UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
UNIDADE ACADÊMICA DE SISTEMAS E COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

IGOR ARAÚJO TAVARES DE FARIAS

CARACTERIZAÇÃO DE COMPONENTES *OUTLIERS* EM SISTEMAS *FRONT-END REACT*: UM ESTUDO A PARTIR DE MÉTRICAS DE ENGENHARIA DE *SOFTWARE*

CAMPINA GRANDE - PB

2021

IGOR ARAÚJO TAVARES DE FARIAS

CARACTERIZAÇÃO DE COMPONENTES *OUTLIERS* EM SISTEMAS *FRONT-END REACT*: UM ESTUDO A PARTIR DE MÉTRICAS DE ENGENHARIA DE *SOFTWARE*

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Professor Dr. Matheus Gaudencio do Rêgo.

CAMPINA GRANDE - PB

2021

F224c Farias, Igor Araújo Tavares de.
Caracterização de componentes outliers em sistemas front-end react: um estudo a partir de métricas de Engenharia de Software. / Igor Araújo Tavares de Farias. – 2021.

14 f.

Orientador: Prof. Dr. Matheus Gaudencio do Rêgo
Trabalho de Conclusão de Curso - Artigo (Curso de Bacharelado em Ciência da Computação) - Universidade Federal de Campina Grande; Centro de Engenharia Elétrica e Informática.

1. Componentes outliers. 2. Sistemas front-end react. 3. Engenharia de Software. 4. Métricas de Engenharia de Software. Arquitetura de software. I. Rêgo, Matheus Gaudencio do. II. Título.

CDU:004(045)

Elaboração da Ficha Catalográfica:

Johnny Rodrigues Barbosa
Bibliotecário-Documentalista
CRB-15/626

IGOR ARAÚJO TAVARES DE FARIAS

CARACTERIZAÇÃO DE COMPONENTES *OUTLIERS* EM SISTEMAS *FRONT-END REACT*: UM ESTUDO A PARTIR DE MÉTRICAS DE ENGENHARIA DE *SOFTWARE*

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA:

Professor Dr. Matheus Gaudencio do Rêgo
Orientador – UASC/CEEI/UFCG

Professor Dr. Dalton Dario Serey Guerreiro
Examinador – UASC/CEEI/UFCG

Professor Dr. Tiago Lima Massoni
Professor da disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em: 20 de outubro de 2021.

CAMPINA GRANDE - PB

RESUMO (ABSTRACT)

The production of software structured in components is a complex task. Despite the emergence of new tools and technologies to facilitate the production of front-end systems, there is no consensus on how to build components and organize the code architecture for this type of development. Therefore, this work proposes to analyze large open source front-end projects that make use of the React library and, based on software engineering metrics and specific attributes of modern React development, to characterize components that stand out for these metrics and describe how they are structured into systems of great relevance to the community. Thus, the objective is to highlight types of unusual components in the projects, relevant characteristics of their construction, and how the architectural decisions impacted the software engineering metrics, aiming to provide a better understanding of the process of developing front-end systems and favor decision-making for the construction of new projects in the area.

Caracterização de componentes outliers em sistemas front-end React: um estudo a partir de métricas de engenharia de software

Igor Araújo Tavares de Farias
Universidade Federal de Campina Grande
Campina Grande, Brazil
igor.farias@ccc.ufcg.edu.br

Matheus Gaudencio do Rêgo
Universidade Federal de Campina Grande
Campina Grande, Brazil
matheusgr@computacao.ufcg.edu.br

RESUMO

A produção de softwares estruturados em componentes é uma tarefa complexa. Apesar do surgimento de novas ferramentas e tecnologias para facilitar a produção de sistemas front-end, não há consenso quanto à forma de construção de componentes e organização da arquitetura de código para esse tipo de desenvolvimento. Portanto, este trabalho se propõe a analisar grandes projetos front-end de código aberto que fazem uso da biblioteca React, baseando-se em métricas da engenharia de software e atributos específicos do desenvolvimento React moderno, para caracterizar componentes que se destacam para essas métricas e descrever como eles são estruturados em sistemas de grande relevância para a comunidade. Com isso, objetiva-se destacar tipos de componentes incomuns nos projetos, características relevantes da construção desses, e de que maneira as decisões de arquitetura impactaram as métricas da engenharia de software, visando proporcionar melhor compreensão do processo de desenvolvimento de sistemas front-end e favorecer a tomada de decisões para a construção de novos projetos da área.

Palavras-chaves

Métricas, front-end, React, arquitetura de software, engenharia de software, componentes.

1. INTRODUÇÃO

Com a popularização do desenvolvimento web e emergência de novas tecnologias, temos que a criação de bibliotecas e frameworks front-end, principalmente do AngularJS em 2010 e React em 2013 [10] impulsionou uma forma diferente de construir softwares: a hierarquia de componentes.

Com novas tecnologias, surgiram também novas formas de gerenciar a lógica da aplicação, como a arquitetura Flux, criada pelo Facebook, que definiu um novo conceito para o fluxo de dados baseado em um fluxo unidirecional com estruturas bem definidas que transitam pelas entidades: Action → Dispatcher → Store → View e que posteriormente deu origem ao Redux. [5, 11]

Ainda no âmbito do React, temos a introdução dos hooks desde a versão 16.8 que permite o uso de estado e diversas outras funcionalidades sem a necessidade de criar classes, o que favoreceu a resolução de diversos problemas envolvendo a manutenção de grandes quantidades de componentes [8] e definiu uma nova forma de desenvolvimento usando a biblioteca, que se mostrou como tendência para o desenvolvimento de futuros projetos dada a grande proporção de adoção dessa tecnologia nos repositórios encontrados.

No entanto, apesar do surgimento de novas ferramentas para facilitar o desenvolvimento front-end e superar desafios para estruturar códigos para a web, o desenvolvimento em componentes apresenta grande complexidade no que diz respeito à arquitetura de software, especialmente em sistemas de grande porte.

Além disso, não há descrição dos possíveis diferentes tipos de componentes que podem existir e como eles se comportam. De modo que é difícil discutir diferentes possibilidades de arquitetura e estrutura de código.

Diante disso, esse trabalho procura identificar possíveis novos tipos de componentes, ou componentes que são estruturados de forma única a partir da detecção de *outliers* em métricas comumente usadas em engenharia de software e em como esses componentes fazem uso de hooks. Assim, visa-se possibilitar melhor compreensão sobre o processo de construção desse tipo de software e como esse é conduzido em grandes sistemas de maneira que esse conhecimento possa fomentar melhores decisões práticas para a construção de futuros sistemas front-end.

Para esse estudo, foram analisados repositórios front-end de código aberto que fazem uso da biblioteca React, que foi escolhida por ser considerada a ferramenta front-end mais popular e a que os desenvolvedores mais estão interessados em aprender para a mesma categoria e a segunda colocada em popularidade considerando desenvolvimento web como um todo, segundo a pesquisa realizada pela plataforma StackOverflow em 2020. [1]

Por conseguinte, foram selecionadas métricas consideradas relevantes para análise de qualidade de código pelos materiais disponíveis na literatura de engenharia de software, bem como propriedades importantes do desenvolvimento React moderno voltado para componentes funcionais, os *hooks*. Com os repositórios e as métricas definidos, foram extraídos os dados referentes às métricas de cada um dos repositórios e separados os componentes *outliers* para cada uma destas para análise e definição de características, comportamento e decisões arquiteturais.

Com os resultados obtidos, observamos estruturas semelhantes nos diferentes projetos, refletindo inclusive alterações nas mesmas métricas, como no caso dos arquivos utilitários. Mas, ainda, temos formas diferentes de tratar questões presentes na maioria dos projetos front-end, como: o gerenciamento de rotas, compartilhamento de estado e organização da definição de tipos, que podem representar boas referências para a construção de softwares desse tipo, dado que essas questões são comuns para a maioria desses sistemas.

2. FUNDAMENTAÇÃO TEÓRICA

No que concerne à qualidade de código, pesquisas apontam que esta é uma característica importante de ser analisada, de modo que a não avaliação desse aspecto em um sistema pode levar a problemas como: dificuldade de manutenção do código, uso incorreto do sistema, aumento de custos, dificuldade de integração e até problemas de segurança. [2, 4].

Dado que esse é um tema amplamente discutido no setor da engenharia de software até os dias atuais, é possível estabelecer métricas oriundas de estudos na área que são consideradas relevantes para qualidade de código, das quais foram escolhidas as seguintes para servir de base ao estudo:

Linhas de código: métrica extremamente simples porém relacionada a diversos aspectos de qualidade de software, tais como: complexidade, segurança, confiabilidade, reutilização, eficiência e manutenção. [13] Consistindo em uma métrica essencial à produção de cada software, ainda que simples, está relacionada a tantos fatores distintos que os constituem e mostra-se bastante relevante para apontar possíveis problemas de implementação, especialmente no que diz respeito a acúmulo de responsabilidades em algum componente, que pode gerar vários dos problemas listados.

Quantidade de importações e exportações: as métricas de importações e exportações estão diretamente relacionadas ao nível *afferent coupling* e *efferent coupling*, nome dado às relações quando uma classe A faz referência a uma classe B (gerando um *efferent coupling* em A e um *afferent coupling* em B), indicando dependência entre elas. Essas métricas são de grande importância nos softwares pois modificações ou problemas em algum dos arquivos pode afetar o funcionamento de vários outros no sistema. [14]

Quantidade de funções: considera apenas as funções declaradas em um arquivo e é uma métrica bastante relevante por estar relacionada a fatores como: segurança, confiabilidade, tolerância a falhas, reutilização, eficiência, complexidade, manutenção, estabilidade, dentre outros. [13] Assim, essa métrica passa a ser relevante para definir o número de funcionalidades proporcionado por um arquivo e em grande quantidade pode ser um sinal de que a classe apresenta muitas responsabilidades, o que pode acarretar potenciais problemas relacionados aos fatores listados. [14] Ainda, quando acompanhada de elevadas métricas de importações ou exportações e, portanto, apresentar alto acoplamento, pode ainda ocasionar em problemas em diversos outros componentes do sistema. [13, 14]

Considerando as particularidades de um sistema front-end, as métricas de qualidade existentes não são suficientes para mensurar qualidade de código para sistemas web. Nesse sentido, Lin et al [3] propuseram novas métricas focadas em códigos JavaScript e componentes React.

Contudo, por essas métricas incluírem as métricas já mencionadas da engenharia de software porém com grande complexidade agregada e desconsiderar aspectos mais específicos como utilização de hooks em componentes funcionais que regem grande parte do desenvolvimento React dos novos projetos, como veremos nos repositórios, propomos utilizar as chamadas dos principais *hooks* disponíveis na documentação oficial do React: *useEffect*, useState, *useContext, useMemo* e *useCallback.* [8]

Com as métricas de *useEffect* e *useState* esperamos compreender aspectos mais gerais da construção de um software, como a utilização de funções com efeitos colaterais e o uso de estado local; com a terceira, esperamos compreender características relacionadas ao compartilhamento de estado na aplicação e com as duas últimas espera-se entender de que forma são utilizadas para melhoria de performance na aplicação.

3. METODOLOGIA

Para realização desse estudo, foi realizado um processo constituído por 4 etapas:

1. Seleção de repositórios de código aberto React;
2. Extração de métricas dos repositórios;
3. Identificação de outliers;
4. Análise de componentes outliers e métricas.

3.1 Seleção de repositórios

Para que a análise seja consistente com a produção de softwares de larga escala, a seleção de repositórios a serem estudados foi realizada de forma que respeitassem os seguintes critérios:

- **Projeto de código aberto disponível na plataforma GitHub**
  - A possibilidade de acesso ao código fonte é um critério essencial para extração de dados e análise. Desse modo, a busca por repositórios se deu através da plataforma GitHub, por ser a maior coleção de softwares de código aberto do mundo [12], utilizando as palavras-chaves *react* e *front-end* para filtrar repositórios correspondentes ao escopo de estudo.
- **Projetos com pelo menos 10 colaboradores**
  - Este indicador define que o projeto foi desenvolvido por uma equipe, o que sugere um desenvolvimento sem viés individual ao longo da construção do código e reflete práticas reais de um trabalho coletivo.
- **Projetos ativos**
  - A fim de garantir que os projetos analisados representem práticas atuais de desenvolvimento, foram selecionados repositórios que apresentem contribuições nos últimos 4 meses - considerando o período de realização desse trabalho -, sendo assim considerados projetos ativos.
- **Projetos com no mínimo 200 estrelas**
  - A quantidade de estrelas de um repositório pode ser um bom indicador de que o projeto foi bem aclamado pela comunidade e portanto apresenta estruturas e padrões aceitos por membros desta.
- **Projetos para usuários finais**
  - Critério definido posteriormente para definir que os projetos analisados sejam apenas para

usuários finais e reflitam características desse tipo de desenvolvimento, haja vista que *frameworks* podem apresentar aspectos de desenvolvimento muito particulares.

- **Projetos com código majoritariamente constituído por componentes funcionais**
    - Desde o lançamento dos hooks, o desenvolvimento de componentes funcionais ganhou grande espaço no ecossistema React e aparenta ser a tendência de desenvolvimento para sistemas futuros.

3.2 Extração de métricas dos repositórios

A fim de extrair dados sobre as métricas estabelecidas no ponto anterior foi desenvolvido um *script* utilizando a linguagem Python que consiste em: realizar a varredura em arquivos e pastas de um repositório e, para cada arquivo, realizar a leitura de cada linha individualmente e identificar a ocorrência de dados correspondentes às métricas selecionadas a partir de expressões regulares.[1]

Ao fim da execução do programa, os dados sobre todos os arquivos do repositório são armazenados em uma planilha.

3.3 Identificação de outliers

Para designar o que são componentes e arquivos *outliers* nos projetos, fez-se uso do código anterior para extração dos dados e, para cada métrica em cada projeto, foi aplicada a fórmula padrão para definição de *outliers* em uma distribuição de dados:

Definem-se um quartil inferior (Q1) correspondente ao 25º percentil e um quartil superior (Q3) correspondente ao 75º percentil. Em seguida, define-se a diferença interquartil (IQR) dada por: **IQR = Q3 - Q1**.

Dada a diferença interquartil, são potenciais *outliers* todos os dados que compreendam a alguma das seguintes situações:

X < Q1 - 1.5 * IQR ou **X > Q3 + 1.5 * IQR** em que X é o dado analisado e candidato a outlier. [9]

3.4 Análise de componentes *outliers* e métricas

Após o levantamento de *outliers* em cada projeto, discorre-se sobre cada uma das métricas e é feita uma análise dos principais componentes de destaque para essas métricas a partir dos dados e leitura do código. Essa análise leva em consideração o significado das métricas e o que elas podem significar para a qualidade do código, bem como as documentações de referência para desenvolvimento React a fim de levantar discussões acerca de decisões arquiteturais, estratégias de modularização do código, função dos componentes e a existência ou fuga de padrões entre os projetos.

4. RESULTADOS

Nesta seção iremos definir os repositórios escolhidos com base nos critérios selecionados e realizar uma análise destes baseando-se em cada uma das métricas estabelecidas. Destaca-se que nas análises realizadas, para algumas das métricas - mais especificamente alguns hooks React - não foram encontrados *outliers* no projeto Cypress RWA em virtude da não utilização destes no sistema.

[1] Código disponível no GitHub.

Nessa análise, veremos características, funções, comparações entre diferentes outliers para cada um dos projetos, bem como padrões ou divergências na estruturação dos códigos dos sistemas escolhidos.

4.1 Seleção de repositórios

Seguindo esses critérios, foram selecionados inicialmente 10 repositórios, dos quais 4 foram descartados por não atenderem estritamente a cada uma das condições. Dos restantes, os 4 projetos mais atualizados - com contribuições mais recentes até o momento - foram selecionados para análise. Dessa forma, temos:

- **OpenShift:** Selecionado para análise.
- Takenote: Descartado por apresentar última contribuição fora do prazo definido, sendo considerado um projeto inativo.
- HospitalRun: Descartado por critério de contribuição mais recente.
- Simorgh: Descartado por se tratar de uma framework.
- Excalidraw: Descartado por apresentar elevado uso de classes.
- Sentry: Descartado por se tratar de um framework.
- **GoAlert:** Selecionado para análise.
- Spectrum: Descartado por critério de contribuição mais recente.
- **Cypress Real World App:** Selecionado para análise.
- **Fiora:** Selecionado para análise.

4.2 Analisando componentes por métrica

Uma vez que as métricas genéricas podem nos dar indícios a respeito da qualidade de código [3,4] e as chamadas de *hooks* podem ser bons indicadores sobre decisões tomadas na construção de componentes funcionais, a análise de componentes considerando apenas cada métrica individualmente pode nos dar informações a respeito de decisões de arquitetura e o processo construção da aplicação.

4.2.1 Quantidade de linhas

Considerando a quantidade de linhas por projeto, temos a princípio os seguintes dados gerais para cada um dos analisados:

Repositório	**Média**	**Mediana**	**IQR**	**Q1 - 1.5 * IQR**	**Q3 + 1.5 * IQR**
OpenShift	209,10	137	205,5	N/A[2]	568,75
Cypress RWA	79,15	58	76	N/A[2]	219
Fiora	109,94	67	113,25	N/A[2]	304,87
GoAlert	103,32	77	101,5	N/A[2]	292,75

4.2.1.1 Outliers no OpenShift

Com base nos dados extraídos, temos o componente *alerting.tsx* do projeto OpenShift como um dos destaques para esta métrica. Com 1836 linhas, este componente não apenas é muito superior à

2 Valor menor que zero.

média de linhas do projeto, como se apresenta como o maior arquivo de toda a aplicação.

No *alerting.tsx* é possível perceber a definição de uma grande quantidade de funções e constantes utilitárias e também de diversos tipos, além dos já importados, que também se apresentam em grande quantidade. Ainda, temos a criação de sub-componentes de interface visual que servem a criação de componentes maiores no próprio arquivo. Para essa construção, diversos outros elementos, funções e tipos são importados, de forma a tornar o arquivo bastante robusto em quantidade de linhas.

Considerando essas características, não é fácil inferir qual o propósito do componente, dado que vários componentes e funções são exportados deste. Ainda que um componente maior em quantidade de linhas e conectado a componentes de alta ordem, nomeado como *AlertRulesDetailsPage,* seja exportado pelo arquivo, a grande quantidade de elementos definidos e também exportados torna sua compreensão confusa.

Desse modo, uma possibilidade diante da construção desse componente é que **houve acúmulo de responsabilidade ou dificuldade em subdividir elementos em arquivos distintos que pudessem facilitar a compreensão e propósito deste no projeto**, como a criação de um arquivo para as funções utilitárias ou para os tipos, como veremos posteriormente nos outros projetos. Portanto, apresenta os possíveis problemas indicados para essa métrica, como complexidade e dificuldade de manutenção e reutilização.

4.2.1.2 Outliers no Fiora

Outro destaque em quantidade de linhas é o componente *reducer.ts* do projeto Fiora, com 626 linhas. Arquivos de *reducer* são bastante comuns em projetos React que utilizam a biblioteca Redux para gerenciamento de estado da aplicação.

Redux utiliza uma *store* para armazenamento de estado, comumente chamada de "fonte única de verdade", de modo a tornar as mudanças de estado previsíveis e facilmente identificáveis. [5] Para isso, seu funcionamento consiste em um conjunto de regras: o estado deve ser imutável, a informação sobre mudanças no estado devem estar contidas em ações e, por fim, as ações devem passar por reducers, funções puras que atuam definindo a forma como as ações disparadas transformarão o estado na store. [6]

Dessa maneira, arquivos de *reducers* funcionam como funções que recebem ações contendo uma informação sobre mudança de estado para a store e retornam um novo objeto com as informações de estado atualizadas, precisamente como feito no projeto Fiora. [6]

Assim, é esperado que arquivos de *reducers* tenham grande quantidade de linhas, haja vista que além de importar todas as ações, precisam tratar o comportamento esperado de cada uma separadamente, como pode ser visto na Figura 1. No entanto, uma particularidade do *reducer.ts* no Fiora é que além do *switch case* típico [6] para definir a lógica de alterações por ação, o arquivo também apresenta a definição de diversas funções utilitárias e também de tipos correspondentes a estados do Redux, fatores que contribuem para uma elevada quantidade de linhas no arquivo.

```
function reducer(state: State = initialState, action: Action): State {
    switch (action.type) {
        case ActionTypes.Ready: {
            return {
                ...state,
                status: {
                    ...state.status,
                    ready: true,
                },
            };
        }
        case ActionTypes.Connect: {
            return {
                ...state,
                connect: true,
            };
        }
        case ActionTypes.Disconnect: {
            return {
                ...state,
                connect: false,
            };
        }
  (...)
```

Figura 1. Função reducer no arquivo reducer.ts

Embora o projeto conte com uma pasta de tipos, os desenvolvedores do Fiora optaram por utilizá-la apenas para armazenar um arquivo que redefine tipos globais e decidiram manter a declaração de tipos das informações de estado que serão disparados pelas ações e passarão pelo reducer no próprio arquivo do reducer, o que ocasionou uma maior quantidade de linhas.

4.2.1.3 Discussão

De maneira geral, para os arquivos *outliers* para a métrica de quantidade de linhas, temos que casos específicos como a criação de reducers, arquivos utilitários de funções e testes de integração são geralmente esperados e não representam decisões que fogem ao padrão.

Contudo, essa métrica ainda pode ser um indicador de possíveis problemas de legibilidade e manutenibilidade dos componentes [7], situação evidente no *alerting.tsx* do projeto OpenShift e que se repete em arquivos dos demais projetos também.

Assim, é comum para os projetos analisados que, os componentes que não se encaixam nos casos especiais mencionados anteriormente, apresentem acúmulo de responsabilidades geralmente relacionado a uma grande quantidade de funções auxiliares e definição de tipos, mas não se limitando a isso, é possível visualizar com frequência a criação de sub componentes de UI nesses arquivos, o que tende a prejudicar a legibilidade e manutenção destes.

Para esses casos e considerando que alguns dos projetos possuíam estruturas auxiliares como arquivos de tipos e funções utilitárias mas ainda assim apresentaram componentes com as características citadas, supõe-se um crescimento inesperado e desordenado dos componentes durante o processo de desenvolvimento que o levaram a serem caracterizados como *outliers* para essa métrica e até para outras, como veremos posteriormente.

4.2.2 Quantidade de importações

Em termos gerais, possuímos os seguintes dados considerando a métrica quantidade de importações:

Repositório	Média	Mediana	IQR	Q1 - 1.5 * IQR	Q3 + 1.5 * IQR
OpenShift	8,84	8	8	N/A[3]	24
Cypress RWA	5,04	4	6	N/A[3]	16
Fiora	6,72	4	8	N/A[3]	22
GoAlert	6,34	5	7	N/A[3]	19,5

Para a quantidade de imports, podemos destacar os elementos *catalog-item-icon.tsx,* mais uma vez o *alerting.tsx,* do projeto OpenShift e também o *PrivateRoutesContainer.tsx* do projeto Cypress RWA.

4.2.2.1 Outlier no OpenShift

O arquivo *catalog-item-icon.tsx* apresenta 99 imports, número muito superior à média do projeto e atua como um catálogo de ícones, de modo que quase todas as importações são de ícones. Como um catálogo de ícones, o componente também apresenta funções utilitárias relacionadas a estes e exporta componentes de interface para renderização personalizável de ícones. **Assim, esse componente representa uma decisão do projeto em criar um serviço para lidar com ícones.**

4.2.2.2 Outlier no Cypress RWA

Tratando-se do *PrivateRoutesContainer.tsx*, o destaque para a métrica de importações do projeto Cypress RWA com 16 importações, este componente é responsável por encapsular rotas protegidas na aplicação, como destacado na Figura 2, e portanto deve importar todos os componentes que serão renderizados nessas rotas, bem como os serviços envolvidos na autorização de acesso para cada página.

```
return (
  <MainLayout notificationsService={notificationsService} authService={authService}>
    <UserOnboardingContainer
      authService={authService}
      bankAccountsService={bankAccountsService}
    />
    <Switch>
      <PrivateRoute isLoggedIn={isLoggedIn} exact path={"/(public|contacts|personal)?"}>
        <TransactionsContainer />
      </PrivateRoute>
      <PrivateRoute isLoggedIn={isLoggedIn} exact path="/user/settings">
        <UserSettingsContainer authService={authService} />
      </PrivateRoute>
      <PrivateRoute isLoggedIn={isLoggedIn} exact path="/notifications">
        <NotificationsContainer
          authService={authService}
          notificationsService={notificationsService}
        />
      </PrivateRoute>
      <PrivateRoute isLoggedIn={isLoggedIn} path="/bankaccounts*">
        <BankAccountsContainer
          authService={authService}
          bankAccountsService={bankAccountsService}
        />
      </PrivateRoute>
      <PrivateRoute isLoggedIn={isLoggedIn} exact path="/transaction/new">
        <TransactionCreateContainer authService={authService} snackbarService={snackbarService} />
      </PrivateRoute>
      <PrivateRoute isLoggedIn={isLoggedIn} exact path="/transaction/:transactionId">
        <TransactionDetailContainer authService={authService} />
      </PrivateRoute>
    </Switch>
  </MainLayout>
);
```

Figura 2. Estrutura do componente PrivateRoutesContainer.tsx do projeto Cypress RWA

3 Valor menor que zero.

4.2.2.3 Gerenciamento de rotas

Nesse sentido, é válido compreender as abordagens dos demais projetos para o gerenciamento de rotas: no projeto Fiora não há uso de rotas distintas, enquanto que no GoAlert as rotas são definidas em arquivos distintos e específicos - como vemos no *AdminRouter.js*, na Figura 3 - e seguem a mesma estrutura, encapsulando a lógica de rotas por setor.

```
function AdminRouter() {
  const { isAdmin, ready } = useSessionInfo()
  if (!ready) {
    return <Spinner />
  }
  if (!isAdmin) {
    return <GenericError error='Access Denied' />
  }

  return (
    <Switch>
      <Route exact path='/admin/config' component={AdminConfig} />
      <Route exact path='/admin/limits' component={AdminLimits} />
      <Route exact path='/admin/toolbox' component={AdminToolbox} />
      <Route component={PageNotFound} />
    </Switch>
  )
}
```

Figura 3. Componente AdminRouter do projeto GoAlert

Os diferentes arquivos de rotas do GoAlert são utilizados em um arquivo principal denominado *routes.js* que faz o mapeamento de endereço e domínio de rotas para os respectivos arquivos e é renderizado no arquivo principal *App.js*, de modo que este projeto apresenta a maior modularização dentre todos para este propósito específico. **Dessa forma, os diferentes arquivos de rotas apresentam valores mais baixos para a métrica de importações e não se caracterizam como *outliers*, o que é um bom sinalizador de boa decisão de estruturação de código para esse contexto quando comparado aos dos demais projetos.**

Ademais, temos também no OpenShift uma divisão de rotas em arquivos, mas que não apresentam o papel de arquivos específicos para rotas como no GoAlert. A definição de rotas no projeto é realizada em componentes onde estas serão utilizadas, estando divididas em 4 arquivos distintos: *app-content.tsx*, que apresenta em sua maioria a declaração de rotas que são carregadas por meio de *lazy loading*; *app.jsx* contendo rotas principais da aplicação; *alerting.tsx* contendo rotas correspondentes à página declarada no arquivo e *horizontal-nav.tsx* que apresenta rotas para navegação. Destes, apenas o *app.jsx* e *alerting.tsx* são classificados como *outliers* para a métrica de importações, seguindo a tabela abaixo.

Repositório	Arquivo	Qtd. importações
OpenShift	*app-content.tsx*	17
OpenShift	*app.jsx*	**39**
OpenShift	*alerting.tsx*	**45**
OpenShift	*horizontal-nav.tsx*	13

Por fim, no Cypress RWA, temos o container já apresentado para as rotas privadas que é renderizado no arquivo principal da aplicação - *App.tsx* - em conjunto com as demais rotas.

Dessa forma, percebemos que cada projeto lida com o gerenciamento de formas distintas e, embora haja certo esforço para modularização na declaração dessas, não há um padrão em como isso é realizado, de modo que o único aspecto em comum é a renderização final dos componentes de rotas no arquivo principal da aplicação, geralmente o arquivo *App.(js | tsx)*.

4.2.3 Quantidade de exportações

Para esta métrica, temos os seguintes dados por projeto:

Repositório	**Média**	**Mediana**	**IQR**	**Q1 - 1.5 * IQR**	**Q3 + 1.5 * IQR**
OpenShift	4,67	2	4	N/A[4]	11
Cypress RWA	2,49	2	1	N/A[4]	3,5
Fiora	1,87	1	0	1	1
GoAlert	1,47	1	0	1	1

4.2.3.1 Declaração de tipos no OpenShift

Levando em conta a métrica de exportações, um dos grandes destaques é o arquivo *types.ts* do projeto OpenShift localizado na pasta do módulo “k8s”. Neste projeto uma das características de organização dos arquivos é a criação de módulos, organizados em pastas. Assim, o arquivo *types.ts* possui o papel de centralizar a maioria das definições de tipos do seu respectivo módulo.

Embora os autores do OpenShift tenham optado pela criação de alguns arquivos de propósito exclusivo para definição de tipos distribuídos em pastas no projeto, essa prática não se restringe apenas a estes arquivos. Dessa maneira, há declaração de tipos em outros arquivos ao longo da aplicação e não existe um padrão aparente relacionado à criação de um arquivo de tipos para todos os componentes ou pastas.

4.2.3.2 Declaração de tipos no Cypress RWA

Seguindo uma abordagem um pouco diferente, o projeto Cypress RWA possui uma pasta denominada *models* contendo apenas arquivos para definição de tipos organizados de modo que indica a criação de modelos que correspondem a entidades do banco de dados. Os demais componentes possuem definições de tipos utilizados internamente ou exportados sempre que necessário.

4.2.3.3 Declaração de tipos no Fiora

Como já discutido anteriormente, o projeto Fiora não possui arquivos dedicados à definição de tipos, apresentando apenas um arquivo de tipo cujo propósito é a declaração e sobreposição de tipos globais, seguindo uma estratégia diferente dos demais projetos e definindo os tipos necessários diretamente em arquivos que o utilizam e exportando para os demais quando necessário.

4.2.3.4 Declaração de tipos no GoAlert

No que concerne à estratégia para declaração de tipos, o projeto GoAlert se diferencia dos demais por não apresentar um uso uniforme de TypeScript, de maneira que vários arquivos fazem uso de JavaScript puro e portanto não apresentam tipos, enquanto outros estão escritos em TypeScript e possuem os tipos declarados nos próprios arquivos, sem uso de arquivos auxiliares. No entanto, para os testes de integração do projeto há uma pasta específica para organização de arquivos cuja única função é definir os tipos utilizados nesses testes.

4.2.4 Quantidade de funções

Para esta métrica, temos os seguintes dados por projeto:

Repositório	**Média**	**Mediana**	**IQR**	**Q1 - 1.5 * IQR**	**Q3 + 1.5 * IQR**
OpenShift	5,69	3	6	N/A[5]	16
Cypress RWA	1,7	1	2	N/A[5]	5
Fiora	3,28	2	2,5	N/A[5]	7
GoAlert	2,49	1	2	N/A[5]	6

Considerando a métrica de quantidade de funções por arquivo, é possível observar um padrão presente em todos os projetos: cada um possui ao menos um arquivo considerando *outlier* a métrica cujo propósito é definir e exportar funções utilitárias para componentes do projeto. Os representantes desse tipo de arquivo para cada projeto são:

Repositório	**Arquivo**	**Qtd. funções**
OpenShift	*util.ts*	22
Cypress RWA	*transactionUtils.ts*	34
Fiora	*service.ts*	30
GoAlert	*util.js*	18

É perceptível que cada um desses arquivos apresenta quantidade de funções muito superior à média dos respectivos projetos e dos projetos como um todo. O papel desses arquivos é bastante semelhante para quase todos os projetos: servem ao propósito de definir funções que serão utilizadas em um ou mais componentes na aplicação.

4.2.4.1 Reutilização de código no Fiora

No projeto Fiora, uma estratégia um pouco diferente é seguida quando comparado aos demais projetos: o repositório conta com

4 Valor menor que zero.

5 Valor menor que zero.

uma organização em pacotes e pastas distintas para o projeto web, aplicativo, servidor e outras pastas de utilidade (como por exemplo a i18n para armazenar arquivos de internacionalização). Dentre estas pastas, há uma pasta denominada *utils* que contém funções auxiliares que são utilizadas em qualquer um dos demais repositórios, definindo uma estratégia para maximizar o reuso de funções através das diferentes partes do sistema.

Desse modo, o arquivo *service.ts* do Fiora apresenta um propósito diferente dos arquivos *outliers* para a métrica de quantidade de funções quando comparado aos demais projetos: ele define chamadas assíncronas para o servidor, funcionando de fato como um serviço, porém um serviço genérico - que contém a maioria das requisições HTTP utilizadas no sistema web.

4.2.4.2 Discussão

Para todos os demais projetos, os arquivos *outliers* atuam como uma biblioteca de funções utilitárias, que trabalham apenas com os parâmetros definidos nelas, da mesma maneira que as funções definidas em arquivos da pasta *utils* externa ao repositório web no projeto Fiora, de modo que se evidencia assim um padrão para reutilização de código que caracteriza estes arquivos como *outliers* para a métrica em questão.

Por essa característica, dado que o propósito dessas funções é serem reutilizadas, esses componentes também são *outliers* para a métrica de quantidade de exportações, de modo que temos:

Repositório	**Arquivo**	**Qtd. exportações**
OpenShift	*util.ts*	22
Cypress RWA	*transactionUtils.ts*	40
Fiora	*service.ts*	33
GoAlert	*util.js*	11

Dessa forma, é possível observar que a quantidade de exportações é muito próxima da quantidade de funções para cada um dos arquivos. Assim, **os *outliers* apresentados para essa métrica apresentam características típicas e padronizadas entre os repositórios.**

4.2.5 Ocorrências de useEffect

Para esta métrica, todos os projetos apresentaram medianas e IQR como zero, e média 0,22, 0,18, 0,18 e 0,08 para os projetos OpenShift, Cypress RWA, Fiora e GoAlert.

Da mesma forma que veremos a seguir para os demais hooks, é perceptível o baixo valor de média e mediana para todos os projetos. Logo, qualquer componente que faça uso deste será caracterizado como *outlier* para a métrica, o que não necessariamente representa características fora do padrão para a construção destes.

Dado que o *useEffect* é um hook do React voltado para a execução de efeitos colaterais em componentes funcionais [8], temos a princípio uma observação inicial: todos os componentes *outliers* para esta métrica seguem esse princípio. Assim, temos três principais casos de uso para o uso desse hook nos projetos analisados: busca de dados por requisições HTTP, causar mudanças de estado com base na alteração de outro estado ou propriedade e subscrição e remoção de subscrição para eventos ou timers.

4.2.5.1 Requisições com useEffect

O primeiro caso de uso para este hook é a chamada de requisições HTTP para busca de dados, processo que geralmente é realizado após a construção dos elementos no DOM. [8]

Alguns exemplos de *outliers* que apresentam esse caso de uso são:

- **OpenShift:** *resource-log.tsx* e *metrics.tsx*
- **Cypress RWA:** *AppOkta.tsx* e *AppAuth0.tsx*
- **Fiora:** *Chat.tsx*
- **GoAlert:** *Login.js*

Um exemplo simples de uso desse hook para buscar dados uma única vez - após a montagem do componente no DOM - é o utilizado no componente *Login.js* do projeto GoAlert, visível na Figura 4.

```
useEffect(() => {
  // get providers
  fetch(PROVIDERS_URL)
    .then((res) => res.json())
    .then((data) => setProviders(data))
    .catch((err) => setError(err))
}, [])
```

Figura 4. Uso do hook useEffect para busca de dados no Login.js do projeto GoAlert

4.2.5.2 Alterações de estado

Outro caso comum de efeitos colaterais no qual o uso do *useEffect* é pertinente é provocar mudanças de estado a partir de alterações em um outro estado.

Como exemplos de *outliers* que apresentam esse caso de uso, temos:

- **OpenShift:** *resource-log.tsx*
- **Cypress RWA:** *MainLayout.tsx*
- **Fiora:** *ChatInput.tsx*
- **GoAlert:** *Search.js*

Nesse caso, o *useEffect* atua a partir da verificação de mudanças das propriedades passadas no array de dependências e executando a função recebida caso alguma dessas propriedades apresente alterações no valor.

Tomando o *resource-log.tsx* como exemplo, temos na Figura 5 que o hook executa uma função que faz uma verificação condicional e modifica o estado *stale* caso as condições sejam verdadeiras, estas condições por sua vez são baseadas nas propriedades *previousResourceStatus* e *resourceStatus* e o bloco é executado sempre que houver mudanças em alguma dessas duas propriedades. Desse modo, temos a possibilidade de alteração do estado *stale* quando *previousResourceStatus ou resourceStatus* apresentarem mudanças em seus valores.

```
// If container comes out of restarting state, currently displayed logs might be stale
React.useEffect(() => {
  if (
    previousResourceStatus === LOG_SOURCE_RESTARTING &&
    resourceStatus !== LOG_SOURCE_RESTARTING
  ) {
    setStale(true);
  }
}, [previousResourceStatus, resourceStatus]);
```

Figura 5. Uso do hook useEffect para mudança de estado como efeito colateral no resource-log.tsx do projeto OpenShift

4.2.6 Ocorrências de useState

Para esta métrica, todos os projetos apresentaram medianas e IQR como zero, e média 0,57, 0,06, 0,84 e 0,48 para os projetos OpenShift, Cypress RWA, Fiora e GoAlert respectivamente.

O hook *useState* é uma adição do React que permite o uso de estado em componentes funcionais, eliminando assim a necessidade de se escrever uma classe para isso. [8]

Considerando o propósito da métrica, temos que os *outliers* para esta são em geral componentes que possuem *forms* ou outros tipos de *inputs* de usuários que causem alterações na interface.

No entanto, alguns dos componentes merecem atenção, como: *query-browser.tsx* do projeto OpenShift e o *App.tsx* do projeto Fiora.

4.2.6.1 Outlier no OpenShift

Com 14 ocorrências do hook *useState*, o arquivo *query-browser.tsx* é um dos principais *outliers* para esta métrica.

Embora apenas uma ocorrência do hook seja suficiente para um componente ser caracterizado como *outlier* para a métrica, o motivo de esse componente se destacar tanto dos demais se diferencia um pouco quando comparado a outros arquivos, cujo número elevado de ocorrências está normalmente associado à presença de formulários.

O motivo de o *query-browser.tsx* apresentar tantos hooks de estado é a declaração de múltiplos sub-componentes funcionais que possuem estados próprios, **o que representa um indício de ausência de modularização e provável acúmulo de responsabilidades do arquivo**.

4.2.6.2 Outlier no Fiora

No projeto Fiora, o que chama atenção é o fato de o principal *outlier* para esta métrica ser também o arquivo principal da aplicação - o *App.tsx* - com 8 ocorrências de *useState*.

Nesse sentido, há dois fatores que chamam a atenção para este arquivo: a grande quantidade de estados no arquivo principal e a ausência de compartilhamento de estado com outros componentes de forma global, geralmente associada a contextos, como é possível encontrar nos projetos OpenShift e GoAlert.

Assim, **essa forma de organização de código e utilização excessiva de estados isolados no arquivo principal foge ao padrão quando comparado aos demais projetos e apresenta ocorrências de useState acima da média dos outros projetos para esse arquivo.**

4.2.7 Ocorrências de useMemo

Sendo mais uma métrica de baixa ocorrência, todos os projetos apresentaram medianas, e IQR como zero, e média 0,07, 0, 0,01 e 0,0059 para os projetos OpenShift, Cypress RWA, Fiora e GoAlert.

Considerando o propósito deste hook, seu uso é consistente em todos os projetos que o utilizam. Tendo em vista que os dados indicam baixa ocorrência geral do *useMemo* nos projetos, ao analisarmos os componentes percebemos que os *outliers* não indicam qualquer característica especial de construção, apenas demonstram a presença do hook.

Dessa maneira, temos que o hook em questão é utilizado para armazenar valores que são passados como *props* para componentes filhos e evitar renderizações desnecessárias destes quando as propriedades transmitidas não forem alteradas. [8]

4.2.7.1 Economia de operações custosas

O componente *Avatar.tsx* do projeto Fiora se destaca como um bom exemplo de uso do *useMemo,* como podemos ver na Figura 6, para evitar execução de operações custosas e renderizações desnecessárias, haja vista que o valor observado é resultado de operações assíncronas e expressões regulares, como vemos na Figura 7, cujos processamentos podem ser lentos.

```
    const url = useMemo(() => {
        if (/^(blob|data):/.test(src)) {
            return src;
        }
        return getOSSFileUrl(
            src,
            `image/resize,w_${size * 2},h_${size * 2}/quality,q_90`,
        );
    }, [src]);
```

Figura 6. Uso de useMemo no componente Avatar.tsx do projeto Fiora.

```
export function getOSSFileUrl(url = '', process = '') {
    const [rawUrl = '', extraPrams = ''] = url.split('?');
    if (ossClient && rawUrl.startsWith('oss:')) {
        const filename = rawUrl.slice(4);
        // expire 5min
        return `${ossClient.signatureUrl(filename, { expires: 300, process })}${
            extraPrams ? `&${extraPrams}` : ''
        }`;
    }
    if (/\/\/cdn\.suisuijiang\.com/.test(rawUrl)) {
        return `${rawUrl}?x-oss-process=${process}${
            extraPrams ? `&${extraPrams}` : ''
        }`;
    }
    return `${url}`;
}
```

Figura 7. Função assíncrona chamada para calcular valor armazenado pelo useMemo

4.2.8 Ocorrências de useContext

Mais uma vez considerando os hooks de React, temos que todos os projetos apresentaram medianas e IQR como zero, com média 0,02, 0, 0,04 e 0,01 para os projetos OpenShift, Cypress RWA, Fiora e GoAlert.

Considerando que o propósito do hook *useContext* é o compartilhamento de estado, observamos que os *outliers* para essa métrica apresentam dois casos de uso distintos: compartilhamento de estado global e não-global e que o projeto Cypress RWA não apresenta ocorrências do hook.

4.2.8.1 Compartilhamento de estado global

O uso de contexto para compartilhamento de estado em âmbito global é o mais encontrado dentre os projetos, estando presente em arquivos como o *PageAction.js* do GoAlert e o *FunctionBar.tsx* do Fiora, porém não se encontra presente no projeto OpenShift.

4.2.8.2 Compartilhamento de estado não-global

Além de compartilhar estado com toda a aplicação, outra possibilidade é o compartilhamento de estado em um contexto mais específico, passando adiante informações que sejam usadas por componentes descendentes de um nó mais abaixo na hierarquia da aplicação.

Esse tipo de aplicação do *useContext* é bastante encontrado no projeto OpenShift, que possui uma estrutura de módulos e utiliza o hook para compartilhar estados para qualquer componente contido em alguns dos módulos utilizados. Um exemplo é o componente *utilization-card.tsx* que consome o contexto *ProjectDashboardContext*, o qual compartilha estados com alguns componentes na aplicação mas não de maneira global.

4.2.9 Ocorrências de useCallback

De modo semelhante aos demais hooks, todos os projetos apresentaram medianas e IQR como zero, e média 0,07, 0, 0,03 e 0,0088 para os projetos OpenShift, Cypress RWA, Fiora e GoAlert.

Sendo mais um exemplo de hook com baixas ocorrências nos projetos (e não utilizado no Cypress RWA, por exemplo), temos novamente ao analisar os componentes *outliers* que estes não representam características especiais, apenas fazem uso dessa funcionalidade.

De maneira geral, por funcionar de modo semelhante ao *useMemo*, o *useCallback* se caracteriza por memorizar declarações de funções e evitar que estejas sejam redefinidas desnecessariamente e provoquem renderizações extras em componentes filhos que as recebem por meio de *props*.

Assim, temos o componente *useBoolean.ts*, um *custom hook* do projeto OpenShift, como um bom exemplo de uso característico de *useCallback*, visível na Figura 8, dado que retorna funções que serão utilizadas em diferentes componentes e economizam o custo de múltiplas renderizações destes considerando a imutabilidade para a referência das funções recebidas.

```
import * as React from 'react';

export const useBoolean = (
  initialValue: boolean,
): [boolean, () => void, () => void, () => void] => {
  const [value, setValue] = React.useState(initialValue);
  const toggle = React.useCallback(() => setValue((v) => !v), []);
  const setTrue = React.useCallback(() => setValue(true), []);
  const setFalse = React.useCallback(() => setValue(false), []);
  return [value, toggle, setTrue, setFalse];
};
```

Figura 8. Uso do hook useCallback no custom hook useBoolean do projeto OpenShift

5. CONCLUSÕES

Analisando diferentes projetos de grande porte, fica nítido o desafio de estruturação e arquitetura dos softwares front-end. O desenvolvimento em componentes apresenta muitos desafios nesse sentido e é difícil estabelecer padrões não apenas para a arquitetura de software como também para organização de pastas e arquivos.

Podemos perceber com base nas métricas que grande parte dos desafios envolvidos estão relacionados à modularização. É possível notar diversos componentes que aparentam ter acumulado mais responsabilidades do que deveriam, provavelmente à medida que o próprio sistema cresceu, o que fez com que apresentassem dados fora da curva para diversas métricas, mas principalmente as advindas da literatura de engenharia de software: como linhas de código e quantidade de funções.

Nesse sentido, temos que meio ao desafio da modularização, observamos formas distintas de organização de código que foram determinantes para definir certos componentes como *outliers* para algumas das métricas e que chamam atenção: gerenciamento de rotas e compartilhamento de estado são alguns destaques nesse sentido, nos quais as decisões tomadas fizeram a diferença para que componentes de mesmo papel fossem classificados como *outliers* em alguns projetos e não em outros.

Diante dessas diferenças, podemos tomar lições importantes das decisões de cada projeto, como por exemplo a forma de organizar rotas do GoAlert. A divisão das rotas em arquivos distintos (como rotas para usuário, rotas para administração, etc) e sua utilização num arquivo global de rotas, aspecto encontrado apenas neste projeto, parece uma solução eficaz para escalar o sistema de forma adequada e evitar a criação de mais arquivos *outliers* para as métricas analisadas, de modo a reduzir os possíveis problemas associados a cada uma delas.

Outra característica interessante é a forma como o projeto OpenShift dividiu o compartilhamento de estado em módulos distintos utilizando contextos, evidenciando que nem sempre que houver a necessidade de compartilhar estado para níveis distintos de componentes eles devem ser levados a nível global da aplicação. Ainda, percebemos soluções distintas para a maneira como o compartilhamento de estado é realizado entre os diferentes projetos, como o uso de contexto em alguns destes e o uso do Redux no Fiora, salientando que não há uma maneira uniforme de tratar essa questão mas que ambas são eficazes.

No entanto, além de características de destaque que aparentam agregar valor aos projetos, percebemos também decisões diferentes que não necessariamente são benéficas, como por exemplo o uso conjunto de contexto do React e Redux no projeto Fiora. Dado que ambas as alternativas resolvem o mesmo problema de compartilhamento de estado em diferentes níveis, a utilização concomitante destas pode causar confusão para os desenvolvedores à medida que o sistema cresce, uma vez que uma delas terá de ser escolhida para cada estado que tiver de ser compartilhado para componentes em níveis distantes. Ainda, temos que os componentes relacionados às formas utilizadas se mostraram como *outliers* para métricas distintas, evidenciando características diferentes na construção de uma solução de mesmo propósito no sistema.

Apesar de todas as diferenças entre as bases de código analisadas, é válido ressaltar que certos padrões podem ser encontrados e utilizados como referência em outros projetos front-end, como por exemplo a forma como se usam os hooks do React de maneira geral e os arquivos de serviços e funções utilitárias.

Observamos que os *outliers* para as métricas de *useEffect*, *useMemo*, *useContext* e *useCallback* apresentam uso consistente e semelhante nos projetos, de modo que o fato de terem se destacado para essas métricas não aponta problemas relacionados à qualidade de software.

Por fim, temos que arquivos de funções utilitárias estão presentes em todos os projetos, apresentando destaque para as métricas de quantidade de funções e exportações. Embora o projeto Fiora apresente uma localização diferente para permitir compartilhamento de código com outros sistemas, temos que o uso de arquivos utilitários para serviços é comum a todos, representando um padrão entre os projetos e também para métricas nos quais os componentes se caracterizam como *outliers*.

Diante disso, embora o desafio de construir a arquitetura de um software estruturado em componentes fique ainda mais evidente mediante à análise de grandes repositórios, podemos identificar padrões e extrair lições de boas decisões tomadas durante o processo de desenvolvimento, especialmente pela forma que elas impactam as métricas da engenharia de software e se comportam mediante às métricas de hooks.

6. REFERÊNCIAS

[1] Most Popular Technologies. StackOverflow, 2020. Disponível em: https://insights.stackoverflow.com/survey/2020#most-popular-technologies. Acesso em: 2 de ago. de 2021.

[2] Sultana, Kazi Zakia. "Towards a software vulnerability prediction model using traceable code patterns and software metrics." 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2017.

[3] Lin, Yiwei, et al. "A Code Quality Metrics Model for React-Based Web Applications." International Conference on Intelligent Computing. Springer, Cham, 2017.

[4] Barkmann, H., Lincke, R., & Löwe, W. (2009, May). Quantitative evaluation of software quality metrics in open-source projects. In 2009 International Conference on Advanced Information Networking and Applications Workshops (pp. 1067-1072). IEEE.

[5] Caspers, M. K. (2017). React and redux. Rich Internet Applications w/HTML and Javascript, 11.

[6] Conceitos principais do Redux. Documentação oficial. https://redux.js.org/introduction/core-concepts. Acesso em: 12 de set. de 2021.

[7] Lee, M. C. (2014). Software quality factors and software quality metrics to enhance software quality assurance. British Journal of Applied Science & Technology, 4(21), 3069-3095.

[8] Documentação oficial do React. https://reactjs.org/. Acesso em: 19 de set. de 2021.

[9] Walfish, S. (2006). A review of statistical outlier methods. Pharmaceutical technology, 30(11), 82.

[10] History of front-end frameworks. https://blog.logrocket.com/history-of-frontend-frameworks/. Acesso em: 23 de set. de 2021.

[11] Gackenheimer, C. (2015). Introducing flux: An application architecture for react. In Introduction to React (pp. 87-106). Apress, Berkeley, CA.

[12] Borges, H., Hora, A., & Valente, M. T. (2016, October). Understanding the factors that impact the popularity of GitHub repositories. In 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME) (pp. 334-344). IEEE.

[13] Lincke, R., & Löwe, W. (2006, July). Validation of a standard-and metric-based software quality model. In Proceedings of the 10th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2006) (pp. 81-90).

[14] Ferreira, K. A., Bigonha, M. A., Bigonha, R. S., Mendes, L. F., & Almeida, H. C. (2012). Identifying thresholds for object-oriented software metrics. Journal of Systems and Software, 85(2), 244-257.