

# RAX: A porting complexity evaluation tool for RISC-V

Anonymous Author(s)

## ABSTRACT

In recent years, the RISC-V's open-source and reduced features have received active promotion of porting from the industry. Due to the lack of publicly available standards for assessing software portability, the software porting process for the RISC-V architecture has been predominantly reliant on expert experience, presenting obstacles. This paper introduces an automated tool, RAX, for assessing the porting complexity specifically for RISC-V by considering architecture-specific code and code Cyclomatic Complexity. RAX provides precise evaluation solutions and establishes a high-quality dataset for porting complexity assessment, enabling recommendations of suitable software difficulty levels for developers of varying skill levels. We continue to expand the dataset and conduct effective evaluations of the tool within the open-source community. We have gathered valuable feedback, including 47 positive responses, from 51 specific open-source projects. Interested users can explore the open-source implementation of RAX at (<https://github.com/wangyuliu/RAX-2024>), or watch the RAX video demo at (<https://github.com/wangyuliu/RAX-2024>).

## CCS CONCEPTS

• **Software and its engineering** → **Software libraries and repositories**; *Software post-development issues*.

## KEYWORDS

ISA (Instruction set architecture), RISC-V, Porting Complexity Evaluation

## 1 INTRODUCTION

In recent years, the RISC-V Instruction Set Architecture (ISA) has garnered significant attention from academia, particularly the industry, due to its open-source and reduced design.[? ]. OpenEuler has joined RISC-V International organization, and RISC-V has officially become openEuler's official support architecture. The mainline integration of openEuler RISC-V 23.09 was finished by the RISC-V SIG. The main repository of openEuler 23.03 has successfully ported 4,244 software packages to the RISC-V64, while the openEuler 23.09 main repository has currently completed the porting work of 5,694 software packages[? ]. OpenCloudOS Kernel Stream 2207.2 kernel version adds support for the RISC-V 64 architecture[? ].

Despite the rapid development of the RISC-V ISA, which foundational software such as operating systems and compilers are relatively mature, there is still a significant demand for porting a large number of application software.[? ]. To adapt to the RISC-V, efficiently porting software package is greatly depending on domain expertise. During the initial planning phase of porting, the lack of standardized evaluation criteria for software portability can lead to misallocation of resources in the development process. Therefore, it is desirable to propose a method that automates the assessment of porting complexity, this will enable different development teams to access software packages that are tailored to their proficiency

levels, thus assisting distribution vendors in the porting of RISC-V architecture.

Du et al. have developed a basic statistical evaluation plan for CPU architecture code in project[? ], however, their work lacks a thorough validity analysis of factors, and fails to consider the complexity associated with the software operation and maintenance process.

RAX can provide developers with accurate evaluation, recommend software with different levels of porting difficulty based on their skill levels. The contributions of this paper are as follows: The contributions of this paper are as follows:

- (1) In recent years, the RISC-V's open-source and reduced features have received active promotion of porting from the industry.
- (2) We explore building high-quality datasets that can cover various porting scenarios to solve the problem of imbalanced datasets of previous tools. Tuning machine learning models to address ambiguous identification of samples of medium and hard complexity.
- (3) We nearly doubled the dataset compared to other tool, and verified the effectiveness of the evaluation of 51 extension projects in real porting scenarios through their respective communities or forums. Our findings indicate that RAX outperforms other tool, delivering substantial improvements in terms of performance and accuracy.

## 2 RAX

The following is an introduction to the process of RAX construction, as shown in Figure ?? . During the analysis phase, architecture-related factors are obtained through the Commit tool for selection while investigating the impact of code complexity metrics on porting. A scanning tool is built to get potential modification workload. Finally, an appropriate machine learning model is selected as a classification.

### 2.1 Motivation

To design RAX that assists developers with porting work, the following research questions are proposed:

**RQ 1:** How can we select architecture-relevant actors that is fine-grained to design an automated tool that assesses the porting difficulty?

**RQ 2:** How to solve the problem of vague classification of projects with medium and high porting difficulty in previous tools?

To answer research question 1, this section provides a validity analysis of porting-related factors on RAX. First, we try to use code complexity metrics to evaluate porting work. The software metrics of code Cyclomatic Complexity (CC) can help developers make macro judgments on software complexity and maintenance difficulty, evaluate potential and hidden porting difficulties, which help the tool improve accuracy [? ]. At present, CC indicators are mostly used in project code quality assessment, defect inspection, code reconstruction, etc [? ]. During the porting process, developers will

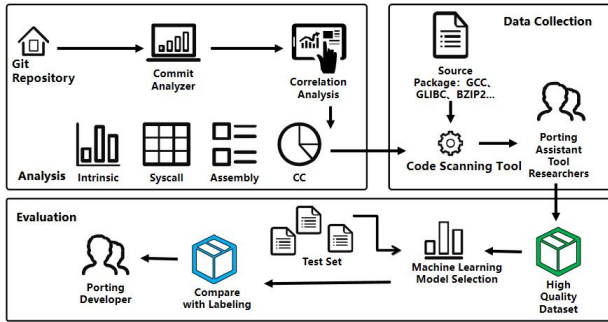


Figure 1: Flow chart for method to build RAX

target a large number of project source codes, build scripts, assembly codes, and a macro understanding of the project is required in the early stage of porting. This part of the work has a lot of overlap with the scope of code complexity. Through our survey of community, we have discovered that even though architectural code can be separated quite clearly from common code, when faced with complex and large-scale projects, we still require numerous iterative tests to uncover unknown architectural code. Building upon this, we delve into the quantification of the significance of cyclomatic complexity in porting work.

**2.1.1 Factor 1: Cyclomatic Complex(CC).** We use the lizard CC detection tool and optimize its functions, and it will introduce it in Section ?? . This study analyzes the relationship between CC and software porting complexity using the Spearman Rank correlation coefficient analysis method. We invited five Master's students who are engaged in research on porting auxiliary tools to score the porting complexity (score) of 131 software adapted for RISC-V, collected from the OpenEule list [?] and GitHub. According to the collection of Internet events, log analysis, source code dismantling, and similar project analogy [?], we form a scoring interval in the range of [0 5]. The Spearman coefficient of CC and score is 0.608, with a P-value of  $1.39e-14$ , which is less than 0.01. The results suggest that CC has a moderate positive correlation with porting difficulty. This is because developers need a holistic understanding of the project during the porting process, and CC can represent the overall complexity of the project, potentially reflecting porting barriers, as shown in Figure ??.

Second, we consider the effectiveness of the architecture binding code structure proposed by the other tool and conduct comparative experiments using more potential factors.

**2.1.2 Factor2: Arch\_code.** Based on the analysis of previous evaluation tools, we selected factors with the greatest potential and that can be extracted through automated methods to define Arch\_code. Arch\_code includes assembly code, conditional compilation and

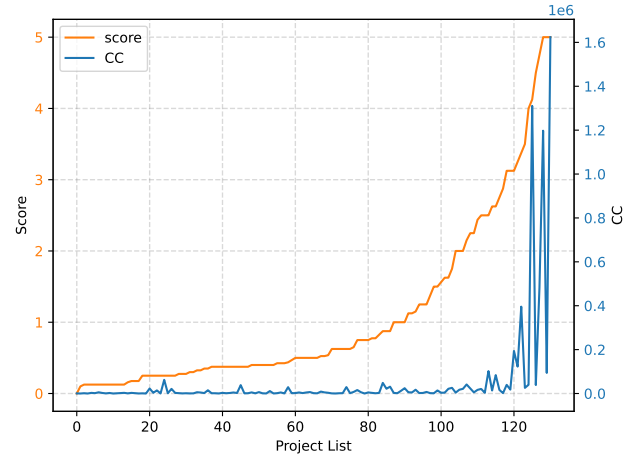


Figure 2: The relationship between score and CC

macro structure, Intrinsic functions, Builtin functions, architecture-related build scripts, and system calls. We use an automated architecture binding code scanning tool to obtain statistics on the number of lines and frequencies of Arch\_code.

Continue to collect other factors that may affect the evaluation of porting complexity, and select the following parameters through the Commit analyzer: architecture-related code lines, architecture-related files, and architecture keywords. The number of architecture-related code lines and files is located using the keywords of the RISC-V architecture, such as riscv, rv32, rv64, RISC-V, RISC-V and so on.

Correlation comparison between Arch\_code and other factors acquired by Commit analyzer, as shown in Figure ?? : The results indicate that Arch\_code has a higher correlation with score compared to other factors, and it also aligns with the significant results of the Pearson correlation test. The use of only architecture keywords for statistics yields moderate effectiveness in evaluating tool because keywords can only be detected from comments and macro sections of Commit, while critical porting code may lack these keywords, but the keyword count can reflect the amount of porting modification from the side. As shown in the Figure ??, the parameters of Arch\_code show a correlation of 0.45 to 0.75 with score, indicating a strong relationship. However, using a single factor alone does not result in a significantly strong correlation, as the combined modifications of these factors make up the porting workload.

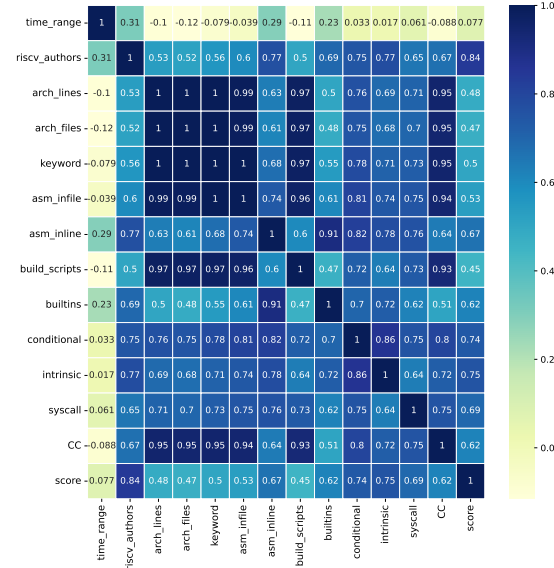
In the end, our tool selection was CC and the Arch\_code build scanning tool, as shown in Table ??.

## 2.2 Dataset

We used the project upstream warehouse to build the data set, that projects mainly targeting C and C++ are from the OpenEuler [?] list and GitHub, and invited 5 postgraduate students who are engaged in the research of RISC-V porting auxiliary tools to participate in the experimental evaluation. Specific evaluation methods can be found in 2.1.1. The evaluation scores are divided into three categories: low,

**Table 1: Tool factors comparison**

	Full Assembly	Inline Assembly	Intrinsic	Builtin	System Call	Conditional compilation and macro	Buildscripts	CC
Other tool	✓	✓	✓	✓	✓	✓	✓	
RAX	✓	✓	✓	✓	✓	✓	✓	✓

**Figure 3: The heat map of pearson correlation**

medium, and high[? ].This includes 139 projects as training, testing, and validation datasets, while more than 80 additional projects were used as a tool evaluation dataset. To gather insights from developers, the community, forums, and emails were utilized, resulting in over 51 meaningful feedback responses. In order to address the issue of data imbalance caused by collecting datasets solely from the openEuler mailing list, where more than 80% of the projects were initially evaluated as having low porting complexity, a greater emphasis was placed on projects that had higher numbers of likes on GitHub and were deemed more important and active. Additionally, efforts were made to collect projects from the bottom such as operating systems, kernel-level projects, mathematical computation libraries, and machine learning libraries.

## 2.3 Method

The RAX combines the CC calculation module and the Arch\_code location module to obtain the porting vector, and the complexity vector and score together form the data set.

**2.3.1 Lizard CC calculation scan function.** Lizard’s default computing method is basically consistent with McCabe’s theory [? ]. However, Lizard tool only detects defects at the function level and uses the average function CC for file-level statistics, which is not suitable for current porting complexity detection requirements [? ].

This paper proposes a new method for evaluating CC specifically for porting assessment work:

- (1) First, calculate the function-level CC for all files in the project using McCabe’s method.
- (2) Following Liu’s mention of System Cyclomatic Complexity (SCC) in software evolution assessment techniques based on code change detection [? ], we define SCC as the total sum of function-level values in the system. The advantage is that it can avoid the use of overly complex methods, which will cause deviations to affect the classification results. Additionally, due to the system’s complexity being calculated based on a single function, it eliminates the problem of decreased detection performance caused by the conversion of such a large code structure of all modules into syntax abstract trees and control flow graphs. Our CC calculation approach is expressed by the following equation, where lowercase cc represents the cyclomatic complexity at the function or file level. The uppercase CC represents the overall cyclomatic complexity of the software package, serving as a crucial factor in our tool. Here,  $|E|$  denotes the number of edges in the function control flow graph, and  $|N|$  represents the number of nodes. The function control flow graph is constructed based on branching structures such as if, while, and switch statements.

$$cc(\text{function}) = |E| - |N| + 2 \quad (1)$$

$$cc(\text{file}) = \sum_{i=1}^n cc(\text{function}) \quad (2)$$

$$CC = \sum_{i=1}^k cc(\text{file}) \quad (3)$$

**2.3.2 Arch\_code scan function.** Other tools only consider x86 architecture to locate architecture-related code for statistical analysis. We propose a new idea that explore whether multiple architectures can be selected or if other architectures can be individually targeted. Based on this, we conducted comparative experiments using various architectures, including X86, ARM, MIPS, RISC-V, SPARC, and PowerPC. We performed scanning and counting based on the keyword and function dictionaries specific to each architecture, which were derived from common patterns summarized from official websites. By analyzing the correlation coefficient (Pearson’s) between the keyword coverage line statistics of the six architectures and the score representing the porting complexity, we found that the correlation ranged from 0.44 to 0.58, indicating an average correlation across all architectures. Additionally, the correlation between the total values for all architectures and score was only 0.49. Based on these results, analyze that our tool try to effectively designed to assess the porting difficulty between two architectures. It is not as

**Table 2: Experimental results of models**

	Accuracy	Pmacro	Rmacro	F1macro
Adaboost	0.786	0.594	0.775	0.650
Random Forest	<b>0.857</b>	0.817	0.873	0.836

simple as adding up the modification amounts for different architectures, as it may only reflect an increase in code volume. Moreover, Due to the large differences in row\_based statistical results caused by the differences in code structure reflected by each architecture, this approach that easily considers all architectures simultaneously without considering the code structure cannot improve the accuracy of classification results and may lead to ambiguous outcomes. Therefore, based on these experiments, we still selected x86 as the architecture to build Arch\_code scan function, utilizing its keywords and critical function dictionary, given its mainstream position.

we form the Arch\_code dictionary: **Conditional compilation and architecture macros** Use conditional compilation statements, such as “if defined”, “ifdef”, in conjunction with all possible architecture macros, “\_\_x86\_64\_\_”. We use a similar bracket matching algorithm to cover all the code within macros, avoiding the impact of nested layers. **Assembly** We use matching forms such as “\_\_asm\_\_” and “\_\_volatile\_\_”. In future work, we will consider striving to achieve the currently unachievable architecture identification function of assembly code and identify assembly code in the form of instructions; **Intrinsic and Builtin** We compile dictionaries of Intrinsic functions based on official documentation from x86[? ]. Builtin function also employ a similar approach; In the future, we may assign difficulty weights to them. **System calls** Considering glibc’s support for system call encapsulation, reorganize the syscall dictionary. The system calls that do not support RISC-V have been filtered out currently. **Buildscripts** We locate architecture keywords in the build scripts.

### 3 RESULT AND ANALYSIS

To answer research question 2, based on the work of Chen Xiang et al. [? ], aiming at the status quo of unbalanced datasets and small sample datasets, we try to use ensemble learning methods [? ], SMOTE [? ], and random oversampling methods for data preprocessing. Ultimately, we chose to train four models: Random Forest, Support Vector Machine, Adaboost, and Xgboost. By comparing multiple indicators, we select the Random Forest machine learning as the classifier, as shown in Table ??

We continued to expand 100 projects as a supplementary data set, and the prediction results were determined through comments from the community and forums. Appendix presents a comparative assessment of the porting complexity evaluation tool developed by Du et al. across various projects, the comparison results are shown in Table ?? Through real-world evaluations within the developer community, our tool has exhibited a significant improvement in accuracy, surpassing other comparable tools by a margin of 15.6%. In the appendix, we provide a comprehensive compilation of the community’s valuable feedback comments, accompanied by the relevant web links for further reference and validation.

**Table 3: Tool effectiveness comparison from community feedback**

Porting Complexity	Du’s Tool		RAX	
	TP	FN	TP	FN
Low	32/37	5/37	35/37	2/37
Medium	5/8	3/8	6/8	2/8
High	2/6	4/6	6/6	0/6
Total	<b>39/51</b>	12/51	<b>47/51</b>	4/51

After adding the code complexity factor and modifying the original tool design model, RAX’s result prediction accuracy is improved. This is because code complexity can clearly distinguish three types of porting difficulty Motivation Chapters present the cause analysis. Adding factors also makes RAX more robust and more friendly to projects with unbalanced complexity vectors. For example, kernel-level projects, boot layers, hardware drivers and other projects may involve a large amount of assembly code, corresponding to larger porting complexity, but other factors of the original tool design are at low values, which is not conducive to distinguishing complexity. Chibios, u-boot are facing such a situation. Another reason is that we have added more medium or difficult complex samples to the data set to increase robustness and improve the innovativeness of the tool.

Within this assessment, there are instances of inaccurate predictions. In depth analysis of these variations reveals that certain projects exhibit inherent complexity, entailing a substantial presence of architecture-specific code defined by Arch\_code. Nevertheless, the practical porting process benefits from an array of auxiliary tools designed to support developers. In response to valuable suggestions from developers within the OpenEuler community, we conducted an evaluation of the porting difficulty for various versions of the Chromium. Initially, we were able to accurately assess the porting difficulty, yielding promising results. However, when it came to evaluating the work involved in iterative updates between versions, existing tools proved insufficient for precise estimation. Sometimes developers design their work has generic code for unknown architecture and CPU architecture parts are well separated from the common code, that just need only to adjust the macro to recognize RISC-V, like nuttx.

### 4 RELATED WORK

The development of the RISC-V ecosystem is rapidly progressing. In the realm of scientific computing, Tine Blaise et al. [? ] accomplished the porting of the OpenCL framework to RISC-V through a compiler framework, thereby expanding RISC-V’s accessibility to a wider range of scientific computing applications. In the context of architectural migration of foundational software and libraries, Cheng Yuanhu et al. [? ] designed an embedded multi-instruction set processor, addressing the software ecosystem challenges present in the RISC-V embedded domain. As a result, auxiliary efforts in porting are highlighted as valuable endeavors. This tool approaches code complexity from the perspective of its evaluation role in software evolution and maintenance, considering its influence on porting complexity assessment [? ]. Alvares et al. [? ] proposed using the



development team’s “tolerance” for code complexity to infer team development capabilities and performance indicators for projects. Vard et al. [?] emphasized that the internal quality of software impacts developers’ capabilities and maintenance durations. They conducted a comprehensive study on various existing code complexity metrics’ validity and suggested incorporating empirically observed code features as complexity triggers.

## 5 CONCLUSION

We propose RAX, with the aim of recommending projects of varying porting difficulties to developers. We incorporate the code complexity to reflect developers’ actual workload. We verified the effectiveness of the evaluation of 51 extension projects in real porting scenarios through their respective communities or forums. Observed significant performance improvements compared to previous tools. Moreover, RAX provides developers with the functionality to locate target code segments. Our next steps involve attempting text classification methods to categorize the text results of the scanning tool, designing more granular statistical methods for architecture-related factors.

## REFERENCES

- [?] [n. d.]. OpenCloudOS adds support for RISC-V. ([n. d.]). <https://baijiahao.baidu.com/s?id=1742039074702920281&wfr=spider&for=pc>
- [?] [n. d.]. RISC-V has officially become openEuler’s official support architecture. ([n. d.]). <https://forum.openeuler.org/t/topic/3110>
- [?] 2023. openEuler:riscv64:BaseOS:stage2. (2023). <https://build.openeuler.openatom.cn/project/show/openEuler:C>
- [?] 2023. porting complexity experimental evaluation. (2023). <https://doi.org/10.1109/TR.2013.2259203>
- [?] 2023. x86 Intrinsic. (2023). <https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html>
- [?] Vard Antinyan, Mirosław Staron, and Anna Sandberg. 2017. Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time. *Empirical Software Engineering* (2017).
- [?] Anish Athalye, Adam Belay, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2019. Notary: a device for secure transaction approval. In *Symposium on Operating Systems Principles*.
- [?] Rajiv D. Banker, Srikant M. Datar, Chris F. Kemerer, and Dani Zweig. 1993. Software Complexity and Maintenance Costs. *Commun. ACM* 36, 11 (1993), 81–94.
- [?] Marcos Alves Barbosa, Fernando Buarque de Lima Neto, and Tshilidzi Marwala. 2016. Tolerance to complexity: Measuring capacity of development teams to handle source code complexity. In *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. 002954–002959. <https://doi.org/10.1109/SMC.2016.7844689>
- [?] Gabriel Bessler, Josh Cordova, Shaheen Cullen-Baratloo, Sofiane Dissem, Emily Lu, Sofia Devin, Ibrahim Abughararh, and Lucas Bang. 2021. Metrinome: Path Complexity Predicts Symbolic Execution Path Explosion. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 29–32. <https://doi.org/10.1109/ICSE-Companion52605.2021.00028>
- [?] Andrea Capiluppi, Alvaro Faria, and J. F. Ramil. 2005. Exploring the relationship between cumulative change and complexity in an open source system evolution. In *IEEE*.
- [?] G. K. Gill and C. F. Kemerer. 1991. Cyclomatic complexity density and software maintenance productivity. *Software Engineering IEEE Transactions on* 17, 12 (1991), 1284–1288.
- [?] Haibo He and Edward A. Garcia. 2009. Learning from Imbalanced Data. *IEEE Transactions on Knowledge and Data Engineering* 21, 9 (2009), 1263–1284. <https://doi.org/10.1109/TKDE.2008.239>
- [?] LIU Huihui. [n. d.]. *Techniques of Evaluating Software Evolution Based on Code Change Detection*. Ph. D. Dissertation. Southeast University.
- [?] Du Jiman. 2023. Design and implementation of a RISC-V porting task planning system based on RPM source package dependency. (2023).
- [?] Wu JZ Liang GY, Wu YJ and Zhao C. 2020. Open Source Software Supply Chain for Reliability Assurance of Operating Systems. *Journal of Software* 31, 10 (2020), 18.
- [?] T.J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering SE-2*, 4 (1976), 308–320. <https://doi.org/10.1109/TSE.1976.233837>
- [?] Lourdes Pelayo and Scott Dick. 2007. Applying Novel Resampling Strategies To Software Defect Prediction. In *NAFIPS 2007 - 2007 Annual Meeting of the North American Fuzzy Information Processing Society*. 69–72. <https://doi.org/10.1109/NAFIPS.2007.383813>
- [?] Pasquale Davide Schiavone, Francesco Conti, Davide Rossi, Michael Gautschi, and Luca Benini. 2017. Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications. In *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*.
- [?] Blaise Tine, Seyong Lee, Jeffrey S. Vetter, and Hyesoon Kim. 2021. Bringing OpenCL to Commodity RISC-V CPUs. (6 2021). <https://www.osti.gov/biblio/1830102>
- [?] Shuo Wang and Xin Yao. 2013. Using Class Imbalance Learning for Software Defect Prediction. *IEEE Transactions on Reliability* 62, 2 (2013), 434–443. <https://doi.org/10.1109/TR.2013.2259203>
- [?] A Waterman, Y Lee, Da Patterson, and K Asanovi. 2014. The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Version 2.0. (2014).

**Temporary page!**

L<sup>A</sup>T<sub>E</sub>X was unable to guess the total number of pages correctly. As there was some unprocessed data that should have been added to the final page this extra page has been added to receive it.

If you rerun the document (without altering it) this surplus page will go away, because L<sup>A</sup>T<sub>E</sub>X now knows how many pages to expect for this document.