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

THAYANNE LUIZA VICTOR LANDIM SOUSA

CUSTOMIZAÇÃO DO CLIENTE HTTP PARA BIBLIOTECA CLOJURE COGNITECT.AWS-API

CAMPINA GRANDE - PB

2023

THAYANNE LUIZA VICTOR LANDIM SOUSA

CUSTOMIZAÇÃO DO CLIENTE HTTP PARA BIBLIOTECA CLOJURE COGNITECT.AWS-API

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

Orientador: Professor Fábio Jorge Almeida Morais.

CAMPINA GRANDE - PB

2023

THAYANNE LUIZA VICTOR LANDIM SOUSA

CUSTOMIZAÇÃO DO CLIENTE HTTP PARA BIBLIOTECA CLOJURE COGNITECT.AWS-API

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

BANCA EXAMINADORA:

Professor Fábio Jorge Almeida Morais

Orientador – UASC/CEEI/UFCG

Professor Marcus Salerno de Aquino

Examinador – UASC/CEEI/UFCG

Professor Tiago Lima Massoni

Professor da Disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em: 14 de Fevereiro de 2023.

CAMPINA GRANDE - PB

ABSTRACT

The cognitect.aws-api is a Clojure library that allows programmatic access to Amazon Web Services (AWS) by using the library cognitect.http-client to make HTTP communications. The cognitect.http-client being the only possibility of an HTTP client restricts users from changing the behavior of requests to be adequate for their use cases or needs, as well as this client has known issues because it uses Jetty in version 9, which does not contain support from the community and is not recommend to be used anymore, and also contains diverse security vulnerabilities reported by users. This work enables a complete customization of the HTTP client used by the library, in a way that users may choose any HTTP client to plug into the library through a simple and public interface abstraction. Additionally, through the customization provided, an alternative HTTP client using the client from the Java 11 native package java.net is now available, thus quickly solving the problems that some users were having with the default HTTP client. None of the work done causes any compatibility-breaking changes that may cause unexpected failures or errors for current users of the library.

CUSTOMIZAÇÃO DO CLIENTE HTTP PARA BIBLIOTECA CLOJURE COGNITECT.AWS-API

Thayanne Luiza Victor Landim Sousa
Universidade Federal de Campina Grande
Campina Grande, Paraíba, Brasil
thayanne.sousa@ccc.ufcg.edu.br

Fábio Jorge Almeida Morais
Universidade Federal de Campina Grande
Campina Grande, Paraíba, Brasil
fabio@computacao.ufcg.edu.br

RESUMO

A *cognitect.aws-api* é uma biblioteca em Clojure que permite o acesso programático aos serviços da Amazon Web Services (AWS) e que usa outra biblioteca chamada *cognitect.http-client* para realizar as comunicações HTTP. A restrição de *cognitect.http-client* ser a única possibilidade de cliente HTTP impossibilita usuários de flexibilizar o comportamento de requisições para adequar aos seus casos de uso ou necessidades, como também, tal cliente possui complicações por utilizar *Jetty* na versão 9 para implementar o cliente, versão esta que não possui mais suporte da comunidade e não é mais recomendada a utilização, além das diversas vulnerabilidades reportadas por usuários. Este trabalho possibilita uma completa customização do cliente HTTP utilizado pela biblioteca, de forma que agora os usuários podem escolher qualquer cliente HTTP para acoplar através de uma abstração simples utilizando uma interface pública disposta na biblioteca. Adicionalmente, através da customização, é disponibilizada uma alternativa pronta de cliente HTTP utilizando o cliente do pacote nativo *java.net* do Java 11, solucionando assim rapidamente os problemas que alguns usuários estavam tendo com o cliente HTTP padrão. Nenhuma das evoluções desenvolvidas causa qualquer quebra de compatibilidade que possa causar falhas ou erros inesperados para atuais usuários da biblioteca.

PALAVRAS-CHAVE

AWS, Clojure, HTTP.

REPOSITÓRIO

https://github.com/thayannevls/aws-api.

1. INTRODUÇÃO

A computação em nuvem é um modelo que provê recursos e serviços de computação através da internet, possibilitando fazer uso de servidores, bancos de dados, softwares, máquinas para processamento de dados e outros recursos sem que seja preciso comprar ou manter um conjunto de recursos computacionais físicos. Organizações e empresas de diferentes tamanhos e portes podem utilizar este modelo sob demanda para construir seus produtos e soluções usando recursos flexíveis e escaláveis, tendo assim menos preocupações e custos para uma infraestrutura própria e ganhando agilidade para desenvolvimento de sua solução ou produto.

Lançada em 2006, a Amazon Web Services (AWS) oferece mais de 200 serviços e recursos voltados para computação em nuvem, tendo sido eleita como a líder do ramo em 2022 pela Gartner [1]. A maioria de suas soluções são disponibilizadas via *Application Programming Interfaces* (Interface de Programação de Aplicação - APIs) Web e uma parte do uso se dá pela utilização de ferramentas e bibliotecas que permitem o uso programático a estas APIs. Uma das bibliotecas de destaque é a *cognitect.aws-api* [1], mantida e disponibilizada pela *Cognitect Labs*[2], que permite o acesso programático e orientado a dados aos serviços da AWS. A biblioteca disponibiliza um pequeno conjunto de funções que os usuários podem utilizar em suas aplicações, implementadas na linguagem de programação de paradigma funcional *Clojure*[3]. Desta forma, as aplicações do usuários passam a ter acesso às APIs da AWS.

O protocolo de comunicação mais utilizado pelas APIs de serviços da AWS é o *Hypertext Transfer Protocol* (HTTP - Protocolo de Transferência de Hipertexto), portanto, tais bibliotecas que oferecem comunicação com serviços da AWS precisam de um cliente HTTP responsável por realizar a comunicação entre a aplicação do usuário e a API do serviço web. O cliente utilizado pela *aws-api* é a biblioteca *cognitect.http-client* [4], que utiliza o cliente do *Jetty* versão 9, projeto mantido pela *Eclipse Foundation*[5], desenvolvido em Java.

Embora esse cliente HTTP possua as funcionalidades básicas desejadas, a comunidade costuma reportar a necessidade de customizações que permitam injetar novas funcionalidades visando a adequação a casos de uso mais específicos. Por exemplo, tratamento de dados da requisição ou monitoramento de métricas do cliente. Além da necessidade de customização, a versão do *Jetty* usada é a 9, versão já considerada legada e sem

[1] https://github.com/cognitect-labs/aws-api
[2] https://cognitect-labs.github.io/
[3] https://clojure.org/
[4] https://mvnrepository.com/artifact/com.cognitect/http-client
[5] https://www.eclipse.org/jetty/

suporte da comunidade, existindo a recomendação de utilizar a versão 10 ou posterior [2]. Ainda sobre o *Jetty*, o projeto e suas dependências apresentaram vulnerabilidades de segurança. Apenas nos anos de 2021 e 2022 foram reportadas 10 vulnerabilidades detectadas [3], que ocasionaram problemas aos usuários que reportam aos mantenedores da *Cognitect*, solicitando urgentemente a atualização para a versão que remove a vulnerabilidade ou requisitando que outro cliente seja usado. Adicionalmente, por uma limitação da implementação da biblioteca *cognitect.http-client*, e não do *Jetty*, dados enviados e recebidos nas requisições HTTP sempre possuem tipo da classe Java *ByteBuffer*, restringido os usuários de utilizarem outros tipos, a exemplo do *InputStream*, que já é utilizado por outros componentes e funções da *aws-api*.

Os motivos citados acima justificam os pedidos da comunidade para que exista outro cliente para se comunicar com os serviços da AWS na biblioteca. Ainda, ressalta-se que não é uma possibilidade apenas trocar o cliente HTTP ou aumentar a versão do *Jetty*, pois pode-se causar mudanças de comportamento ou quebra de compatibilidade da *cognitect.aws-api.* uma vez que é adotada uma política de não lançar versões que causem quebra de compatibilidade. O suporte para customização do cliente HTTP é o ideal para não mudar o comportamento atual da biblioteca, como também permitir que os usuários possam facilmente implementar casos de uso específicos para a comunicação HTTP entre suas aplicações e os serviços da AWS.

Desta forma, este trabalho tem como objetivo evoluir a biblioteca *aws-api* da *Cognitect Labs* para que se permita a utilização de um cliente HTTP além do que é utilizado por padrão pela biblioteca, sem que seja modificado o comportamento atual ou que se cause qualquer tipo de falhas ou erros para os usuários que não desejem modificar o cliente HTTP atual. Como segundo objetivo, e talvez mais impactante a curto prazo, disponibilizar a implementação um cliente HTTP alternativo para ser utilizado desde já pela comunidade, a fim de resolver os atuais problemas com o cliente *Jetty.* O cliente escolhido é o nativo disponibilizado a partir do *Java Net* por apresentar uma solução completa, ter suporte nativo e não possuir dependências além da própria linguagem Java a partir da versão 11.

Na seção 2, detalhamos os problemas do estado atual da biblioteca e a solução proposta. Na seção 3 é apresentada a arquitetura da solução e todos os componentes que a compõem. Na seção 4 é relatado o processo de desenvolvimento e os principais desafios enfrentados ao longo do projeto. Na seção 5, apresentamos os resultados para, em seguida, na seção 6, concluir o artigo comentando sobre limitações e trabalhos futuros que serão feitos.

2. PROBLEMA E SOLUÇÃO

A *cognitect.aws-api* é uma biblioteca Clojure que provê acesso programático e orientado a dados das APIs da AWS, que se comunicam via protocolo HTTP. Para utilizar a biblioteca, basta incluí-la nas dependências do seu projeto, seguir os passos de configurações e utilizar as funções disponíveis para se comunicar com a API ou serviço que desejar da AWS.

Nesta seção, descreveremos os componentes principais que compõem a biblioteca e suas limitações e problemas. Ao fim, discutimos a solução proposta.

2.1 Cliente AWS

Com apenas duas funções, chamadas *client* e *invoke*, é feita a comunicação com API da AWS. Como mencionado anteriormente, a comunicação HTTP é realizada utilizando outra biblioteca da *Cognitect Labs* chamada *cognitect.http-client* que, por fim, utiliza o cliente HTTP *Jetty* versão 9, desenvolvida pela *Eclipse Foundation*.

A função *client* cria uma instância de um cliente AWS, que não é cliente HTTP em si, mas sim um cliente de um serviço ou API AWS, como por exemplo um cliente do *Amazon S3*[6]. Já a função *invoke* realiza operações disponíveis de um cliente AWS, o que pode incluir salvar, consultar, atualizar dados ou qualquer outro tipo de funcionalidade que a AWS ofereça. As duas funções recebem e retornam mapas para os dados, em um formato chave-valor. Adicionalmente, a *invoke* também precisa receber o cliente AWS sendo utilizado.

Na Figura 1, é demonstrado um exemplo na linguagem *Clojure* interagindo com estas duas funções. Na linha 5 do exemplo, podemos ver a função *client* sendo utilizada para criar um cliente para a API S3, onde a entrada é passada dentro de um mapa *("{:api :s3}")*. Logo em seguida, nas linhas 8 e 11, podemos ver invocações de operações sendo executadas com esse cliente, uma de listagem de dados e outra da criação de um dado, respectivamente, com detalhe que na segunda operação podemos ver dados serem passados dentro de um parâmetro do mapa chamado *request*.

Na criação do primeiro cliente HTTP, a biblioteca também cria um cliente HTTP que será utilizado por todos os clientes AWS criados em seguida. Na próxima seção abordaremos essa implementação.

```
1 ; Importa dependênca
2 (require '[cognitect.aws.client.api :as aws)
3
4 ;; Cria um cliente
5 (def cliente-s3 (aws/client {:api :s3}))
6
7 ;; Invoca operação de listagem
8 (aws/invoke s3 {:op :ListBuckets})
9
10 ;; Invoca operação de criação
11 (aws/invoke s3 {:op :CreateBucket
12                 :request {:Bucket "um-bucket"}})
```

Figura 1 - Exemplo de código em Clojure utilizando as funções *client* e *invoke*.

2.2 Cliente HTTP e suas limitações

[6] https://aws.amazon.com/s3/

Por padrão, um único cliente HTTP é utilizado por todos os clientes AWS. Sendo assim, o primeiro cliente AWS a ser criado irá instanciar o cliente da *cognitect.http-client* em uma variável que pode ser acessada pelos componentes internos quando o usuário executa a função *client*. O motivo dessa escolha de design é para economizar recursos, pois, se cada cliente criar uma nova instância de cliente HTTP, pode-se ocasionar que o número de Threads utilizadas para requisições cresça linearmente com o número de clientes AWS sendo criados e operações sendo invocadas.

Discutindo um pouco sobre características do cliente HTTP da Cognitect e sua interação com a biblioteca, o corpo das requisições e respostas precisam ser do tipo da classe Java *ByteBuffer*[7], que representa uma sequência finita de Bytes que consistem os dados serializados. Esse tipo se desalinha com a função *invoke* da *aws-api,* que espera *InputStream*[8] ou *Array* de *Bytes* para dados de operações, e sempre retorna *InputStream* no resultado da invocação[4]. Por conta desta diferença, a biblioteca sempre precisa realizar conversões internas na entrada e saída de dados para o cliente HTTP. Essas conversões ocasionam implicitamente outras limitações, por exemplo, atualmente os dados passados na função *invoke* precisam caber em memória mesmo que sejam do tipo *InputStream.* Funcionalidades como Streaming de dados muito grandes, como *BLOBs (Binary Large Object* - Objeto Grande Binário), são impossibilitadas de serem implementadas por conta dessa limitação.

Como citado anteriormente, *cognitect.http-client* utiliza a versão 9 do *Jetty*, que não possui mais suporte da comunidade e é recomendada a utilização da versão 10 ou posterior [2]. Além disso, diversas vulnerabilidades foram reportadas por usuários [3] ao longo dos anos de 2021 e 2022, que por sua vez, requisitam aos mantenedores da *aws-api* que uma ação seja tomada para a escolha de um cliente HTTP estável e mais confiável, além de possibilitar novas funcionalidades.

Além destes problemas, usuários também reportam a necessidade de customizações na submissão de requisições HTTP, como a coleta e a exposição de métricas, ou modificação de como um cabeçalho é montado na requisição. É notório que tais funcionalidades de customização estão muito mais relacionadas à implementação do cliente HTTP do que o foco principal da *aws-api*, que é o acesso orientado à dados aos serviços da AWS. Implementar e dar suporte para customizações de diferentes tipos para diferentes casos de uso criam uma dificuldade para os mantenedores como também abrangem o escopo da biblioteca, tornando-a mais complexa e difícil de manter.

A princípio, a solução óbvia e direta para o problema de versão parece ser utilizar uma versão mais recente do *Jetty*. É importante observar, no entanto, que *aws-api* não é a única usuária da biblioteca *http-client*, como também, mudar a versão do *Jetty* ocasiona problemas de compatibilidade e mudanças de comportamento inesperadas tanto aos usuários da *http-client* como da *aws-api*. Outra solução com menores efeitos colaterais consiste na criação ou escolha de um novo cliente HTTP para ser utilizado pela biblioteca e remover a dependência da *cognitect.http-client.* Entretanto, essa mudança também causa problemas de compatibilidade para os usuários finais; sendo assim, não é algo que pode ser feito sem o usuário optar por isso.

2.3 Solução

Ao analisarmos que o objetivo principal da *coginitect.aws-api* é o acesso programático e orientado a dados às APIs e serviços da AWS, podemos notar que a comunicação HTTP não é sua funcionalidade, mas sim uma relação de dependência. Sendo assim, não importa qual cliente HTTP seja utilizado, desde que submeta requisições da maneira esperada, a biblioteca funcionará e cumprirá seu papel.

Diante disso, a solução proposta se baseia primeiramente em criar uma separação clara entre a *aws-api* e o cliente HTTP, na qual a relação entre os dois seja descrita por um contrato que descreva comportamentos e parâmetros esperados. Isso significa que todas as referências diretas e limitações impostas pela *coginitect.http-client* serão removidas e, no lugar, será colocado um contrato que atenda melhor às necessidades da biblioteca e dos usuários. O contrato pode ser entendido como uma abstração de interface do que é esperado; aqui, será definido através de um *Protocolo Clojure*[9] com nome de *HttpClient* e com a assinatura dos métodos que serão chamados para realizar interagir com o cliente. É importante destacar que, a fim de cumprir o requisito de não ocasionar problemas de compatibilidade, foram feitas adaptações para que a *cognitect.http-client* atenda ao novo contrato mas mantenha o comportamento atual, sem precisar que qualquer ação seja tomada pelos usuários.

Com a separação clara entre os módulos e o protocolo *HttpClient* público, o usuário consegue, através de uma nova API, definir o cliente global que será utilizado por padrão, como também, definir diferentes clientes HTTP para diferentes instâncias de clientes AWS criados pela biblioteca. Dessa forma, a customização agora é possível e acompanha uma documentação que orienta usuários a como acoplar um novo cliente HTTP e remover *cognitect.http-client* das suas dependências caso esta esteja inutilizada após a customização.

Na Figura 2, é demonstrado como os componentes da biblioteca interagem com o protocolo que abstrai o cliente HTTP. A implementação do protocolo não faz uso direto da biblioteca, podendo ser acoplada pelo usuário utilizando a nova API *set-default-http-client!*. Por padrão, caso o usuário não defina customize o cliente HTTP, a implementação será a *cognitect.http-client*.

[7] https://docs.oracle.com/javase/7/docs/api/java/nio/ByteBuffer.html

[8] https://docs.oracle.com/javase/7/docs/api/java/io/InputStream.html

[9] https://clojure.org/reference/protocols

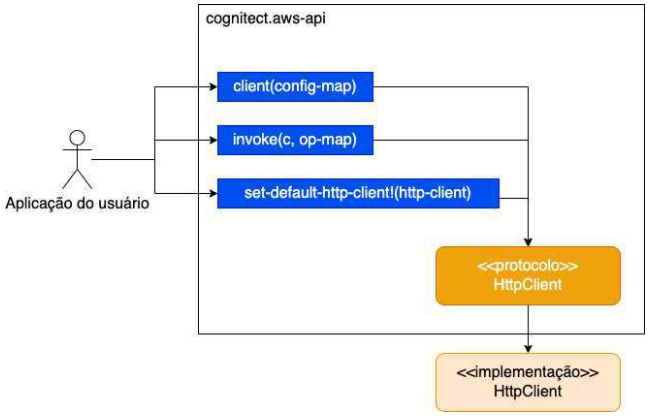


Figura 2 - Diagrama dos componentes da solução.

Adicionalmente, com intuito de atender demandas e resolver os problemas atuais, uma outra parte da solução é a disponibilização de um cliente HTTP seguindo o protocolo já pronto para uso e sem utilizar *Jetty*, por conta dos problemas descritos anteriormente. Além disso, a implementação do protocolo é uma prova de conceito da solução que valida as decisões de design e arquiteturais, e foi de grande auxílio para experimentações durante o projeto. O cliente escolhido é o disponibilizado pelo pacote nativo *java.net*[10] a partir da versão 11 do *Java*, sobre o qual foi feita uma implementação para trazê-lo para o *Clojure* e seguindo o contrato proposto pela biblioteca.

Desta forma, podemos resumir que a solução é composta por três entregas:

1. Protocolo *HttpClient,* que define o contrato entre a cognitect.aws-api e uma implementação qualquer de cliente HTTP.
2. API para modificar o cliente HTTP global, de forma que ofereça uma opção de customizar o cliente utilizado por padrão pela biblioteca.
3. Implementação do protocolo *HttpClient* utilizando o cliente HTTP fornecido pelo pacote nativo do Java chamado *java.net*, oferecendo então uma alternativa de cliente HTTP além da *cognitect.http-client*.

Na próxima seção, são discutidas a arquitetura da solução, as mudanças feitas e a forma como os componentes interagem entre si, acompanhado de exemplos de uso.

3. ARQUITETURA DA SOLUÇÃO

Os componentes da solução foram construídos seguindo o paradigma funcional de programação, compostos por funções que recebem dados e retornam dados relacionados à comunicação com AWS, sem gerenciamentos de estados ou dados mutáveis, com exceção da API criada para sobrescrever o cliente HTTP padrão, o que será explicado mais à frente.

3.1 Visão Geral

A partir da Figura 3, podemos visualizar que um cliente AWS criado pela função *client* é um mapa com chave e valores, contendo informações como o provedor de credenciais e qual serviço da AWS se refere. Uma das informações armazenadas neste mapa, é a chave *:http-client*, que contém como valor uma instância do cliente HTTP que é utilizada para realizar as comunicações do serviço AWS em questão. O cliente é sempre uma implementação do protocolo *HttpClient*, como também demonstrado também na Figura 3.

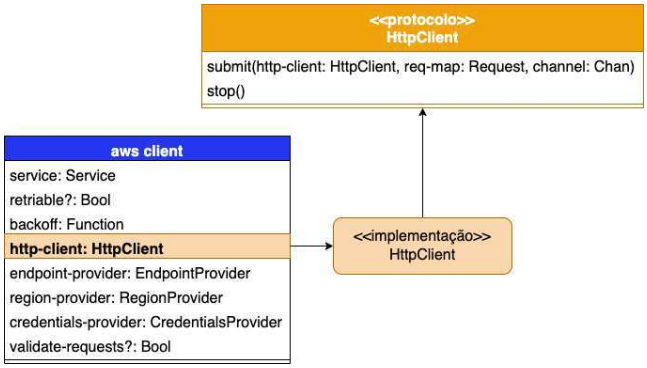


Figura 3 - Mapa do cliente AWS e sua dependência com o Cliente HTTP.

O *http-client* pode ser resolvido de três formas diferentes, dependendo da ordem de execução e dos parâmetros fornecidos durante a criação do cliente AWS. A função *client* pode receber, no seu primeiro parâmetro, que é um mapa, um cliente HTTP. Esta opção já existia na biblioteca anteriormente, porém era apenas para uso interno. Agora, é aberta e documentada, permitindo especificar clientes HTTP isolados entre si, sendo assim sendo possível até usar mais de uma implementação de cliente diferente.

Caso um cliente HTTP não tenha sido provido pelos argumentos da função *client*, será utilizado o cliente HTTP global e compartilhado por padrão entre todas as instâncias de clientes AWS, e aqui existem duas opções de caminhos. Se esse é o primeiro cliente AWS sendo criado no processo atual, um cliente HTTP será criado, salvo em uma variável chamada *shared-http-client* e repassado. Caso contrário, simplesmente é referenciado o cliente já criado que está salvo na variável *shared-http-client*. Na seção 3.3, detalhamos melhor o funcionamento dessa variável e como ela é manipulada. Na Figura 4, podemos visualizar o fluxo de decisão para resolver a chave *http-client* do cliente AWS.

10 https://docs.oracle.com/en/java/javase/12/docs/api/java.net.http/java/net/http/HttpClient.html

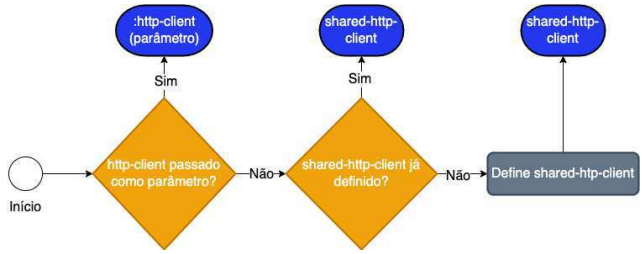


Figura 4 - Fluxo de resolução do valor de :http-client.

Para invocar operações com os clientes AWS, a função *invoke* espera que o contrato do protocolo *HttpClient* descrito na Figura 3 seja cumprido, ou seja, que exista a implementação de duas funções: *submit* e *stop*. A primeira função deve conseguir submeter assincronicamente uma requisição HTTP representada por um mapa com chaves e valores. Já a *stop*, idealmente, deve interromper o cliente HTTP e liberar recursos, como memória e *threads* em execução. Na Figura 5 podemos visualizar que a função *invoke* depende da função *submit* do protocolo, que utiliza a implementação do cliente HTTP. Na seção 3.2, é descrito o esquema dos parâmetros dessas funções e o funcionamento.

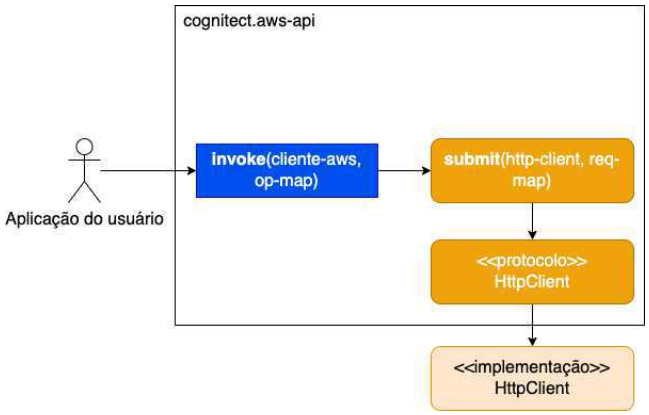


Figura 5 - Relação entre a função invoke da aws-api e a submit do protocolo HttpClient.

Nas seções a seguir descreveremos cada componente com mais detalhes, especificando quais mudanças foram feitas durante o projeto, além de expor sobre o cliente HTTP do *java.net* e API que permite customização do cliente HTTP padrão e compartilhado.

3.2 Protocolo HttpClient

A relação entre os componentes e funções da *cognitect.aws-api* e o cliente HTTP é definida por um contrato de interface utilizando protocolo do *Clojure*, que contém a assinatura de métodos e parâmetros esperados. A documentação também é utilizada como recurso para definir o contrato, já que não existe tipagem.

Vale ressaltar que uma versão do protocolo *HttpClient* já existia dentro da biblioteca, porém apenas para uso interno e sendo moldado de acordo com as limitações e especificações da *cognitect.http-client*. Por exemplo, os dados do corpo das requisições e respostas estavam definidos para serem um tipo da classe *ByteBuffer* do *Java* e outros componentes da biblioteca precisavam transformar esse dado *ByteBuffer* para os tipos esperados pela função *invoke*, que são *InputStream* e Array de *Bytes*.

Como descrito na seção anterior, o protocolo possui duas funções, *submit* para submeter de maneira assíncrona requisições HTTP e *stop* para interromper a execução do cliente e liberar recursos. A função *submit* recebe três parâmetros: o primeiro é o próprio cliente HTTP, o segundo é o mapa da requisição que será submetida e o último é um *channel* que deve receber o resultado da requisição. Em Clojure, *channels* são implementações do *core.async*[11] que permitem a comunicação independente entre processos através de mensagens [5]. Ao receber um *channel*, a biblioteca pode abrir uma nova thread à espera de que uma mensagem com o resultado da requisição chegue ao canal de maneira assíncrona. A Figura 6 apresenta o que compõe uma requisição, representada por *Request*, e o que é esperado na resposta que vai ao *channel*, representada por *Response*.

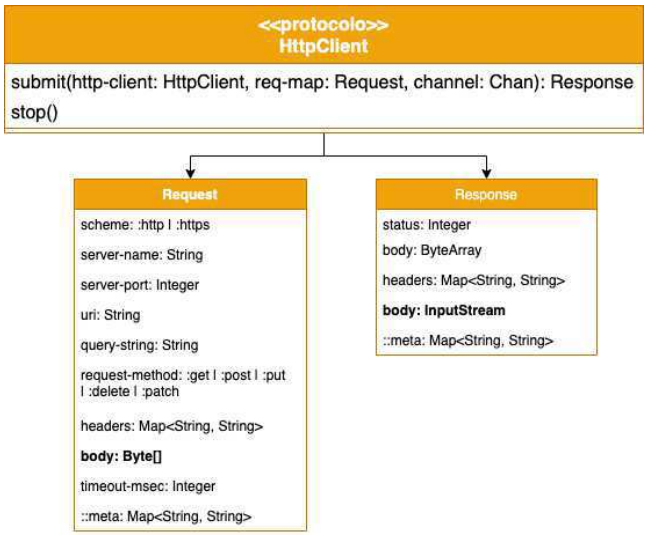


Figura 6 - Diagrama contendo o que é esperado em Request e Response pelo protocolo.

A mudança de destaque na versão anterior do protocolo, é o tipo esperado no corpo da requisição e da resposta, para a requisição o cliente HTTP recebe um *Array* de *Bytes*, já na resposta deve-se retornar um tipo da classe *InputStream* do *Java*. Esse contrato agora se alinha com o que é esperado pela função *invoke* e o tipo *InputStream* para a resposta facilita implementações de funcionalidades futuras, como *Download* de dados de serviços da AWS, principalmente para arquivos muito grandes.

Deseja-se que, no futuro, o corpo da requisição também permita *InputStream* além de *Array* de *Bytes*, para que o protocolo fique totalmente compatível com os tipos de dados que a função *invoke* lida. Inclusive, durante o projeto foi implementada uma versão do protocolo e da biblioteca que permitia tal tipo, porém

[11] https://github.com/clojure/core.async

foi descartada por ter sido identificado que outros componentes possuíam complicações que poderiam ocasionar falhas e comportamentos inesperados ao usuário. A primeira complicação foi encontrada em operações da AWS que exigem autenticação e criptografia através de cabeçalhos que utilizam os algoritmos *sha-256* e *md5*. Tais algoritmos exigem que os dados que o usuário provê para a função *invoke* sejam lidos para serem criptografados e colocados nos cabeçalhos. A operação de leitura neste processo não é um problema para *Array* de *Bytes*, porém para *InputStream* é um empecilho pois, pelo funcionamento da classe e de *Streams*, o consumo do dado esvazia o *Stream* e no ponto que o *InputStream* chegar à função *submit* do cliente HTTP estará vazio. Uma possível solução que foi experimentada durante o projeto é recompor o *InputStream* ao estado anterior depois de montar os cabeçalhos, utilizando as funções *reset*[12] e *mark*[13] da implementação Java. Porém, tais funções não possuem suporte de todos os subtipos de *InputStream*, como também podem ter comportamento inesperado dependendo do tamanho do dado dentro do *InputStream* ou se o usuário também estiver manipulando o estado do *InputStream*.

Após estas experimentações e estudos, foi decidido não incluir esse tipo no corpo da requisição na versão inicial até que seja feito um estudo mais aprofundado para suporte de *InputStream*, principalmente com dados que não cabem em memória.

3.2.1 Implementação do cognitect.http-client

Com as mudanças de tipos feitas e o novo protocolo disposto, foi feita uma adaptação para que *cognitect.http-client*, cliente HTTP padrão, atenda ao contrato porém não cause falhas ou mudanças de comportamentos para os usuários. As adaptações são converter o corpo da requisição de *Array* de *Bytes* para *ByteBuffer*, e o da resposta de *ByteBuffer* para *InputStream* com o detalhe que, por conta do *channel*, foi preciso utilizar a implementação de *pipeline* da *core.async* para aplicar a conversão no dado antes que ele chegue aos outros processos.

Na Figura 7 está a implementação do protocolo *HttpClient* utilizando *cognitect.http-client*, na qual na linha 8 é feita a conversão do corpo da requisição para *ByteBuffer* utilizando a função *->bbuf*. Para converter o corpo da resposta, foi criada a função *handle-response* que é utilizada na linha 6 da Figura 7.

```
1  (defn create
2    []
3    (let [client (impl/create {})]
4      (reify aws/HttpClient
5        (-submit [_ request channel]
6          (handle-response
7            (impl/submit client
8                         (update request :body ->bbuf))
9            channel))
10       (-stop [_]
11         (impl/stop client)))))
```

Figura 7 - Implementação do protocolo *HttpClient* utilizando *cognitect.http-client*.

A implementação de *handle-response* pode ser visualizada na Figura 8, em que na linha 4 é utilizada a função *pipeline* da *core.async* para aplicar uma conversão de *ByteBuffer* para *InputStream* na resposta na linha 6. Desta forma, atende-se ao protocolo e ao mesmo tempo não se causam falhas ao cliente HTTP.

```
1  (defn ^:private handle-response
2    [initial-response-chan
3     parsed-response-chan]
4    (a/pipeline 1
5                parsed-response-chan
6                (map (fn [response]
7                       (update response
8                               :body bbuf->input-stream)))
9                initial-response-chan)
10   parsed-response-chan)
```

Figura 8 - Implementação da função *handle-response* para converter o corpo da resposta de *ByteBuffer* para *InputStream*.

3.3 Customização do cliente HTTP global

Como demonstrado anteriormente na seção 3.1 e na Figura 4, ao criar um cliente AWS, se nenhum cliente HTTP é recebido como parâmetro, a biblioteca recorre ao cliente global que pode ser compartilhado com diferentes instâncias de clientes AWS. Permitir que o usuário defina um cliente HTTP customizado como o padrão e seja compartilhado é fundamental para o lançamento da funcionalidade de customização. Isto evita que recursos sejam gastos desnecessariamente ao precisar criar várias instâncias de um mesmo cliente HTTP, como também, permite que o usuário possa excluir definitivamente *cognitect.http-client* e *Jetty* de suas dependências, dependendo apenas do cliente que desejar.

Desta forma, foi criada uma nova implementação para permitir clientes compartilhados e a customização, na qual se utiliza *Atoms*[14] do *Clojure* para gerenciar o estado da variável. Com um *Atom* é possível criar e gerenciar uma variável mutável no *Clojure*, permitindo, assim, gerenciar o estado do cliente HTTP padrão sendo utilizado.

[12] https://docs.oracle.com/javase/7/docs/api/java/io/InputStream.html#reset()

[13] https://docs.oracle.com/javase/7/docs/api/java/io/InputStream.html#mark(int)

[14] https://clojure.org/reference/atoms

O cliente padrão e compartilhado sem customização será o *cognitect.http-client*. Como demonstrado na Figura 9, através da nova função *set-default-http-client!*, o usuário pode prover uma instância do cliente HTTP ou a função que o retorna para ser o novo cliente padrão. Importante observar que clientes AWS criados anteriormente à execução de *set-default-http-client!* não irão atualizar os seus clientes HTTP.

```
1 (set-default-http-client! meu-client-http)
```

Figura 9 - Exemplo uso da função set-default-http-client!

Essa função adiciona um novo valor ao *Atom shared-http-client* utilizando *reset!* após resolver o parâmetro que representa o novo cliente HTTP padrão, como demonstrado na sua implementação na Figura 10. A função *resolve-http-client* é utilizada tanto nessa operação, como também para resolver o cliente quando o usuário não customiza o cliente, sabendo como recorrer ao padrão que é a *cognitect.http-client.*

```
1 (defn set-default-http-client!
2   "Create a new instance of the default http-client to share explicitly across multiple
  aws-api clients.
3
4   http-client-constructor-or-instance can be either the constructor function of the
  client or the instance of the HTTP Client."
5   [http-client-constructor-or-instance]
6   (reset! #'shared/shared-http-client
7           (http/resolve-http-client http-client-constructor-or-instance)))
```

Figura 10 - Implementação da função set-default-http-client!

Conforme mostra a Figura 11, existe um condicional que verifica se o cliente a ser resolvido já é uma instância (linha 6) ou é uma função que o retorna (linha 9). Caso não seja nenhum dos dois, o *cognitect.http-client* será utilizado e está salvo numa referência englobada por um *delay*[15] e a função *dynaload/load-var* que carrega a dependência de maneira dinâmica, apenas quando realmente necessária, desta forma a biblioteca não é referenciada a não ser que realmente esteja sendo utilizada. Assim, o usuário, caso deseje, pode remover de suas dependências sem causar falhas.

```
1 (def ^:private cognitect-http-client-ref
2   (delay (dynaload/load-var 'cognitect.aws.http.cognitect/create)))
3
4 (defn resolve-http-client
5   [http-client-or-sym]
6   (let [c (cond (client? http-client-or-sym)
7                 http-client-or-sym
8
9                 (fn? http-client-or-sym)
10                (http-client-or-sym)
11
12                :else
13                (@cognitect-http-client-ref))]
14    (when-not (client? c)
15      (throw (ex-info "not an http client" {:provided http-client-or-sym
16                                            :resolved c})))
17    c))
```

Figura 11 - Implementação da função *resolve-http-client* e referência ao cliente *cognitect.http-client*.

[15] https://clojuredocs.org/clojure.core/delay

3.4 Implementação do protocolo utilizando o cliente HTTP do java.net

Clojure é projetada como uma linguagem de programação para ser hospedada e executada dentro da Máquina Virtual Java (JVM) por padrão, ou seja, possui suporte à interoperabilidade com classes, bibliotecas e funcionalidades da plataforma do Java [6]. Aproveitando esse suporte, foi feita uma implementação do protocolo *HttpClient* utilizando o cliente HTTP disponibilizado no pacote nativo *java.net* do Java.

Para utilizar essa implementação, basta o usuário importar o cliente e utilizar os métodos descritos anteriormente para sobrescrever o cliente HTTP padrão da biblioteca. Na Figura 12, podemos observar um exemplo que importa o cliente HTTP na linha 3 e o utiliza de duas formas diferentes: na linha 7, customizando um cliente AWS específico; já na linha 10, sobrescrevendo o cliente HTTP global com a função *set-default-http-client!*.

```
1 (require '[cognitect.aws.client.api :as aws])
2 ; Importa a implementação do cliente HTTP
3 (require '[cognitect.aws.http.java :as java-http-client])
4
5 ; Customiza um cliente AWS específico
6 (def s3 (aws/client {:api         :s3
7                      :http-client (java-http-client/create)}))
8
9 ; Sobrescreve o cliente HTTP global
10 (aws/set-default-http-client! (java-http-client/create))
```

Figura 12 - Exemplo utilizando a implementação do cliente HTTP do *java.net.*

Para adequar o cliente HTTP ao protocolo esperado pela *cognitect.aws-api*, foi codificada a conversão entre o mapa da requisição recebido como parâmetro na função *submit* para a classe *java.net.HttpRequest*[16] e da classe *Java.net.HttpResponse*, que é o resultado da requisição, para o mapa de resposta que é esperado estar no *channel* também recebido como parâmetro da função *submit*.

Para exemplificar como foi usada a interoperabilidade com Java, na Figura 13 podemos observar a função que transforma o mapa da requisição em uma objeto da classe *java.net.HttpRequest*. Nas linhas 9 a 14, métodos Java da classe *HttpRequest* são chamados em conjunto com funções utilitárias desenvolvidas durante o projeto, para aplicar a conversão do mapa da requisição para o que a classe espera.

[16] https://docs.oracle.com/en/java/javase/11/docs/api/java.net.http/java/net/http/HttpRequest.html

```clojure
(defn request->java-net-http-request
  "Build a java.net.HttpRequest based on an aws client request"
  [{:keys [body request-method timeout-msec]
    :or   {timeout-msec 3000} :as request}]
  (let [body-publisher (body->body-publisher body)
        uri            (request->complete-uri request)
        http-method    (method-string request-method)
        timeout        (Duration/ofMillis timeout-msec)]
    (-> (doto (HttpRequest/newBuilder)
          (.method http-method body-publisher)
          (.uri (URI/create uri))
          (.timeout timeout)
          (build-headers request))
        (.build))))
```

Figura 13 - Implementação da função que converte o mapa da requisição em um objeto da classe *java.net.HttpRequest*.

Durante o desenvolvimento dessa parte do projeto, foi identificada uma falha causada pela biblioteca tentar, em algumas operações, prover cabeçalhos que causavam uma exceção no cliente do *java.net*. Após um estudo, foi descoberto que existe uma definição no *RFC* (Pedido para comentários - *Request For Comments*) do HTTP que proíbe alguns cabeçalhos de serem modificados programaticamente [7] e o *cognitect.http-client* não atende a essa especificação, mas o *java.net* sim. Foram realizados testes manuais conectando a vários serviços da AWS sem modificar programaticamente esse cabeçalho na implementação Java para verificar se ocorriam erros. Após verificar que não ocorreram erros, foi escrita uma função que faz o tratamento e remoção dos mesmos, permitindo que o problema fosse resolvido sem modificar o comportamento atual da biblioteca.

```clojure
(def ^:private restricted-headers
  #{:host :accept-charset :accept-encoding :access-control-request-headers
    :access-control-request-method :connection :content-length :cookie :date :dbt
  :expect
    :feature-policy :origin :keep-alive :referer :te :trailer :transfer-encoding
  :upgrade :via})

(defn remove-restricted-headers
  "Remove restricted headers.
  More info:
    - https://www.rfc-editor.org/rfc/rfc7230#section-3.2
    - https://developer.mozilla.org/en-US/docs/Glossary/Forbidden_header_name"
  [headers]
  (->> headers
       (dissoc-by #(contains? restricted-headers
                              (keyword (string/replace (string/lower-case (name %)) #" "
  "-"))))
       (dissoc-by #(string/starts-with? (string/lower-case (name %)) "sec-"))
       (dissoc-by #(string/starts-with? (string/lower-case (name %)) "proxy-"))))
```

Figura 14 - Função responsável por realizar o tratamento e remoção de cabeçalhos não permitidos pelo cliente do java.net.

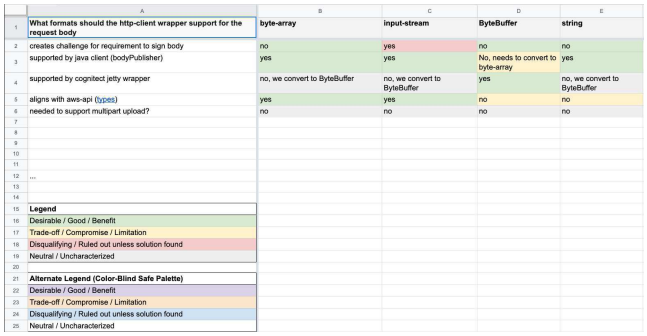
4. EXPERIÊNCIAS

Nesta seção, são discutidos a experiência de desenvolvimento, os processos adotados e os desafios do projeto.

4.1 Elaboração e Investigação

Do início ao fim do projeto, todas as ideias e propostas precisaram ser discutidas com os mantenedores e principais contribuidores da *Cognitect Labs*. Nos primeiros meses de projeto, entre maio e junho de 2022, foi escrito um documento[17] com a proposta de customização do cliente HTTP seguindo o modelo de *Request for Comments* (Pedido para comentários - RFC), detalhando o funcionamento da solução e que componentes precisariam ser modificados. Em conjunto com o documento de *RFC*, foi desenvolvida uma prova de conceito para demonstrar que era possível realizar a customização do cliente HTTP sem modificar o comportamento da biblioteca *cognitect.aws-api*. O documento e a implementação foram compartilhados com mantenedores da biblioteca, a partir dos quais ciclos de revisões, sugestões e evoluções foram constantemente feitos durante todo o projeto.

Ao longo das revisões pelos pares e experimentações feitas na prova de conceito, muitas ideias foram descartadas e novas foram propostas. Um processo de elaboração de ideias constantemente adotado era a construção de matrizes de decisões comparando soluções e alternativas pelas suas características, pontos positivos e negativos. Qualquer mudança significativa na arquitetura do projeto precisava ser apoiada por uma matriz comprovando que a solução escolhida realmente possuía mais pontos positivos ou viabilidade do que as demais alternativas, além de servir de documentação que era compartilhada com os interessados. No total, foram criadas e discutidas seis matrizes de decisões ao longo do projeto e todas possuíam um modelo como demonstrado na Figura 15.



What formats should the http-client wrapper support for the request body	byte-array	input-stream	ByteBuffer	string
creates challenge for requirement to sign body	no	yes	no	no
supported by java client (bodyPublisher)	yes	yes	No, needs to convert to byte-array	yes
supported by cognitect jetty wrapper	no, we convert to ByteBuffer	no, we convert to ByteBuffer	yes	no, we convert to ByteBuffer
aligns with aws-api (types)	yes	yes	no	no
needed to support multipart upload?	no	no	no	no
...				
Legend				
Desirable / Good / Benefit				
Trade-off / Compromise / Limitation				
Disqualifying / Ruled out unless solution found				
Neutral / Uncharacterized				
Alternate Legend (Color-Blind Safe Palette)				
Desirable / Good / Benefit				
Trade-off / Compromise / Limitation				
Disqualifying / Ruled out unless solution found				
Neutral / Uncharacterized				

Figura 15 - Exemplo de matriz de decisão comparando alternativas com seus positivos e negativos.

4.2 Processo

Inicialmente, o processo de desenvolvimento consistia de experimentações e aprendizados sobre a biblioteca *cognitect.aws-api* e o cliente HTTP do *java.net*. Posteriormente, entregas semanais eram feitas, revisadas e testadas. Os acompanhamentos semanais eram feitos com o principal mantenedor do projeto, David Chelimsky, onde eram esclarecidas dúvidas e realizadas discussões sobre hipóteses e ideias. A partir de agosto de 2022, também eram feitos acompanhamentos semanais com outros contribuidores da biblioteca. Nestes acompanhamentos eram discutidas as matrizes de decisões e compartilhados aprendizados e feedbacks, tanto do projeto sendo

17 https://docs.google.com/document/d/14G3aTT4v0wcxQROaOzePby_nkFN3cudnTdQQCkQ8XIc

desenvolvido bem como outras contribuições que estavam sendo feitas para a biblioteca em paralelo.

Similarmente, eram feitos acompanhamentos esporádicos com o orientador Fábio Morais, fornecendo *feedbacks* sobre a escrita do documento e direcionamentos para priorização de atividades e condução do projeto.

4.3 Principais Desafios

O requisito de não poder causar mudanças de comportamento ou possíveis falhas foi o mais desafiador de atender, pois além de considerar o comportamento atual esperado pelos usuários, era preciso pensar como continuar atendendo esse requisito no futuro e como a solução sendo desenvolvida para a customização do cliente HTTP poderia impactar positivamente ou negativamente nesse aspecto. Por conta desse requisito, nenhuma decisão poderia ser tomada sem ser discutida, analisada e comparada com outras alternativas.

O desenvolvimento se baseou em experimentações e discussões, algumas funcionalidades foram implementadas porém descartadas da solução final. Por exemplo, de início era planejado permitir submeter *InputStream* no corpo da requisição, o que foi descartado logo depois por ter sido identificada possíveis problemas que poderiam causar no futuro. Outro exemplo, uma nova biblioteca seria criada para dar suporte a customização, tanto do cliente HTTP como de outros componentes utilizados na biblioteca, porém após algumas experimentações e feedbacks, concluíu-se que não trazia muitas vantagens para os usuários e o escopo do projeto seria muito incerto.

5. RESULTADOS

Para avaliação e apuração do projeto, foram realizados testes de integração, manuais e automatizados, conectando alguns serviços reais da AWS através da biblioteca com e sem a customização do cliente HTTP. Os testes verificam se a conexão HTTP foi feita corretamente e se a biblioteca entrega o resultado correto ao usuário.

Na Figura 16, temos um exemplo de teste de integração automatizado que customiza um cliente AWS do serviço S3 com a implementação do cliente HTTP Java. O teste tenta criar um dado esperado pelo serviço S3 na operação *:CreateBucket* e verifica se ele realmente foi criado.

```
(deftest ^:integration test-s3
  (let [s3 (aws/client {:api          :s3
                        :http-client (java-http-client/create)})
        bucket-name (str "aws-api-test-bucket-" (.getEpochSecond (Instant/now)))]
    (testing ":CreateBucket"
      (aws/invoke s3 {:op :CreateBucket :request {:Bucket bucket-name}})

      (is (bucket-exists? s3 bucket-name))))
```

Figura 16 - Exemplo de teste se conectando com o serviço AWS S3 utilizando o cliente HTTP Java.

Um ponto adicional da implementação desses testes de integração[18] é que anteriormente a biblioteca não continha testes desse tipo, sendo então uma contribuição significativa para a validação dos componentes da biblioteca daqui em diante.

Também foram criados testes unitários[19] para os componentes que fizeram parte da solução, em adição aos testes que já existiam. Na Figura 17 está disposto o resultado de todos os testes, unitários e de integração, relacionados à implementação do cliente HTTP do *java.net* em conjunto com o protocolo *HttpClient*. Na Figura 18, é possível observar o resultado da execução de todas as suítes de testes da biblioteca após o desenvolvimento da solução.

```
(clojure.test/run-tests 'cognitect.aws.http.java.java-http-client-test)

Testing cognitect.aws.http.java.java-http-client-test

Ran 5 tests containing 22 assertions.
0 failures, 0 errors.
=> {:test 5, :pass 22, :fail 0, :error 0, :type :summary}
(clojure.test/run-tests 'cognitect.aws.http.java.integration-test)

Testing cognitect.aws.http.java.integration-test

Ran 1 tests containing 8 assertions.
0 failures, 0 errors.
=> {:test 1, :pass 8, :fail 0, :error 0, :type :summary}
```

Figura 17 - Resultado das suítes de testes relacionados a implementação do cliente HTTP java.net em conjunto com o protocolo HttpClient.

```
Ran 55 tests containing 3769 assertions.
0 failures, 0 errors.
```

Figura 18 - Resultado de todas as suítes de testes da cognitect.aws-api.

6. CONCLUSÃO

Concluímos que a solução proposta oferece uma funcionalidade completa e flexível de customização do cliente HTTP utilizado pela biblioteca Clojure *cognitect.aws-api* sem ocasionar quebras de compatibilidade para usuários e criando uma separação clara entre as responsabilidades dos componentes da biblioteca e o cliente HTTP. Além disso, também disponibiliza uma alternativa pronta de customização com o cliente HTTP do *java.net* adequado para a biblioteca, oferecendo uma solução imediata para a substituição do cliente HTTP da *cognitect.http-client*. A seguir, discutimos possíveis trabalhos futuros que podem ser desenvolvidos.

[18] https://github.com/thayannevls/aws-api/tree/main/test/src/integration

[19] https://github.com/thayannevls/aws-api/blob/main/test/src/cognitect/aws/http/java_net_test.clj

6.1 Trabalhos Futuros

O lançamento das funcionalidades desenvolvidas ainda não foi feito pelos mantenedores da Cognitect, porque ainda está em processo de revisão final por pares que não participaram do processo. Portanto, é possível que após revisão ainda sejam necessárias discussões, adaptações e evoluções antes do lançamento oficial. Uma vez que estas funcionalidades tenham sido lançadas, é esperado que sejam estudados quais clientes HTTP os usuários mais estão utilizando com a customização para, assim, iniciar uma discussão sobre se o cliente padrão da biblioteca deve ser modificado para um mais popular adequado para as necessidades da comunidade. Evidentemente, essa alteração não deve causar mudanças de comportamento ou impacto aos usuários.

Outro ponto relevante, mencionado anteriormente, é que o protocolo *HttpClient* permite apenas *Array* de *Bytes* nos corpos das requisições, excluindo o tipo *InputStream* que é permitido pela biblioteca na função *invoke.* Atualmente, o corpo da requisição sempre é convertido para *Array* de *Bytes*, o que impossibilita lidar com dados que não cabem em memória, além de não existir uma total concordância entre os tipos lidados entre componentes da biblioteca e o cliente HTTP. Desta forma, um ponto de evolução é que a biblioteca possa dar suporte para que os clientes HTTP recebam dados com tipo de *InputStream* para submeter requisições.

Uma funcionalidade essencial e costumeiramente usada em serviços de computação em nuvem é a possibilidade de transmissão e recuperação de dados muito grandes em tamanho, usando técnicas como *Streaming* ou *Multi-part* Upload/Download. A responsabilidade de entregar essa funcionalidade faz parte tanto dos componentes da biblioteca *cognitect.aws-api* como também do cliente HTTP acoplado a ela. Atualmente, pelo fato da biblioteca não permitir transmissão de dados que não cabem em memória, o protocolo *HttpClient* não foi desenhado pensando nesse requisito, porém é essencial que esta funcionalidade seja considerada como um trabalho futuro.

7. AGRADECIMENTOS

Gratidão aos meus queridos Verônica e Jack por todo amor e por sempre acreditarem em mim. Agradecimentos à minha família que me deu suporte e apoio. Aos meus amigos que estiveram comigo durante toda esta jornada, compartilhando tanto felicidades como tristezas, sempre contribuindo para o crescimento um do outro. Obrigada à comunidade OpenDevUFCG pelas experiências e sonhos que realizamos juntos durante a graduação.

Obrigada David Chelimsky que tornou esse projeto possível, acreditou no meu potencial e me ensinou muito durante todo o processo. Agradecimentos também ao meu orientador Fábio Morais pela paciência e dedicação nos acompanhamentos. Também gostaria de agradecer a todos os colegas de trabalho que contribuíram para esse projeto e o meu desenvolvimento pessoal e profissional.

8. REFERÊNCIAS

[1] 2022 Gartner Report Magic Quadrant for Cloud AI Developer Services Amazon Web Services. Retrieved January 2023 from https://pages.awscloud.com/Gartner-Magic-Quadrant-for-Cloud-AI-Developer-Services.html.

[2] Announcement for End of Community Support for Jetty 9.x. Retrieved January 2023 from https://github.com/eclipse/jetty.project/issues/7958.

[3] Jetty Security Reports. Retrieved January 2023 from https://www.eclipse.org/jetty/security_reports.php.

[4] Data Types of cognitect.aws-api. Retrieved January 2023 from https://github.com/cognitect-labs/aws-api/blob/main/doc/types.md.

[5] Higginbotham, D. (2015). “Chapter 11: Mastering Concurrent Processes with core.async” in Clojure for the brave and true: Learn the ultimate language and become a better programmer. San Francisco, CA.

[6] Rathore, A. (2016) “Exploring Clojure and Java interop,” in Clojure in action. Shelter Island, NY: Manning.

[7] Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. Retrieved January 2023 from https://www.rfc-editor.org/rfc/rfc7230.