Explorando Estilos Arquiteturais 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, Abril 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 variadas finalidades: uma simples tradução feita no Google Tradutor, recomendações de playlists com músicas adequadas ao perfil de qualquer pessoa com o Spotify, até mesmo revolucionando a pesquisa em biocomputação com o AlphaFold (Ewen Callaway , 2022). Tais sistemas são semelhantes à sistemas da informação tradicionais, principalmente no que diz respeito a ter um objetivo e entregar valor aos seus usuários. No entanto, a inteligência oriunda de modelos de aprendizado de máquina caracterizam unicamente tais sistemas.

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. 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 eixo de código deve se estruturar para comportar as mudanças promovidas ao longo do tempo pelos outros dois eixos.

Assim como sistemas de software tradicionais, o processo de design de sistemas inteligentes não deve garantir apenas que o sistema funcione com um propósito claro, mas também, que seja um sistema robusto, barato de manter, e com um longo tempo de vida. 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. A estrutura dessa *forma* se traduz em como os componentes do sistema são divididos, como os componentes estão dispostos e, como os componentes 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 (Martin , 2017).

Dentro da arquitetura de um sistema inteligente, surge um importante questionamento relacionado à escalabilidade (Lakshmanan **et al.** , 2020): *Como servir um modelo de tal forma que ele suporte milhões de requisições de predições em um curto período de tempo?* Para abordar tal questionamento, é fundamental entender a comunicação entre quem serve e quem consome o modelo. Comumente, a principal característica de uma comunicação a se determinar é a sua natureza temporal – 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 arquiteturais 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.

Este documento 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 não inteligentes. 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. 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;
- Como orquestrar o ciclo de vida da inteligência.

2.2 Aprendizado de Máquina

A área da inteligência artificial, ou IA, assim como na filosofia e psicologia, 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 é uma área de IA que se preocupa em construir algoritmos através de dados, os quais podem vir da natureza, criados pelos humanos ou 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 através 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. 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, ou seja, é o que o torna maleável o suficiente para deixar o maior número de opções disponíveis pelo maior tempo possível (Martin , 2017).

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 aos usuários finais ou, pelo custo de tempo e esforço, a arquitetura acaba sendo deixada de lado no processo de desenvolvimento do software. Desta forma, é importante notar que o design de arquitetura não se refere apenas ao processo inicial de desenvolvimento, mas sim à todo ciclo de vida de um sistema.

Via de regra, um maior tempo de vida de sistema sempre irá beneficiar o uso de boas práticas de desenvolvimento, independentemente de todo gasto com 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 compressão do código pelos programadores, melhorando a manutenibilidade do sistema, facilitando o entendimento da divisão entre requisitos importantes e requisitos que são detalhes ao sistema, etc.

2.4 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). Basicamente, 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 de software.

2.4.1 Princípio da Responsabilidade Única

O princípio pode ser condensado na seguinte frase (Martin , 2017):

"Um módulo deve ser responsável por um, e apenas, um ator"

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).

2.4.2 Princípio Aberto-Fechado

O princípio estabelece que o comportamento de um artefato de software deve buscar ser extensível, de tal forma que não haja modificações neste artefato (Martin , 2017). Apesar da simples declaração, tal princípio tem um grande valor na arquitetura de sistemas, já que a prática de tal princípio faz com que os componentes sejam separados em como, porque, e quando eles mudam. Desta forma, os componentes ficam organizados numa hierarquia de dependências que protege componentes de alto-nível de mudanças provocadas por componentes de baixo-nível.

2.4.3 Princípio da Substituição de Liskov

O princípio estabelece diretrizes para o uso adequado de herança a partir da definição da relação de subtipos. Um bom exemplo para elucidar o princípio, é o exemplo do problema da relação quadrado/retângulo que viola o mesmo (Martin , 2017). Neste exemplo, como observado na Figura 2.1,

a classe *Square* não é um subtipo adequado da classe *Rectangle* pois a altura e a largura de um *Rectangle* são valores independentes entre si, em contrapartida, a altura e a largura de um *Square* deve mudar em conjunto.

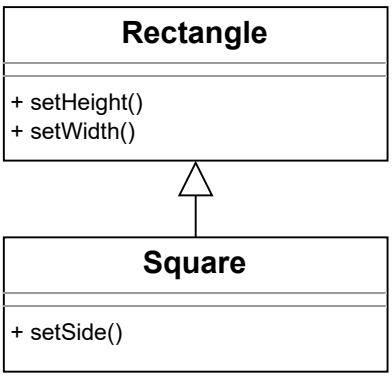

Figura 2.1: *Exemplo de violação do princípio da substituição de Liskov.*

Quando o princípio é estendido a nível arquitetural, a violação da substituibilidade pode ocasionar a poluição da arquitetura com mecanismos extras para tratar cada *mau cheiro* de código.

2.4.4 Princípio da Segregação de Interface

O princípio estabelece que nenhum código deve ser forçado a depender em métodos que ele não usa (Martin , 2017). Em linguagens estaticamente tipadas, como o Java, essa preposição faz bastante sentido se pensarmos na implementação de interfaces. Tal fato, pode conduzir para o pensamento equivocado que o princípio resolve um problema exclusivo a um paradigma de linguagem específica. No entanto, o principio é intimamente ligado a um problema de arquitetura, já que a dependência em coisas que não precisamos pode causar problemas não esperados. Um exemplo de um problema arquitetural relacionado a isso, é o caso de dependências transitivas em que indiretamente um sistema pode depender coisas que não deveria por depender de algo que depende dessas coisas.

2.4.5 Princípio da Inversão de Dependência

O princípio estabelece que os sistemas mais flexíveis são aqueles cujas dependências do código fonte se referem apenas a abstrações, e não concretizações (Martin , 2017). Apesar de parecer uma declaração radical ao indicar que deve-se apenas depender de abstrações, os elementos concretos se referem apenas aos elementos *voláteis* do sistema, ou seja, aqueles em que há constante desenvolvimento e passam por mudanças frequentes. Tal princípio tem o seu valor, principalmente pelo fato de que abstrações são estáveis em relações a mudanças, o que possibilita dividir o sistema em dois componentes: um abstrato e outro concreto.

2.5 Padrões de Projeto

Padrões de projeto é uma maneira barata, rápida, e eficiente de resolver problemas comuns em programação (Beck , 2007), características as quais diferenciam os padrões aos princípios de projeto (2.4). Cada padrão engloba um problema em específico, com a discussão de fatores que afetam o problema e um conselho de como resolver o problema de forma rápida para criar uma solução satisfatória. No design de arquitetura de software (2.3), os padrões são uma importante ferramenta para os desenvolvedores, contribuindo para reduzir o gasto de tempo e energia para definir uma forma de resolver um problema. No paradigma de programação orientada a objetos, um padrão de projeto pode ser classificado a partir do seu propósito (Gamma **et al.** , 1996):

1. Padrão de criação: Abstraí o processo de instanciação de objetos, tornando o sistema independente de como os objetos são criados, compostos, e representados.

2. Padrão estrutural: Explica como classes e objetos são compostos para formar estruturas maiores, de tal forma que sejam mantidos a flexibilidade e a eficiência da mesma.

3. Padrão comportamental: Explica como organizar as responsabilidades e a comunicação entre objetos.

2.5.1 Padrão Método Fábrica

O padrão *Método Fábrica* é um padrão de criação, comumente utilizado no paradigma de orientação a objetos. Uma interface é definida para criar um objeto, no entanto, as subclasses decidem qual classe instanciar (Gamma **et al.** , 1996). O padrão, via de regra, segue os princípios de segregação de interface (2.4.4) e o de inversão de dependência (2.4.5), o que torna o padrão útil para definir uma fronteira entre classes abstratas e concretas. Desta forma, naturalmente as dependências do código fonte são invertidas contra o fluxo de controle do código.

Na Figura 2.2 temos um exemplo de aplicação do padrão, onde é possível notar que a linha azul define uma fronteira entre o que é abstrato e o que é concreto. Um outro ponto importante a se notar, é que a classe *Main* depende apenas de abstrações, o que facilita a extensão da aplicação para qualquer outro tipo de *Entrypoint*, desde que haja uma *Factory* adequada e a injeção da dependência dessa nova *Factory* dentro da classe *Main*.

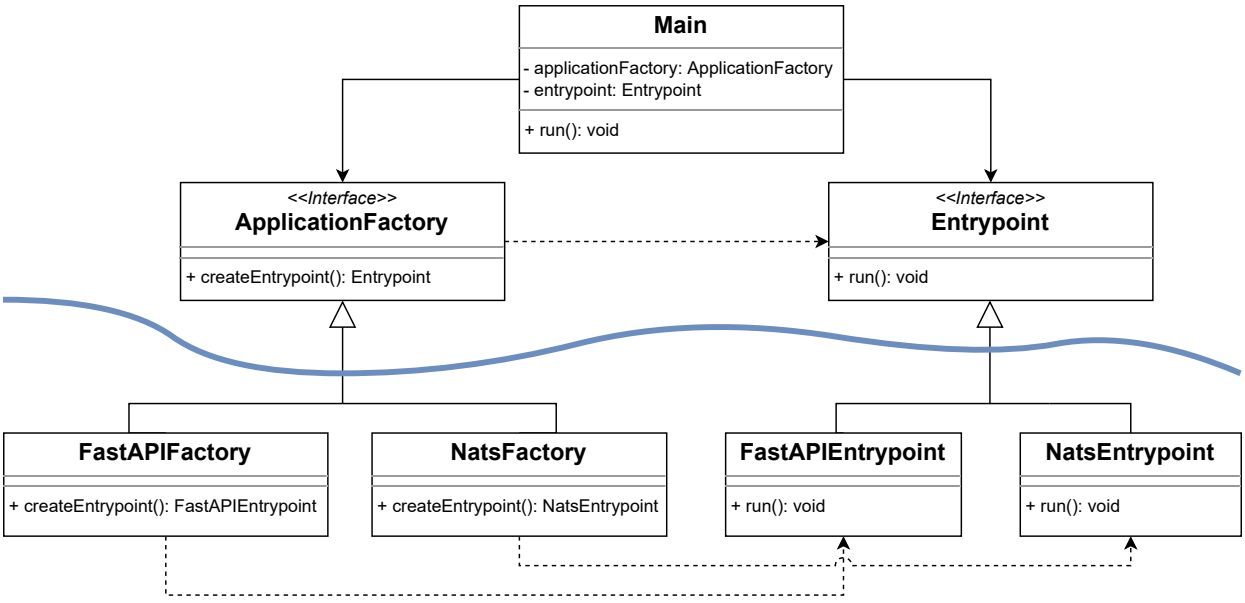

Figura 2.2: *Aplicação Padrão Método Fábrica*

2.5.2 Padrão *Publisher-Subscriber*

O padrão *Publisher-Subscriber* 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 *publisher* notifica um número qualquer de *subscribers* sobre a mudança do seu estado. Esse padrão é baseado no padrão comportamental *Observador* descrito para aplicações em programação orientada a objetos (Gamma **et al.** , 1996). No entanto, a sua definição pode ser estendida para diferentes tipos de sistemas distribuídos, pois permite uma alternativa desacoplada para notificar qualquer mudança de estado – do *publisher* – para uma quantidade qualquer de *subscribers*.

2.5.3 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 *assembler*, 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 depedências em classes concretas, como a do exemplo abaixo em que é instanciado uma classe concreta *ModelGateway* é instanciada diretamente no construtor da classe *PredictUseCase*:

```
class PredictUseCase():
    def __init__(self):
        self.model_gateway = ModelGatewayImpl()
```

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 *setter*. Ajustando o exemplo acima para aplicar a injeção de construtor, passamos a desacoplar a instanciação da dependência *ModelGateway* do construtor da classe *PredictUseCase*:

```
class PredictUseCase():
    def __init__(self, model_gateway: ModelGateway):
        self.model_gateway = model_gateway
```

2.6 Estilos Arquiteturais

Um estilo de arquitetura é a inspiração que está por trás da ideia da arquitetura de software (2.3). Tal inspiração é definida como um conjunto de princípios (2.4) e padrões (2.5) a serem adotados para construir uma estrutura que atenda os requisitos funcionais e não funcionais do sistema. 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 (Garlan e Shaw , 1993).

2.6.1 Arquitetura em Camadas

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

Um exemplo de aplicação de um sistema inteligente que emprega o estilo arquitetural em camadas pode ser visualizado na Figura 2.3. Na figura o *FastAPI Controller* é responsável por ser uma camada de apresentação do sistema para os clientes, o *Predict Service* é responsável por realizar o processamento da entrada de dados, buscar o modelo adequado do *ML Model Repository* para realizar a predição, para enfim retornar uma resposta ao cliente do sistema.

2.6.2 Arquitetura Hexagonal

A arquitetura hexagonal é um estilo arquitetural que busca separar as regras de negócio da aplicação dos detalhes de implementação, tais como o *framework*, 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 nos princípios de responsabilidade única (2.4.1), aberto-fechado (2.4.2), segregação de interface (2.4.4), e inversão de dependência (2.4.5). Além destes princípios, um padrão que é importante para o estilo, é o de injeção de dependências (2.5.3). Tal padrão sustenta o cumprimento do princípio da *Regra da Dependência* (Martin , 2017), na qual as

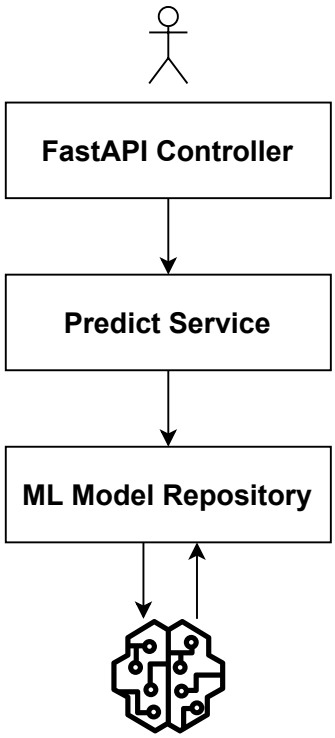

Figura 2.3: *Exemplo de Arquitetura em Camadas aplicada num sistema inteligente*

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 um porta será nada mais que uma simples interface que não possui nenhum conhecimento da implementação concreta até ser injetada em tempo de execução. Enquanto que, os *Adaptadores* são classes que adaptam uma interface em outra, e podem ser classificados de duas formas:

- Adaptador Condutor (ou Primário): São adaptadores que dependem de uma porta e injetam uma implementação concreta da porta, que no caso é um caso de uso do domínio da aplicação a ser utilizado.

- Adaptador Conduzido (ou Secundário): São adaptadores que são uma implementação concreta de uma porta e são injetados no domínio da aplicação. Vale destacar que o domínio apenas conhece a porta que foi implementada pelo adaptador, ou seja, o domínio conhece apenas a interface da porta.

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 do que ocorre na Arquitetura em Camadas (2.6.1), o domínio não possui conhecimento do que é externo a ele, ou seja, o que é interno encapsula totalmente a infraestrutura externa como banco de dados, controladores, sistemas de mensageria, etc.

Na Figura 3.3 é demonstrado uma aplicação do estilo num sistema inteligente análogo ao exemplo da Arquitetura em Camadas da Figura 2.3. 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.

2.7 Padrões de comunicação

A integração entre diferentes aplicações via rede lida com diversos desafios: conexão não confiável e lenta, diferenças nas aplicações, e mudanças inevitáveis. 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 afim de minimizar essas desvantagens (Hohpe e Woolf , 2012).

2.7.1 *RPC*

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.

2.7.2 *Publisher-Subscriber*

Publisher-Subscriber é uma forma de comunicação que se baseia no uso de protocolos de comunicação de mensageria e no padrão de projeto de mesmo nome (2.5.2). 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 pelo motivo de 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.

2.8 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.

A Linguagem de Modelagem Unificada (da sigla *UML* em 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 *blueprints* 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.

2.8.1 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 *abstraction-first* 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 2.7 se comportam em um sistema inteligente Seção 2.1 foram construídos dois sistemas de software:

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 capaz de servir modelos de aprendizado de máquina provisionados na plataforma `MLFlow`. Os modelos são servidos através de uma porta de entrada utilizando `FastAPI` e outra porta que utiliza o sistema de mensageria `NATS`. O repositório do sistema pode ser acessado em https://github.com/washington-ygor-tcc/intelligent-system.

De modo geral, ambos os sistemas foram desenvolvidos seguindo boas práticas de arquitetura de software (2.3), adotando variados princípios (2.4) e padrões de projeto (2.5). Uma outra importante decisão arquitetural, se diz respeito ao estilo arquitetural (2.6) adotado, que foi o estilo Hexagonal (2.6.2). Tais decisões foram fundamentais para cumprir com requisitos não funcionais essenciais para cumprir com os objetivos do projeto: os sistemas devem ter **baixo acoplamento** e um **alto grau de extensibilidade**. Desta forma, a arquitetura viabiliza a capacidade de integrar diferentes modelos de aprendizado de máquina (2.2) ao sistema inteligente ou – principalmente, que diferentes padrões de comunicação possam ser testados pela metodologia proposta pelo projeto.

No restante do capítulo é documentado a arquitetura da aplicação (2.8) mediante a adoção do Modelo C4 (2.8.1), apresentando o *Contexto* em que os clientes e os sistemas se encaixam e, a abstração de *Contêiner* da aplicação e, por fim, a abstração de *Componente* de cada um dos sistemas desenvolvidos.

3.1 Contexto

O diagrama presente na Figura 3.1, destaca os requisitos de cada um dos sistemas de software, pode-se destacar ambos os sistemas estão diretamente ou indiretamente relacionados com as necessidades do Cliente. O Cliente do sistema de *benchmark* solicita a realização de um teste de carga parametrizado. A depender do parâmetro de porta de entrada a ser utilizado, o sistema de *benchmark* irá consumir a porta de entrada adequada exposta pelo sistema inteligente. Ao final, com a coleta de métricas realiza pelo sistema de *benchmark*, o Cliente poderá analisar e avaliar a performance do consumo de um dado modelo por diferentes padrões de comunicação.

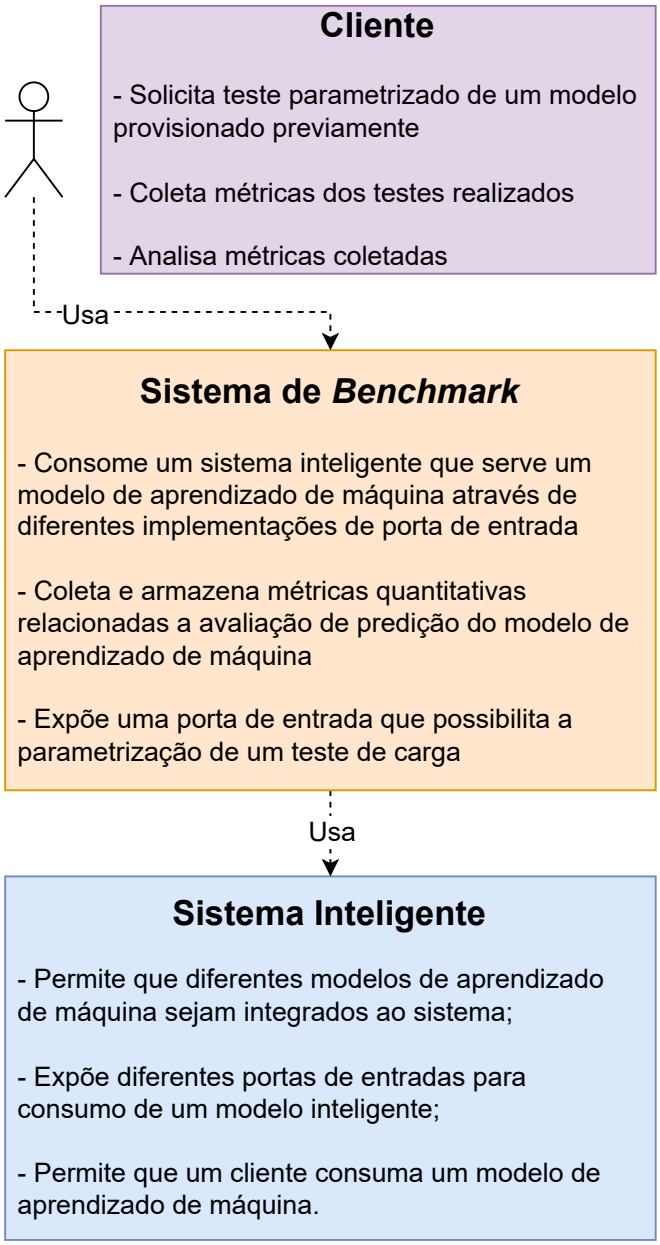

Figura 3.1: ***Diagrama de Contexto****. Representação dos requisitos funcionais e as relações dos atores da arquitetura.*

3.2 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 inteligente, o sistema de *benchmark*, e a infraestrutura necessária para o funcionamento de cada um dos sistemas.

O sistema inteligente é capaz de servir modelos de aprendizado de máquina – provisionados na plataforma `MLflow` – através de duas portas de entrada: uma síncrona via API RPC (2.7.1) e outra, assíncrona via sistema de mensageria (2.7.2). Apesar da suposta limitação de haver suporte para apenas duas portas de entrada, a adoção do Padrão Método Fábrica (2.5.1) permite a extensão da aplicação para qualquer outra porta de entrada, desde que seja feito a devida configuração e instanciação da nova porta de entrada no sistema inteligente.

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 funcional de passos é seguida, onde são construídas e enviadas diversas requisições de predições para o sistema inteligente. Ao final de cada requisição, métricas de performance são salvas em um repositório de dados, no caso, é utilizado uma planilha `CSV` para armazenar tais métricas.

Um outro ponto a se destacar, se a plataforma `MLflow`, que é responsável por gerenciar o ciclo de vida de modelos de aprendizado de máquina que serão utilizados para realizar predições pelo sistema inteligente. 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`).

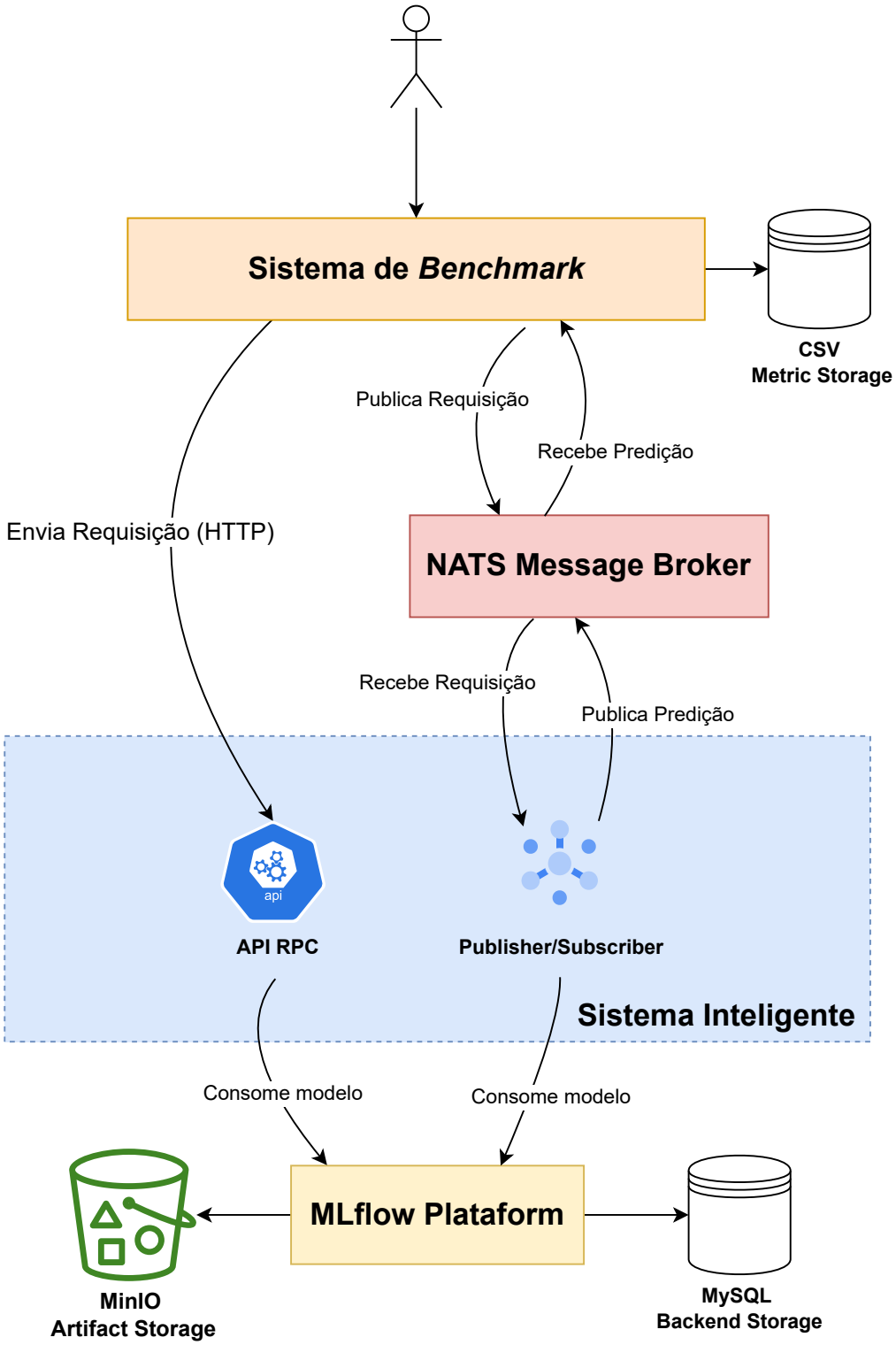

Figura 3.2: ***Diagrama de Contêiner****. Representação dos sistemas e das suas dependências de infraestrutura.*

3.3 Componente do Sistema Inteligente

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 entradas, requisitando um dado modelo provisionado no `MLflow`, realizando a predição utilizando a entrada recebida e retornando a predição como resposta para a porta de entrada adequada. Tal comportamento é encapsulado em um único caso de uso *Predict Request Handler Use Case* presente no domínio da aplicação.

As implementações das interfaces das portas do sistema – os adaptadores – se baseiam em chamadas para bibliotecas ou *frameworks*. Para o adaptador de porta de entrada síncrona, que serve uma API RPC, foi utilizado o framework `FastAPI`, enquanto que 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 referente foi implementado utilizando uma biblioteca interface para a plataforma do `MLFlow`.

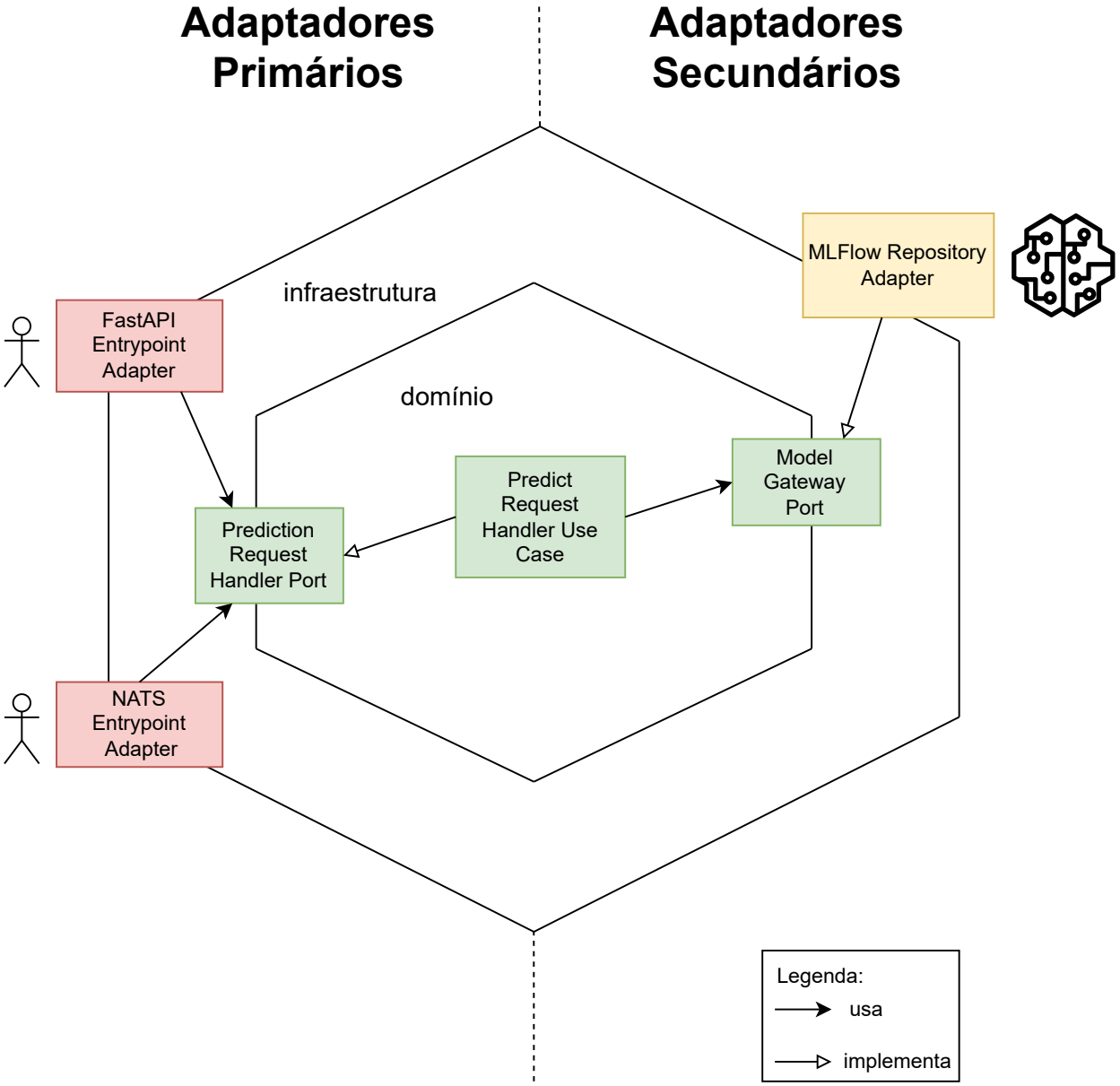

Figura 3.3: ***Diagrama de Componente****. Representação do componente núcleo do sistema inteligente.*

3.4 Componente do Sistema de *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).

Antes de qualquer coisa, um ponto a se notar é que não existem portas de entrada primárias no sistema, o que se traduz em um certo relaxamento do estilo arquitetural hexagonal adotado. Tal fato, se deve a natureza funcional do testes realizados, que em sua essência segue passos fixos para completar um dado teste. Soma-se a isso, a opção de linguagem de implementação do sistema – o `Python`, que permite disparos de funções dinamicamente. Por esse motivo, existe uma seta que liga diretamente o adaptador primário e o domínio do sistema.

Aporta de entrada do sistema, é exposta através de uma *CLI* (*Command Line Interface*) para o cliente do sistema. Tal interface permite a entrada de diferentes parâmetros pelo cliente, como:

1. tipo de padrão de comunicação,
2. número de requisições a serem realizadas,
3. um tempo de execução limite para os testes,
4. o tamanho do lote de requisições a serem realizadas,
5. um intervalo de tempo a ser tomado entre cada um dos lotes de requisição,
6. uma *flag* que define se as métricas devem ser salvas em uma planilha `CSV`,
7. uma *flag* que define se as métricas devem ser exibidas na saída padrão, e
8. uma *flag* que define se informações estatísticas devem ser exibidas na saída padrão.

A depender da entrada do tipo de padrão de comunicação, é instanciado a implementação de um cliente de serviço análogo a escolha.

Partindo para o domínio do sistema, existem dois casos de uso implementados, o *Run Benchmark Use Case* e o *Save Benchmark Use Case*. O primeiro deles 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.

As implementações das interfaces das portas – os adaptadores – dos geradores de identificador e marcação de tempo, são apenas chamadas para bibliotecas padrão do `Python`. Apesar da decisão questionável de implementar adaptadores que utilizam bibliotecas padrão da linguagem, em contrapartida, de fazer as chamadas para as bibliotecas padrões retamente no código do caso de uso. É explicado pelo fato de que identificadores e marcação de tempo são detalhes para o domínio, assim, o desacoplamento desses aspectos em relação ao domínio possibilita o uso de qualquer biblioteca externa ou o uso de algum algoritmo específico para gerar-los.

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.

Um último ponto a ser destacado, é em relação a implementação da porta do repositório de métricas, que é apenas uma interface para uma planilha `CSV`.

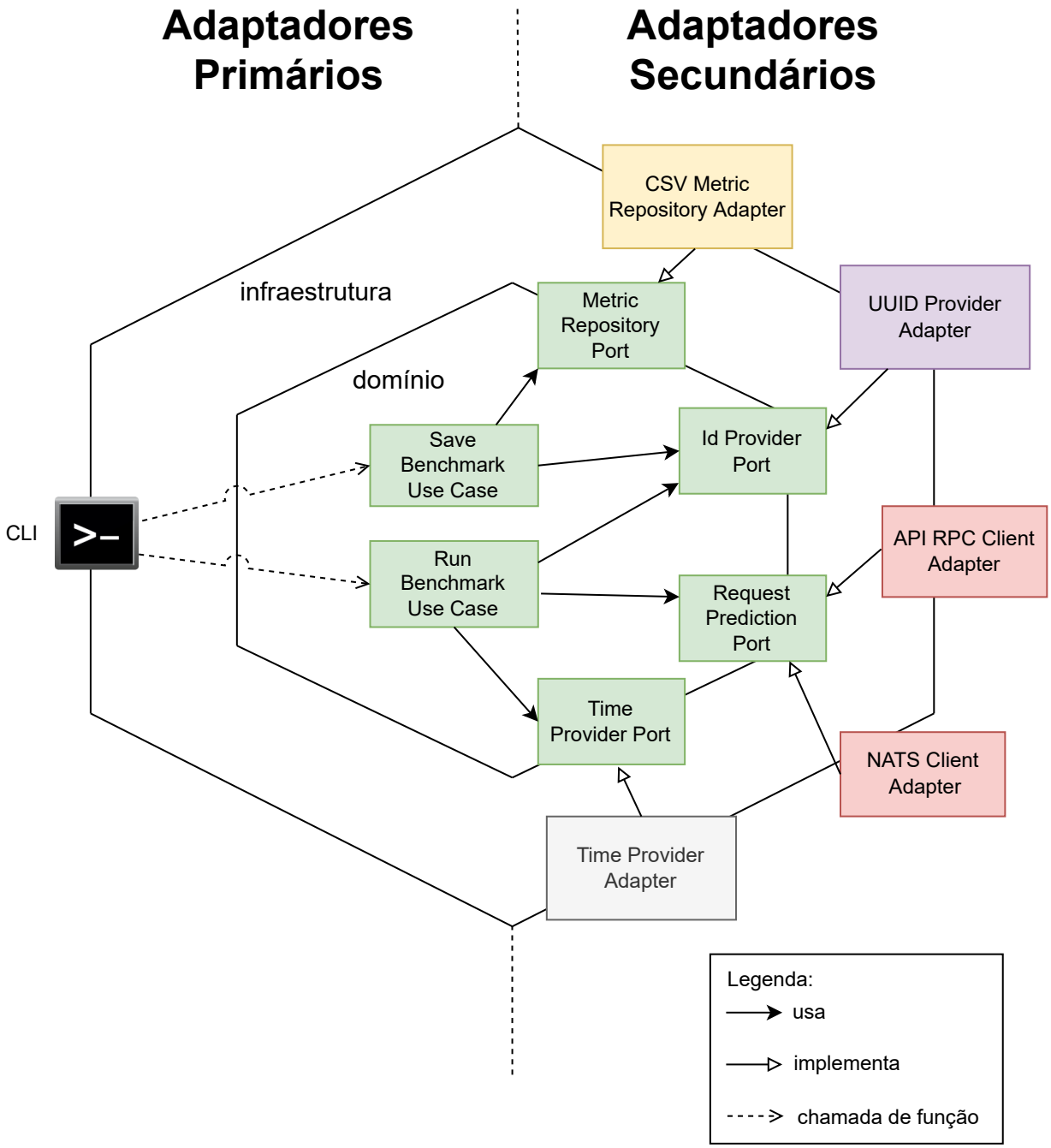

Figura 3.4: ***Diagrama de Componente****. Representação do componente núcleo do sistema de benchmark.*

Capítulo 4

Metodologia

Neste capítulo está presente a metodologia utilizada para obter os resultados...

Capítulo 5

Análise dos Resultados

Neste capítulo está presente a comparação entre os resultados obtidos...

Capítulo 6

Conclusões e considerações

Abaixo, apresentamos as principais conclusões sobre as frentes de trabalho apresentadas nas seções anteriores.

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.

Houve uma certa negligência em relação a criação de testes automatizados para ambos os sistemas, tal fato contribui para deixar um frágil pilar em relação a qualidade dos sistemas. No futuro, a criação de mais testes pode facilitar na criação de novas funcionalidades para ambos os sistemas, além de ser uma ótima alternativa para documentar cada pedaço dos sistemas.

Capítulo 7

Considerações pessoais

Abaixo, apresentamos as principais conclusões sobre as frentes de trabalho apresentadas nas seções anteriores.

7.1 Washington

7.2 Ygor

Bibliografia

Beck (2007) Kent Beck. **Implementation Patterns**. Citado na pág. 5

Booch et al. (2005) Grady Booch, James Rumbaugh e Ivar Jacobson. **The Unified Modeling Language User Guide Second Edition**. Citado na pág. 9

Brown (2018) Simon Brown. The C4 Model for Software Architecture, 6 2018. URL https://www.infoq.com/articles/C4-architecture-model/. Citado na pág. 9, 10

Bushmann et al. (1996) F Bushmann, R Meunier, H Rohnert e Soft Ware Architecture. **Pattern-Oriented Software Architecture**, volume 1. Citado na pág. 6

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

Ewen Callaway (2022) Ewen Callaway. What's next for AlphaFold and the AI protein-folding revolution. URL https://www.nature.com/articles/d41586-022-00997-5. Citado na pág. 1

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. 9

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, 3

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, 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. 7

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. 4

Nelson (1981) Bruce Jay Nelson. **Remote Procedure Call**. Tese de Doutorado, XEROX PARC. Citado na pág. 9

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

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