Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Bacharelado em Ciência da Computação

ESTUDO SOBRE O IMPACTO DA ADOÇÃO DA COMPUTAÇÃO SERVERLESS EM APLICAÇÕES OPEN SOURCE

Juliana Aparecida de Paula Silva

JUIZ DE FORA
JANEIRO, 2023

ESTUDO SOBRE O IMPACTO DA ADOÇÃO DA COMPUTAÇÃO SERVERLESS EM APLICAÇÕES OPEN SOURCE

Juliana Aparecida de Paula Silva

Universidade Federal de Juiz de Fora

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Bacharelado em Ciência da Computação

Orientador: Gleiph Ghiotto Lima de Menezes

JUIZ DE FORA

JANEIRO, 2023

ESTUDO SOBRE O IMPACTO DA ADOÇÃO DA COMPUTAÇÃO SERVERLESS EM APLICAÇÕES OPEN SOURCE

Juliana Aparecida de Paula Silva

MONOGRAFIA SUBMETIDA AO CORPO DOCENTE DO INSTITUTO DE CIÊNCIAS EXATAS DA UNIVERSIDADE FEDERAL DE JUIZ DE FORA, COMO PARTE INTEGRANTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE BACHAREL EM CIÊNCIA DA COMPUTAÇÃO.

Aprovada por:

Gleiph Ghiotto Lima de Menezes
Doutor em Computação

André Luiz de Oliveira
Doutor em Ciência da Computação

José Maria Nazar David
Doutor em Engenharia de Sistemas e Computação

JUIZ DE FORA

12 DE JANEIRO, 2023

Resumo

O número de aplicações *Web* e usuários conectados à Internet cresce exponencialmente ao longo dos anos. Com isso, as maneiras de se construir um *software* precisaram ser repensadas para atender escalabilidade e velocidade de entrega. Nos últimos anos, surgiu uma nova proposta, conhecida como *serverless*, cujo objetivo é que os desenvolvedores possam focar na construção das regras de negócio e que a demanda de gerenciamento de infraestrutura seja atendida pelo provedor de *cloud*. Portanto, este trabalho visa realizar um estudo comparativo a respeito de como a adoção da computação *serverless* pode impactar na construção de um sistema, realizando uma pesquisa em repositórios *open source* e coletando amostras de projetos monolíticos e *serverless* desenvolvidos nas linguagens de programação *Python* e *JavaScript*. Os resultados foram avaliados em relação a quantidade e o tamanho dos arquivos dos projetos, além da quantidade de funções, métodos e classes e como a complexidade ciclomática e o tamanho das funções e métodos se comportam nos dois cenários. A análise demonstrou que as aplicações *serverless* tendem a apresentar projetos com mais arquivos e com média superior de linhas, bem como a presença de funções e métodos maiores e com complexidade ciclomática superior.

Palavras-chave: Métricas de software; arquitetura; *cloud*; microsserviços.

Abstract

The number of Web applications and users connected to the Internet grows exponentially over the years. As a result, the ways of building software needed to be rethought to meet scalability and delivery speed. In recent years, a new proposal has emerged, known as serverless, whose objective is that developers can focus on building business rules and that the infrastructure management demand is met by the cloud provider. Therefore, this work aims to carry out a comparative study on how the adoption of serverless computing can impact the construction of a system, conducting a research in open source repositories and collecting samples of monolithic and serverless projects developed in the programming languages Python and JavaScript. The results were evaluated in relation to the number and size of project files, as well as the number of functions, methods and classes and how the cyclomatic complexity and size of functions and methods behave in both scenarios. The analysis showed that serverless applications tend to present projects with more files and with a higher average of lines, as well as the presence of larger functions and methods and with higher cyclomatic complexity.

Keywords: Software metrics; architecture; cloud; microservices.

Agradecimentos

A Deus pela força e discernimento que me foram concedidos durante todos esses anos para que eu não desistisse.

Principalmente a mim, por todos os inúmeros dias de esforço, dedicação extrema e noites estudando.

A meu orientador, Gleiph, pela paciência e motivação constante para que sempre que eu desanimava com a construção do trabalho, ele estava lá para ajudar a tornar mais fácil esse caminho.

A minha equipe de trabalho que me proporcionou muitos conhecimentos que me induziram a esse estudo. E aos professores do DCC que, ao longo desses 4 anos, me ensinaram diversos conceitos que contribuíram para minha formação profissional.

"Nós somos aquilo que fazemos repetidamente. Excelência, então, não é um modo de agir, mas um hábito.".

Aristóteles

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Abreviações

DCC Departamento de Ciência da Computação

UFJF Universidade Federal de Juiz de Fora

1 Introdução

1.1 Apresentação do tema

A computação *serverless* é um novo modelo que permite o desenvolvimento de aplicações sem a necessidade de gerenciamento de infraestrutura. Com *serverless*, a lógica da aplicação é dividida em funções que são sem estado e são direcionadas individualmente para o provedor de *cloud* para serem dinamicamente gerenciadas. As funções são invocadas por eventos, como uma requisição HTTP, e, após finalizarem sua responsabilidade, elas são desativadas e não salvam ou levam informações para a próxima requisição (GOLI, 2020). Além disso, o modelo de cobrança é baseado no consumo por utilização, no qual há existência de custos baseado no número de requisições e no tempo que o código leva para ser executado, não havendo cobranças no momento em que os recursos não estão sendo utilizados (FAN, 2020).

1.2 Descrição do problema

O número de usuários que acessam a Internet regularmente no mundo se aproximou da marca de 5 bilhões de pessoas em janeiro de 2022, o que corresponde a cerca de 63% da população mundial (KEMP, 2022). Diante disso, novas soluções e tecnologias que rodam na *Web* surgem exponencialmente. Para atender a essa demanda, novas empresas são criadas e as atuais tentam se adaptar a um cenário cada vez mais digital. Como ainda se trata de um ambiente novo, as companhias precisam testar suas propostas de forma rápida, validando se o produto atende de fato as necessidades dos clientes. Como fruto dessa conjuntura, o desenvolvimento de *software*, motor propulsor de toda essa cadeia, também é afetado por essa dinâmica. Atualmente, é necessário pensar não somente em entregar funcionalidades, mas também em questões de escalabilidade para atender a demanda de usuários, arquitetura, construção de um melhor processo de manutenção para o longo prazo, como também, em custos envolvidos no processo.

1.3 Motivação e objetivo

De acordo com Ghosh, Addya, Somy, Nath, Chakraborty e Ghosh (2020), a adoção da computação *serverless* traz como principais benefícios a ausência da necessidade de gerenciamento de servidores, a escalabilidade, alta disponibilidade e a oportunidade de pagar pelos recursos apenas enquanto estão ativos. Contudo, não existem muitos estudos que abordam quais foram os impactos no *design* do *software* com a adoção deste tipo de abordagem, buscando entender se no futuro, o número de problemas com manutenção e complexidade do projeto tenderá a ser superior. Logo, este trabalho tem o intuito de verificar quais são os impactos da adoção da computação *serverless* nos aspectos arquiteturais de um projeto. Para isso, será analisado dados como a quantidade de arquivos, além da quantidade de linhas das funções, dos métodos e dos arquivos, bem como a complexidade ciclomática das funções e métodos e a quantidade de classes, funções e métodos presentes em cada arquivo. Para isso, será utilizado como base uma amostra de repositórios *open source* desenvolvidos nas linguagens de programação *Python* e *JavaScript* que utilizam a arquitetura monolítica e outra amostra que utiliza *serverless*. Essas duas linguagens de programação foram escolhidas pelo fato de serem atualmente as mais populares, principalmente na *AWS Lambda* (TAIBI, 2020). Para nortear o estudo foram elencadas 3 questões de pesquisa que serão respondidas ao longo do desenvolvimento, sendo elas:

- QP1 - A quantidade de arquivos em projetos com arquitetura *serverless* é maior do que a monolítica?
- QP2 - Como os aspectos físicos e sintáticos dos arquivos se comportam em cada uma das duas arquiteturas?
- QP3 - A complexidade ciclomática e o número de linhas das funções e métodos entre as duas arquiteturas apresenta diferenças?

1.4 Organização

Este trabalho está organizado em 5 capítulos, sendo eles: Introdução, que tem por objetivo apresentar de maneira inicial o tema, a Fundamentação Teórica que tem o intuito de

conceituar tópicos importantes para a melhor compreensão do estudo, a Coleta de métricas dos projetos, que visa explicar o processo construído para se extrair as métricas que serão exploradas, o capítulo de Resultados, que tem por objetivo demonstrar como se comportaram os dados obtidos e o que eles podem trazer de conhecimento e, por fim, o capítulo de Conclusão, que visa abordar os principais resultados encontrados e elencar possíveis trabalhos futuros.

2 Fundamentação Teórica

Neste capítulo são apresentados os principais conceitos para compreender o tema da pesquisa. A Seção 2.1 descreve sobre o conceito de computação em nuvem, a Seção 2.2 traz uma breve introdução sobre como funcionam a construção de aplicações monolíticas e em microsserviços. Em contraste a isso, a Seção 2.3 traz as principais características da computação *serverless*. A Seção 2.4 apresenta uma comparação em como as principais plataformas de computação em nuvem trabalham atualmente provendo recursos de funções como serviço, trazendo informações sobre *AWS Lambda*, *Azure Functions* e *Google Cloud Functions*. A Seção 2.5 apresenta uma descrição sobre as métricas de *software* que serão mensuradas ao longo do estudo. A Seção 2.6 descreve como funciona a prática de mineração de repositórios. A Seção 2.7 detalha alguns trabalhos relacionados e suas principais contribuições. Finalmente, a Seção 2.8 traz as considerações finais para este capítulo.

2.1 Computação em nuvem

A computação em nuvem refere-se a uma rede de computadores oferecida por um provedor de serviço, normalmente conectada pela Internet, conferindo acesso sob demanda a recursos de armazenamento e processamento para atender às necessidades do usuário (KÖHLER; BENKNER, 2011). A ilustração da Figura 2.1 demonstra alguns dos serviços oferecidos, como banco de dados, armazenamento, servidores, recursos para dispositivos móveis e diversas outras aplicações, podendo ser provedores públicos, privados ou híbridos.

Figura 2.1: Representação de computação em nuvem - Adaptado de (PETERS, 2019).

Por muitos anos, quem desejasse disponibilizar seu *software* na Internet, precisava contar com uma infraestrutura de servidores própria, necessitando de gerenciamento dos equipamentos e de pessoas capacitadas para operá-los. Desta forma, conforme a demanda de acessos crescia, também seria necessário a alocação de mais servidores físicos, inclusive realizar uma preparação prévia para atender a demanda de grandes picos, como no Natal, *Black Friday* ou outras datas comemorativas (ADZIC; CHATLEY, 2017). Toda essa alocação de recursos extras ficaria ociosa em dias com um menor número de acessos, mas ainda sim, geria custos, uma vez que a máquina já foi comprada, além de despesas com manutenções e espaço físico. Esse processo limitava o acesso para um grupo de empresas consolidadas e que já tinham caixa para investir em *data centers* e pessoas para trabalharem nessa operação, dificultando assim, o desenvolvimento de empresas menores.

Conforme a tecnologia evoluiu, novas soluções surgiram para atender a essa limitação. Surge então o conceito de computação em nuvem, uma rede de servidores remota que pode ser acessada pela Internet. Com essa evolução, os desenvolvedores com ideias de produtos e serviços não precisam mais investir um grande capital em *hardware* ou colaboradores para disponibilizar sua solução ao mundo. Além disso, empresas que possuem um pico de tarefas que são agendadas para serem executadas em momentos específicos do dia podem obter resultados de forma mais fácil, mesmo conforme o programa escala,

haja vista que utilizar 1000 servidores por uma hora não custa mais do que apenas um servidor alocado por 1000 horas (ARMBRUST, 2009).

2.2 Arquiteturas monolítica e microsserviços

De acordo com Pachghare (2016), na arquitetura monolítica um único código é utilizado por todos os componentes de uma aplicação. Ou seja, o sistema é baseado em um único *design* e o processo de *deploy* envolve toda a aplicação, mesmo que uma pequena alteração tenha sido realizada em apenas um simples componente do código.

Essa característica do monolítico impacta diretamente no tempo de *deploy* da aplicação, pois como todo o código está concentrado de uma forma unitária, o tempo para que tudo seja implantado será maior (VILLAMIZAR, 2015). Por outro lado, um benefício dessa estrutura é a facilidade maior no gerenciamento da infraestrutura, uma vez que o número de serviços a serem gerenciados é menor.

A arquitetura monolítica apresenta algumas outras desvantagens, como a dificuldade de realizar alterações de linguagem ou *framework*, uma vez que toda a estrutura está acoplada (NEWMAN, 2019). Além da dificuldade em escalar esse tipo de arquitetura, já que qualquer alteração que conter um problema, pode impactar diretamente em outra parte da aplicação.

Com o passar do tempo, surge a proposta de microsserviços. Nessa arquitetura, a aplicação inteira é dividida em um pequeno número de serviços, os quais são independentes uns dos outros. Cada um desses serviços é responsável por desempenhar uma parte específica do sistema (PACHGHARE, 2016).

O uso de microsserviços traz vantagens por permitir que cada equipe possa trabalhar em uma parte específica do sistema de forma independente. Além disso, reduz o tempo de *deploy*, já que cada serviço está separado, em bases de código diferentes. Além disso, os microsserviços são mais fáceis de escalar, pela redução de acoplamento. Contudo, uma das grandes desvantagens é a necessidade de gerenciar a completa infraestrutura de cada componente individualmente.

2.3 *Serverless*

Serverless provê uma plataforma para desenvolver e realizar o *deploy* de aplicações e disponibilizá-las na rede sem a necessidade de gerenciamento de nenhum tipo de infraestrutura (TAIBI, 2020). O grande benefício desse novo tipo de arquitetura é que os programadores possam focar no desenvolvimento dos sistemas e terceirizar o gerenciamento de infraestrutura para um provedor de *cloud*.

O conceito, por utilizar a palavra *serverless*, por muitas vezes é confundido com a ideia de que não há mais a existência de servidores. Contudo, essa definição está equivocada, haja vista que a proposta é que exista os servidores, mas que o gerenciamento não seja mais uma responsabilidade dos desenvolvedores.

Somado a isso, a computação *serverless* promete uma redução de custos, uma vez que as funções são executadas somente quando são invocadas e a cobrança só é a realizada pelo momento que elas estavam ativas (FAN, 2020). Funções *serverless* são orientadas a eventos, logo, são executadas apenas quando necessário. Ao terminar a execução, a instância de computação que executa a função é desativada (TAIBI, 2021).

Finalmente, outro fator característico é o fato da escalabilidade automática, em que o próprio servidor de *cloud* gerencia para atender 100 ou 100 mil requisições. Pois na computação sem servidor, o provedor é responsável por alocar dinamicamente os servidores e o código é executado em contêineres sem estado que são acionados por eventos (TAIBI, 2021).

2.3.1 *Serverless Framework*

Serverless Framework é um *framework* escrito em *JavaScript* que abstrai os aspectos técnicos de implementação e tem compatibilidade com pelo menos 5 plataformas, sendo elas: *AWS Lambda, Azure Functions, IBM OpenWhisk, Google Cloud Functions* e *SpotInst* (KRITIKOS; SKRZYPEK, 2018). Logo, *Serverless* se trata de um programa de execução via linha de comando que auxilia na construção de aplicações *serverless*, abstraindo diversos aspectos técnicos do desenvolvedor.

2.4 Provedores de computação em nuvem

Essa seção visa descrever como os principais provedores de computação em nuvem do mercado trabalham com o conceito de funções como serviço.

2.4.1 *AWS Lambda*

O *AWS Lambda* executa o código em uma infraestrutura de computação altamente disponível e realiza toda a administração dos recursos de computação. Incluindo manutenção do servidor e do sistema operacional, provisionamento de capacidade e escalabilidade automática, implantação de código e *patch* de segurança, além de monitoramento e registro do código (AWS, 2022).

Com o *AWS Lambda* é necessário apenas escrever o código desejado, sem necessidade de aprender algum novo *framework*, pois ele suporta as principais linguagens de programação ou ambientes de execução atuais, como Java, Go, PowerShell, Node.js, C#, Python e Ruby.

Além disso, outro grande benefício é que o usuário só paga enquanto aquela determinada função está sendo executada, ao contrário da arquitetura convencional. Contudo, há a presença de uma limitação de tempo, no qual é possível executar funções que demandam até 15 minutos de computação e, quando esse tempo é atingido, o processamento é encerrado.

Funções *Lambda* recebem como parâmetro duas variáveis, um evento e um contexto. O primeiro argumento contém informações do serviço que invocou aquela função que será processada. Já o segundo, o contexto, contém métodos e propriedades que fornecem informações sobre a chamada daquela função e do ambiente de execução. A função da Figura 2.2 exemplifica o processo, ela recebe esses dois parâmetros e retorna um status HTTP 200, e no corpo da mensagem, um texto simples com a sentença: *"Hello from Lambda!"*

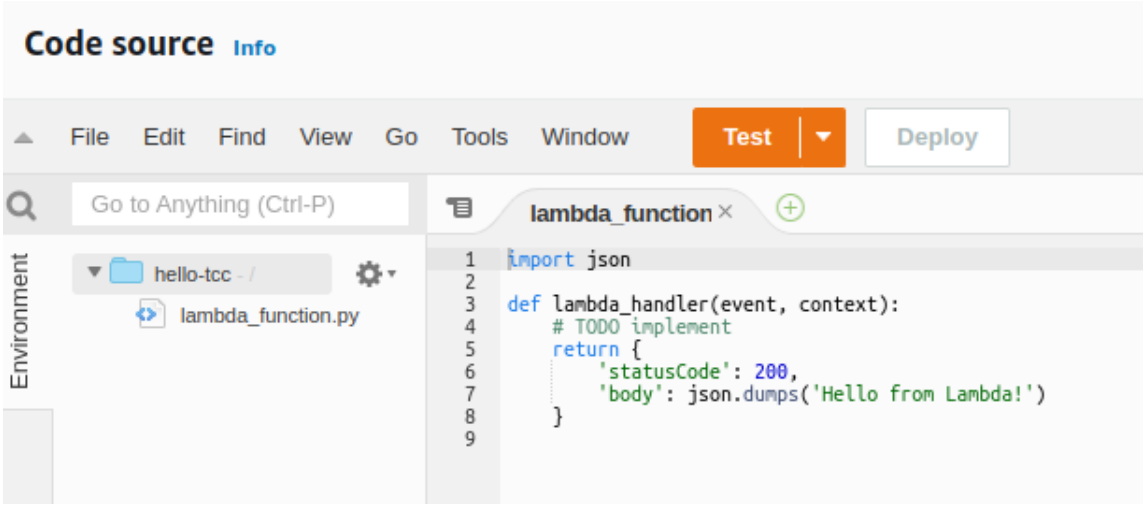

Figura 2.2: Exemplo de *AWS Lambda* - Fonte: Autoria própria.

2.4.2 *Azure Functions*

Na plataforma da *Microsoft*, a hospedagem de funções como serviço variam de acordo com o plano escolhido, sendo disponíveis o de consumo, *premium* e dedicado. No plano de consumo, a duração padrão do tempo limite de um aplicativo de funções é de 5 minutos e o máximo, 10 minutos. Nos planos *premium* e dedicado, os valores padrões são de 30 minutos e o máximo é ilimitado (MICROSOFT, 2022). As linguagens de programação suportadas são C#, JavaScript, F#, Java, PowerShell, Python e TypeScript. Sendo que todas essas podem ser executadas em ambientes Linux e Windows, com exceção do Python, que só pode ser executado em ambiente Linux.

2.4.3 Google Cloud Functions

Na plataforma da *Google*, cada função tem um tempo limite padrão de 1 minuto, chegando até o valor máximo de 9 minutos. Com relação à memória disponível para execução, por padrão esse valor é de 256 MB (CLOUD, 2022). Node.js, Python, Go, Java, .NET, Ruby e PHP são as linguagens de programação suportadas nessa plataforma.

A Tabela 2.1 apresenta um comparativo dos recursos entre as plataformas de *cloud* que disponibilizam funções *serverless*. Ela está dividida em 3 linhas que informam os valores de tempo limite de execução, a quantidade de memória disponível e as linguagens suportadas para esse tipo de aplicação. Por exemplo, percebe-se que com a *AWS*, o tempo máximo disponível é de 15 minutos, já com a *Azure*, esse tempo pode não ter um limite

predefinido, dependendo do plano escolhido.

Tabela 2.1: Comparativo de recursos entre as plataformas

	AWS	Azure	Google Cloud
Tempo limite de execução (minutos)	15	5 - ilimitado	1 - 9
Memória disponível	10GB	1,5 - 14 GB	256MB
Linguagens suportadas	Java, Go, PowerShell, Node.js, C#, Python e Ruby	C#, JavaScript, F#, Java, PowerShell, Python e TypeScript	Node.js, Python, Go, Java, .NET, Ruby e PHP

2.5 Métricas de software

Uma métrica pode ser definida como uma medida quantitativa de o quanto determinado sistema possui um certo atributo. Já uma métrica de qualidade consiste em uma função, que recebe como entrada um *software* e que produz como resultado um valor numérico que pode ser interpretado como o quanto o código passado como entrada possui um determinado atributo de qualidade que se deseja conhecer melhor (COMMITTEE, 1983). As métricas podem medir a qualidade do código, a robustez do software, a facilidade de manutenção e outros aspectos. Elas também podem ser usadas para monitorar o progresso do projeto e identificar problemas no desenvolvimento.

O tamanho de um *software* pode ser mensurado usando três métricas: comprimento, funcionalidade e complexidade. O comprimento de um *software* pode ser definido como seu tamanho físico. A medida mais comum para esse tipo de métrica é o número de linhas de código (FENTON; BIEMAN, 2014).

Algumas siglas comumente usadas para representar esse tipo de métrica baseada no tamanho do código são: *LOC* e *SLOC*. A sigla *LOC*, do inglês, *Lines of Code*, mensura a quantidade total de linhas de determinada função ou arquivo. Já a *SLOC*, *Source lines of Code*, contabiliza apenas as linhas em que há, de fato, a presença de um código, ou seja, ela desconsidera linhas em branco e aquelas que são apenas comentários (ASWINI; YAZHINI, 2017).

Outra métrica importante é a complexidade ciclomática. Ela foi proposta por

McCabe (MCCABE, 1976) e tem o intuito de indicar o número de possíveis caminhos de execução de um algoritmo. Na maior parte dos casos, uma função ou método com alta complexidade ciclomática está associado a um código que tende a ser mais difícil de testar e manter, o que indica um possível caso de refatoração (LIU, 2018).

McCabe durante seu estudo, utiliza o conceito de Grafo de Fluxo de Controle como forma de representação do fluxo de um código fonte. O grafo é composto por vértices que fazem referência aos blocos ou linhas de código fonte, e arestas que representam os caminhos do fluxo de execução (ALLEN, 1970). A Equação 2.1 é utilizada para calcular a complexidade ciclomática onde e representa o número de arestas e n o número de vértices.

$$CC = e – n + 2 \tag{2.1}$$

A Figura 2.3 traz um exemplo de código e seu grafo de fluxo correspondente. O vértice 1 representa a expressão *switch* e cada case A, B e C é representado pelos vértices 2, 3 e 4, respectivamente. O *case* C possui uma condicional, que pode gerar dois caminhos possíveis, o que é refletido no grafo com os vértices 6 e 7. De forma semelhante, o vértice 3, representado pelo caminho do *case* B, possui um outro caminho que é representado pelo vértice 5. Finalmente, o vértice 8 representa a próxima instrução após o *switch* que unifica todos os caminhos. Logo, com a presença de 8 vértices e 11 arestas, a complexidade ciclomática do exemplo em questão pode ser dada pela Equação 2.2.

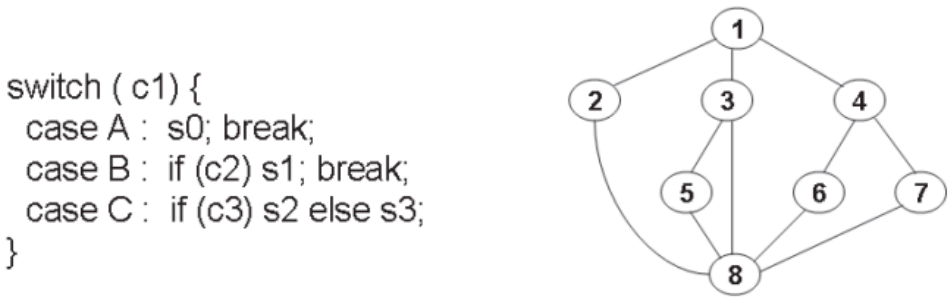

Figura 2.3: Complexidade ciclomática e Grafo de Fluxo de Controle - Fonte: (LOPEZ-HERREJON; TRUJILLO, 2008).

$$CC = 11 – 8 + 2 = 5 \tag{2.2}$$

Outras métricas que serão coletadas ao longo do desenvolvimento serão: a quantidade de classes, funções e métodos de um arquivo. O exemplo ilustrado na Listagem 2.1 ilustra o trecho de um arquivo de um dos projetos coletados ao longo da pesquisa. Neste caso, há a presença de uma classe, denominada de *PandasFile*, uma função, denominada *dask_read_excel* e mais seis métodos, sendo eles: *write, read, _write_functions, _read_dask, __init__* e *supports_s3*.

Listagem 2.1: Trecho de arquivo coletado durante a pesquisa

```
1  def dask_read_excel(path: str, **kwargs) -> dd.DataFrame:
2      delayed_df = delayed(pd.read_excel)(path)
3      return dd.from_delayed(delayed_df)
4
5
6  class PandasFile(File):
7
8      def __init__(self, path: Union[str, FilePath]) -> None:
9          super().__init__(path=path)
10
11     def read(self, lazy: bool = False, chunk_size: Union[int, str
           ] = '256MB', reset_index_if_eager: bool = True,
12              **kwargs) -> DataFrameType:
13         storage_options = {'config_kwargs': {'
               max_pool_connections': 32}, 'skip_instance_cache': True
               }
14         df = self._read_dask(chunk_size, storage_options=
               storage_options, **kwargs) if lazy \
15              else self._read_dask(chunk_size, storage_options=
                   storage_options, **kwargs).compute()
16
17         if reset_index_if_eager and not lazy:
18             df = df.reset_index(drop=True)
19
20         return df
21
22     def write(self, data: DataFrameType, repartition: bool =
           False, divisions: Union[int, str] = '64MB',
23               scheduler: str = 'threads', cb: Optional[Callback]
                   = None, **kwargs) -> None:
24         if not isinstance(data, (pd.DataFrame, dd.DataFrame)):
25             raise PandasFileError(message=f'Data passed to write
                   method isn\'t a pandas or dask DataFrame. '
26                                           f'Please use a
                                               DataFrame for {self.
                                               DASK_SUPPORTED_FORMATS
                                               }')
27
28         if isinstance(data, pd.DataFrame):
29             data = dd.from_pandas(data, npartitions=1, sort=False
```

```
            )
30
31          if repartition:
32              data = data.repartition(divisions=divisions)
33
34          if cb is None:
35              self._write_functions(data, self.path, scheduler)
36          else:
37              with cb:
38                  self._write_functions(data, self.path, scheduler)
39
40      @staticmethod
41      def _write_functions(df: DataFrameType, path: FilePath,
           scheduler: str) -> None:
42          if path.file_type in ('xls', 'xlsx'):
43              df.compute(scheduler=scheduler).to_excel(path),
44          elif path.file_type in ('csv', 'dat', 'data'):
45              df.to_csv(path, compute_kwargs={'scheduler':
                   scheduler}, single_file=True)
46          elif path.file_type in ('parquet',):
47              df.to_parquet(path, compute_kwargs={'scheduler':
                   scheduler})
48          else:
49              raise ValueError(f'Failed writing dataframe to file.
                   File type {path.file_type} not known.')
50
51      def _read_dask(self, chunk_size: Union[int, str], **kwargs)
           -> dd.DataFrame:
52          if self.path.file_type not in self.DASK_SUPPORTED_FORMATS
               :
53              raise PandasFileError(f'File type {self.path.
                   file_type} not supported for lazy loading.')
54
55          if self.path.file_type in ('xls', 'xlsx'):
56              return self.DASK_LOADING_METHODS[self.path.file_type
                   ](self.path, **kwargs)
57          elif self.path.file_type in ('parquet',):
58              return self.DASK_LOADING_METHODS[self.path.file_type
                   ](self.path, chunksize=chunk_size, **kwargs)
59
60          return self.DASK_LOADING_METHODS[self.path.file_type](
               self.path, blocksize=chunk_size, **kwargs)
61
62      @staticmethod
63      def supports_s3():
64          return True
```

2.5.1 Ferramentas para extração de métricas de software

Esta subseção visa detalhar sobre duas ferramentas que auxiliam na extração automatizada das métricas mencionadas nesta Seção. A Subseção 2.5.1 discorre sobre a *Radon*, uma ferramenta utilizada para coleta de métricas em projetos *Python*. e a Subseção 2.5.1, descreve a biblioteca *Plato*, utilizada para coletar métricas em projetos em *JavaScript*.

Radon

Radon é uma ferramenta *Python* que permite calcular algumas métricas de código. Ela pode ser utilizada tanto por linha de comando, quanto por meio da sua *API*. As métricas suportadas pela plataforma são: o índice de manutenibilidade, métricas de *Halstead*, complexidade ciclomática e algumas métricas estáticas como a quantidade de linhas em branco, a quantidade que são apenas comentadas, quantidade de linhas totais do arquivo, etc.

Plato

Plato é uma ferramenta que permite a visualização de código-fonte *JavaScript*, a análise estática, o cálculo de complexidade, entre outras informações. Ela gera, para cada arquivo do diretório que está sendo analisado, uma pasta que contém 5 arquivos, um deles possui o nome *report* e é disponibilizado tanto em formato *JavaScript* quanto em *JSON*, e ele é responsável por armazenar as métricas do momento atual do arquivo. Outro arquivo gerado é um *html* que pode ser visualizado diretamente no *browser* e que traz diversos gráficos a respeito dos dados calculados. Além disso, Plato constrói um arquivo chamado de *report.history*, que é responsável por armazenar todo o histórico das métricas daquele arquivo específico e também está disponível nos formatos *JavaScript* e *JSON*.

2.6 Mineração de repositórios

Repositórios de *software* são artefatos produzidos e armazenados durante a evolução de um software. Eles armazenam uma vasta quantidade de informação sobre o processo de construção daquele sistema como: dados sobre as mudanças individuais e informações

sobre cada alteração, como quem alterou, o motivo e quando a modificação foi feita (KAGDI, 2007).

A mineração de repositórios é uma técnica usada para analisar e extrair dados de repositórios de código-fonte. Essa análise pode ser usada para compreender melhor o código, bem como para encontrar padrões e tendências no desenvolvimento do software. A mineração também pode ser usada para monitorar a atividade dos desenvolvedores e entender como o código é alterado ao longo do tempo.

A plataforma *Github* é uma enorme base de repositórios, disponível para hospedagem de código. Ela é baseada no sistema de controle de versionamento, Git, e permite a criação de repositórios públicos ou privados. Diante disso, pesquisadores interessados em realizar mineração de repositórios, frequentemente utilizam esse acervo de repositórios públicos disponibilizados na plataforma como base de pesquisa (YAMAMOTO, 2020).

As pesquisas relacionadas à mineração de repositórios têm como foco dois aspectos principais: a criação de técnicas para automatizar e tornar o processo de extração de informações melhor e mais fácil, bem como, descobrir e validar novas técnicas para minerar informações importantes desses repositórios (HASSAN, 2008) .

2.7 Trabalhos Relacionados

Esta seção introduz alguns trabalhos que discutem temas relacionados à adoção da computação *serverless*. Para encontrar esses estudos, foi adotada a metodologia híbrida sugerida por Mourão, Pimentel, Murta, Kalinowski, Mendes e Wohlin (2020), na qual é utilizada uma base de busca digital, acrescida da técnica de *Snowballing*. Essa técnica refere-se ao uso da lista de referências ou citações de um artigo para identificar trabalhos adicionais relacionados (WOHLIN, 2014). A estratégia híbrida se iniciou com a montagem de uma *string* de busca para ser utilizada na base de pesquisa da *Scopus*[1]. A elaboração da *string* foi baseada em palavras-chave que eram relacionadas ao tema de pesquisa, sendo composta por quatro parâmetros: população, intervenção, comparação e saída, sendo esses detalhados abaixo:

População: Pesquisas relacionadas ao tema *serverless* ou funções como serviço.

[1]https://www.scopus.com

Intervenção: impacto na arquitetura ou nas métricas de software.

Comparação: com a arquitetura monolítica.

Saída: técnicas, abordagens, métodos, metodologias, ferramentas, ou processos de identificação de métricas em aplicações serverless.

Diante disso, o resultado obtido foi:

População: ("serverless"OR "AWS Lambda"OR "FaaS"OR "Function-as-a-Service")

Intervenção: ("architectural"OR "impact"OR "metrics"OR "performance")

Comparação: ("client-server"OR "monolithic")

Saída: (characterization OR approach OR method OR methodology OR procedure OR definition OR mechanism OR experience OR findings OR research OR study OR technique OR knowledge OR tool OR support)

Tendo como resultado final a seguinte string:

("serverless"OR "AWS Lambda"OR "FaaS"OR "Function-as-a-Service") AND ("architectural"OR "impact"OR "metrics"OR "performance") AND ("client-server"OR "monolithic") AND (characterization OR approach OR method OR methodology OR procedure OR definition OR mechanism OR experience OR findings OR research OR study OR technique OR knowledge OR tool OR support)

O resultado dessa pesquisa retornou 22 artigos, dos quais foram lidos o título e o resumo, a fim de identificar quais deles se encaixavam no alvo da pesquisa. A partir disso, foram escolhidos 3 resultados dessa busca. Os demais trabalhos foram obtidos utilizando a técnica de *Snowballing*. A seguir, os próximos parágrafos detalham as principais conclusões obtidas em cada um desses estudos selecionados.

Fan, Jindal e Gerndt (2020) construíram uma análise de uma aplicação *Web*, que consistia em um portal de gerenciamento do tempo empregado de um funcionário em um determinado projeto, que foi refatorada tanto utilizando microsserviços, quanto com *serverless*. Essa análise foi realizada sob os aspectos de escalabilidade, confiabilidade, custos e latência. O *frontend* da aplicação e tipo de banco de dados utilizados permaneceram iguais, mas o *backend* foi refatorado. Como conclusões, foi abordado que as aplicações *serverless* sofrem com o problema de *cold-start*, microsserviços sofrem com

o problema de balanceamento de carga e redistribuição de tráfego, mas superam o desempenho em requisições pequenas e repetitivas. Porém, *serverless* é mais ágil em relação à escalabilidade.

Adzic e Chatley (2017) trouxeram dois estudos de caso de empresas que refatoraram suas aplicações e transicionaram para uma arquitetura *serverless*. A primeira, *MindMup*, observou que um ano depois da transição, mesmo que o número de usuários ativos tenha aumentado em mais de 50%, os custos de hospedagem caíram um pouco menos de 50%, mesmo com novos serviços que foram adicionados. Já a segunda empresa, *Yubl*, tinha um sistema monolítico provendo o *backend* de um aplicativo de uma rede social. A estimativa após a mudança para utilização de funções *Lambda* é que proporcionou uma redução de custo operacional superior a 95% para uma quantidade comparável de recurso computacional. Além disso, a empresa alegou que antes da migração, o time de engenharia conseguia entregar cerca de 4 a 6 *releases* em produção por mês. Após uma alteração na equipe e a migração realizada, o time passou a entregar consistentemente mais de 80 *releases* por mês, com o mesmo tamanho de equipe. Alguns dos apontamentos realizados concluíram que a quebra do sistema monolítico em várias funções *Lambda*, significou que foi possível dividir melhor o time em trabalhar em mais tarefas em paralelo e realizar o *deploy* de cada funcionalidade separadamente, com menos conflitos de código.

Toader, Uta, Musaafir e Iosup (2019) propõem uma nova estrutura de processamento grafo chamada *Graphless*, projetada para ser executada em ambientes *serverless*. *Graphless* usa um sistema distribuído de funções *Lambda* para processar grafos em paralelo. O processamento de grafos é uma tarefa computacionalmente intensiva que é frequentemente usada na análise de dados. No entanto, as atuais estruturas de processamento grafo não são adequadas para ambientes *serverless* devido à dependência de armazenamento centralizado e recursos de computação. Os autores descobriram que o *Graphless* supera as estruturas de processamento grafo existentes em até 2 ordens de magnitude, ao mesmo tempo em que fornece precisão comparável. Além disso, eles descobriram que o *Graphless* pode ser dimensionado de forma dinâmica para atender às demandas de processamento súbitas que ocorrem em curtos espaços de tempo. Segundo as observações feitas, isso poderia levar a um processamento grafo mais escalável e eficiente, bem como a custos

mais baixos. O artigo também descreve alguns dos desafios associados ao processamento de grafo sem servidor, como gerenciar o estado e lidar com a localidade dos dados.

Bajaj, Bharti, Goel e Gupta (2020) propuseram realizar um estudo sobre como uma aplicação monolítica poderia ser refatorada parcialmente, identificando casos de uso que poderiam se encaixar como microsserviços ou *serverless*.Como prova de conceito, a proposta foi aplicada em um estudo de caso usando uma aplicação *Web* de *feedbacks* para professores. Como principal contribuição desse artigo, pode ser elencada a construção de uma nova estratégia para refatorações parciais de aplicações monolíticas e a demonstração de como esse conceito pode ser usado em uma aplicação prática.

Goli, Hajihassani, Khazaei, Ardakanian, Rashidi e Dauphinee (2020) desenvolveram um estudo de caso de uma migração de uma aplicação monolítica de serviços financeiros, com o intuito de implantar uma arquitetura *serverless* distribuída e performática. Os principais problemas encontrados no sistema original eram velocidade e escalabilidade. Eles concluíram que a implementação *serverless* forneceu aumento de velocidade e melhoria de desempenho comparada à monolítica e sem provocar mudanças drásticas no design do software.

2.8 Considerações finais

Os conceitos apresentados nessa seção são fundamentais para a compreensão do estudo proposto. Foi coberta uma explicação sobre as principais características das arquiteturas de *software* monolíticas e de microsserviços e como elas se diferenciam da proposta *serverless*. Somado a isso, a descrição sobre computação em nuvem e como cada provedor trabalha com esse paradigma visaram trazer a tona as semelhanças e diferenças entre eles, cobrindo pontos como memória, tempo de processamento e linguagens suportadas. Esse capítulo também apresentou os conceitos de métricas de *software* e mineração de repositórios através da introdução de algumas técnicas que serão utilizadas para analisar bases de código e inferir qual o impacto da utilização da arquitetura *serverless* nesses projetos, além de apresentar alguns trabalhos relacionados que contribuem para ampliar a percepção sobre o assunto.

3 Coleta de métricas dos projetos

Este capítulo visa discorrer acerca da estratégia adotada para extrair as métricas de *software* utilizadas na comparação dos projetos que utilizam as arquiteturas *serverless* e monolítica. Ele está dividido em dois momentos principais: a Seção 3.1 que traz uma visão global do processo que foi implementado e relata quais tecnologias que foram adotadas, bem como quais informações serão coletadas. Já a Seção 3.2 é responsável por detalhar o processo prático realizado para obtenção dos dados.

3.1 Abordagem

Para realizar um estudo comparativo dos projetos *open source* foi desenvolvida uma estratégia para obtenção de algumas métricas de *software* que pudessem ser posteriormente comparadas para analisar como os dados se comportavam de acordo com as características do projeto.

A estratégia recebe como entrada um caminho de um repositório de um projeto *serverless* ou monolítico escrito nas linguagens de programação *JavaScript* ou *Python*. A abordagem consiste em navegar no histórico do projetos, utilizando os recursos de versionamento dos sistemas de controle de versões, e coletar métricas daquela versão. Para cada arquivo dentro do diretório serão coletadas quatro métricas, a saber: a quantidade de métodos, quantidade de classes e quantidade de funções/métodos presentes, bem como a quantidade total de linhas que ele contém. Além disso, para cada função ou método existentes são coletadas as métricas de complexidade ciclomática, a quantidade de linhas totais e o nome que lhe foi atribuído.

A fim de automatizar o processo de extração e torná-lo replicável, foi elaborado um *script* na linguagem *Python*, bem como construído um banco de dados utilizando o sistema de gerenciamento *MySQL* para armazenamento das informações e futura exploração dos dados obtidos. O *script* desenvolvido basicamente se divide em dois momentos, o de extração de métricas para *JavaScript* e o para *Python*, pois ambos utilizam uma biblioteca

diferente para extração dos dados. Diante disso, a Seção 3.2.1 discorre sobre o processo de implementação para *Python* e a Seção 3.2.2 relata para *JavaScript*.

3.2 Implementação

Esta seção visa detalhar o processo de implementação prática do *script* desenvolvido. As tabelas do banco de dados e suas relações foram pensadas de acordo com as métricas desejadas, chegando ao resultado que pode ser visualizado na Figura 3.1. A figura ilustra as 4 tabelas criadas, na qual a tabela *file* se relaciona com todas as demais, possuindo uma relação de um para muitos com a *project* e a *commit*, representando que um projeto pode conter muitos arquivos, mas cada arquivo está associado a um projeto específico. Além disso, em um *commit*, pode haver a presença de vários arquivos, mas cada arquivo também está associado a algum *commit* específico. Por fim, a tabela *function_method* também apresenta a relação de um para muitos, haja vista que em um arquivo pode haver a presença de várias funções e métodos, mas cada um deles estará associado a um arquivo.

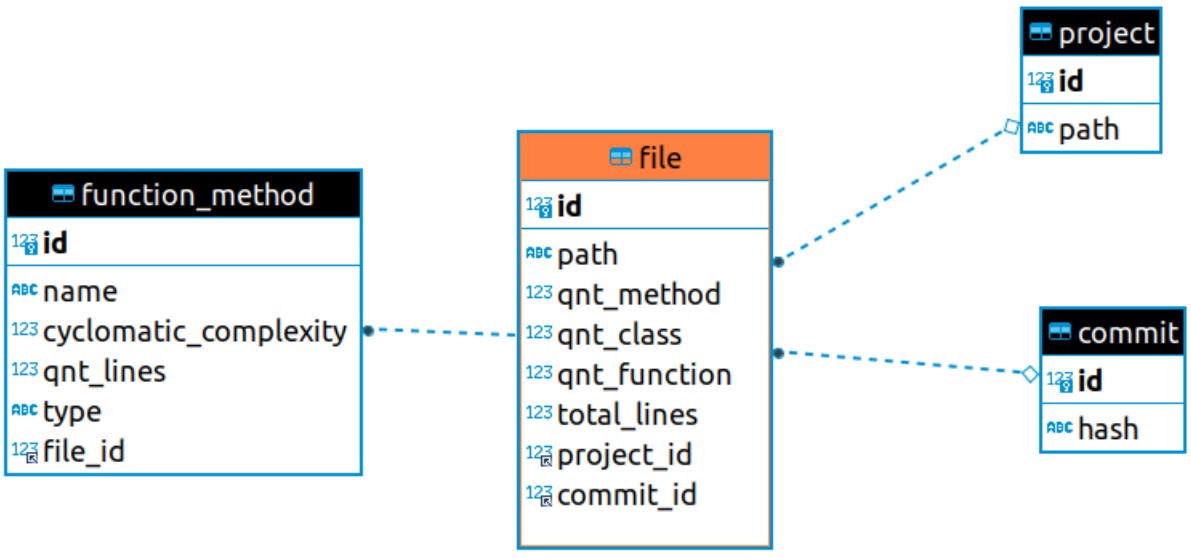

Figura 3.1: Diagrama entidade relacionamento - Fonte: Autoria própria.

O processo se inicia pela tabela *project*, cuja função é armazenar apenas o caminho do diretório do projeto que será analisado. Posteriormente, será obtida a lista de todos os

hashes dos *commits* daquele repositório utilizando o comando apresentado na Listagem 3.1.

Listagem 3.1: Exemplo de customização do *git log*.

```
1 $ git log --pretty='format:%h'
```

O comando *git log* permite visualizar o histórico de alterações de determinado projeto ao longo do tempo. Por padrão, ele retornará quatro informações sobre cada *commit*: o *hash* (*Secure Hash Algorithm*), o autor do *commit*, a data e a descrição da alteração realizada naquele ponto do projeto.

Contudo, é possível customizar o retorno do comando, procurando retornar a informação que será mais relevante para a situação. Com a junção das *flags* *–pretty=format*, é possível indicar na sequência quais são os dados desejados. A tabela 3.1 mostra possíveis *flags* que podem ser utilizadas juntamente com o *–pretty=format* e o seu respectivo *output*. Neste trabalho, como o objetivo é iterar por todo o histórico do projeto, foi utilizada a *flag* *%h*, que é responsável por retornar todos *hashes* abreviados dos *commits*. A Listagem 3.2 traz um exemplo do resultado.

Tabela 3.1: Opções possíveis para git log –pretty=format

Opção	Descrição do output
%H	*hash* do *commit*
%h	*hash* abreviado do *commit*
%T	*hash* da árvore
%t	*hash* abreviado da árvore
%P	*hash* dos pais
%p	*hash* abreviado dos pais
%s	mensagem

Listagem 3.2: Exemplo de *output* para o comando *git log*

```
1  git log --pretty='format:%h'
2  2369e24
3  a463e0f
4  4867b0b
5  ea78627
6  948962b
7  d22fdb4
8  527bdc9
9  3450a0f
10 3cac73e
11 b529a76
12 c998cbc
13 b67815c
14 4538014
15 895bea0
16 b5ddea9
17 c3f6819
18 4e333f0
19 13bdb6f
20 dc821e8
21 bd7e413
22 a209b50
23 90674cf
24 1e1a273
25 00292f6
26 93ca4e3
27 c9a0581
28 46f7d86
```

De posse dessa lista, o próximo passo é realizar uma iteração por cada *commit*, utilizando o comando *git checkout*, visando extrair as métricas para aquele momento específico do código e observar como ele foi evoluindo ao longo do tempo. Diante disso, o *hash* do *commit* que está sendo analisado no momento é armazenado na tabela *commit*. A partir disso, o objetivo será obter os dados de cada arquivo, função ou método. Esta análise é dependente de linguagem de programação e a explicação é dividida em duas subseções: 3.2.1 e 3.2.2.

3.2.1 Métricas para *Python*

Para auxiliar na extração de métricas nos projetos desenvolvidos em *Python*, foi utilizada a biblioteca *Radon*. *Radon* é uma ferramenta que computa diversas métricas de um código fonte.

O primeiro comando utilizado foi o apresentado na Listagem 3.3.

Listagem 3.3: Comando utilizado para calcular a complexidade ciclomática

```
1 radon cc {folder} -sj
```

Este comando analisa o código fonte do diretório que foi passado como argumento e computa a complexidade ciclomática das funções e métodos presentes. Após o traço, há a presença de duas *flags*, s e j. A *flag* s é responsável por mostrar a pontuação da complexidade encontrada, pois, por padrão, ela não é retornada. A letra j é responsável por converter o resultado no formato *JSON*, facilitando assim a manipulação dos dados.

Para cada arquivo dentro desse diretório que foi passado, é retornado um conjunto de informações sobre as funções e classes presentes, bem como os métodos dentro de cada classe. Entre essas informações retornadas, há a presença da linha de início e de fim do dado em questão, sua complexidade, bem como o tipo, podendo ser um dos três a seguir: *function*, *method* ou *class*.

Com a junção destes dados disponíveis, foi realizado o somatório da quantidade desses três tipos de informações em cada arquivo, preenchendo assim as colunas *qnt_method*, *qnt_class*, *qnt_function* na tabela *file* do banco de dados. Além disso, foi realizada a análise individual de cada função ou método, preenchendo a coluna de *cyclomatic_complexity* com o valor automaticamente retornado e a *qnt_lines* foi obtida por meio da subtração dos campos *endline* e *lineno*, que representam linha de fim e de início, respectivamente. A coluna *name* diz respeito ao nome da função ou método em questão e a *type* indica se o dado se trata de uma função ou de um método.

O segundo comando utilizado foi o apresentado na Listagem 3.4.

Listagem 3.4: Comando utilizado para se obter métricas estáticas do arquivo

```
1 radon raw {folder} -j
```

Ele é responsável por calcular algumas métricas estáticas do programa. Ao executar o comando, ele traz para cada arquivo, as métricas apresentadas na Listagem 3.5.

Listagem 3.5: *Output* das métricas estáticas

```
{
    "loc": 78,
    "lloc": 20,
    "sloc": 58,
    "comments": 7,
    "multi": 4,
    "blank": 13,
    "single_comments": 3
  }
```

O valor retornado por *loc*, que representa o número total de linhas de código, foi utilizado para definir a coluna *total_lines* na tabela *file*.

3.2.2 Métricas para *JavaScript*

Para obter as métricas em projetos *JavaScript*, foi utilizada a biblioteca *Plato*. O processo é similar ao anterior, porém se difere em alguns pontos. O primeiro comando utilizado nessa fase foi o ilustrado na Listagem 3.6:

Listagem 3.6: Comando utilizado para se obter as métricas em *JavaScript*

```
plato -r -d {folder}/result {folder}
```

A *flag* -r no comando o diz para considerar recursivamente todas as pastas do diretório que foi passado como raiz, já a *flag* -d é responsável por identificar qual será o caminho do diretório que guardará os resultados das métricas encontradas, o que no caso, será uma nova pasta chamada de *result*, que será criada dentro do diretório raiz. Após isso, há apenas o caminho do diretório que será efetivamente analisado.

O próximo comando utilizado foi o representado na Listagem 3.7.

Listagem 3.7: Comando utilizado para se obter a lista de arquivos gerada

```
cd {folder}/result/files; ls;
```

O comando inicial nesta subseção criou uma pasta chamada *result* e dentro dela ele cria outra pasta chamada *files*, que contém um conjunto de arquivos com diversos dados sobre cada arquivo do diretório raiz. Sendo assim, o que o comando 3.7 faz é obter o nome de todos os arquivos presentes na pasta *files*.

De posse dessa lista, é realizada uma iteração por cada um dos itens e iniciada

a leitura do arquivo *report.json*, que é o responsável por armazenar as métricas de cada arquivo no formato *JSON*. Este arquivo contempla um objeto chamado *complexity* que possui diversas informações relacionadas às métricas desejadas. Ele possui 2 *arrays* denominados de *classes* e *methods*, que armazenam informações sobre cada classe ou método presentes no arquivo, respectivamente. Um ponto importante é que como esses *arrays* trazem informações individuais sobre cada função, o tamanho de cada um deles representa a quantidade total de classes ou métodos de um dado arquivo e essa informação foi utilizada para preencher os campos *qnt_method*, *qnt_class* e *qnt_function* de cada arquivo na tabela *file*. Esse objeto também contém as propriedades *lineEnd* e *lineStart*, que representam as linhas de fim e início do arquivo e foram utilizadas para calcular a variável *total_lines*.

A fim de obter os dados sobre cada função ou método, foi realizada uma iteração sobre o *array methods* e foi utilizada a propriedade *cyclomatic* para representar a variável de complexidade ciclomática, a subtração de *lineEnd* e *lineStart* do próprio método para se referir a *qnt_lines* na tabela *function_method* e a propriedade *name* para corresponder ao nome do método. A Listagem 3.8 ilustra um exemplo de uma parte do arquivo *report.json* na qual essas informações estão presentes.

Listagem 3.8: Exemplo de parte do arquivo *report.json*

```
1
2     "methods": [
3       {
4         "cyclomatic": 5,
5         "cyclomaticDensity": 38.462,
6         "halstead": {
7           "bugs": 0.146,
8           "difficulty": 10.214,
9           "effort": 4487.332,
10          "length": 82,
11          "time": 249.296,
12          "vocabulary": 41,
13          "volume": 439.319,
14          "operands": {
15            "distinct": 28,
16            "total": 44,
17            "identifiers": ["__stripped__"]
18          },
19          "operators": {
20            "distinct": 13,
21            "total": 38,
22            "identifiers": ["__stripped__"]
23          }
24        },
25        "params": 1,
26        "sloc": { "logical": 13, "physical": 54 },
27        "errors": [],
28        "lineEnd": 61,
29        "lineStart": 8,
30        "name": "<anonymous>"
31      }
32    ]
```

4 Resultados

Este capítulo tem o intuito de apresentar e discutir os resultados encontrados durante os experimentos realizados. Para isso está organizado em seis seções. A Seção 4.1 introduz as questões de pesquisa que guiarão o estudo. A Seção 4.2 explica o processo utilizado para encontrar os repositórios de projetos escritos em *JavaScript* e *Python* que são monolíticos e *serverless*. A Seção 4.3 apresenta as informações encontradas durante os experimentos, já a Seção 4.4 discute como essas informações ajudam a responder as questões de pesquisa e Seção 4.5 apresenta as possíveis ameaças à validade do estudo.

4.1 Questões de pesquisa

Como descrito anteriormente, o objetivo deste trabalho é o de entender se a adoção da computação *serverless* traz impactos na construção de um *software* quando comparado com a arquitetura monolítica. Para guiar este experimento, são propostas três Questões de Pesquisa (QP) descritas a seguir:

QP1 - A quantidade de arquivos em projetos com arquitetura *serverless* é maior do que a monolítica?

Este questionamento visa entender se a adoção da computação *serverless* aumenta o número de arquivos do projeto. Para respondê-la, será feita uma análise da quantidade média de arquivos durante todo o histórico de versionamento dos projetos selecionados. Estes projetos são apresentados na Seção 4.2.

QP2 - Como os aspectos físicos e sintáticos dos arquivos se comportam em cada uma das duas arquiteturas?

A resposta para esta pergunta medirá em qual das arquiteturas o número de classes, funções e métodos dos arquivos, considerados aqui como os aspectos sintáticos, é superior. Uma análise complementar é feita considerando o aspecto físico que é medido pela quantidade total de linhas de código em cada arquivo, verificando se esse valor tende a crescer ou reduzir dependendo de cada arquitetura.

QP3 - A complexidade ciclomática e o número de linhas das funções e métodos entre as duas arquiteturas apresenta diferenças?

Este questionamento tem o intuito de verificar se a adoção de alguma dessas estruturas tende a piorar a capacidade de manutenção do código, já que funções ou métodos maiores e com complexidade superior tendem a ser mais difíceis de manter. Para conseguir esse resultado, será mensurado para cada função e método de todos os arquivos, sua complexidade ciclomática e o número de linhas de código.

4.2 Busca dos repositórios

Com o intuito de mensurar qual o impacto da adoção da arquitetura *serverless* nas aplicações, uma busca por amostras de projetos de *software* monolíticos e *serverless* foi desenvolvida, a fim de metrificar os resultados de acordo com a abordagem sugerida no capítulo 3. No processo de pesquisa, os principais critérios utilizados foram:

- considerar repositórios que foram escritos apenas em *JavaScript* ou *Python*;
- considerar apenas repositórios com mais de 30 *commits*;
- foram excluídos da análise os repositórios que se tratavam apenas de *frameworks* ou bibliotecas com o propósito de auxiliar na construção de aplicações *serverless*;
- foram excluídos aqueles que representavam utilidades para facilitar o desenvolvimento, projetos de ensino ou estudo, além daqueles que foram arquivados;
- remover todos que continham apenas uma função *Lambda*, pois não teriam muito o que proporcionar para análise;

Um dos meios utilizados para realizar a busca por repositórios no *GitHub* foi o uso do *GitHub Advanced Search*. A busca avançada permite uma pesquisa mais direcionada em repositórios de código-fonte públicos baseada em critérios específicos definidos pelo usuário. Por exemplo, é possível buscar por repositórios que contenham um nome específico, que sejam de um determinado usuário ou que sejam desenvolvidos com uma linguagem de programação desejada. Além disso, é possível filtrar os repositórios por quantidade de estrelas, *forks*, tamanho, licença do projeto, entre outras.

Outra possibilidade na busca avançada do *GitHub* é a filtragem baseada no código, permitindo a busca por uma extensão, nome ou tamanho de arquivo, que estejam contidos dentro de um repositório.

Outra funcionalidade interessante que o *Github* dispõe é a de tópicos. Um tópico é uma forma de classificação de um repositório, sendo que eles podem expressar a finalidade pretendida do código em questão, a área de assunto, a comunidade ou a linguagem.

As subseções 4.2.1 e 4.2.2 tem o intuito de explicar como essas funcionalidades foram utilizadas para encontrar repositórios de projetos *serverless* e monolíticos.

4.2.1 Repositórios *Serverless*

Conforme descrito na Seção 2.3.1, *Serverless* é um *framework* bastante utilizado para auxiliar na construção de aplicações *serverless*. Para descrever todos os serviços desejados ao *framework*, *Serverless* trabalha normalmente com a construção de um arquivo chamado serverless.yml, no qual será definido todas as funções, eventos e recursos do provedor de *cloud* necessários para realizar o *deploy* daquele sistema.

A Figura 4.1 mostra um exemplo da estrutura desse arquivo, no qual na *tag functions* são definidas todas as funções *Lambda*, com o nome e o respectivo evento que as invoca. Ou seja, neste exemplo há a presença de duas funções, a *usersCreate*, que é chamada pelo evento no *endpoint* /users/create de POST, e a *usersDelete*, chamada por meio de um evento no *endpoint* /users/delete pelo método DELETE. Fazendo uso dessa informação e utilizando a ferramenta de busca avançada do *Github*, uma das estratégias adotadas para encontrar esse tipo de repositório foi a busca por códigos que continham filename:serverless.yml.

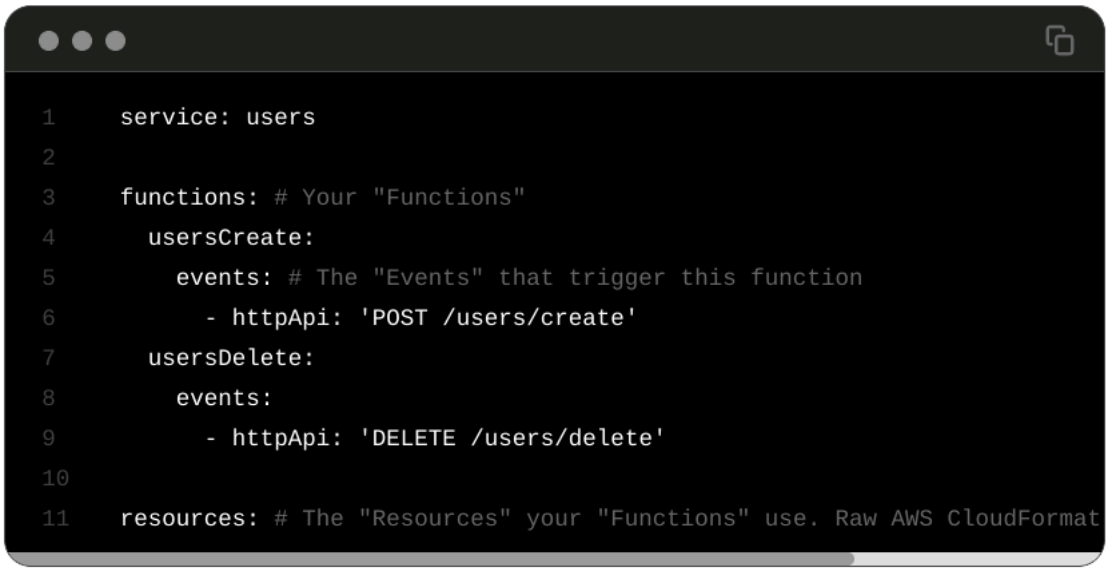

Figura 4.1: Exemplo de arquivo serverless.yml - Fonte: (SERVERLESS, 2022).

A busca consistiu em filtrar pelo nome do arquivo e também adicionar outra restrição de tamanho, restringindo que esse arquivo deveria ser maior do que 10KB, a fim de encontrar projetos maiores e que conteriam mais dados para serem analisados. Um exemplo da *string* de busca utilizada pode ser visualizado na Listagem 4.1. Utilizando essa abordagem, foi possível selecionar 4 projetos escritos em *JavaScript* e 2 projetos desenvolvidos em *Python*.

Listagem 4.1: Exemplo da *string* de busca utilizada

```
1 filename:serverless.yml size:>10000
```

Outra forma complementar foi a pesquisa realizada em 02/12/2022 utilizando o tópico *Serverless* e filtrando pela linguagem de programação *Python*. A busca retornou no total 1297 repositórios públicos e, analisando os primeiros resultados, foi possível selecionar uma lista de 3 projetos que se encaixavam nos critérios predefinidos.

O mesmo tópico *Serverless* foi utilizado para encontrar mais projetos escritos em *JavaScript*, filtrando agora por essa linguagem. Dos 3885 repositórios retornados, foram analisados os primeiros resultados e, após utilizar os critérios de seleção, 1 projeto foi selecionado. A Tabela 4.1 apresenta a lista dos 10 projetos selecionados nesta etapa para o experimento, com o nome do repositório, data do último *commit*, quantidade de *commits* e a linguagem do projeto. Uma observação é que, alguns deles não eram 100% escritos na linguagem de programação desejada, contendo também outras, dessa forma, nesta coluna

estão dispostas a porcentagem referente à linguagem predominante.

Tabela 4.1: Lista dos repositórios *serverless* escolhidos

Nome	Data do último commit	N° de commits	Linguagem
chabelitas-back[2]	28/09/2022	105	100% *JavaScript*
ruuvi-network-serverless[3]	13/06/2022	173	99% *JavaScript*
serverless-survey-forms[4]	13/06/2022	1210	87% *JavaScript*
moonMail[5]	13/06/2022	875	97% *JavaScript*
claudia-bot-builder[6]	17/02/2021	525	100% *JavaScript*
Kindergarten-Backend[7]	03/09/2022	348	100% *Python*
aws-serverless-ecommerce-platform[8]	27/03/2022	625	75% *Python*
aws-serverless-shopping-cart[9]	28/03/2022	92	42% *Python*
streamalert[10]	20/07/2022	1904	94% *Python*
domain-protect[11]	18/10/2022	479	68% *Python*

4.2.2 Repositórios Monolíticos

O processo de busca por repositórios de projetos monolíticos se iniciou com a utilização do tópico *monolith* do *GitHub*. Utilizando o filtro de linguagem de programação e filtrando por *Python*, a busca retornou 16 repositórios públicos. Algumas das opções retornadas se tratavam apenas de auxiliares para realizar a migração de aplicações monolíticas e não foram consideradas. Outro resultado se tratava de uma ferramenta para solucionar o jogo chamado de *Treasure Hunter! Monolith*, que não era relacionado à pesquisa, então também não foi considerado. Depois disso, analisando a descrição dos projetos que restaram, 3 repositórios foram selecionados. Fazendo uso deste mesmo tópico, só que filtrando pela linguagem de programação *JavaScript*, a busca retornou 13 repositórios e foi possível selecionar 1 projeto que atendia aos critérios.

Outra variação de tópico utilizada para pesquisa foi o *monolith-architecture*. Filtrando por *JavaScript*, a busca retornou 7 repositórios e para *Python*, apenas 3. Analisando os resultados, foi possível escolher 1 projeto *JavaScript* e 2 projetos *Python*. Para complementar os resultados, outro filtro de busca utilizado no *Github* foi o apresentado na Listagem 4.2. A busca retornou 26 resultados e analisando a descrição dos projetos e verificando se se enquadravam nos critérios de seleção, foram selecionados mais 3 projetos.

Listagem 4.2: Exemplo da *string* de busca utilizada para os monolíticos.

```
1 monolithic architecture language:JavaScript
```

A Tabela 4.2 apresenta a relação de todos os repositórios selecionados, sendo dividida em 4 colunas: nome do projeto, data do último *commit*, quantidade *commits* e a linguagem majoritária do repositório, incluindo a respectiva porcentagem.

Tabela 4.2: Lista dos repositórios monolíticos escolhidos

Nome	Data do último commit	N° de commits	Linguagem
TheChaotic[12]	06/12/2021	99	53% *JavaScript*
YelpCamp[13]	25/04/2022	189	37% *JavaScript*
Auction website monolith[14]	26/02/2021	85	94% *JavaScript*
Bulletproof vanilla nodejs[15]	23/10/2022	33	98% *JavaScript*
Nodejs shopping-cart[16]	25/07/2022	82	63% *JavaScript*
GoOutSafe[17]	23/11/2020	300	79% *Python*
Finance-Full-Stack-Web-App[18]	12/05/2022	98	83% *Python*
Highlowbackend[19]	20/01/2021	606	97% *Python*
Monolith-filemanage[20]	10/12/2021	138	100% *Python*
Django-data-browser[21]	21/08/2022	922	91% *Python*

4.3 Análise dos experimentos

Esta seção visa trazer os dados referentes aos experimentos práticos realizados, procurando responder as questões de pesquisa levantadas na Seção 4.1. Ela está organizada em 3 subseções que pretendem analisar os resultados nos seguintes cenários: projetos *serverless* escritos em *Python*, projetos *serverless* em *JavaScript*, projetos monolíticos escritos em *Python* e projetos monolíticos em *JavaScript*. Para isso, os dados estão dispostos em tabelas que estão organizadas em 6 colunas da seguinte maneira: Projeto, que apresenta o nome do projeto; Média, Mínimo e Máximo, que apresentam os valores médio, mínimo e máximo de arquivos por projeto; N° final, que apresenta o valor obtido da métrica na última versão analisada; e Tipo que apresenta a classificação do projeto em *serverless* ou monolítico.

4.3.1 A quantidade de arquivos em projetos com arquitetura *serverless* é maior do que a monolítica?

Com o intuito de entender como se comporta o crescimento do número de arquivos em cada tipo de aplicação, foi calculada a quantidade média de arquivos em cada versão do projeto, bem como os valores mínimo e máximo da quantidade de arquivos em cada versão. Essa coleta de dados foi realizada na tabela *Files*, descrita na Seção 3.2, que armazena todo o histórico de arquivos dos projetos. Para se obter a média, foi utilizada uma *query SQL* que calcula a divisão do número total de arquivos para um dado *project_id* pela quantidade de *commit_id* distintos presentes para este determinado projeto.

A query da Listagem 4.3 mostra um exemplo do cálculo, no qual o x da cláusula *where* deve ser substituído pelo *id* do projeto desejado que se encontra na tabela *project*. Na coluna *"Media de arquivos"* conterá o resultado desejado. Para se obter o valor máximo e mínimo de arquivos em uma certa versão do repositório, a *query* da Listagem 4.4 foi utilizada. Ela retorna para cada *commit*, a quantidade de arquivos presente naquele estágio. Para se obter os extremos, foi realizada uma ordenação dos resultados em ordem ascendente e descendente para se obter o menor e o maior número de arquivos, respectivamente.

Listagem 4.3: Query utilizada para obter a média de arquivos

```
1 SELECT
2       COUNT(f.`path`) AS "Num total de arquivos",
3       COUNT(DISTINCT f.commit_id) AS "Total de commits",
4       COUNT(f.`path`) / COUNT(DISTINCT f.commit_id) AS "Media de
            arquivos"
5 FROM
6       tcc.file f
7 WHERE
8       f.project_id = x
```

Listagem 4.4: Query utilizada para se obter o número de arquivos a cada *commit*

```
1 SELECT
2       COUNT(f.commit_id) AS "Num de arquivos"
3 FROM
4       tcc.file f
5 WHERE
6       f.project_id = x
7 GROUP BY
8       f.commit_id
```

Outro valor escolhido para ser mensurado foi número de arquivos na versão final. Para atingir este objetivo, foi utilizada a *query* descrita em 4.5, que é responsável por obter o valor do menor *commit* para aquele projeto. Isso acontece porque como o *script* desenvolvido itera sobre todo o versionamento do início ao fim, o *commit* com menor valor representa a versão mais atual do repositório. Posteriormente, o valor retornado por ela foi inserido juntamente na cláusula *where* da *query* da Listagem 4.6, filtrando agora não somente pelo *project_id*, quanto também pelo *commit_id*. A letra y representada simula a proposta para um *commit* de valor genérico que deve ser substituído pelo *id* desejado.

Listagem 4.5: Query utilizada para se obter o id do último *commit*

```
1 SELECT
2       MIN(f.commit_id)
3 FROM
4       tcc.file f
5 WHERE
6       f.project_id = x
```

Listagem 4.6: Query utilizada para se obter o número final de arquivos

```
1 SELECT
2       COUNT(f.`path`)
3 FROM
4       tcc.file f
5 WHERE
6       f.project_id = x
7     AND f.commit_id = y
```

Python

Esta subseção visa apresentar os resultados encontrados sobre o número de arquivos nos projetos escritos na linguagem de programação *Python*. Diante disso, a Tabela 4.3 dispõe os dados obtidos, e, de acordo com as informações, foram realçados os valores extremos

do conjunto de resultados, visando facilitar a interpretação.

Consoante a isso, pode-se verificar que nas aplicações *serverless*, os valores mais expressivos foram obtidos pelo projeto *streamalert* em todas as categorias. Por outro lado, nas colunas de média, valor máximo e versão final, o repositório *aws-serverless-shopping-cart* apresentou as menores taxas. Com relação ao menor valor mínimo, o projeto *aws-serverless-ecommerce* apresentou o menor resultado, de apenas um arquivo.

Na seção de aplicações monolíticas, é possível salientar o desempenho do projeto *monolith-filemanager*, que obteve os maiores resultados em todas as métricas analisadas. Em contrapartida, analisando os valores menos expressivos obtidos, o repositório *Finance-Full-Stack-Web-App-using-Flask-and-SQL* apresentou as menores taxas em todas as categorias.

A partir desse pressuposto, observa-se que os projetos monolíticos obtiveram uma redução de 30,69% na média, uma vez essa categoria apresentou uma taxa média de 28,31 e os projetos *serverless* o valor de 40,84. Além disso, houve redução de 64,45% no valor máximo de arquivos, com os monolíticos obtendo uma taxa média de 102,4 e os projetos serverless, o valor de 36,4. Contudo, os monolíticos apresentaram um aumento de 112,5% no valor mínimo, com o índice médio de 8, frente ao valor de 17, apresentado nas aplicações serverless.

Tabela 4.3: Resultado do número de arquivos para os projetos *Python*

Projeto	Média	Mínimo	Máximo	N° final	Tipo
Kindergarten-Backend	30,88	10	74	73	*serverless*
aws-serverless-ecommerce	42,95	**1**	73	72	*serverless*
aws-serverless-shopping	**12,80**	11	**13**	**13**	*serverless*
streamalert	**93,11**	**16**	**310**	**310**	*serverless*
domain-protect	24,47	2	42	42	*serverless*
Finance-Full-Stack-Web-App	**2,83**	**2**	**3**	**3**	monolítico
highlowbackend	13,08	9	17	17	monolítico
GoOutSafe	40,84	12	53	53	monolítico
monolith-filemanager	**54,90**	**51**	**59**	**59**	monolítico
django-data-browser	29,88	11	50	50	monolítico

Os gráficos das Figuras 4.2 e 4.3 ilustram o crescimento do número de arquivos ao longo do processo de versionamento dos projetos. É possível observar que os projetos da Figura 4.2 seguiram basicamente uma linha de crescimento um pouco mais linear, com o projeto *streamalert* apresentando o maior número de *commits* e arquivos. Contudo, os repositórios representados na 4.3, não apresentaram uma curva semelhante, com o *monolith-filemanager* já iniciando com um número maior de arquivos e o *django-data-browser* crescendo de forma incremental.

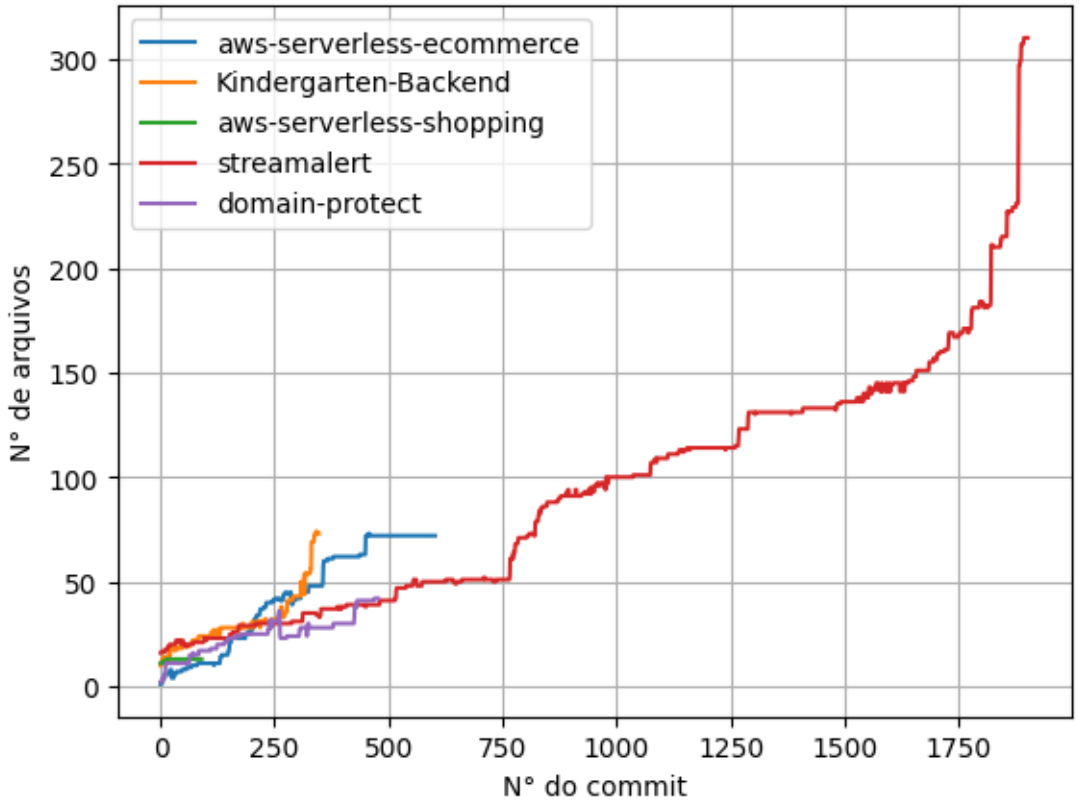

Figura 4.2: Crescimento do número de arquivos durante o versionamento dos projetos *serverless*.

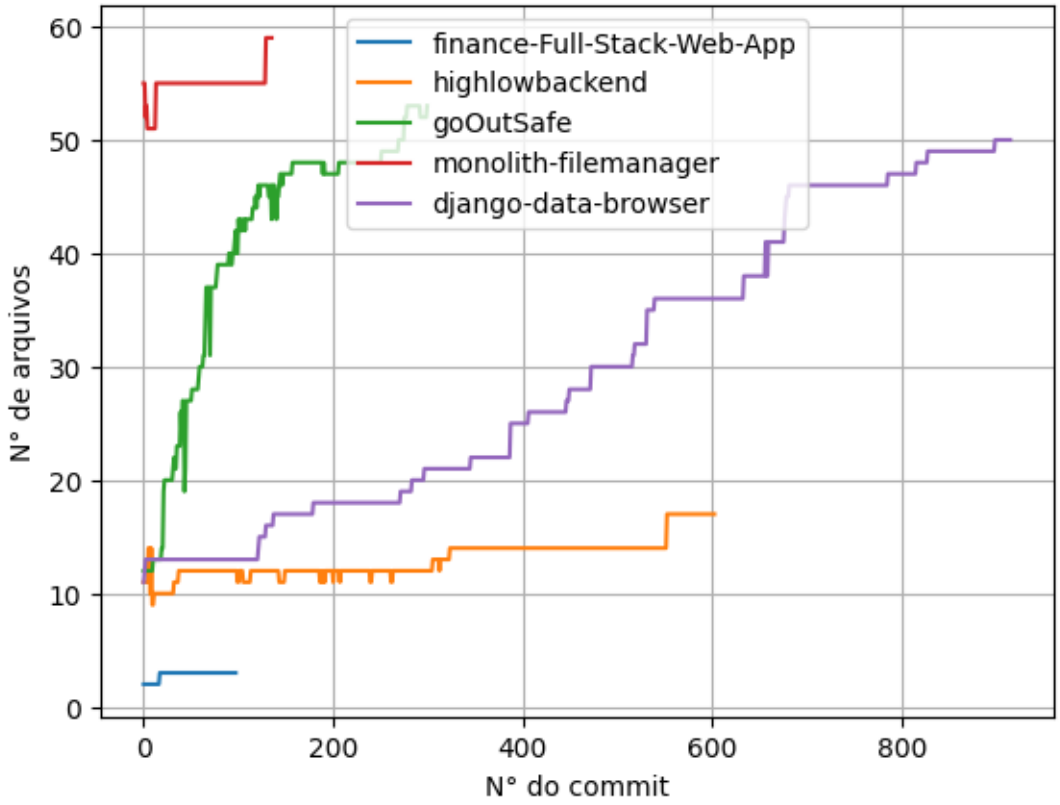

Figura 4.3: Crescimento do número de arquivos durante o versionamento dos projetos monolíticos.

JavaScript

Esta subseção visa trazer os resultados encontrados para os projetos escritos na linguagem de programação *JavaScript*, dispondo as informações detalhadamente na Tabela 4.4. De acordo com os dados obtidos, é possível salientar que para as aplicações *serverless*, o projeto *chabelitas-back* obteve os menores valores para a média, valor máximo e versão final. Por outro lado, o projeto *moonMail* obteve os maiores valores nesses mesmos quesitos. Além disso, na coluna de valor mínimo, o repositório *serverless-survey-forms* apresentou o menor índice, com apenas um arquivo e o *ruuvi-network-serverless* o valor máximo, de 7.

Com relação a seção dos monolíticos, é possível analisar que o projeto *auction-website-monolith* apresentou os valores mais expressivos nas colunas de média, valor máximo e versão final, em contrapartida do repositório *nodejs-shopping-cart* com os menores índices nessas mesmas métricas. Levando em consideração o valor mínimo dos dados, o repositório *bulletproof-vanilla-nodejs* apresentou a maior taxa, e os projetos *auction-*

website-monolith e *yelpCamp* obtiveram o menor valor, de apenas um arquivo.

A partir da análise dos resultados obtidos, é possível observar que houve uma redução de 64,07% na média de arquivos nos projetos monolíticos, haja vista que eles obtiveram uma taxa média de 31,51 e os projetos *serverless* obtiveram o índice de 87,68. Com relação ao valor máximo, também houve redução nesta categoria, com os monolíticos apresentando uma taxa média de 39 e os projetos *serverless* o índice de 193,2, representando redução de 79,81%. Contudo, houve um crescimento de 62,5% na métrica de valor mínimo, já que os projetos monolíticos obtiveram a taxa de 7,8 e as aplicações serverless, o valor de 4,8.

Tabela 4.4: Resultado do número de arquivos para os projetos *JavaScript*

Projeto	Média	Mínimo	Máximo	N° final	Tipo
chabelitas-back	**24,25**	6	**30**	**30**	*serverless*
moonMail	**183,03**	6	**536**	**396**	*serverless*
serverless-survey-forms	144,31	**1**	240	240	*serverless*
ruuvi-network-serverless	35,99	**7**	71	71	*serverless*
claudia-bot-builder	50,84	4	89	89	*serverless*
auction-website-monolith	**76,59**	**1**	**84**	**82**	monolítico
bulletproof-vanilla-nodejs	36,03	**32**	40	40	monolítico
nodejs-shopping-cart	**8,99**	3	**13**	**13**	monolítico
yelpCamp	15,95	**1**	24	18	monolítico
theChaotic	19,97	2	34	33	monolítico

Os gráficos das Figuras 4.4 e 4.5 mostram o comportamento do número de arquivos ao longo do versionamento do projeto. Em 4.4, no projeto *moonMail* é notável o ponto ressaltado anteriormente do pico de arquivos máximo que ocorre no *commit* 867. Na Figura 4.5, é notável que o projeto *auction-website-monolith* inicia com o maior número de arquivos, comparado com os demais projetos.

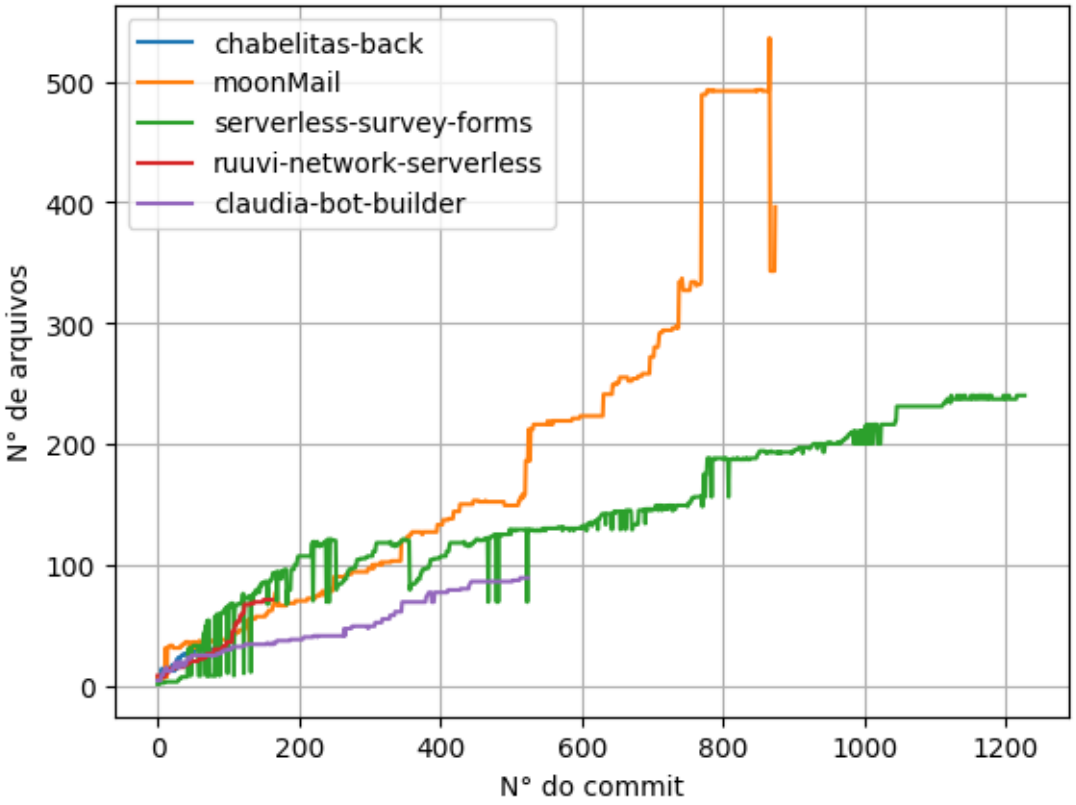

Figura 4.4: Crescimento do número de arquivos durante o versionamento dos projetos *serverless*.

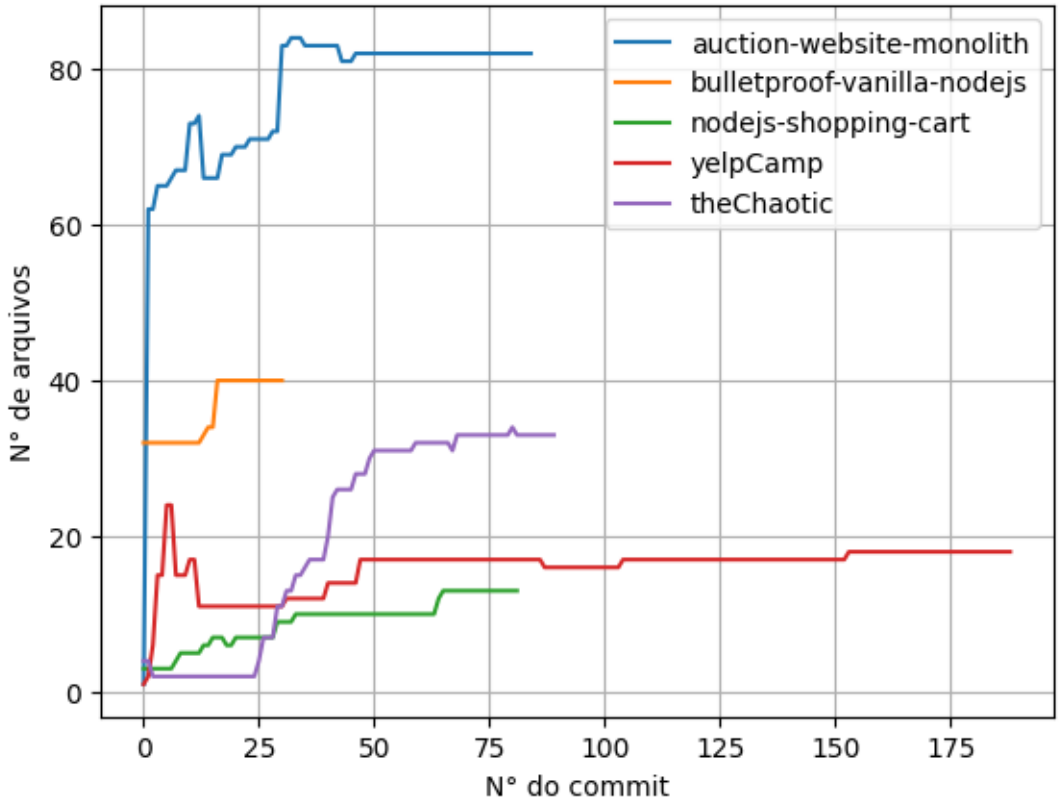

Figura 4.5: Crescimento do número de arquivos durante o versionamento dos projetos monolíticos.

Já o gráfico da Figura 4.6 representa a média de arquivos em função dos projetos. A partir da representação, é notório que a média dos projetos *serverless* desenvolvidos em *JavaScript* foi superior a todas as demais e em relação aos projetos *Python*, os *serverless* também apresentaram as maiores taxas.

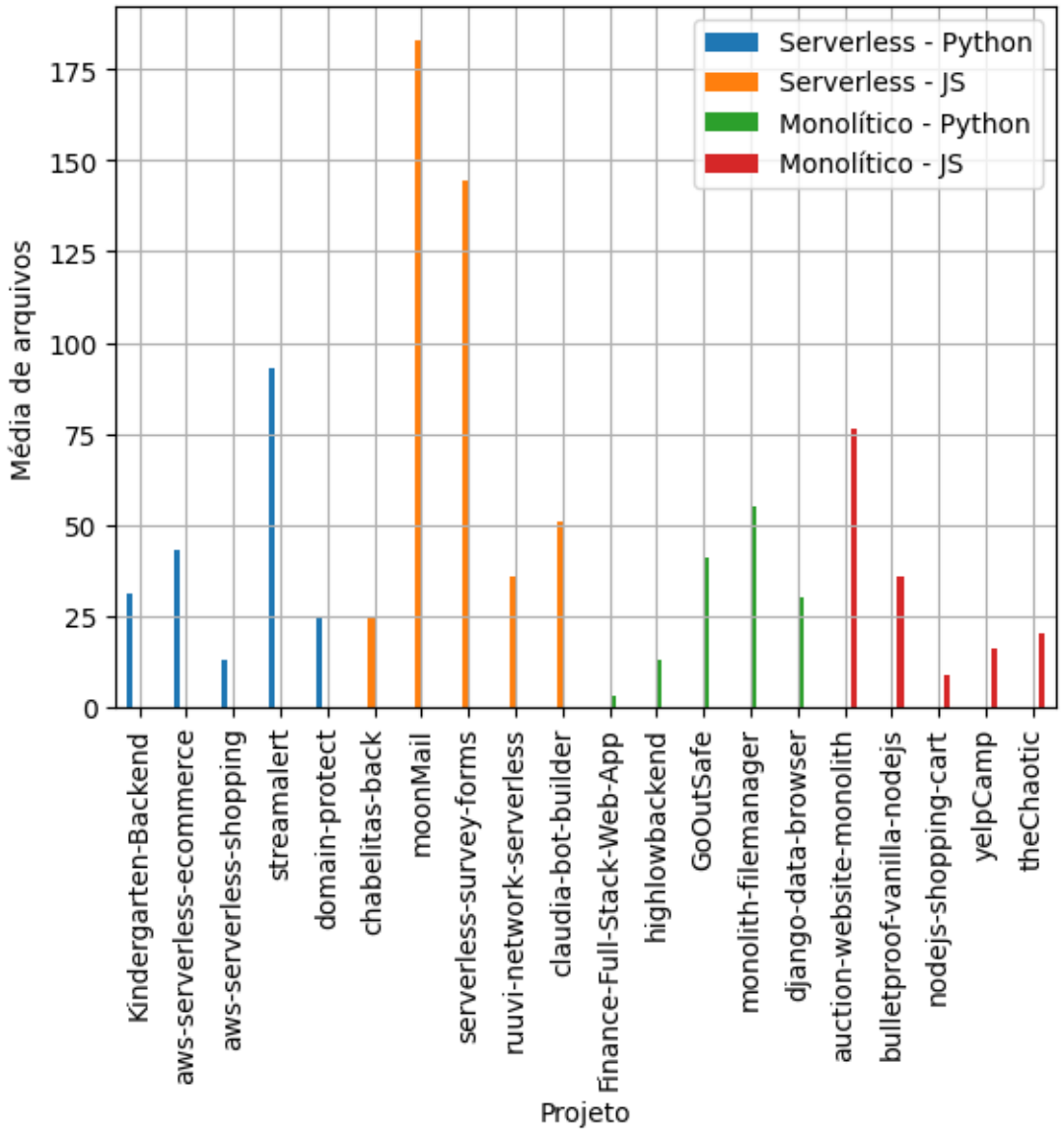

Figura 4.6: Média de arquivos em função dos projetos.

Conclusão

Os números mostraram que, para *Python*, a média de arquivos em aplicações *serverless* foi de 40,84, contra 28,31 em aplicações monolíticas, representando um crescimento de 44,29%. Além disso, o número mínimo de arquivos neste mesmo segmento foi de 8 em *serverless* e 17 em monolíticas, representando uma redução de 52,94%. Somado a isso, a média do número máximo de arquivos em *serverless* foi de 102,40 contra 36,4 nas

aplicações monolíticas, representando crescimento de 181,32%.

Com relação à *JavaScript*, as aplicações *serverless* apresentaram uma média de 87,68 arquivos, em contrapartida as monolíticas apresentaram 31,51, representando um crescimento de 178,31%, no número mínimo de arquivos, em *serverless* essa média ficou em 4,8 e nas monolíticas, em 7,8, mostrando redução de 38,46%. Já no número máximo, a média em *serverless* foi 193,2, contra 39 nas monolíticas, representando crescimento de 395,38%.

Diante disso, pode-se inferir que o número de arquivos tende a ser maior em aplicações *serverless*, principalmente em aplicações desenvolvidas em *JavaScript*, que apresentaram uma média 2,15 vezes superior aos projetos desta mesma categoria em *Python*.

4.3.2 Como os aspectos físicos e sintáticos dos arquivos se comportam em cada uma das duas arquiteturas?

Esta pergunta visa entender como os aspectos sintáticos medidos pelo número de classes, funções e métodos, bem como os aspectos físicos, medidos pelo tamanho dos arquivos em linhas, têm grandes variações nas arquiteturas monolíticas e *serverless*. Para respondê-la, será utilizado como base a tabela *file* mencionada na Seção 3.2. Nela, há a presença de 4 colunas que serão importantes nesta parte da avaliação, sendo elas: *qnt_method*, *qnt_class*, *qnt_function* e *total_lines*.

O primeiro dado coletado foi a média de todos esses dados para cada repositório, para isso fez-se uso de uma *query SQL* que utiliza como base a função *AVG*, responsável por calcular a média de um determinado valor. Um exemplo de uso para um *project_id* hipotético x pode ser visualizado na Listagem 4.7.

Listagem 4.7: Query para se obter a média de métodos, funções, classes e linhas de um arquivo

```
1 SELECT
2       AVG(f.qnt_method),
3       AVG(f.qnt_function),
4       AVG(f.qnt_class),
5       AVG(f.total_lines)
6 FROM
7       tcc.file f
8 WHERE
9       f.project_id = x
```

Também foram coletados os maiores e menores somatórios de métodos, classes, funções e linhas dos arquivos ao longo do versionamento. Para isso, fez-se uso da função *SUM*, que retorna o somatório de determinado campo. A *query* da Listagem 4.8 exemplifica o processo.

Listagem 4.8: Query para se obter os somatórios máximos e mínimos de métodos, funções, classes e linhas de um arquivo

```
1  SELECT
2      SUM(f.qnt_method),
3      SUM(f.qnt_class),
4      SUM(f.qnt_function),
5      SUM(f.total_lines)
6  FROM
7       tcc.file f
8  WHERE
9       f.project_id = x
10 GROUP BY
11      f.commit_id
```

Métodos

Nesta seção serão analisados os resultados obtidos considerando os métodos. Os resultados para *Python* são apresentados na Tabela 4.5. Para projetos do tipo *serverless* foi possível observar que os maiores valores obtidos nas quatro métricas aconteceram no projeto *streamalert*. O projeto *domain-protect*, em contrapartida, apresentou a menor média, enquanto *aws-serverless-shopping* apresentou como resultado menores índices no valor máximo de métodos, além da versão final. Ademais, dois dos repositórios analisados apresentaram um número mínimo de zero métodos.

Considerando os projetos monolíticos, é possível enfatizar os resultados de *monolith-*

filemanager, que obteve os maiores valores considerando as colunas de valor máximo, mínimo e da versão final. Em contrapartida, o *Finance-Full-Stack-Web-App-using-Flask-and-SQL* apresentou índices mais baixos nas mesmas métricas. Considerando os aspectos apontados como a média dos dados obtidos, o projeto *highlowbackend* apresentou a obtenção de maiores índices, com 6,57, em comparação com o *GoOutSafe* que apresentou índice de 0,57.

A partir da análise dos resultados obtidos, é possível observar que a média de métodos para os projetos monolíticos foi de 4,11 e para *serverless*, de 1,51, o que representa um aumento de 171,56% na categoria de média para projetos monolíticos. Para o número mínimo, a média dos valores foi de 75,8 nos monolíticos e de 17 para *serverless*, o que indica um crescimento de 345,88% no número mínimo de métodos nos projetos monolíticos. Já no que refere ao número máximo, ocorreu uma redução de 69,81% nas aplicações monolíticas, que obtiveram uma média de 164,6, contra o valor de 545,2 obtida nas aplicações *serverless*.

Tabela 4.5: Número de métodos em projetos *Python*

Projeto	Média	Máximo	Mínimo	N° final	Tipo
Kindergarten-Backend	0,82	44	2	44	*serverless*
aws-serverless-ecommerce	0,47	49	**0**	15	*serverless*
aws-serverless-shopping	0,16	**2**	2	**2**	*serverless*
streamalert	**5,97**	**2615**	**81**	**2615**	*serverless*
domain-protect	**0,14**	16	**0**	16	*serverless*
Finance-Full-Stack-Web-App	2,0	**6**	**4**	**6**	monolítico
highlowbackend	**6,57**	167	42	167	monolítico
GoOutSafe	**0,57**	37	6	37	monolítico
monolith-filemanager	5,63	**331**	**275**	**331**	monolítico
django-data-browser	5,76	282	52	282	monolítico

Para os dados obtidos em *JavaScript*, a Tabela 4.6 mostra que em *serverless*, o projeto *ruuvi-network-serverless* apresentou todas as colunas zeradas e, portanto, obteve os menores índices. Este cenário também se repetiu na coluna de valor mínimo, na qual somente *chabelitas-back* apresentou o valor 1. Em contrapartida, o projeto denominado *serverless-survey-forms* obteve os valores de média, máximo e versão final mais expressivos. Ainda nesse contexto, no que se refere aos projetos monolíticos, três repositórios evidenciaram todos os valores zerados, enquanto o projeto *bulletproof-vanilla-nodejs* foi o único representante com maiores valores nesta categoria.

A partir desse pressuposto, observa-se que na categoria de média, os projetos *serverless* obtiveram uma taxa de 0,65, contra o valor de 0,09 obtido pelos monolíticos, o que representa um crescimento de 638,63% na categoria de média nos projetos *serverless*. No número máximo de métodos, também ocorreu um crescimento nesta categoria, agora de 4295%, uma vez que a média do número máximo foi de 175,8 nos projetos *serverless* e de 4 nos monolíticos. Entretanto, houve uma redução de 92,31% no valor mínimo, pois a média desse índice nos projetos *serverless* foi de 0,2 e nos projetos monolíticos, de 2,6.

Tabela 4.6: Número de métodos em projetos *JavaScript*

Projeto	Média	Máximo	Mínimo	N° final	Tipo
chabelitas-back	0,28	9	**1**	7	*serverless*
moonMail	0,07	347	0	173	*serverless*
serverless-survey-forms	**1,48**	**357**	0	**357**	*serverless*
ruuvi-network-serverless	**0,00**	**0**	0	**0**	*serverless*
claudia-bot-builder	1,42	166	0	166	*serverless*
auction-website-monolith	0,05	5	0	4	monolítico
bulletproof-vanilla-nodejs	**0,39**	**15**	**13**	**15**	monolítico
nodejs-shopping-cart	0,00	0	0	0	monolítico
yelpCamp	0,00	0	0	0	monolítico
theChaotic	0,00	0	0	0	monolítico

O gráfico da Figura 4.7 representa a média de métodos em relação aos projetos. Sendo assim, é possível observar que os projetos monolíticos em *JavaScript*, identificados pela cor vermelha, especificamente os três últimos, por possuírem valores zerados, não aparecem no gráfico. Outrossim, é possível constatar que os projetos monolíticos em *Python* obtiveram as maiores médias em comparação com os demais projetos. Vale salientar que o repositório *streamalert* destacou-se como o de maior pico em sua categoria.

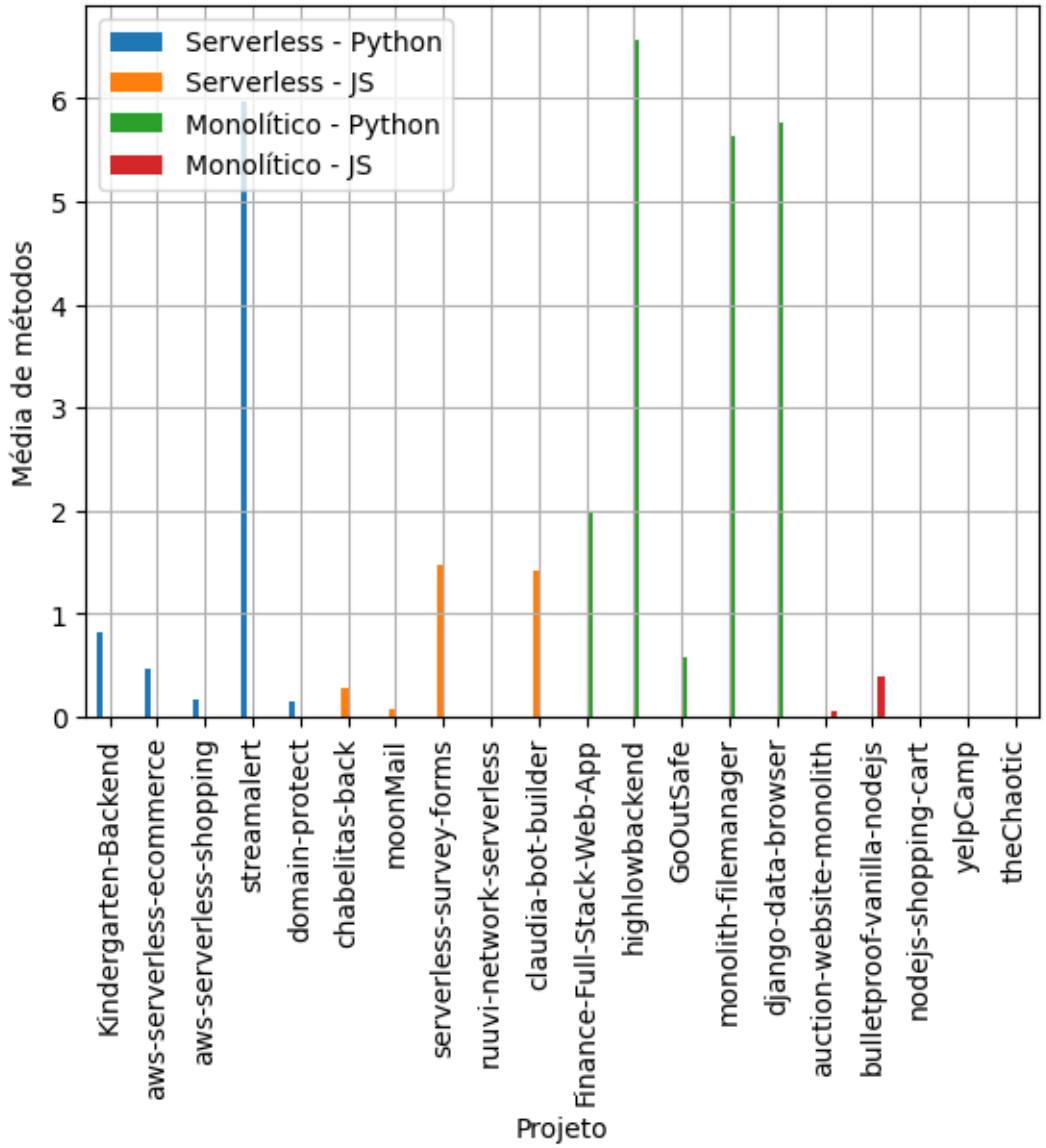

Figura 4.7: Média de métodos em função dos projetos.

Classes

Esta subseção visa discorrer sobre os resultados encontrados considerando classes. A Tabela 4.7 apresenta os dados obtidos para os projetos escritos em *Python*. Diante disso, é possível observar que para *serverless*, o repositório *streamalert* obteve os maiores valores para todas as métricas analisadas. Em contrapartida, os menores valores no quesito média e valor mínimo foram obtidos pelo *aws-serverless-ecommerce*. Já no que se refere aos valores máximo e versão final, o projeto *aws-serverless-shopping* obteve os menores índices.

Para os projetos monolíticos, o repositório *Finance-Full-Stack-Web-App-using-Flask-and-SQL* apresentou a maior média, com o valor de 3,12, contra a menor, de 0,71,

obtida pelo *GoOutSafe*. Na coluna de valor máximo e versão final, o projeto *django-data-browser* obteve os valores mais expressivos, ao contrário do *Finance-Full-Stack-Web-App-using-Flask-and-SQL*, que obteve os menores resultados nesta categoria. No quesito número mínimo, pode-se destacar a quantidade de 6 classes do projeto *highlowbackend* e o maior índice, de 65 classes no projeto *monolith-filemanager*.

Diante disso, pode-se verificar que em relação à média, nos projetos monolíticos, houve um aumento de 411,69% nessa métrica, já que a média de classes nesta categoria foi de 1,75, contra o valor de 0,34 obtido em serverless. O número mínimo também sofreu um crescimento similar, de 414,28%, uma vez que a média deste campo foi de 4,2 em *serverless* e de 21,6 nos monolíticos. Por outro lado, houve redução de 41,07% em relação ao número máximo, com *serverless* obtendo uma média de 89,6 e os projetos monolíticos, o valor de 52,8.

Tabela 4.7: Número de classes em projetos *Python*

Projeto	Média	Máximo	Mínimo	N° final	Tipo
Kindergarten-Backend	0,23	12	2	12	*serverless*
aws-serverless-ecommerce	**0,14**	10	**0**	10	*serverless*
aws-serverless-shopping	0,16	**2**	2	**2**	*serverless*
streamalert	**0,97**	**408**	**16**	**408**	*serverless*
domain-protect	0,21	16	1	9	*serverless*
Finance-Full-Stack-Web-App	**3,12**	**9**	8	**9**	monolítico
highlowbackend	0,75	14	**6**	14	monolítico
GoOutSafe	**0,71**	38	8	37	monolítico
monolith-filemanager	1,24	74	**65**	74	monolítico
django-data-browser	2,93	**129**	21	**128**	monolítico

Para os resultados dos projetos escritos em *JavaScript*, os dados estão dispostos na Tabela 4.8. É possível salientar que houve diversos casos de valores zerados, como no exemplo do projeto *ruuvi-network-serverless*, que não apresentou nenhuma classe durante todo o versionamento. Na coluna de média, a maior quantidade apresentada foi do repositório *claudia-bot-builder*, por outro lado, nas métricas de valor máximo e versão final, o projeto *serverless-survey-forms* obteve os maiores índices. Por fim, o repositório *chabelitas-back* apresentou o maior número mínimo, comparado aos outros projetos. Com relação aos monolíticos, o cenário de linhas zeradas também se repetiu, com 3 projetos que não apresentaram nenhuma classe. O único destaque nessa categoria foi o repositório *bulletproof-vanilla-nodejs*, que, mesmo apresentando valores pequenos, foi o responsável

pelos maiores índices em todas as métricas.

A partir dessas informações, percebe-se que nos projetos monolíticos houve redução de 87,13% na média, com os monolíticos obtendo uma taxa de 0,03 e os projetos serverless, o valor de 0,20. Também ocorreu redução no número máximo de classes, com os monolíticos obtendo uma média de 1,4 nesta categoria e os projetos *serverless* de 29,8, representando uma redução de 95,30%. Por outro lado, houve um aumento de 200% na métrica de número mínimo comparado com as aplicações *serverless*, uma vez que os monolíticos obtiveram uma taxa média de 0,6 e os projetos serverless, o valor de 0,2.

Tabela 4.8: Número de classes em projetos *JavaScript*

Projeto	Média	Máximo	Mínimo	N° final	Tipo
chabelitas-back	0,28	9	**1**	7	*serverless*
moonMail	0,01	27	0	20	*serverless*
serverless-survey-forms	0,32	**64**	0	**64**	*serverless*
ruuvi-network-serverless	**0,0**	**0**	0	**0**	*serverless*
claudia-bot-builder	**0,40**	49	0	49	*serverless*
auction-website-monolith	0,03	3	0	3	monolítico
bulletproof-vanilla-nodejs	**0,10**	**4**	**3**	**4**	monolítico
nodejs-shopping-cart	0,0	0	0	0	monolítico
yelpCamp	0,0	0	0	0	monolítico
theChaotic	0,0	0	0	0	monolítico

O gráfico da Figura 4.8 ilustra a média de classes em função dos projetos. É possível notar que os repositórios monolíticos escritos em *Python*, representados pela cor verde, se destacam pelos maiores valores, em contraste com os monolíticos em *JavaScript*, representados na cor vermelha, que apresentaram 3 projetos com valores zerados, portanto, não são visíveis no gráfico. Com relação às aplicações *serverless*, os projetos em *Python* também tiveram maior destaque, frente aos desenvolvidos em *JavaScript*.

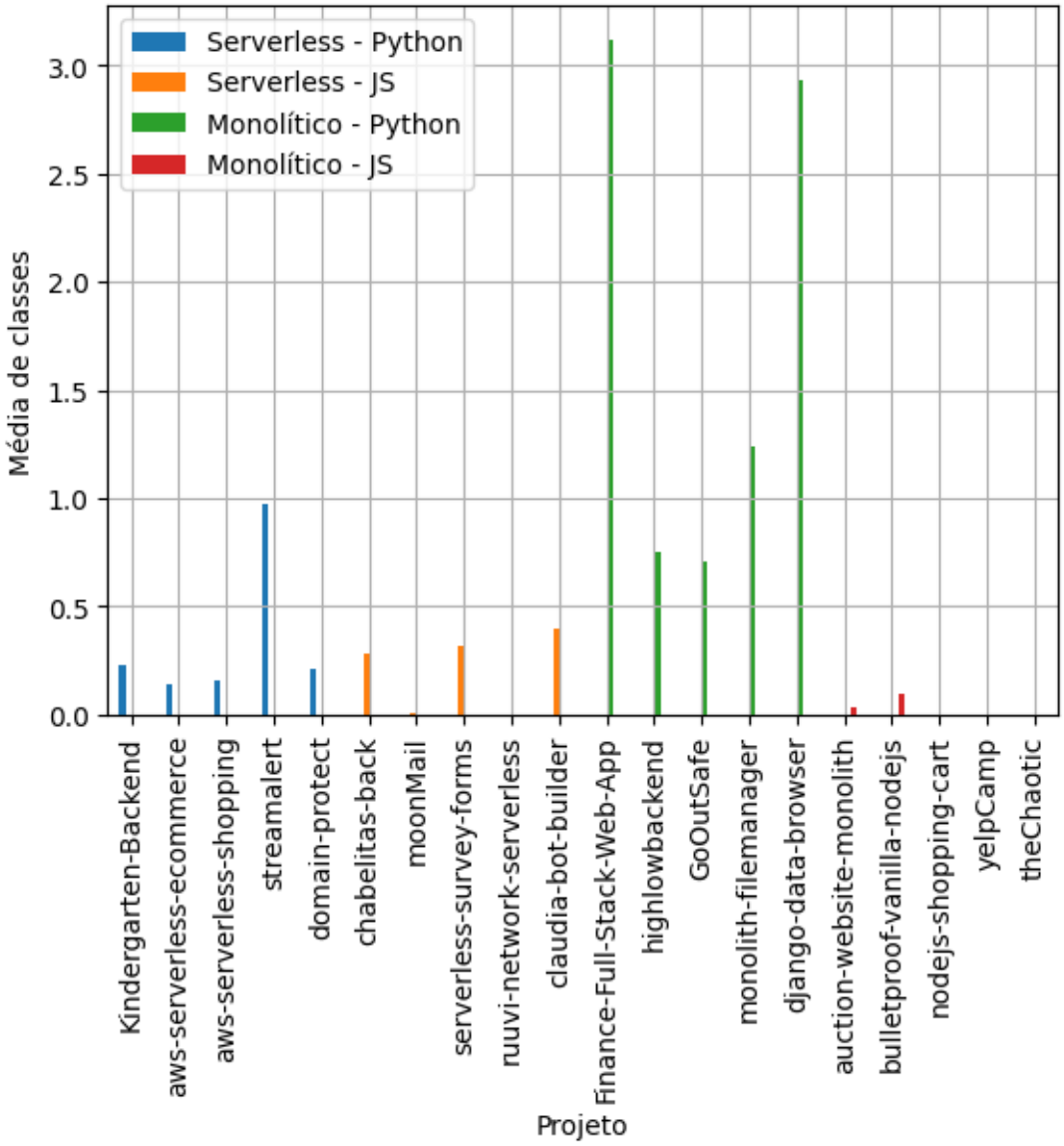

Figura 4.8: Média de classes em função dos projetos.

Funções

Esta subseção apresenta os resultados encontrados no âmbito das funções em projetos *JavaScript* e *Python*. A Tabela 4.9 apresenta os resultados para projetos *Python*. Sendo assim, é possível observar que, para as aplicações *serverless*, os maiores valores nos quesitos média, valor máximo e versão final foram obtidos pelo projeto *aws-serverless-ecommerce*. Em contrapartida, o maior valor mínimo de funções foi observado no repositório *streamalert*. Por outro lado, analisando os menores valores, percebe-se que, o valor mínimo, dois projetos obtiveram o valor 8, sendo eles o *domain-protect* e o *Kindergarten-Backend*. Este último também obteve a menor taxa em relação a média de funções, já nas colunas de valor máximo e versão final, o repositório *aws-serverless-shopping* foi o responsável pelos

menores índices nessas duas categorias.

Somado a isso, analisando os resultados dos monolíticos, é notório que nas colunas de valor máximo e versão final, os valores mais expressivos foram obtidos pelo projeto *GoOutSafe*. Com relação ao maior valor mínimo, o repositório *django-data-browser* ocupou essa posição, em contrapartida ao projeto *Finance-Full-Stack-Web-App*, que apresentou o valor mais expressivo no quesito de média de funções. Por outro lado, observando os menores valores, pode-se verificar que o projeto *monolith-filemanager* obteve os menores índices em todas as colunas, sendo que, na métrica de valor mínimo tanto ele quanto o projeto *highlowbackend* coincidiram com o mesmo resultado.

Portanto, percebe-se que nos projetos monolíticos houve um aumento de 44,29% na métrica relacionada à média, já que nos monolíticos a média desse valor foi de 4,20 e em serverless, de 2,91. Contudo, os índices de número máximo e mínimo sofreram reduções de 39,54% e 14,49% em relação aos projetos *serverless*, respectivamente, uma vez que a média do número máximo em *serverless* foi de 226,6 e dos monolíticos, de 137. Já no número mínimo, esses valores atingiram a marca média de 13,8 nos projetos *serverless* e de 11,8 nos monolíticos.

Tabela 4.9: Número de funções em projetos *Python*

Projeto	Média	Máximo	Mínimo	N° final	Tipo
Kindergarten-Backend	**1,25**	101	**8**	101	*serverless*
aws-serverless-ecommerce	**6,03**	**457**	13	**445**	*serverless*
aws-serverless-shopping	1,42	**19**	18	**18**	*serverless*
streamalert	2,16	350	**22**	343	*serverless*
domain-protect	3,68	206	**8**	206	*serverless*
Finance-Full-Stack-Web-App	**7,11**	21	14	17	monolítico
highlowbackend	4,15	100	**3**	100	monolítico
GoOutSafe	4,89	**302**	12	**302**	monolítico
monolith-filemanager	**0,20**	**13**	**3**	**13**	monolítico
django-data-browser	4,63	249	**27**	249	monolítico

Com relação aos dados encontrados para *JavaScript*, a Tabela 4.10 dispõe os resultados. Diante disso, é possível verificar que, para *serverless* os maiores índices de valor máximo e versão final foram obtidos pelo projeto *moonMail*, já no que se refere à média, o valor mais expressivo foi obtido pelo *claudia-bot-builder* e, na coluna de valor mínimo, pelo projeto *chabelitas-back*. Por outro lado, ao se observar os menores índices, pode-se analisar que o projeto *chabelitas-back* foi o representante com os menores valores

em todas as categorias, exceto no valor mínimo, posição ocupada pelo *serverless-survey-forms*.

No que se refere aos monolíticos, as maiores taxas de valor máximo e versão final foram obtidos pelo projeto *auction-website-monolith*. Na categoria de média, o projeto *yelpCamp* trouxe o maior índice e na coluna de valor mínimo, a maior taxa foi obtida pelo *bulletproof-vanilla-nodejs*. Com relação aos menores índices, é possível verificar que o projeto *theChaotic* obteve os valores menores expressivos nas categorias de média e valor mínimo e o repositório *nodejs-shopping-cart* os menores índices nas colunas de versão final e valor máximo.

A partir da análise dos resultados obtidos, é possível observar que, nos projetos monolíticos, houve uma redução de 35,38% na média, já que eles obtiveram a taxa média de 4,23 funções nesta categoria e os projetos serverless, atingiram o valor de 6,55. Além disso, também ocorreu redução de 89,64% no valor máximo, uma vez que os projetos *serverless* obtiveram uma média de 1553,8 e os projetos monolíticos, a taxa de 161. Contudo, ocorreu um aumento de 3,17% no valor mínimo de funções, com *serverless* apresentando uma taxa 12,6 e os monolíticos o valor de 13.

Tabela 4.10: Número de funções em projetos *JavaScript*

Projeto	Média	Máximo	Mínimo	N° final	Tipo
chabelitas-back	**3,74**	**131**	**20**	**131**	*serverless*
moonMail	5,41	**3982**	13	**2666**	*serverless*
serverless-survey-forms	5,97	1711	**1**	1711	*serverless*
ruuvi-network-serverless	3,76	331	13	331	*serverless*
claudia-bot-builder	**13,85**	1614	16	1614	*serverless*
auction-website-monolith	5,47	**467**	2	**459**	monolítico
bulletproof-vanilla-nodejs	1,61	63	**53**	63	monolítico
nodejs-shopping-cart	3,86	**47**	4	**47**	monolítico
yelpCamp	**8,61**	164	5	160	monolítico
theChaotic	**1,60**	64	**1**	62	monolítico

O gráfico da Figura 4.9 ilustra a média de funções em relação aos projetos. Com essa informação, é perceptível que os projetos *serverless* escritos em *JavaScript* obtiveram as maiores taxas, com o projeto *claudia-bot-builder* se sobressaindo com uma média de 13,85 funções. Além disso, também é notório que nos monolíticos, os projetos escritos em *JavaScript* obtiveram maiores taxas, frente aos desenvolvidos em *Python*.

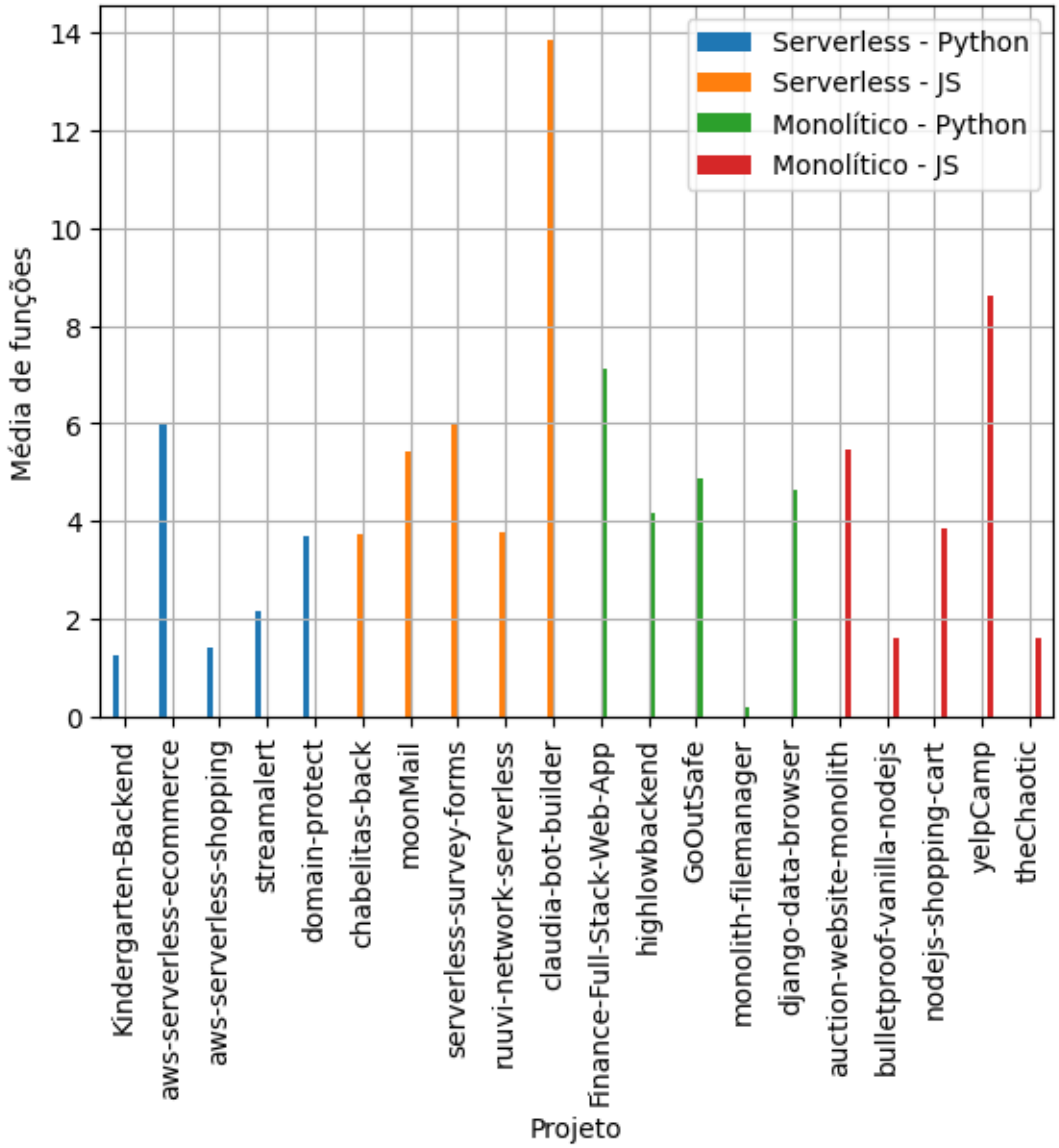

Figura 4.9: Média de funções em relação aos projetos.

Linhas

Esta subseção visa discorrer sobre os resultados no aspecto do número de linhas dos arquivos. A Tabela 4.11 apresenta os resultados encontrados para os repositórios desenvolvidos em *Python*. Observando os dados, pode-se verificar que para os projetos *serverless*, o repositório *streamalert* obteve os maiores índices em todas as categorias. Por outro lado, os menores valores foram obtidos pelo projeto *aws-serverless-shopping* nas colunas de máximo e versão final, bem como pelo *Kindergarten-Backend* na média e pelo *aws-serverless-ecommerce* com o menor valor mínimo.

No que se refere aos monolíticos, os menores valores foram obtidos pelo *highlowbackend* em todas as categorias. Por outro lado, analisando as taxas mais expres-

sivas, pode-se verificar que o projeto *Finance-Full-Stack-Web-App-using-Flask-and-SQL* apresentou a maior média. Além disso, na coluna de valor mínimo, o projeto *monolith-filemanager* foi responsável pelo maior valor. Com relação às colunas de valor máximo e versão final, os maiores valores foram obtidos pelo repositório *django-data-browser*.

Diante disso, percebe-se que nos projetos monolíticos houve uma redução de 2,27% na métrica de média, uma vez que eles obtiveram uma taxa de 112,39, frente ao valor de 115 obtido nas aplicações serverless. O valor máximo também apresentou redução nos monolíticos, haja vista que essa categoria obteve a taxa média de 4031 e os projetos serverless, o valor de 16197,4, representando redução de 75,11%. Contudo, houve um aumento de 35,60% no valor mínimo, com os projetos monolíticos apresentando uma taxa média de 1207,4 e os projetos serverless, o valor de 890,4.

Tabela 4.11: Número de linhas nos arquivos em projetos *Python*

Projeto	Média	Mínimo	Máximo	N° final	Tipo
Kindergarten-Backend	**31,18**	252	2426	2426	*serverless*
aws-serverless-ecommerce	177,56	**183**	13163	13030	*serverless*
aws-serverless-shopping	61,88	731	**821**	**801**	*serverless*
streamalert	**191,58**	**2984**	**59894**	**59894**	*serverless*
domain-protect	112,81	302	4683	4662	*serverless*
Finance-Full-Stack-Web-App	**215,94**	574	643	605	monolítico
highlowbackend	**13,08**	**9**	**17**	**17**	monolítico
GoOutSafe	99,15	338	6552	6550	monolítico
monolith-filemanager	82,79	**4050**	4904	4904	monolítico
django-data-browser	150,98	1066	**8039**	**8039**	monolítico

Para os dados obtidos em *JavaScript*, a Tabela 4.12 dispõe os resultados encontrados. Observando os dados, é possível verificar que, para *serverless*, o projeto *chabelitas-back* obteve os números mais representativos para a média de linhas e o valor mínimo. Na coluna de valor máximo, o projeto *moonMail* ocupou o lugar de destaque e, por fim, na de versão final, o maior índice foi obtido pelo *serverless-survey-forms*. Por outro lado, analisando os menores valores, pode-se verificar que na categoria de média, a menor foi obtida pelo *moonMail* e o de menor valor mínimo, pelo *serverless-survey-forms*. Já nas métricas de valor máximo e versão final, o projeto *chabelitas-back* apresentou os menores índices.

Já na seção de monolíticos, realizando a análise dos maiores índices, é possível enfatizar que o projeto *yelpCamp* obteve a maior média e o *bulletproof-vanilla-nodejs* a

quantidade mais representativa no valor mínimo. Contudo, nas colunas de valor máximo e versão final, o projeto *auction-website-monolith* obteve maior destaque. Verificando os menores resultados, é possível salientar que o projeto *bulletproof-vanilla-nodejs* obteve a menor média, frente ao repositório *auction-website-monolith*, com o menor número mínimo de linhas nos arquivos. Por fim, nas colunas de máximo e versão final, o projeto responsável pelos menores valores foi o *nodejs-shopping-cart*.

A partir das informações levantadas, percebe-se que nos projetos monolíticos houve reduções em todas as métricas, sendo de 38,42% na média, por causa da taxa de 85,97 apresentado nos projetos *serverless* e de 52,94 nos monolíticos. Houve também redução de 26,75% no valor mínimo de linhas, com *serverless* apresentando a taxa média de 245,2 e os monolíticos, o valor de 179,6. E uma redução de 83,68% no valor máximo, com *serverless* apresentando uma taxa média de 13399,8 e os monolíticos a taxa de 2186,2.

Tabela 4.12: Número de linhas nos arquivos em projetos *JavaScript*

Projeto	Média linhas	Mínimo	Máximo	N° final	Tipo
chabelitas-back	**130,71**	**701**	**4715**	**4715**	*serverless*
moonMail	**30,33**	115	**23006**	16985	*serverless*
serverless-survey-forms	75,58	**6**	20531	**20531**	*serverless*
ruuvi-network-serverless	98,35	248	7981	7981	*serverless*
claudia-bot-builder	94,89	156	10766	10766	*serverless*
auction-website-monolith	73,34	**7**	**6235**	**6186**	monolítico
bulletproof-vanilla-nodejs	**24,15**	**775**	963	963	monolítico
nodejs-shopping-cart	45,49	61	**564**	**564**	monolítico
yelpCamp	**80,45**	41	1491	1491	monolítico
theChaotic	41,28	14	1678	1678	monolítico

O gráfico da Figura 4.10 ilustra a média de linhas em função dos projetos. Diante disso, é notório que os desenvolvidos em *Python* obtiveram os maiores índices neste aspecto, com o *Finance-Full-Stack-Web-App-using-Flask-and-SQL* apresentando o maior pico. Em *JavaScript*, os projetos *serverless* se superaram nos valores, frente aos monolíticos.

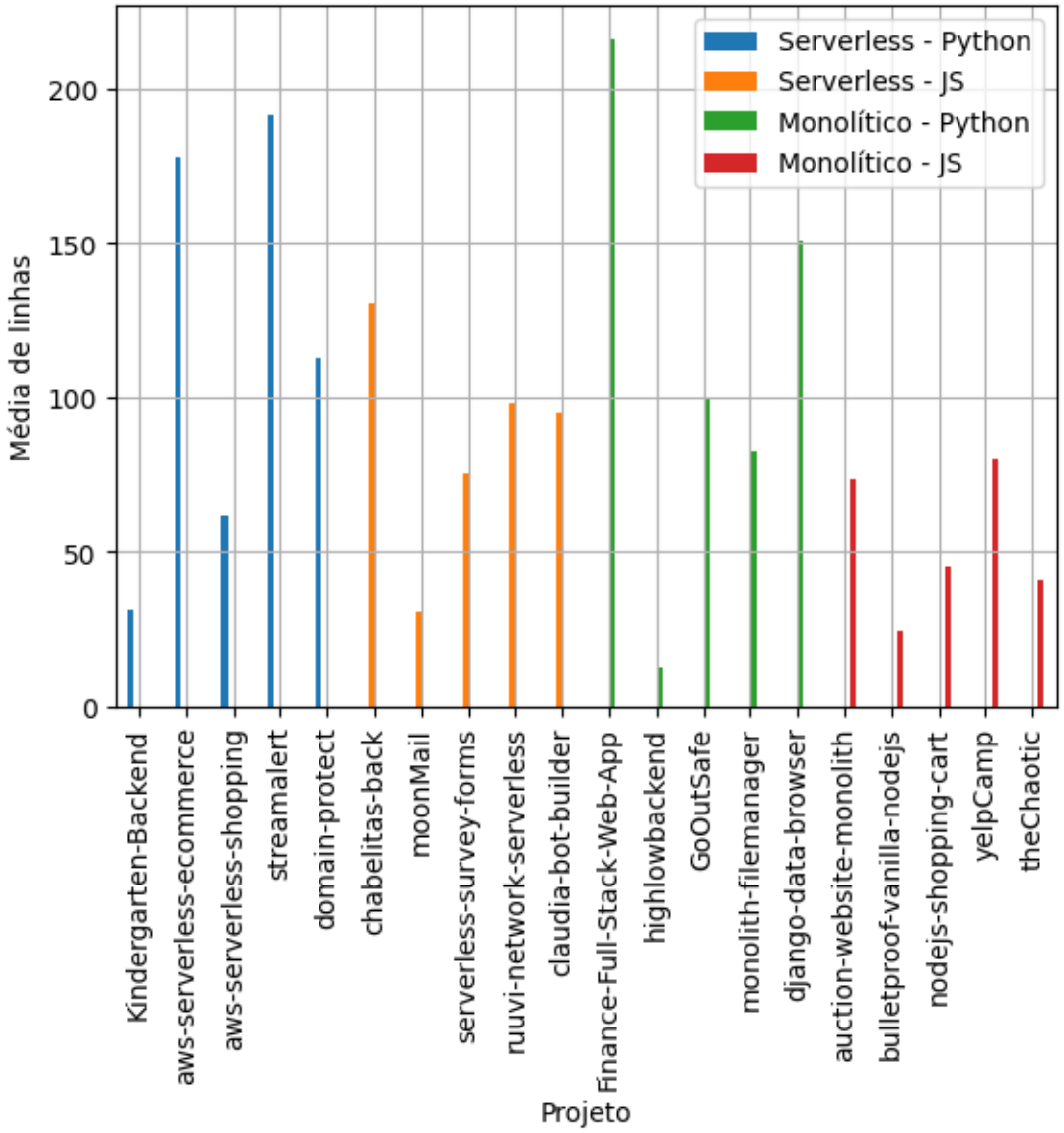

Figura 4.10: Média de linhas em função dos projetos.

Conclusão

Diante dos resultados encontrados, pode-se verificar que a média de classes em *Python* foi superior em aplicações monolíticas, obtendo uma média de 1,75, contra uma média de 0,34 em aplicações *serverless*. Para *JavaScript*, o valor foi superior nos projetos *serverless*, obtendo uma média de 0,20, contra uma média de 0,03 nos monolíticos. Uma observação é que em *JavaScript*, houve a presença de 3 repositórios monolíticos e 1 em *serverless* que não apresentaram nenhuma classe durante o versionamento.

Com relação aos métodos, os repositórios que apresentaram valores zerados também não apresentaram métodos. Além disso, em *Python*, a média de métodos foi superior em aplicações monolíticas, com uma média de 4,11, enquanto para *serverless* a média foi

igual a 1,51. Em *JavaScript*, o valor se manteve superior nas aplicações *serverless*, com uma média de 0,65, contra 0,09 das monolíticas.

Já para as funções, os projetos em *Python* obtiveram média superior nas aplicações monolíticas, com uma média de 4,20, frente a média de 2,91 em *serverless*. Para *JavaScript*, o valor foi superior nas aplicações *serverless*, com uma média de 6,55, contra o valor de 4,23 das aplicações monolíticas.

Por fim, com relação ao número de linhas dos arquivos, foi possível observar que *serverless* obteve os maiores índices em ambas as linguagens. Em *Python*, a média em aplicações *serverless* foi de 115, contra 112,39 nas aplicações monolíticas. Já para *JavaScript*, em *serverless*, a média apresentada foi de 85,98, contra 52,94 nas aplicações monolíticas.

4.3.3 A complexidade ciclomática e o número de linhas das funções e métodos entre as duas arquiteturas apresenta diferenças?

Esta subseção tem o intuito de demonstrar como os resultados de complexidade ciclomática e quantidade de linhas dos métodos e funções se comportam. As *queries* utilizadas foram bastante similares às demonstradas na subseção 4.3.2, com a diferença que a tabela utilizada como base foi a *function_method*. Essa subceção está dividida em dois momentos: a 4.3.3, que aborda o comportamento da complexidade ciclomática e a 4.3.3 que discorre sobre a quantidade de linhas.

Complexidade ciclomática

Esta subseção visa discorrer sobre os resultados no aspecto da complexidade ciclomática das funções e métodos. A Tabela 4.13 dispõe os resultados para os métodos em projetos desenvolvidos em *Python*. Analisando as informações, é possível perceber que para projetos *serverless*, o *streamalert* obteve os maiores valores, exceto para a média, lugar que ocupou o *domain-protect*, com o valor de 6,53. Sob a ótica dos menores índices, percebe-se que o projeto *aws-serverless-shopping* apresentou os valores menos representativos em todas as categorias.

No que se relaciona aos projetos monolíticos, pode-se enfatizar que, em relação à média dos dados obtidos, a maior taxa foi obtida pelo *highlowbackend*. Na coluna valor mínimo, a quantidade mais representativa foi obtida pelo *monolith-filemanager*. Logo após, nas colunas de valor máximo e versão final, o projeto *django-data-browser* apresentou maior destaque. Contudo, o repositório *Finance-Full-Stack-Web-App-using-Flask-and-SQL* apresentou os menores índices em todas as métricas analisadas.

Tabela 4.13: Complexidade ciclomática dos métodos em projetos *Python*

Projeto	Média	Máximo	Mínimo	N° final	Tipo
Kindergarten-Backend	1,59	73	3	73	*serverless*
aws-serverless-ecommerce	3,42	167	3	57	*serverless*
aws-serverless-shopping	**1,0**	**2**	**2**	**2**	*serverless*
streamalert	1,85	**4237**	**179**	**4237**	*serverless*
domain-protect	**6,53**	99	18	18	*serverless*
Finance-Full-Stack-Web-App	**1,29**	**8**	**4**	**8**	monolítico
highlowbackend	**2,55**	529	80	525	monolítico
GoOutSafe	1,54	69	6	69	monolítico
monolith-filemanager	1,39	477	**372**	477	monolítico
django-data-browser	2,31	**649**	112	**622**	monolítico

Já o resultado para os métodos em *JavaScript* estão dispostos na Tabela 4.14, na qual é possível visualizar que 4 colunas estão com o símbolo -, indicando que para aquele projeto, por não haver métodos, não havia complexidade ciclomática para ser calculada. Diante disso, no caso *serverless*, os maiores valores foram obtidos pelo projeto *serverless-survey-forms*, exceto na média, na qual o projeto *claudia-bot-builder* obteve valor superior. Por outro lado, o projeto *chabelitas-back* apresentou os menores valores em todas as categorias analisadas. Ainda nesse contexto, no que se refere aos projetos monolíticos, o repositório *bulletproof-vanilla-nodejs* obteve os maiores valores em todas as métricas e o *auction-website-monolith* os menores índices, exceto na coluna de média, na qual essa ordem foi invertida.

Com os resultados encontrados sobre a complexidade ciclomática dos métodos, percebe-se que os projetos monolíticos escritos em *Python* obtiveram reduções na categoria de média e valor máximo, sendo de 36,90% e 62,17%, respectivamente, já que a média nos projetos monolíticos foi de 1,82 e em *serverless*, foi de 2,88, já o valor máximo obteve uma taxa média de 346,4 nas aplicações monolíticas e de 915,6 nas aplicações *serverless*. Contudo, apresentaram crescimento de 180% na métrica de valor mínimo, comparado

aos projetos *serverless* desenvolvidos na mesma linguagem, uma vez que os monolíticos apresentaram uma taxa média de 114,8 e os projetos serverless, o índice de 41.

Com relação a *JavaScript*, os projetos monolíticos apresentaram os menores valores em todas as categorias, uma vez que a média obtida nos projetos *serverless* foi de 1,42 e nos monolíticos, o valor de 0,65, representando uma redução de 54,21%. Para o valor máximo, a taxa média nos projetos monolíticos foi de 6 e nos projetos *serverless*, esse índice foi de 368,4, representando redução de 98,37% no valor máximo. No valor mínimo, os projetos monolíticos apresentaram uma taxa média de 4,6, contra o índice de 22,2 apresentado nos projetos *serverless*, o que representa redução de 79,28% no valor mínimo.

Tabela 4.14: Complexidade ciclomática dos métodos em projetos *JavaScript*

Projeto	Média	Máximo	Mínimo	N° final	Tipo
chabelitas-back	**1,00**	**9**	**1**	**7**	*serverless*
moonMail	1,30	454	46	207	*serverless*
serverless-survey-forms	1,78	**756**	**54**	**756**	*serverless*
ruuvi-network-serverless	-	-	-	-	*serverless*
claudia-bot-builder	**3,04**	623	10	623	*serverless*
auction-website-monolith	**1,83**	**9**	**4**	**9**	monolítico
bulletproof-vanilla-nodejs	**1,43**	**21**	**19**	**21**	monolítico
nodejs-shopping-cart	-	-	-	-	monolítico
yelpCamp	-	-	-	-	monolítico
theChaotic	-	-	-	-	monolítico

O gráfico da Figura 4.11 ilustra a média da complexidade ciclomática dos métodos em função dos projetos. Com essa representação, é notável que os projetos desenvolvidos em *Python* tiveram os maiores índices, sendo o *serverless*, com os maiores valores. Nos projetos *JavaScript*, houve a presença de 4 deles sem nenhum método, e portanto, não apareceram no gráfico.

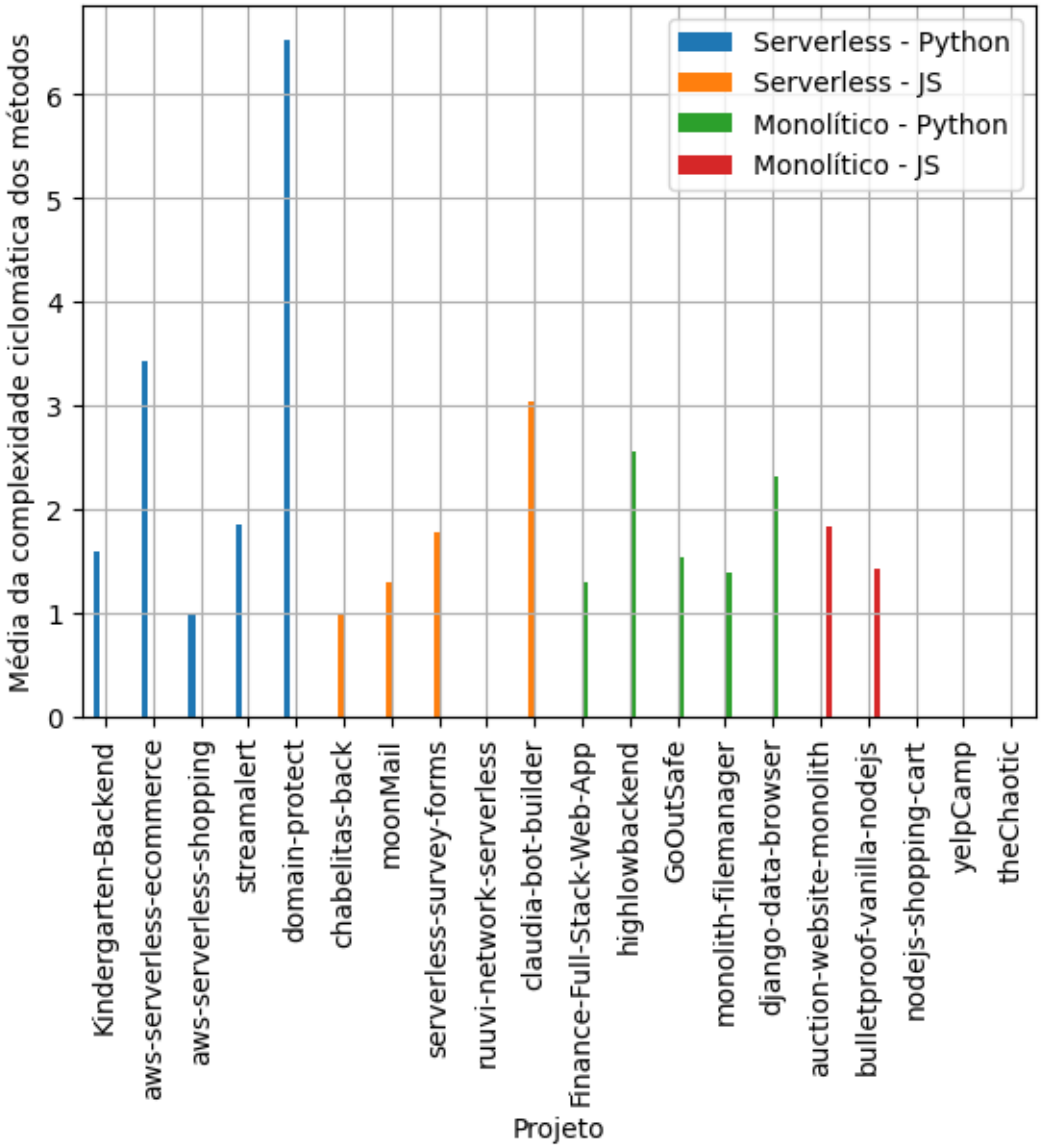

Figura 4.11: Média da complexidade ciclomática dos métodos em função dos projetos.

Expandindo a análise para o âmbito das funções, a Tabela 4.15 dispõe os resultados encontrados para os projetos desenvolvidos em *Python*. No caso dos projetos *serverless*, é possível enfatizar que os maiores valores nas colunas de valor máximo e versão final foram obtidos pelo *aws-serverless-ecommerce*. Com relação à média, o destaque foi para o repositório *domain-protect* e na coluna de valor mínimo, para o *aws-serverless-shopping*. Por outro lado, os menores valores foram obtidos pelo *aws-serverless-shopping* na coluna de máximo e versão final e nas métricas de média e valor mínimo, o projeto *Kindergarten-Backend* obteve os valores menos expressivos.

Com relação aos monolíticos, o representante com maior valor nas colunas de valor máximo e versão final foi o repositório *GoOutSafe*. Já nas colunas de valor mínimo

e versão final, o projeto *django-data-browser* obteve os maiores índices. Contudo, no que diz respeito aos menores valores, o projeto *monolith-filemanager* obteve as menores taxas nas colunas de valor máximo e versão final. Considerando a média dos dados, o repositório *django-data-browser* apresentou o menor índice. Por fim, na coluna de valor mínimo, o projeto *highlowbackend* obteve menor valor.

Tabela 4.15: Complexidade ciclomática das funções em projetos *Python*

Projeto	Média	Máximo	Mínimo	N° final	Tipo
Kindergarten-Backend	**1,76**	235	**8**	235	*serverless*
aws-serverless-ecommerce	3,03	**1358**	37	**1329**	*serverless*
aws-serverless-shopping	3,08	**57**	**55**	**56**	*serverless*
streamalert	2,38	976	44	971	*serverless*
domain-protect	**4,86**	678	41	672	*serverless*
Finance-Full-Stack-Web-App	2,75	58	46	54	monolítico
highlowbackend	2,35	253	**3**	253	monolítico
GoOutSafe	2,78	**867**	25	**866**	monolítico
monolith-filemanager	**1,73**	**22**	8	**22**	monolítico
django-data-browser	**2,88**	743	**72**	743	monolítico

Já para o caso dos projetos desenvolvidos em *JavaScript*, a Tabela 4.16 traz as informações relacionadas. Diante disso, é notório que nos projetos *serverless*, os maiores valores foram obtidos pelo *moonMail* nas colunas de máximo e final, pelo *ruuvi-network-serverless* na média e o *chabelitas-back* com maior valor mínimo, ele também foi responsável pelo menor valor máximo e final. Além disso, o *moonMail* obteve a menor média e o *serverless-survey-forms* o menor número mínimo.

No entanto, na abordagem relacionada aos monolíticos, a menor média foi de 1,46 do *bulletproof-vanilla-nodejs*, contra 2,53 do *yelpCamp*. A coluna de número máximo obteve como extremos 775, do *auction-website-monolith*, e 79, do *nodejs-shopping-cart*. Além disso, na coluna de versão final esses extremos se mantiveram, com exceção do valor de 775 que se reduziu para 769. Na categoria de valor mínimo, o menor valor obtido foi 1, pelo projeto *theChaotic* e o maior de 78, pelo *bulletproof-vanilla-nodejs*.

Com os dados obtidos, percebe-se que para os projetos desenvolvidos em *Python*, os monolíticos obtiveram redução de 17,34% na média, uma vez que os projetos *serverless* apresentaram uma taxa média de 3,02 e os projetos monolíticos o índice de 2,50. Além disso, houve redução de 41,20% no número máximo, com os monolíticos apresentando taxa média de 388,6 e os projetos *serverless*, o valor de 660,8. Também ocorreu uma redução

de 16,76% no número mínimo, já que a taxa média dessa métrica nos monolíticos foi de 30,8 e nos projetos *serverless* apresentou o valor de 37. Para *JavaScript*, houve redução nos monolíticos de 85,30% no valor máximo, uma vez que para essa categoria, a taxa média apresentou o valor de 313,2, contra o índice de 2130,6 apresentado em *serverless*. Além disso, houve redução de 22,69% no valor mínimo, haja vista que nos monolíticos a taxa média nessa métrica foi de 18,4 e nos projetos *serverless*, foi obtido o valor de 23,8. Contudo, ocorreu um aumento de 13,20% na média, uma vez que nos projetos monolíticos a taxa média dessa métrica foi de 2,04 e em *serverless*, apresentou o índice de 1,8.

Tabela 4.16: Complexidade ciclomática das funções em projetos *JavaScript*

Projeto	Média	Máximo	Mínimo	N° final	Tipo
chabelitas-back	2,18	**301**	**44**	**301**	*serverless*
moonMail	**1,13**	**4513**	17	**3039**	*serverless*
serverless-survey-forms	1,51	2646	**1**	2646	*serverless*
ruuvi-network-serverless	**2,83**	871	39	871	*serverless*
claudia-bot-builder	1,37	2322	18	2322	*serverless*
auction-website-monolith	1,65	**775**	2	**769**	monolítico
bulletproof-vanilla-nodejs	**1,46**	92	**78**	92	monolítico
nodejs-shopping-cart	1,66	**79**	6	**79**	monolítico
yelpCamp	**2,53**	429	5	424	monolítico
theChaotic	2,91	191	**1**	187	monolítico

O gráfico da Figura 4.12 ilustra a média da complexidade ciclomática das funções em relação aos projetos. Os dados mostram que os projetos *serverless* em *Python* obtiveram as maiores taxas, porém os monolíticos em *JavaScript* tiveram um aumento, comparado ao *serverless*.

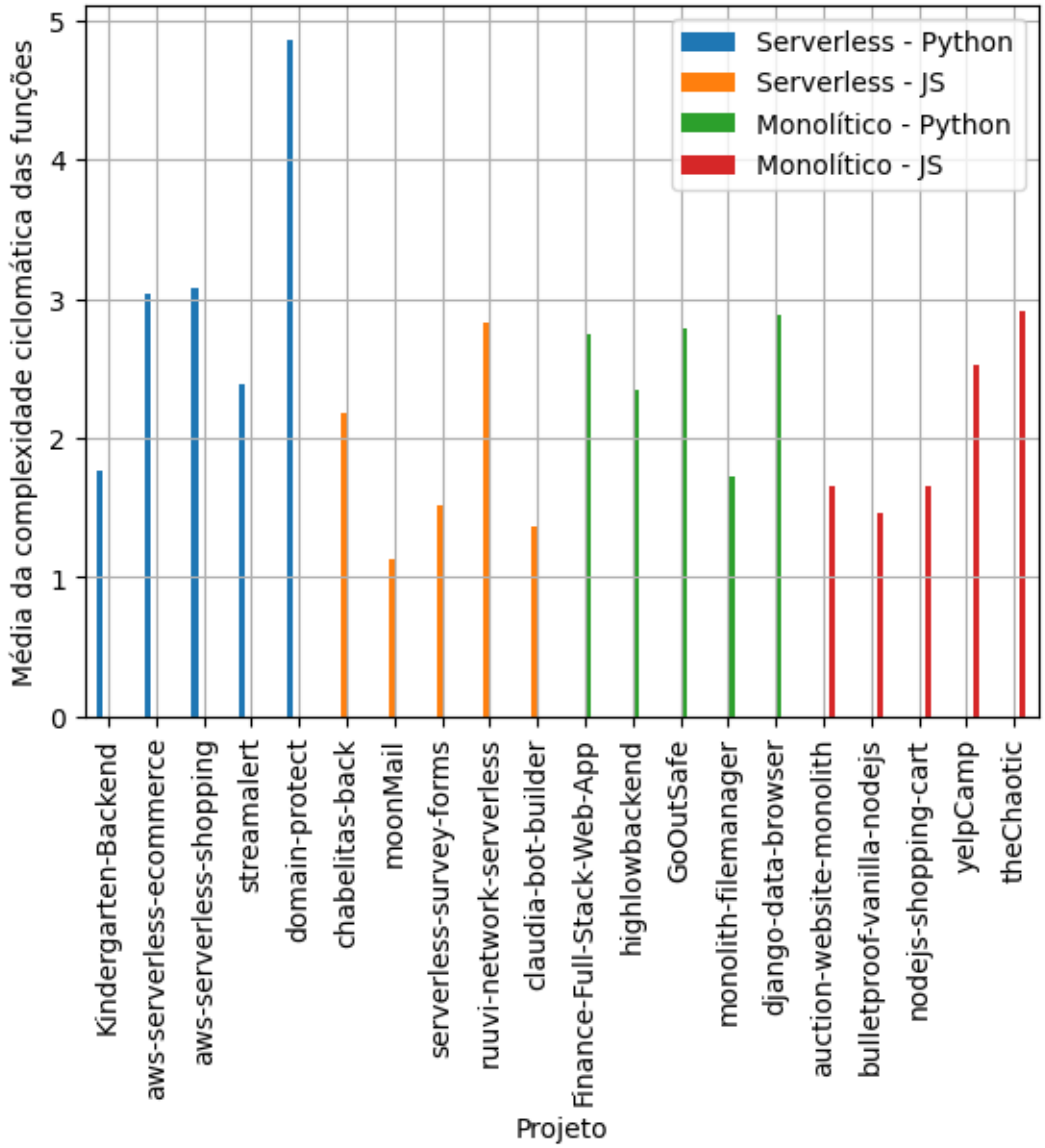

Figura 4.12: Média da complexidade ciclomática das funções em relação aos projetos.

Linhas

Esta subseção visa discorrer sobre os resultados encontrados no aspecto do número de linhas para as funções e métodos. A Tabela 4.17 dispõe os dados para os métodos em projetos desenvolvidos em *Python*. Com as informações obtidas, pode-se verificar que para os projetos *serverless*, o *streamalert* obteve os maiores valores em todas as categorias, exceto na média, valor que foi superior no projeto *domain-protect*. Contudo, as menores taxas foram obtidas pelo *aws-serverless-shopping* em todas as métricas analisadas.

No que se refere aos projetos monolíticos, os maiores índices foram obtidos pelo projeto *highlowbackend*, exceto para o número mínimo, cujo valor mais expressivo foi apresentado pelo *monolith-filemanager*. Já os menores valores foram obtidos pelo repositório

Finance-Full-Stack-Web-App-using-Flask-and-SQL, exceto o menor valor mínimo, o qual responsável foi o projeto *GoOutSafe*.

Tabela 4.17: Número de linhas dos métodos em projetos *Python*

Projeto	Média	Máximo	Mínimo	N° final	Tipo
Kindergarten-Backend	8,10	371	13	365	*serverless*
aws-serverless-ecommerce	15,24	785	6	274	*serverless*
aws-serverless-shopping	**1,0**	**2**	**2**	**2**	*serverless*
streamalert	15,07	**33653**	**1713**	**33653**	*serverless*
domain-protect	**20,98**	325	43	43	*serverless*
Finance-Full-Stack-Web-App	**2,83**	**16**	16	**16**	monolítico
highlowbackend	**22,31**	**3826**	705	**3826**	monolítico
GoOutSafe	4,23	181	**6**	181	monolítico
monolith-filemanager	8,85	2934	**2459**	2934	monolítico
django-data-browser	2,85	944	140	944	monolítico

Já para *JavaScript*, a Tabela 4.18 ilustra os resultados encontrados. Nas aplicações *serverless*, o projeto *serverless-survey-forms* obteve os maiores índices em todas as colunas. Em contrapartida do *chabelitas-back*, com os menores. Já para as monolíticas, o projeto *auction-website-monolith* obteve os menores índices e o *bulletproof-vanilla-nodejs* os superiores. O projeto *ruuvi-network-serverless* e os 3 últimos monolíticos, possuem o símbolo -, pelo fato de não possuírem nenhum método.

Com as informações obtidas, é possível observar que nos projetos desenvolvidos em *Python*, os monolíticos obtiveram uma redução de 31,99% na média, uma vez que apresentaram uma taxa média de 8,21, frente ao valor de 12,08 obtido pelos projetos *serverless*. Além disso, também houve redução 77,51% no valor máximo, uma vez que nos monolíticos essa taxa média foi de 1580,2 e em *serverless*, apresentou o valor de 7027,2. Contudo, no valor mínimo, os monolíticos apresentaram um índice médio de 665,2 e os projetos *serverless*, a taxa de 355,4, representando um crescimento de 87,17% no valor mínimo. Em *JavaScript*, nos monolíticos houve redução 54,25% na média, uma vez que eles apresentaram uma taxa média de 2,71 e os projetos *serverless*, o valor de 5,93. A taxa média no valor máximo nos projetos monolíticos foi de 28,8 e nos projetos *serverless* esse índice foi de 1743,8, o que representou uma redução de 98,35% no valor máximo em projetos monolíticos. Além disso, houve redução de 78,21% no valor mínimo dos monolíticos, uma vez que os projetos *serverless* apresentaram uma taxa média de 107,4 nesta categoria e os projetos monolíticos obtiveram o valor de 23,4.

Tabela 4.18: Número de linhas dos métodos em projetos *JavaScript*

Projeto	Média	Máximo	Mínimo	N° final	Tipo
chabelitas-back	**1,02**	**15**	**1**	**15**	*serverless*
moonMail	4,83	1590	113	651	*serverless*
serverless-survey-forms	**13,80**	**5270**	**375**	**5270**	*serverless*
ruuvi-network-serverless	-	-	-	-	*serverless*
claudia-bot-builder	9,99	1844	48	1844	*serverless*
auction-website-monolith	**5,66**	**28**	**12**	**28**	monolítico
bulletproof-vanilla-nodejs	**7,90**	**116**	**105**	**116**	monolítico
nodejs-shopping-cart	-	-	-	-	monolítico
yelpCamp	-	-	-	-	monolítico
theChaotic	-	-	-	-	monolítico

O gráfico da Figura 4.13 representa a média de linhas dos métodos em função dos projetos. Analisando as informações, percebe-se que os projetos *serverless* obtiveram os maiores índices, com exceção do projeto *highlowbackend*, que é monolítico e obteve o maior valor geral. Por outro lado, 4 repositórios *JavaScript* não tiveram a presença de nenhum método, portanto não apareceram no gráfico.

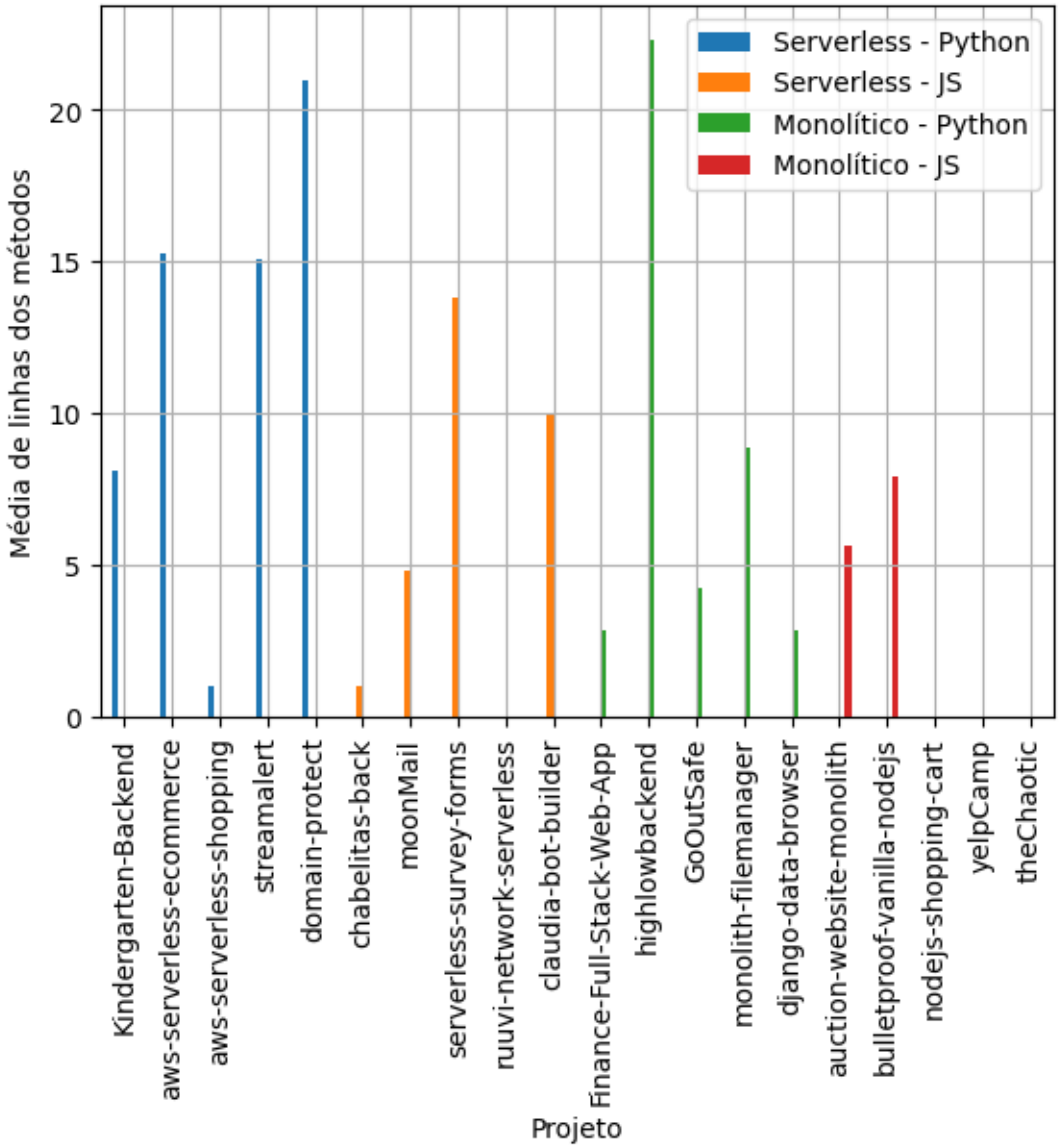

Figura 4.13: Média de linhas dos métodos em função dos projetos.

Para o cenário das funções, a Tabela 4.19 dispõe os resultados para os projetos escritos em *Python*. Segundo as informações, é notável que, para *serverless*, os maiores valores vieram do projeto *aws-serverless-shopping* nas colunas de média e valor mínimo, já os de máximo e final, o projeto *streamalert* foi o responsável pelos maiores índices. Contudo, observando as menores taxas obtidas, é possível salientar que nas colunas de valor máximo e versão final, o repositório *aws-serverless-shopping* obteve os menores índices. No que se refere à média dos dados, o projeto *Kindergarten-Backend* foi responsável pelo valor menos representativo, com o projeto *aws-serverless-ecommerce* na coluna de valor mínimo.

Expandindo a análise para os monolíticos, é possível inferir que nas métricas

de média e valor mínimo, o projeto *Finance-Full-Stack-Web-App-using-Flask-and-SQL* obteve os índices mais representativos. Além disso, nas colunas de valor máximo e versão final, o repositório *GoOutSafe* foi o responsável pelos maiores valores, contrastando com o projeto *monolith-filemanager* que apresentou os menores índices nessas duas métricas. Prosseguindo a análise dos menores índices, pode-se verificar que com relação à média o projeto *GoOutSafe* foi o representante que obteve menor valor e o projeto *highlowbackend* apresentou o menor índice na coluna de valor mínimo.

Tabela 4.19: Número de linhas das funções em projetos *Python*

Projeto	Média	Máximo	Mínimo	N° final	Tipo
Kindergarten-Backend	**6,92**	883	88	883	*serverless*
aws-serverless-ecommerce	16,34	7568	**29**	7433	*serverless*
aws-serverless-shopping	**26,41**	**500**	**460**	**480**	*serverless*
streamalert	21,83	**7653**	341	**7511**	*serverless*
domain-protect	22,26	3418	113	3393	*serverless*
Finance-Full-Stack-Web-App	**19,22**	399	**382**	395	monolítico
highlowbackend	14,79	1464	**12**	1464	monolítico
GoOutSafe	**6,11**	**2175**	109	**2175**	monolítico
monolith-filemanager	10,08	**134**	36	**134**	monolítico
django-data-browser	7,27	1836	166	1836	monolítico

Para os projetos desenvolvidos em *JavaScript*, a Tabela 4.20 dispõe as informações encontradas. Diante dos dados, é possível verificar que para as aplicações *serverless*, o projeto *chabelitas-back* obteve o maior valor pa,ra a média e linhas mínimas, contudo, apresentou os menores valores nas colunas de máximo e versão final. Além disso, o projeto *moonMail* obteve a menor média, mas o maior índice no número máximo. Por fim, o projeto *serverless-survey-forms* apresentou o menor valor mínimo, mas o maior índice na categoria versão final.

Somado a isso, para os monolíticos, o repositório *nodejs-shopping-cart* obteve os menores valores nas colunas média, máximo e versão final. O menor valor mínimo foi apresentado pelo *auction-website-monolith*, com 0 linhas. Este mesmo repositório trouxe os maiores índices na categoria de máximo e versão final. Além disso, a maior média foi de 16,39, do projeto *theChaotic* e 626 linhas mínimas no projeto *bulletproof-vanilla-nodejs*.

Com os dados obtidos, percebe-se que nos projetos desenvolvidos em *Python*, houve redução de 38,70% na média nos monolíticos, haja vista que eles apresentaram uma taxa média de 11,49, frente ao valor de 18,75 apresentado nos projetos *serverless*.

Ademais, no valor máximo, os monolíticos apresentaram o índice médio de 1201,6 e os projetos *serverless*, obtiveram o valor de 4004,4, o que representou uma redução de 69,99% no valor máximo dos monolíticos. Somado a isso, a redução no valor mínimo foi de 31,62%, haja vista que os monolíticos apresentaram a taxa média de 141 e os projetos *serverless*, o valor de 206,2. Já para *JavaScript*, os projetos monolíticos tiveram redução de 33,49% na média, pois apresentaram uma taxa média de 12,28 e os projetos *serverless* obtiveram o valor de 18,46. No que se refere ao valor máximo, os projetos monolíticos obtiveram a taxa média de 2075, frente ao valor de 21037 obtido nas aplicações *serverless*, mostrando uma redução de 90,14% no número máximo para os monolíticos. Além disso, houve redução de 29,00% no valor mínimo nos projetos monolíticos, haja vista que eles apresentaram uma taxa média de 132,2 e as aplicações *serverless* obtiveram o valor de 186,2.

Tabela 4.20: Número de linhas das funções em projetos *JavaScript*

Projeto	Média	Máximo	Mínimo	N° final	Tipo
chabelitas-back	**29,33**	**4017**	**544**	**4017**	*serverless*
moonMail	**8,92**	**39382**	62	27951	*serverless*
serverless-survey-forms	17,50	30767	**4**	**30767**	*serverless*
ruuvi-network-serverless	21,49	7150	217	7150	*serverless*
claudia-bot-builder	15,05	23869	104	23869	*serverless*
auction-website-monolith	13,18	**6203**	**0**	**6155**	monolítico
bulletproof-vanilla-nodejs	12,36	806	**626**	806	monolítico
nodejs-shopping-cart	**8,21**	**410**	16	**410**	monolítico
yelpCamp	11,24	1871	14	1865	monolítico
theChaotic	**16,39**	1085	5	1079	monolítico

O gráfico da Figura 4.14 ilustra a média de linhas das funções em relação aos projetos. Com a observação dos dados, é notório que os projetos *serverless* obtiveram as maiores taxas em ambas as linguagens.

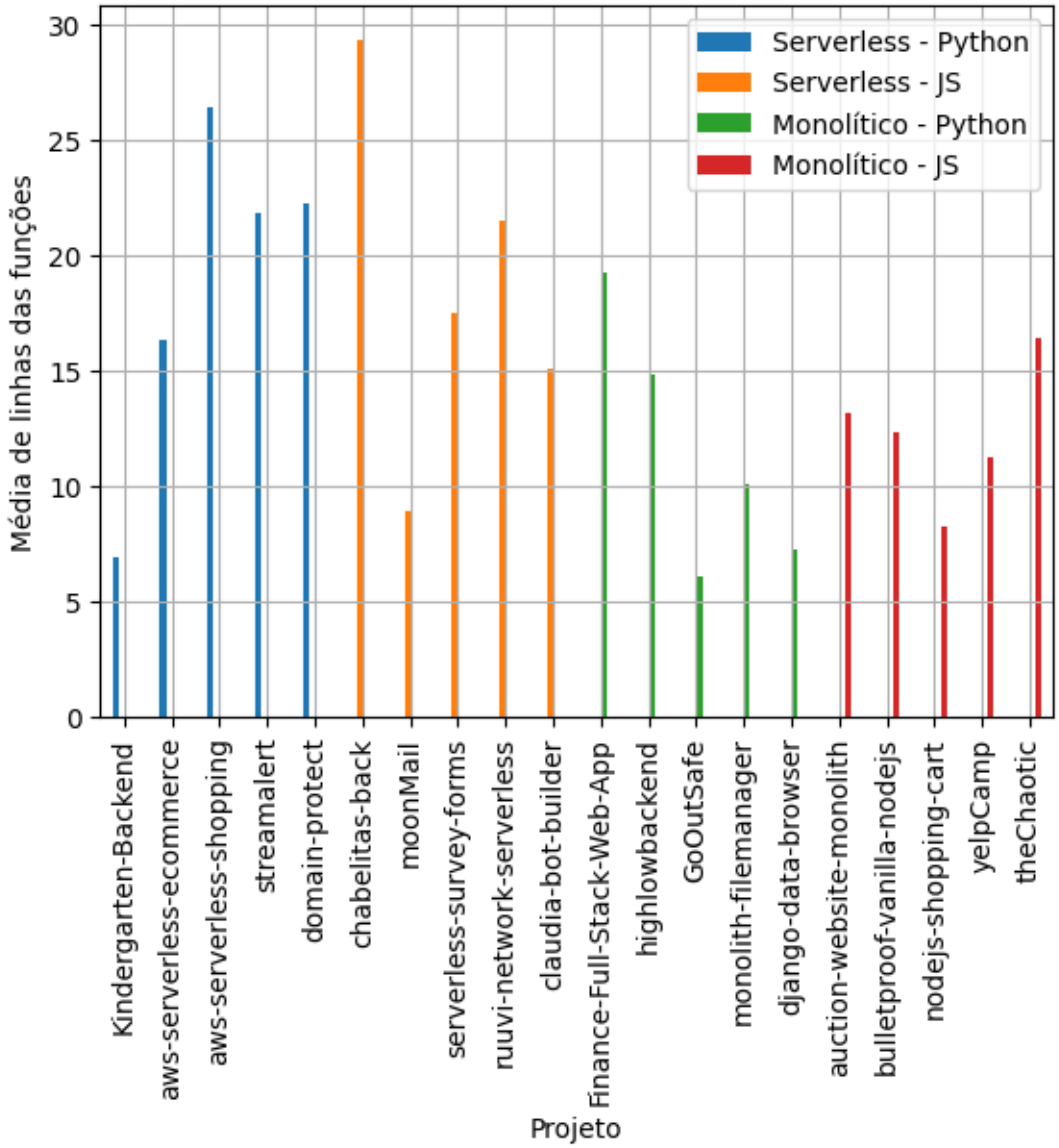

Figura 4.14: Média de linhas das funções em relação aos projetos.

Conclusão

Com os resultados obtidos, foi possível verificar que para os métodos, em projetos *serverless* todos os valores foram superiores, tanto na complexidade ciclomática, quanto no número de linhas em ambas as linguagens. Com relação às funções, o mesmo cenário se repete, com exceção da complexidade ciclomática dos projetos desenvolvidos em *JavaScript*, que concebeu um aumento de 13,19% nos monolíticos.

4.4 Discussão dos resultados

Com base nas informações obtidas ao longo dos experimentos, essa seção visa discorrer a respeito dos resultados encontrados, inferindo possíveis interpretações, utilizando como base as três questões de pesquisa.

Inicialmente, foi possível observar que o número de arquivos em aplicações *serverless* foi superior em ambas as linguagens, apresentando crescimento na média de 44,29% em projetos *Python* e de 178,31% nos projetos escritos em *JavaScript*.

Uma possível explicação para esse comportamento é o fato da granularidade desse tipo de aplicação, na qual cada função é executada de forma independente e sem armazenamento de estado, tendendo a aumentar a divisão de componentes e, consequentemente, aumentar a necessidade de se criar arquivos. Outro ponto observável é que esse número foi ainda maior nos projetos *serverless* em *JavaScript*.

No que se refere aos aspectos físicos e sintáticos dos arquivos, foi possível analisar que, o uso de classes e métodos foi superior em aplicações desenvolvidas em *Python*, principalmente as monolíticas. Em *JavaScript*, esse valor foi sutilmente superior nas aplicações *serverless*, porque houve 3 projetos monolíticos que não apresentaram sequer nenhuma classe ou método. Uma possível observação para esse aspecto é que aplicações *serverless* tendem a ser menos acopladas, a ideia de classes e métodos pode ter sido reduzida, haja vista a menor capacidade de reutilização. Além disso, o paradigma de orientação a objetos se mostrou mais utilizado em projetos desenvolvidos em *Python*.

Outro ponto relevante é que o número de linhas dos arquivos, bem como a complexidade ciclomática e o número de linhas das funções e dos métodos foi superior em aplicações *serverless* em ambas as linguagens, exceto para a complexidade ciclomática das funções dos projetos escritos em *JavaScript*. Uma inferência sobre esse resultado é que, pelo fato das aplicações *serverless* terem responsabilidades mais contidas e individuais, a tendência dos arquivos é de serem maiores, a fim de agrupar todas as responsabilidades necessárias.

4.5 Ameaças à validade

Embora o estudo proposto tenha o intuito de seguir uma metodologia científica e minimizar os erros, alguns fatores podem ter influenciado nos resultados descritos. Nesta seção, são abordados os potenciais pontos que podem ameaçar a validade deste trabalho.

O primeiro ponto se trata de que as ferramentas de métricas utilizadas para realizar a extração dos dados foram diferentes para as duas linguagens. Embora ambas descrevessem que implementavam as mesmas funcionalidades, alguma delas pode ter alguma divergência que não foi conhecida no cálculo das métricas.

Além disso, o estudo se baseou na aplicação de coleta em 5 repositórios *serverless* e monolíticos nas linguagens *Python* e *JavaScript*. Contudo, havia diferenças no número de *commits* dos repositórios, principalmente nos monolíticos escritos em *JavaScript*, que tiveram os menores valores, devido ao número reduzido encontrado desse tipo de projeto.

E por fim, durante a implementação, a coleta de métricas só utilizou a linguagem de programação que era majoritária naquele projeto, ou seja, projetos que eram majoritariamente escritos em *Python*, mesmo que houvesse uma parcela escrita em *JavaScript*, ela não foi considerada.

5 Conclusão

Este estudo buscou entender quais são os impactos na construção de um código com a adoção da computação *serverless*. Para atingir esse objetivo, foi realizado um estudo visando responder às três questões de pesquisa mapeadas com 20 repositórios de código *open source*, divididos em 4 categorias de 5 projetos, sendo eles, monolíticos desenvolvidos em *JavaScript*, monolíticos desenvolvidos em *Python*, *serverless* escritos em *JavaScript* e *serverless* escritos em *Python*.

Diante disso, após a construção de uma ferramenta de extração de métricas de código para as duas linguagens, foi possível observar na QP1 (Seção 4.3.1) que o número de arquivos em aplicações *serverless* foi superior em ambas as linguagens de programação. Na QP2 (Seção 4.3.2) foi possível observar que o uso de classes e métodos foi superior em *Python*, principalmente em aplicações monolíticas. Com relação ao número de linhas, o valor obtido também foi mais expressivo nas aplicações *serverless*. Somado a isso, na QP3 (Seção 4.3.3), foi possível observar que o número de linhas das funções e métodos, bem como a complexidade ciclomática tendem a ser superiores nas aplicações serverless.

Portanto, foi possível concluir que a adoção da computação *serverless*, apesar de trazer inúmeros benefícios como a redução de custos com infraestrutura e a capacidade de entregar soluções de forma mais rápida, pode levar a construção de um código um pouco mais difícil de ser mantido, com um número superior de arquivos e que possuem uma média de linhas mais elevada, além disso, houve a presença de funções e métodos com complexidade ciclomática mais expressiva.

Como principais contribuições do trabalho pode-se elencar:

- Uma ferramenta para extração de métricas de código para *JavaScript* e *Python*, com integração com banco de dados para futura exploração dos dados.
- Os resultados encontrados durante a aplicação da ferramenta desenvolvida em 20 repositórios *open source*.

5.1 Trabalhos futuros

Para expandir as conclusões aqui apresentadas, novas pesquisas podem ser realizadas utilizando variáveis diferentes. Um possível trabalho futuro é realizar a aplicação da ferramenta desenvolvida para extração das métricas não somente na linguagem de programação majoritária, mas também nos arquivos das outras linguagens de programação presentes no repositório. Além disso, é possível expandir o número de repositórios para aplicação da pesquisa, pois apenas 5 representantes de cada categoria foram analisados. Outra oportunidade é expandir o leque de linguagens de programação, analisando outras além de *JavaScript* e *Python*.

Bibliografia

ADZIC, G.; CHATLEY, R. Serverless computing: economic and architectural impact. In: *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. [S.l.: s.n.], 2017. p. 884–889.

ALLEN, F. Control flow analysis. in proceedings of a symposium on compiler optimization. *SIGPLAN Not*, v. 5, n. 7, 1970.

ARMBRUST, M.; FOX, A.; GRIFFITH, R.; JOSEPH, A. D.; KATZ, R. H.; KONWINSKI, A.; LEE, G.; PATTERSON, D. A.; RABKIN, A.; STOICA, I. et al. *Above the clouds: A berkeley view of cloud computing*. [S.l.], 2009.

ASWINI, S.; YAZHINI, M. An assessment framework of routing complexities using loc metrics. *2017 Innovations in Power and Advanced Computing Technologies (i-PACT)*, IEEE, p. 1–6, 2017.

AWS. *Recursos do AWS Lambda*. 2022. Disponível em: ⟨https://aws.amazon.com/pt/lambda/features/⟩.

BAJAJ, D.; BHARTI, U.; GOEL, A.; GUPTA, S. Partial migration for re-architecting a cloud native monolithic application into microservices and faas. In: SPRINGER. *International Conference on Information, Communication and Computing Technology*. [S.l.], 2020. p. 111–124.

CLOUD, G. *Ambiente de execução do Cloud Functions*. 2022. Disponível em: ⟨https://cloud.google.com/functions/docs/concepts/exec?hl=pt_br#functions-concepts-after-timeout-nodejs⟩.

COMMITTEE, I. C. S. S. E. T. *IEEE Standard Glossary of Software Engineering Terminology*. [S.l.]: IEEE, 1983. v. 729.

FAN, C.-F.; JINDAL, A.; GERNDT, M. Microservices vs serverless: A performance comparison on a cloud-native web application. In: *CLOSER*. [S.l.: s.n.], 2020. p. 204–215.

FENTON, N.; BIEMAN, J. *Software metrics: a rigorous and practical approach*. [S.l.]: CRC press, 2014.

GHOSH, B. C.; ADDYA, S. K.; SOMY, N. B.; NATH, S. B.; CHAKRABORTY, S.; GHOSH, S. K. Caching techniques to improve latency in serverless architectures. In: IEEE. *2020 International Conference on COMmunication Systems & NETworkS (COMSNETS)*. [S.l.], 2020. p. 666–669.

GOLI, A.; HAJIHASSANI, O.; KHAZAEI, H.; ARDAKANIAN, O.; RASHIDI, M.; DAUPHINEE, T. Migrating from monolithic to serverless: A fintech case study. In: *Companion of the ACM/SPEC International Conference on Performance Engineering*. [S.l.: s.n.], 2020. p. 20–25.

HASSAN, A. E. The road ahead for mining software repositories. In: IEEE. *2008 frontiers of software maintenance*. [S.l.], 2008. p. 48–57.

KAGDI, H.; COLLARD, M. L.; MALETIC, J. I. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of software maintenance and evolution: Research and practice*, Wiley Online Library, v. 19, n. 2, p. 77–131, 2007.

KEMP, S. *DIGITAL 2022: GLOBAL OVERVIEW REPORT*. 2022. Disponível em: ⟨https://datareportal.com/reports/digital-2022-global-overview-report⟩.

KÖHLER, M.; BENKNER, S. Vce-a versatile cloud environment for scientific applications. In: *The Seventh International Conference on Autonomic and Autonomous Systems (ICAS 2011)*. [S.l.: s.n.], 2011. p. 22–27.

KRITIKOS, K.; SKRZYPEK, P. A review of serverless frameworks. In: IEEE. *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. [S.l.], 2018. p. 161–168.

LIU, H.; GONG, X.; LIAO, L.; LI, B. Evaluate how cyclomatic complexity changes in the context of software evolution. In: IEEE. *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*. [S.l.], 2018. v. 2, p. 756–761.

LOPEZ-HERREJON, R. E.; TRUJILLO, S. How complex is my product line? the case for variation point metrics. In: *VaMoS*. [S.l.: s.n.], 2008. p. 97–100.

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

MICROSOFT. *Azure Functions hosting options*. 2022. Disponível em: ⟨https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale⟩.

MOURÃO, E.; PIMENTEL, J. F.; MURTA, L.; KALINOWSKI, M.; MENDES, E.; WOHLIN, C. On the performance of hybrid search strategies for systematic literature reviews in software engineering. *Information and Software Technology*, Elsevier, v. 123, p. 106294, 2020.

NEWMAN, S. *Monolith to microservices: evolutionary patterns to transform your monolith*. [S.l.]: O'Reilly Media, 2019.

PACHGHARE, V. K. Microservices architecture for cloud computing. *architecture*, v. 3, p. 4, 2016.

PETERS, L. *Cloud Computing Trends for 2019*. 2019. Disponível em: ⟨https://www.networksunlimited.com/cloud-computing-trends-for-2019⟩.

SERVERLESS. *Serverless Framework Concepts*. 2022. Disponível em: ⟨https://www.serverless.com/framework/docs/providers/aws/guide/intro⟩.

TAIBI, D.; SPILLNER, J.; WAWRUCH, K. Serverless computing-where are we now, and where are we heading? *IEEE software*, IEEE, v. 38, n. 1, p. 25–31, 2020.

TAIBI, D.; SPILLNER, J.; WAWRUCH, K. Serverless where are we now and where are we heading? *IEEE Software*, v. 1, 01 2021.

TOADER, L.; UTA, A.; MUSAAFIR, A.; IOSUP, A. Graphless: Toward serverless graph processing. In: IEEE. *2019 18th International Symposium on Parallel and Distributed Computing (ISPDC)*. [S.l.], 2019. p. 66–73.

VILLAMIZAR, M.; GARCÉS, O.; CASTRO, H.; VERANO, M.; SALAMANCA, L.; CASALLAS, R.; GIL, S. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In: IEEE. *2015 10th Computing Colombian Conference (10CCC)*. [S.l.], 2015. p. 583–590.

WOHLIN, C. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: *Proceedings of the 18th international conference on evaluation and assessment in software engineering*. [S.l.: s.n.], 2014. p. 1–10.

YAMAMOTO, K.; KONDO, M.; NISHIURA, K.; MIZUNO, O. Which metrics should researchers use to collect repositories: an empirical study. In: IEEE. *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*. [S.l.], 2020. p. 458–466.