# Understanding Cyclomatic Complexity

Jim Fitzpatrick

May 13, 2024

## Contents

## List of Figures

## 1 Introduction

This paper looks into the cyclomatic complexity in the Kuadrant project. Cyclomatic complexity is a metric design to indicate the complexity of an application. In the Kuadrant project, there are multiple applications, and combining these metrics gives some interesting insights.

The main goal is to learn what cyclomatic complexity is and how best to apply it to a project. Using the Kuadrant project is a means to an end. The project has a large enough code base spread across multiple repos and languages to be interesting.

### 1.1 Kuadrant Project

The Kuadrant Project is a collection of applications that provides gateway policies for kubernetes and cross multiply clusters. There is more to the project than its tag line and the components that are shipped with every release. Other repos add value to the project, such as the test suite and website repos.

## 2 Understanding Cyclomatic Complexity

Cyclomatic complexity[1] is a measurement created from the control flow of an application. It uses the number of edges and nodes in the control flow graph along with the number of connected components. The common formula is M = E - N + 2P, where

- E = the number of edges of the graph.
- N = the number of nodes of the graph.
- P = the number of connected components.

There are other formulas depending on if the graph is a strongly connected graph but for this scenario those do not matter.

The simplest understanding of how to calculate this metric is every time the function makes a choice the one gets added to complexity and a function has a base level complexity of one. Below are two example functions that produce the same output and both a cyclomatic complexity of two. But the structure of each function is different.

The above examples show that readability may have no effect on the cyclomatic complexity and by that point, the style of a language or a team of programmers should not affect the overall metric. What will affect the metric is how the tooling authors interpret the language control flow features. For example, a switch statement with cases that fall through to the next case can be counted as one or the total number of cases that are passed through.

As the Kuadrant project uses multiple languages, we can only use the metric from a high-level, trending viewpoint. Scores across multiple languages should not be compared with each other.

[1] https://en.wikipedia.org/wiki/Cyclomatilc_complexity

| **CC Score** | **Rank** | **Risk** |
|---|---|---|
| 1 - 5 | A | Low - simple block |
| 6 - 10 | B | Low - well-structured and stable block |
| 11 - 20 | C | Moderate - slightly complex block |
| 21 - 30 | D | More than moderate - more complex block |
| 31 - 40 | E | High - complex block, alarming |
| 40+ | F | Very high - error-prone, unstable block |

Figure 1: Table grouping cyclomatic complexity scores.

### 2.1 Classifying Cyclomatic Complexity

The metric itself is a number but as a single number, the metric is hard to reason about, so it is bucketed into categories. While researching this work, I came across a number of different bucketing systems. I will use the bucket classification found on radon.readthedocs.io, it gives a good range at both ends of the spectrum. These classifications can be seen in **Figure 1**. This ranking will be used and from a high level you want scores in the ABC grouping and not many Fs.

## 3 Case Study: Cyclomatic Complexity in Kuadrant

### 3.1 Data collection and analysis

As the Kuadrant project is an open source project the analysis was only done on the public repos. Any publicly archived repo was excluded along with any repo forked by the Kuadrant organisation.

#### 3.1.1 Tooling

To analysis the cyclomatic complexity of each repo, the following tools were used:

- gocyclo
- rust-code-analysis-cli
- RuboCop

These three tools covered five of the languages found in the project.

- Go
- Rust
- Python
- JavaScript
- Ruby

In total, 34 languages were detected in the project repos. The full list can be found in section 3.5.2. To collect the different languages scc was used. This tool will break down the lines of code in a project. Which was helpful to graph how much of the projects are being analyzed by the different cyclomatic complexity tools.

### 3.1.2 Excluded code

Some projects had third party vendor code saved within. This vendor code was primarily present in the early days of the projects. These third party blocks have been excluded from the analysis. Their results skewed graphs with high peaks that did not add any value to the overall outcome.

### 3.1.3 Choosing Temporal Data Points

To select the temporal data points which each repo was scanned and analyze the merge commits were used. In the case of the kuadrant-operator repo directly, the method of merging PRs changed late 2022 and stopped adding the merge message to the git log. In this case, every commit after the latest merge was scanned and analyzed. This process of checking the commits after the latest merge commit was applied to every repo that was scanned.

### 3.1.4 Processing the data

All this data is collected with a notebook to allow for reproducible collection. The results are normalized and stored in a Json file. The results need to be normalized as the cyclomatic complexity tools all have different outputs. The collection process is currently not additive and requires a collection of data from the being of repos every time.

A second notebook provides the classification and visualizations of the collected data. This only requires the Json file.

## 3.2 Inital Assumptions

In the beginning when starting to think about cyclomatic complexity within Kuadrant, assumptions were made about what the data would show. Understanding these assumptions can give an insight into how the outcome from this research was made.

### 3.2.1 Applications V's Operators

The applications are Authorino and Limitador. The operators are all the other operators and controllers, not just the operators related to the applications. The assumption is the applications will have a much higher CC score but have less high-ranked scores. The reconcile loops can have very high cyclomatic complexity scores.

### 3.2.2 Language Difference

The three main languages that triggered this assumption cannot be more different in how they are written. These languages are Golang, Rust and Python.

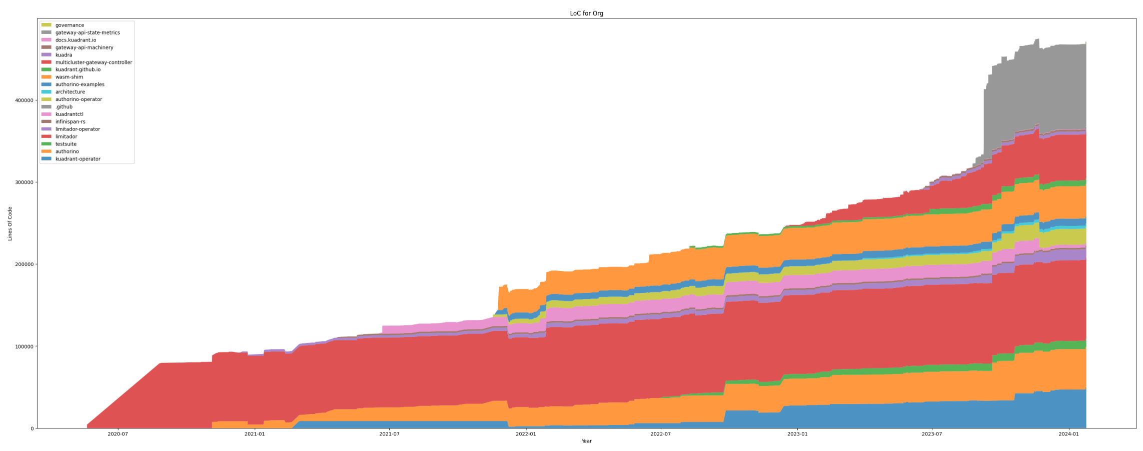

Figure 2: Lines of code for all Kuadrant projects.

Golang is the primary language in all the operators and controllers. As Golang treats errors as values, the code bases are littered with 'if err != nil' and this adds choices to every function that will affect cyclomatic complexity score. This feature means developers are forced to deal with the errors, but any argument that the CC score is too high in functions needs to take into account errors as values will inflate the CC score.

Rust on the other hand is different. Its compiler does a lot of error checking that Golang just does not. This comes from features like the Borrow Checker, it's much harder to get nil pointers, so it should be safe to have a code base with no nil pointer checks. This language safety should lower the cyclomatic complexity score for its functions. However, the mental loops a developer needs to jump through in Rust will be higher because of features like the Borrow Checker.

Python being used in the test suite repo and not compiled to a single binary. Also does not have strict types and does not have errors as values like Golang. It also did not have a 'switch' like statement until a few years ago. It is the language I expect to have the lowest-ranking scores of the three different languages.

## 3.3 Lines of Code within the Kuadrant Project

An overall size of the Kuadrant project is required to know how much effect the cyclomatic complexity has over the project. The first chat, **Figure 2**, is interesting as it shows the LoC for every repo. It is interesting to see how and where the Kuadrant project grow over time.

Now to see the amount of the project code base that was scanned by the cyclomatic complexity tools, **Figure 3**.

As seen there is a very small amount of the LoC covered by the scan tools. A lot of the projects are for working on kubernetes which relies heavily on YAML which we generate. This could be a good explanation for the differences. The one thing to note on the scannable code is the rate of change over time, while always increasing there are no steep increases.

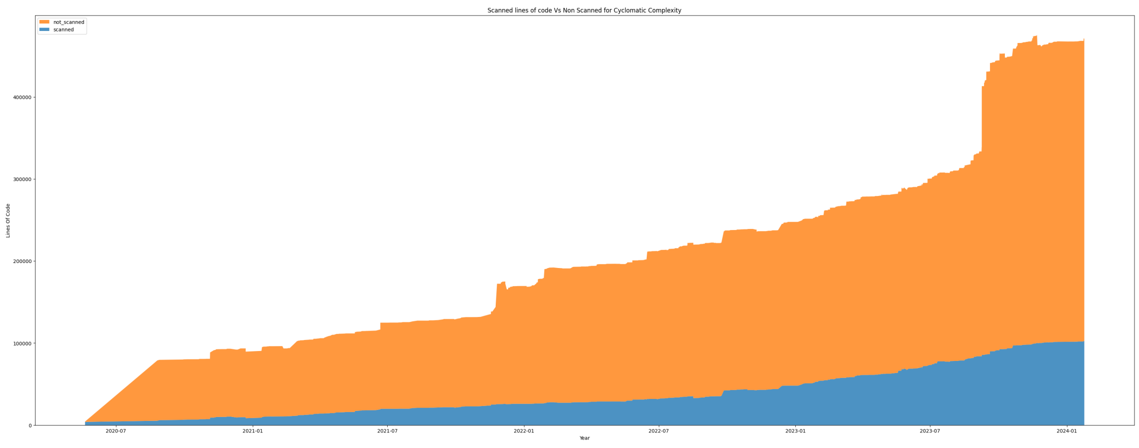

Figure 3: Lines of code in the Kuadrant projects scanned by the cyclomatic complexity tools.

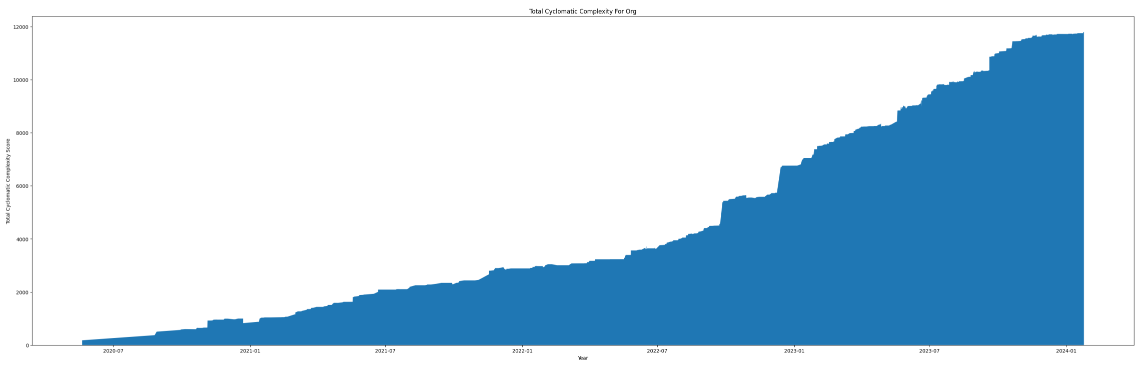

Figure 4: cyclomatic complexity for the source code

## 3.4 Diving into the Complexity

Now that the scale of the scanned code base, what does the cyclomatic complexity of that code base look like?

In **Figure 4** the cyclomatic complexity is scored overtime. The main takeaway from this is how the project as a whole is nearly doubling in complexity every year. A new user joining the project a year ago would have found it much easier to onboard. When the data is ranked, it shows a different picture.

In **Figure 5**, by far the largest Rank is the A group which is at least two thirds the overall

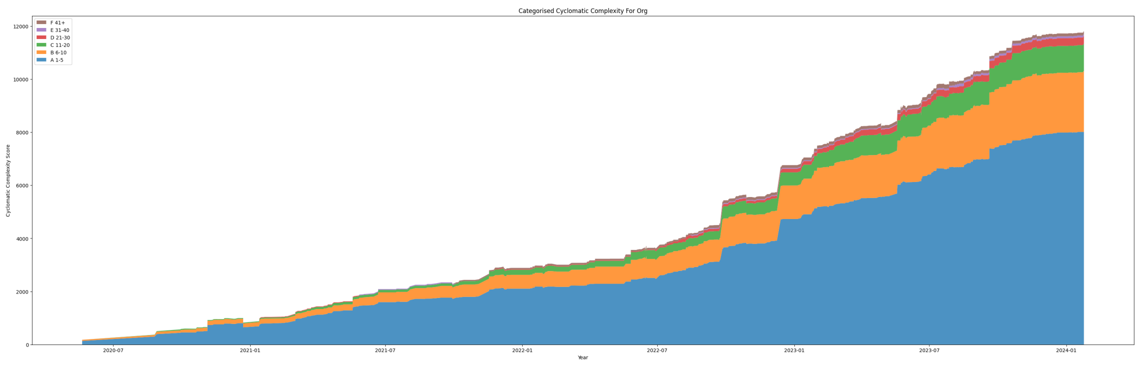

Figure 5: cyclomatic complexity score ranked

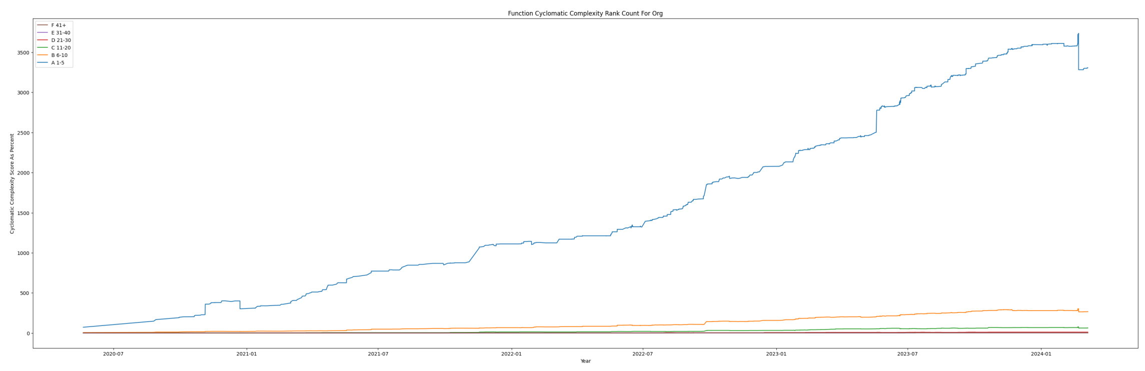

Figure 6: Count of functions in each Rank for the Kuadrant Project

score. Over the history of the project, this has always been the largest group. For new users, this means most functions are simpler to understand the logic, but it does mean there are a lot more interfaces / functions that need to be learned.

Forgetting about cyclomatic complexity as a sum but looking at the number of functions in each Ranking, we see how different the scale between each rank. In **Figure 6** the number of Rank A functions out scales all other ranks. New users would be required to understand how these Rank A function's tide together.

Only two components in the Kuadrant project have the Rank A grouping less than 50% of the overall ranking (authorino-operator, multicluster-gateway-controller). Other components have some other interesting aspects that well be covered. Not all graphs for each component will be covered here, as there are too many and the data will be out of date quickly. For that reason, the notebooks that generate all the charts are located here.

### 3.4.1 The wasam shim project

The wasam shim shows a nice outcome from looking at the cyclomatic complexity, it shows how the project evolved over time. In **Figure 7** shows how the project grows in complexity over time but also that it can be refactored. For some time, there were Rank E functions in the code base, which possibly replaced some Rank D functions. However, now all the Rank E and D functions have been written out, but the over all cyclomatic complexity has not changed much.

A project's complexity will increase over time, and the Ranking of that complexity can also increase, but does not mean it cannot be refactored out at a later date. This would also point to adding CI checks to ensure the cyclomatic complexity is kept below some arbitrary number might harm productivity within the project over time.

The pattern of refactoring out these complex functions from different Rankings can be seen in other components, but the wasam shim has the cleanest graph for showing this. While it might be hard, we can make changes to refactor the API without changing the overall complexity of the component.

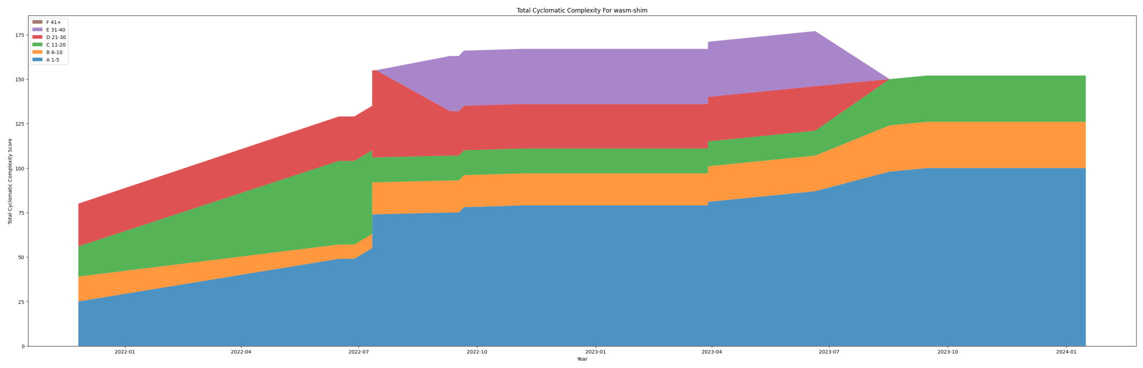

Figure 7: Total cyclomatic complexity over time for the wasam shim

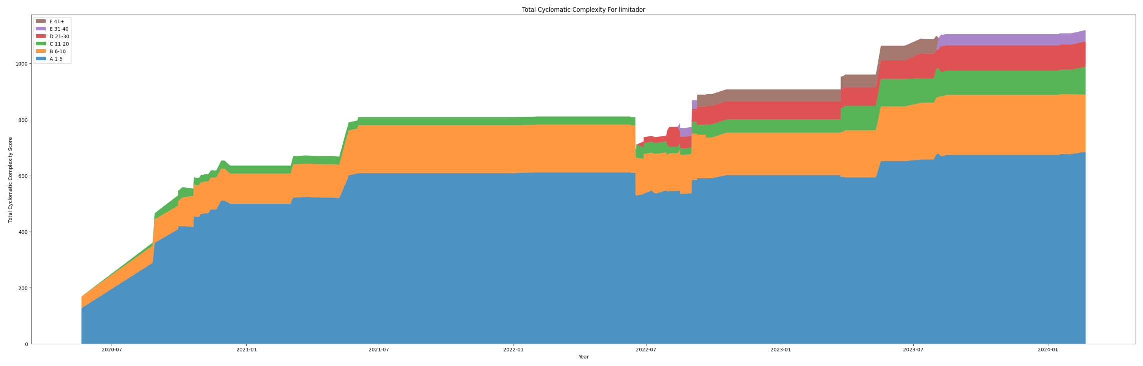

Figure 8: Total cyclomatic complexity over time for limitador

### 3.4.2 The limitador project

Limitador was one of the projects that were going to be interesting as it is written in Rust, and one of the assumptions made was about Rust. This showed a very surprising turn around, one that I was not expecting. As a reminder, the assumption was Rust code because of its compile checks and language features would lead to functions that naturally would have a lower cyclomatic complexity score.

However, that is not the case. In **Figure 8** the Ranking of the cyclomatic complexity changed over time, going to from lower Ranks to higher Ranks. Around July 2022 there was a change to the project, and this change showed an increase in the cyclomatic complexity . This was the joining of new members to the project. New members can have their own views on what complexity can and should be. If there is a belief in deep interfaces over wide interfaces, the cyclomatic complexity score will naturally be higher.

This shows that the cyclomatic complexity is as much affected by team culture and design philosophy as language features and project complexity.

### 3.4.3 The testsuite repo

The testsuite repo was always going to be an interesting component of the Kuadrant project. It can't be called a component as it is never shipped to the user, it is our test suite for our shipped components. There is also the fact it is not written by the core engineering team but quality engineering, who are always understaffed and has other focus. The other outlier is the

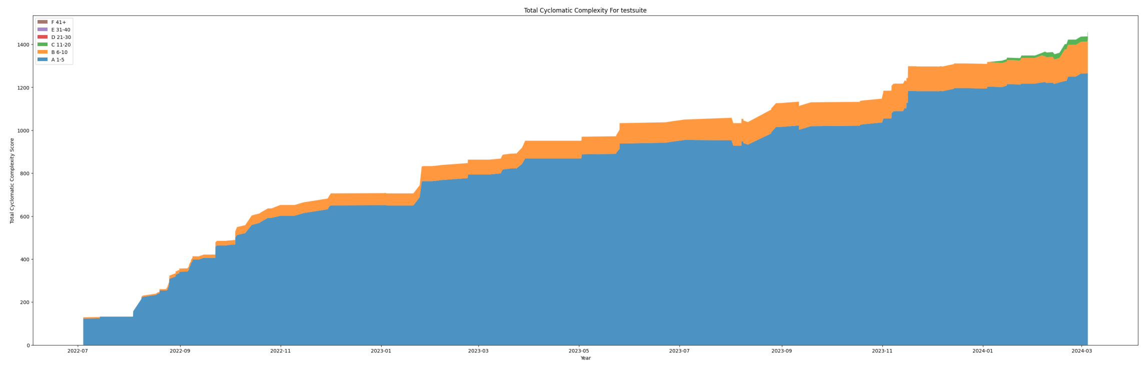

Figure 9: Total cyclomatic complexity over time for testsuite.

repo is written in Python. Python falls into the language assumption that cyclomatic complexity of functions would be less as Python duck typing allows the writing to get away with avoiding complexity.

What was not expected was the shear scale of the difference between the testsuite repo and every other repo in the Kuadrant project. **Figure 9** shows the scale of the differences. Nearly 90% of all functions are in the Rank A grouping. This difference is so large that I wanted to know what the breakdown of the Rank A function. The current state of the repo can be seen in the pie chart, **Figure 10**.

While this does align with the assumption made in the start, but seeing these numbers raised a concern for me. How do new developers understand this repo? Hopefully, the documentation for the repo explains how to combine all the functions to get the required result.

But is there more going on here? This is a testsuite that uses pytest to run all the tests, so it is not being run by Python in the normal sense. In pytest the *assert* keyword is used to create any checks, which should not be used in production Python. The *assert* keyword can be disabled when using Python directly, this may mean the tool for calculating the cyclomatic complexity does not count the *assert* keyword, and the project is far more complex than expected. This was very much the case. Looking at some of the near 500 functions that have a cyclomatic complexity score of 1, a lot have many *assert* keywords with in the one function. The cyclomatic complexity for the whole project is much higher than the tooling is leading us to believe, and not just that some functions are underreported.

### 3.4.4 The multicluster-gateway-controller project

Looking at the history of a project can revel change over time. Apart from the normal growth of a project, historical events can be detected. This can be events where features were added or removed. These events normally show as jumps in the graphs over being a spike in the graph.

No project shows a better example of this than the multicluster-gateway-controller, as seen in **Figure 11**. This relates to the removal of the DNS and TLS controllers from the project. While these features are being refactored into the kuadrant-operator, it is still nice to see such changes. In this case, it was a massive removal change that is easy to spot. In very project, you can see a number of spikes that would be the result of new features being added to a project.

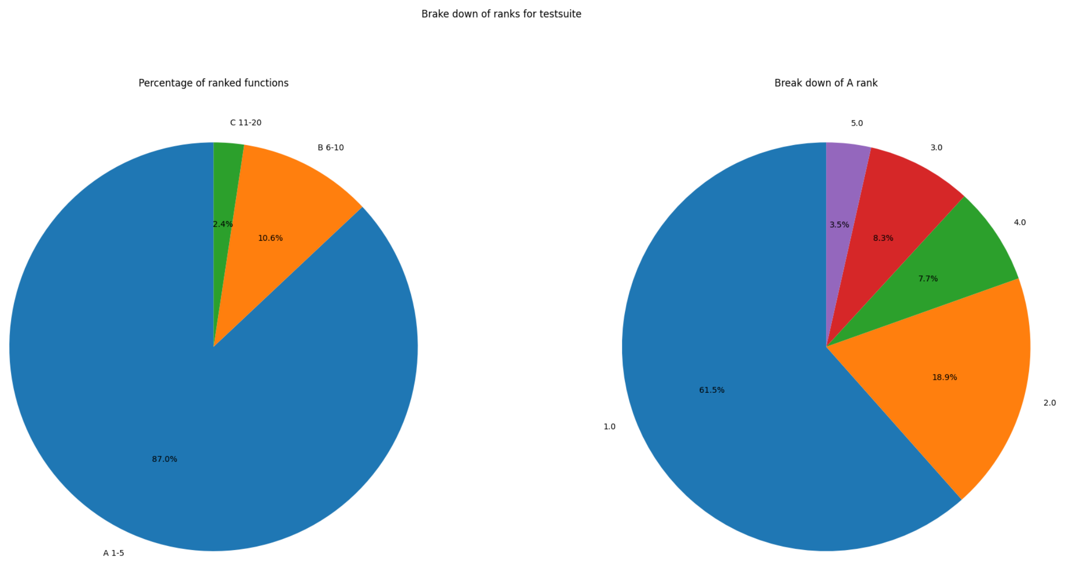

Figure 10: Pie chart of testsuite rankings.

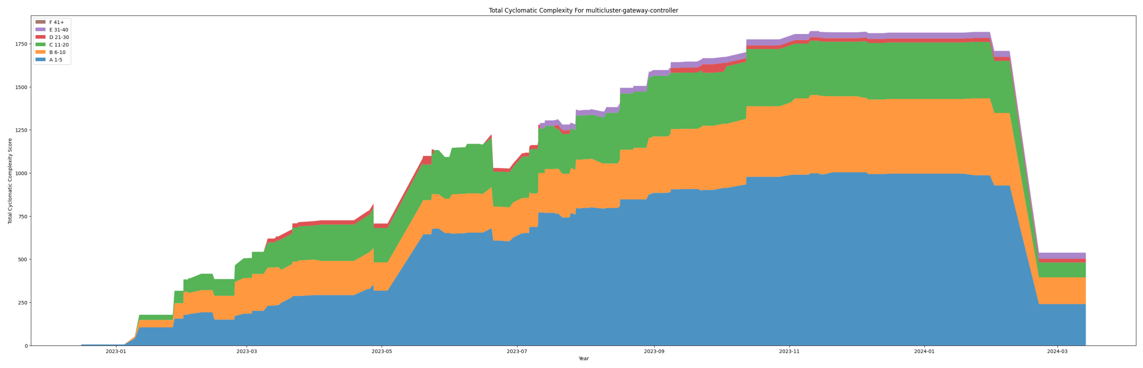

Figure 11: Total cyclomatic complexity over time for multicluster-gateway-controller.

## 3.5 Reference

### 3.5.1 Projects

| **Project** | **First Merge** |
|---|---|
| limitador | 2020-05-20 |
| authorino | 2020-11-04 |
| limitador-operator | 2029-12-15 |
| kuadrant-operator | 2021-03-01 |
| infinispan-rs | 2021-04-20 |
| kuadrantctl | 2021-06-21 |
| authorino-operator | 2021-11-17 |
| authorino-examples | 2021-11-23 |
| wasm-shim | 2021-11-25 |
| testsuite | 2022-07-04 |
| kuadrant.github.io | 2022-08-08 |
| multicluster-gateway-controller | 2022-12-16 |
| kuadra | 2023-06-13 |
| gateway-api-machinery | 2023-06-20 |
| docs.kuadrant.io | 2023-07-05 |
| .github | 2023-07-26 |
| gateway-api-state-merics | 2023-08-24 |
| governance | 2024-01-18 |

### 3.5.2 Languages

- CSS
- CloudFormation (YAML)
- Docker ignore
- Dockerfile
- Extensible Stylesheet Language Transformations
- Gemfile
- Go
- Go (gen)
- HTML
- Handlebars
- JSON
- JavaScript
- License
- Makefile
- Markdown
- Plain Text
- Plain Text (min)
- Protocol Buffers
- Python
- Rakefile
- Ruby
- Ruby (gen)
- Ruby HTML
- Rust
- SQL
- SVG (min)
- Sass
- Shell
- Smarty Template
- TOML
- XML
- YAML
- YAML (min)
- gitignore

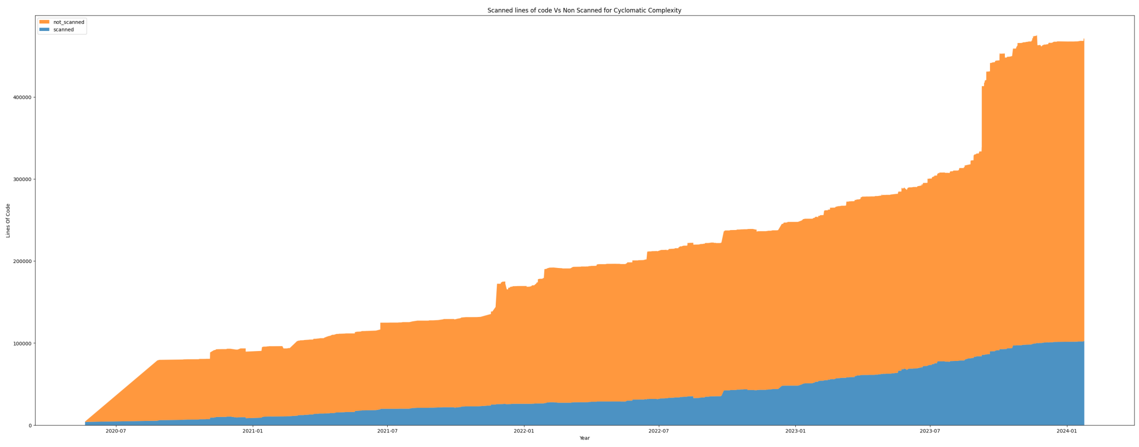

Figure 12: Go function with a cyclomatic complexity of 3

## 4 Using the cyclomatic complexity metric

The cyclomatic complexity metric can be used to guide the development within the Kuadrant projects. But it cannot be used as a metric that to be shown on some dashboard. It is not even a metric that should be tracked overtime. The two main usages I have seen do not require the tracking overtime.

### 4.1 Guide for testing

Reading many articles on cyclomatic complexity, one of the use cases that I came across was using the metric for writing unit tests. The idea is that you should write at least the same number of unit tests for a function as the cyclomatic complexity metric. This metric will not cover all the possible iterations of a function or the edge case that need to be tested, but do give a good starting point.

To explore this concept, there is a sample function that gives time stamps a weight depending on when the time stamp is. If we look at the function defined in **Figure 12** we can see that the cyclomatic complexity metric is 3, as there are two *if statements* and a *return statement*. Currently, four unit tests would cover every possible outcome from that function. The function makes a decision on weather the give timestamp is a weekend day and if the time is after noon.

Let's make this code more complex, let's change the weighting base on if there is a *"r"* in the days' name. This change can be seen in **Figure 13**. The interesting thing about this code is the cyclomatic complexity metric is now 4, but it requires eight unit tests to cover all possible outcomes of the function. It shows that as the cyclomatic complexity metric goes up, the number of unit tests can explode.

The other interesting thing about the function in **Figure 13** is how code coverage is affected. You can achieve 100% code coverage with using only two test cases. This falls far below the testing all possible outcomes, but it makes the metric watchers happy.

So the cyclomatic complexity metric gives a developer of how many test cases they should be writing for a function to get a good coverage of the possible outcomes in the function. While at the same time, not being crazy in the number of tests that is required.

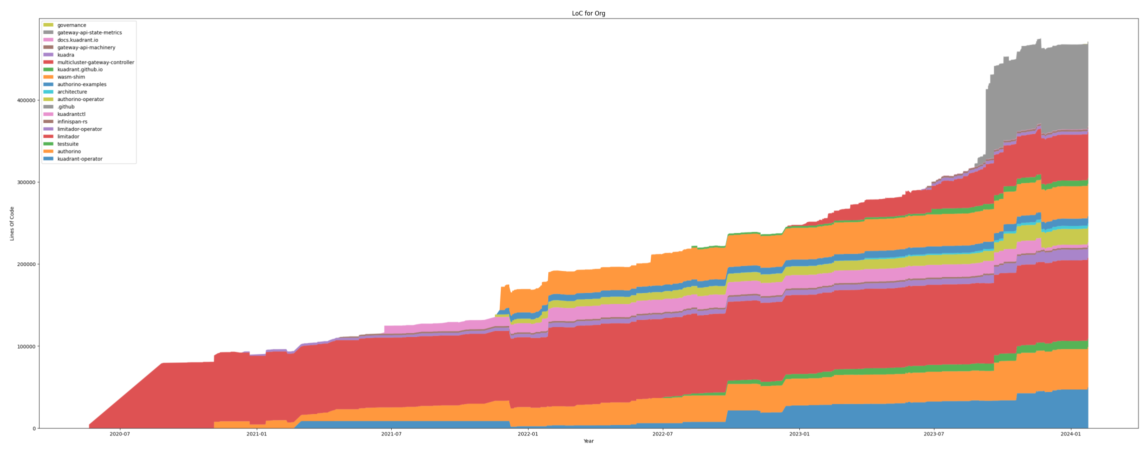

Figure 13: Go function with a cyclomatic complexity of 4

#### 4.1.1 The 10th September 1752 problem.

I bring up this problem as it shows how the cyclomatic complexity metric cannot answer many questions a developer many need to be asking about their code. On the 10th September 1752 there are many places in the world that you could have been. But you could not have been in most areas in Canada or the United States or the United Kingdom and its colonies because that date did not exist. This was the time these places switched from the Julian Calendar to the Gregorian Calendar, a process that took 300 years around to complete[2]. With the last happening in Turkey at the start of 1927.

The example functions above many handle these dates but how can we be sure. If this code needs to be location-aware and allow for times in the past, then these are edge cases that could happen. The responsibility is still on the developer to know what should be tested. No metric, not even cyclomatic complexity will help with identifying edge cases that need to be tested.

## 4.2 Aiding system design

Using cyclomatic complexity to aid in system design will depend somewhat on the school of though around what scores are acceptable. If you are a follower of the *"Clean Code"* teaching were functions should be short, then the cyclomatic complexity metric shows what functions need refactoring to lower their metric score. But if you believe interfaces should be deep and not shallow, then the metric can show you functions that are really not doing much but require developers to know of their existing.

The tool used to calculate the cyclomatic complexity score in Ruby by default only showed results with a score of seven or higher. It even went as far as to return a none zero result. This shows the authors of the tooling were opinionated about what score functions should have. An opinion that would align with the teachings in the *Clean Code*.

My personal opinion is functions should try to be in Rank B (6–10 range). This would greatly reduce the number of APIs that a developer would be required to learn. Yes, there can be other ranked functions, and there should be. A reconcile loop function should have many checks which

[2] Gregorian Calendar Reform: Why Are Some Dates Missing?

drive the cyclomatic complexity score upward. But the idea of having wrapper functions to reduce the cyclomatic complexity score of one function seems wrong.

## 5 Missing Information

With looking into cyclomatic complexity and how to use the metric, it becomes clear there is some information that would be added to the usefulness. The idea of code churn which would highlight functions within a project could show if

The churn of code in a function could highlight functions that are changing, and if these functions have a high cyclomatic complexity score, would indicate a code area that should be addressed.

## 6 Conclusion

The conclusion of this research into cyclomatic complexity is as follows. While the cyclomatic complexity scores can show how large a project is, it will not say anything about how well it is designed. It is a good metric for identifying areas within a code base that maybe too complex, and should have a refactor done. The inverse is also true, it can identify areas that are overly simple requiring developers to hold more information in their heads.

Begin able to calculate the cyclomatic complexity manually in your head can be useful as a guide to how many test cases a function should be created. The test cases should still be meaningful, and not created to just please some metric on a dashboard.

As the testsuite showed know how the projects are run and how the tooling calculates cyclomatic complexity scores can change the results greatly. This means there may be other ways to trick the metric into lower results at the cost of creating less stable software.

Tracking the metric over time is also worthless, as the project grows, so does the cyclomatic complexity and therefore the metric grows. Where the over time metric can be used is reminding existing develops how much more complex the project is for new developer's start. The onboard time for a new developer may take longer, and any existing developer may require more patience with the new developer. Tracking the Ranks proportional over time may have a benefit of identifying when a project is starting to slip and get "untidy." That is if a project can agree on a proportional ranking score.

Overall, it is a nice metric, just like code coverage, is a bit meaningless but can help drive decisions on a case by case bases. It can quickly identify complex functions that may need a refactor, ranks D and above. For public facing APIs, it can help identify area that many be too broad requiring a developer to hold more context than required to use the API.