Explorando Padrões de Comunicação para Servir Sistemas Inteligentes

Washington Luiz Meireles de Lima
Ygor Tavela Alves da Silva

Monografia Trabalho de Formatura Supervisionado
apresentado à disciplina
MAC0499

Orientadores:
Prof. Dr. Alfredo Goldman
Me. Renato Cordeiro Ferreira

São Paulo, Dezembro de 2022

Conteúdo

Resumo

Sistemas Inteligentes fazem a ponte entre usuários e o Aprendizado de Máquina (ML) para atingir objetivos significativos. Em particular, a inteligência evolui e melhora com o tempo, analisando como os usuários interagem com o sistema. Tais sistemas possuem três eixos de mudança: código, dados e modelo. Isso amplia a complexidade de desenvolvimento se comparado a sistemas tradicionais, o que traz à tona novos desafios. Dentre eles, está a escalabilidade, que engloba as etapas de coleta de dados, engenharia de características (*features*), treinamento do modelo de ML, e o serviço do modelo de ML em produção. Nessa última etapa, uma decisão arquitetural importante é a escolha do padrão de comunicação utilizado para servir o modelo, que pode ser classificado de acordo com o tipo de responsividade: síncrono ou assíncrono. No contexto de sistemas inteligentes, a escolha de um padrão de comunicação pode não se basear apenas em premissas já conhecidas do desenvolvimento de sistemas tradicionais. Por isso, o objetivo desta pesquisa é de explorar *trade-offs* de diferentes padrões, em particular, a API RPC (síncrona) e a Mensageria (assíncrona).

Palavras-chave: Sistema Inteligente, *Benchmark*, Padrão de Comunicação

Abstract

Intelligent Systems connect users to Machine Learning (ML) to achieve meaningful objectives. In particular, the intelligence evolves and improves by analyzing how users interect with the system. These systems have three axes of change: code, data, and model. This increases the complexity of development if compared to traditional systems, bringing up new challenges. Among these challenges is scalability, which encompasses the steps of data collection, feature engineering, ML model training, and model serving in production. In the latter step, a key architectural decision is the choice of the communication pattern, which can be classified as synchronous or asynchronous, depending on the interaction with the client. Within intelligent systems, this choice may not rely on the development experience of traditional systems. For this reason, this research aims to explore some scenarios and trade-offs of different patterns, particularly the RPC API (synchronous) and the Broker (asynchronous).

Keywords: Intelligent System, Benchmark, Communication Pattern

Capítulo 1

Introdução

Um sistema de software que possui uma inteligência capaz de evoluir e melhorar com o tempo, particularmente analisando como os usuários interagem com o sistema, é referido como um sistema inteligente (Hulten , 2019). Sistemas inteligentes podem possuir várias finalidades: uma tradução feita no Google Tradutor, recomendações de *playlists* com músicas adequadas ao perfil de qualquer pessoa com o Spotify, ou mesmo a definição de um motorista para um passageiro do Uber.

Dentro do ciclo de vida de um sistema inteligente, podemos dividir uma aplicação de aprendizado de máquina em três eixos de mudança: dados, modelo, e código (Sato **et al.** , 2019). Os dados e modelos concedem uma característica particular aos sistemas inteligentes, já que os modelos são retreinados conforme há entrada de mais dados nos sistemas. Ou seja, o sistema se encontra em um estado de constante evolução, o que o torna mais complexo, mais difícil de entender, e mais difícil de testar. Tal fato influencia a concepção da arquitetura de software para sistemas inteligentes, isto é, como o código deve se estruturar para comportar as mudanças promovidas pelos outros dois eixos.

Assim como sistemas de software tradicionais, o processo de projeto de sistemas inteligentes deve garantir que o sistema funcione com um propósito claro, e que também possua características arquiteturais tais como robustez e manutenibilidade. Essas características estão diretamente relacionados com a concepção de uma boa arquitetura de software. A arquitetura de software de um sistema é uma **forma** dada para o sistema por aqueles que o constroem (Martin , 2017). A estrutura dessa forma se traduz em como os componentes do sistema são divididos, estão dispostos, e se relacionam entre si. Tal forma tem como objetivo garantir uma base sólida para o sistema, tornando-o fácil de entender, desenvolver e servir para o cliente.

Dentro da arquitetura de um sistema inteligente, surge um importante questionamento relacionado à escalabilidade (Lakshmanan **et al.** , 2020): **Como servir um modelo em produção de tal forma que ele suporte milhões de requisições de predições em um curto período de tempo?** Para abordar esse questionamento, é fundamental entender o padrão de comunicação entre quem serve e quem consome o modelo. Comumente, a principal característica de um padrão de comunicação a se determinar é a sua responsividade – síncrona ou assíncrona. A partir disso é possível escolher o protocolo mais adequado para a comunicação.

Dadas as peculiaridades de um sistema inteligente e a necessidade de se construir uma boa arquitetura de software, o objetivo desta pesquisa é **o de explorar padrões de comunicação para servir um sistema inteligente, buscando avaliar as diferenças entre os tipos de responsividade de um padrão de comunicação**. Consequentemente, este projeto visa encontrar os principais prós e contras de cada abordagem, além de melhor discutir a aplicabilidade de cada um dos padrões.

No Capítulo 2, serão apresentados conceitos relevantes para o que está sendo discutido nesta monografia. No Capítulo 3, é documentado a arquitetura das aplicações implementadas para a realização de experimentos. No Capítulo 4, é apresentado experimentos utilizados para responder alguns questionamentos. Por fim, no Capítulo 5 são apresentadas as conclusões e considerações futuras desta monografia.

Capítulo 2

Revisão de Literatura

2.1 Sistemas Inteligentes

Sistemas Inteligentes são aqueles em que existe alguma inteligência (utilizando técnicas de inteligência artificial ou aprendizado de máquina) aprendendo e evoluindo com dados (Hulten , 2019). Por este motivo, a implementação de um sistema inteligente impõe diferentes exigências que os diferenciam de sistemas de software tradicionais. O ciclo de vida de tais sistemas incluem: como criar a inteligência, como mudar a inteligência, como organizar a inteligência, e como lidar com erros ao longo do tempo (Hulten , 2019). Para isso, construir um sistema inteligente efetivo requer balancear cinco componentes principais:

- como definir um objetivo que seja claro,
- como apresentar a saída dos modelos de aprendizado aos usuários,
- como executar a inteligência,
- como criar uma inteligência que cumpra com o seu objetivo, e
- como orquestrar o ciclo de vida da inteligência.

2.2 Aprendizado de Máquina

A Inteligência Artificial (IA) busca entender o funcionamento de agentes inteligentes e como contruí-los (Russel e Norvig , 2012). Algumas definições de IA podem se agrupar em quatro categorias de sistemas:

1. que pensam como humanos,
2. que pensam racionalmente,
3. que agem como humanos, e
4. que agem racionalmente.

As definições 1 e 3 se baseiam na capacidade humana e em estudos empíricos, envolvendo hipóteses e experimentos. Por outro lado, as definições 2 e 4 baseiam-se no conceito ideal de inteligência, tido como racionalidade, no qual combinam-se matemática e engenharia. Dentre essas definições, a quarta é onde se enquadra o Aprendizado de Máquina.

O Aprendizado de Máquina (do inglês *Machine Learning*, ou ML) é uma área de IA que se preocupa em construir algoritmos por meio de dados, os quais podem vir da natureza, serem criados pelos humanos, ou serem gerados por outros algoritmos. Pode ser definido também pelo processo de coleta de um conjunto de dados e pelo treinamento de um modelo estatístico usando esse conjunto por meio de algum algorítimo de aprendizado.

2.3 Arquitetura de Software

A arquitetura de software define as partes que compõem o software, como são as sua estruturas e como elas se relacionam entre si (Martin , 2017). A estrutura de um software é o que permite que ele seja flexível o suficiente para que evolua rapidamente e mude o seu comportamento para atender novas necessidades. Em outras palavras, é o que o torna maleável o suficiente para deixar o maior número de opções disponíveis pelo maior tempo possível (Martin , 2017).

Uma definição mais moderna se estende para além da estrutura em si. O conhecimento das características arquiteturais, decisões arquiteturais e princípios de projeto são necessários para entender totalmente uma arquitetura de sistema (Richards e Ford , 2020).

2.3.1 Características Arquiteturais

Características arquiteturais definem critérios de sucesso do sistema, que geralmente são ortogonais às funcionalidades do sistema (Richards e Ford , 2020). Uma característica arquitetural atende a três critérios: especifica uma consideração de projeto não relacionada ao domínio da aplicação, influencia um aspecto estrutural do projeto, e é crítica ou importante para o sucesso da aplicação. Alguns exemplos de características são disponibilidade, escalabilidade, extensibilidade, manutenibilidade, privacidade, segurança, etc.

2.3.2 Decisões Arquiteturais

Decisões arquiteturais definem regras de como o sistema deve ser construído (Richards e Ford , 2020). As decisões formam restrições do sistema e direcionam os times de desenvolvedores para o que é e o que não é permitido fazer. Caso uma decisão arquitetural não possa ser implementada em uma parte do sistema em virtude de uma condição ou outra restrição, ela pode ser quebrada em algo chamado **variância** (Richards e Ford , 2020).

2.3.3 Princípios de Projeto

Princípios de projeto nos dizem como devemos organizar as funções e as estruturas de dados em agrupamentos, e como esses agrupamentos devem estar interconectados (Martin , 2017). Os princípios de projeto promovem diretrizes gerais que podem ou não serem seguidas durante o desenvolvimento de um software, com o objetivo de melhorar a estrutura de um sistema computacional. A possibilidade de seguir ou não um princípio é o que o difere das decisões que são definidas como regras que devem ser seguidas sempre que possível (Richards e Ford , 2020).

2.3.3.1 Princípio da Responsabilidade Única

Este princípio pode ser condensado na seguinte frase:

> *"Um módulo deve ser responsável por um, e apenas, um ator"* (Martin , 2017)

Um **módulo** pode ser classificado como um arquivo fonte, ou então, em algumas linguagens, como um conjunto coeso de funções e estruturas de dados. O **responsável** diz respeito ao fato de que o módulo só deve ter uma razão para mudar, e pensando num sistema de software, tais razões para mudança sempre serão requeridas por um **ator** (um usuário do sistema, por exemplo).

2.3.3.2 Princípio do Aberto-Fechado

Este princípio estabelece que, para um artefato de software ser facilmente modificado, ele deve ser projetado para permitir que o seu comportamento seja alterado adicionando novo código em vez de alterar código já existente. Isto é, aberto para **extensão**, mas fechado para **modificações** (Martin , 2017). A sua prática possibilita que os componentes sejam separados em **como**, **porque**, e **quando**

eles mudam. De tal forma, os componentes ficam organizados numa hierarquia de dependências que protege componentes de alto-nível de mudanças provocadas por componentes de baixo-nível.

2.3.3.3 Princípio da Substituição de Liskov

Este princípio estabelece diretrizes para o uso adequado de herança a partir da definição da relação de subtipos. Essencialmente, para construir um sistema de software a partir de partes que podem ser alternadas, é necessário que essas partes respeitem um contrato que permite que elas sejam substituíveis umas pelas outras (Martin , 2017). Consolida-se, portanto, a ideia de que herança não deve ser abusada como um mecanismo de reúso de código. Um exemplo para elucidar o princípio, é o presente na Figura 2.1 que apresenta uma violação do princípio. Tomando como base o domínio geométrico Euclidiano, um objeto da classe `Quadrado` não é um subtipo adequado da classe `Retângulo` já que a altura e a largura de um `Retângulo` são valores que podem ser diferentes entre si, ao passo que, a altura e a largura de um `Quadrado` são sempre iguais.

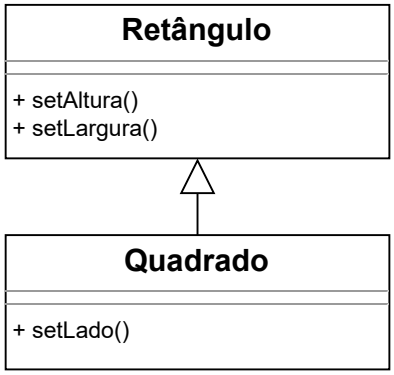

Figura 2.1: *Exemplo de violação do princípio da substituição de Liskov.*

Quando o princípio é estendido a nível arquitetural, a violação do princípio pode ocasionar na poluição da arquitetura com a criação de diversos mecanismos extras para tratar cada um dos artefatos de software que não são substituíveis por não respeitarem o princípio.

2.3.3.4 Princípio da Segregação de Interface

Este princípio estabelece que a criação de dependências desnecessárias devem ser evitadas (Martin , 2017). Esse princípio incentiva o encapsulamento e o desacoplamento do código, reforçando a questão de boa modularização do código para facilitar mudanças futuras. Essencialmente, a lição é de que depender de algo que carrega uma bagagem que você não precisa pode causar problemas que você não esperava (Martin , 2017).

2.3.3.5 Princípio da Inversão de Dependência

Este princípio estabelece que os sistemas mais flexíveis são aqueles cujas dependências do código fonte se referem apenas a abstrações e não concretizações (Martin , 2017). Os elementos concretos se referem aos elementos **voláteis** do sistema, ou seja, aqueles em que há constante desenvolvimento e passam por mudanças frequentes. Por outro lado, os elementos abstratos são **estáveis** em relações a mudanças. Dessa forma, tal princípio possibilita a divisão do sistema em dois componentes em relação aos seus níveis de **estabilidade**: um abstrato e outro concreto. Assim, contribui-se para que o código seja mais suscetível a mudanças, já que os detalhes (concretos) podem mudar sem afetar os contratos (abstratos).

2.3.4 Qualidade da Arquitetura

Além do próprio valor entregue pela solução do software, uma outra ótica a se avaliar no valor de um software se dá pela sua estrutura (Martin , 2017). Muitas vezes, por não ser algo aparente ao usuário final, ou pelo custo de tempo e esforço associado, a arquitetura acaba sendo deixada de lado no processo de desenvolvimento do software. Dessa forma, é importante notar que a concepção da arquitetura não se refere apenas ao processo inicial de desenvolvimento, mas a todo ciclo de vida de um sistema.

Um maior tempo de vida de sistema sempre irá se beneficiar do uso de boas práticas de desenvolvimento, independentemente de tempo e esforço para o desenvolvimento de um software de alta qualidade (Martin Fowler , 2019). Com isso, a principal finalidade da construção da arquitetura é reduzir custos de desenvolvimento, aumentando a compreensão do código pelos programadores, melhorando a manutenibilidade do sistema, facilitando a priorização de requisitos, etc.

2.4 Padrões de Projeto

Padrões de projeto são soluções recorrentes para problemas conhecidos no processo de desenvolvimento de software (Gamma **et al.** , 1996). Cada padrão engloba um problema em específico, com a discussão de fatores que afetam o problema e uma sugestão de como resolvê-lo de forma satisfatória. Na concepção de arquitetura de software, os padrões são uma importante ferramenta para os desenvolvedores, contribuindo para reduzir o gasto de tempo ao resolver um problema.

2.4.1 Padrão Método Fábrica

O padrão Método Fábrica é um padrão de criação de objetos, comumente utilizado no paradigma orientado a objetos. Uma interface é definida para criar um objeto, no entanto, as subclasses decidem qual classe instanciar (Gamma **et al.** , 1996). O padrão tem por objetivo tornar o código desacoplado, encapsulado e extensível. Um dos principais resultados do seu uso, é a definição de uma fronteira entre classes abstratas e concretas, invertendo a direção das dependências do código fonte contra o fluxo de controle do código.

Na Figura 2.2, temos um exemplo de aplicação do padrão para a criação de objetos de formas geométricas. Nesse exemplo, é definido uma classe abstrata `FábricaGeométrica` que define no contrato um método para criação de uma outra classe abstrata, a `FormaGeométrica`. Os subtipos concretos de `FábricaGeométrica` são responsáveis por criar um subtipo concreto de `FormaGeométrica`, de tal forma que uma fronteira é definida entre o que é abstrato e o que é concreto pela curva pontilhada (em azul).

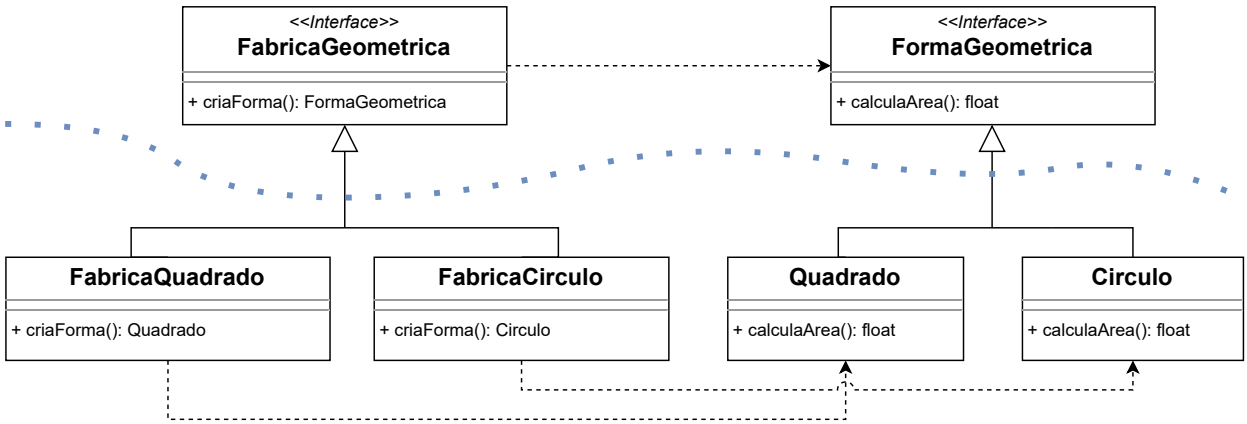

Figura 2.2: *Aplicação Padrão Método Fábrica*

2.4.2 Padrão Estratégia

O padrão ESTRATÉGIA é um padrão comportamental que define uma família de algoritmos, encapsula cada um deles e os torna alternáveis entre si (Gamma **et al.** , 1996). O padrão tem por objetivo isolar detalhes de implementação de diferentes algoritmos do código que os usa, tornando o código encapsulado e extensível.

Na Figura 2.3, temos um exemplo de aplicação do padrão para a definição do comportamento de uma classe responsável por colorir formas geométricas com diferentes estilos de pintura. Nesse exemplo, é definido uma classe abstrata `EstrategiaColorirFormaGeometrica` que define um contrato para colorir uma forma geométrica. Cada um dos subtipos dessa classe abstrata define um estilo diferente de pintura: `PinturaAbstrata`, `PinturaCubista`, ou `PinturaImpressionista`. Desta forma, para a classe cliente responsável por pintar as formas geométricas, definida por `ColorirFormaGeometrica`, o algoritmo executado para colorir uma forma geométrica pouco importa, isolando-o dos detalhes do algoritmo para colorir. Assim, a classe cliente pode usar qualquer uma das estratégias definidas, incluindo novas estratégias que implementem o contrato `EstrategiaColorirFormaGeometrica`.

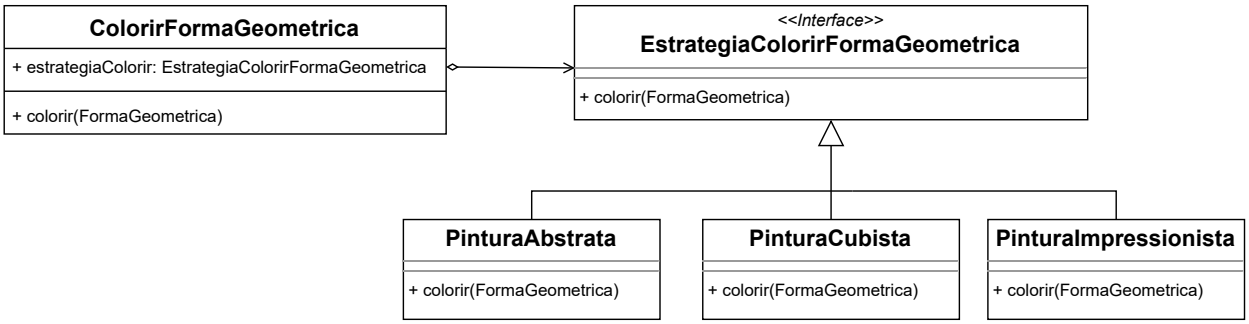

Figura 2.3: *Aplicação Padrão Estratégia*

2.4.3 Padrão Injeção de Dependência

O padrão INJEÇÃO DE DEPENDÊNCIA é um padrão aplicado principalmente em linguagens orientadas a objetos. A sua ideia se baseia na existência de um objeto separado, um **injetor**, cuja função é a de popular as dependências de uma classe apropriadamente (Martin Fowler , 2004). Desta forma, é possível evitar o acoplamento de classes com dependências em classes concretas. No exemplo abaixo, uma classe concreta `FabricaQuadrado` é instanciada diretamente no construtor da classe `CaixaFormasGeometricas`:

```
class CaixaFormasGeometricas():
    def __init__(self):
        self.fabrica_quadrado = FabricaQuadrado()
        self.fabrica_circulo = FabricaCirculo()
```

O padrão fornece uma forma de inversão de controle do fluxo do código que reduz o acoplamento das classes com as suas dependências, já que separa a instanciação dos objetos de dependência com o seu uso em si. São definidos três tipos de injeção de dependência: Injeção de **construtor**, de **interface**, e método de atribuição. Ajustando o exemplo acima para aplicar o tipo de injeção de construtor, passamos a desacoplar a construção da dependência `FabricaQuadrado` do construtor da classe `CaixaFormasGeometricas`:

```
class CaixaFormasGeometricas():
    def __init__(
        self,
        fabrica_quadrado: FabricaQuadrado,
        fabrica_circulo: FabricaCirculo
    ):
        self.fabrica_quadrado = fabrica_quadrado
        self.fabrica_circulo = fabrica_circulo
```

2.4.4 Padrão Produtor-Consumidor

O padrão PRODUTOR-CONSUMIDOR ajuda a manter o estado de componentes concorrentes sincronizados (Bushmann **et al.** , 1996). Para permitir isso, o padrão habilita a propagação de mudanças unidirecionalmente: um **Produtor** notifica um número qualquer de **Consumidores** sobre a mudança do seu estado. A sua adoção resulta numa forma de notificação desacoplada entre Produtores e Consumidores o que favorece características arquiteturais como **escalabilidade** e **disponibilidade**.

2.5 Estilos Arquiteturais

Um estilo de arquitetura é a inspiração que está por trás da ideia da arquitetura de software (2.3), sendo uma importante decisão arquitetural (2.3.2) durante o processo de concepção da arquitetura. Tal inspiração é definida como um conjunto de princípios (2.3.3) e padrões (2.4) a serem adotados para construir uma estrutura que atenda os requisitos funcionais e características arquiteturais de um sistema (Garlan e Shaw , 1993). Mais especificamente, um estilo arquitetural determina o *vocabulário* de componentes e as formas de interação entre eles que podem ser usadas nesse estilo, em um conjunto de restrições.

2.5.1 Camadas

A arquitetura em camadas é um dos estilos mais simples e de menor custo a ser implementado. Neste estilo arquitetural, são criados camadas com diferentes responsabilidades cujas dependências devem ser sempre no sentido das camadas adjacentes (Martin , 2017). Comumente, são definidas três camadas principais (Martin Fowler , 2015): a camada de apresentação (Interface de Usuário, ou *User Interface*, UI), a camada da lógica de negócios, e a camada de acesso aos dados.

Supondo um sistema responsável por renderizar formas geométricas para o cliente em uma dada interface de usuário, uma implementação do sistema que adota o estilo arquitetural em camadas é exibido na Figura 2.4. Na figura, o `Visualizador` é responsável por ser uma camada de apresentação do sistema para os clientes. O `Serviço de Busca de Formas Geométricas` – uma camada de lógica de negócios – é responsável por realizar o processamento da entrada de dados, buscar formas geométricas no `Repositório de Formas Geométricas` – a camada de dados, para enfim retornar uma resposta ao cliente por meio da camada de apresentação.

2.5.2 Hexagonal

A arquitetura hexagonal é um estilo arquitetural que busca separar as regras de negócio da aplicação dos detalhes de implementação, tais como o arcabouço, a interface de usuário, banco de dados, etc (Cockburn , 2005). A ideia fundamental se baseia no uso de **Portas** e **Adaptadores**, que são conceitos que se baseiam fortemente no uso de diferentes princípios de projeto (2.3.3), e no uso do padrão de injeção de dependências (2.4.3). O uso de tal padrão é fundamental para o estilo, já que ele sustenta o cumprimento do princípio da **Regra da Dependência** (Martin , 2017), na

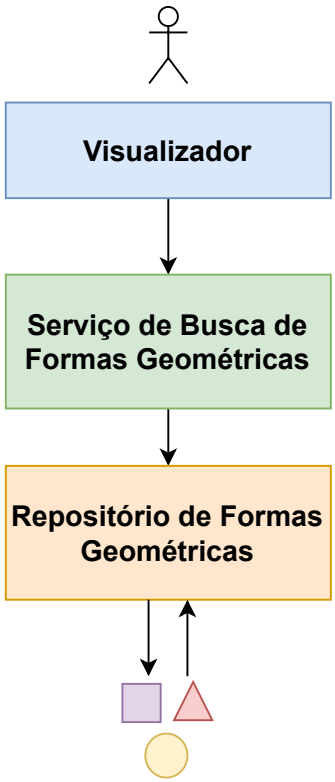

Figura 2.4: *Exemplo de aplicação do estilo em camadas*

qual as dependências devem sempre apontar para camadas mais internas da arquitetura, que devem ser completamente ignorantes em relação às camadas externas.

Em relação aos conceitos fundamentais do estilo, as **Portas** são definidas como um ponto de entrada e saída independente do consumidor para dentro/fora da aplicação. Em muitas linguagens orientadas a objetos, uma porta será uma interface que não possui conhecimento da implementação concreta até ser injetada em tempo de execução. Em contrapartida, os **Adaptadores** são classes que adaptam uma interface em outra.

Um dos principais problemas que esse estilo arquitetural se propõe a resolver é a de criar uma fronteira clara entre o que é regra de negócio e o que é detalhe para o sistema. Diferentemente da Arquitetura em Camadas (2.5.1), o domínio não possui conhecimento do que é externo a ele. Ou seja, a lógica de negócios encapsula totalmente a infraestrutura externa como banco de dados, controladores, sistemas de mensageria, etc.

A Figura 2.5 demonstra uma aplicação do estilo análogo ao exemplo da Arquitetura em Camadas da Figura 2.4. O aumento da complexidade da aplicação é nítido, o que é um fator a ser considerado ao escolher tal estilo. No entanto, a distinção bem definida entre o que é infraestrutura (externo) e o que é domínio (interno) é o que torna a escolha do estilo Hexagonal um grande diferencial.

2.6 Padrões de comunicação

A integração entre diferentes aplicações via rede lida com diversos desafios: conexão não confiável e lenta, diferenças nas aplicações, e mudanças de contrato. Algumas das formas de integração são: arquivo de transferência, banco de dados compartilhado, chamada de procedimento remoto, e troca de mensagens. Todas essas abordagens têm vantagens e desvantagens. Por isso, não é incomum que uma aplicação use múltiplas formas de integração (Hohpe e Woolf , 2012).

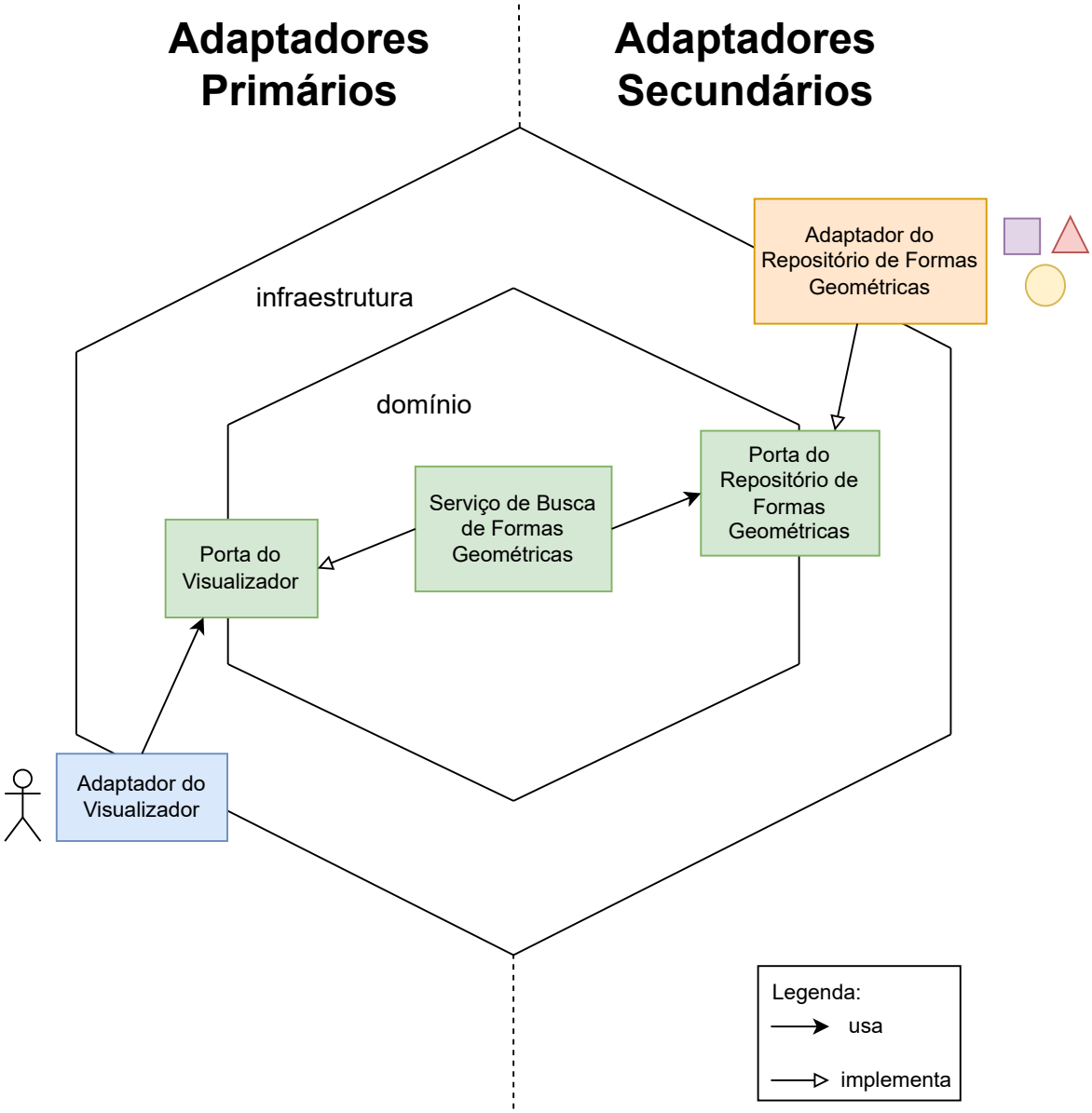

Figura 2.5: *Exemplo de aplicação do estilo hexagonal*

2.6.1 Chamada de Procedimento Remoto

Uma Chamada de Procedimento Remoto, do inglês *Remote Procedure Call (RPC)*, é uma transferência de controle síncrona entre programas em espaços de endereçamento disjuntos cujo meio de comunicação primário é um canal de rede (Nelson , 1981). Tal padrão pode funcionar em diferentes protocolos de comunicação, como o `HTTP`, e se baseia no modelo cliente-servidor. A natureza síncrona da comunicação entre um cliente e servidor é apontado como uma desvantagem do padrão de comunicação, já que é gerado acoplamento entre os participantes da comunicação.

2.6.2 Produtor-Consumidor

Produtor-Consumidor é uma forma de comunicação que se baseia no uso de protocolos de comunicação de mensageria e no padrão de projeto de mesmo nome (2.4.4). O seu funcionamento está associado a sistemas de mensageria que são responsáveis por coordenar e gerir o envio e recebimento de mensagens, assim como a leitura e persistência das mensagens (Hohpe e Woolf , 2012).

O padrão de comunicação é por sua natureza assíncrono, tendo em vista que o produtor e o consumidor não estabelecem uma comunicação direta entre si. Tal fato contribui para a redução do acoplamento e, consequentemente, aumenta a disponibilidade de sistemas que o adotam por permitir o processamento de mensagens independente da disponibilidade de um dado produtor. Todavia, uma desvantagem é que para estabelecer uma comunicação via mensagens, é necessário o uso de um sistema de mensageria, aumentando o grau de dependência dos sistemas que adotam tal padrão com este componente.

2.7 Documentação Arquitetural

No desenvolvimento de software, a modelagem e a diagramação são conceitos importantes quando se pensa em como representar uma aplicação. A principal distinção entre ambos conceitos é que a modelagem providencia uma representação abstrata do sistema, enquanto a diagramação providencia uma representação concreta (Brown , 2018). Justamente por providenciar uma representação abstrata, com uma definição única de todos elementos e como eles se relacionam entre si, a modelagem se torna uma tarefa que exige mais rigor e trabalho. Desta forma, a diagramação é uma alternativa mais simples e preferível entre os desenvolvedores de software quando o assunto é representação de um sistema.

2.7.1 Linguagem de Modelagem Unificada

A Linguagem de Modelagem Unificada (mais conhecido pela sigla UML, do inglês *Unified Modeling Language*) é a principal linguagem gráfica para visualizar, especificar, construir, e documentar artefatos de um sistema de software (Booch **et al.** , 2005). O *UML* oferece uma forma padrão para escrever esquemas de sistemas, cobrindo não só conceitos abstratos, como processos de negócio e funcionalidades do sistema, mas também conceitos concretos, como componentes de software e esquemas de banco de dados. A sua complexidade e dificuldade para aprendizado, no entanto, é apontado como um entrave para ampla adoção entre times de desenvolvimento de software.

2.7.2 Modelo C4

O modelo C4 é inspirado pela UML. Ele se propõe a ser simples de aprender e usar, cumprindo com os objetivos de ajudar times de desenvolvedores a descrever e comunicar uma arquitetura de software, além de reduzir a lacuna entre a descrição de uma arquitetura e o seu código fonte (Brown , 2018). O modelo se baseia numa abordagem que privilegia a abstração para diagramação de uma arquitetura de software. Nesta abordagem são definidas 4 abstrações principais:

1. **Contexto**:

   Representa como o sistema se encaixa no mundo real em termos de como as pessoas e como outros sistemas de software interagem com o sistema.

2. **Contêiner**:

   Representa as aplicações, base de dados, microsserviços, etc., que compõem o sistema de software.

3. **Componente**:

   Representa um contêiner individualmente, destacando cada um das partes que o compõem.

4. **Código**:

   Representa um componente individual a nível de código.

Capítulo 3

Arquitetura da Aplicação

Com o objetivo de avaliar como diferentes padrões de comunicação (Seção 2.6) se comportam em um sistema inteligente (Seção 2.1), foram construídos dois sistemas de software seguindo boas práticas de arquitetura de software (Seção 2.3), adotando diversos princípios (Subseção 2.3.3) e padrões de projeto (Seção 2.4):

1. O sistema ***Benchmark***, que expõe uma interface para realizar um teste de carga parametrizado. O teste se baseia em várias requisições de predições feitas para um sistema inteligente, utilizando diferentes implementações de clientes que dependem do padrão de comunicação escolhido pelo pesquisador. Ao final de uma requisição de predição, o sistema coleta e armazena métricas relacionadas à avaliação da predição para análises posteriores aos testes. O repositório do sistema pode ser acessado em: https://github.com/washington-ygor-tcc/benchmark;

2. O sistema inteligente, **Preditor**, capaz de servir modelos de aprendizado de máquina por meio de múltiplos padrões de comunicação. O repositório do sistema pode ser acessado em: https://github.com/washington-ygor-tcc/intelligent-system.

O restante do capítulo documenta a arquitetura da aplicação mediante a adoção de alguns diagramas do Modelo C4, e a adição de detalhes pertinentes às características arquiteturais e decisões de arquitetura que foram definidas.

3.1 Contexto

O diagrama presente na Figura 3.1, destaca os requisitos funcionais de cada um dos sistemas de software. Pode-se destacar que ambos os sistemas estão direta ou indiretamente relacionados com as necessidades do pesquisador. O pesquisador que usa o sistema de *Benchmark* solicita a realização de um teste de carga parametrizado. A depender do parâmetro, o *Benchmark* irá consumir o Preditor pelo padrão de comunicação adequado. Ao final, com a coleta de métricas realiza pelo sistema de *Benchmark*, o pesquisador poderá analisar e avaliar a performance do consumo de um dado modelo por diferentes padrões de comunicação.

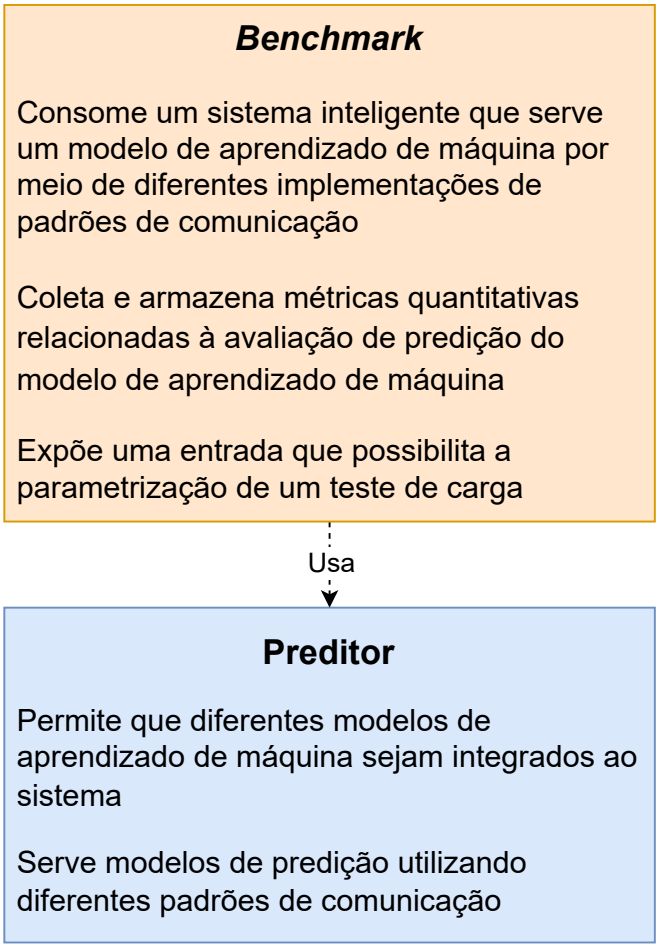

Figura 3.1: ***Diagrama de Contexto****. Representação dos requisitos funcionais e as relações dos atores da arquitetura.*

3.2 Características Arquiteturais

Analisando o contexto de ambos os sistemas (Figura 3.1), uma semelhança em comum se destaca: ambos sistemas devem dar suporte a mais de um tipo de padrão de comunicação, tanto para consumir como para servir um modelo de predição. Tal requisito foi fundamental para mapear algumas características arquiteturais (Subseção 2.3.1) tidas como essenciais para o sucesso da aplicação:

- **Extensibilidade**: É necessário replicar comportamentos idênticos para diferentes maneiras de consumir ou servir um dado modelo. Além disso, o Preditor deve ser capaz de integrar com diferentes modelos de predição.
- **Manutenibilidade**: É necessário aplicar mudanças e criar novas funcionalidades, de tal forma que a adição de um novo padrão de comunicação não implique em problemas no código já existente.

3.3 Decisões Arquiteturais

Para cumprir com os requisitos funcionais e características arquiteturais dos sistemas, a definição de algumas decisões arquiteturais foram fundamentais. Dentre elas, podemos destacar duas:

1. Adoção do estilo arquitetural Hexagonal introduzido na Subseção 2.5.2. Por meio dessa decisão, os detalhes de implementação de cada um dos padrões de comunicação ficam completamente isolados das regras do negócio de cada sistema. Assim, os sistemas ganham flexibilidade e reduzem o acoplamento em relação aos componentes relacionados à comunicação.

2. Adoção da plataforma `MLFlow`, usada pelo Preditor para gerenciar o ciclo de vida de modelos de aprendizado de máquina. Um dos principais motivos para tal decisão se dá pelo fato da plataforma ser agnóstica a qualquer biblioteca de aprendizado de máquina. Por esse motivo, os detalhes relacionados aos modelos de predição são centralizados, permitindo que o sistema preditor se aproveite do estilo arquitetural adotado, e desacople esses detalhes em um adaptador fora do domínio do sistema. Desta forma, o preditor ganha mais flexibilidade e reduz o acoplamento, pois a plataforma permite que o preditor se integre com modelos de ML de diferentes bibliotecas como `sklearn`, `XGBoost`, `PyTorch`, etc. Além dessa motivação, um outro benefício da plataforma é a criação de modelos customizáveis utilizando funções `Python` (https://www.mlflow.org/docs/latest/python_api/mlflow.pyfunc.html). Isso permite a criação de "modelos" de teste baseados em código, que não utilizam aprendizado de máquina e podem ser usados para simular modelos reais.

3.4 Contêiner

O diagrama de **Contêiner** da Arquitetura, presente na Figura 3.2, apresenta a aplicação como um todo, destacando onde se encaixa o sistema preditor, o sistema *benchmark*, e a infraestrutura necessária para o funcionamento de cada um deles.

O sistema preditor é capaz de servir modelos de aprendizado de máquina de duas maneiras: uma síncrona, via API RPC (Subseção 2.6.1) e outra, assíncrona, via sistema de mensageria (Subseção 2.6.2). Apesar da suposta limitação de haver suporte para apenas dois padrões de comunicação, a adoção do Padrão Método Fábrica (Subseção 2.4.1), do Padrão Estratégia (Subseção 2.4.2) e do estilo arquitetural Hexagonal permite a extensão da aplicação para qualquer outro padrão de comunicação de maneira simplificada.

O funcionamento do sistema *Benchmark* se baseia exclusivamente na entrada do pesquisador, sendo apenas uma interface para realizar testes de carga. Ao receber uma requisição de teste, segue-se uma sequência de passos, onde são construídas e enviadas diversas requisições de predições para o sistema Preditor de acordo com os parâmetros de entrada. Ao final de cada requisição, métricas de performance são coletadas e podem ser salvas a depender do interesse do pesquisador (a princípio é coletado apenas o tempo de resposta de cada requisição, que é armazenado em arquivos `CSV`).

Um outro ponto a se destacar são as dependências associadas ao `MLFlow`. A plataforma depende de dois tipos de infraestrutura para armazenamento: uma para persistir artefatos como os modelos (providenciada pelo `MinIO`), e outra, para persistir metadados dos modelos (providenciada pelo banco relacional `MySQL`).

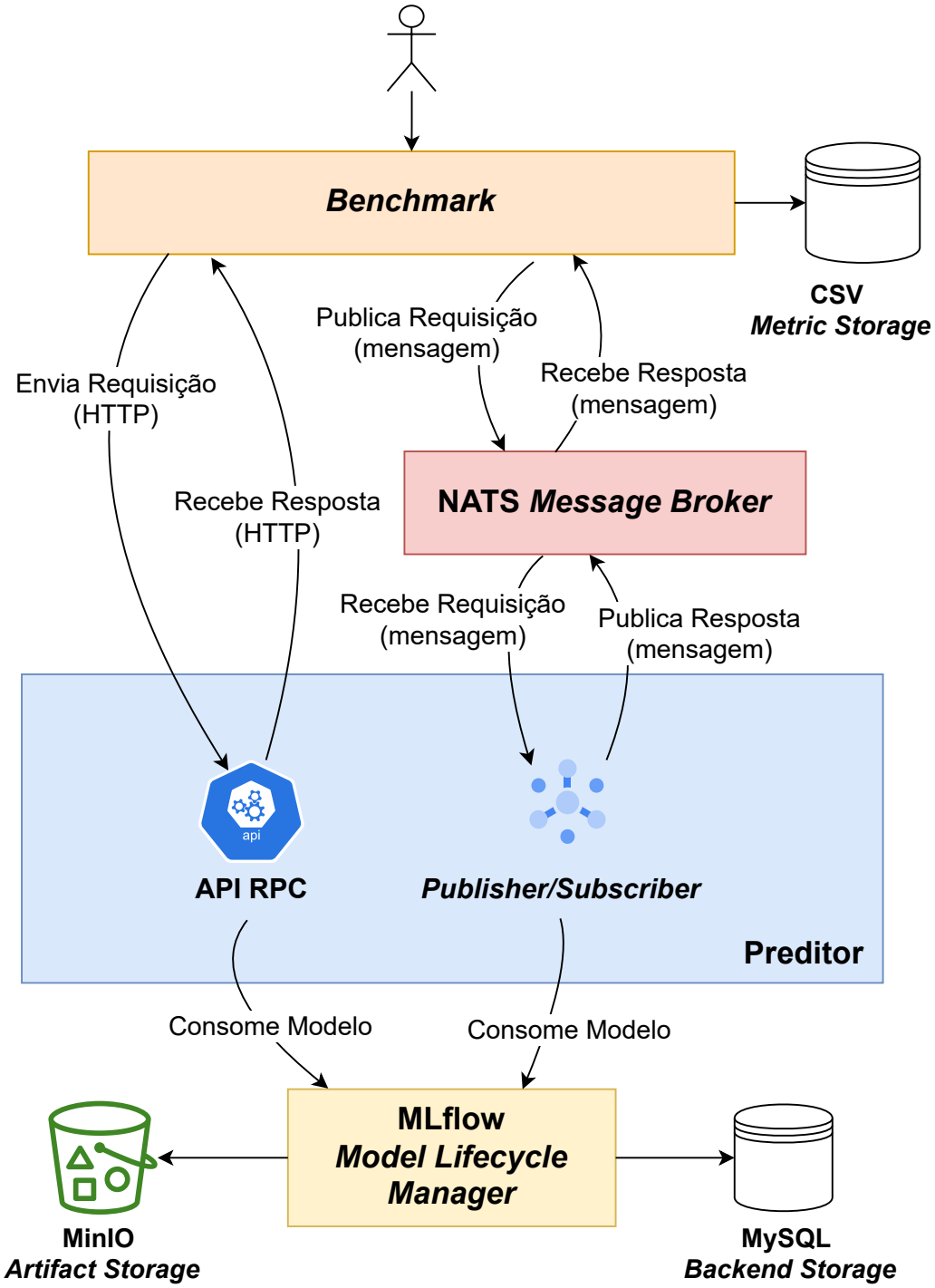

Figura 3.2: ***Diagrama de Contêiner****. Representação dos sistemas e das suas dependências de infraestrutura.*

3.5 Componente do Preditor

A Figura 3.3 exibe a camada de abstração do **Componente** do sistema inteligente, que se divide em:

- o domínio do sistema (em verde), que é responsável por obter os modelos provisionados e realizar as predições requisitadas,
- entradas do sistema (em vermelho), que expõem as funcionalidades do sistema utilizando diferentes padrões de comunicação, e
- repositório de modelos de aprendizado de máquina (em amarelo).

O domínio da aplicação é responsável por encapsular toda lógica de negócio do sistema. O seu fluxo se resume em:

1. coordenar as requisições de predições recebidas pelas portas de entrada,
2. requisitar um modelo provisionado na plataforma de gerenciamento de ciclo de vida de modelos de predição,
3. realizar a predição utilizando a entrada recebida, e
4. retornar a predição como resposta para a porta de entrada adequada.

Tal comportamento é encapsulado em um único caso de uso presente no domínio da aplicação: o `Predict Request Handler Use Case`.

As implementações dos adaptadores do sistema se baseiam em chamadas para bibliotecas ou arcabouços. Para o adaptador da porta de entrada síncrona, que serve uma API RPC, foi utilizado a biblioteca `FastAPI`, enquanto o adaptador da entrada assíncrona se baseia na implementação de um consumidor e de um produtor de mensagens que utilizam uma biblioteca para o sistema de mensageria `NATS`. Em relação ao repositório de modelos, o adaptador foi implementado utilizando a biblioteca do `MLFlow`.

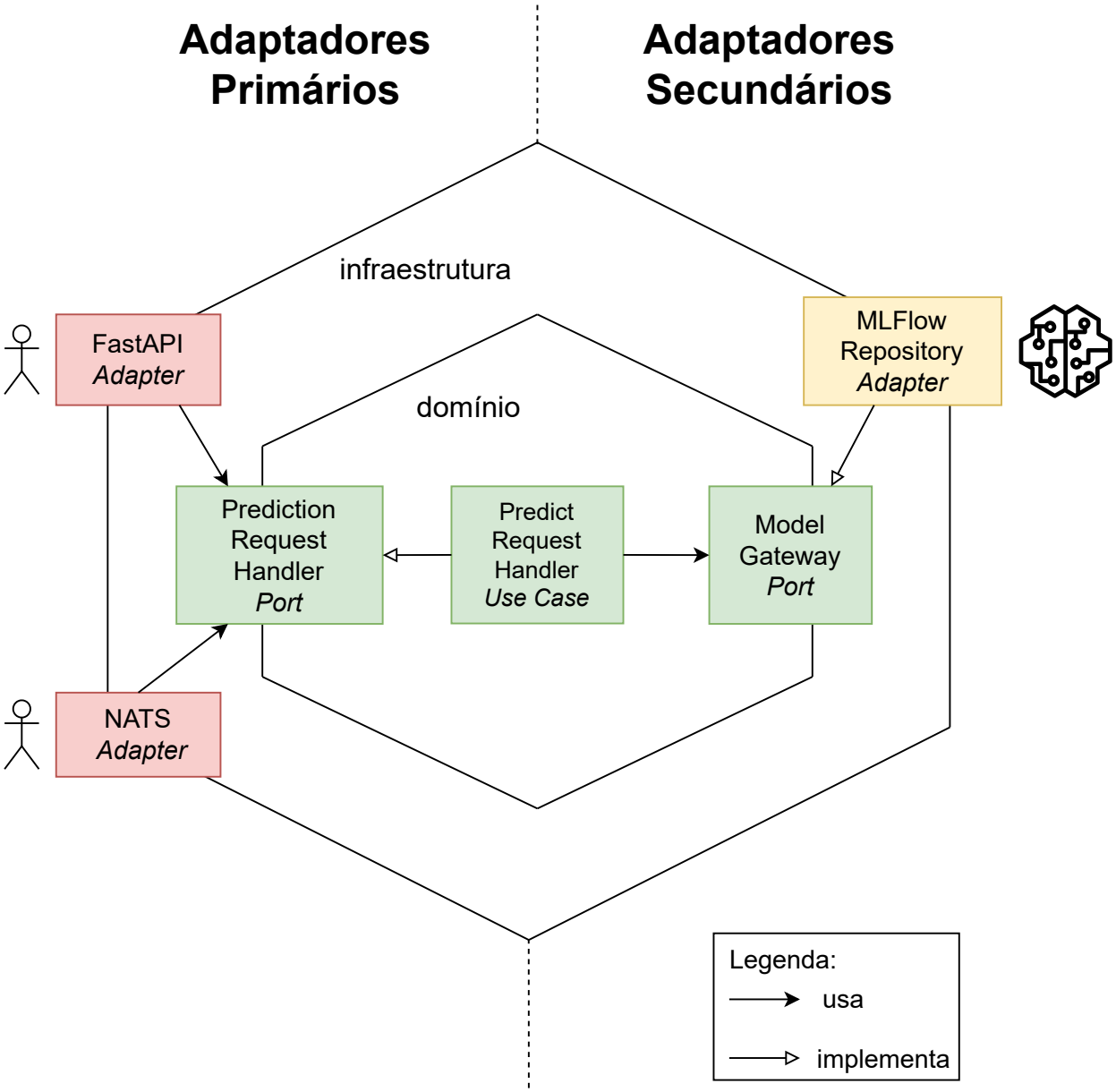

Figura 3.3: ***Diagrama de Componente****. Representação do componente núcleo do sistema inteligente.*

3.6 Componente do *Benchmark*

A Figura 3.4 exibe a camada de abstração do **Componente** do sistema de *benchmark*, que se divide em:

- o domínio do sistema (em verde), que é capaz de realizar testes parametrizados e salvar métricas relativas aos testes,
- implementações de clientes de serviço (em vermelho), que utilizam diferentes padrões de comunicação,
- um repositório de métricas (em amarelo), obtidas à partir dos testes realizados,
- um gerador de identificadores únicos (em roxo), e
- um gerador de marcação de tempo (em branco).

Um ponto a se notar é que não existem portas de entrada primárias neste sistema e, portanto, o "adaptador" (entre aspas por não implementar uma porta do domínio) `Benchmark Library` executa o fluxo interno diretamente. Tal fato se traduz em um certo relaxamento do estilo arquitetural hexagonal adotado (uma variância da decisão arquitetural). A motivação da variância se deve à natureza dos testes realizados pelo *Benchmark*, que segue passos fixos. Soma-se a isso a opção de linguagem de implementação do sistema, `Python`, que permite disparos de funções dinamicamente. Por esse motivo, existe uma dependência que liga diretamente o adaptador primário ao domínio do sistema.

O adaptador de entrada do sistema é exposto por meio de uma biblioteca de funções, podendo ser utilizada por *scripts* `Python`, *notebooks* `Jupyter`, etc. O fluxo de execução do *Benchmark* pode ser controlado por diversos parâmetros, tais como:

- tipo de padrão de comunicação (API RPC ou mensageria),
- complexidade de processamento do modelo de predição,
- quantidade de memória consumida pelo modelo de predição,
- número de requisições a serem realizadas,
- tempo em segundos para disparar requisições continuamente (usado como alternativa ao número de requisições),
- tamanho de lote de requisições a serem disparadas concorrentemente, e
- intervalo de tempo em segundos entre cada lote de requisição.

Um ponto a se destacar em relação ao adaptador de entrada do sistema é que ele atua como uma função `main` do sistema, sendo responsável por injetar as dependências das classes e executar os fluxos internos do sistema de acordo com os parâmetros de entrada do *Benchmark*. Tal decisão se enquadra como uma variação de uma decisão arquitetural, já que viola regras do estilo arquitetural adotado. Novamente, a justificativa para isso é explicada pela natureza de um sistema *Benchmark*. Ainda em relação ao adaptador de entrada, foi implementado uma interface de consumo da biblioteca que é mais amigável aos pesquisadores, via linha de comando (*Command Line Interface*, ou CLI). Maiores informações em relação ao uso do *Benchmark* via CLI podem ser encontrados no Apêndice A.

Partindo para o domínio do sistema, existem dois casos de uso implementados: o `Save Benchmark Use Case` e o `Run Benchmark Use Case`. O primeiro é responsável por salvar os resultados de todas as requisições feitas utilizando a porta de repositório de métricas. O segundo, abstrai uma requisição de predição – agnóstica ao padrão de comunicação – feita para o sistema inteligente:

- gera um identificador único para a requisição em questão utilizando a porta que gera um identificador,
- inicia a contagem de tempo utilizando a marcação de tempo implementada pela porta que gera marcações de tempo,
- realiza a requisição de predição utilizando a porta que abstrai requisições de predição,
- agrega a contagem de tempo ao final da requisição, e
- retorna o resultado da requisição.

A geração de métricas de tempo e identificadores únicos foi separada em dois adaptadores. Tal decisão foi motivada porque esse ponto é um detalhe que pode ser desacoplado e testado à parte do domínio. Além disso, vale destacar que os componentes tem uma natureza aleatória, e o desacoplamento dos mesmos facilita a criação de testes de unidade sem estado. A implementação manteve a chamada de bibliotecas padrões da linguagem (`time` e `uuid`), pois uma implementação dos adaptadores diferente da biblioteca padrão não iria gerar valor para o objetivo do projeto.

Da mesma forma que o sistema inteligente expõe duas entradas (via API RPC e mensageria), o sistema *Benchmark* implementa dois clientes de serviço distintos que são capazes de consumir cada uma das entradas. Cada um deles é uma implementação da porta de requisições de predições. A instanciação do adaptador específico depende exclusivamente do parâmetro de entrada do tipo de padrão de comunicação a ser adotado para realizar as requisições.

Finalmente, a implementação da porta do repositório de métricas é apenas um adaptador para criar um arquivo `CSV`.

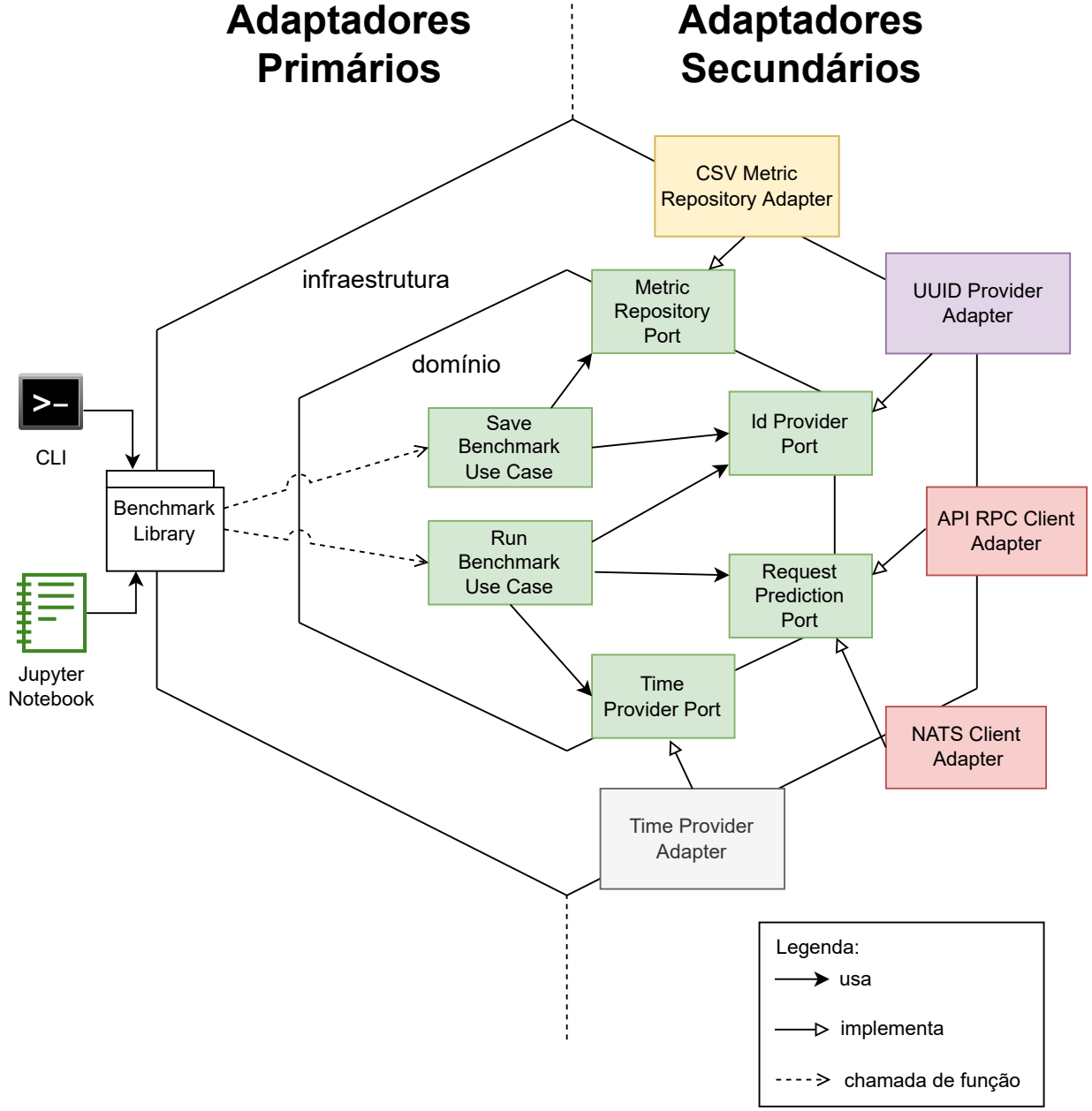

Figura 3.4: ***Diagrama de Componente****. Representação do componente núcleo do sistema de benchmark.*

Capítulo 4

Experimentos

Utilizando o sistema *Benchmark* implementado, este capítulo apresenta experimentos realizados para responder questões que tem o objetivo de **entender as vantagens e desvantagens de diferentes padrões de comunicação (via API RPC e mensageria) ao realizar requisições de predição para um sistema inteligente**. Cada experimento se associa a um questionamento, apresentando diferentes métricas quantitativas e observações encontradas durante a experimentação que contribuem para análise dos resultados.

4.1 Ambiente de Testes

A coleta de métricas foi feita num mesmo ambiente computacional com as seguintes características:

- **Sistema Operacional**: Manjaro Linux 5.15.78-1.
- **CPU**: Intel Core i7-9700KF CPU @ 3.60GHz, arquitetura x86-64 com 8 núcleos de processamento.
- **Memória RAM**: 16 GB.
- **Placa de Rede**: Intel Ethernet I219-V 1GbE.

Um ponto a se destacar é que o *Benchmark* e Preditor foram executados concorrentemente no mesmo ambiente, dentro de contêineres `Docker`.

4.2 Modelo de Predição Adotado

A princípio, havia uma ideia de utilizar modelos de predição reais. Houve uma tentativa de experimentação com modelos de regressão para um problema apresentado no `Kaggle`, que propõe predizer estimativas de tempo para viagens de táxi na cidade de Nova Iorque (https://www.kaggle.com/competitions/nyc-taxi-trip-duration/overview). No entanto, a dificuldade para controlar diferentes níveis de complexidade de processamento e memória ao realizar uma predição se mostrou impeditivo para tal abordagem. Dessa forma, uma simulação de modelo de ML foi criada e utilizada para a realização de todos os experimentos. Tal decisão se relaciona com uma das decisões arquiteturais (Subseção 2.3.2) tomadas, já que o `MLFlow` possibilita a criação e o provisionamento desse tipo de modelo em sua plataforma.

O modelo de predição adotado soluciona um sistema linear utilizando a biblioteca `NumPy`. A entrada do modelo espera um objeto com dois parâmetros: `complexity_factor` (CF) e `memory_overhead` (MO), que definem, respectivamente, a complexidade de processamento e uso de memória. O algoritmo cria uma matriz quadrada A e um vetor b com dimensões dada pelo valor CF, um vetor *dummy* (sem uso) de dimensão MO, e por fim, resolve o sistema linear $Ax = b$. Desta

forma, o modelo descreve uma complexidade de processamento dada por $\mathcal{O}(CF^3)$ e de espaço utilizado por $\mathcal{O}(CF^2 + MO)$. A implementação do modelo de simulação pode ser encontrada em https://github.com/washington-ygor-tcc/intelligent-system/blob/main/mlflow/log_pyfunc_model.py.

4.3 Primeiro Experimento

Este experimento busca responder a seguinte questão: **Qual padrão de comunicação têm maior capacidade de concorrência?** Para respondê-la, as seguintes métricas quantitativas podem contribuir para sua solução:

- tempo de requisição,
- desvio padrão do tempo de requisição,
- requisições por segundo, e
- tempo total para realizar um conjunto de requisições.

4.3.1 Parametrização do *Benchmark*

O parâmetro de **tamanho de lote de requisições** executadas concorrentemente é o parâmetro chave para este experimento. Para cada padrão de comunicação, tal parâmetro foi variado seguindo uma sequência finita de potências de 2, definida por: $a_n = 2^n, 0 \leq n \leq 12$.

Para cada permutação de tamanho de lote de requisições e padrão de comunicação, um valor fixo de requisições definido por 4096 foram distribuídas entre os diferentes tamanhos de lote, com cada requisição utilizando um modelo de predição com fator de complexidade de processamento igual a 100. A motivação desses valores fixos se deve, respectivamente, a coincidência com o valor máximo de loteamento das requisições de 2^{12} (um valor maior exigiria um ambiente de testes com mais recursos) e a replicação de processamento de um modelo simples (observado empiricamente em um modelo de regressão).

4.3.2 Resultados

Os dados coletados buscam refletir as métricas associadas a questão em gráficos. A Figura 4.1 apresenta *boxplots* da distribuição formada por 4096 requisições para cada tamanho de lote definido. A Figura 4.2 apresenta um gráfico que relaciona o tempo total consumido para executar cada conjunto de 4096 requisições para cada tamanho de lote definido. A Figura 4.3 apresenta um gráfico que relaciona a quantidade de requisições realizadas por segundo para cada tamanho de lote definido. A Figura 4.4 apresenta um gráfico que relaciona o desvio padrão do tempo de resposta observado para cada tamanho de lote definido. O código responsável por obter os dados e desenhar os gráficos do experimento pode ser acessado em:
https://github.com/washington-ygor-tcc/benchmark-notebook/tree/main/experiments/1.

4.3.3 Observações

Além das métricas quantitativas observadas, alguns comportamentos discrepantes foram observados durante a experimentação, podendo agregar às análises posteriores:

- A API RPC exigiu uma grande quantidade de descritores de arquivo do sistema operacional (SO), o que era esperado já que cada requisição em aberto exige um *socket* de rede conectado. Por esse motivo, antes de iniciar a experimentação, foi necessário aumentar o limite de criação de descritores de arquivo do SO.
- Em alguns momentos, ao usar a API RPC, as elevadas cargas de requisições provocou quedas de conexões a nível de camada de transporte. O erro em questão é descrito pela mensagem

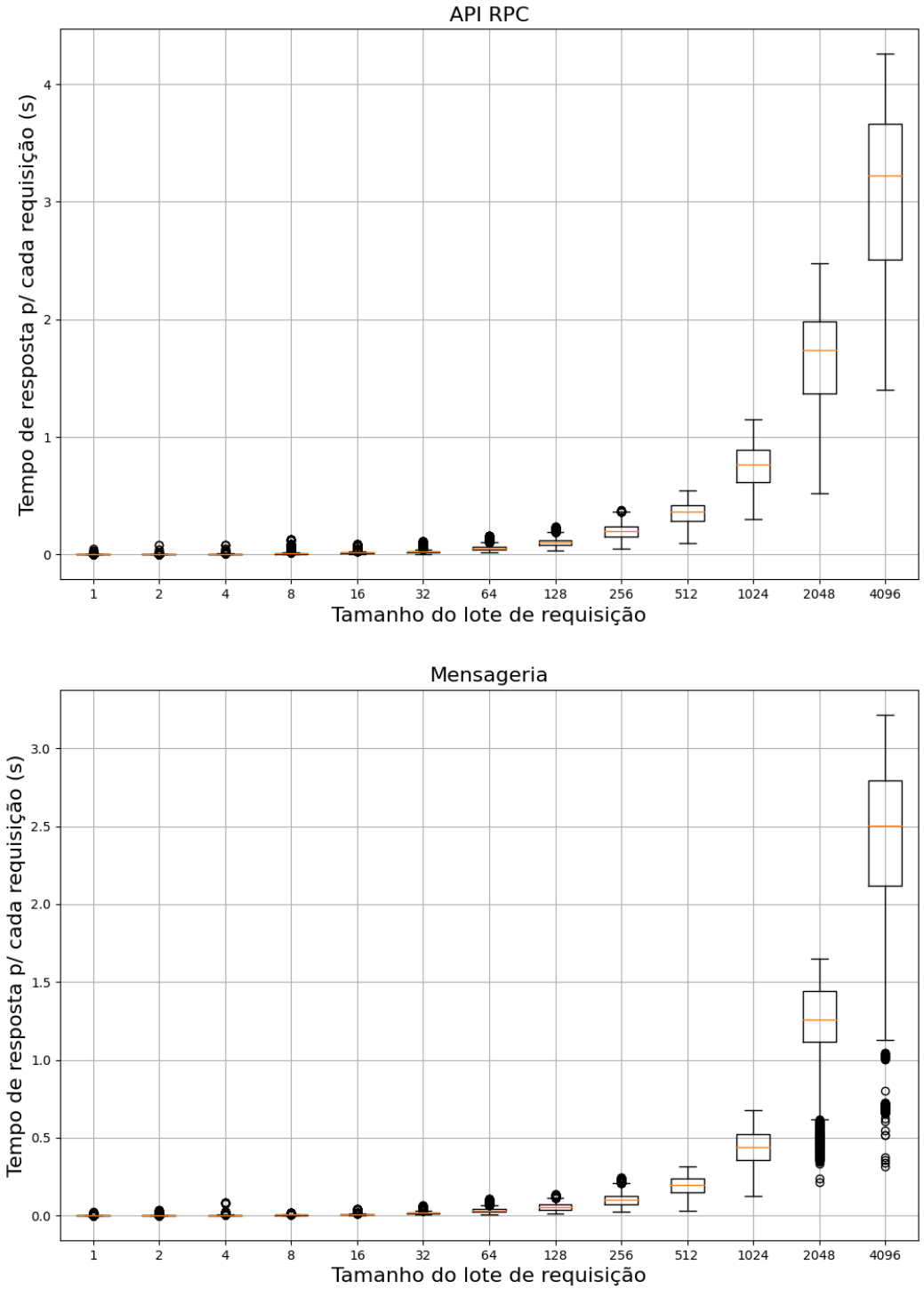

Figura 4.1: *Boxplots que definem a distribuição de 4096 requisições distribuídas por diferentes tamanhos de lote.*

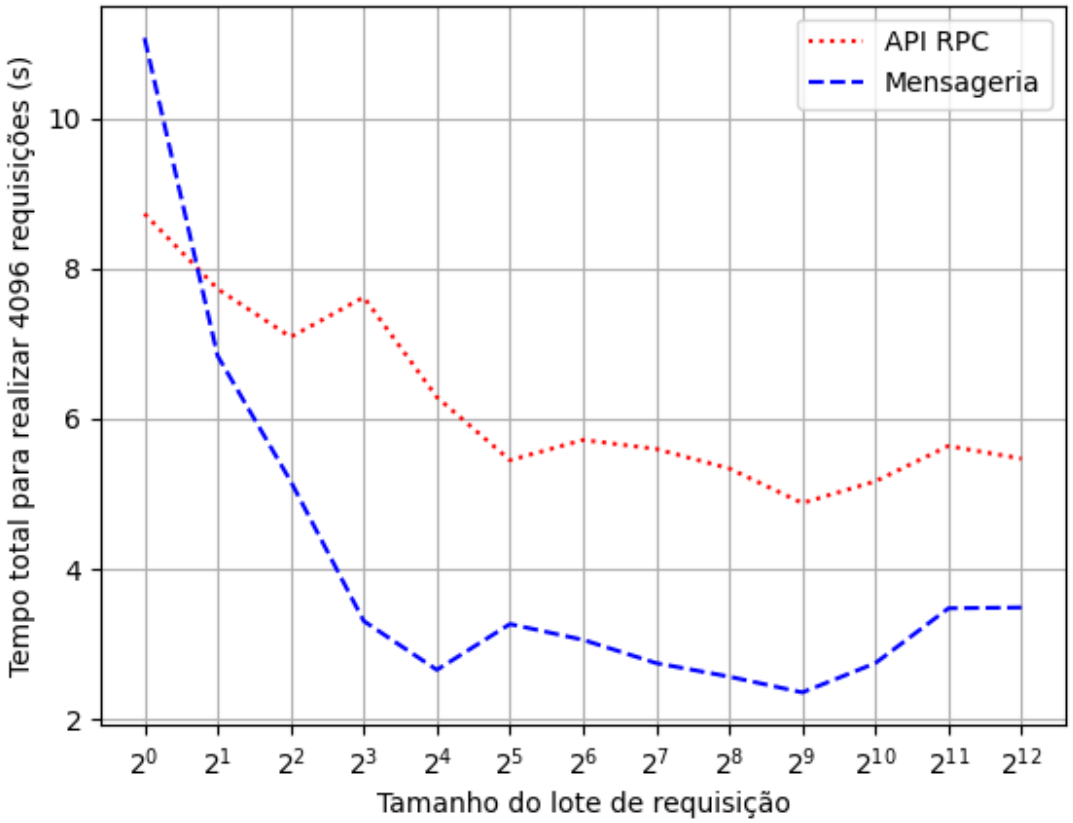

Figura 4.2: *Tempo total utilizado para executar cada conjunto de 4096 requisições distribuídas por diferentes tamanhos de lote.*

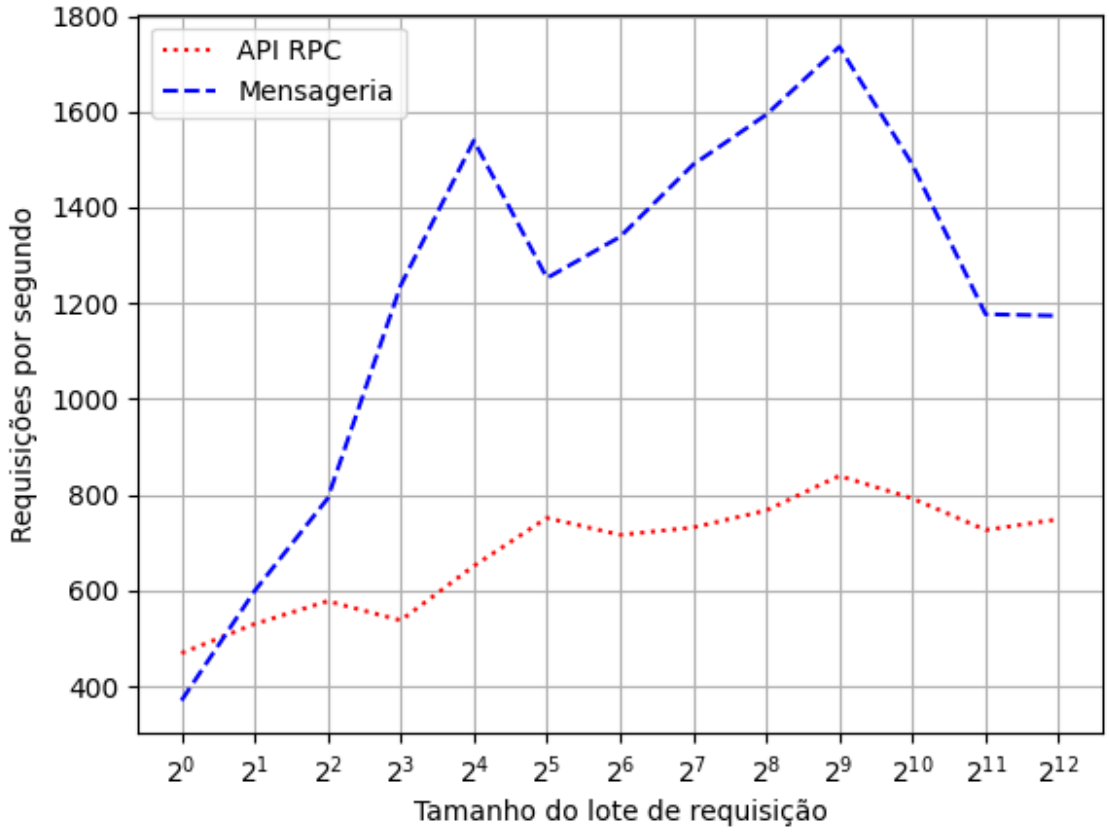

Figura 4.3: *Requisições feitas por segundo para diferentes tamanhos de lote.*

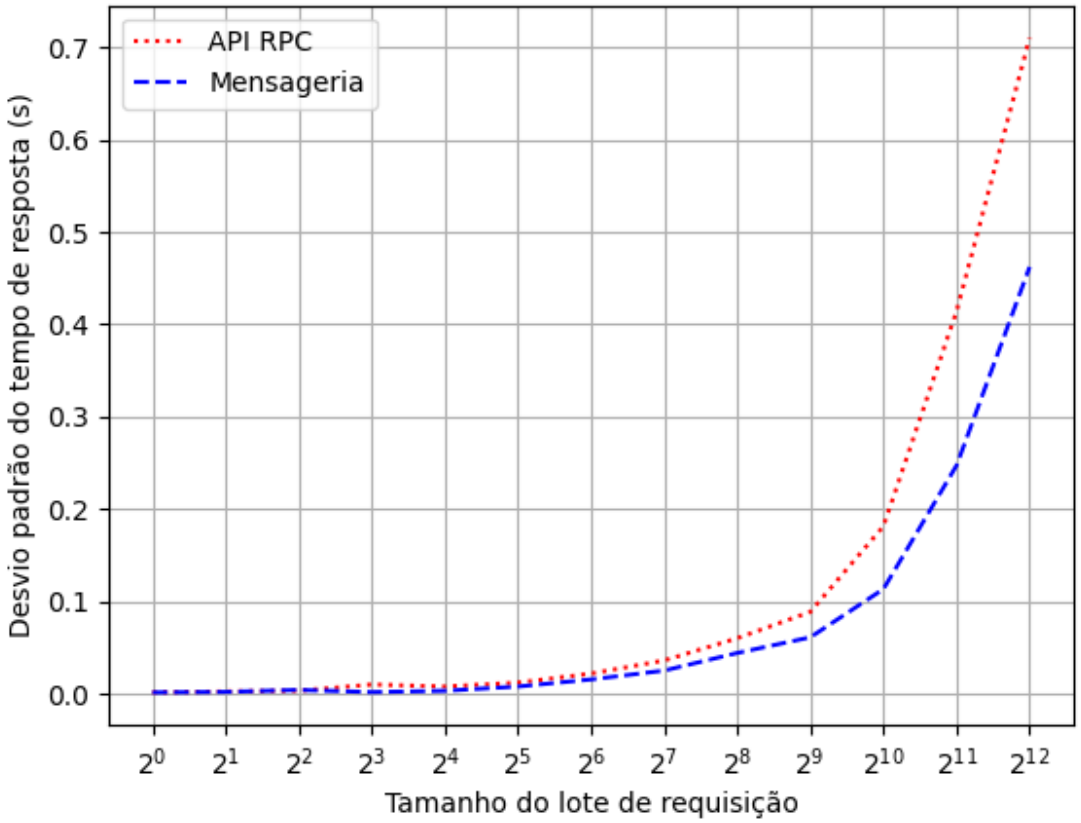

Figura 4.4: *Desvio padrão do tempo de execução de 4096 requisições distribuídas por diferentes tamanhos de lote.*

"Connection reset by peer" e está relacionado ao esgotamento do limite de tempo de uma conexão TCP. Para contornar o erro, foi necessário inserir um intervalo de tempo *sleep* entre cada teste.

4.3.4 Análise

Analisando a Figura 4.1, um ponto esperado a ser notado é que para ambos os padrões de comunicação, o tempo de resposta aumenta conforme há maiores lotes de requisições sendo executadas concorrentemente. Além disso, é possível tirar uma conclusão importante relativa aos formatos dos *boxplots*: os limites inferiores, superiores, e a amplitude interquartil do *boxplot* crescem conforme há maior nível de concorrência. A partir dessa observação, podemos concluir que conforme há mais requisições sendo feitas concorrentemente, o preditor apresenta uma maior dispersão relativa ao tempo de resposta. A Figura 4.4 deixa perceptível o quanto a dispersão da distribuição aumenta com maiores níveis de concorrência, principalmente quando o padrão de comunicação utilizado é a API RPC. Um outro ponto a se destacar em relação aos *boxplots* se refere à quantidade de pontos discrepantes abaixo do limite superior para a mensageria em cenários com grande nível de concorrência (tamanho de lote 2^{11} e 2^{12}). Isso apresenta indícios de que o sistema de mensageria – o *broker* – está cada vez mais sobrecarregado e precisa ser escalonado em recursos ou novas instâncias.

Analisando a Figura 4.2 e a Figura 4.3, a performance da mensageria se mostrou melhor em todos os cenários que houve requisições sendo feitas concorrentemente (tamanho de lote ≥ 2). A mensageria foi capaz de atingir um pico de aproximadamente 1720 requisições atendidas por segundo quando as requisições foram separadas em lotes de tamanho 512, sendo uma diferença maior que o dobro do pico de 820 requisições atendidas por segundo pela API RPC para lotes de tamanho 512. O parâmetro ótimo de loteamento de ambos padrões, em 512, é um ponto de interrogação. Não foi encontrado uma hipótese clara para explicar tal coincidência.

Apesar da diferença entre os padrões de comunicação quando as requisições são atendidas concorrentemente, quando o preditor teve que responder requisições sequencialmente (tamanho de lote = 1), a API RPC teve uma performance um pouco superior à mensageria, já que ela conseguiu responder as 4096 requisições num menor tempo e com uma maior taxa de requisições antedidas por segundo.

4.3.5 Conclusões

Por uma larga vantagem, é possível concluir que o padrão via mensageria é o mais performático em cenários que o preditor precisa atender mais de uma requisição ao mesmo tempo. Essa conclusão deriva de dois fatos observados nos gráficos: conseguir atender mais rapidamente as requisições por conseguir maiores taxas de requisições respondidas por segundo, e apresentar menor degradação da performance por apresentar menor dispersão para maiores quantidades de requisições sendo feitas concorrentemente. Pelas observações apontadas na Subseção 4.3.3, é indicado que o consumo concorrente em larga escala da API RPC pode exigir mais recursos – computacionais e de rede – do que a mensageria se levarmos em consideração apenas a infraestrutura do sistema inteligente.

Um ponto que merece destaque é de que a performance da API RPC para atender requisições uma a uma foi superior (tamanho de lote = 1). Tal fato aumenta o número de cenários em que a API RPC pode ser uma melhor escolha se comparada a um sistema que serve um modelo via mensageria.

4.4 Segundo Experimento

Este experimento busca responder a seguinte questão: **Qual padrão de comunicação têm maior capacidade para atender grandes volumes de requisições?** Para respondê-la, as seguintes métricas quantitativas podem contribuir para sua solução:

- média do tempo de requisição,
- desvio padrão do tempo de requisição, e
- requisições por segundo.

4.4.1 Parametrização do *Benchmark*

O parâmetro de **quantidade de requisições** a serem executadas é o parâmetro chave para este experimento. Para cada padrão de comunicação, tal parâmetro foi variado seguindo uma progressão aritmética finita com valor inicial e razão igual a 1000, que pode ser definida por: $a_n = 1000n, 1 \leq n \leq 25$.

Para cada permutação de quantidade de requisição e padrão de comunicação, um valor fixo de loteamento concorrente das requisições foi definido como 500, com cada requisição utilizando um modelo de predição com fator de complexidade de processamento igual a 100. A motivação do valor utilizado para lotear as requisições se deve em virtude aos resultados do primeiro experimento (4.3). Quando o lote foi definido como $2^9 \approx 500$, ambos os padrões de comunicação tiveram a sua melhor performance. Assim, devido à pouca importância do parâmetro para o nosso questionamento, foi decidido fixar o parâmetro de loteamento no valor ótimo. Em relação à fixação do parâmetro de complexidade de processamento, a motivação é a mesma apresentada no primeiro experimento.

4.4.2 Resultados

Os dados coletados buscam refletir as métricas associadas à questão em gráficos. A Figura 4.5 apresenta um gráfico com a média de tempo necessária para executar uma dada quantidade de requisições. A Figura 4.6 apresenta um gráfico que relaciona a quantidade de requisições realizadas por segundo para executar uma dada quantidade de requisições. A Figura 4.7 apresenta um gráfico com o desvio padrão do tempo de resposta observado para executar uma dada quantidade de requisições. O código responsável por obter os dados e desenhar os gráficos do experimento pode ser acessado em:
https://github.com/washington-ygor-tcc/benchmark-notebook/tree/main/experiments/2.

4.4.3 Observações

Além das métricas quantitativas observadas, um comportamento discrepante foi observado durante a execução dos testes para a API RPC. Para grandes volumes de requisições (quantidade de requisições acima de 28000), houve erros que provocaram quedas de conexões a nível de camada de transporte. O erro em questão é descrito pela mensagem *"Connection reset by peer"* e está relacionado ao esgotamento do limite de tempo de uma conexão TCP. Para contornar o erro, foi necessário reduzir o volume máximo de requisições a serem realizadas e inserir um intervalo de tempo *sleep* entre cada teste.

4.4.4 Análise

Analisando a Figura 4.5, é possível destacar que a mensageria foi capaz de atender as requisições com uma média de tempo inferior, e em muitos momentos teve uma performance quase que duas vezes superior à API RPC. Um outro ponto a ser destacado é a instabilidade apresentada pela API RPC, já que sua curva apresentou diversos pontos mínimos e máximos.

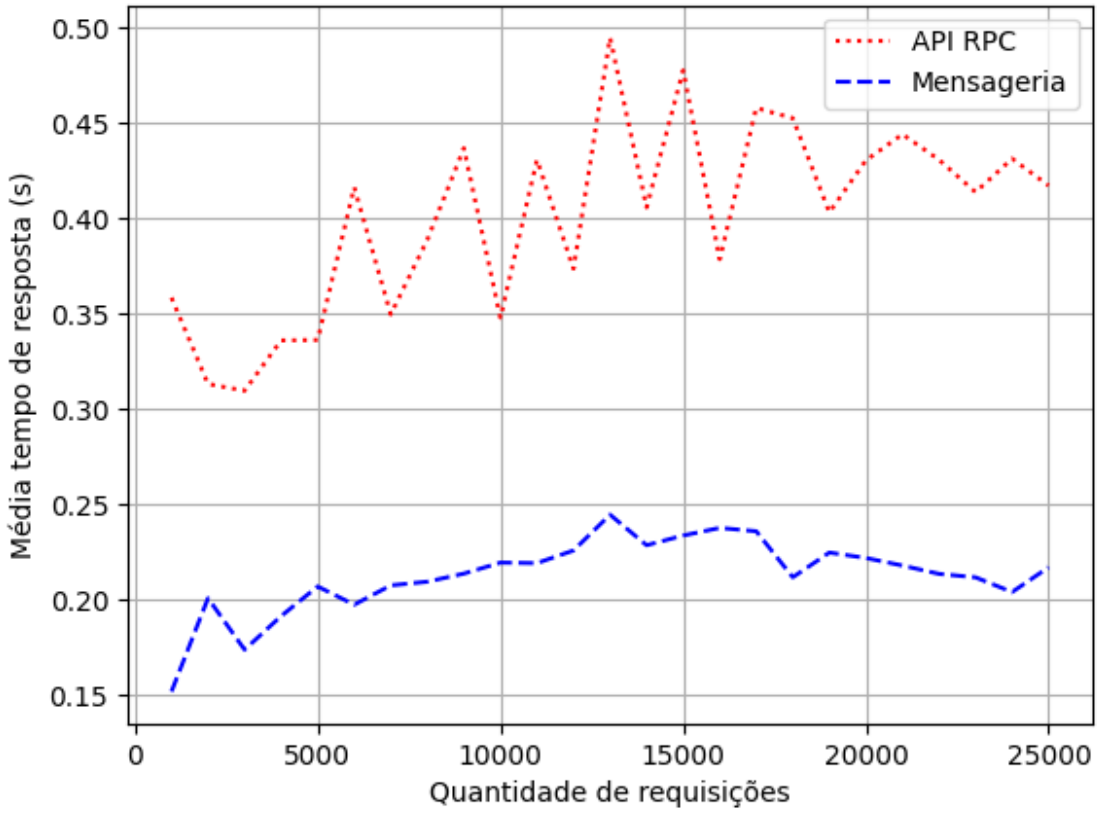

Figura 4.5: *Média de tempo de resposta de requisições em função da quantidade de requisições realizadas.*

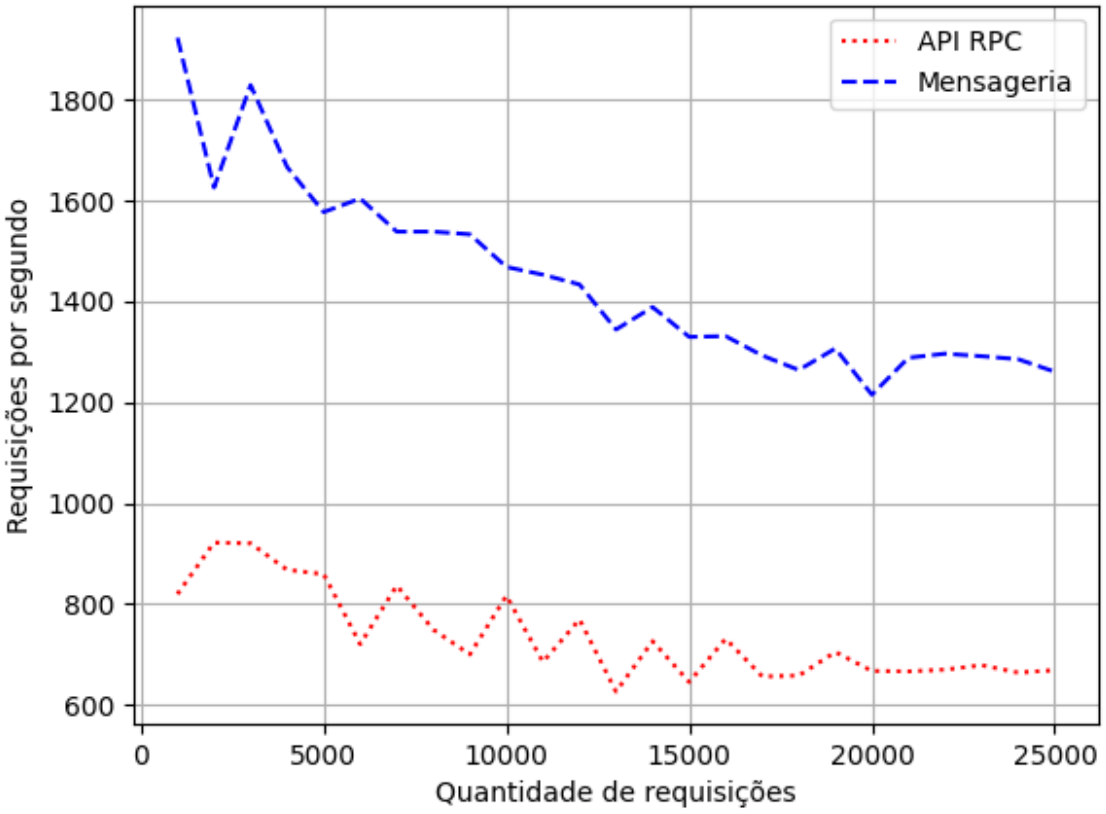

Figura 4.6: *Requisições por segundo em função da quantidade de requisições realizadas.*

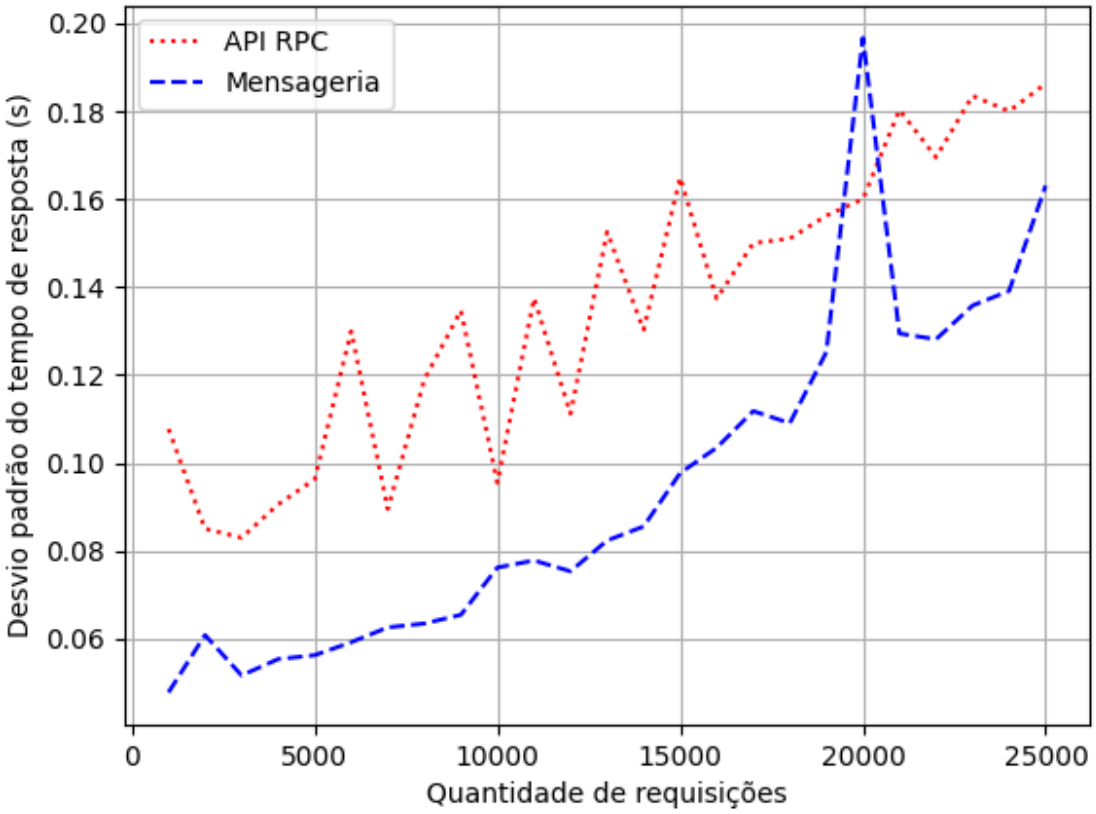

Figura 4.7: *Desvio padrão observado para a distribuição de requisições realizadas em função da quantidade de requisições realizadas.*

Pela Figura 4.6, a melhor performance da mensageria é evidenciada com altas taxas de requisições por segundo. Um ponto a se notar, no entanto, é que a curva da mensageria claramente apresenta uma taxa de decrescimento maior se comparado a API RPC. Isso pode ser preocupante, já que o escalonamento do sistema de mensageria envolve maiores custos, e também há imprevisibilidade da curva aumentar a sua taxa de decrescimento ou criar um ponto de inflexão que se traduz em indisponibilidade do sistema.

A Figura 4.7 mostra que ambos os padrões de comunicação se degradam conforme o volume de requisições aumenta. A curva da mensageria reitera a preocupação apresentada pelo gráfico de requisições atendidas por segundo. O desvio padrão do tempo de resposta observado cresce e se torna cada vez mais próximo da curva da API RPC. Inclusive, houve volumes de requisição em que o desvio da mensageria foi superior ao da API RPC.

4.4.5 Conclusões

A mensageria se mostrou mais performática em relação a API RPC para atender maiores volumes de requisições com um tempo de resposta inferior. É importante ressaltar que a degradação do sistema de mensageria se mostrou mais acentuada.

A opção de escolha pela API RPC pode ser a melhor quando levado em consideração os custos para manter o sistema e cenários onde não há necessidade de atender um alto volume de requisições num curto período de tempo (limitação apresentada pela observação da Subseção 4.4.3).

4.5 Terceiro Experimento

Este experimento busca responder a seguinte questão: **Existe alguma diferença de performance para servir modelos de ML com diferentes níveis de complexidade de processamento utilizando um padrão de comunicação em específico?** Para respondê-la, as seguintes métricas quantitativas podem contribuir para sua solução: tempo total para realizar uma dada quantidade de requisições, e desvio padrão do tempo de requisição.

4.5.1 Parametrização do *Benchmark*

O parâmetro de **complexidade de processamento do modelo de predição** é o parâmetro chave para este experimento. Para cada padrão de comunicação, tal parâmetro foi variado seguindo uma progressão aritmética finita com valor inicial e razão igual a 100, que pode ser definida por: $a_n = 100n, 1 \leq n \leq 40$.

Para cada permutação de complexidade de processamento do modelo de predição e padrão de comunicação, um valor fixo de loteamento concorrente das requisições foi definido como 2 para separar um total de 100 requisições. A motivação do valor utilizado para lotear as requisições se deve em virtude aos resultados do primeiro experimento (4.3). Quando o lote foi definido como 2, a performance de ambos os padrões foi o mais próxima possível (ambas as curvas se cruzam perto desse ponto). Dessa forma, o objetivo da escolha foi de reduzir a interferência do parâmetro de concorrência sobre os resultados do experimento. Ademais, a quantidade pequena de requisições (100), é explicada pelo alto uso de recursos para altos valores do parâmetro de complexidade de processamento do modelo de predição e a limitação do ambiente de testes.

4.5.2 Resultados

Os dados coletados buscam refletir as métricas associadas à questão em gráficos. A Figura 4.8 apresenta um gráfico com o tempo total para realizar 100 requisições em função da complexidade de processamento do modelo de predição. A Figura 4.9 apresenta um gráfico com o desvio padrão do tempo de resposta observado para executar 100 requisições em função da complexidade de processamento do modelo de predição. O código responsável por obter os dados e desenhar os gráficos do experimento pode ser acessado em:
https://github.com/washington-ygor-tcc/benchmark-notebook/tree/main/experiments/3.

4.5.3 Análise

Analisando a Figura 4.8 e a Figura 4.9, um único ponto a se notar é de que as curvas de ambos padrões de comunicação se comportaram de maneira semelhante conforme a complexidade de processamento do modelo de predição cresceu.

O formato das curvas é explicado pela complexidade de execução do algoritmo que o modelo de teste implementa, no caso, $\mathcal{O}(n^3)$.

4.5.4 Conclusões

Tomando como consideração apenas o tempo de resposta das requisições, não há diferenças entre ambos padrões comunicações para atender requisições de predição de modelos com diferentes níveis de complexidade de processamento.

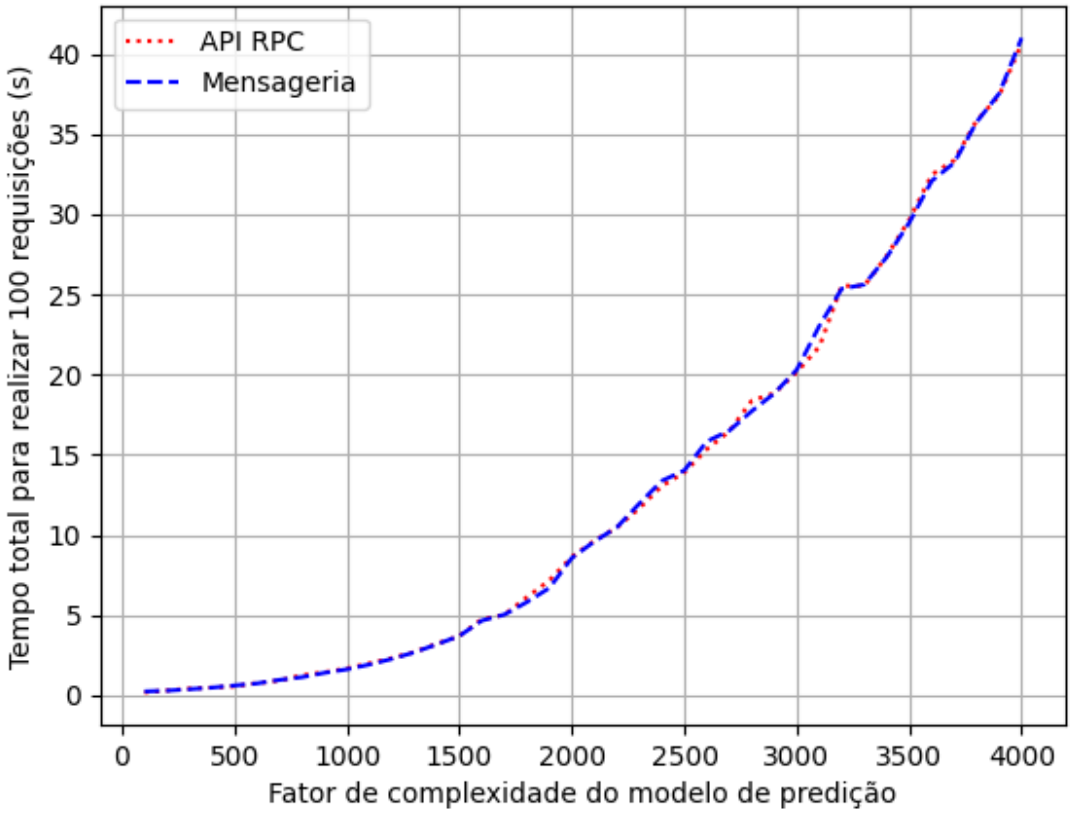

Figura 4.8: *Tempo total para responder* 100 *requisições em função da complexidade de processamento do modelo de predição.*

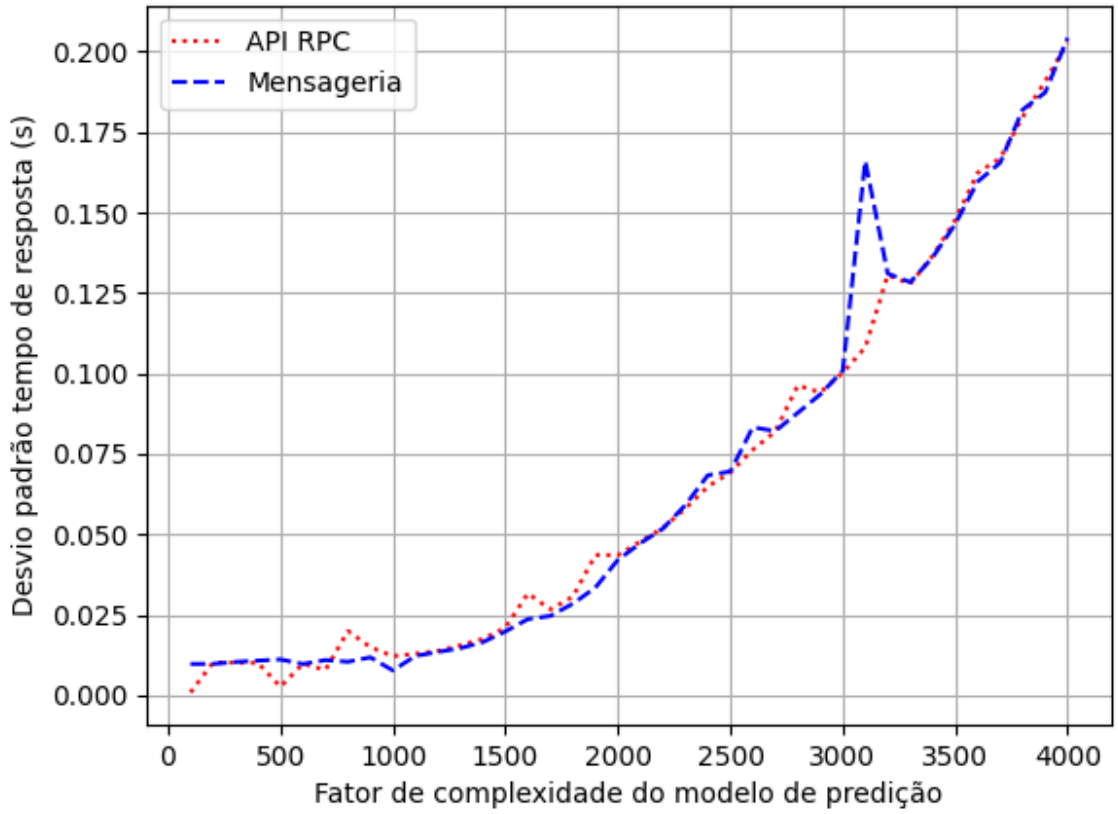

Figura 4.9: *Desvio padrão observado para a distribuição de* 100 *requisições realizadas em função da complexidade de processamento do modelo de predição.*

4.6 Quarto Experimento

Este experimento busca responder a seguinte questão: **Existe alguma diferença de performance para servir modelos de ML com diferentes níveis de consumo de memória utilizando um padrão de comunicação em específico?** Para respondê-la, as seguintes métricas quantitativas podem contribuir para sua solução: tempo total para realizar uma dada quantidade de requisições, e desvio padrão do tempo de requisição.

4.6.1 Parametrização do *Benchmark*

O parâmetro de **fator de uso de memória pelo modelo de predição** é o parâmetro chave para este experimento. Para cada padrão de comunicação, tal parâmetro foi variado seguindo uma progressão aritmética finita com valor inicial e razão igual a 100, que pode ser definida por: $a_n = 100n, 1 \leq n \leq 20$.

Para cada permutação de fator de uso de memória pelo modelo de predição e padrão de comunicação, um valor fixo de loteamento concorrente das requisições foi definido como 2 para separar um total de 100 requisições. A motivação do valor utilizado para lotear as requisições se deve em virtude dos resultados do primeiro experimento (4.3). Quando o lote foi definido como 2, a performance de ambos os padrões foi o mais próxima possível (ambas as curvas se cruzam perto desse ponto). Dessa forma, o objetivo da escolha foi de reduzir a interferência do parâmetro de concorrência sobre os resultados do experimento. Ademais, a quantidade pequena de requisições (100), é explicada pelo alto uso de recursos para altos valores do parâmetro de consumo de memória pelo modelo de predição e a limitação do ambiente de testes.

Um outro ponto a se destacar é de que o parâmetro de complexidade do modelo de predição foi fixado em 3. Assim, o consumo de memória é apenas influenciado pelo parâmetro que controla o uso de memória. Além disso, o tempo de execução do algoritmo será influenciado apenas pelo uso de memória, já que a resolução de um sistema linear de dimensão 3 é trivial.

4.6.2 Resultados

Os dados coletados buscam refletir as métricas associadas à questão em gráficos. A Figura 4.10 apresenta um gráfico com o tempo total para realizar 100 requisições em função do fator de uso de memória pelo modelo de predição. A Figura 4.11 apresenta um gráfico com o desvio padrão do tempo de resposta observado para executar 100 requisições em função do fator de uso de memória pelo modelo de predição. O código responsável por obter os dados e desenhar os gráficos do experimento pode ser acessado em:
https://github.com/washington-ygor-tcc/benchmark-notebook/tree/main/experiments/4.

4.6.3 Análise

Analisando a Figura 4.8 e a Figura 4.9, nota-se que as curvas de ambos padrões de comunicação se comportaram de maneira semelhante conforme o uso de memória da predição cresceu. Além disso, apesar da complexidade de processamento do modelo ser constante, o tempo total para executar todas requisições cresceu conforme o uso de memória do modelo cresceu.

O formato das curvas é explicado pela complexidade de espaço do algoritmo que o teste implementa, que é linear quando desconsideramos o parâmetro de complexidade de processamento.

4.6.4 Conclusões

Considerando apenas o tempo de resposta das requisições, não há diferenças significativas entre os padrões de comunicações para atender requisições de predição com diferentes níveis de uso de memória. Porém, quanto maior o consumo de memória, pior será a performance para atender as requisições de predição.

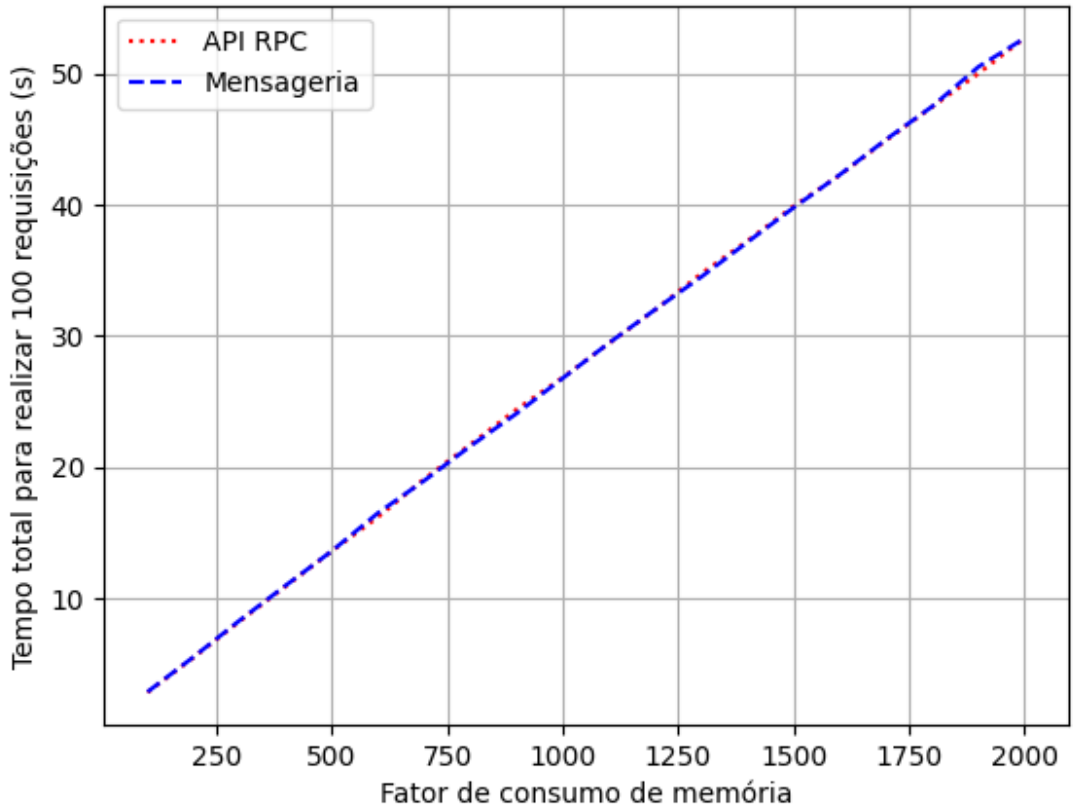

Figura 4.10: *Tempo total para responder* 100 *requisições em função do fator de uso de memória pelo modelo de predição*

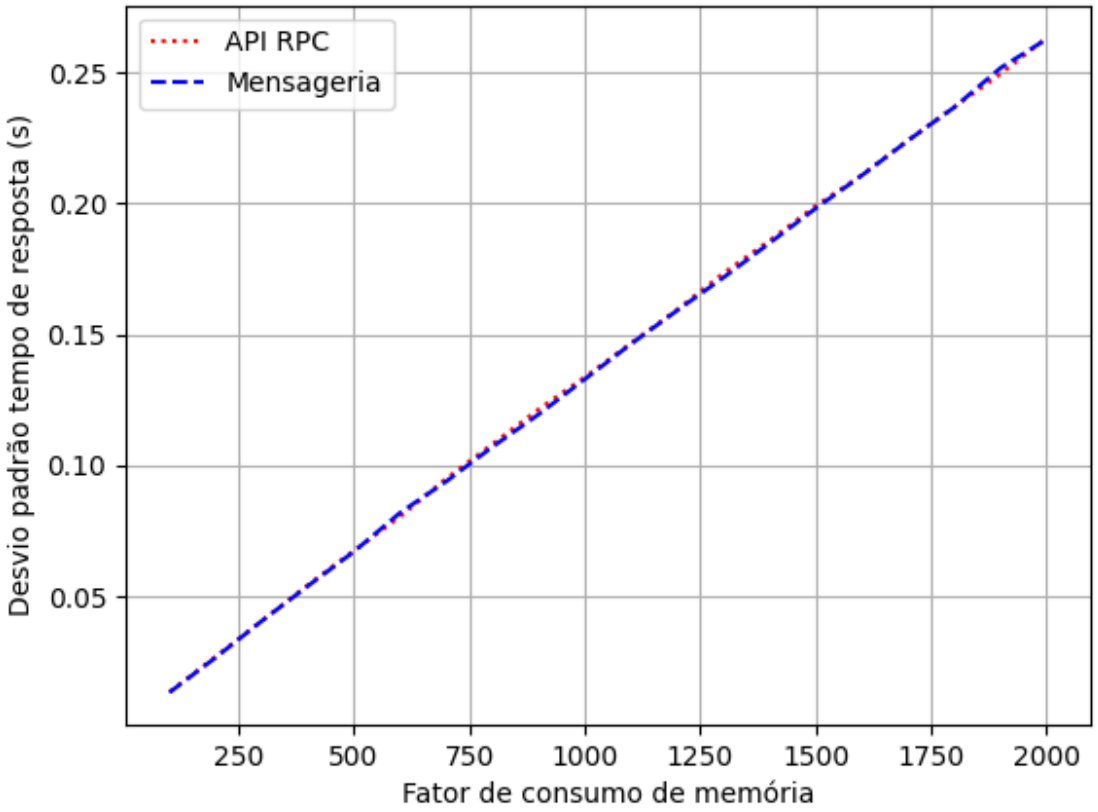

Figura 4.11: *Desvio padrão observado para a distribuição de* 100 *requisições realizadas em função do fator de uso de memória pelo modelo de predição.*

Capítulo 5

Conclusões e considerações

5.1 Conclusão

Considerando que o tipo de responsividade é o único aspecto de um padrão de comunicação considerada para avaliar as diferenças dos padrões para servir um modelo de aprendizado de máquina, a conclusão é evidente: o padrão de comunicação assíncrono é melhor que o síncrono. Em cenários de concorrência e grandes volumes de requisições, onde o tempo de resposta é a única métrica avaliada, por uma grande diferença o sistema que serve um modelo de predição via mensageria (assíncrono) se mostrou superior a um sistema que serve um modelo utilizando um padrão síncrono.

Alguns conhecimentos sobre tecnologias para servir sistemas ainda se mantém válidas para sistemas inteligentes. Por esse motivo, nem sempre o padrão assíncrono pode ser considerado o melhor para servir o modelo de um sistema inteligente. Por exemplo, acrescentando complexidade de desenvolvimento, custos com infraestrutura e manutenção, etc, os padrões síncronos tendem a ser superiores por não serem acoplados a um sistema intermediário que gerencia a troca de informação entre o cliente e o sistema em si. Além disso, o uso dos sistemas em ambiente de produção também deve ser levado em conta. Nem sempre um sistema inteligente irá operar nos seus limites tendo que lidar com altas cargas de requisições ou alto consumo de recursos. Por esse motivo, a performance superior do padrão assíncrono seria irrelevante, e inclusive, poderia criar gargalos de custo e operacionais.

5.2 Considerações futuras

No futuro, além do tempo de resolução de uma requisição de predição, os sistemas poderiam expor outras métricas a serem analisadas. A princípio, existia uma ideia de também coletar métricas de performance do sistema inteligente, como consumo de memória, CPU e rede. Pensando na restrição de que a arquitetura dos sistemas deveriam ser mantidas sem maus cheiros, respeitando o estilo arquitetural escolhido, uma ideia pensada seria de aproveitar a infraestrutura de mensageria providenciada pelo `NATS`. De tal forma que, a cada predição feita pelo sistema inteligente, uma mensagem poderia ser gerada num tópico próprio para receber métricas de performance do sistema (destacando que cada requisição já possui um id próprio associado a si mesmo), e ao final dos testes, tais mensagens poderiam ser coletadas pelo sistema de *benchmark* e propriamente armazenadas no repositório de métricas para uso posterior.

Um ponto a se destacar é que as decisões arquiteturais tomadas para a implementação do sistema preditor permitem que o sistema utilize diferentes modelos de predição e não só o modelo de simulação implementado para a realização dos experimentos desta monografia. Para tal, basta que o modelo seja provisionado no `MLflow` e o adaptador do repositório de modelos seja ajustado de acordo com a biblioteca de modelos de ML utilizada. Além disso, as decisões permitem que outros padrões de comunicação sejam implementados para servir um modelo de predição. Assim, além do aspecto de tipo de responsividade, é possível avaliar outras características que possam vir a ser interessantes para o pesquisador.

Apêndice A

Instruções Uso do *Benchmark* via Linha de Comando

Com o sistema preditor em execução (https://github.com/washington-ygor-tcc/intelligent-system), e com a devida configuração local do projeto de *benchmark* (https://github.com/washington-ygor-tcc/benchmark), é possível executar um teste utilizando a linha de comando à partir do uso de diferentes parâmetros:

- tipo de padrão de comunicação **API** ou **MSG** (`--type, -t, default = API`),
- número de requisições a serem realizadas (`--requests-number, -n`),
- tempo em segundos para disparar requisições continuamente alternativo ao número de requisições (`--runtime, -r`),
- tamanho do lote de requisições a serem disparadas em paralelo (`--batch-size, -b, default = 100`),
- intervalo de tempo em segundos a ser esperado entre cada um dos lotes de requisição (`--interval, -i, default = 0`),
- fator de complexidade de processamento (**cf**) do modelo de predição, que descreve uma complexidade de tempo de $\mathcal{O}(cf^3)$ e de espaço de $\mathcal{O}(cf^2)$ (`--complexity-factor, -cf, default = 3`),
- fator de sobrecarga de memória (**mo**) que incrementa o uso de memória pelo modelo de predição por um fator de $\mathcal{O}(mo)$ (`--memory-overhead, -mo, default = 1`),
- valor que define um diretório para salvar um arquivo `CSV` com as métricas coletadas pelo teste (`--csv`),
- *flag* que define se as métricas devem ser exibidas na saída padrão (`--table`),
- *flag* que define se informações estatísticas devem ser exibidas na saída padrão (`--stats`),
- *flag* que se o progresso dos lotes de requisição do teste deve ser exibido (`--batch-progress, -bp`), e
- *flag* que se o progresso total do teste deve ser exibido (`--total-progress, -tp`).

Um exemplo de execução de um teste com os parâmetros: API e MSG, com 1000 requisições separadas em lotes paralelizáveis de tamanho 100, com um modelo de predição que descreve um fator de complexidade 100, e exibindo dados estatísticos do teste. Poderia ser executado com o comando e teria saída descrita pela Figura A.1:

```
poetry run benchmark -t API -t MSG -n 1000 -b 100 -cf 100 --stats
```

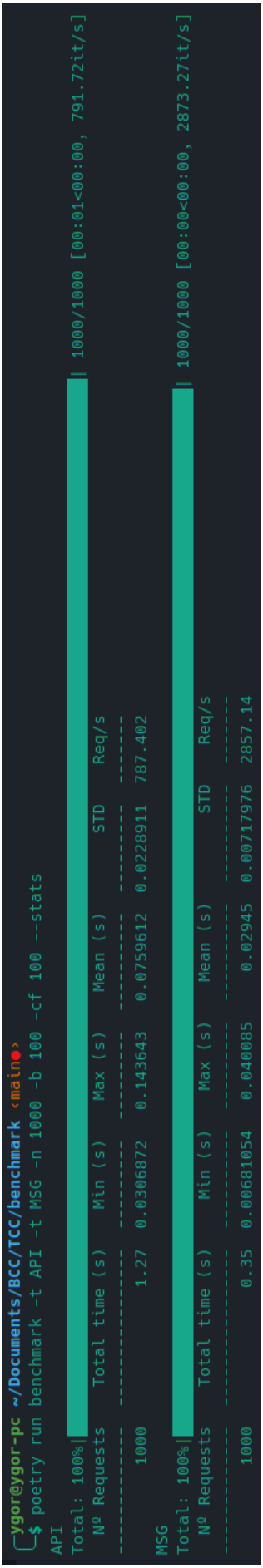

Figura A.1: *Saída do benchmark após execução de um teste via linha de comando.*

Bibliografia

Booch et al. (2005) Grady Booch, James Rumbaugh e Ivar Jacobson. **The Unified Modeling Language User Guide Second Edition**. Citado na pág. 10

Brown (2018) Simon Brown. The C4 Model for Software Architecture, 6 2018. URL https://www.infoq.com/articles/C4-architecture-model/. Citado na pág. 10

Bushmann et al. (1996) F Bushmann, R Meunier, H Rohnert e Soft Ware Architecture. **Pattern-Oriented Software Architecture**, volume 1. Citado na pág. 7

Cockburn (2005) Alistair Cockburn. Hexagonal Architecture, 2005. URL https://alistair.cockburn.us/hexagonal-architecture/. Citado na pág. 7

Gamma et al. (1996) E Gamma, R Helm, R Johnson e J Vlissides. Design Patterns: Elements of Reusable Software. **Addison-Wesley Professional Computing Series**. ISSN ISBN: 0-201-63361-2. Citado na pág. 5, 6

Garlan e Shaw (1993) David Garlan e Mary Shaw. An Introduction to Software Architecture. doi: 10.1142/9789812798039{_}0001. Citado na pág. 7

Hohpe e Woolf (2012) Gregor Hohpe e Bobby Woolf. **Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions (Google eBook)**. Citado na pág. 8, 10

Hulten (2019) Geoff Hulten. **Building Intelligent Systems**. Apress, Berkeley, CA. ISBN 978-1-4842-3933-9. doi: 10.1007/978-1-4842-3933-9. Citado na pág. 1, 2

Lakshmanan et al. (2020) Valliappa Lakshmanan, Sara Robinson e Michael Munn. **Machine Learning Design Patterns: Solutions to Common Challenges in Data Preparation, Model Building, and MLOps** . Citado na pág. 1

Martin (2017) Robert C Martin. **Clean Architecture: A Craftsman's Guide to Software Structure and Design**. Citado na pág. 1, 3, 4, 5, 7

Martin Fowler (2004) Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern, 1 2004. URL https://martinfowler.com/articles/injection.html. Citado na pág. 6

Martin Fowler (2015) Martin Fowler. Presentation Domain Data Layering, 8 2015. URL https://martinfowler.com/bliki/PresentationDomainDataLayering.html. Citado na pág. 7

Martin Fowler (2019) Martin Fowler. Is High Quality Software Worth the Cost?, 2019. URL https://martinfowler.com/articles/is-quality-worth-cost.html. Citado na pág. 5

Nelson (1981) Bruce Jay Nelson. **Remote Procedure Call**. Tese de Doutorado, XEROX PARC. Citado na pág. 10

Richards e Ford (2020) Mark Richards e Neal Ford. **Fundamentals of Software Architecture: An Engineering Approach**. O'Reilly. Citado na pág. 3

Russel e Norvig (2012) Stuart Russel e Peter Norvig. **Artificial intelligence—a modern approach 3rd Edition**. doi: 10.1017/S0269888900007724. Citado na pág. 2

Sato et al. (2019) Danilo Sato, Arif Wider e Christoph Windheuser. Continuous Delivery for Machine Learning. **Martin Fowler**. Citado na pág. 1