Understanding Cyclomatic Complexity

Jim Fitzpatrick

March 7, 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 complexity of an application. In the Kuadrant project there are multiple applications and combining these metrics give some interesting insights.

1.1 Kuadrant Project

The Kuadrant Project is a collection of applications that provides gateway policies for kubernetes and cross multi cluster.

2 Understanding Cyclomatic Complexity

Cyclomatic complexity 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 cases that are passed through.

As the Kuadrant project uses multiple languages we can only use the metric from a high level, trending viewpoint.

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 classificataions can be seen in **Figure 1**.

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.

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. To analysis the 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 the Languages section of the Reference. To collect the different languages scc was used. This tool will also give a breakdown of the lines of code in a project. Which was helpful to graph how much of the projects were being analysed by the different cyclomatic complexity tools.

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.

To select the points in time which each repo was scanned and analyse 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 analysed. This process of checking the commits after the latest merge commit was applied to every repo that was scanned.

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

A second notebook provides the classification and visualisations 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 outcomes 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 higher ranked scores. The reconcile loops can have very high CC scores.

3.2.2 Language Difference

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

Golang of course being 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 which 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 is safe to have a code base with no nil pointer checks. This language safety should lower the CC 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.

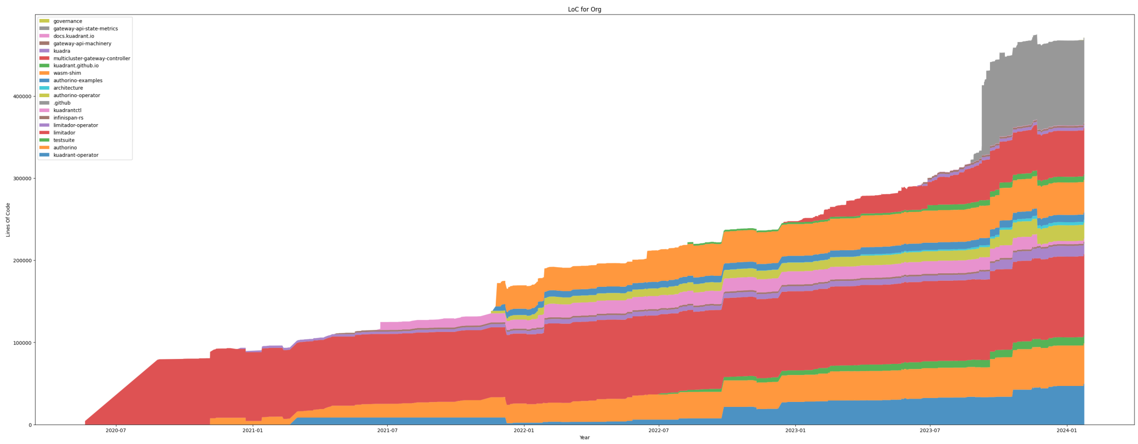

Figure 2: Lines of code for all Kuadrant projects.

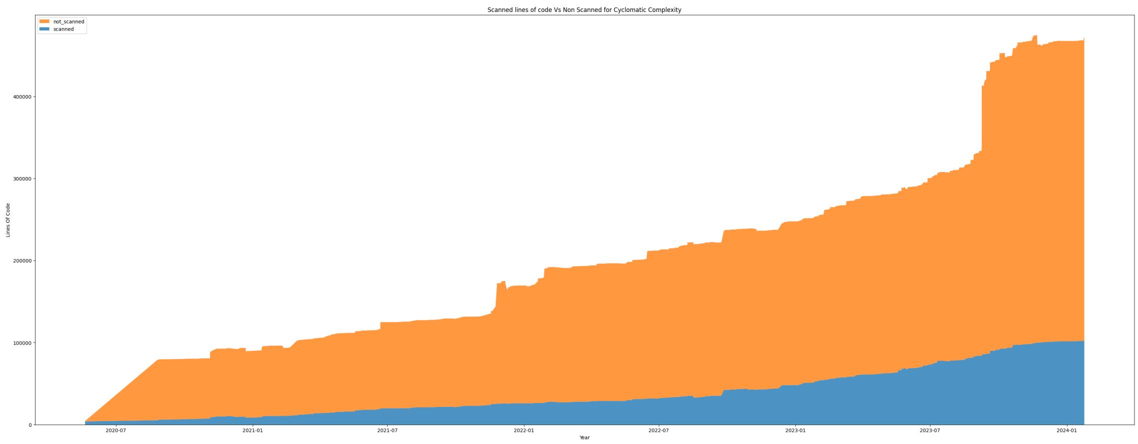

Figure 3: Lines of code in the Kuadrant projects scanned by the cyclomatic complexity tools.

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.

3.4 Diving into the Complexity

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

In **Figure 4** the cyclomatic complexity is scored overtime. The main take for 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

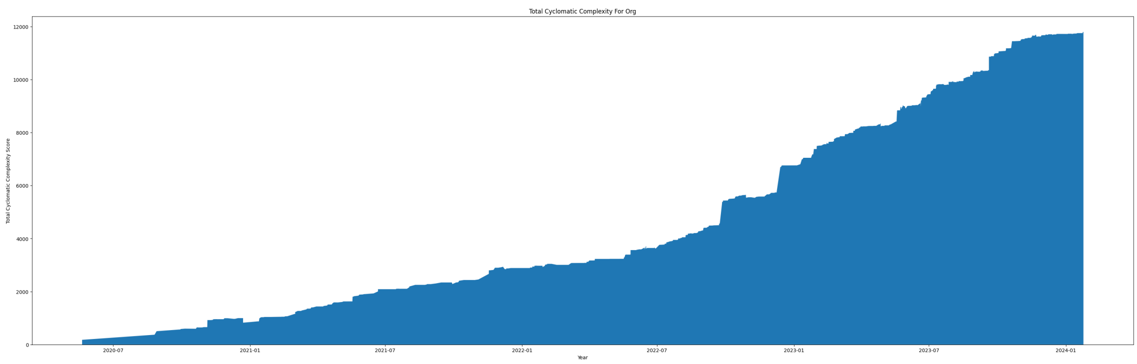

Figure 4: cyclomatic complexity for the source code

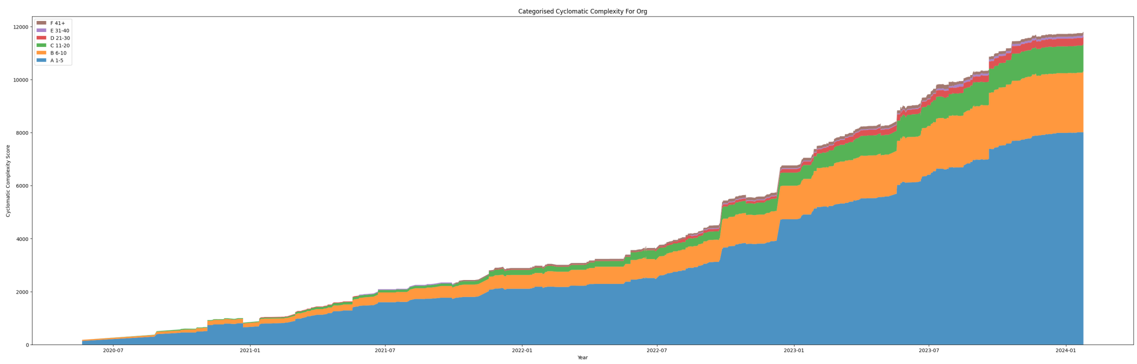

Figure 5: cyclomatic complexity score ranked

picture.

In **Figure 5**, by far the largest Rank is the A group which is at least two thirds the overall 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 / function's that needs 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 functions tides together.

Only two components in the Kuadrant project have the Rank A grouping less than 50% of

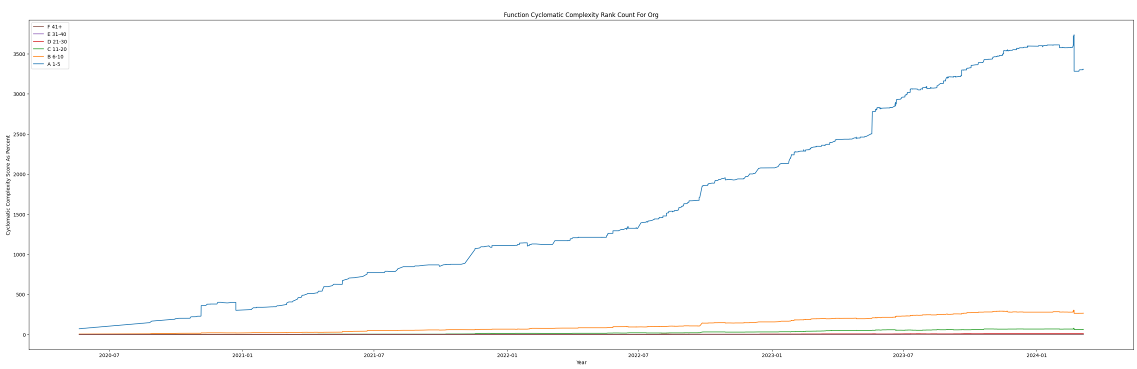

Figure 6: Count of functions in each Rank for the Kuadrant Project

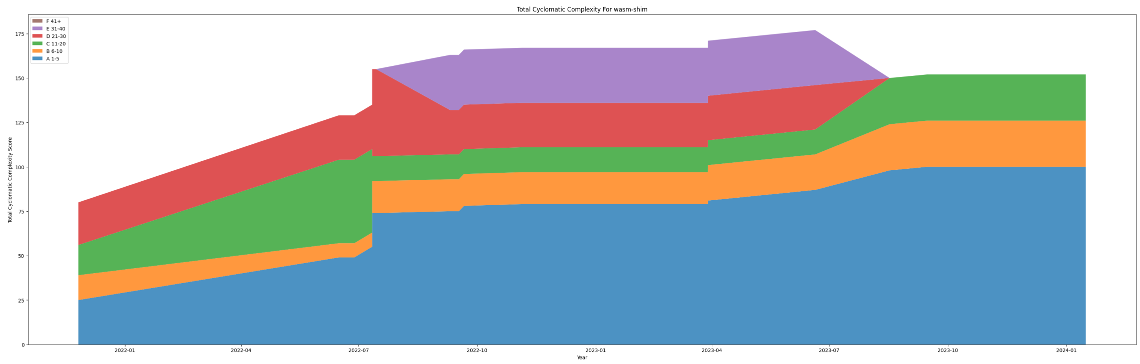

Figure 7: Total cyclomatic complexity over time for the wasam shim

the overall ranking (authorino-operator, multicluster-gateway-controller). Other components have some other interesting aspects which well be covered. Not all graphs for each component will be covered here, as there is too many and the data will be out of date quickly. For that reason the notebooks that generate the all the charts is 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 grow in complexity overtime but also that it can be refactored. For some time there was Rank E functions in the code base, which possible replaced some Rank D functions, but now all the Rank E and D functions have being written out, but the over all cyclomatic complexity has not changed much.

A project's complexity will increase overtime and the Ranking of that complexity can also increase, but does not mean it can not 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 overtime.

The pattern of refactoring out these complex functions from different Rankings can be seeing 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's without changing the overall complexity of the component.

3.4.2 The limitador project

Limitador was one of the project that was 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 it's 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.

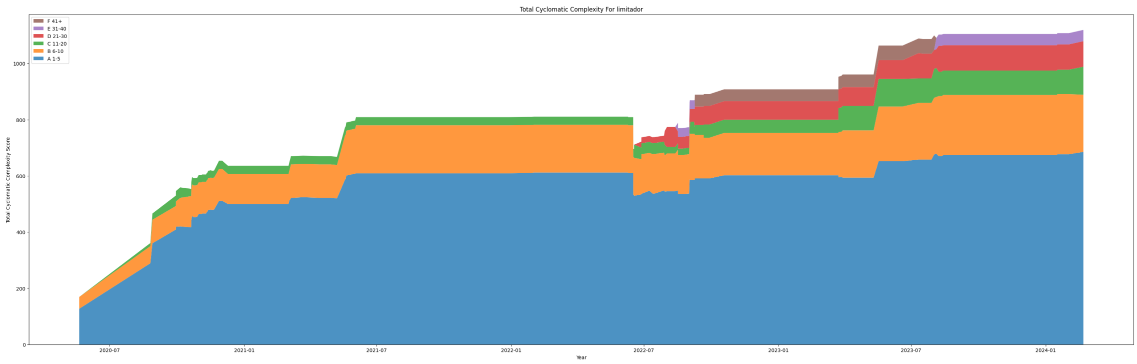

Figure 8: Total cyclomatic complexity over time for limitador

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 engernning team but quitilly engering, who are always under staffed and has other focus. The other outliner is the repo is writen in Python. Python falls into the langauge assumsption that cyclomatic complexity of functions would be less as Pythons duck typing allows the write 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 algin with the assuption made in the start, but seen these numbers raised a concern for me. How do new developers understand this repo? Hopefully the documantion for the repo explains how to combine the 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 ran by Python in the normal sense. To the checks for pytest is to use the *assert* keyword, which should not be used in production Python. The *assert* keyword can be disabled when using Python directly, this may mean the tool for caluclating the cyclomatic complexity does not count the *assert* keyword.

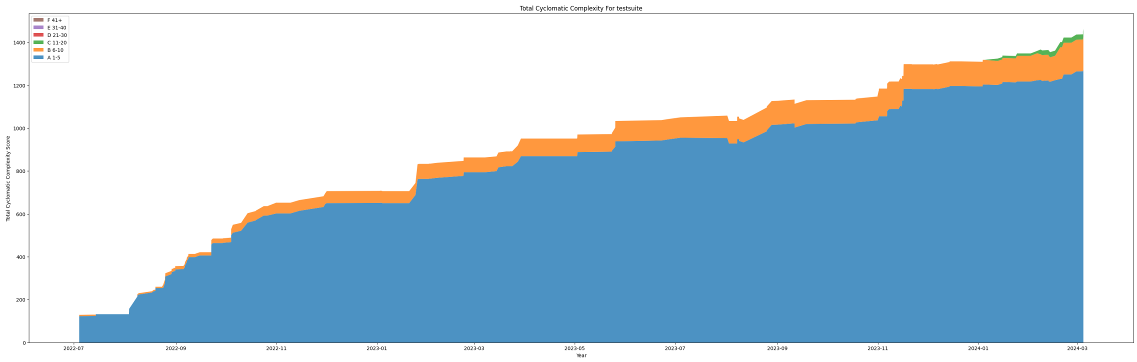

Figure 9: Total cyclomatic complexity over time for testsuite.

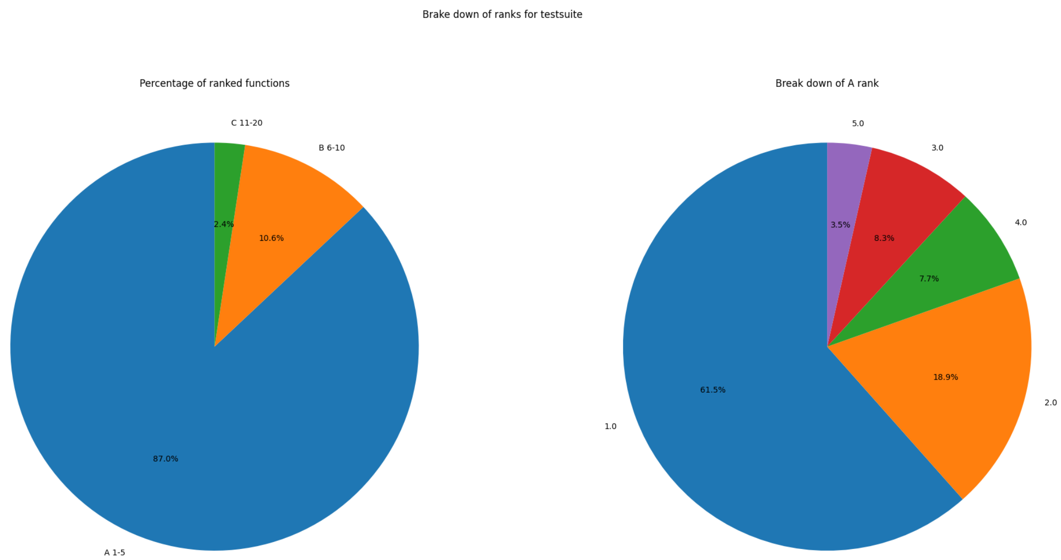

Figure 10: Pie chart of testsuite rankings.

3.5 Reference

3.5.1 Datafile

3.5.2 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

3.5.3 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

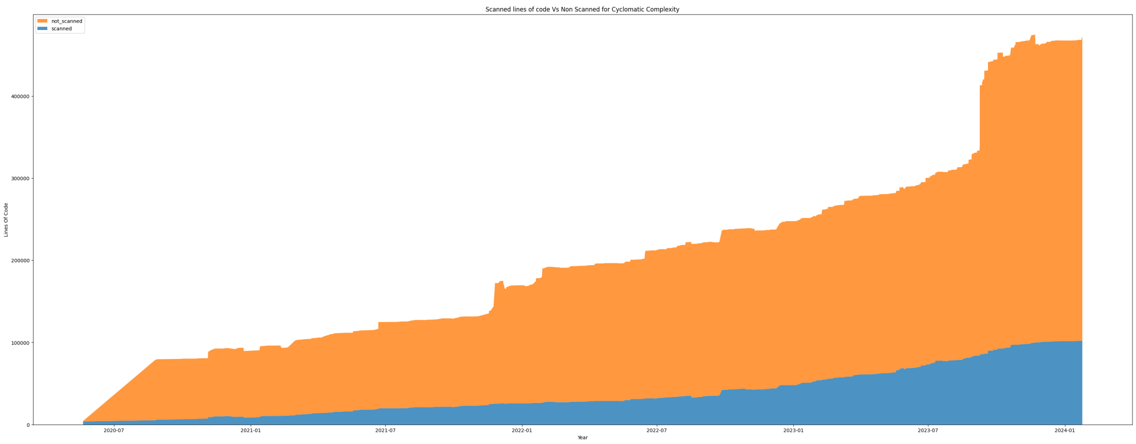

Figure 11: Go function with a cyclomatic complexity of 3

4 Using the cyclomatic complexity metric

The cyclomatic complexity metric can be used to guide the development within the Kuadrant projects. But it can not 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 test 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 11** we can see that the cyclomatic complexity metric is 3, as there is 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 12**. The interesting thing about this code is the cyclomatic complexity metric is now 4, but it requires 8 unit tests to cover all possible outcomes of the function. It shows that as the cyclomatic complexity metric goes up number of unit test can explode.

The other interesting thing about the function in **Figure 12** is how code coverage is affected. You can achieve 100% code coverage with using only 2 unit tests. 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 tests cases they should be writing for a function to get a good coverage of the possible outcomes of a function. While at the same time not being crazy in the number of tests that is required.

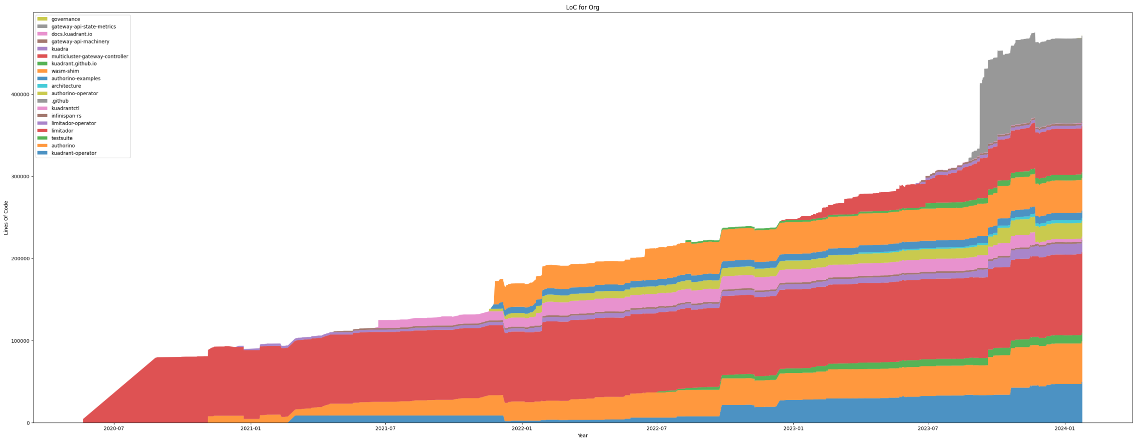

Figure 12: Go function with a cyclomatic complexity of 4

4.1.1 The 10$^{\text{th}}$ September 1752 problem.

I bring up this problem as it shows how the cyclomatic complexity metric can not answer many questions a developer many need to be asking about there code. On the 10$^{\text{th}}$ September 1752 there are many place in the world that you could have being, but you could not have being 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 place switched from the Julian Calendar to the Gregorian Calendar, a process that took 300 years around to complete. 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 case 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.

5 Conclusion