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

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 em análises de crédito realizadas por bancos para conceder empréstimos.

Dentro do ciclo de vida de um sistema inteligente, podemos dividir uma aplicação de aprendizado de máquina em três eixos de mudança (Sato **et al.** , 2019): dados, modelo, e código. Os dados e modelos concedem uma característica particular aos sistemas inteligentes já que os modelos são retreinados conforme há a 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, proporcionando um sistema 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 tal 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, e a partir disso escolher o protocolo mais adequado para a comunicação.

Dadas as peculiaridades de um sistema inteligente e as necessidades 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 cenários e *trade-offs* entre os padrões adotados**. Consequentemente, este projeto visa encontrar os principais prós e contras de cada abordagem, além de melhor discutir a aplicabilidade de cada um dos padrões.

Esta monografia descreve a proposta de desenvolvimento de alternativa para servir um sistema inteligente. No Capítulo 2, serão discutidos conceitos relevantes para o que está sendo discutido. No **??** é detalhado a proposta deste trabalho. No **??**, é apresentado um plano de projeto para executar o trabalho.

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 impõem 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 rapidamente evolua e mude o seu comportamento para atender uma dada necessidade. 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 estruturação 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 as 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, tal decisão 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 requerido 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 que os componentes fiquem 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 à partir de partes que podem ser alternadas, é necessário que tais partes respeitem um contrato que permite que elas sejam substituíveis umas pelas outras (Martin , 2017). Consolidando 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, enquanto que, a altura e a largura de um `Quadrado` são sempre iguais.

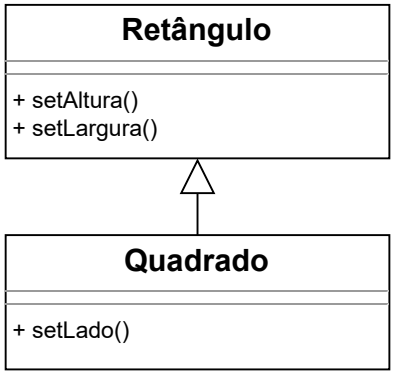

Figura 2.1: *Exemplo de violação do princípio da substituição de Liskov.*

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). Tal 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. Enquanto que, os elementos abstratos são **estáveis** em relações a mudanças. Desta forma, tal princípio possibilita a divisão do sistema em dois componentes em relação aos seus níveis de **estabilidade**: um abstrato e outro concreto. Contribuindo para que o código seja mais suscetível a mudanças, já que os detalhes (concretos) passam a mudar ser afetar os contratos (abstratos).

2.3.4 Qualidade de Software

Além do próprio valor entregue pela solução do software em si, uma outra ótica a se avaliar o valor de um software se da pela sua estrutura (Martin , 2017). Muitas vezes, por não ser algo aparente ao usuário final, ou pelo custo de tempo e esforço, 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 sim à todo ciclo de vida de um sistema.

Um maior tempo de vida de sistema sempre irá beneficiar o 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 (2.3), os padrões são uma importante ferramenta para os desenvolvedores, contribuindo para reduzir o gasto de tempo para definir uma forma de 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).

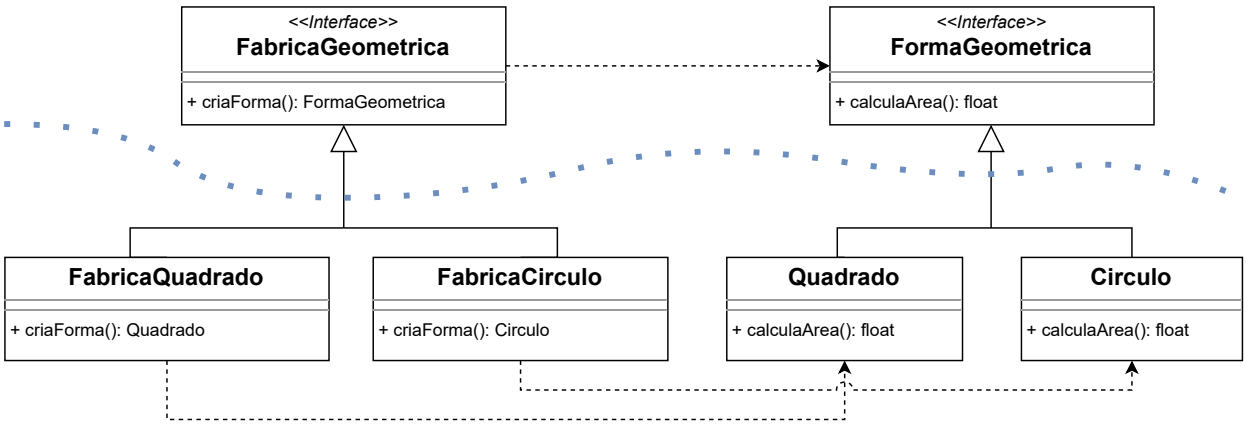

Figura 2.2: *Aplicação Padrão Método Fábrica*

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 com um estilo de pintura 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`.

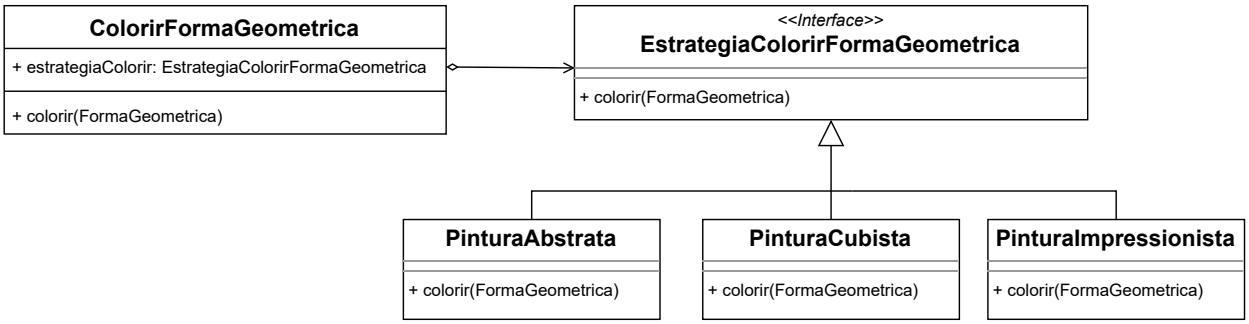

Figura 2.3: *Aplicação Padrão Estratégia*

2.4.3 Padrão Injeção de Dependência

O padrão INJEÇÃO DE DEPENDÊNCIA é um padrão aplicado principalmente em linguagens orientada 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, como a do exemplo abaixo em que é instanciado 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

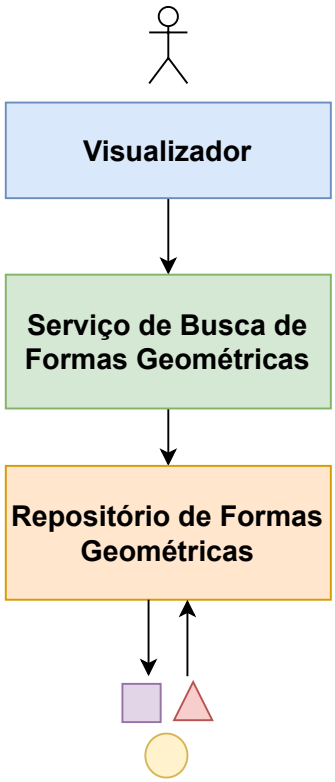

Figura 2.4: *Exemplo de aplicação do estilo em camadas*

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 orientada a objetos uma porta será nada mais que 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 (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).

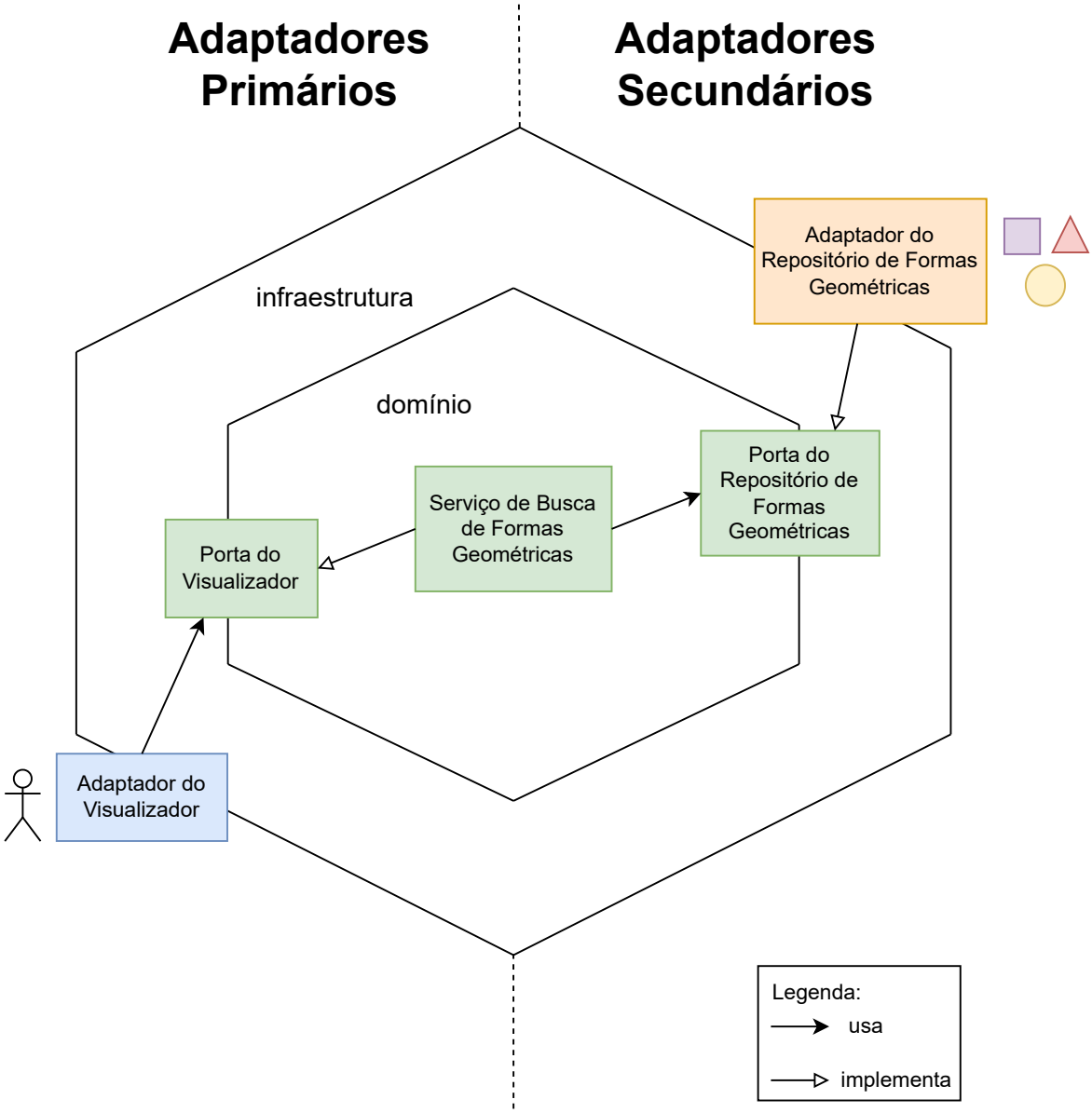

Figura 2.5: *Exemplo de aplicação do estilo hexagonal*

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 *publisher* e o *subscriber* 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 *subscriber*. 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.

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 que 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 dos seus componentes que o compõem.

4. **Código**:

   Representa um componente individualmente 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 de ***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 em relação ao padrão de comunicação adotado pelo cliente. Ao final de uma requisição de predição, o sistema coleta e armazena métricas relacionadas a 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 que é um **Preditor** capaz de servir modelos de aprendizado de máquina provisionados numa plataforma de gerenciamento de ciclo de vida de modelos de predição. 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 (Seção 2.7) mediante a adoção de alguns diagramas do Modelo C4 (Subseção 2.7.2), e a adição de detalhes pertinentes as características arquiteturais (Subseção 2.3.1) e decisões de arquitetura (Subseção 2.3.2) 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 diretamente 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 de porta de entrada a ser utilizado, o *Benchmark* irá consumir a porta de entrada adequada exposta pelo sistema Preditor. 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.

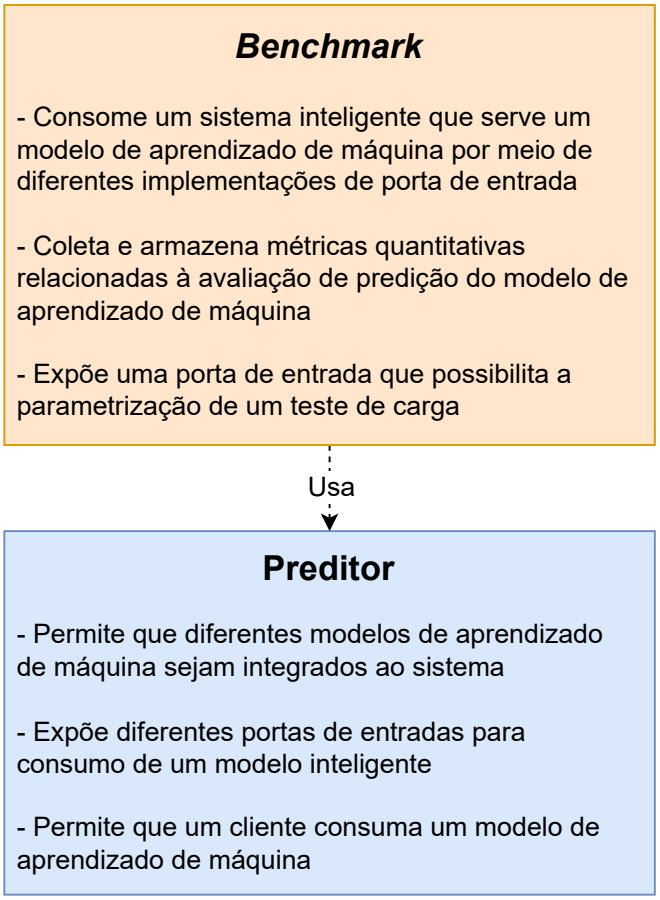

Figura 3.1: ***Diagrama de Contexto****. Representação dos requisitos funcionais e as relações dos atores da arquitetura.*

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 porta de entrada, tanto para consumir um modelo de predição pelo *Benchmark*, como para servir um modelo de predição pelo Preditor. Tal semelhança foi fundamental para mapear algumas características arquiteturais 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 de Arquitetura

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 (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 um dos sistemas. Assim, os sistemas ganham bastante flexibilidade e desacoplamento em relação aos componentes relacionados a comunicação.

2. Adoção da plataforma do MLFlow usada pelo Preditor para gerenciar o ciclo de vida de modelos de aprendizado de máquina. Um dos principais motivos para tal decisão se da pelo fato da plataforma ser agnóstica a qualquer biblioteca de aprendizado de máquina, centralizando detalhes relacionados aos modelos de predição. 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 aumenta o nível de extensibilidade e desacoplamento, 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 não "inteligente", que podem ser usados para simular modelos reais.

3.4 Contêiner

No diagrama de **Contêiner** da Arquitetura, presente na Figura 3.2, é apresentado a aplicação como um todo, destacando onde se encaixa o sistema preditor, o sistema de *benchmark*, e a infraestrutura necessária para o funcionamento de cada um dos sistemas.

O sistema preditor é capaz de servir modelos de aprendizado de máquina por meio 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), o Padrão Estratégia (Subseção 2.4.2) e o estilo arquitetural adotado permitem a extensão da aplicação para qualquer outro padrão de comunicação sem complicações.

O funcionamento do sistema de *benchmark* se baseia exclusivamente na entrada do seu cliente, sendo apenas uma interface para realizar um teste de carga. Ao receber uma requisição de teste, uma sequência de passos é seguida, 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 (a princípio é coletado apenas o tempo de resposta de cada requisição) podem ser salvas em um repositório de dados a depender do interesse do cliente, no caso, são utilizados planilhas CSV para armazenar tais métricas.

Um outro ponto a se destacar, são as dependências associadas a plataforma 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 entidades da própria plataforma como parâmetros dos modelos (providenciada pelo banco relacional MySQL).

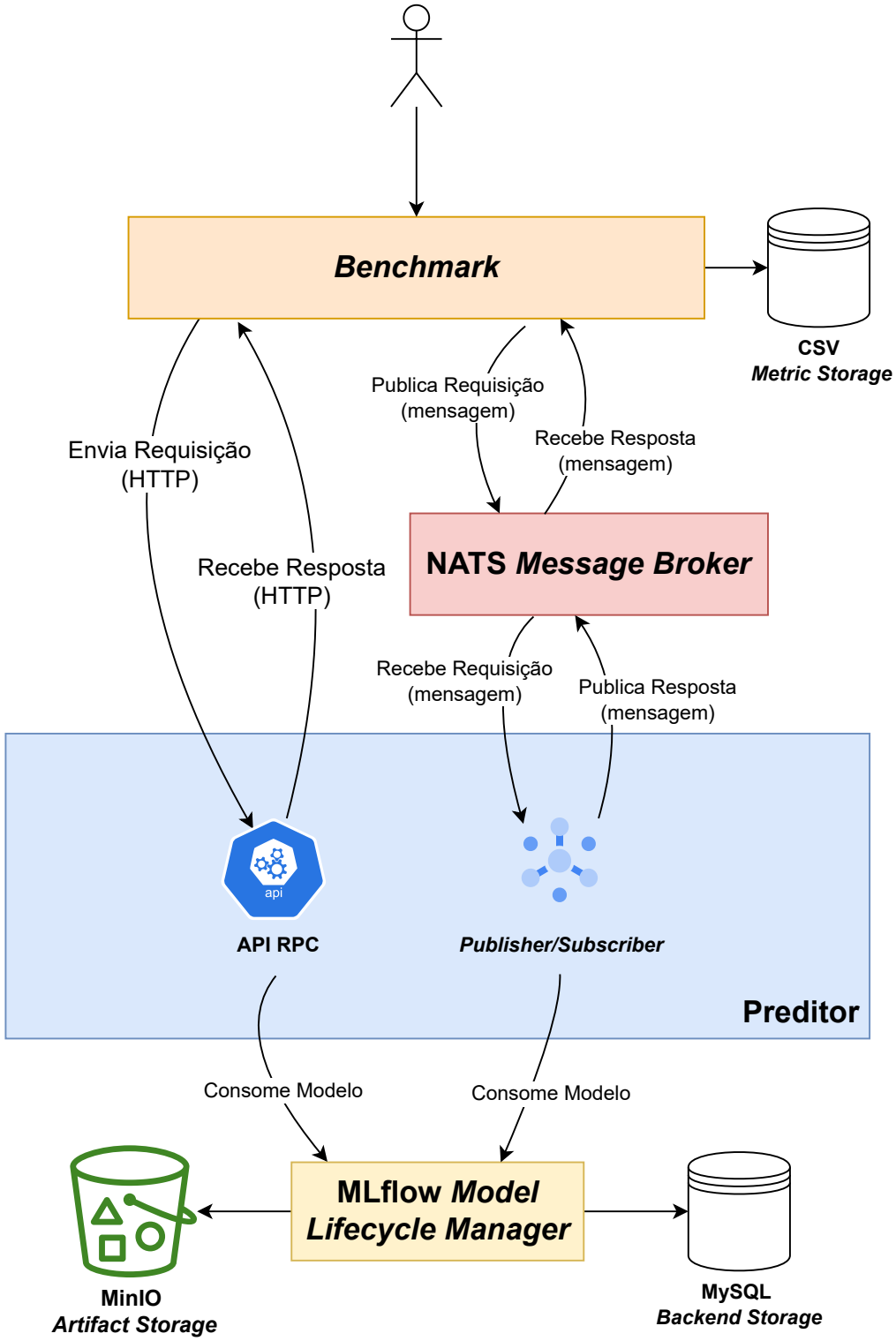

Figura 3.2: ***Diagrama de Contêiner****. Representação dos sistemas e das suas dependências de infraestrutura.*

3.5 Componente do Preditor

Na Figura 3.3 é exibido a camada de abstração do **Componente** do sistema inteligente, que basicamente se divide em:

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

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

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

Tal comportamento é encapsulado em um único fluxo da aplicação 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 o arcabouço `FastAPI`, enquanto o adaptador da porta de entrada assíncrona se baseia na implementação de um consumidor e um produtor de mensagens que utiliza uma biblioteca interface para o sistema de mensageria `NATS`. Em relação ao repositório de modelos, o adaptador foi implementado utilizando a biblioteca interface do `MLFlow`.

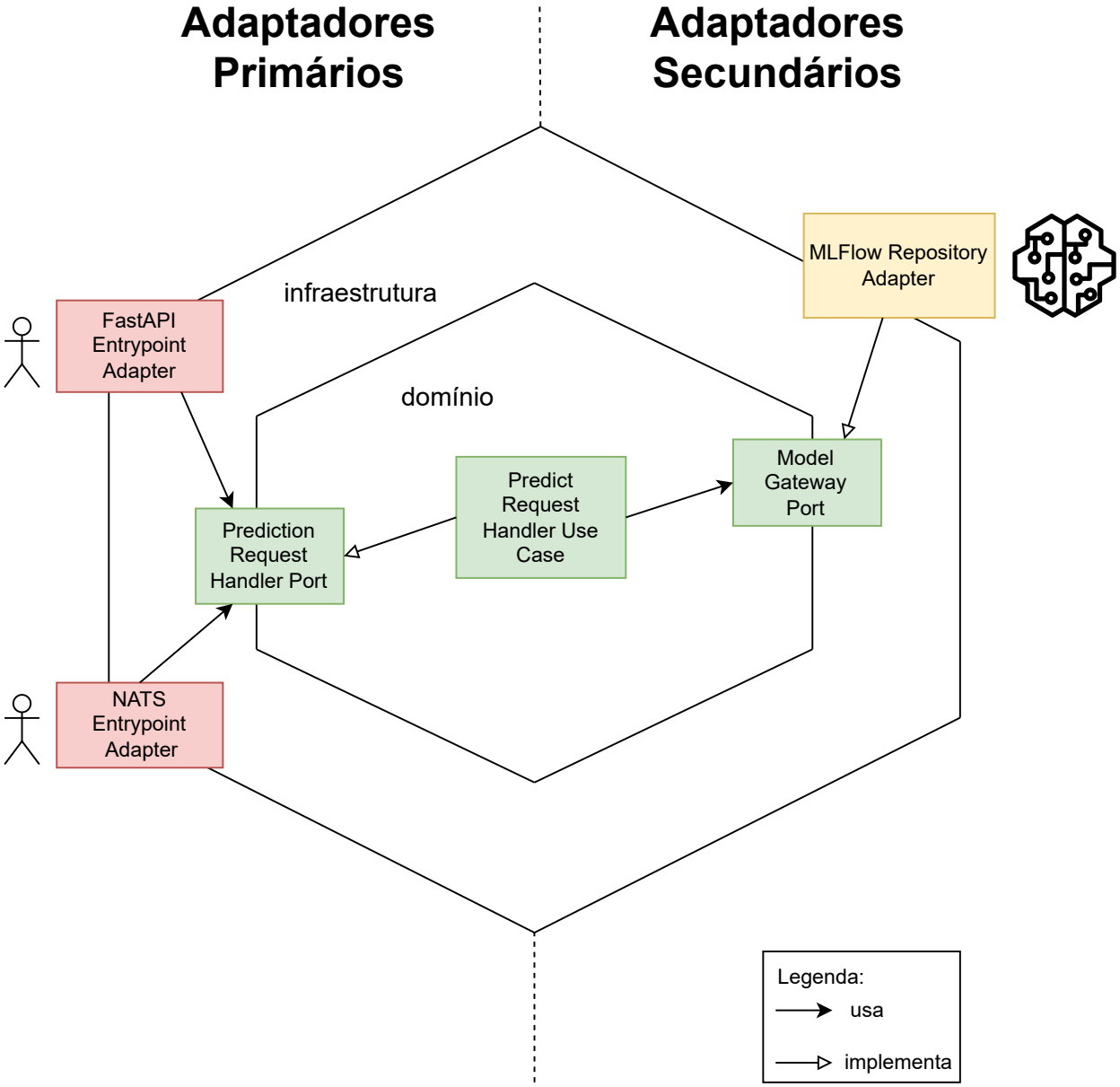

Figura 3.3: ***Diagrama de Componente****. Representação do componente núcleo do sistema inteligente.*

3.6 Componente do *Benchmark*

Na Figura 3.4 é exibido a camada de abstração do *Componente* do sistema de *benchmark*, que basicamente 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 no 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 em sua essência segue passos fixos para se completar. 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 e o 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 de *benchmark* pode controlar diversos parâmetros, tais como:

- tipo de padrão de comunicação (API RPC ou mensageria),
- fator que controla a complexidade de execução do modelo de predição,
- fator que controla a quantidade de memória usada durante a predição,
- número de requisições a serem realizadas,
- tempo em segundos para disparar requisições continuamente (usado como alternativa ao parâmetro de número de requisições),
- tamanho de lote de requisições a serem disparadas paralelamente, e
- intervalo de tempo em segundos entre cada lote de requisição.

Um ponto a se destacar em relação ao adaptador, é que ele atua como uma `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 variância de uma decisão arquitetural já que viola regras do estilo arquitetural adotado. Novamente, a justificativa para isso é explicada pela natureza de um sistema de *Benchmark*. Ainda em relação ao adaptador de entrada, foi implementado uma interface de consumo da biblioteca que é mais amigável aos usuários finais do sistema 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 fluxos implementados, o `Run Benchmark Use Case` e o `Save Benchmark Use Case`. O primeiro abstrai uma única requisição de predição 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.

Em relação ao segundo caso de uso, a sua responsabilidade é de salvar os resultados de todas as requisições feitas utilizando a porta de repositório de métricas.

A geração de métricas de tempo e identificadores únicos foram separadas em dois adaptadores. Tal decisão foi motivada porque esse ponto é um detalhe que pode ser desacoplado e testado a parte do domínio. No entanto, 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. Deixando aberto um caminho para possíveis extensões e evoluções no futuro.

Da mesma forma que o sistema inteligente expõe duas portas de entrada (via API RPC e mensageria) o sistema de *benchmark* implementa dois clientes de serviço distintos que são capazes de consumir cada uma das portas de entrada. Cada um deles é uma implementação da porta de requisições de predições, e 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`.

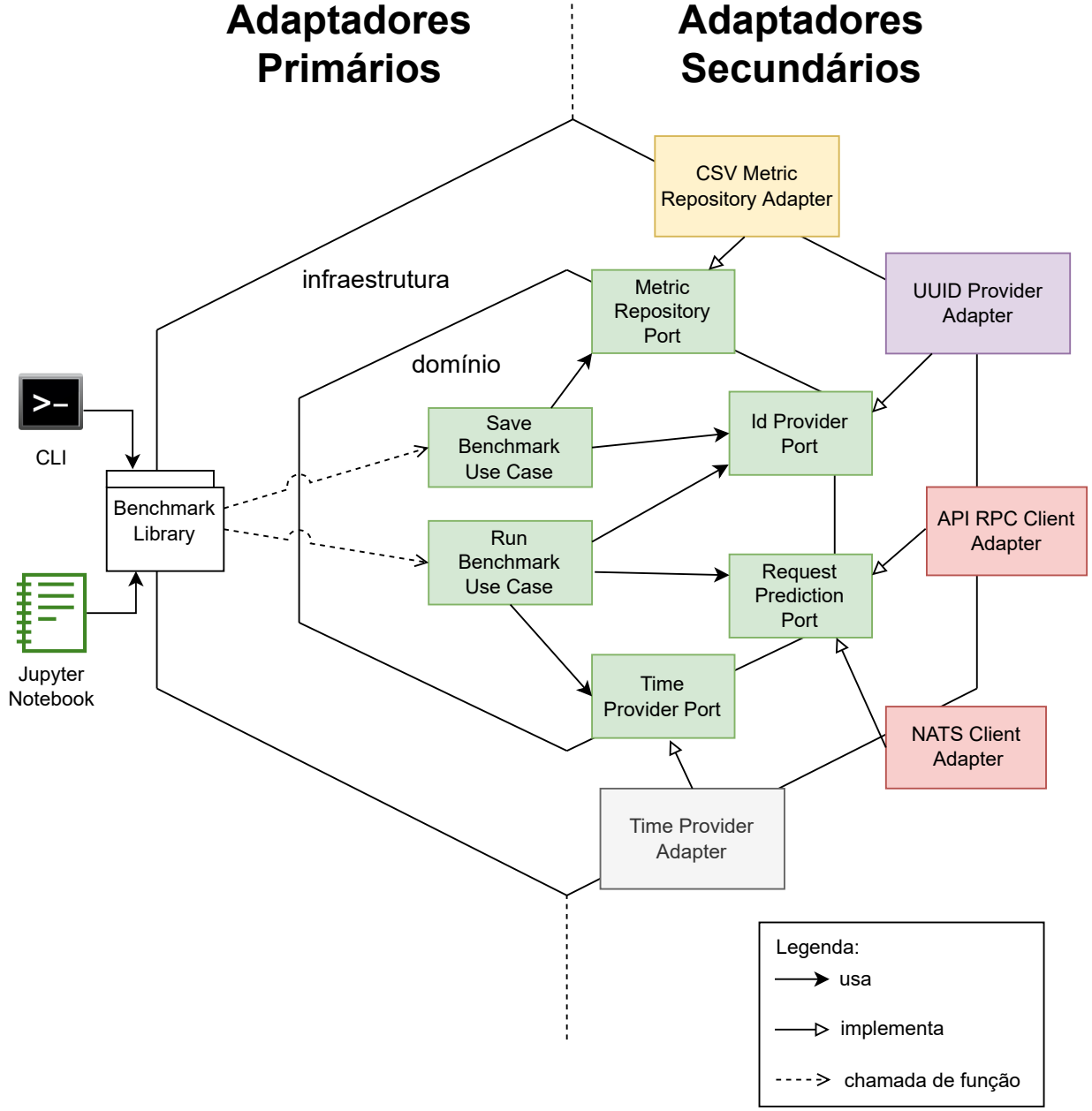

Figura 3.4: ***Diagrama de Componente****. Representação do componente núcleo do sistema de benchmark.*

Capítulo 4

Experimentos

Utilizando o *benchmark* implementado, o 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 com um questionamento, apresentando diferentes métricas quantitativas e observações encontradas durante a experimentação que contribuem para análise dos resultados em capítulos posteriores da monografia.

4.1 Ambiente de Testes

A coleção de métricas foram feitas 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.

Um ponto a se destacar é que o *benchmark* e o sistema preditor foram executados concorrentemente no mesmo ambiente e dentro de contêineres `Docker`.

4.2 Modelo de Predição Adotado

A princípio, havia uma ideia de utilizar modelos de predição reais, inclusive, houve tentativas 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 replicar diferentes níveis de complexidade de execução e espaço ao realizar uma predição, se mostrou como um grande impeditivo para tal abordagem. Desta 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 tempo de execução e uso de memória. A partir desses valores, são criados duas matrizes quadradas com valores aleatórios, uma com dimensão igual ao valor CF e outra com dimensão igual ao valor MO. A primeira matriz

é utilizada para resolver a solução de um sistema linear $Ax = b$, com b sendo um vetor gerado aleatoriamente com dimensão igual ao valor CF. Desta forma, o modelo descreve uma complexidade de execução dada por $\mathcal{O}(CF^3)$ e de espaço utilizada por $\mathcal{O}(CF^2 + MO^2)$. A implementação do modelo de simulação pode ser encontrado no GitHub.

4.3 Primeiro Experimento

O experimento busca responder a seguinte questão: **Como requisições de predição realizadas paralelamente afetam a performance do sistema preditor**? Para responder tal questão, as seguintes métricas quantitativas podem contribuir para sua solução:

- tempo de requisição,
- média do 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 dos Testes do *Benchmark*

O parâmetro de **tamanho de lote de requisições** executadas paralelamente é o parâmetro chave para o 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 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 execução igual a 100.

4.3.2 Dados Obtidos

Os dados coletados buscam refletir as métricas associadas ao 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 no GitHub.

4.3.3 Observações

Além das métricas quantitativas observadas, alguns comportamentos discrepantes foram observados durante a experimentação, podendo agregar em 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 elevadas cargas de requisições ao usar a API RPC provocou 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 inserir um intervalo de tempo entre cada teste.

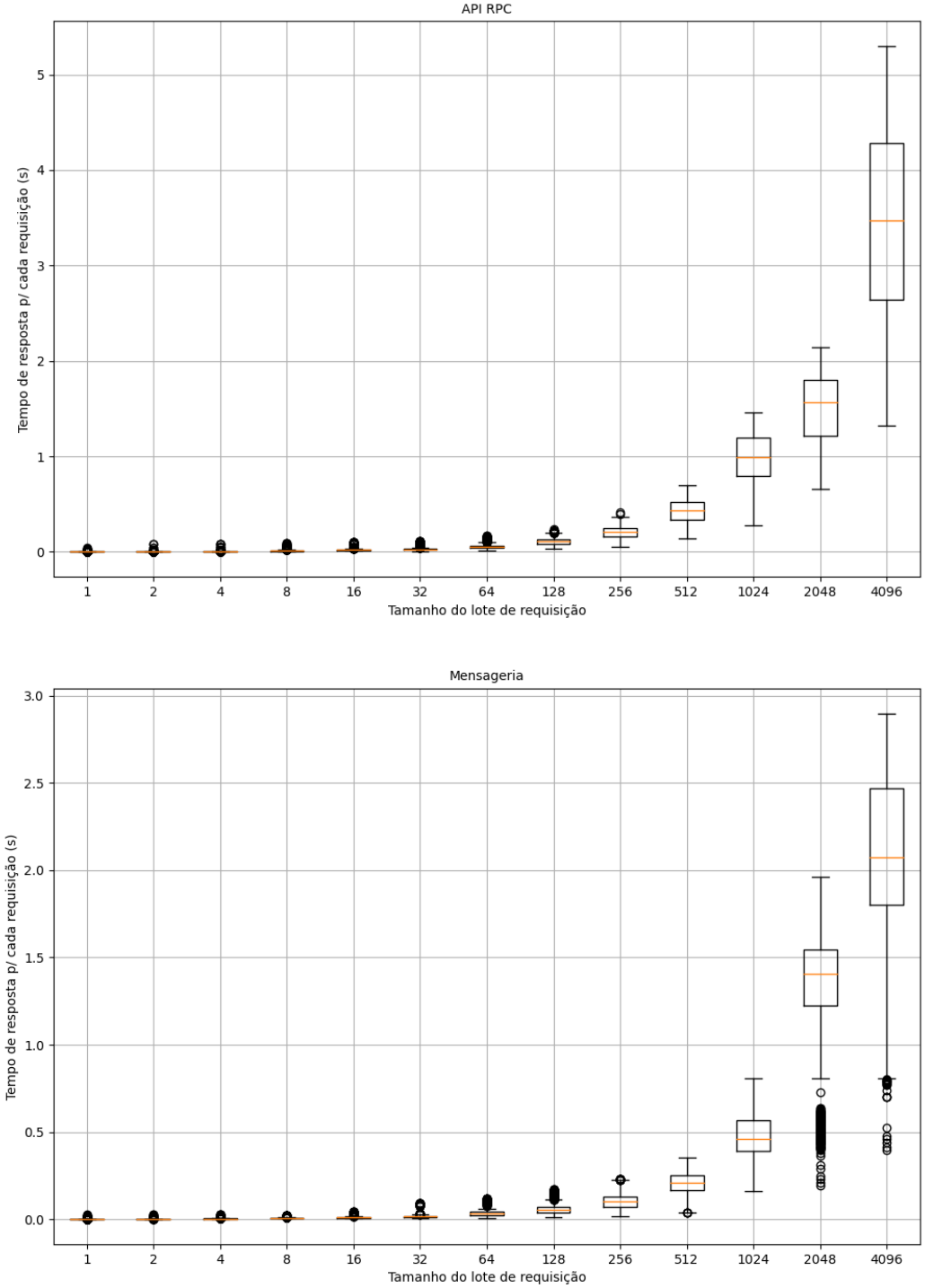

Figura 4.1: *Boxplots que definem a distribuição de 4096 requisições distribuídas por diferentes tamanhos de lote.*

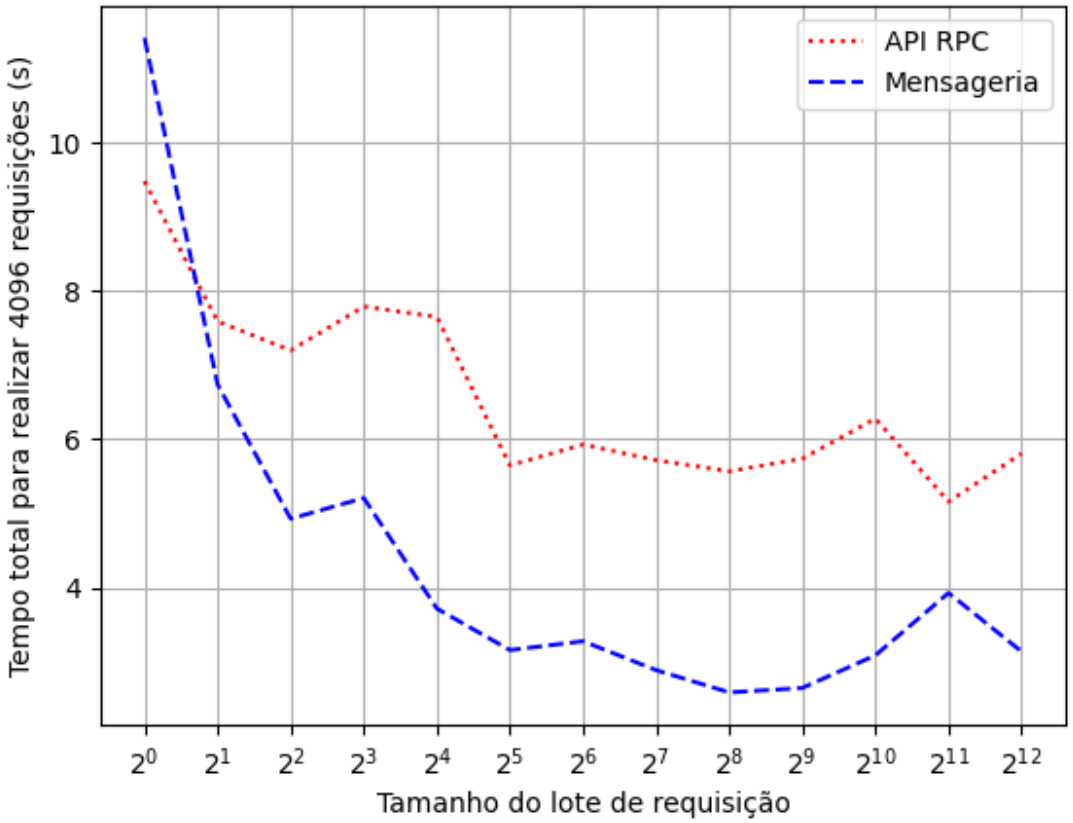

Figura 4.2: *Tempo total utilizado para executar cada conjunto de 4096 requisições distribuídas por diferentes tamanhos de lote.*

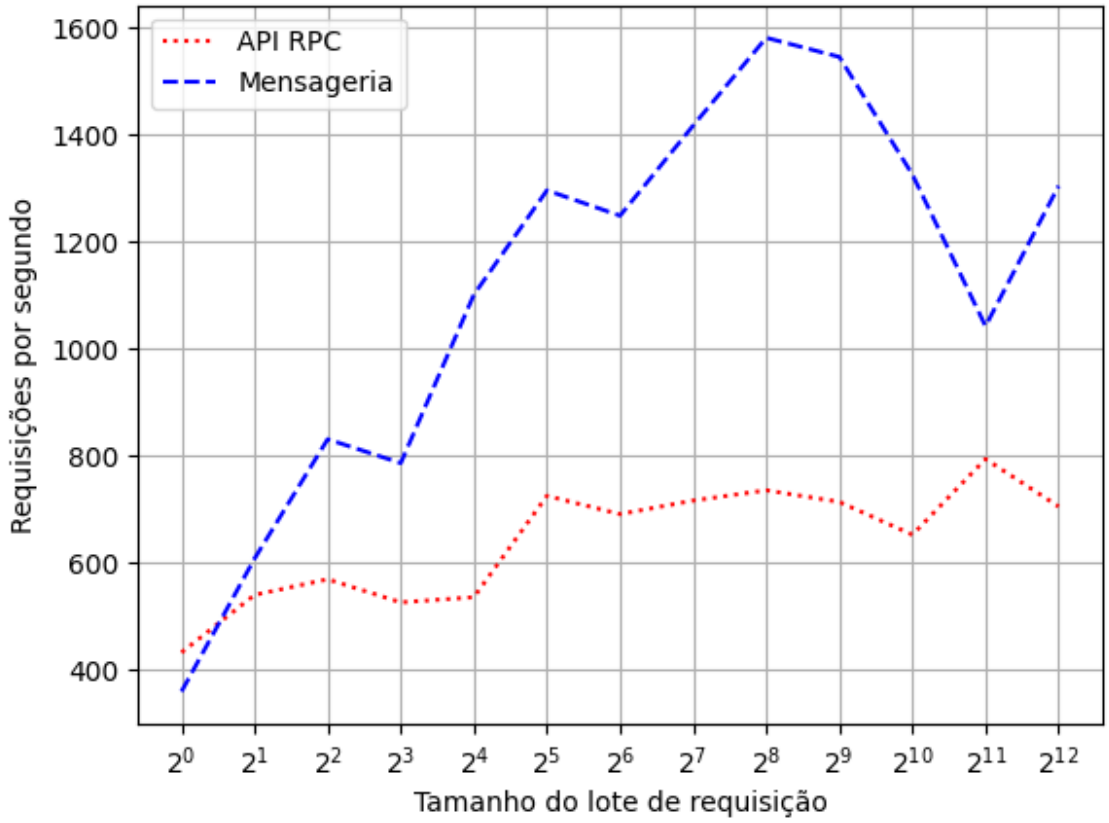

Figura 4.3: *Requisições feitas por segundo para diferentes tamanhos de lote.*

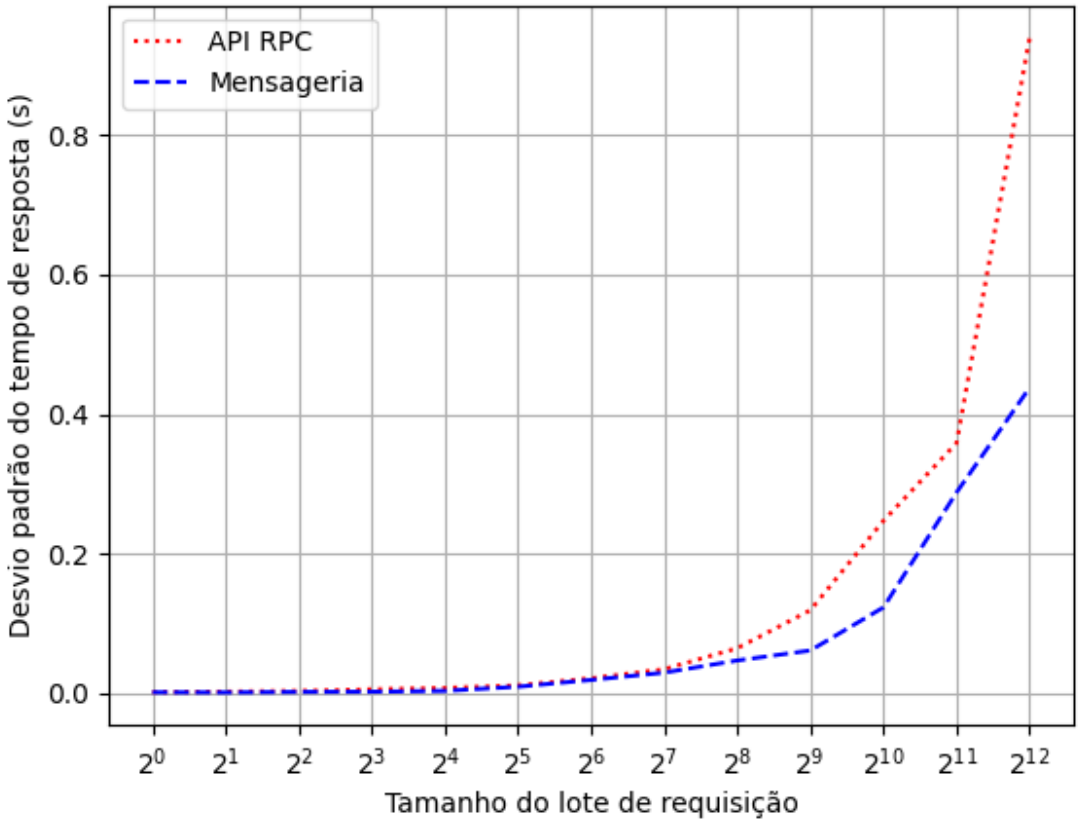

Figura 4.4: *Desvio padrão do tempo de execução de 4096 requisições distribuídas por diferentes tamanhos de lote.*

Capítulo 5

Discussão

5.1 Primeiro Experimento

5.1.1 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 paralelamente. Além disso, é possível tirar três conclusões importantes relativa aos formatos dos *boxplots*:

- os limites inferiores, superiores, e a amplitude interquartil do *boxplot* crescem conforme há maior nível de paralelismo,
- a distribuição do tempo de resposta está bem distribuída entre todos os quartis, e
- a mediana (linha laranja do *boxplot*) se encontra próxima ao centro do intervalo interquartil.

A partir disso, podemos concluir que conforme há mais requisições sendo feitas paralelamente, o preditor apresenta uma maior dispersão relativa ao tempo de resposta, mantendo uma distribuição simétrica. A Figura 4.4 deixa perceptível o quanto a dispersão da distribuição aumenta com maior paralelismo, principalmente quando o padrão de comunicação utilizado é a API RPC.

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 em paralelo (tamanho de lote ≥ 2). A mensageria foi capaz de atingir um pico de aproximadamente 1600 requisições atendidas por segundo quando as requisições foram separadas em lotes de tamanho 256, sendo uma diferença próxima do dobro do pico de 800 requisições atendidas por segundo pela API RPC para lotes de tamanho 2048. Apesar da diferença entre os padrões de comunicação, um ponto importante a se destacar é que quando o preditor teve que responder requisições sequencialmente (tamanho de lote $= 1$), a API RPC perfomou melhor que a mensageria já que ela conseguiu responder as 4096 requisições num menor tempo e com uma maior taxa de requisições antedidas por segundo.

5.1.2 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. Seja pelo o que foi observado nos gráficos, de conseguir atender mais rapidamente as requisições por conseguir maiores taxas de requisições respondidas por segundo, ou de apresentar menor degradação da performance por apresentar menor dispersão para maiores quantidades de requisições sendo feitas paralelamente. Como também, pelas observações apontadas na Subseção 4.3.3, que podem indicar que a API RPC exige mais recursos – computacionais e rede – do que a mensageria se levarmos em consideração apenas o ambiente de execução do sistema inteligente.

Um ponto que merece atenção, é de que a performance da API RPC para atender requisições uma a uma foi superior. 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.

Capítulo 6

Conclusões e considerações

6.1 Conclusão

6.2 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.

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 (**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 $\mathcal{O}(mo^2)$ (`--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
```

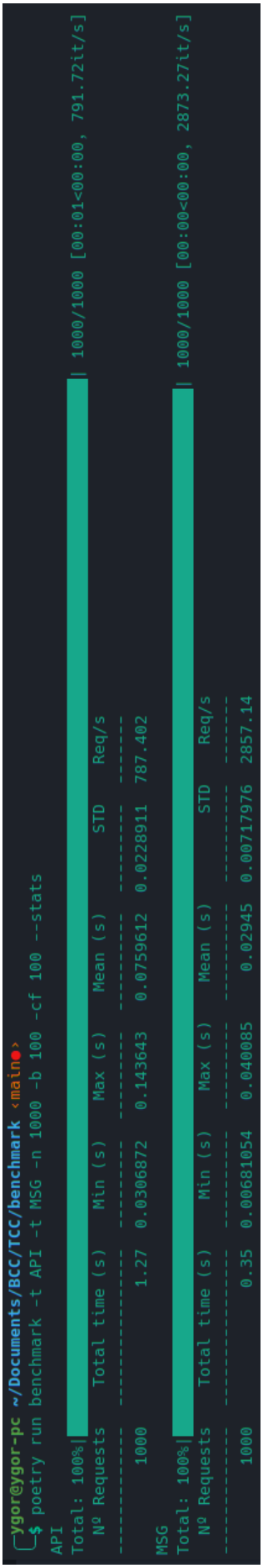

Figura A.1: *Saída do benchmark após execução de um teste via linha de comando.*

Considerações pessoais

A.1 Washington

A.2 Ygor

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