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

EZEQUIAS DE OLIVEIRA ROCHA

ESTUDO E ANÁLISE DE TÉCNICAS PARA MELHORAR DESEMPENHO DE SISTEMAS FRONT-END COM REACT

CAMPINA GRANDE - PB

2023

EZEQUIAS DE OLIVEIRA ROCHA

ESTUDO E ANÁLISE DE TÉCNICAS PARA MELHORAR DESEMPENHO DE SISTEMAS FRONT-END COM REACT

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. Cláudio de Souza Baptista

CAMPINA GRANDE - PB

2023

EZEQUIAS DE OLIVEIRA ROCHA

ESTUDO E ANÁLISE DE TÉCNICAS PARA MELHORAR DESEMPENHO DE SISTEMAS FRONT-END COM REACT

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. Cláudio de Souza Baptista

Orientador – UASC/CEEI/UFCG

Professora Dra. Joseana Macêdo Fechine Régis de Araujo

Examinador – UASC/CEEI/UFCG

Professor Tiago Lima Massoni

Professor da Disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em: 14 de Fevereiro de 2023.

CAMPINA GRANDE - PB

ABSTRACT

Web application systems are extremely important for large companies today. These systems can have a high level of complexity, requiring more performance to succeed in the competitive market and deliver a better interactive experience for the users. In this work, some techniques that can be used to improve the performance of a web application using React, one of the most common libraries today, are described and analyzed. With this in mind, a literature and development community search was conducted to obtain techniques such as code-splitting, lazy load, and code minification. Finally, these techniques were evaluated in an existing real system, in which data was collected following some metrics, such as first content rendering and time to interactivity, to observe the efficiency of these practices, starting a catalog of good and bad practices for front-end system performance.

Estudo e Análise de Técnicas para Melhorar Desempenho de Sistemas Front-end com React

Ezequias de Oliveira Rocha
ezequias.rocha@ccc.ufcg.edu.br
Universidade Federal de Campina Grande
Campina Grande, Paraíba

Cláudio de Souza Baptista
baptista@computacao.ufcg.edu.br
Universidade Federal de Campina Grande
Campina Grande, Paraíba

Hugo Feitosa de Figueirêdo
hugo.figueiredo@ifpb.edu.br
Instituto Federal da Paraíba
Campina Grande, Paraíba

RESUMO

Os sistemas de aplicação web são de extrema importância para as grandes empresas atualmente. Esses sistemas podem ter um nível de complexidade elevado, precisando, desta forma, de mais desempenho para ter sucesso no mercado competitivo e entregar uma experiência de interação melhor para os usuários. Neste trabalho, são descritas e analisadas algumas técnicas que podem ser utilizadas para aumentar o desempenho de uma aplicação web que utiliza React, uma das bibliotecas mais recorrentes na atualidade. Para tal fim, foi realizada uma pesquisa na literatura e em comunidades de desenvolvimento para obter técnicas, como a de *code-splitting*, *lazy load* e de minificação de código. Em seguida, essas técnicas avaliadas em um sistema real já existente, no qual foram coletados dados seguindo algumas métricas, como a de primeira renderização de conteúdo e a de tempo até interatividade, para ser possível observar a eficiência dessas práticas, iniciando, desta forma, uma catalogação das boas e más práticas para o desempenho dos sistemas front-end.

Palavras-chave

React, desempenho, re-render, desenvolvimento web, otimização.

1. INTRODUÇÃO

O desenvolvimento de uma aplicação web moderna se tornou uma tarefa importante para as empresas, e as equipes de desenvolvimento começaram a busca por maneiras de desenvolver, dar manutenção e implantar as aplicações de maneira eficaz, visto que sistemas com uma lógica de negócios mais complexa precisam ter alta confiabilidade, alta escalabilidade, alto desempenho e precisam lidar com uma grande quantidade de dados. Além disso, os usuários utilizam as páginas web em diversas circunstâncias, variando a velocidade da rede e o poder de processamento da máquina que está sendo utilizada, onde, por exemplo, cerca de 55% das pessoas acessam os sites por meio de dispositivos móveis [1] e, de acordo com um artigo da Google, conforme o tempo de carregamento da página aumenta de um até sete segundos, a probabilidade do visitante abandoná-lo cresce 113% [2].

Esta necessidade de melhorar a experiência do usuário aconteceu com o Facebook em 2011, momento em que o seu aplicativo de anúncios começou a ganhar grande notoriedade, ocasionando grandes quantidades de atualizações em seu código e dificuldades para dar manutenção. Na tentativa de ajudar a solucionar esse problema, Jordan Walke criou o FaxJs, que viria a ser o primeiro protótipo do React, uma biblioteca JavaScript para criação de interfaces de usuário (UI), que só foi lançado publicamente em 2013. Já em 2015, a biblioteca atingiu um ponto considerado como estável e grandes empresas como Netflix e Airbnb começaram a apoiar e usar o React, além do Facebook e Instagram. Nos períodos seguintes, algumas *features* notáveis fizeram o React continuar no *mainstream* das bibliotecas JavaScript [3][4], se tornando uma das principais bibliotecas utilizadas por desenvolvedores profissionais [5].

Uma das arquiteturas usadas atualmente é a de Aplicação de Página Única (*Single-Page Applicatio*n - SPA). A SPA reescreve dinamicamente a página atual à medida que o usuário interage, em vez de carregar novas páginas inteiras do servidor. Isso leva a constantes manipulações no Modelo de Objeto de Documento (Document Object Model - DOM), que são lentas. Nesse caso, o ReactJS se apresenta como uma boa solução, visto que a biblioteca apresenta vários métodos de otimização, como o DOM virtual (*Virtual DOM* - VDOM), que decide se o componente deve ser recarregado ou não com base no estado atual do componente e nas alterações que ocorreram. Contudo, os métodos que o ReactJS implementa de forma nativa nem sempre são suficientes [6].

Desta forma, se faz necessário a implementação de otimizações adicionais por parte dos desenvolvedores, que, também devem ter cautela para não utilizar más práticas que possam piorar o desempenho da aplicação. Existem várias opções de técnicas que podem ser aplicadas, cada uma com suas próprias vantagens e desvantagens, mas não há um catálogo dessas técnicas que podem melhorar o desempenho do front end e que validem a diferença da utilização delas na prática em um sistema real, estando todas espalhadas em sites oficiais, artigos, vídeos no YouTube, em fóruns e em blogs.

Sendo assim, neste trabalho foi realizado pesquisas em artigos e na comunidade de desenvolvedores por técnicas adotadas para melhorar o desempenho do front end, assim como a implementação dessas técnicas em um sistema já desenvolvido com o intuito de coletar e analisar os dados de desempenho do sistema antes e após a aplicação destas técnicas para, por fim, iniciar uma catalogação das boas e más práticas para o desempenho do front end.

2. REVISÃO DA LITERATURA

No trabalho de Javeed [7], são descritas algumas maneiras de melhorar o desempenho de aplicações ReactJS. Essas maneiras incluem a redução do número de estados (*States*) e de variáveis *Prop*, assim como a separação de um componente principal em

componentes independentes, para que a quantidade de renderizações possam ser reduzidas e renderizações desnecessárias evitadas. Entretanto, não há uma avaliação das técnicas apresentadas nesse trabalho.

Já Kainu [6], além dos métodos já citados para redução de renderizações desnecessárias, menciona o uso de chaves estáveis no mapeamento de uma lista de conteúdos. Ademais, o artigo exibe técnicas nativas do React para a implementação de cache no nível da aplicação, com o uso dos Hooks *memo*, *useCallback* e *useMemo*. Também são citadas estratégias como o *code-splitting* e *tree shaking* para diminuir o tempo de carregamento. Porém, apesar de serem apresentadas as ferramentas Lighthouse, React Developer Tools Profiler e React.js Profiler Component para auxiliar na analise do desempenho, não foi feita uma análise sobre as técnicas apresentadas.

Pavić e Brkić [1], similarmente, descreveram e discutiram algumas abordagens para melhorar o desempenho e o carregamento inicial de uma aplicação web em ReactJS. São elas: *Code bundling*, *code-splitting*, *tree shaking*, otimização de imagens, métodos de renderização e a utilização de *frameworks*, como Gatsby e Next.js. Para realização dos testes utilizaram as ferramentas Lighthouse, Chrome DevTools, webpack-bundle-analyzer. No entanto, não foi realizada uma análise individual das técnicas apresentadas, sendo feita apenas uma comparação entre os sistemas nas versões sem e com as otimizações e utilizando os frameworks Gatsby e Next.js.

Neste trabalho, serão utilizadas algumas das técnicas que foram apresentadas nesta seção e outras técnicas que não foram citadas nos trabalhos mencionados, mas que podem melhorar o desempenho do front end.

3. METODOLOGIA

Nesta seção, será apresentada a metodologia utilizada para o desenvolvimento dos experimentos realizados neste trabalho. Inicialmente, houve o processo de pesquisa em comunidades e literatura por sugestões de técnicas de melhoria de desempenho do front end, progredindo para a etapa da aplicação de algumas técnicas encontradas e da etapa da obtenção dos dados seguindo algumas métricas e, por fim, a análise dos dados coletados.

A pesquisa por técnicas de melhoria de desempenho ocorreram de duas formas. A primeira foi através da literatura acadêmica, onde foram utilizados páginas como o Google Scholar[1] e o IEEE Xplore[2] para encontrar artigos relacionados ao tema. A pesquisa feita nessas páginas foi realizada com termos-chave, como: ReactJS, desempenho, performance, JavaScript, *re-render, optimization* e *web development*. Além de filtrar os artigos publicados nos últimos cinco anos.

A segunda forma foi através de comunidades no Reddit[3], onde foi solicitado sugestões de técnicas para que sistemas front-end utilizando React tivessem um bom desempenho. As perguntas foram realizadas tanto no idioma inglês como em português, sendo elas as seguintes: "*What are the best practices for frontend performance using React?*" e "Quais as melhores práticas para desempenho do front end usando React?".

Após coletadas, foi realizada a etapa de aplicação das técnicas. Na qual, para efetuar os testes das práticas de desenvolvimento em React, foi utilizada uma tela do Sistema de Gestão de Licitações e Contratos do Tribunal de Contas do Estado do Acre (TCE-AC) que contém uma grande quantidade de dados e que realiza uma grande quantidade de requisições.

Para isso, foi feito o clone da versão mais recente até o momento do código-fonte localizado na *branch* principal. Em seguida, para cada técnica ou conjunto de técnicas aplicadas foi criada uma nova *branch* para que fosse possível, posteriormente, coletar os dados do desempenho da tela antes e depois da aplicação de cada prática.

A coleta dos dados foi realizada em uma máquina executando apenas o projeto do TCE-AC com o banco de dados de forma local, por meio da ferramenta Lighthouse, uma ferramenta de análise de desempenho de páginas web que pode ser usada para medir e avaliar várias métricas de desempenho, sendo elas:

- Tempo Total de Bloqueio (Total Blocking Time - TBT), que mede o tempo total em milissegundos durante o qual o JavaScript de uma página bloqueia o processamento principal da página [8];
- Índice de velocidade (*Speed Index* - SI), que mede o tempo necessário para que o conteúdo da página seja exibido completamente na tela do usuário [9];
- Maior Renderização de Conteúdo (Largest Contentful Paint - LCP), que mede o tempo levado para que o elemento de conteúdo mais importante da página seja exibido na tela do usuário [10];
- Primeira renderização de conteúdo (First Contentful Paint - FCP), que mede o tempo necessário para que o primeiro elemento visível de conteúdo seja exibido na tela do usuário [11];
- Tempo até interatividade (Time to Interactive - TTI), que calcula o tempo que leva para que a página esteja pronta para interação do usuário [12];
- Mudança cumulativa de layout (Cumulative Layout Shift - CLS), que mede a quantidade de mudanças no layout da página durante o carregamento [13].

Essa ferramenta fornece relatórios detalhados sobre o desempenho da página, incluindo pontuações e recomendações para melhorar o desempenho, sendo executado dez vezes para cada *branch* criada, tendo em vista a realização de uma análise melhor de cada técnica.

Durante a última etapa, a de análise dos dados coletados, foram calculadas as médias e os desvios padrão tanto da pontuação de desempenho geral quanto dos resultados das métricas de forma individual.

4. RESULTADOS

Os resultados gerais obtidos para as práticas testadas podem ser vistos na tabela 1. Assim como a Tabela 2 exibe a média e o desvio padrão de cada métrica para o código-fonte sem modificações. As práticas "Paginar listas longas", "Utilizar Load-spinners / Skeleton" e "Evitar o uso de funções *inline*" já

[1] https://scholar.google.com/

[2] https://ieeexplore.ieee.org/Xplore/home.jsp

[3] https://www.reddit.com/

```
import { lazy } from 'react';
import { Switch } from 'react-router';
import UrlRouter from '~/constants/UrlRouter';
const BoardLicitacoesIndexPage = lazy(() => import('~/pages/licitacao/boardLicitacoes'));
const NewLicitacao = lazy(() => import('~/pages/licitacao/new'));
const NotFound = lazy(() => import('fc/pages/NotFound'));
const EditLicitacao = lazy(() => import('~/pages/licitacao/edit'));
const ProximaFaseLicitacao = lazy(() => import('~/pages/licitacao/proximaFase'));
import TCERoute from '~/components/router/TCERoute';

const RoutesLicitacao = () => {
  return (
    <Switch>
      <TCERoute path={UrlRouter.cadastrosConsulta.licitacao.index} exact component={BoardLicitacoesIndexPage} />
      <TCERoute path={UrlRouter.cadastrosConsulta.licitacao.novo} exact component={NewLicitacao} />
      <TCERoute path={UrlRouter.cadastrosConsulta.licitacao.editar} exact component={EditLicitacao} />
      <TCERoute path={UrlRouter.cadastrosConsulta.licitacao.proximaFase} exact component={ProximaFaseLicitacao} />
      <TCERoute path={UrlRouter.cadastrosConsulta.licitacao.openDetails} exact component={BoardLicitacoesIndexPage} />
      <TCERoute component={NotFound} />
    </Switch>
  );
};

export default RoutesLicitacao;
```

Figura 1: Arquivo das rotas contendo o arquivo da tela de Licitação

eram utilizadas no projeto, logo para estes casos o código foi modificado, retirando essas práticas, para que fosse possível obter os dados necessários para as análises.

Ao longo desta seção serão realizadas as análises referentes a cada técnica.

Técnica	Média	Desvio Padrão
Código-fonte sem modificações	26,4	0,5164
Code split e Lazy load	44,5	0,7071
Function Component	36,3	0,6749
Keys	37	1,5635
Paginação de listas longas	26,1	0,3162
Load-spinners ou Skeleton	25,4	0,5164
Funções *inline*	27,1	0,3162
Minificador	57,5	2,4152

Tabela 1: Média e desvio padrão para a pontuação geral de cada técnica aplicada

4.1 Function Components

A forma mais simples de definir um componente em React é através de uma função JavaScript [14]. Essas funções podem receber as propriedades (*props*) e retornar elementos React normalmente, assim como podem ter estado interno e gerenciar ciclos de vida [15].

Para substituir o componente de classe utilizado na tela de Licitação as seguintes alterações foram necessárias: (i) usar função ao invés de classe, (ii) remover o construtor (*constructor*), (iii) remover o método *render()*, mantendo o *return* que se localiza dentro desse método, (iv) transformar todos os métodos em constantes, (v) remover todas as ocorrências de "*this.state*" e de "*this*" do componente, (vi) configurar o estado inicial usando o *useState()*, (vii) modificar todas as ocorrências de *this.setState()* pelo método que foi criado no passo anterior para atualizar o estado e, (viii) trocar o *componentDidMount* por *useEffect* [16].

	Média	Desvio Padrão
FCP (s)	0,6	0,0488
TTI (s)	8,6	0,3498
SI (s)	5,45	0,5103
TBT (ms)	4.070	382,2099
LCP (s)	6,8	0,2601
CLS	0,001	0,0003

Tabela 2: Média e desvio padrão para as métricas obtidas com o Lighthouse para o projeto TCE-AC sem alterações em seu código-fonte

Ao concluir essas modificações e executar a ferramenta Lighthouse, obtemos os resultados exibidos na Tabela 3, assim como é possível observar a pontuação geral obtida na Tabela 1. Com uma média de 36,3, foi perceptível um aumento de 37,5% em relação à média obtida na *branch* "*master*". Da mesma forma, é possível examinar que métricas como TBT obteve uma redução de 67,9%, saindo de 4.070 ms para 1.305 ms.

Já a métrica SI obteve uma redução de 62,4%, partindo de 5,45 s para 2,05. Sendo essas duas métricas as que obtiveram reduções mais significativas para esta técnica.

	Média	Desvio Padrão
FCP (s)	0,4	0
TTI (s)	5,65	0,3471
SI (s)	2,05	0,1476
TBT (ms)	1.305	51,6935
LCP (s)	5,1	0,0632
CLS	0	0

Tabela 3: Média e desvio padrão para as métricas obtidas com o Lighthouse para o projeto TCE-AC utilizando *Function Components*

4.2 *Code-Splitting* e *Lazy Load*

Code-splitting é uma técnica para dividir o código da aplicação em múltiplos *bundles*, de forma que somente os módulos necessários são carregados dinamicamente e executados quando são requeridos [6][17].

Já o *Lazy Loading* é um técnica onde componentes ou módulos só são carregados quando são realmente necessários [17]. De forma que pode ser utilizado em conjunto com o *code-splitting* para melhorar o carregamento da aplicação.

A melhor maneira de introduzir *code-splitting* na aplicação é através de importações dinâmicas. Em React, existe a função "*React.lazy*" que permite a renderização de uma importação dinâmica como um componente regular [17]. Essa função é usada da seguinte forma:

const <nomeDaConst> = React.lazy((() => import('<caminhoComponente>'));

Como é possível observar no início da Figura 1.

```
import { Suspense } from 'react';
import { Route } from 'react-router-dom';

const TCERoute = (props) => {
  return (
    <Suspense
      fallback={
        <div className="card page">
          <i className="pi pi-spin pi-spinner" style={{ fontSize: '2em' }}></i>
        </div>
      }
    >
      <Route {...props} />
    </Suspense>
  );
};

export default TCERoute;
```

Figura 2: Componente responsável por encapsular os componentes que são importados com o *React.lazy*

O componente que for carregado de forma preguiçosa (*lazy*) deve ser renderizado dentro de um componente *Suspense* que permite exibir algum conteúdo alternativo até que o componente preguiçoso seja carregado [18]. Para este fim, foi criado o arquivo TCERoute apresentado na Figura 2, que irá exibir um indicador de carregamento como conteúdo alternativo. Na figura 1, podemos ver como o componente TCERoute é utilizado, precisando apenas das propriedades *path*, *exact* e *component*.

	Média	Desvio Padrão
FCP (s)	0,4	0,0422
TTI (s)	4,8	0,1838
SI (s)	2,9	0,2201
TBT (ms)	1.700	114,5814
LCP (s)	2,4	0,0675
CLS	0,003	0,0005

Tabela 4: Média e desvio padrão para as métricas obtidas com o Lighthouse para o projeto TCE-AC usando *code-splitting* e *lazy load*

Ao analisar os dados podemos observar que houve um aumento de 68,6% na média geral em comparação com a média obtida na *branch* "*master*" (Tabela 1). Para a métrica LCP, antes tínhamos 6,8 segundos e após a aplicação da técnica obtivemos 2,4s (Tabela 4), representando uma diminuição de 64,7%.

A segunda maior redução foi para a métrica TBT que tinha um valor médio de 4.070ms e ficou com 1.700ms (Tabela 4), representando uma redução de aproximadamente 58%.

4.3 Load Spinners

Load spinners são elementos visuais que indicam ao usuário que uma ação está sendo realizada e que o usuário deve aguardar. Uma outra técnica similar se chama esqueleto (*skeleton*), que é usada para simular o carregamento de conteúdo, criando um estrutura similar ao do conteúdo que está sendo carregado, antes que o conteúdo real esteja completo. Isto também fornece uma indicação visual de que uma ação está sendo executada.

	Média	Desvio Padrão
FCP (s)	0,7	0,0667
TTI (s)	10	0,3529
SI (s)	6,6	0,5657
TBT (ms)	5.080	179,1337
LCP (s)	7,5	0,1595
CLS	0,002	0,0003

Tabela 5: Média e desvio padrão para as métricas obtidas com o Lighthouse para o projeto TCE-AC sem utilizar *load spinners*

O sistema do TCE-AC já utilizava esta técnica, então, para fins de comparação, foram retirados da tela os componentes que indicavam algum carregamento.

Como resultado, obtivemos uma média geral de 25,4 pontos (Tabela 1), que comparados à média geral do sistema antes das alterações (26,4 pontos) representa uma diminuição de 3,8%. Além disso, ao comparar individualmente os dados da Tabela 5 com os dados da Tabela 2, observa-se que todas as métricas obtidas aumentaram.

4.4 Paginação de listas longas

Paginação é uma técnica para dividir uma grande quantidade de dados em várias páginas menores, onde apenas uma página é exibida de cada vez, permitindo que os usuários naveguem com maior facilidade por uma grande quantidade de informação.

Assim como a técnica de *load spinners*, o TCE-AC já implementa esta prática, então ela foi retirada, tal como foram removidos os filtros aplicados pela tela e alterado o tamanho definido por padrão da lista a ser retornada pela requisição, no intuito de obter o material necessário para as comparações.

Assim como na técnica analisada anteriormente, foi verificado uma diminuição na pontuação do *Lighthouse*, partindo de 26,4 pontos para 26,1 (Tabela 1), uma leve redução de aproximadamente 1%.

	Média	Desvio Padrão
FCP (s)	0,6	0,0966
TTI (s)	10,3	0,2756
SI (s)	5,25	0,4001
TBT (ms)	4.680	613,6313
LCP (s)	6,8	0,1155
CLS	0,001	0,0009

Tabela 6: Média e desvio padrão para as métricas obtidas com o Lighthouse para o projeto TCE-AC sem utilizar paginação de listas longas

Examinando os resultados da Tabela 6, podemos concluir que o SI é a única métrica que simboliza uma sutil melhora de 3,7% se comparado ao sistema usando paginação. Essa pequena diferença entre os resultados pode ter ocorrido devido a execução do projeto de forma local, reduzindo complexidades como as apresentadas durante o tráfego de rede, fazendo com que as requisições realizadas pelo sistema tenham respostas mais rápidas, influenciando, assim, os dados coletados.

4.5 Funções *Inline*

Funções *inline* são funções declaradas e usadas diretamente dentro do código JSX de um componente.

```
_toggleDetails(item, callback) {
  this.setState((oldState) => {
    const newShowValue = !oldState.showDetails;
    const { idLicitacao } = this.props.match.params;
    if (idLicitacao && !newShowValue) {
      this.refreshPage();
    } else {
      return { showDetails: newShowValue, selectedItem: item };
    }
  }, callback);
}

switchProrrogarDialog() {
  this.setState({
    showProrrogarDialog: !this.state.showProrrogarDialog,
  });
}
```

Figura 3: Definição das funções _toggleDetails e switchProrrogarDialog

Em uma publicação na comunidade "r/programacao"[4] realizada durante a etapa de pesquisa, algumas recomendações de práticas foram realizadas, como por exemplo as práticas, "suba o estado compartilhado para o ancestral mais comum", "usememo para cálculos pesados", "service worker/web workers para trabalhar com dados em background", onde dentre elas foi recomendado a não utilização de funções *inline*. A tela de licitações do projeto do TCE-AC não cria as funções diretamente dentro do código de um componente, ou seja, não utiliza funções *inline*. Dessa forma, para realizar os testes de desempenho, adicionamos as funções já definidas no código anteriormente diretamente nos componentes.

```
<OcorrenciaProrrogarFormPage
  showProrrogarDialog={this.state.showProrrogarDialog}
  switchProrrogarDialog={this.switchProrrogarDialog}
  idLicitacao={this.state.selectedItem?.id}
  toggleDetails={this._toggleDetails}
  reload={this.store.load}
/>
```

Figura 4: Utilização do componente OccorrenciaProrrogarFormPage, utilizando as funções definidas anteriormente na Figura 3

	Média	Desvio Padrão
FCP (s)	0,4	0,0516
TTI (s)	8,25	0,3018
SI (s)	5,4	0,3204
TBT (ms)	4.400	274,6088
LCP (s)	6,3	0,1766
CLS	0,001	0,0004

Tabela 7: Média e desvio padrão para as métricas obtidas com o Lighthouse para o projeto TCE-AC utilizando funções *inline*

[4] https://www.reddit.com/r/programacao/

Nas figuras 3 e 4, está representada a utilização de um componente com as funções definidas anteriormente. Já na Figura 5, está representada a utilização com componente com as funções definidas de forma *inline*.
A utilização das funções de forma *inline* resultou em um aumento de 2,7% na pontuação geral do *Lighthouse* (Tabela 1), tendo reduções de aproximadamente 33% na métrica FCP e de 7% na métrica TTI (Tabela 7). Por outro lado, a legibilidade do código foi comprometida, podendo aumentar o tamanho do código e dificultar a sua manutenção, principalmente se essas mesmas funções forem usadas em outras partes do código, assim como dificultar a depuração de erros.

```
<OcorrenciaProrrogarFormPage
  showProrrogarDialog={this.state.showProrrogarDialog}
  switchProrrogarDialog={() =>
    this.setState({
      showProrrogarDialog: !this.state.showProrrogarDialog,
    })
  }
  idLicitacao={this.state.selectedItem?.id}
  toggleDetails={(item, callback) => {
    this.setState((oldState) => {
      const newShowValue = !oldState.showDetails;
      const { idLicitacao } = this.props.match.params;
      if (idLicitacao && !newShowValue) {
        this.refreshPage();
      } else {
        return { showDetails: newShowValue, selectedItem: item };
      }
    }, callback);
  }}
  reload={this.store.load}
/>
```

Figura 5: Utilização do componente OccorrenciaProrrogarFormPage, utilizando as funções definidas de forma *inline*

4.6 Keys

Ao renderizar componentes em um array o atributo "*key*" pode ser atribuído. Esse atributo ajuda o React a identificar quais itens no *array* são novos, removidos ou alterados, permitindo que a biblioteca possa atualizar a interface do usuário de forma eficiente quando os dados mudarem [19]. O atributo *key* deve conter um valor único para cada item no *array*. Caso seja usada alguma chave instável enquanto é mapeada uma lista de conteúdo em JSX, ocasionará a recriação desnecessária dos componentes em cada nova renderização [6]. Desse modo, não é recomendado o uso do índice do elemento no array como *key* caso a ordem dos elementos possam mudar, pois pode afetar negativamente o desempenho e causar problemas com o estado do componente [19].

```
{DadosEstaticosService.getFasesLicitacao().map((ray) => (
  <React.Fragment key={`${ray.phaseKey}-item`}>
    <LazyScroller
      phaseKey={ray}
      refresh={this.state.reloadLazys}
      service={this.store.service}
      advancedSearchParams={this.store.advancedSearchParams}
      sortOptions={this._getLazyScrollerSortFields()}
      store={this.store}
      _toggleDetails={this._toggleDetails}
    />
  </React.Fragment>
))}
```

Figura 6: Utilização do atributo *key*

Na figura 6, é possível verificar como foi utilizado o atributo *key* no componente React.Fragment que está sendo criado durante o mapeamento do item do *array*.

A utilização do atributo *key* resultou em um aumento de cerca de 40% na pontuação geral. Já nas métricas de forma individual houve uma diminuição do tempo de 70% na métrica TBT, que é medida em milissegundos, e uma melhora de aproximadamente 65% no índice de velocidade (SI), como pode ser observado na Tabela 8.

	Média	**Desvio Padrão**
FCP (s)	0,4	0,0422
TTI (s)	5,6	0,0876
SI (s)	1,9	0,3974
TBT (ms)	1.220	23,2140
LCP (s)	5,05	0,0699
CLS	0,001	0

Tabela 8: Média e desvio padrão para as métricas obtidas com o Lighthouse para o projeto TCE-AC utilizando *keys* para os componentes renderizados a partir de uma lista

4.7 Minificadores

Minificação é o processo de reduzir o tamanho do código, removendo caracteres desnecessários, como espaços em branco, quebra de linha e comentários, também encurtando nomes de variáveis e funções.

```
new OptimizeCSSAssetsPlugin({
  cssProcessorOptions: {
    parser: safePostCssParser,
    map: shouldUseSourceMap
      ? {
          inline: false,
          annotation: true,
        }
      : false,
  },
  cssProcessorPluginOptions: {
    preset: ['default', { minifyFontValues: { removeQuotes: false } }],
  },
}),
```

Figura 7: Utilização do OptimizeCSSAssetsPlugin

Para realizar a minificação do JavaScript e do CSS do projeto, foram utilizados dois pacotes nas dependências de desenvolvimento, *terser-webpack-plugin*[5] e optimize-css-assets-webpack-plugin[6], respectivamente. Essas duas dependências foram configuradas para serem utilizadas apenas no modo de produção no arquivo webpack.config.js, como pode ser visto nas figuras 7 e 8. Desta forma, foi possível executar o comando *yarn build* para gerar a versão de produção do projeto e,

[5] https://www.npmjs.com/package/terser-webpack-plugin
[6] https://www.npmjs.com/package/optimize-css-assets-webpack-plugin

em seguida, executar o comando serve -s build para acessar a versão de produção localmente no navegador.

Ao utilizar as técnicas de minimização do código houve um aumento significativo na pontuação geral, que passou de 26,4 para 57,5 (Tabela 1), ou seja aproximadamente 118% de melhora na pontuação.

```
new TerserPlugin({
  terserOptions: {
    parse: {
      ecma: 8,
    },
    compress: {
      ecma: 5,
      warnings: false,
      comparisons: false,
      inline: 2,
    },
    mangle: {
      safari10: true,
    },
    keep_classnames: isEnvProductionProfile,
    keep_fnames: isEnvProductionProfile,
    output: {
      ecma: 5,
      comments: false,
      ascii_only: true,
    },
  },
  sourceMap: shouldUseSourceMap,
}),
```

Figura 8: Utilização do TerserPlugin

	Média	Desvio Padrão
FCP (s)	0,5	0
TTI (s)	3,2	0,1506
SI (s)	1,4	0,3062
TBT (ms)	530	46,3801
LCP (s)	2,8	0,0483
CLS	0	0

Tabela 9: Média e desvio padrão para as métricas obtidas com o Lighthouse para o projeto TCE-AC usando minificadores

Ao analisar as métricas individualmente também foi possível perceber a melhora com a redução de 87% no tempo de TBT, 74% no tempo de SI e de 62% no de TTI (Tabela 9). Além da redução de 100% na métrica CLS.

4.8 Considerações finais

O gráfico apresentado, na figura 9, oferece uma visão do tempo, em segundos, que as práticas obtiveram nas métricas FCP, TTI, SI e LCP.

Ao analisar o gráfico, percebe-se que a técnica que obteve melhor resultado nos experimentos considerando as métricas TTI e SI foi a de minificação, que reduz o tamanho do código, removendo caracteres desnecessários, comentários e encurtando nomes de variáveis e de funções, obtendo 3,2 e 1,4 segundos, respectivamente. Valores muito interessantes, visto que valores abaixo de 3,8 segundos são considerados rápidos para a métrica TTI [12] e valores abaixo de 3,4 segundos são considerados rápidos para a métrica SI [9].

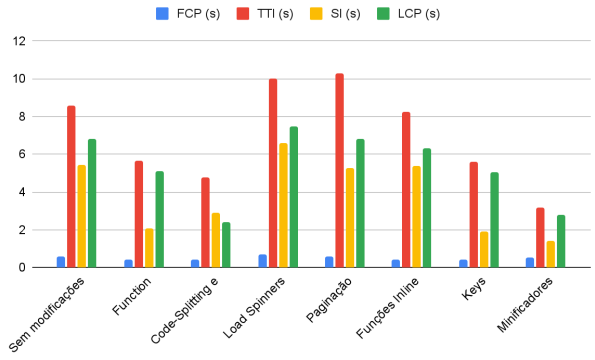

Figura 9: Média das métricas coletadas para cada técnica

Para a métrica de LCP, a técnica que mais se destacou com apenas 2,4 segundos foi o conjunto das técnicas *code-splitting* e *lazy load*, que divide o código em *bundles* permitindo o carregamento dos mesmos apenas quando necessários, ficando abaixo dos 2,5 segundos recomendados [10].

Para a métrica FCP várias técnicas se destacaram, todas com 0,4 segundos. Foram elas: *code-splitting* e *lazy load*. Assim como, *function components*, uma forma simples de definir um componente, funções *inline*, que consiste na declaração das funções diretamente no componente, e a utilização das *keys* para identificar os componentes criados durante a renderização de um *array*. Em adição, todas as técnicas se saíram bem nessa métrica pontuando abaixo do 1,8 segundos recomendado [11].

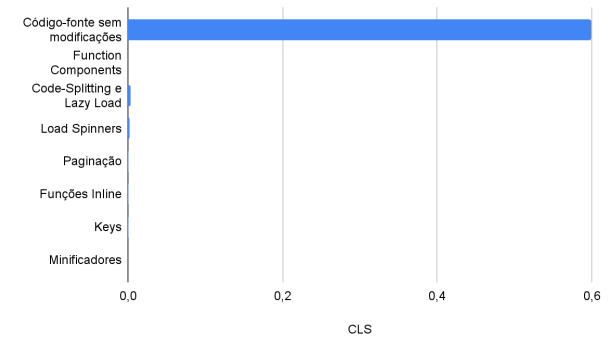

Figura 10: Média da métricas CLS

Já as práticas que mais se destacaram na métrica CLS foram *Function Components* e Minificadores, como pode ser visto na figura 10, que não registraram nenhuma mudança inesperada no *layout* da página durante o carregamento. Porém, de forma geral,

todas as técnicas analisadas obtiveram bons resultados, visto que a pontuação recomendada para uma boa experiência do usuário é de 0,1 ou menos [13].

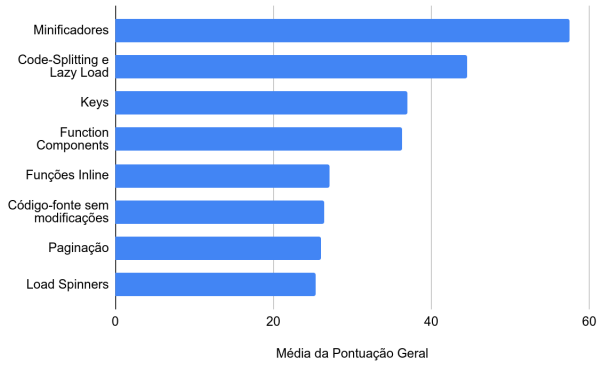

Figura 11: Média da pontuação geral das métricas coletadas

Por fim, as técnicas que mais se destacaram de forma geral foram as de minificação de código, *code-splitting* e *lazy load*, a utilização de *keys* e *function components* como é possível observar na figura 11. Da mesma forma, é possível observar que os testes realizados sem a utilização das técnicas de paginação de listas longas e sem a utilização de *load spinners* tiveram as menores pontuações.

5. CONCLUSÃO

Este trabalho tem como principal contribuição o início de uma catalogação de forma centralizada das boas e más técnicas para o desempenho, validando-as na prática, com o intuito de melhorar o desempenho dos sistemas front-end.

Embora existam muitos princípios e técnicas de programação que podem ser aplicados para obter melhorias de desempenho, neste trabalho focamos em apenas 7 práticas, sendo elas, a utilização de *function components*, *code-splitting* e *lazy load*, *load spinners*, paginação de listas longas, utilização de funções *inline*, de *keys*, e dos minificadores. Para realização das análises dessas técnicas, foi utilizada a tela um sistema já existente do Tribunal de Contas do Estado do Acre, assim como a ferramenta Lighthouse para a coleta dos dados.

Através disso, foi possível observar ao longo do trabalho que técnicas como a minificação do código, *code-splitting* e a utilização de *keys* na renderização de componentes de um *array*, podem melhorar consideravelmente o desempenho do sistema mesmo sendo modificações simples de serem feitas no quesito de código.

Foi visto também que embora algumas técnicas possam favorecer o desempenho do sistema há os *trade-offs*, como por exemplo a utilização das funções *inline*, que embora tenha melhorado um pouco o desempenho, prejudica a legibilidade, podendo dificultar também a manutenção e depuração do código.

Por fim, as análises levaram em consideração apenas as práticas de forma individual, sem a combinação de duas ou mais técnicas ao mesmo tempo, o que poderia indicar quais as melhores combinações para o desempenho da página web.

6. AGRADECIMENTOS

Agradeço primeiramente a Deus, por ter permitido que eu tivesse saúde e perseverança durante os meus anos de estudo. Agradeço a todo o corpo docente e aos demais funcionários do curso de Ciência da Computação da Universidade Federal de Campina Grande (UFCG) por todos os ensinamentos proporcionados ao longo da graduação. Agradeço principalmente ao meu orientador Prof. Dr. Cláudio de Souza Baptista e ao co-orientador Prof. Dr. Hugo Feitosa de Figueirêdo pela dedicação, comprometimento e por todos os ensinamentos, oportunidades e experiências compartilhadas, que permitiram que eu melhorasse como estudante, profissional e pessoa. Agradeço aos amigos formados durante a graduação, em especial aos do Laboratório de Sistemas de Informação por todos os conhecimentos partilhados. Agradeço a minha família, aos meus pais e as minhas irmãs, por todo apoio e confiança depositada em mim. Por fim, um agradecimento especial a minha irmã Mirelle e meu amigo João Vitor por toda ajuda ao longo deste trabalho.

7. REFERÊNCIAS

[1] F. Pavić and L. Brkić, "Methods of Improving and Optimizing React Web-applications," 2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO), Opatija, Croatia, 2021, pp. 1753-1758, doi: 10.23919/MIPRO52101.2021.9596762.

[2] Think with Google - Find out how you stack up to new industry benchmarks for mobile page speed. https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-page-speed-new-industry-benchmarks/

[3] Medium - ReactJS: A brief history. https://medium.com/@sjarancio/reactjs-a-brief-history-3c1e969a477f

[4] RisingStack - The History of React.js on a Timeline. https://blog.risingstack.com/the-history-of-react-js-on-a-timeline/

[5] Stack Overflow Developer Survey 2022 - Most popular technologies - Web frameworks and technologies. https://survey.stackoverflow.co/2022/#technology-most-popular-technologies

[6] Kainu, “Optimization in React.js: Methods, Tools, and Techniques to Improve Performance of Modern Web Applications”, Bachelor's Programme in Computer Sciences Informaatioteknologian ja viestinnän tiedekunta - Faculty of Information Technology and Communication Sciences, 2022. https://urn.fi/URN:NBN:fi:tuni-202205164974

[7] Javeed, "Performance Optimization Techniques for ReactJS," 2019 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT), Coimbatore, India, 2019, pp. 1-5, doi: 10.1109/ICECCT.2019.8869134.

[8] Chrome Developers - Lighthouse - Total Blocking Time. https://developer.chrome.com/docs/lighthouse/performance/lighthouse-total-blocking-time/.

[9] Chrome Developers - Lighthouse - Speed Index. https://developer.chrome.com/docs/lighthouse/performance/speed-index/.

[10] Chrome Developers - Lighthouse - Largest Contentful Paint. https://developer.chrome.com/docs/lighthouse/performance/lighthouse-largest-contentful-paint/.

[11] Chrome Developers - Lighthouse - First Contentful Paint. https://developer.chrome.com/docs/lighthouse/performance/first-contentful-paint/.

[12] Chrome Developers - Lighthouse - Time to Interactive. https://developer.chrome.com/docs/lighthouse/performance/interactive/.

[13] web.dev by Chrome DevRel - Cumulative Layout Shift (CLS). https://web.dev/cls/.

[14] React - Docs - Components and Props. https://reactjs.org/docs/components-and-props.html

[15] React - Docs - Introducing Hooks. https://reactjs.org/docs/hooks-intro.html

[16] DigitalOcean - How To Convert React Class Components to Functional Components with React Hooks. https://www.digitalocean.com/community/tutorials/five-ways-to-convert-react-class-components-to-functional-components-with-react-hooks

[17] React - Docs - Code-Splitting. https://reactjs.org/docs/code-splitting.html

[18] React Docs - <Suspense>. https://beta.reactjs.org/reference/react/Suspense#suspense

[19] React - Docs - Lists and Keys. https://reactjs.org/docs/lists-and-keys.html