Universidade Federal do Rio Grande do Norte
Centro de Tecnologia
Departamento de Engenharia de Computação e Automação
Curso de Engenharia de Computação

Avaliação de Custo de Construção de Software com Uso de Distintos BaaS

Victor Emanuel Ribeiro Silva

Natal-RN
Fevereiro de 2022

Universidade Federal do Rio Grande do Norte - UFRN
Sistema de Bibliotecas - SISBI
Catalogação de Publicação na Fonte. UFRN - Biblioteca Central Zila Mamede

Silva, Victor Emanuel Ribeiro.
Avaliação de custo de construção de software com uso de distintos BaaS / Victor Emanuel Ribeiro Silva. - 2022.
36 f.: il.

Monografia (graduação) - Universidade Federal do Rio Grande do Norte, Centro de Tecnologia, Curso de Engenharia de Computação, Natal, RN, 2022.
Orientador: Prof. Dr. Samuel Xavier de Souza.
Coorientador: M.Sc Júlio Gustavo Soares Firmo da Costa.

1. Engenharia da computação - Monografia. 2. Métrica de Software - Monografia. 3. YCodify - Monografia. 4. BaaS - Monografia. I. Souza, Samuel Xavier de. II. Costa, Júlio Gustavo Soares Firmo da. III. Título.

RN/UF/BCZM CDU 004

Elaborado por FERNANDA DE MEDEIROS FERREIRA AQUINO - CRB-15/301

Victor Emanuel Ribeiro Silva

Avaliação de Custo de Construção de Software com Uso de Distintos BaaS

Trabalho de Conclusão de Curso na modalidade Monografia, submetido como parte dos requisitos necessários para conclusão do curso de Engenharia de Computação pela Universidade Federal do Rio Grande do Norte.

Orientador(a)

Prof. Dr. Samuel Xavier de Souza

Universidade Federal do Rio Grande do Norte – UFRN
Departamento de Engenharia de Computação e Automação – DCA

Natal-RN

Fevereiro de 2022

Universidade Federal do Rio Grande do Norte - UFRN
Sistema de Bibliotecas - SISBI
Catalogação de Publicação na Fonte. UFRN - Biblioteca Central Zila Mamede

Silva, Victor Emanuel Ribeiro.
Avaliação de custo de construção de software com uso de distintos BaaS / Victor Emanuel Ribeiro Silva. - 2022.
36 f.: il.

Monografia (graduação) - Universidade Federal do Rio Grande do Norte, Centro de Tecnologia, Curso de Engenharia de Computação, Natal, RN, 2022.
Orientador: Prof. Dr. Samuel Xavier de Souza.
Coorientador: M.Sc Júlio Gustavo Soares Firmo da Costa.

1. Engenharia da computação - Monografia. 2. Métrica de Software - Monografia. 3. YCodify - Monografia. 4. BaaS - Monografia. I. Souza, Samuel Xavier de. II. Costa, Júlio Gustavo Soares Firmo da. III. Título.

RN/UF/BCZM CDU 004

Elaborado por FERNANDA DE MEDEIROS FERREIRA AQUINO - CRB-15/301

Trabalho de Conclusão de Curso na modalidade Monografia sob o título *Avaliação de Custo de Construção de Software com Uso de Distintos BaaS* apresentada por Victor Emanuel Ribeiro Silva e aceita pelo Departamento de Engenharia de Computação e Automação da Universidade Federal do Rio Grande do Norte, sendo aprovada por todos os membros da banca examinadora abaixo especificada:

Prof. Dr. Samuel Xavier de Souza
Orientador(a)
Departamento de Engenharia de Computação e Automação
Universidade Federal do Rio Grande do Norte

M.Sc Júlio Gustavo Soares Firmo da Costa
Co-orientador(a)
Programa de Pós-graduação em Engenharia Elétrica e Computação
Universidade Federal do Rio Grande do Norte

Prof. Dr. Eduardo de Lucena Falcão
Departamento de Engenharia de Computação e Automação
Universidade Federal do Rio Grande do Norte

Prof. M.Sc Anderson Bráulio Nóbrega da Silva
Instituto Federal de Educação, Ciência e Tecnologia da Paraíba

Natal-RN, 11 de Fevereiro de 2022

Dedico este trabalho à Maísa Maria, que todos os dias me presenteia com seu amor.

If you think good architecture is expensive, try bad architecture.

Brian Foote and Joseph Yoder

Avaliação de Custo de Construção de Software com Uso de Distintos BaaS

Autor: Victor Emanuel Ribeiro Silva
Orientador(a): Prof. Dr. Samuel Xavier de Souza

Resumo

O presente trabalho tem como objetivo fazer a comparação de esforços de construção de uma aplicação móvel mantendo constante seu *frontend* e alternando o seu *backend*. Foram desenvolvidas duas versões do mesmo aplicativo móvel, que compartilham das mesmas telas e funções. Elas distinguem-se apenas no serviço usado como *backend*, ou seja, na camada responsável pela comunicação externa, conhecida como *Repository*. Os dois serviços em questão são o Ycodify e o Hasura. A fim de obter uma comparação objetiva, foram pesquisadas em artigos métricas de complexidade e manutenibilidade de software. O critério para escolha das métricas foi fundamentada parte em sua alta aceitação e recorrência na literatura, e parte na disponibilidade de ferramentas capazes de coletá-las. Ao final, encontrou-se que o esforço de desenvolvimento do código do *frontend* assemelha-se de várias formas a partir do uso das métricas que aqui serão abordadas.

Palavras-chave: Métrica de Software, YCodify, BaaS.

Evaluation of Software Construction Cost when Using Different BaaS

Author: Victor Emanuel Ribeiro Silva
Advisor: Prof. Dr. Samuel Xavier de Souza

Abstract

The present work aims to compare the efforts of building a mobile application keeping its frontend constant and alternating its backend. Two versions of the same mobile application were developed, which share the same screens and functions. They differ only in the service used as the backend, that is, in the layer responsible for external communication, known as the Repository. The two services in question are Ycodify and Hasura. In order to obtain an objective comparison, metrics of software complexity and maintainability were searched in articles. The criterion for choosing the metrics was based partly on their high acceptance and recurrence in the literature, and partly on the availability of tools capable of collecting them. In the end, it was found that the frontend code development effort is similar in many ways from the use of the metrics that will be discussed here.

Keywords: Software Metrics, YCodify, BaaS.

Lista de figuras

Lista de abreviaturas e siglas

API – Application Programming Interface

REST – Representational State Transfer

BaaS – Backend as a Service

LOC – Lines of Code

CC – Cyclomatic complexity

MI – Manutenilibity Index

WMC – Weighted Methods per Class

NLOC – Number of Lines of Code

SGBDs – Sistemas de Gerenciamento de Bancos de Dados

MVPs – Minimum Viable Product

SQL – Standard Query Language

CLI – Command Line Interface

IDE – Integrated Development Environment

HTML – HyperText Markup Language

CSS – Cascading Style Sheets

CK – Chidamber e Kemerer

Sumário

1 Introdução

Sistemas de software costumam sofrer constantes mudanças. Essas mudanças tendem à incrementá-lo com novos requisitos, provocando um aumento em sua complexidade e promovendo a redução na qualidade do código. Nesse contexto, prover manutenção adequada se torna uma desafio cada vez maior. Métricas de software têm sido usadas por cerca de três décadas para avaliar ou prever propriedades de sistemas de software. Por todo esse tempo, novas métricas têm surgindo a cada ano, fazendo deste campo um rico espaço de estudo, reflexão e discussão. Mas seu sucesso tem sido limitado por fatores como a falta de modelos sólidos e a incapacidade de comparar dados obtidos por diferentes pesquisadores ou de diferentes sistemas.

A manutenibilidade de software pode ser descrita como a facilidade com que um sistema ou componente de software pode ser modificado para corrigir falhas, melhorar o desempenho, ou adaptar-se a um ambiente em alteração (COMMITTEE et al., 1990). Essa métrica é uma propriedade crucial nos projetos de software. Os trabalhos de (VASCONCELOS et al., 2017) e (ALIJA, 2017) afirmam que os custos aplicados ao desenvolvimento de software é majoritariamente alocado para a sua evolução, chegando à representar algo em torno de 90% dos recursos disponíveis. Esse fato revela que este é um dos fatores mais decisivos para sucesso ou fracasso de um produto e deve ser alvo de preocupação desde as fases inicias do projeto.

O *backend* é, na maioria dos casos, a parte mais complexa e, consequentemente, mais difícil de manter da aplicação web. Dentre as inúmeras soluções para proporcionar uma redução significativa nos custos financeiros e pessoal de seu desenvolvimento, foi criado o conceito de *Backend as a Service*. Essa abordagem, disponível online, objetiva a automatização de grande parte do trabalho despendido nesta camada da aplicação por meio da criação quase instantânea de API s REST ou GraphQL. Alguns exemplos populares são o Hasura, o Back4App e o Firebase.

Dada a disponibilidade de várias ferramentas, foi feito o experimento de realizar a

comparação objetiva entre BaaS a partir de métricas de software confiáveis. Isto é, o objetivo principal foi identificar qual delas representa o menor esforço de programação e manutenção. Como detalhes da implementação dos *backends* são desconhecidas, a avaliação foi feita exclusivamente da perspectiva de um programador *frontend*, que desenvolveu várias versões do mesmo *app*, uma para cada um dos serviços.

Portanto, ao final, teremos um quadro comparativo que auxiliará a compreender as distinções de complexidade, em termos de esforço necessário para codificar uma aplicação, entre distintos usos de serviços de *Backend as a Service*. Em linhas gerais, optamos por fazer uma avaliação do quanto de esforço deve ser despendido para desenvolver uma pequena aplicação tendo por base dois serviços distintos, o Hasura e o Ycodify. As seguintes métricas foram usadas neste trabalho: LOC (Número de linhas de Código); CC (ou Complexidade Ciclomática); e MI (Índice de Manutenibilidade).

2 Teoria

Escolhemos as métricas LOC, CC e MI por duas razões. A principal delas é a dificuldade de encontrar ferramentas de código aberto que ajudasse a avançar com o cálculo de métricas de forma automatizada. Existem ferramentas que apresentam o cálculo automatizado de várias métricas além das três apresentadas no parágrafo anterior, mas ou elas são pagas ou as *open-source* apresentaram *bugs* em sua instalação ou operação. A segunda razão é que essas três métricas estão entre a mais usadas nos artigos pesquisados por ambas as revisões usadas para embasar com esse trabalho. São eles (NUÑEZ-VARELA et al., 2017) e (ARDITO et al., 2020).

No trabalho de (NUÑEZ-VARELA et al., 2017), os autores pesquisaram 226 estudos publicados entre os anos de 2010 e 2015 e, ao final, encontram em torno de 300 métricas distintas. Ranqueando essas métricas encontradas pela quantidade de vezes que são citadas ao longo dos trabalhos publicados, as métricas LOC e CC aparecem entre as 10 mais citadas, sendo a LOC citada 79 vezes e CC 55 vezes. A métrica mais citada, com 89 ocorrências, foi a WMC (*Weighted Methods per Class*), métrica calculada a partir do somatório das complexidades das funções numa classe. No referido trabalho, ambas as métricas apresentam um crescimento quanto à quantidade de vezes que são citadas no período. Os autores também destacam que entre as métricas mais aplicadas a programas procedurais, tanto LOC quanto CC são as mais usadas, apesar de ambas serem também aplicadas a programas orientados a objetos. Outra informação sobre o uso delas é que são usadas para basear estudos de qualidade, complexidade e manutenibilidade de códigos. Adicionalmente, ele também apresenta como resultado da pesquisa uma lista de 41 ferramentas, tanto pagas quanto *open-source*, que são usadas para o cálculo automatizado de cadas uma das métricas investigadas.

O trabalho de (ARDITO et al., 2020) é mais recente e apresenta uma pesquisa que listou 174 métricas distintas, mas que foram filtradas para as 15 métricas mais comumente usadas ou referenciadas em outro artigos. Ele também relatou a existência de 38 ferramentas disponíveis para o cálculo automatizado dessas métricas no artigo, mas apenas 13 dessas

ferramentas são de código aberto. O estudo pesquisou um total de 47 trabalhos publicados entre os anos de 2000 e 2019. Nesse estudo, porém, a CC e LOC são as métricas mais referenciadas.

De ambos os trabalhos e das métricas que as ferramentas de cálculo automatizado disponibilizam para uso efetivo, e entre as métricas mais relevantes apontadas pelos artigos, restaram apenas três: Jscomplexity (uma variação do Escomplexity), o Understand e o Lizard, porém o último não fornece o Índice de Manutenibilidade. De todas as ferramentas pesquisadas, a maioria delas foram construídas para linguagens como C, C++ e Java. Poucas ferramentas foram construídas para Typescript ou Javascript.

Nas sessões seguintes, são descritas cada uma dessas métricas.

2.1 Linhas de Código - LOC

LOC ou NLOC consiste simplesmente em contar linhas de texto dentro de cada arquivo do projeto. Ela pode ser encontrada na literatura de várias formas diferentes devido aos vários critérios que podem ser adotados para definir quais linhas devem entrar na contagem. Algumas das outras definições parecidas são (NUÑEZ-VARELA et al., 2017):

- *Source Lines of Code* (SLOC): Considera apenas as linhas preenchidas com código;
- *Blank Lines of Code* (BLOC): Considera apenas as linhas em branco;
- *Comments Lines of Code* (CLOC): Considera apenas as linhas de comentários não vazio.

Estima-se que sua primeira menção date de fins da década de 60, segundo (FENTON; NEIL, 1999). Entretanto, ainda nos dias atuais, ela é uma das métricas mais usadas para ajudar a calcular o trabalho de codificação de software, além de apresentar-se na imensa maioria das ferramentas de do tipo.

A ideia por trás dessa métrica é associar a quantidade de código com o esforço em prover manutenção ao mesmo. Isto é, incentivar uma escrita mais compacta e direta das funcionalidades. Apesar de ser facilmente entendida e bastante usada, esta métrica acumula algumas críticas:

- Valores diferentes dessa métrica só são comparáveis caso elas se referiram à mesma linguagem.

- Às vezes, reduzir o código para que fique o mais compacto possível pode torná-lo de difícil compreensão.

2.2 Complexidade Ciclomática

Em 1976 (MCCABE, 1976), Thomas J. McCabe definiu a Complexidade Ciclomática, que calcula o quão complexo é determinado programa usando teoria de grafos. A ideia por trás é modelar trechos de código sequenciais como nós e construir um grafo representando todos os fluxos de execuções possíveis. Então, uma aresta de A para B, significa que possivelmente o trecho de código B pode ser executado após o A. Para as estruturas condicionais e de repetição mais conhecidas, há os seguintes grafos de fluxo de controle, apresentados a seguir na figura 1.

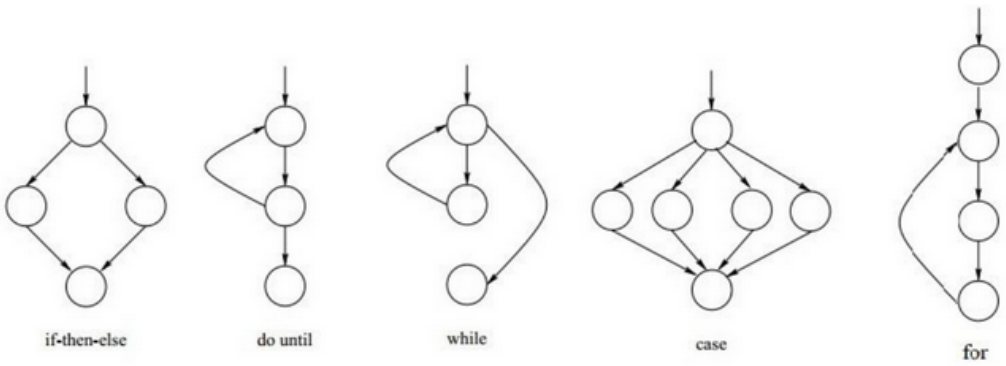


Figura 1: Esquema de grafos de estruturas de controle de fluxo.

Uma vantagem dessa abordagem é que pode-se criar grafos para as estruturas base de determinada linguagem e construir o fluxo de controle para qualquer trecho e código a partir das suas concatenações. Voltando para o cálculo da métrica, sabe-se que ela parte do princípio de que a complexidade depende do número de condições (caminhos) independentes em um software. De posse do grafo, a CC é definida matematicamente pela fórmula proposta por McCabe:

$M = E - N + 2P$

em que:

- E = quantidade de arestas
- N = quantidade de nós
- P = quantidade de componentes conectados

Caso o grafo seja fortemente conectado, isto é, seu nó de saída é conectado com o de entrada, formando um ciclo, a fórmula passa a ser:

$M = E - N + P$

Sobre o que são componentes conectados em grafos, um conjunto de nós é dito conectado quando há um caminho partindo de cada um dele para todos os outros do mesmo grafo. No caso de um grafo direcionado, deve-se respeitar a direção das arestas. Quando essa propriedade vale para todos os nós do grafo, ele é chamado de grafo conectado. Quando vale apenas para um subconjunto de nós, ele é chamado de componente conectado.

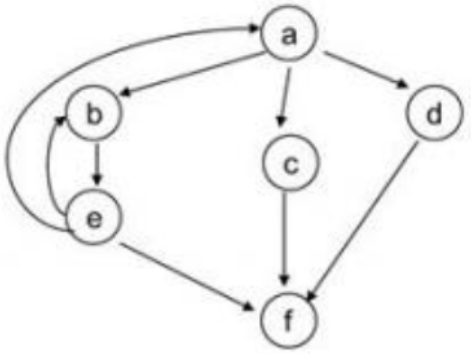


Figura 2: Exemplo de grafo não conectado que contém um componente conectado.

Na figura 2 o conjunto de nós A, B, E formam um componente pois podemos, a partir de qualquer um deles, caminhar até qualquer outro nó desse mesmo conjunto. Finalmente, é possível traçar algumas relações entre a Complexidade Ciclomática e a manutenibilidade, pois essa métrica também pode ser interpretada como o número mínimo de testes que devem ser escritos. Isso ocorre devido ao fato de, quanto mais bifurcações no grafo, mais testes são necessários para contemplar cada caso possível de execução. Por consequência, se mais testes são necessários, maior o esforço de desenvolvimento como um todo. Também foi descoberto que a Complexidade Ciclomática e as métricas de quantidade de Linhas de Código (LOC) compartilham uma correlação linear que transcende a linguagem em questão, como visto em (GRAYLIN et al., 2009).

2.3 Índice de Manutenibilidade - MI

Uma das técnicas mais conhecidas de mensuração de esforço de desenvolvimento é o Índice de Manutenibilidade, inicialmente proposta por (OMAN; HAGEMEISTER, 1992). Ela utiliza outras métricas previamente calculadas para gerar um indicador do esforço de se

prover manutenção a um software. A definição original do Índice de Manutenibilidade se dá pela fórmula a seguir:

$$MI = 171 - 5.2 * ln(aveV) - 0.23 * aveCC - 16.2 * ln(aveLOC)$$

Sendo:

- aveV: volume médio de Halstead
- aveCC: complexidade ciclomática média
- aveLOC : número médio de linhas de código

O volume de Halstead consiste na quantidade de operandos e operadores em cada arquivo. É, portanto, uma métrica de complexidade de código assim como o CC. A qualidade do código é proporcional ao valor obtido, logo, quanto maior melhor.

Importante mencionar que métricas LOC e CC compõem esse cálculo, o que demonstra, ou reforça sua a relevância de ambas como mecanismo para o cálculo da qualidade de código construído por desenvolvedores de software.

Outras sugestões de definições surgiram com o passar do tempo motivadas por algumas críticas que a fórmula original estava acumulando. Um dos principais problemas refere-se ao fato de que as métricas escolhidas para entrar na fórmula não consideravam o paradigma orientado a objetos, deixando de lado questões como acoplamento e coesão, segundo (MOLNAR; MOTOGNA, 2017) e (HEITLAGER; KUIPERS; VISSER, 2007). Por fim, essa métrica tem sido comumente usada para calcular a qualidade entre versões distintas de uma mesma aplicação.

3 Problema

Dada a situação na qual um desenvolvedor que se identifique como *Frontend Developer* deseja iniciar um projeto complexo que necessite de um *backend*, uma das primeiras questões a serem resolvidas é: quais tecnologias usar e qual provedor de *backend* servirá melhor. As opções disponíveis são várias, e elas eliminam, senão todas, a maioria das preocupações relativas ao desenvolvimento de *backend* para aplicações de software.

Dito isto, o objetivo deste trabalho é avaliar com objetividade o quanto de esforço de desenvolvimento e manutenção deve haver para a construção de um *frontend*, quando ele trabalha em conjunto com um determinado *backend*. E calcular o mesmo esforço quando precisamos usar outra plataforma de *backend* como serviço. Assim, poderemos ter um mínimo de qualidade para comparar os esforços de construção de um *frontend* relativo aos serviços de *backend* atualmente comercializados no mercado de serviços de software.

O *frontend* escolhido para ser produzido do zero foi um aplicativo móvel de agendamento de consultas em uma clínica, pelo seu potencial de tornar-se um produto real por meio de parcerias futuras. Em relação aos *backends*, o desenvolvimento de uma API do zero, que é o método tradicional, não será abordado neste trabalho por ser obviamente a opção mais custosa em termos de tempo. Portanto, foi dada a preferência aos BaaS. Foram escolhidos dois produtos, o Hasura e o Ycodify.

3.1 Backend as a Service

As aplicações de hoje em dia, tanto web, quanto mobile, geralmente estão dividas em duas partes de responsabilidades distintas: o *frontend* e o *backend*. O *frontend* corresponde à interface de uso da aplicação, sendo o meio no qual o usuário interage com o sistema. No caso de uma aplicação web, por exemplo, seria o site acessível num navegador. No caso de uma aplicação mobile, a própria aplicação representa o *frontent*. A grande maioria desses programas citados não trabalham *offline* e precisam se comunicar com um servidor remoto, pois não trabalham com dados armazenados nele. Esse é o *backend*, a parte do projeto que

serve de meio para a aplicação usar o banco de dados. Essa camada é comumente escondida do usuário, apesar de ser a mais robusta e mais custosa para construção e manutenção. Ou seja, o *frontend* deve ser meramente um meio pelo qual o usuário faz requisições específicas ao *backend*, que deve interpretar a demanda, montar uma resposta com as informações pedidas e responder de volta. Em caso de falha de operação no *backend*, ele deve apresentar uma mensagem de erro. Logo, em teoria, cabe ao *backend* realizar o tratamento dos dados antes de serem enviados e identificar inconsistências nos dados recebidos, ficando com processamento mais complexo e trabalhoso. Quanto ao *frontend*, cabe enviar dados consistentes e apenas apresentar os dados recebidos, sem a necessidade de realizar o computações onerosas, sobretudo, em caso de uso de dispositivos móveis.

É do senso comum o desequilíbrio laboral entre a construção de um *frontend* e um *backend*, sendo esse segundo o que demanda mais esforço. Isso se deve ao fato de seu desenvolvimento envolver uma série de variáveis não triviais, como Autenticação de usuários, Gerenciamento de permissões, Armazenamento, Serviço de notificações, Integração com serviços externos, Performance e Escalabilidade.

Um *backend* mal projetado e uma infraestrutura mal planejada podem ser decisivos para o fracasso de uma aplicação. O grande desafio atrelado ao desenvolvimento e à manutenção de ambos fez surgir soluções que tornassem tal trabalho menos repetitivo e mais automatizado, são os chamados BaaS.

O BaaS, é uma plataforma que automatiza a construção do *backend*, ao mesmo tempo que cuida de toda infra-estrutura envolvida na sua disponibilização na web. Este modelo mostrou potencial e motivou a criação de *startups* para desenvolver e fornecer os serviços de BaaS, graças à grande demanda de pequenos desenvolvedores que não tinham tempo, conhecimento ou recursos para cuidar de seu *backend*. O funcionamento geralmente consiste em fornecer um esquema de entidades, que serão traduzidos em tabelas, e especificar suas relações, assim como ocorre nos SGBDs relacionais. Após criadas as tabelas e seus relacionamentos, a plataforma trata de gerar por conta própria uma API REST ou GraphQL pronta para uso que já permite as operações de leitura, escrita e exclusão de dados. Ou seja, pôde-se substituir um projeto que muitas vezes duraria semanas para estar pronto, requisitaria mais de uma pessoa para desenvolver, produziria várias linhas de código e a priori executaria apenas localmente, por um único artefato, o esquema de entidades, que gera um *backend* automaticamente.

Muitas empresas, *startups* e desenvolvedores *freelancers* têm avaliado usar BaaS para reduzir custos e adiantar a entrega de MVPs de suas aplicações. Contudo, esse modelo,

que parecia chegar para acabar com o desenvolvimento *backend*, ainda apresenta alguns desafios em relação ao método tradicional. Resumidamente, é preciso ter em mente as seguintes características sobre o modelo BaaS.

Vantagens:

- Redução no investimento de tempo e dinheiro na infraestrutura da aplicação, bem como na manutenção. O cliente não se preocupa com infraestrutura, balanceamento de carga, disponibilidade e segurança. Essas coisas ficam de responsabilidade da empresa que oferece o serviço;
- Fica dispensada a necessidade de gerenciar dezenas de pastas, com dezenas de arquivos, com centenas de linhas de código cada. O cliente deve, no máximo, definir as entidades, seus relacionamentos e as regras de negócio. Assim, o esforço resume-se à modelagem do sistema. Os desenvolvedores codificam quase exclusivamente o *front*.
- O serviço atende virtualmente a uma infinidade de sistemas ao mesmo tempo.

Desvantagens:

- *Vendor-Lock in*, ou seja, não há como migrar automaticamente de um BaaS para outro, dado que eles não seguem os mesmos padrões e não usam as mesmas tecnologias. Começar a usar um novo BaaS significa reconstruir o projeto na nova plataforma, mesmo que o cliente já tenha seu projeto bem estabelecido em outro lugar.
- Ainda não é possível se livrar completamente do desenvolvimento. Conforme o sistema se tornam complexos e específicos, é possível que os desenvolvedores optem por fazer algumas funcionalidades por conta própria. Esse tipo de situação pode conduzir ao caso de não compensar mais pagar pelo BaaS, devido a grande quantidade de funcionalidades feitas localmente

A seguir apresentaremos brevemente as características de cada um dos serviços escolhidos, primeiro relativo ao Hasura, em segundo o Ycodify.

3.2 Provedores de BaaS

3.2.1 Hasura

Hasura é um *Backend as a Service* voltado para a construção de APIs que respondem à linguagem GraphQL e que usa o banco de dados *Postgres*. Uma das maiores vantagens

do Hasura é a redução de codificação no processo de construção do *Backend*, todas as tabelas do banco são cadastradas e visualizadas através da interface gráfica no site. Outra vantagem é o fato de trabalhar com APIs GraphQL, permitindo ao desenvolvedor realizar consultas inteligentes que buscam no banco exatamente o que se é pedido na *Query*.

Com as tabelas todas prontas, é possível realizar requisições a fim de populá-las e consultá-las usando o console GraphiQL, disponível no site. Esse console auxilia o desenvolvedor criando requisições automaticamente via interface gráfica, ou seja, sem a necessidade de escrita de *queries*. Nessa fase inicial, são feitos os testes iniciais antes de lançar publicamente o BaaS.

Todo esse processo pode ser feito por alguém com pouco conhecimento tanto em SGBDs quanto em SQL , como é o caso do autor. Da perspectiva de um desenvolvedor majoritariamente *Frontend* com pouca experiência em *Backend*, pode-se dizer que a plataforma facilita a construção e manutenção de servidores dando a oportunidade de trabalhar autonomamente como *Full-stack* em projetos pequenos.

No Hasura, criar *Schemas* e suas tabelas é bastante facilitado devido à sua interface. Para gerar uma nova tabela, basta inserir a lista de colunas, especificar qual é a chave primária, quais são as chaves estrangeiras e com quais outras tabelas essas fazem referência. Em cada coluna, é possível determinar o nome, o tipo do dado, se é um valor *Nullable* e se é *Unique*. Após esse preenchimento, o Hasura gera automaticamente o comando SQL de criação e executa-o no banco de dados remoto que ele fornece. Após geradas as tabelas, o *backend* já está pronto para ser preenchido com dados e responder requisições de um *frontend*.

Para realizar quaisquer operações na plataforma, por padrão é necessário comunicar-se enviando requisições escritas na linguagem GraphQL. Há três tipos de requisições: *Query*, *Mutation* e *Subscription*. A *query* é usada exclusivamente para realizar leitura no banco, sem alterar nenhum dado. Equivale ao método GET no padrão REST. A *mutation* realiza ao mesmo tempo inserção e atualização de dados no banco. Equivale às operações POST, PUT, PATCH e DELETE no padrão REST. A *subscription* é usada exclusivamente para consultas, assim como a *query*, mas seu diferencial em relação à ambas as operações anteriores é a atualização de *socket* para manter a conexão ativa com o servidor a fim de mandar uma nova resposta sempre que que algum dos dados consultados sofrerem alteração. Logo, enquanto a *query* é usada para fazer uma leitura e receber uma única resposta antes da conexão ser finalizada, a *subscription* é uma *query* que permite ao *frontend* a observação em tempo real dos atributos de uma entidade do *backend*.

3.2.2 Ycodify

Ycodify é um *Backend as a Service*, como o Hasura, voltado para a oferta de serviço de computação em nuvem para desenvolvedores de software que desejem serviços de *backend* para suas aplicações. Ele faz isso sem expor o banco de dados e sua estrutura, não é necessário saber da implementação por trás. A vantagem de seu uso é porque acelera o desenvolvimento da aplicação oferecendo uma interface CLI para que possamos descrever o modelo de dados da aplicação. Esse modelo de dados é do tipo entidade-relacionamento. Depois que descrevemos o modelo para a plataforma, precisamos utilizar uma tecnologia própria da plataforma que possui semelhanças com GraphQL, para que operações de escrita e leitura de dados possam ser realizadas na base de dados da plataforma.

Para utilizar o Ycodify, é preciso aprender a usar os comandos do CLI para que os dados da aplicação sejam descritos na plataforma. Assim, profissionais que constroem *Frontends* com nenhuma ou pouca experiência em *Backend* podem ter o trabalho bem reduzido para construir a aplicação.

Assim como a maioria dos BaaS, o Ycodify dispensa que os profissionais precisem se preocupar com a instalação, ou configuração de serviços de infraestrutura para que o *backend* possa funcionar.

Para que possamos usar o Ycodify precisamos realizar o download e instalação da CLI oferecida pela plataforma. Depois, é preciso criar uma conta de usuário e para então serem descritos os modelos de dados. Por meio desse modelo são especificados detalhes dos atributos das entidades que informam, por exemplo, a unicidade do espaço de valores do atributo. Também é possível definir se valores para um determinado atributo pode ser anulado ou não, além é claro, do tipo do valor do atributo. Podemos também definir metadados a nível da entidade, como por exemplo, se é preciso fazer controle de acesso em termos de concorrência, ou de autorizações.

Apesar de descrevermos esse modelo, não sabemos de que forma isso é mapeado para a base de dados. A plataforma nos impede de acessar essa estrutura a não ser por meio do CLI da plataforma.

Após a especificação do modelo, o serviço já está pronto para uso.

4 Metodologia

4.1 Descrição do ambiente de trabalho

A máquina usada para a produção desse trabalho roda o sistema operacional Windows 10 Pro. Nele, foi instalado o Windows Terminal versão 1.11.3471.0 executando o ambiente de comandos Powershell. O frontend da aplicação foi desenvolvida usando as tecnologias Angular 12.1.5, Ionic 6.0.0 e Typescript 4.2.4. O node.js instalado é o 14.17.3. O pacote usado para integrar o backend Hasura com o front foi o Apollo na versão 2.6.0. Para testar a API REST do Ycodify, foi usado o Insomnia como cliente HTTP. Para o cálculo das métricas de código estático, foi instalado a extensão do VSCode JavaScript Complexity Analysis na versão 2.0.0. Como essa extensão não funciona em versões posteriores à 1.32.3 do VS Code, foi feito o *downgrade* da IDE para esse valor. Por fim, foram usados o Lizard 1.17.9 e o Python 3.9.0.

4.2 As ferramentas e métricas usadas

4.2.1 Lizard

Lizard é uma solução *open-source* para o cálculo da CC com suporte para várias linguagens de programação. Adicionalmente, ele é capaz de realizar várias análises de código estático (e.g., detectar código duplicado).

O relatório final do Lizard apresenta primeiramente as funções e métodos analisados e, em seguida, os arquivos nos quais estão contidos, ambos dispostos numa tabela com as devidas métricas. Mais à frente estão as *Warnings* configuradas pelo usuário e, finalmente, a tabela final com algumas métricas globais. O Lizard gera um relatório com as seguintes colunas:

Para cada função do um arquivo:

- NLOC: número de linhas desconsiderando comentários
- CCN: complexidade ciclomática
- token: quantidade de tokens
- param: quantidade de parâmetros

Para cada arquivo:

- LOC: número de linhas desconsiderando comentários
- AvgNLOC: média do NLOC de cada função
- AvgCCN: média do CNN das funções
- Avg.token: média de tokens das funções
- function_cnt: quantidade de funções

Globalmente:

- Total NLOC: somatório do LOC de todos arquivo
- Avg.NLOC: média entre LOCs de cada arquivo
- Avg CCN: média entre CCNs de todas funções
- Avg token: média de tokens de todas funções
- Fun Cnt: quantidade de todas as funções
- Warning cnt: quantidade de warnings

4.2.2 Escomplex

O Escomplex é um pacote do Node.js que calcula um conjunto de métricas tais como:

- Linhas de código
- Número de parâmetros de funções
- Complexidade ciclomática de McCabe (MCCABE, 1976)

- Índice de Manutenibilidade (OMAN; HAGEMEISTER, 1992)

A ferramenta oferece suporte apenas ao Javascript. Contudo, devido à escassez de ferramentas que suportassem nativamente a linguagem Typescript, o Escomplex foi usado como alternativa confiável. Essa mudança de planos foi viabilizada pela disponibilidade de compiladores que fizessem a tradução de uma linguagem pra outra.

Entretanto, devido à problemas técnicos que impossibilitaram a execução do pacote diretamente via terminal, foi usado sua versão disponível como extensão na IDE VS Code. O JavaScript Complexity Analysis serve de interface gráfica para o Escomplex e apresenta os resultados da análise organizados em um site renderizado dentro do VS Code.

4.3 Descrição da aplicação

Para iniciar, foi utilizado o pacote node chamado "@ionic/cli", que possibilita, por meio de comandos no terminal, a criação de um projeto Angular já integrado ao pacote "@ionic/angular". Por mais que o Angular seja um robusto *framework* da Google voltado para a criação de *frontends* web e móvel, foi usado o Ionic como uma camada a mais de estilização. Esse último pacote fornece vários componentes visuais prontos para uso bastante recorrentes nas interfaces de *apps* de sucesso no mercado. Exemplos de componentes estilizados que o Ionic oferece são botões, ícones, entradas de texto, menus de múltiplas escolhas etc.

O *app* foi desenvolvido imaginando-se um sistema de gerências de clínicas, de profissionais e agendamento de consultas. Sendo assim, haveria 3 tipos de usuários:

- Administradores de clínicas: com permissão de agendar atendimentos, cadastrar profissionais da saúde e acessar/escrever seus dados;
- Profissionais da saúde: com permissão de confirmar atendimentos agendados, cadastrar seus dados profissionais e emitir documentos relacionados à sua profissão (atestado, receituário, laudo etc);
- Pacientes: com permissão de solicitar um atendimento, ver atendimentos anteriores, ver documento emitidos para ele e, por fim, cadastrar seus dados pessoais.

Para tal, o diagrama de classes foi disposta conforme é mostrado na figura 3.

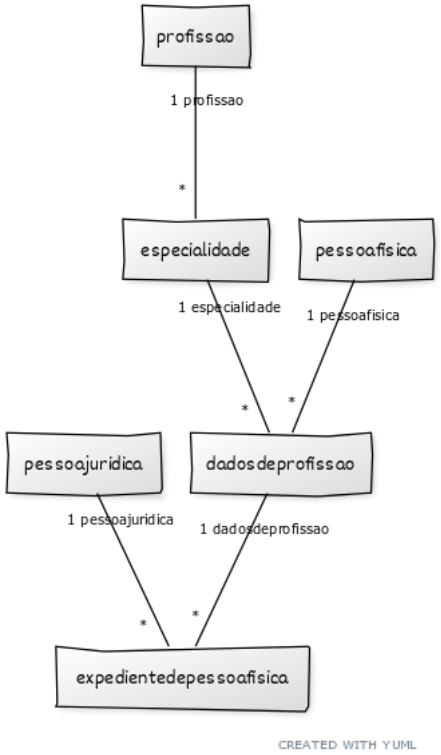
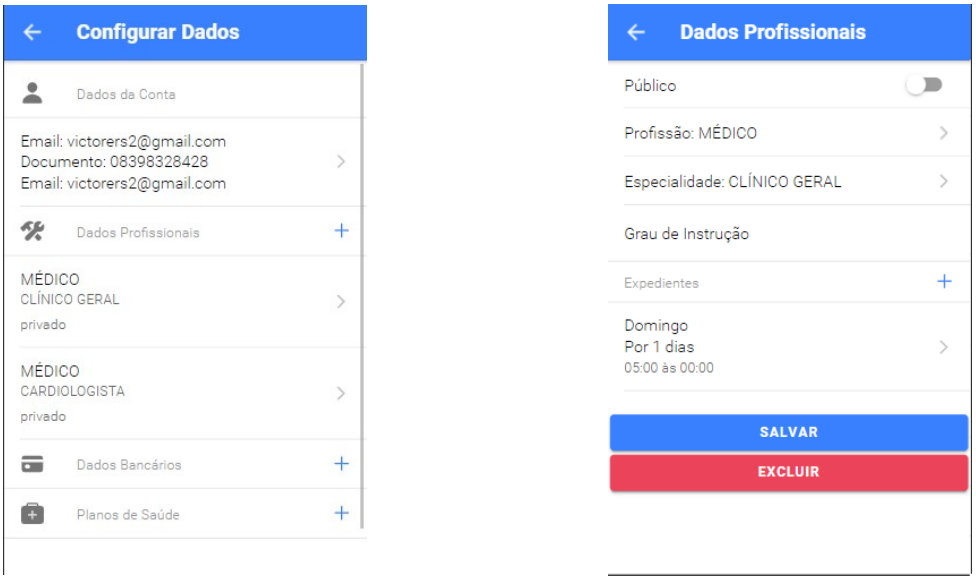


Figura 3: Diagrama de classes inserido nos BaaS

Apesar dessa complexidade na idealização, a maioria das telas precisaram apenas ser populadas com dados fictícios e não foram efetivamente integradas a algum *backend*. Isso porque, visando simplificar o trabalho, dentre todas as funcionalidades citadas acima, foram escolhidas apenas as necessárias para o cumprimento das operações de leitura, escrita, atualização e deleção sobre uma das entidades.

Foi dado um foco maior nas telas que seriam usadas pelo usuário com papel de Profissional. Portanto, foi construída a tela de cadastro de dados profissionais. Nela, o usuário pode pôr várias entradas do item que representa suas experiências profissionais atuais. Um "Dado de profissão" contém basicamente a informação da profissão, especialidade e uma lista de expedientes nos quais determinada profissão é exercida. Um expediente contém o dia da semana inicial, a recorrência ao longo da semana e as horas de início e término. Tomamos a centralidade da entidade "expedientedepessoafisica" por ela sintetizar a relação entre diversas entidades, algo relevante para atribuir mínimo de complexidade relativo às teias de relacionamento dos dados. Essa relação possibilita a leitura de dados realizando um *join* entre as entidades "dadosdeprofissao", "expedientedepessoafisica" e "pessoajuridica", pois por essas entidades o *app* pode-se encontrar um profissional que trabalha em determinada clínica e atende em determinada hora de um dia específico, sob o título de uma profissão/especialização.



(a) Visualização de dados pessoais (b) Editação e criação de dado de profissão

Figura 4: Exemplos de tela do aplicativo desenvolvido

4.4 Alvo da análise

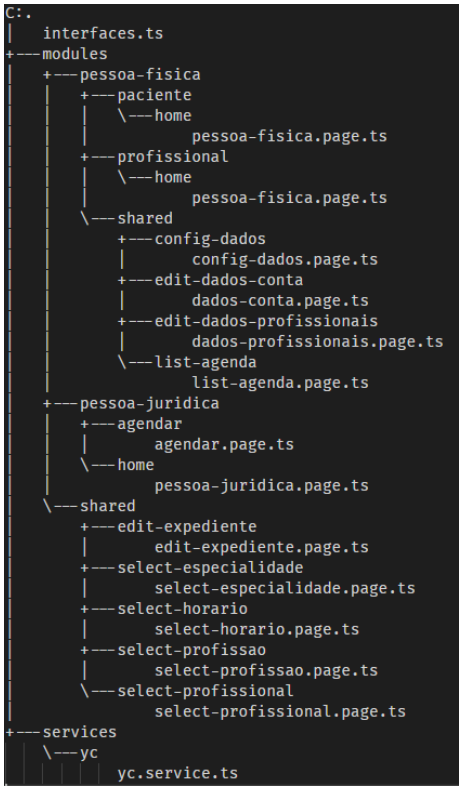
As principais pastas do *frontend* desenvolvido podem ser vistas em uma forma simplificada nas imagens 5b e 5a.

Por motivos de espaço limitado, as imagens acima contam apenas com a pastas *modules* e *services* pois foram as que efetivamente apresentaram diferenças de implementação entre as duas versões. As páginas do *app* foram divididas em três módulos independentes, por isso vê-se a pasta "modules", que não deve ser confundida com a "node_modules". Além disso, foram omitidos da visualização acima os arquivos HTML , CSS , os de módulo e roteamento do Angular, com extensões ".module.ts" e "routing.module.ts", respectivamente. As pastas "components", "models", "pipes" e "utils" também foram omitidas da imagem acima pois não sofreram alterações por causa da mudança de *backend*. Elas estão dispostas no mesmo nível de "modules", como visto na figura 6:

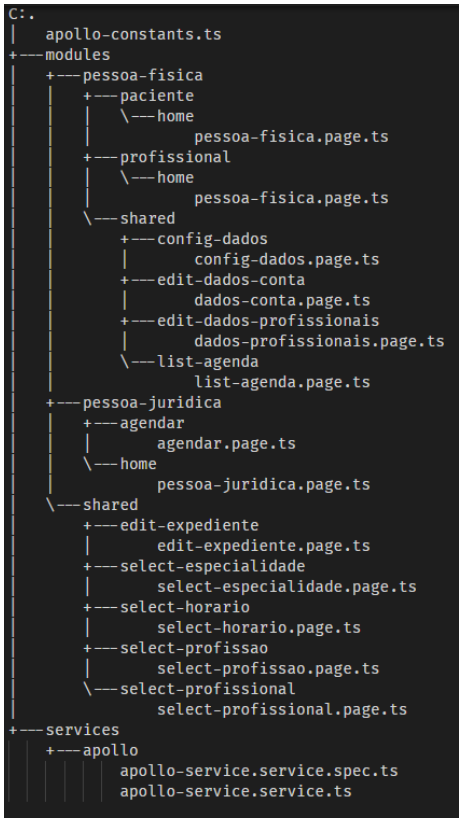
4.5 Descrição da análise estática do código

4.5.1 Download e Instalação das ferramentas

O Lizard foi instalado por meio do Package Installer for Python (PIP) por meio do comando `pip install lizard`. Após a conclusão da execução, a ferramenta é adicionada automaticamente como variável de ambiente do sistema e está pronto para uso via



(a) Pastas do projeto feito com Ycodify



(b) Pastas do projeto feito com Hasura

Figura 5: Pastas do projeto

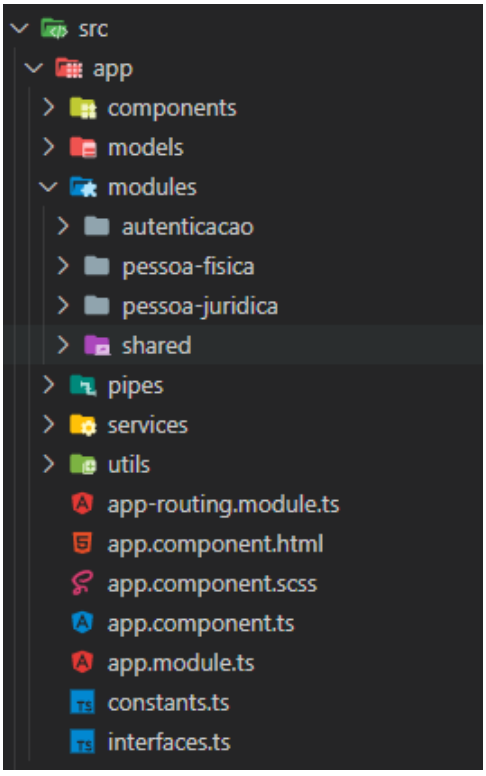


Figura 6: Visão geral da estrutura de pastas do projeto

terminal.

O JS Complexity Analysis, por ser uma extensão do VS Code, tem o processo de instalação facilitado pela interface gráfica da IDE.

4.5.2 Comandos Usados

Para executar o Lizard a fim de calcular as métricas dos arquivos Typescript de uma determinada árvore de pastas, foi usado o comando `lizard.exe -l typescript .\app\`. No caso, a raiz da árvore de pastas que contém o código fonte alvo chama-se "app".

A execução do JS Complexity Analisys foi possível por meio do recurso chamado "Paleta de comandos", do VS Code, como visto na figura 7. Com ele, o usuário pode executar comandos pré-programados pelo VS Code ou por suas extensões. Ao pesquisar pelo nome do comando visto na documentação, há as opções de calcular a complexidade em um arquivo ou em todos da pasta atual. Foi executado o comando que considera todos os códigos numa pasta.

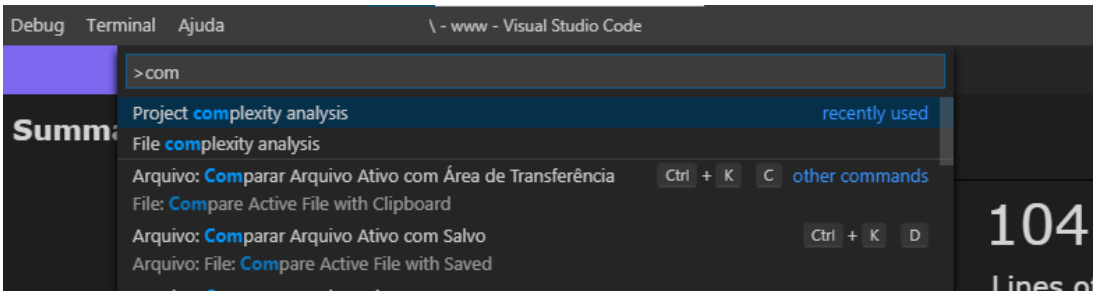


Figura 7: Paleta com o comando executado

Como a extensão não trabalha com Typescript, os arquivos do projeto foram compilados para JavaScript antes de serem usados para análise por meio do comando `ng build`. Ou seja, a compilação foi feita por meio do próprio Angular, uma vez que o artefato final do *framework* é um *frontend* escrito nas linguagens padrões da web, isto é, HTML, CSS e JavaScript puro localizados na pasta "www". Por padrão, o *build* final contém vários arquivos que dizem respeito exclusivamente aos *frameworks* Angular e Ionic.

4.5.3 Saída dos programas

A saída do Lizard consiste em um texto impresso no próprio terminal. A saída do JS Complexity Analisys é um arquivo HTML renderizado dentro da IDE assim que os cálculos terminam. A página contém o MI, SLOC e os CC médios e máximos separados

por arquivos. Em nome da simplificação, foram calculadas as médias dos valores nas colunas referentes à complexidade. Há também uma última coluna que presume apresentar uma estimativa para a quantidade de erros em um arquivo. Esses valores não foram considerados pela falta de clareza na sua obtenção.

5 Resultados

Abaixo, os resultados obtidos. As primeira seções representam as ferramentas utilizadas para o cálculo e depois é feita uma discussão.

5.1 Lizard

Nas tabelas 1 e 2, estão os resultados das análises do Lizard para o código escrito em Typescript e para a sua transpilação para Javascript, respectivamente.

	Total nloc	AvgCCN
Hasura	3492	1.1
YCodify	2991	1.1

Tabela 1: Valores calculados pelo Lizard após analisar código Typescript

	Total nloc	AvgCNN
Hasura	3492	1.1
YCodify	4226	1.3

Tabela 2: Valores calculados pelo Lizard após analisar código Javascript

5.2 JS Complexity Analysis

Esta ferramenta trabalha apenas com a linguagem Javascript, por isso só foi possível obter a tabela 3.

	Média das complexidades dos arquivos	Média das complexidades máximas em cada arquivo	MI
Hasura	1.1809	4.2857	71.2
YCodify	1.1842	5.7368	70.3

Tabela 3: Valores calculados pelo JS Complexity Analysis após analisar código Transpilado

5.3 Discussão

5.3.1 Hasura

Antes da comparação entre os *backends*, é importante notar a discrepância no tamanho entre o código Typescript e o Javascript. Olhando apenas a saída do Lizard, é possível perceber que, após a compilação, vê-se na coluna “Total nloc"que o código ganhou 734 linhas. Entretanto, a complexidade não sofreu grande alteração, de 1.1 para 1.3. Podemos concluir que a compilação não adiciona significativamente novos comandos, mas sim código não executáveis como: linhas vazias, comentário e linhas contendo apenas abertura e fechamento de chaves. Ao dar uma breve olhada em alguns arquivos no resultado da compilação, vê-se que de fato este é o caso, como é visto no código Javascript na fig. 8.

```
(self["webpackChunkdrq_ionic"] = self["webpackChunkdrq_ionic"] || []).push([["src_app_modules_pessoa-fisica_paciente_home_pessoa-fisica_module_ts"],{

/***/ 6506:
/*!*****************************************************************************************!*\
  !*** ./src/app/modules/pessoa-fisica/paciente/home/pessoa-fisica-routing.module.ts ***!
  \*****************************************************************************************/
/***/ ((__unused_webpack_module, __webpack_exports__, __webpack_require__) => {

"use strict";
__webpack_require__.r(__webpack_exports__);
/* harmony export */ __webpack_require__.d(__webpack_exports__, {
/* harmony export */   "PessoaFisicaPageRoutingModule": () => (/* binding */ PessoaFisicaPageRoutingModule)
/* harmony export */ });
/* harmony import */ var tslib__WEBPACK_IMPORTED_MODULE_2__ = __webpack_require__(/*! tslib */ 4762);
/* harmony import */ var _angular_core__WEBPACK_IMPORTED_MODULE_3__ = __webpack_require__(/*! @angular/core */ 7716);
/* harmony import */ var _angular_router__WEBPACK_IMPORTED_MODULE_4__ = __webpack_require__(/*! @angular/router */ 9895);
/* harmony import */ var src_app_constants__WEBPACK_IMPORTED_MODULE_0__ = __webpack_require__(/*! src/app/constants */ 4854);
/* harmony import */ var _pessoa_fisica_page__WEBPACK_IMPORTED_MODULE_1__ = __webpack_require__(/*! ./pessoa-fisica.page */ 1437);




const drqRoutes = new src_app_constants__WEBPACK_IMPORTED_MODULE_0__.DRQRoutes();
const routes = [
    {
        path: '',
        component: _pessoa_fisica_page__WEBPACK_IMPORTED_MODULE_1__.PessoaFisicaPage,
    },
    {
        path: drqRoutes.AGENDA,
        loadChildren: () => __webpack_require__.e(/*! import() */ "src_app_modules_pessoa-fisica_shared_list-agenda_list-agenda_module_ts").then(__webpac
    },
    {
        path: drqRoutes.CONFIG_DADOS,
        loadChildren: () => __webpack_require__.e(/*! import() */ "src_app_modules_pessoa-fisica_shared_config-dados_config-dados_module_ts").then(__webpa
    },
];
let PessoaFisicaPageRoutingModule = class PessoaFisicaPageRoutingModule {
};
PessoaFisicaPageRoutingModule = (0,tslib__WEBPACK_IMPORTED_MODULE_2__.__decorate)([
    (0,_angular_core__WEBPACK_IMPORTED_MODULE_3__.NgModule)({
        imports: [_angular_router__WEBPACK_IMPORTED_MODULE_4__.RouterModule.forChild(routes)],
        exports: [_angular_router__WEBPACK_IMPORTED_MODULE_4__.RouterModule],
    })
], PessoaFisicaPageRoutingModule);
```

Figura 8: Exemplo de código Javascript após compilação

5.3.2 YCodify

Neste caso, o ganho de linhas de código após a compilação para Javascript é ainda maior, 1.138 a mais. Consequentemente, o ganho de complexidade também foi maior nas colunas Avg.Token e Fun Cnt. Possivelmente, alguns trechos de código ou recursos do Typescript nesta versão do app provoca um código bem maior que o visto na seção do Hasura.

5.3.3 Comparação

Segundo os indicadores na saída do Lizard, analisando o código Typescript, têm-se que o *app* usando o YCodify é ligeiramente mais promissor. Porém, segundo os indicadores do JS Complexity Analysis, ocorre o inverso, os indicadores referentes ao Hasura são ligeiramente melhores. Na verdade, ambas as ferramentas, quando tomam como base o código Javascript, apontam que o *backend* do Hasura possibilitou um *frontend* ligeiramente mais manutenível, mas quando analisam o código Typescript avaliam o inverso.

Foi uma surpresa obter que a versão do *frontend* escrita para o *backend* do YCodify tem indicadores ligeiramente melhores que a versão usando o Hasura considerando os artefatos escritos em Typescript, mas não em Javascript. Apesar disso, os resultados para cada tipo de *backend* não são tão discrepantes a ponto de poder-se dar o veredito da superioridade de um sobre outro do ponto de vista do desenvolvedor. Porém, dado os históricos de ambas, o Hasura é disparado a ferramenta mais confiável dentre os dois para se utilizar em projetos reais de larga escala. Ele já está disponível em produção há anos e foi bastante adotado pela comunidade, contando com maior maior conteúdo de ajuda disponível online.

6 Conclusão

Finalizados os *frontends*, foi possível olhar para trás e tirar algumas conclusões sobre todo o processo de desenvolvimento. O esforço empregado para o uso do Hasura e YCodify foram bastante semelhantes. Porém fazendo uma análise considerando fatores externos e contextos mercadológicos, o Hasura parece ser a opção mais confiável devido à alguns motivos periféricos à ferramenta como documentação, interface gráfica, grande quantidade de usuários ativos, tutoriais e cursos online proporcionado por uma comunidade ativa.

6.1 Trabalhos Futuros

Foi visto que determinados tipos de códigos Typescript são capazes de gerar mais ou menos código Javascript quando compilados. Um possível trabalho futuro pode ter como objeto a identificação de padrões que interferem no tamanho do código do artefato final. Outro trabalho, que pode ser considerado de urgente relevância, é a produção de mais ferramentas voltadas para as linguagens Typescript e Javascript que calculam métricas voltadas ao paradigma de Orientação a Objetos.

7 Considerações finais

O maior impeditivo para uma análise mais aprofundada com certeza foi a escassez de programas gratuitos que calculassem um bom número de métricas para as linguagens adotadas, isto é, HTML, CSS e Typescript. Na pesquisa realizada foi visto que apenas Java e C++ têm disponíveis uma quantidade satisfatória de programas analisadores de código estático ao mesmo tempo gratuito e robustos, como o CKJM, JHawk etc. Porém, de forma geral, há poucas ferramentas *open-source* que entregam algo além do LOC e CC. Além disso, foi observado nos poucos programas disponíveis uma baixa qualidade em suas documentações, que muitas vezes se resumem a um brevíssimo README.md no repositório do Github, como é o caso do Escomplex.

Outra dificuldade encontrada é o abandono dos projetos por parte de seus criadores, muitos deles não conseguiram ser executados localmente e, em todos, a última atualização data de vários anos atrás. Houve algumas tentativas de incluir neste trabalho outras ferramentas de análise de código Javascript, como: Plato, JSC, grunt-complexity e o complexity-report. Mas elas ou estão depreciadas ou têm a documentação insuficiente para a utilização ou não têm suporte para a especificação atual do Javascript.

Seria de grande importância para a comunidade que surgissem soluções gratuitas que facilitasse o cálculo de métricas mais relevantes, como as de CK , para linguagens mais voltadas para o desenvolvimento no lado do *Frontend*.

Referências

ALIJA, N. Justification of software maintenance costs. *International Journal of Advanced Research in Computer Science and Software Engineering*, v. 7, n. 3, p. 15–23, 2017.

ARDITO, L. et al. A tool-based perspective on software code maintainability metrics: a systematic literature review. *Scientific Programming*, Hindawi, v. 2020, 2020.

COMMITTEE, I. S. C. et al. Ieee standard glossary of software engineering terminology (ieee std 610.12-1990). los alamitos. *CA: IEEE Computer Society*, v. 169, p. 132, 1990.

FENTON, N. E.; NEIL, M. Software metrics: successes, failures and new directions. *Journal of Systems and Software*, Elsevier, v. 47, n. 2-3, p. 149–157, 1999.

GRAYLIN, J. et al. Cyclomatic complexity and lines of code: empirical evidence of a stable linear relationship. *Journal of Software Engineering and Applications*, Scientific Research Publishing, v. 2, n. 03, p. 137, 2009.

HEITLAGER, I.; KUIPERS, T.; VISSER, J. A practical model for measuring maintainability. In: IEEE. *6th international conference on the quality of information and communications technology (QUATIC 2007)*. [S.l.], 2007. p. 30–39.

MCCABE, T. J. A complexity measure. *IEEE Transactions on software Engineering*, IEEE, n. 4, p. 308–320, 1976.

MOLNAR, A.; MOTOGNA, S. Discovering maintainability changes in large software systems. In: *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement*. [S.l.: s.n.], 2017. p. 88–93.

NUÑEZ-VARELA, A. S. et al. Source code metrics: A systematic mapping study. *Journal of Systems and Software*, Elsevier, v. 128, p. 164–197, 2017.

OMAN, P.; HAGEMEISTER, J. Metrics for assessing a software system's maintainability. In: IEEE COMPUTER SOCIETY. *Proceedings Conference on Software Maintenance 1992*. [S.l.], 1992. p. 337–338.

VASCONCELOS, J. B. D. et al. The application of knowledge management to software evolution. *International Journal of Information Management*, Elsevier, v. 37, n. 1, p. 1499–1506, 2017.