Architectural Guardrails: An Opinionated Framework for Preventing Structural Decay in Greenfield Research Software

Art O'Liathain
22363092@studentmail.ul.ie
University of Limerick

December 2025

Keywords Evolutionary Algorithms · Genetic Programming · Grammar Evolution · Symbolic Regression · Concurrent Computing

List of Figures

List of Tables

1 Abstract

2 Introduction

2.0.1 Research question

Can opinionated objective rules be derived from historical and static analysis on code repositories

Do these rules notably increase code quality

2.0.2 Thoughts on structure

The core idea of the debt analysis is a 3 step process Identify high risk/churn code areas, use a tool maybe machine learning to analyse and understand the code, derive rules from both texts such as architectural smells that are correlated to high *debt interest* [1]

3 Background on Identifying technical debt

- Current methods to identify debt
  - smells
  - machine learning
- Impact debt has on code in the long run
- How do the suggestions work
- AST and how they work
  - Ruff
  - Sonarqube etc
- (MAYBE) Talk about machine learning in pattern analysis for detecting high bug spots for the code and repeating issues.

3.1 What needs to be completed now

- Static Analysis methods
- Code churn detection

3.2 The current state of research software

Research software is interesting for a few different reasons, its software made to investigate new ideas and concepts. This comes with one very prominent issue, researchers who are leading the way to new concepts are generally not programmers by trade, 90% of scientists are self taught in programming [2]. This creates many issues, software is made without a care and it follows the simple check of *It works on my computer*. This is best observed by [3] where over 9000 R files were analysed from the Dataverse Project (http://dataverse.org/). In this study following good practice only 25% of projects were runnable and adding in code cleaning where dependencies were retroactively found and hardcoded paths were altered only 56% were able to be run. This is an abysmal result, this result doesn't even compare if the results are consistent with the results this code produced. [3] did note that journals are beginning to mandate better quality standards for code through research software standards to particularly allow for reproducibility to ensure the results published by the papers are correct.

It is not enough, the crux of the issue is that software is made to fit a funding cycle without a care for the longevity of the codebase. Funding is fundementally not suited for research software [4] notes that unlike commercial products where revenue grows linearly with users research grants are fixes and do not require or reward developers for user acquisition. Funding pushes research software to focus on itself with no incentives outside of a research paper output. A flawed approach which [4] does address with two alternative solutions; Commercial models in which the research software itself transitions to a self sufficient project which keeps itself alive and peer-production models which is a similar approach as open-source. I agree that both of these models have merit, they allow research software to live past the funding cycle and continue to improve over time, there is one key issue with this approach.

As 90% of researchers [5] don't have prior programming experience, they need good clear guidance on how to create and maintain a large software project. Unfortunately research software guidance are laughable as broadly they only focus on software reproducibility [2], [5], [6], [7], [8] (Need to reword this later). Reproducibility is considered such a simple part of industry software that it isn't considered a metric to hit, it is a given yet in research software is it laid out as being the metric for best practice. Interestingly with all the focus on reproducibility is no guidance on maintainability, coding standards or even basic structuring, the papers only if ever mention a vague sense of what should be done without and clear guidelines or criteria on how it should be done. A deficient that directly goes against the goals of [4] in creating long living research software.

This is the gap my paper is seeking to fill, to create an opinionated tool that will ensure code is written well. This is a vital tool for research software as maintainable software is extendable software which will allow research to build upon each other more quickly and easily contributing to more focus put on solving new problems rather than remaking an old one to then use.

4 Static Analysis

Static analysis in code is a storied field in which a primary focus has always been improving code quality. The tooling is imperative in modern engineering to allow for developers to see and fix maintainability and security issues in code. Unfortunately these tools are not prevalent in the research software space when it is most important as self taught developers do not have the mental model to identify the maintainability issue that plague codebases.

Before modern day static analysis using Abstract syntax trees, linters were at the forefront of code quality analysis [9]. Created for the C programming language the first linter was primary focused on an in the weeds analysis of code to identify errors cropping up syntactically and semantically in the codebase. Being the first of its kind it had limitations, it could only parse the code as a string and relied on regex to guess if the data flow allowed certain structures to be called. The tools available limited its uses but it was still able to identify key issues such as warnings regarding suspicious type conversions, non-portable constructs, and unused or uninitialized variables. Lint was a pivotal moment for static code analysis as it paved the way for subsequent static analysis tooling, it garnered wide use and even named a subsection of tooling “linters”. While lint is important, it is clear the limitations of static analysis using regex is too much to justify using it to analyse semantic structure in codebases and more sophisticated tools are required.

Abstract syntax trees are a concept

4.1 Code churn

Code churn is the rate of change of lines of code in a file. A file with high churn might have an incredibly high number of additions and deletions while remaining small. This is an ATD hotspot when assessing points of interest in terms of maintainability in codebases. [10] is paper which assess how code code churn affects the maintainability of codebases, by assessing the code churn of files in large codebases, achieving this by calculating the sum total of lines added and deleted on every file. Supplementing this the maintainability of the code was measured using ColumbusQM probabilistic software quality model [11]. The paper was able to assess the levels of maintainability per file in proportion to the amount of code changes, drawing the conclusion that code that has high churn is code in which it is harder to maintain. This conclusion makes sense, it reaffirms that poorly architect ed code will need more changes which increases the maintainability burden of the code. The method in which the code churn was measured is quite useful and will be the approach taken for this study.

Unfortunately there is one key issue with this paper, the use of ColumbusQM for assessing the maintainability of the codebase. ColumbusQM is a statistical machine which takes metrics such as lines of code, styling, coupling, number of incoming invocations etc.. Aggregates them which then computes a numerical output. The metrics such as styling are something so subjective and minor that the inclusion as one of the key metrics in the evaluation underscores the whole statistical model. Taking into account all of the evaluation metrics this is a case of blind assessment where structural relations or evolutionary dependencies are not measured and only a snapshot of the codebase is measured. Compounding on this the evaluation metric for the output of the statistical model was based on developer feedback and ideas, which is unreliable and subjective.

Taking these points into account, the key takeaway from [10] is the work on code churn particularly the methodology but the conclusions in regards to maintainability will only be used as an idea within this study.

4.2 Maintainability

Maintainability is term used frequently in software engineering, there is no definite definition on what maintainability is but ISO25010 defines it as "The degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements." It defines that the sub sections of maintainability are *Modularity, Re-usability, Analysability, Modifiability, Testability.* [12] This is a very broad definition which simply means how easy can the system be modified for change. Due to the broadness of the definition it can easily be used to bring other non functional requirements as sub requirements ie. Flexibility, Adaptability, Scalability, Fault tolerance, Learn ability. The lack of clear testable outcomes for each quality in the ISO standard leads to a conceptual overlap where maintainability becomes a 'catch-all' category for non-functional requirements.

4.2.1 What does it mean to be maintainable?

To label code as maintainable, modularity is a key factor in that assessment. Modularity done correctly facilitates changeability through logical decomposition of functionality. This may seem simple but this area if done incorrectly can have long lasting consequences on the maintainability of a codebase. D.L Parnas presented a study revealing how easily developers could fall into the pitfall of designing the system in a human like manner [13]. Parnas argued that designing systems following a flowchart of logical operations is a method that creates code that is not resilient and while modular is highly coupled which hurts the maintainability of the codebase. The core purpose of modularisation shouldn't be encapsulating subroutines into modules. Design decisions that are likely to change should serve as the foundational points to creating modular code, allowing subroutines to compositions of modules. This will serve as a foundational idea on how maintainability will be measured in this paper.

4.2.2 Measuring Maintainability

Because maintainability is an abstract concept, various frameworks have attempted to reduce it to a concrete numerical metric. An early approach in this area is the Maintainability Index (MI), which calculates a score based on a weighted combination of Halstead Volume, cyclomatic complexity, lines of code (LOC), and comment density [14]. The formula typically utilized for this assessment is:$MI = 171 - 5.2 \ln(V) - 0.23(G)$

- 16.2 ln(LOC) + 50 sin(sqrt{2.46 times C})Where V represents Halstead Volume, G is Cyclomatic Complexity, and C is the percentage of comment lines.

[14] validated this metric through feedback from 16 developers and empirical testing on industrial systems, including those at Hewlett-Packard and various defense contractors. This methodology provided a pragmatic tool for engineering managers to prioritize maintenance efforts by assigning a tangible value to code quality. While the MI provides a high-level overview of code density, it suffers from what can be described as "semantic blindness." Metrics such as Cyclomatic Complexity and Halstead Volume analyze the control flow and token count of a file but fail to interpret the structural intent or the relationships between components. This is referred to as blind metrics and is a deficiency in this method. Utilising blind metrics allow for efficient and quick but it opens the door to gaming the system in a sense, where developers could reach extremely maintainable scores syntactically while semantically being unmaintainable, thereby abusing the formula. An example of this is a script that maintains a high MI score due to low complexity and short length, yet contain architectural flaws such as "God Objects" or tight coupling to external datasets that inhibit reuse. The formula was a product of its time finely tuned to the teams and projects it was applied to, in the modern landscape the current formula would definitely not be accurate with how modern languages have evolved. Even though the formula could be adapted to a modern context it is clear that even a modernised version would suffer from blind metrics and is not something that should be used as only counting lines cannot yield results on the true quality of code.

An approach that would apply in a more modern context was proposed by Xiao et al, of identifying architectural debt through evolutionary analysis of a codebase [1]. Where architectural debt can be used as a measure of maintainability albeit without a clear score. The paper proposed that there are four key architectural patterns that are the main proponents of ATD, Hub, Anchor Submissive, Anchor Dominant, and Modularity Violation. These patterns are all based on evolutionary dependencies between files, particularly those that are correlated only through commits and lack structural commonality.

Explain briefly the four types with images if possible

This was measured by preforming a pseudo longitudinal study on large open source codebases, using tools such as Understand and Titan to calculate commit coefficient between every file. A metric which relates to the chance of a commit on one file will require a commit on another. An example would be given files A, B and C, with a commit history of {A,B} and {A,C} the commit coefficients in relation to A are there is a 100% chance if B or C is modified A will be also, but if A is modified there is only a 50% chance that B or C will be modified. This would then be called a fileset, this would be extrapolated over the entire commit history and codebase creating many filesets. This would allow the tool to extract semantic relations between files that are not apparent structurally. The study was able to compound this effect by measuring the number of commits labeled as bug fixes against the number of feature commits, this allows simple data analytics to measure the amount of maintenance debt that each fileset would have. Using this method, high maintenance filsets can be labelled and evaluated as a quantitative metric. This is a strong contender for a evaluation metric but there is a caveat, that is it is reliant on good commit messages and issue tracking which is not commonly seen in research code. Meaning that a new approach that is agnostic to the quality of the codebase is required to.

4.3 Technical Debt

Technical debt (TD) was first defined as "Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite… The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise." By Ward Cunningham [15]. In the following years research has shown TD is not a singular type of problem and there are many forms to it, the five most prevalent types of TD are **Design debt, Test debt, Code debt, Architecture debt and Documentation debt** this was extracted from 700+ surveys across 6 countries [16]. The artifacts used to identify design debt, code debt and architectural debt have significant overlap and these artifacts exhibit behaviours similar to Architectural Technical Debt (ATD)[1]. Similarly to Maintainability, this paper defines Technical debt in regards to design, architecture and code as ATD and will be the primary focus of this paper.

Technical debt is a pervasive problem in research software development [17] where "does it compile" is the only quality metric tracked in codebases. This means that technical debt accrues significant interest as time increases.

4.4 Dealing with technical debt

When dealing with technical debt there are two
- How to identify with smells
- How do auto suggestions work look at tools

Bibliography

[1] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng, "Identifying and quantifying architectural debt," in *Proceedings of the 38th International Conference on Software Engineering*, Austin Texas: ACM, May 2016, pp. 488–498. doi: 10.1145/2884781.2884822.

[2] G. Wilson *et al.*, "Best Practices for Scientific Computing," *PLoS Biology*, vol. 12, no. 1, p. e1001745, Jan. 2014, doi: 10.1371/journal.pbio.1001745.

[3] A. Trisovic, M. K. Lau, T. Pasquier, and M. Crosas, "A large-scale study on research code quality and execution," *Scientific Data*, vol. 9, p. 60, Feb. 2022, doi: 10.1038/s41597-022-01143-6.

[4] J. Howison and J. D. Herbsleb, “The sustainability of scientific software:.”

[5] G. Wilson, “Software Carpentry: Getting Scientists to Write Better Code by Making Them More Productive,” *Computing in Science & Engineering*, vol. 8, no. 6, pp. 66–69, Nov. 2006, doi: 10.1109/MCSE.2006.122.

[6] R. C. Jiménez *et al.*, “Four simple recommendations to encourage best practices in research software.” Accessed: Feb. 07, 2026. [Online]. Available: https://f1000research.com/articles/6-876

[7] B. Marwick, “Computational Reproducibility in Archaeological Research: Basic Principles and a Case Study of Their Implementation,” *Journal of Archaeological Method and Theory*, vol. 24, no. 2, pp. 424–450, June 2017, doi: 10.1007/s10816-015-9272-9.

[8] S. J. Eglen *et al.*, “Towards standard practices for sharing computer code and programs in neuroscience,” *Nature neuroscience*, vol. 20, no. 6, pp. 770–773, May 2017, doi: 10.1038/nn.4550.

[9] S. Johnson and M. Hill, “Lint, a C Program Checker,” 1978. Accessed: Feb. 08, 2026. [Online]. Available: https://www.semanticscholar.org/paper/Lint%2C-a-C-Program-Checker-Johnson-Hill/74617cffa3c6438d04aa99bef1cca415de47d0d3

[10] C. Faragó, P. Hegedűs, and R. Ferenc, “Cumulative code churn: Impact on maintainability,” in *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Sept. 2015, pp. 141–150. doi: 10.1109/SCAM.2015.7335410.

[11] “(PDF) A probabilistic software quality model,” in *ResearchGate*, doi: 10.1109/ICSM.2011.6080791.

[12] “ISO 25010.” Accessed: Feb. 02, 2026. [Online]. Available: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

[13] D. L. Parnas, “On the criteria to be used in decomposing systems into modules,” *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972, doi: 10.1145/361598.361623.

[14] P. Oman and J. Hagemeister, “Metrics for assessing a software system's maintainability,” in *Proceedings Conference on Software Maintenance 1992*, Nov. 1992, pp. 337–344. doi: 10.1109/ICSM.1992.242525.

[15] “c2.com/doc/oopsla92.html.” Accessed: Feb. 02, 2026. [Online]. Available: https://c2.com/doc/oopsla92.html

[16] R. Ramač *et al.*, “Prevalence, common causes and effects of technical debt: Results from a family of surveys with the IT industry,” *Journal of Systems and Software*, vol. 184, p. 111114, Feb. 2022, doi: 10.1016/j.jss.2021.111114.

[17] Z. Hassan, C. Treude, M. Norrish, G. Williams, and A. Potanin, “Characterising reproducibility debt in scientific software: A systematic literature review,” *Journal of Systems and Software*, vol. 222, p. 112327, Apr. 2025, doi: 10.1016/j.jss.2024.112327.