Victor Hugo Faria Dias Magalhães

# Arcabouço para comparação de analisadores estáticos de programas Go

Projeto de monografia requisitada na disciplina de Monografia em Sistemas de Informação II do Bacharelado de Sistemas de Informação UFMG

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Orientador: Fernando Magno Quintão Pereira

Belo Horizonte, Minas Gerais
2023

# Sumário

# 1 Introdução

A Análise estática de código é uma técnica que se baseia em analisar o código-fonte de um *software* sem executá-lo [Babati et al. 2017]. Apesar dos estudos demonstrando vantagens do uso de ferramentas de análise estática de código durante o processo de desenvolvimento, pessoas desenvolvedoras [Johnson et al. 2013] afirmam que as ferramentas atuais podem não oferecer informações suficientes para que os *problemas* que elas relatam sejam solucionados. Definem-se como *problemas*, erros relacionados a padrões de formatação da linguagem, erros lógico/funcionais, vazamento de recursos, falhas de segurança entre outros [Emanuelsson e Nilsson 2008].

*Go* é uma linguagem de programação *open-source* multi paradigma desenvolvida pela *Google* e disponibilizada para o público geral em 2009, com os objetivos de ser rápida, eficiente, confiável e simples de desenvolver [McGrath 2020]. *Go* conta com ferramentas de análise estática de código nativas da linguagem, como as ferramentas *go fmt* e *go vet*. Ambas as ferramentas utilizam análise estática do código fonte para validar estilo e heurísticas de qualidade respectivamente [Khatchadourian et al. 2022].

Embora conte com as facilidades das ferramentas de análise estática de código incluídas por padrão na linguagem, vê-se no geral que o uso de tais ferramentas não elimina a existência de *problemas* [Emanuelsson e Nilsson 2008]. Para além disso, algumas propriedades verificadas por essas ferramentas são impossíveis de serem decididas com total certeza, o que torna análises estáticas intrinsecamente imprecisas [Emanuelsson e Nilsson 2008].

Apesar desses pontos, análises estáticas são amplamente difundidas dado o baixo custo de execução e a elevada velocidade em relação a análises totalmente manuais feitas por uma pessoa desenvolvedora [Lavazza, Morasca e Tosi 2021]. Análises estáticas têm a possibilidade de serem poderosas ferramentas para auxiliar durante o desenvolvimento, acelerando as análises manuais e possibilitando que código seja entregue mais rápido e com mais qualidade. Qualidade de código tendo um custo muito importante na indústria de informática: a título de ilustração, em 2020, o Custo da Baixa Qualidade de Software (do inglês, *Cost of Poor Software Quality* -

CPSQ) foi de US$ 2,08 trilhões [Krasner 2021].

Isso posto, ainda que a linguagem *Go* seja relativamente nova, ela já conta com um ecossistema extenso [McGrath 2020], o que permite análises qualitativas e quantitativas de diversos aspectos e propriedades de suas ferramentas de análise estática de código. Logo, surge a necessidade de decidir quais ferramentas são adequadas para diferentes contextos. Desta forma, times de desenvolvimento podem evitar o esforço de decidir entre inúmeros tipos de ferramentas cujas funcionalidades por vezes se sobrepõe e outras vezes se complementam, mas que não foram comparadas.

## 1.1 Objetivos Gerais

O objetivo deste trabalho de Monografia de Sistemas de Informação II é contribuir para a literatura oferecendo um arcabouço, ou em inglês, *framework* de comparação para ferramentas de análise estática para *Go*, utilizando-se dos aprendizados adquiridos durante o trabalho de Monografia de Sistemas de Informação I. Isso pretende ser efetuado através de uma *wiki* colaborativa, baseando-se nos resultados na Monografia de Sistemas de Informação I.

## 1.2 Objetivos Específicos

Os objetivos específicos desta Monografia em Sistemas de Informação II consistem na construção e implementação de um arcabouço para comparação de diferentes aspectos das ferramentas de análise estática disponíveis no ecossistema da linguagem *Go*. Reitera-se que o objetivo não é obter um *rank* ou elencar melhores ou piores ferramentas. Esse tipo de análise não está no escopo deste trabalho e também é impossível de ser feita de forma objetiva, dado o caráter subjetivo para escolha das ferramentas, que depende das necessidades dos times de desenvolvimento e das diferenças em funcionalidades que as ferramentas oferecem. Com a construção do arcabouço e sua aplicação em uma *wiki* colaborativa, espera-se que seja possível obter uma metodologia de comparação sólida e passível de repetição, que possa ser utilizada para comparar ferramentas de análise estática para a linguagem *Go*

por diferentes pessoas desenvolvedoras, construindo uma *wiki* expandida, com resultados das medições e *feedbacks* qualitativos de quem as preencheu.

# 2 Referencial Teórico

## 2.1 Linguagem *Go*

O ecossistema da linguagem *Go* é extenso e conta com múltiplas ferramentas e participação da comunidade que cresce a cada dia [McGrath 2020]. Dado o caráter performático e construído para programação concorrente [Pang 2016] é possível entender o crescimento em popularidade da linguagem, se tornando por exemplo, a ferramenta de programação mais utilizada por profissionais de *DevOps* em 2021 [Dada et al. 2022]. A linguagem *Go* conta com facilidades construídas por padrão na linguagem, como por exemplo: sintaxe simples, suporte à concorrência, sistema de tipos estáticos, coleta de lixo e segurança de memória (ou do inglês, *memory safety*) [McGrath 2020].

## 2.2 Análise estática de código

Análise estática de código permite que validações e métricas sejam realizadas e obtidas sem a execução do programa. O processo de analisar estaticamente um programa não só cobre falhas de análise e operação dos compiladores como também se mostra uma poderosa ferramenta para detectar irregularidades, problemas de padronização e possíveis defeitos [Stefanović et al. 2020]. Ferramentas de análise estática são em síntese, programas que analisam o código fonte de outros programas e conseguem atuar em problemas visíveis no código [Stefanović et al. 2020].


Figura 1 – Tipos de problemas em código - Fonte: [Stefanović et al. 2020]

Diferentes tipos de análises podem ser realizadas para encontrar diferentes tipos de *problemas*. Abaixo exemplos de categorias de regras para as quais ferramentas comerciais conseguem encontrar problemas [Novak, Krajnc e Zontar 2010]:

- Problemas sintáticos
- Código-fonte inacessível
- Variáveis não declaradas
- Variáveis não inicializadas
- Funções e procedimentos não utilizados
- Variáveis utilizadas antes da inicialização
- Não utilização de valores de funções
- Uso incorreto de ponteiros
- Concorrência

- Más práticas
- Interface do Usuário (do inglês, *User Interface, UI*)
- Corretude
- Design
- Exceções
- Interoperabilidade
- Manutenibilidade
- Nomenclatura
- Desempenho
- Portabilidade
- Segurança
- Serialização

Muitas ferramentas implementam análises que cobrem múltiplas destas classes de problemas, o que exige uma análise sistemática que leve em consideração a taxonomia das ferramentas, analisando não apenas as funcionalidades, mas também seu contexto de manutenção, usabilidade, integração, fornecimento de resultados entre outras características intrínsecas às ferramentas [Novak, Krajnc e Zontar 2010].

### 2.2.1 Regras de análise estática de código

Regras de análise estática de código são definições objetivas ou heurísticas, que definem um grupo de características de um código fonte como problemático. Ao longo dos anos, essas regras foram construídas por comunidades e empresas, em um processo altamente manual e que consome muito tempo [Garg 2022]. Existem inúmeras regras adotadas por diversas ferramentas, porém, é possível categorizar boa parte das regras em um conjunto de 9 itens [Novak, Krajnc e Zontar 2010]:

1. Estilo - inspeciona a aparência visual do código-fonte.

2. Nomenclatura - revisão para verificar se as variáveis estão corretamente nomeadas (ortografia, padrões de nomenclatura, ...).

3. Geral - regras gerais da análise de código estático, incluindo detecção de *bugs* não relacionados a outros itens.

4. Concorrência - erros com a execução de código concorrente.

5. Exceções - erros ao lançar ou não lançar exceções.

6. Desempenho - erros com o desempenho das aplicações.

7. Interoperabilidade - erros com o comportamento comum entre partes do código.

8. Segurança - erros que podem afetar a segurança da aplicação.

9. Manutenibilidade - regras para melhorar a manutenibilidade da aplicação.

## 2.3 *Wikis* colaborativas

*Wikis* são conjuntos expansíveis de páginas *web*, colaborativas por natureza, que podem ser acessadas e editadas por qualquer membro de um certo grupo, ou aberta ao público geral [Engstrom e Jewett 2005]. *Wikis* são amplamente utilizadas para documentação e coordenação nos ambientes de desenvolvimento de *software* de código aberto ao redor do mundo [Xiao, Chi e Yang 2007], e vem se difundindo como capazes de proporcionar uma reflexão crítica, estimular a criação de novas ideias, promover o trabalho cooperativo e colaborativo e partilhar o conhecimento [Coutinho e Junior 2007]. Ferramentas modernas de hospedagem de código como o *Github* contam nativamente com a implementação de *wikis* a cada repositório criado, o que também facilita sua utilização [Yagel 2015]. Dessa forma, instrumentar um arcabouço para a construção de uma *wiki* se beneficia de todas as vantagens que elas oferecem.

# 3 Metodologia

Primeiramente, é necessária a instrumentação do processo de comparação realizado na Monografia em Sistemas de Informação I. Planeja-se instrumentalizar através de uma *wiki* colaborativa na plataforma de hospedagem de código *Github*. Será necessário descrever e exemplificar os processos de classificação das ferramentas de análise estática de código para as análises taxonômica [Novak, Krajnc e Zontar 2010] e comparativa [Emanuelsson e Nilsson 2008]. Planeja-se construir um arcabouço flexível e capaz de lidar com quaisquer ferramentas de código aberto (do inglês, *open-source*) disponíveis no mercado sejam populares entre as pessoas desenvolvedoras ou não, de forma independente de análise prévia de suas funcionalidades.

Planeja-se instrumentalizar o processo de análise taxonômica construindo uma versão adaptada e genérica para o contexto das ferramentas da linguagem *Go* de uma árvore de taxonomia presente em outros trabalhos da literatura [Novak, Krajnc e Zontar 2010]. Essa árvore de taxonomia genérica precisará ser flexível para a inclusão de características, e maleável de forma a permitir a construção de árvores para diversas ferramentas com facilidade. Para isso, utilizou-se de diagramas construídos a partir da plataforma de construção de diagramas *draw.io*, possibilitando que possa-se editar facilmente as árvores taxonômicas.

Para além disso, será necessário instrumentalizar a análise comparativa que se baseia nas funcionalidades das ferramentas. Tal qual com a análise taxonômica, planeja-se adaptar a tabela obtida na Monografia em Sistemas de Informação I para aceitar de forma maleável a inclusão de características. Para as medições quantitativas, planeja-se disponibilizar uma forma simples de executar as medições de tempo de processamento, uso de *CPU* e consumo de memória *RAM* através de uma imagem *Docker* e um orquestrador de contêineres, como o *docker-compose*, de forma a permitir uma execução sem grandes interferências do *hardware* dos colaboradores.

Por fim, com o arcabouço construído e instrumentalizado, planeja-se permitir adições de entradas na *wiki*, permitindo que pessoas desenvolvedoras que utilizam

ferramentas de análise estática da linguagem *Go* incluam entradas com os resultados de suas análises taxonômica e comparativa, além de permitir a coleta de opiniões qualitativas sobre as ferramentas utilizadas.

# 4 Resultados

Para a construção do arcabouço na *wiki*, utilizou-se da estrutura de *wiki* da plataforma de hospedagem de código *Github*. A figura 2 contém uma visualização da página principal da *wiki*. É possível perceber que o arcabouço se aproveita bastante dos recursos que a *wiki* oferece, a figura 2 por exemplo, ilustra uma grandeza de *links* entre as páginas da *wiki*, além de uma barra lateral que é extremamente útil para a navegação.

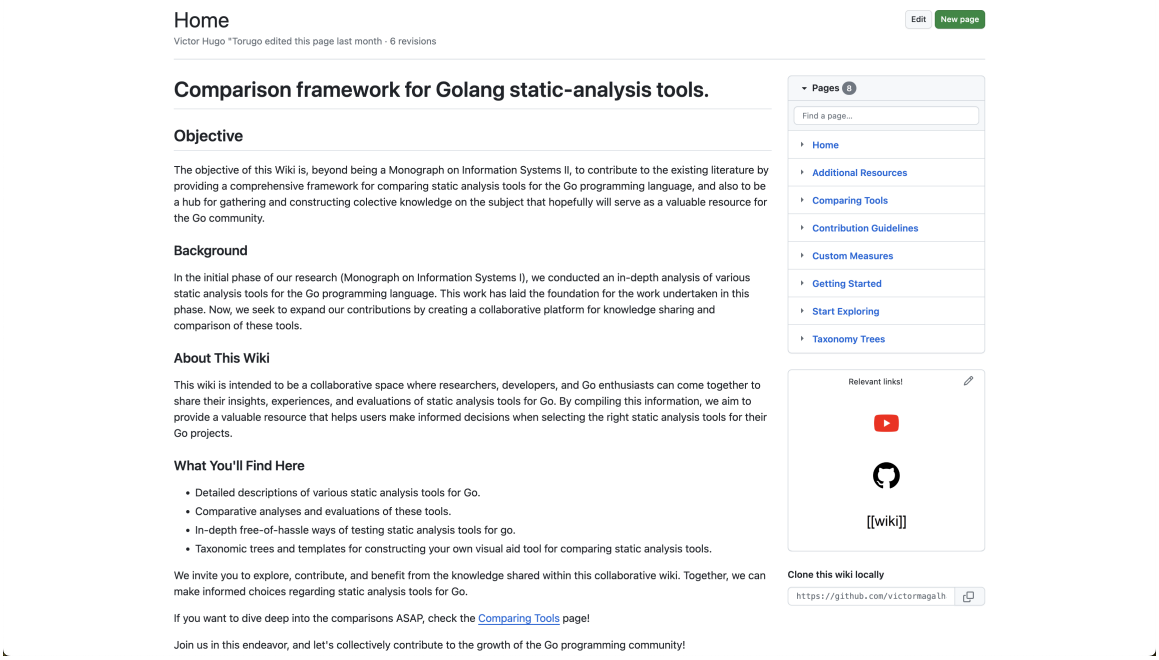


Figura 2 – Visualização da página principal da *wiki* - Figura autoral

Despendeu-se um tempo estruturando o processo de contribuição do arcabouço, focando em garantir a edição da *wiki* através de regras claras e simples, permitindo que qualquer pessoa interessada em contribuir consiga realizar as medições e adicionar seus resultados, mantendo a estrutura e qualidade do arcabouço.

Construiu-se na *wiki* a tabela de comparação entre funcionalidades, migrando e atualizando todos os dados obtidos na Monografia de Sistemas de Informação I. O

resultado obtido está disponível na tabela 1.

| - | *Chronos* | *gokart* | *goreporter* | *go vuln-check* | *prealloc* | *revive* | *Staticcheck* |
|---|---|---|---|---|---|---|---|
| *Race conditions* | Sim | Não | Não | Não | Não | Sim | Sim |
| *Estilo de codificação* | Não | Não | Sim | Não | Não | Sim | Sim |
| *Tipagem* | Não | Não | Sim | Não | Não | Sim | Sim |
| *Má ordem de retorno* | Não | Não | Sim | Não | Sim | Sim | Não |
| *Strings* | Não | Sim | Sim | Não | Não | Sim | Sim |
| *Memory leaks* | Não | Sim | Sim | Não | Não | Sim | Sim |
| *Variáveis não inicializadas* | Não | Não | Sim | Não | Sim | Sim | Sim |
| *Não terminação* | Sim | Não | Não | Não | Não | Sim | Sim |
| *Extensível* | Não | Sim | Não | Sim | Não | Sim | Não |
| *Métricas* | Não | Não | Sim | Não | Não | Sim | Não |
| *Vulnerabilidades* | Não | Sim | Não | Sim | Não | Não | Não |
| *Riscos de segurança* | Não | Sim | Sim | Sim | Sim | Sim | Não |
| *Nomenclatura* | Não | Não | Sim | Não | Não | Sim | Sim |
| *Desempenho* | Não | Não | Sim | Não | Sim | Sim | Sim |
| *Concorrência* | Sim | Não | Não | Não | Não | Sim | Sim |
| *Versão do Golang* | >= 1.15 | >= 1.19 | >= 1.6 | >= 1.18 | >= 1.15 | >= 1.19 | >= 1.19 |
| *Estrelas no Github* | 404 | 2100 | 3100 | 280 | 530 | 4200 | 5400 |
| *Forks no Github* | 13 | 108 | 275 | 43 | 23 | 251 | 340 |
| *Tempo de execução* | 8.24s | 20.37s | 15.73s | 3.82s | 0.1s | 0.27s | 5.41s |
| *RAM (máx)* | 1568kB | 338kB | 99kB | 711kB | 10kB | 36kB | 192kB |
| *CPU utilizada (%)* | 259% | 92% | 40% | 263% | 34% | 76% | 251% |

Tabela 1 – Comparação entre funcionalidades

Além disso, a *wiki* conta com as visualizações das árvores de taxonomia, contando com a árvore de taxonomia genérica, que foi construída para ser utilizada como base para a construção das árvores de taxonomia para novas ferramentas, a figura 3 ilustra a árvore de taxonomia genérica.

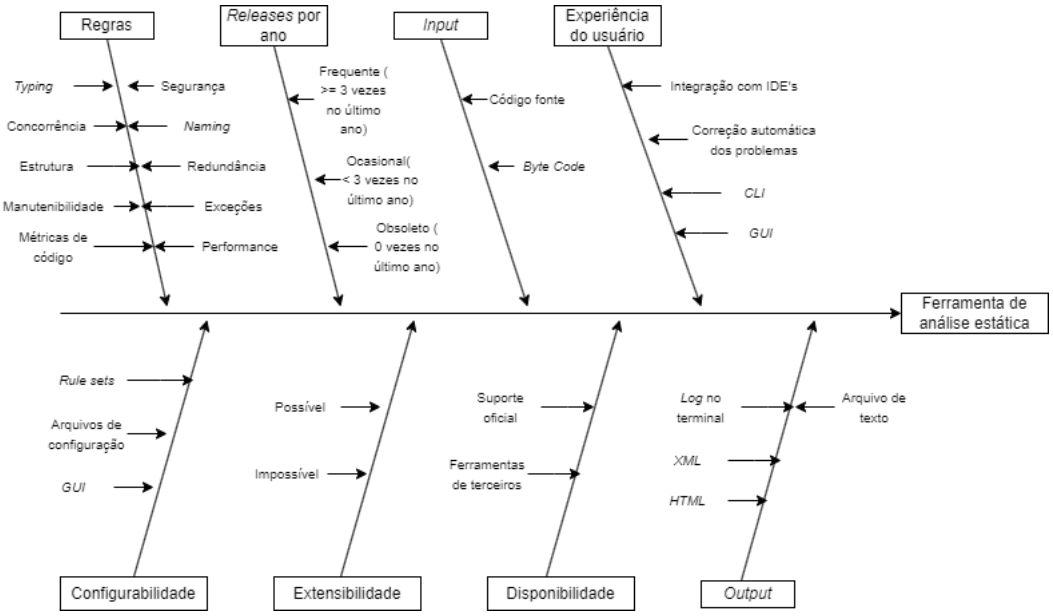


Figura 3 – Árvore de taxonomia genérica - Figura autoral

O processo de instrumentalização foi feito utilizando as ferramentas de contêineres e orquestração de contêineres, *docker* e *docker-compose*. Dessa forma, é possível com poucos requisitos iniciais, é possível configurar uma nova ferramenta e realizar sua medição em qualquer máquina e sistema operacional que comportem a execução do *docker* e do *docker-compose*, sumariamente fazendo estes os dois únicos requisitos para a execução das medições instrumentalizadas.

A figura 4, mostra um diagrama simplificado do processo de medição das ferramentas. O processo se inicia no sistema operacional do usuário, onde é possível definir utilizando um arquivo de configuração qual ferramenta será testada. Após a definição no arquivo de configuração, o sistema operacional do usuário chama o *docker-compose* que utiliza do arquivo de configuração e do *Dockerfile* para iniciar um contêiner *docker*, que realiza as medições através de um *benchmark*. O *benchmark* consiste em executar a ferramenta de análise estática escolhida em uma

versão específica de um projeto *Go*, e realizando medições de tempo, consumo de memória *RAM* e consumo de *CPU* através do comando *time* do software de terminal *zsh*. O projeto *Go* escolhido para o *benchmark* é um projeto relevante de código aberto, o *tsuru*, um projeto de plataforma como serviço, com mais de 4500 estrelas no *Github*. A figura 5 ilustra o resultado da execução, com todas as medições realizadas.

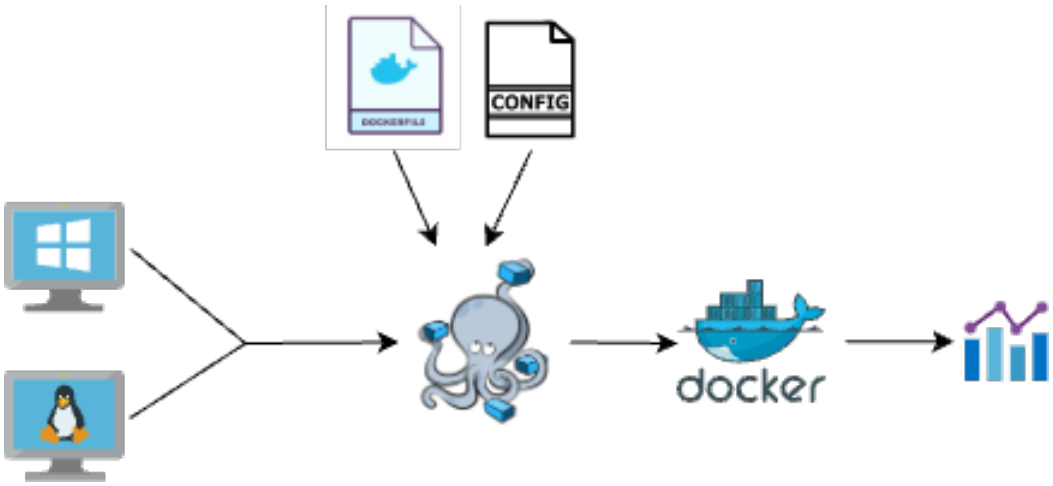


Figura 4 – Instrumentalização da medição - Figura autoral

```
user time: 14.22s
system time: 7.18s
real time: 8.24s
total cpu: 259%
max memory 1568 kB
```

Figura 5 – Resultado da medição do *Chronos* - Figura autoral

Considerando o resultado das medições e do arcabouço, foi possível realizar uma análise sobre a qualidade relativa das ferramentas. Tendo em vista que o objetivo deste trabalho não é elencar as ferramentas como melhores ou piores, ainda assim é possível reconhecer que para o contexto geral da maior parte dos usuários algumas ferramentas atingem um nível de qualidade excepcional que as categoriza como estado da arte, principalmente *Staticcheck* e *revive*.

Existe grande empenho dos times de desenvolvimento em implementar a manter ambas as ferramentas, e isso é refletido em diversas métricas e análises qualitativas e quantitativas. Ambas apresentam compatibilidade a atualizações frequentes para as mais novas versões da linguagem *Go*, sendo juntamente com *gokart* as únicas ferramentas analisadas que foram atualizadas para versões superiores a 1.19.

Por fim, a característica generalista de ambas de suportar diversas funcionalidades no contexto de análise estática e seu reduzido tempo de execução e uso de recursos, colocam ambas as ferramentas em posições muito de confiabilidade e relevância para a maior parte dos contextos dos times de desenvolvimento *Go*.

É relevante ainda pontuar que outras ferramentas são extremamente poderosas em seus devidos contextos, *gokart* por exemplo, foi utilizada de referência para comparação em projetos da literatura que visam desenvolver ferramentas que buscam vulnerabilidades no código fonte [Li et al. 2022].

Dentro ainda deste grupo de duas ferramentas, a *revive* se sobressaiu em diversos aspectos, mas principalmente nos custos de execução, tanto em termos de tempo quanto em recursos, se mostrando bem otimizada e veloz, fator crucial para diversos times de desenvolvimento que incorporam análise estáticas em seus fluxos de integração e *deployment* contínuos (do inglês, *Continuous Integration and Continuous Deployment* ou *CI/CD*), fluxos estes que são relevantes em questões financeiras [Dakić, Todorović e Vranić 2022].

# 5 Conclusão

Ferramentas de análise estática são parte extensa do ecossistema de diversas linguagens, incluindo *Go*, e oferecem diversos recursos durante o processo de desenvolvimento de software, sendo importante a comparação entre os requisitos e funcionalidades das diversas ferramentas para os diferentes contextos de desenvolvimento, fato este que foi evidenciado na Monografia de Sistemas de Informação I.

Dessa forma, este trabalho estendeu o que foi desenvolvido anteriormente e seguiu com uma padronização do processo de avaliação sistemática de ferramentas. A *wiki* se provou uma excelente implementação do arcabouço, possibilitando que os objetivos de se obter um método sólido e passível de repetição fosse plenamente cumprido. Além disso, a estrutura do texto se mostrou imparcial em relação às ferramentas não realizando comparações ou expressando opiniões pessoais sobre a ferramenta, ainda assim não excluindo análises em relação às capacidades das ferramentas.

Apesar de não ser uma das propostas deste trabalho, não houveram contribuições de terceiros para a *wiki*. Isso não fere de forma alguma o escopo planejado para esta Monografia em Sistemas de Informação II, porém abre um espaço interessante para trabalhos futuros que se foquem na divulgação e análise do conteúdo e evolução de arcabouços de comparação de ferramentas de análise estática, tanto para *Go*, quanto para outras linguagens de programação.

Por fim, conclui-se que a implementação do arcabouço cumpriu os objetivos esperados com grande clareza no processo de documentação dos resultados e possibilitando uma visualização amigável ao usuário de uma análise relativamente complexa, mostrando assim que um arcabouço estruturado em *wiki* para a comparação das ferramentas de análise estática para *Go* é válida e útil para o contexto escolhido, principalmente nos âmbitos de visualização e navegação dos resultados, refletindo os pontos levantados por [Coutinho e Junior 2007], sobre a capacidade de proporcionar uma reflexão crítica, estimular a criação de novas ideias, promover o trabalho cooperativo e colaborativo e partilhar o conhecimento.

# Referências

[Babati et al. 2017]BABATI, B. et al. Static analysis toolset with clang. In: *Proceedings of the 10th International Conference on Applied Informatics (30 January–1 February, 2017, Eger, Hungary)*. [S.l.: s.n.], 2017. p. 23–29.

[Coutinho e Junior 2007]COUTINHO, C. P.; JUNIOR, J. B. B. Blog e wiki: os futuros professores e as ferramentas da web 2.0. 2007.

[Dada et al. 2022]DADA, O. A. et al. Hidden gold for it professionals, educators, and students: Insights from stack overflow survey. *IEEE Transactions on Computational Social Systems*, IEEE, 2022.

[Dakić, Todorović e Vranić 2022]DAKIĆ, P.; TODOROVIĆ, V.; VRANIĆ, V. Financial justification for using ci/cd and code analysis for software quality improvement in the automotive industry. In: IEEE. *2022 IEEE Zooming Innovation in Consumer Technologies Conference (ZINC)*. [S.l.], 2022. p. 149–154.

[Emanuelsson e Nilsson 2008]EMANUELSSON, P.; NILSSON, U. A comparative study of industrial static analysis tools. *Electronic notes in theoretical computer science*, Elsevier, v. 217, p. 5–21, 2008.

[Engstrom e Jewett 2005]ENGSTROM, M. E.; JEWETT, D. Collaborative learning the wiki way. *TechTrends*, Springer Nature BV, v. 49, n. 6, p. 12, 2005.

[Garg 2022]GARG, P. Example-based synthesis of static analysis rules. *arXiv preprint arXiv:2204.08643*, 2022.

[Johnson et al. 2013]JOHNSON, B. et al. Why don't software developers use static analysis tools to find bugs? In: IEEE. *2013 35th International Conference on Software Engineering (ICSE)*. [S.l.], 2013. p. 672–681.

[Khatchadourian et al. 2022]KHATCHADOURIAN, R. et al. How many mutex bugs can a simple analysis find in go programs? 2022.

[Krasner 2021]KRASNER, H. The cost of poor software quality in the us: A 2020 report. *Proc. Consortium Inf. Softw. QualityTM (CISQTM)*, 2021.

[Lavazza, Morasca e Tosi 2021]LAVAZZA, L.; MORASCA, S.; TOSI, D. Comparing static analysis and code smells as defect predictors: an empirical study. In: SPRINGER. *Open Source Systems: 17th IFIP WG 2.13 International Conference, OSS 2021, Virtual Event, May 12–13, 2021, Proceedings 17*. [S.l.], 2021. p. 1–15.

[Li et al. 2022]LI, W. et al. Cryptogo: Automatic detection of go cryptographic api misuses. In: *Proceedings of the 38th Annual Computer Security Applications Conference*. [S.l.: s.n.], 2022. p. 318–331.

[McGrath 2020]MCGRATH, M. *GO Programming in easy steps: Discover Google's Go language (golang)*. [S.l.]: In Easy Steps Limited, 2020.

[Novak, Krajnc e Zontar 2010]NOVAK, J.; KRAJNC, A.; ZONTAR, R. Taxonomy of static code analysis tools. 01 2010.

[Pang 2016]PANG, J. Golang-planetary programming language. 2016.

[Safonov 2016]SAFONOV, V. O. *Trustworthy cloud computing*. [S.l.]: John Wiley & Sons, 2016.

[Stefanović et al. 2020]STEFANOVIĆ, D. et al. Static code analysis tools: A systematic literature review. *Annals of DAAAM & Proceedings*, v. 7, n. 1, 2020.

[Xiao, Chi e Yang 2007]XIAO, W.; CHI, C.; YANG, M. On-line collaborative software development via wiki. In: *Proceedings of the 2007 international symposium on Wikis*. [S.l.: s.n.], 2007. p. 177–183.

[Yagel 2015]YAGEL, R. Lido-wiki based living documentation with domain knowledge. In: *SKY*. [S.l.: s.n.], 2015. p. 26–30.