp exhibits a tight, low-variance distribution with almost all functions falling in the very low DFC range. The other three codebases show significantly more skewed distributions. OGRE contains a broad spread with a noticeable long tail, indicating the presence of subsystems whose data-flow structure is substantially more involved. SQLite and `tmux` display the strongest heavy-tailed behaviour: although many functions remain simple, both projects contain a considerable number of very high-DFC outliers, suggesting the existence of deeply interconnected control paths and multi-stage data propagation. These distributions foreshadow the more detailed complexity patterns confirmed in later analyses.

Cross-Project Boxplot

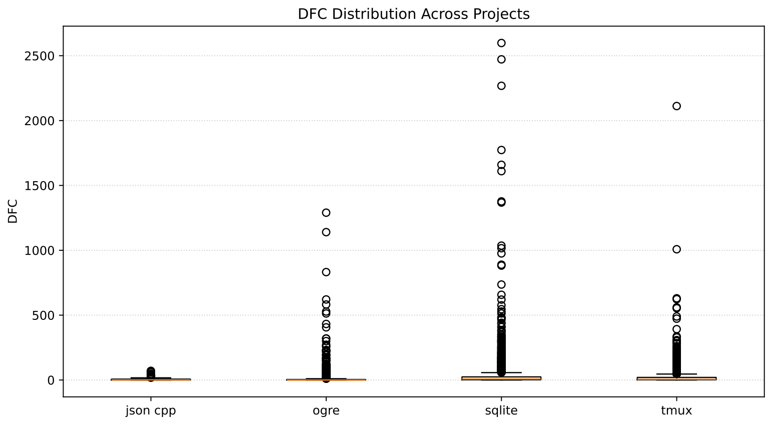


Figure 1.14: Cross-project DFC distribution boxplot.

The cross-project boxplot provides a compact comparative view of spread and variability. JsonCpp is tightly bounded with short whiskers, indicating consistently low DFC across functions. OGRE shows a wider interquartile range, reflecting more architectural diversity, but remains moderate compared to the largest systems. SQLite and `tmux` exhibit long upper whiskers and numerous extreme outliers well beyond their upper quartiles. This skewness signals the existence of function clusters with highly entangled definition–use chains, which represent concentrated testing and maintenance risk.

Total Data-Flow Complexity per project.

Purpose. Figure 1.15 compares total accumulated DFC across repositories, indicating macro-level system complexity driven by size and function interdependence.

SQLite exhibits the highest total DFC, reflecting both its substantial codebase size and the intrinsically interdependent nature of its query-processing pipeline. This is followed by `tmux`, whose event-driven state management produces dense definitionuse chains within fewer but highly complex modules. OGRE, despite being large, shows a more moderate aggregate DFC due to its subsystem-oriented structure, while JsonCpp remains the lowest, consistent with its relatively compact and modular design.

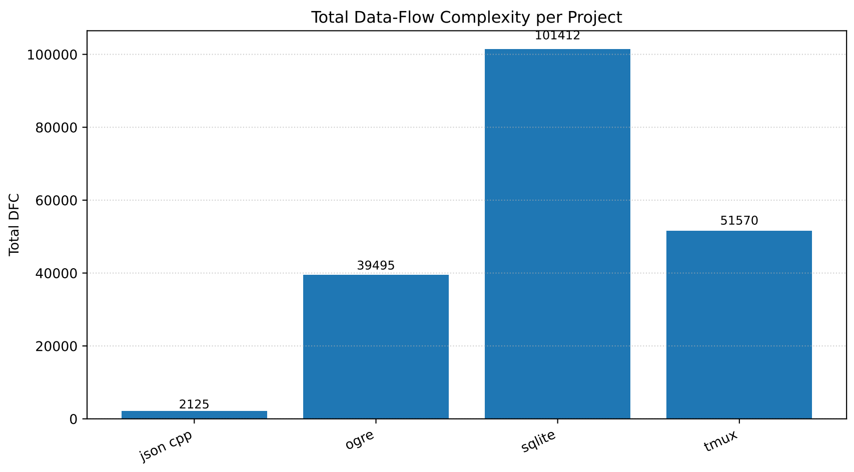


Figure 1.15: Total Data-Flow Complexity per project.

DFC vs NLOC Scatter Plots

Purpose. The scatter plots explore the relationship between function size (NLOC: number of logical source lines) and Data-Flow Complexity. Plotting both together reveals whether larger functions tend to accumulate proportionally more inter-block definitionuse interactions, and highlights atypical functions that exhibit high DFC despite modest size.

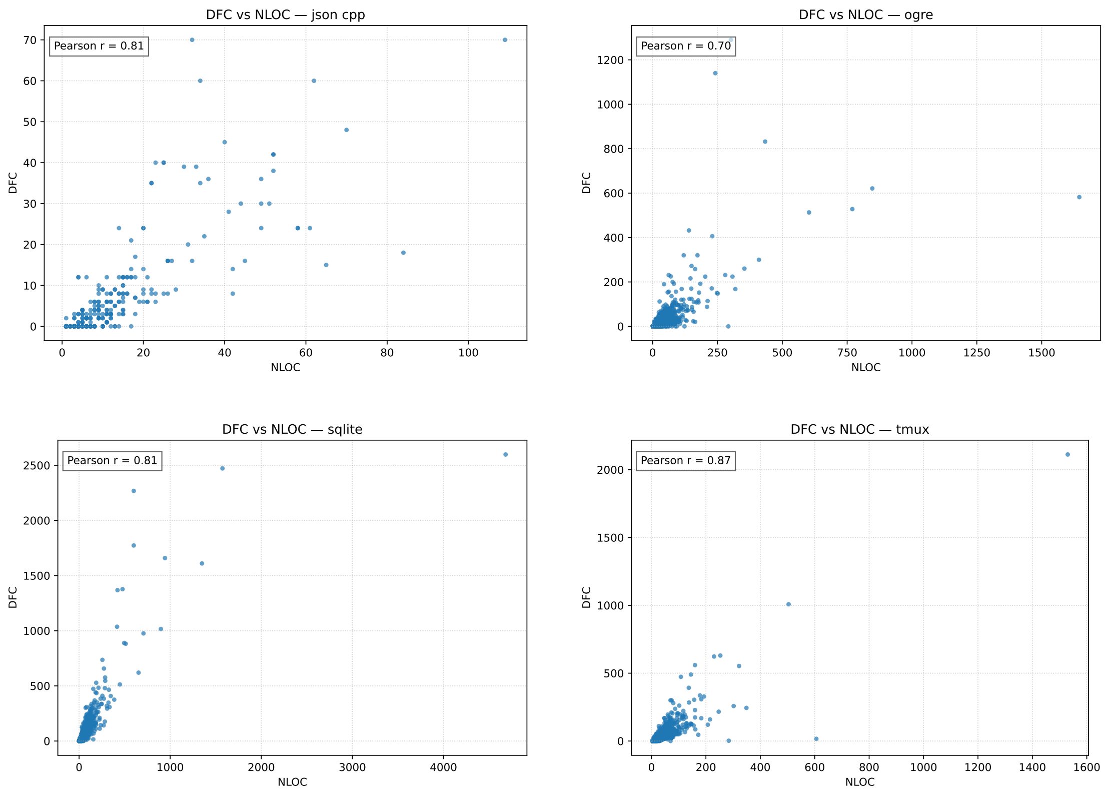


Figure 1.16: DFC vs NLOC for all projects.

The scatter plots show a clear positive relationship between function size and dataflow complexity, with Pearsons r values between 0.70 and 0.87. Since Pearsons r quantifies the strength of linear correlation (where $r = 1$ indicates perfect alignment), these values suggest that larger functions tend to accumulate greater definitionuse interactions.

Notably, SQLite and `tmux` contain high-DFC outliers at only moderate NLOC, indicating that complexity is sometimes driven by dense data dependencies rather than sheer length. Such points highlight architectural hotspots where data flow becomes intricate irrespective of function size.

Implications.

- **Per-function data-flow intensity.** SQLite and `tmux` have the highest average DFC, meaning their functions contain dense definition–use interactions and therefore demand greater effort to understand, test, and safely modify.
- **File-level structural organisation.** JsonCpps many low-DFC functions packed into few files reflect a compact and utility-oriented structure, though the limited file modularity may hinder large-scale evolution.
- **Scale-driven complexity in OGRE.** OGREs moderate DFC but very large size suggest that complexity arises more from subsystem breadth and integration than from deep intra-function data coupling.
- **Heavy-tailed distributions as hotspots.** SQLite and `tmux` show strong heavy tails with extreme-DFC outliers, signalling specific routines where data-flow coupling becomes disproportionately intricate and where maintenance risk is highest.
- **Maintenance and testing priorities.** High-DFC clusters in SQLite and `tmux` should be prioritised for targeted refactoring and regression testing, while JsonCpp and OGRE require attention to modular structure and subsystem coordination, respectively.

3.5 COCOMO Effort and Schedule Estimation

Results Across the Three COCOMO Modes

To evaluate size-driven development effort, the Basic COCOMO model was applied under the three standard modes—Organic, Semi-detached, and Embedded—using the SLOC values obtained earlier. The following tables report estimated effort (person-months), development time (months), and average staffing levels.

The Basic COCOMO effort and development time equations are:

$$E = a_b \times (\text{KLOC})^{b_b}, \qquad D = c_b \times (E)^{d_b},$$

where (a_b, b_b) determine the effort growth rate, and (c_b, d_b) determine the schedule behaviour. Their values differ by development mode, as shown in Table 1.7.

Table 1.7: Basic COCOMO Coefficients for the Three Development Modes

Mode	$\boldsymbol{a_b}$	$\boldsymbol{b_b}$	$\boldsymbol{c_b}$	$\boldsymbol{d_b}$
Organic	2.4	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Table 1.8: COCOMO Organic Mode Estimates

Project	**SLOC (KLOC)**	**Effort (PM)**	**Time (Months)**	**People**
JsonCpp	8.07	21.50	8.02	2.68
OGRE	93.53	281.65	21.32	13.21
SQLite	107.07	324.60	22.50	14.42
tmux	45.37	131.78	15.98	8.25

Table 1.9: COCOMO Semi-detached Mode Estimates

Project	**SLOC (KLOC)**	**Effort (PM)**	**Time (Months)**	**People**
JsonCpp	8.07	31.11	8.33	3.74
OGRE	93.53	483.71	21.75	22.24
SQLite	107.07	562.78	22.94	24.53
tmux	45.37	215.15	16.38	13.13

Table 1.10: COCOMO Embedded Mode Estimates

Project	SLOC (KLOC)	Effort (PM)	Time (Months)	People
JsonCpp	8.07	44.12	8.40	5.25
OGRE	93.53	834.53	21.52	38.78
SQLite	107.07	981.51	22.66	43.31
tmux	45.37	350.32	16.30	21.49

Implications.

Cross-mode behaviour. Across all modes, the ordering of predicted effort remains consistent: *SQLite* > *OGRE* > *tmux* > *JsonCpp*. This follows directly from their KLOC profiles and illustrates the size-sensitivity of Basic COCOMO.

Effort scaling in large systems. OGRE and SQLite exhibit steep increases in effort as mode assumptions tighten, especially in the Embedded model. This reflects the high coordination overheads and subsystem interdependencies characteristic of graphics engines and database kernels.

Stability in compact systems. JsonCpp, being relatively small and modular, shows only modest variation across modes. Its required staffing stays below five people even in the Embedded model.

Team size implications. Predicted staffing levels for OGRE and SQLite exceed 20–40 developers under semi-detached and embedded assumptions, indicating that such systems typically require parallel development teams and substantial integration effort.

Overall takeaway. COCOMO reinforces insights from earlier structural metrics: large, interconnected systems do not simply scale linearly with size—their effort estimates grow superlinearly due to increased complexity, tighter constraints, and integration overheads.

4 Usefulness of the Metrics

Approach: Correlation-Based Usefulness Analysis

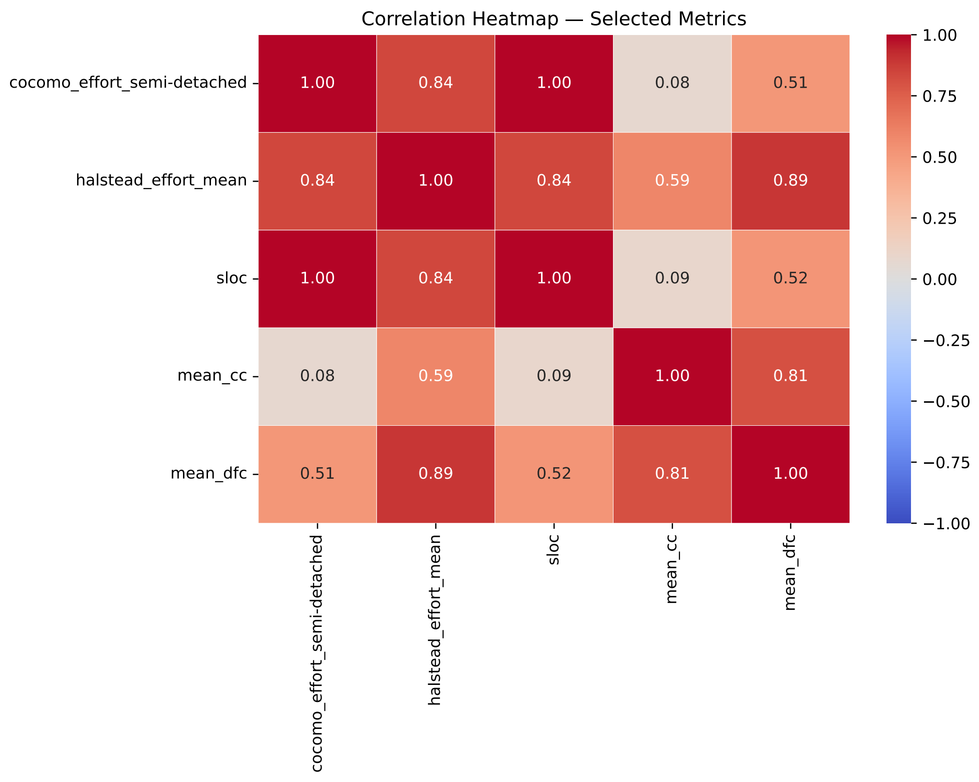
To move beyond subjective commentary, we quantitatively analysed the *usefulness* of each metric by computing how strongly it correlates with each other . The intuition is that a metric is more informative if it consistently tracks underlying structural or cognitive difficulty across projects.

Algorithm.

For each project, we constructed a unified metric table containing:

- SLOC (size)
- Mean Halstead Effort (cognitive complexity)
- Mean Cyclomatic Complexity (control-flow complexity)
- Mean Data-Flow Complexity (data-dependency complexity)
- COCOMO Effort (semi-detached mode), representing cost-oriented impact

We then computed a Pearson correlation matrix across these metrics and visualised it through the heatmap in Figure 1.17. Metrics exhibiting strong, stable correlations with multiple others are considered more predictive of overall software difficulty, and thus more useful for guiding engineering decisions.



	cocomo_effort_semi-detached	halstead_effort_mean	sloc	mean_cc	mean_dfc
cocomo_effort_semi-detached	1.00	0.84	1.00	0.08	0.51
halstead_effort_mean	0.84	1.00	0.84	0.59	0.89
sloc	1.00	0.84	1.00	0.09	0.52
mean_cc	0.08	0.59	0.09	1.00	0.81
mean_dfc	0.51	0.89	0.52	0.81	1.00

Figure 1.17: Correlation heatmap across selected complexity and effort metrics.

Interpreting Correlations.

In the heatmap, a *high positive correlation* (values close to +1) means that two metrics tend to increase together across projects, indicating that they capture related aspects of software scale or complexity. A *low or near-zero correlation* indicates that the metrics vary independently, showing that they measure distinct properties of the codebase.

Moderate correlations suggest partial overlap but still retain complementary information.

Final Interpretation: Which Metrics Are Most Useful?

Based on the correlation structure, the following conclusions can be drawn:

- **Halstead Effort is the most useful single metric.** Its strong correlations with both DFC ($r = 0.89$) and SLOC ($r = 0.84$) show that it reflects a combination of size and structural complexity. This makes it a balanced indicator of overall cognitive effort.
- **Data-Flow Complexity is highly informative for maintainability.** Its strong alignment with Halstead Effort ($r = 0.89$) and CC ($r = 0.81$) suggests that DFC captures deep dependency relationships that are often the main source of testing and debugging difficulty.
- **Cyclomatic Complexity remains useful but narrower in scope.** It reflects branching structure (moderate correlation with DFC) but does not track project size or cost, making it valuable for reviewing control-heavy functions but insufficient alone for system-wide conclusions.
- **SLOC and COCOMO are useful for cost estimation but not complexity.** Their near-perfect correlation ($r = 0.99$) shows that both metrics describe the same phenomenon: project scale. Since they correlate weakly with CC and only moderately with DFC, they provide little insight into structural difficulty.

Overall, the heatmap indicates that **Halstead Effort and DFC are the most informative metrics for analysing software complexity**, while **SLOC and COCOMO primarily measure scale**. Cyclomatic Complexity sits in the middle, offering structural insight but limited predictive power on its own.