EDUARDO ROMANINI

UMA PROPOSTA DE ARQUITETURA COMPONÍVEL PARA SOFTWARE DE ANÁLISE DE INVESTIMENTOS.

São Paulo
2024

EDUARDO ROMANINI

UMA PROPOSTA DE ARQUITETURA COMPONÍVEL PARA SOFTWARE DE ANÁLISE DE INVESTIMENTOS.

Monografia apresentada ao PECE – Programa de Educação Continuada em Engenharia da Escola Politécnica da Universidade de São Paulo como parte dos requisitos para a conclusão do curso de MBA em Tecnologia de Software.

Área de Concentração:
Engenharia de Software.

Orientador:
Prof. Dr. Jorge Luis Risco Becerra.

São Paulo
2024

FICHA CATALOGRÁFICA

Romanini, Eduardo
UMA PROPOSTA DE ARQUITETURA COMPONÍVEL PARA SOFTWARE DE ANÁLISE DE INVESTIMENTOS / E. Romanini -- São Paulo, 2024.
59 p.

Monografia (MBA em Tecnologia de Software) - Escola Politécnica da Universidade de São Paulo. PECE – Programa de Educação Continuada em Engenharia.

1.Arquitetura Componível I.Universidade de São Paulo. Escola Politécnica. PECE – Programa de Educação Continuada em Engenharia II.t.

Nome: ROMANINI, Eduardo

Título: Uma proposta de arquitetura componível para software de análise de investimentos.

Monografia apresentada ao PECE – Programa de Educação Continuada em Engenharia da Escola Politécnica da Universidade de São Paulo como parte dos requisitos para a conclusão do curso de MBA em Engenharia de Software.

Aprovado em: / /

Banca Examinadora

Prof(a). Dr(a). _____

Instituição: _____

Julgamento: _____

Prof(a). Dr(a). _____

Instituição: _____

Julgamento: _____

Prof(a). Dr(a). _____

Instituição: _____

Julgamento: _____

DEDICATÓRIA

A minha família, minha inspiração e alicerce.
A minha namorada, meu amor e companheira.

AGRADECIMENTOS

A Universidade de São Paulo – USP por permitir a existência do curso e fomentar a evolução do ensino no nosso país.

Aos professores do curso de Engenharia de Software do PECE – Programa de Educação Continuada em Engenharia, pela dedicação à arte de ensinar

RESUMO

ROMANINI, E. **Uma proposta de arquitetura componível para software de análise de investimentos**. 2024. 59. Monografia (MBA em Engenharia de Software). Programa de Educação Continuada em Engenharia da Escola Politécnica da Universidade de São Paulo. São Paulo. 2024.

Nos últimos anos, o crescente volume de informações sobre investimentos, aliado à evolução tecnológica no mercado financeiro, tem impulsionado a demanda por soluções de software mais robustas, escaláveis e integráveis. Nesse cenário, arquiteturas componíveis, fundamentadas nos princípios de microsserviços, API-First, desenvolvimento nativo em nuvem e desacoplamento, emergem como uma alternativa promissora. Este trabalho apresenta uma proposta de adaptação de um software monolítico de análise de investimentos para uma arquitetura componível. O modelo componível se destaca por sua modularidade, escalabilidade e facilidade de manutenção, permitindo que cada serviço seja desenvolvido, implantado e escalado de forma independente. Além disso, a arquitetura proposta favorece a flexibilidade tecnológica, possibilitando o uso de diferentes linguagens e frameworks em cada componente, e melhora a resiliência, uma vez que falhas em um serviço não comprometem todo o sistema.

Palavras-chave: composable architecture, arquitetura componível, microsserviços, API-First, desacoplamento, projeção de rentabilidades.

ABSTRACT

ROMANINI, E. **A Proposal for a Composable Architecture for Investment Analysis Software**. 2024. 59. Monografia (MBA em Tecnologia de Software). Programa de Educação Continuada em Engenharia da Escola Politécnica da Universidade de São Paulo. São Paulo. 2024.

In recent years, the increasing volume of investment-related information, combined with technological advancements in the financial market, has driven the demand for more robust, scalable, and integrable software solutions. In this context, composable architectures, based on the principles of microservices, API-First, cloud-native development, and headless applications, have emerged as a promising alternative. This study presents a proposal for adapting monolithic investment analysis software into a composable architecture. The composable model stands out for its modularity, scalability, and maintainability, enabling each service to be developed, deployed, and scaled independently. Furthermore, the proposed architecture promotes technological flexibility by allowing the use of different programming languages and frameworks for each component, while enhancing system resilience, as failures in one service do not compromise the entire system.

Keywords: composable architecture, microservices, API-First, cloud-native, headless, financial software applications.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

Pág.

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
AWS	Amazon Web Services
CAPM	Capital Asset Pricing Model
ETL	Extract, Transform and Load
EKS	Elastic Kubernets Service
MACH	Microservices, API First, Cloud Native and Headless
VPC	Virtual Private Cloud

SUMÁRIO

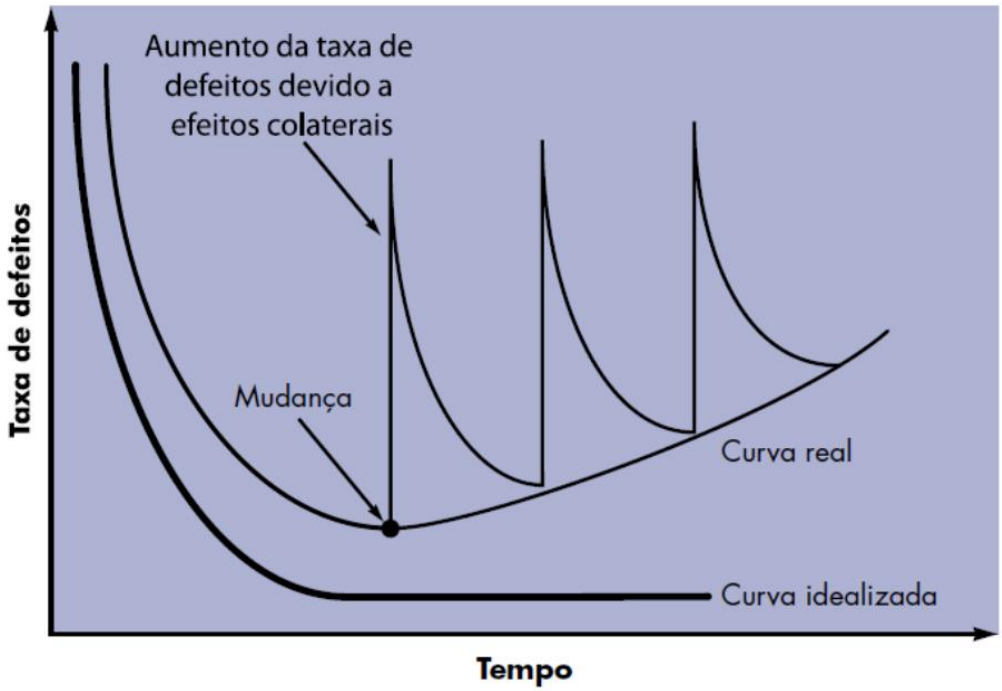
1. INTRODUÇÃO

1.1 Motivações

Nos últimos anos, a crescente difusão de informações sobre investimentos e o avanço das tecnologias financeiras têm impulsionado a adoção de soluções mais robustas e escaláveis no setor bancário, que se destaca como líder nos investimentos em TI (MEIRELLES, 2024). Esses investimentos foram impulsionados principalmente pelos impactos da Era Digital e pela desmaterialização de moedas, formas de investimentos e metodologias de pagamento (MEIRELLES, 2024). Esse contexto corrobora com uma demanda crescente por softwares que atendam os requisitos escalabilidade e reutilização de componentes.

Segundo os autores Mahdy e Fayad (2002), um dos atributos mais desejados no desenvolvimento de software é a estabilidade, embora seja um dos mais difíceis de ser alcançado. A exemplo disso, conforme pesquisa de Pressman e Maxim (2016) os softwares por estarem suscetíveis a alterações, à medida que essas ocorrem, podem gerar novos defeitos causadores de instabilidades ocasionando assim uma curva de defeitos conforme exibido na Figura 1.

Figura 1 - Taxa de defeitos em softwares em relação ao tempo



Fonte: Pressman e Maxim, 2016

Um dos principais objetivos da engenharia de software é a manutenção e desenvolvimento de forma a não impactar o software negativamente por meio de defeitos, sendo estabilidade definida como a facilidade com a qual o software ou um componente consegue ser alterado e evoluído enquanto preserva o máximo de seu design original (GROSSER; SAHRAOUI; VALTCHEV, 2002). Visando reduzir tais instabilidades, uma das áreas muito estudadas por profissionais de tecnologia é a arquitetura de software, a qual segundo Bass, Clements e Kazman (2003) é definida como um conjunto de estruturas para pensar sobre o sistema, levando em consideração elementos do software, as relações entre eles e suas propriedades.

Entre os estilos arquiteturais destacasse os microsserviços, os quais, surgem como solução para problemas de escalabilidade, flexibilidade e cooperatividade. Segundo Wiliams (2016), o uso de frameworks para microsserviços é uma técnica eficaz para implementação de softwares.

Segundo Dragon et al. (2017) microsserviços são unidades independentes de software que executam funções específicas e se comunicam por meio de APIs bem definidas e essa abordagem promove modularidade e escalabilidade em aplicações distribuídas.

Tendo como uma das premissas o uso de microsserviços o modelo de arquitetura componível (Composable Architecture), vêm ganhando amplo destaque por possibilitar a criação de softwares a partir da combinação de componentes independentes e intercambiáveis. Segundo previsões da Gartner (2024), 60% dos novos aplicativos de negócios serão construídos usando componentes reutilizáveis até 2025. Além disso, o mesmo estudo aponta que 35% dos aplicativos legados do setor público serão substituídos por soluções modulares mantidas por equipes multidisciplinares, demonstrando assim o potencial da arquitetura componível na consolidação dos microsserviços e futuro da engenharia de software.

1.2 Objetivo

Levando em consideração o constante investimento de instituições financeiras no setor de tecnologia e a abordagem crescente de aplicativos componíveis, este trabalho tem como objetivo apresentar uma proposta de adaptação de um software monolítico para uma a arquitetura componível (Composable Architecture).

O desenvolvimento e a codificação completa do software não estão incluídos no escopo da monografia, e os aspectos técnicos e econômicos relacionados ao mercado financeiro serão abordados de forma sucinta, considerando as limitações de tempo e a complexidade do tema.

1.3 Justificativas

A digitalização do setor financeiro impulsiona a adoção de soluções tecnológicas que atendam às demandas por inovação, eficiência e segurança. Nesse contexto, a arquitetura de software torna-se central, exigindo escalabilidade, flexibilidade e resiliência para atender às necessidades de adaptação e integração contínuas (BASS; CLEMENTS; KAZMAN, 2003; MEIRELLES, 2024).

A arquitetura de microsserviços tem se consolidado como uma solução promissora para a modernização de aplicações legadas (WILLIAMS, 2016). Como evolução desse conceito, a arquitetura componível (Composable Architecture) permite a construção de aplicações a partir de componentes independentes, reutilizáveis e facilmente combináveis. Essa abordagem promove maior inovação ao viabilizar ecossistemas modulares e ágeis (GARCÍA et al., 2021). Segundo a Gartner (2024), até 2025, 60% dos novos aplicativos empresariais adotarão componentes reutilizáveis, destacando a relevância da arquitetura componível no desenvolvimento de soluções escaláveis e sustentáveis.

A arquitetura componível expande os conceitos de modularização, ao permitir a reconfiguração dos componentes de uma aplicação de forma dinâmica, adaptando-se às mudanças do ambiente de negócios com maior agilidade (MAYS, 2023). Para o setor financeiro, isso representa uma vantagem estratégica significativa, pois permite que instituições possam lançar novos produtos e funcionalidades de maneira mais rápida, sem a necessidade de grandes reestruturações nos sistemas legados (JAMES; ROSSI; SANDERS, 2020).

Estudos recentes também demonstram que a adoção de uma abordagem componível não apenas melhora a escalabilidade e a flexibilidade, mas também aumenta a resiliência das aplicações. As organizações que implementam essa arquitetura relatam uma redução significativa nos custos de manutenção e uma maior capacidade de adaptação a novas exigências regulatórias, algo crítico no

ambiente financeiro altamente regulamentado (GARTNER, 2024; SINGH; JAIN, 2022). Além disso, ao utilizar componentes independentes, as aplicações tornam-se mais seguras, uma vez que a falha em um módulo não compromete necessariamente o restante do sistema (DRAGON et al., 2017).

A importância crescente das arquiteturas componíveis aliada as necessidades do setor financeiro refletem a demanda de criar aplicações que sejam não apenas tecnologicamente avançadas, mas também economicamente viáveis e sustentáveis a longo prazo. A tendência é que esse tipo de arquitetura continue a se expandir, em consonância com a crescente demanda por flexibilidade e interoperabilidade entre os sistemas financeiros globais. Dado esse contexto, o presente trabalho justifica-se pela necessidade de explorar e apresentar uma arquitetura que não apenas atenda às exigências atuais do mercado, mas que também ofereça uma base flexível para o futuro. Sendo que a pesquisa contribuirá para a literatura acadêmica ao destacar as vantagens e os desafios práticos da implementação dessas arquiteturas no ambiente financeiro, além de discutir a modernização de uma aplicação financeira como componentes combináveis.

1.4 Método de Pesquisa

Esse trabalho de monografia tem como plano de ação uma pesquisa descritiva com o objetivo de aprofundar os conhecimentos no que se refere a arquiteturas componíveis (Composable Architecture, com isso, a pesquisa se baseia no estudo de artigos e livros acadêmicos de maneira qualitativa, buscando sempre manter a qualidade dos materiais analisados e manutenção de concisão acadêmica.

Severino (2007) menciona que a pesquisa descritiva tem como principal objetivo retratar com precisão as características de uma situação, população ou área de estudo. Nesse contexto, abordaremos o tema com o objetivo de contextualizá-lo e referenciá-lo, a partir da leitura dos artigos mais relevantes disponíveis a partir de nossa pesquisa.

A revisão bibliográfica conta com o uso das principais ferramentas de pesquisa acadêmica, sendo essas, IEEE Explorer, Google Scholar e ACM Digital Library, os quais receberam como pesquisa as seguintes entradas “Composable Architecture”, “Composable Systems”, “Component-based Software” e “Component-based

Architecture”, além disso, serão pesquisados cada um dos temas vinculados ao princípio MACH de arquiteturas componíveis, sendo esses, “microsservices”, “API Fisrt”, “Cloud-Native” e “Headless”.

1.5 Estrutura do Trabalho

Este trabalho de monografia está divido em 4 capítulos, conforme descrito a seguir:

- O Capítulo 1 – Introdução aborda as motivações que justificam a escolha do tema, bem como os objetivos da pesquisa. Além disso, apresenta o método adotado e organiza o conteúdo para orientar o leitor sobre o desenvolvimento do estudo.

- O Capítulo 2 – Fundamentação Teórica fornece uma base teórica ao nivelar os principais conceitos relacionados a arquiteturas componíveis, microsserviços, softwares nativos em nuvem e soluções headless. Também inclui considerações sobre a teoria moderna de portfólio, relacionando-a ao contexto de investimentos.

- O Capítulo 3 – Processo de Adaptação de um Software de Investimentos para uma Arquitetura Componível explora o software de investimentos em análise, detalhando tanto os aspectos de negócio quanto os aspectos técnicos. O capítulo culmina na apresentação de uma proposta para adaptar a aplicação aos princípios de uma arquitetura componível.

- Por fim, o Capítulo 4 – Considerações Finais sintetiza as principais conclusões do estudo, destacando as contribuições da pesquisa e indicando oportunidades para trabalhos futuros.

2. FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os conceitos vinculados ao tema proposto com base em estudos relevantes, sendo abordado Arquitetura Componível aplicada a partir do conceito de MACH: Microsserviços, API Fisrt (API Primeiro), Cloud-native (Nativo em Nuvem) e Headless (Desacoplamento). Além disso, traçaremos um panorama básico sobre a teoria moderna do portfólio com o objetivo de construir o arcabouço acadêmico para o entendimento da aplicação apresentada no desenvolvimento.

2.1 Arquitetura Componível

A Composable Architecture, frequentemente traduzida como arquitetura componível, combinável ou compostável, tem se consolidado como uma tendência de destaque no desenvolvimento de software. Segundo o relatório de previsões da Gartner (2023), organizações que adotarem essa abordagem poderão superar seus concorrentes em até 80% na velocidade de implementação de novas funcionalidades. Além disso, o mesmo relatório projeta que, até 2025, 60% dos novos aplicativos empresariais serão desenvolvidos com base em componentes reutilizáveis. Essas estimativas evidenciam a crescente importância dessa abordagem, especialmente frente aos desafios de construção de software e à demanda por métodos de desenvolvimento cada vez mais ágeis e eficientes.

A arquitetura componível (ou Composable Architecture) é uma abordagem de design de software que promove a criação de sistemas modulares, formados por componentes independentes e reutilizáveis, capazes de ser recombinados para atender a diversas demandas de negócio (MAYS, 2023). De acordo com Ländin (2023), essa modularidade se beneficia da aplicação dos princípios MACH, definidos pela MACH Alliance (2022), como sendo:

- Microservices (Microsserviços): Funcionalidades de negócios independentes, que são desenvolvidas, implantadas e gerenciadas de forma separada.

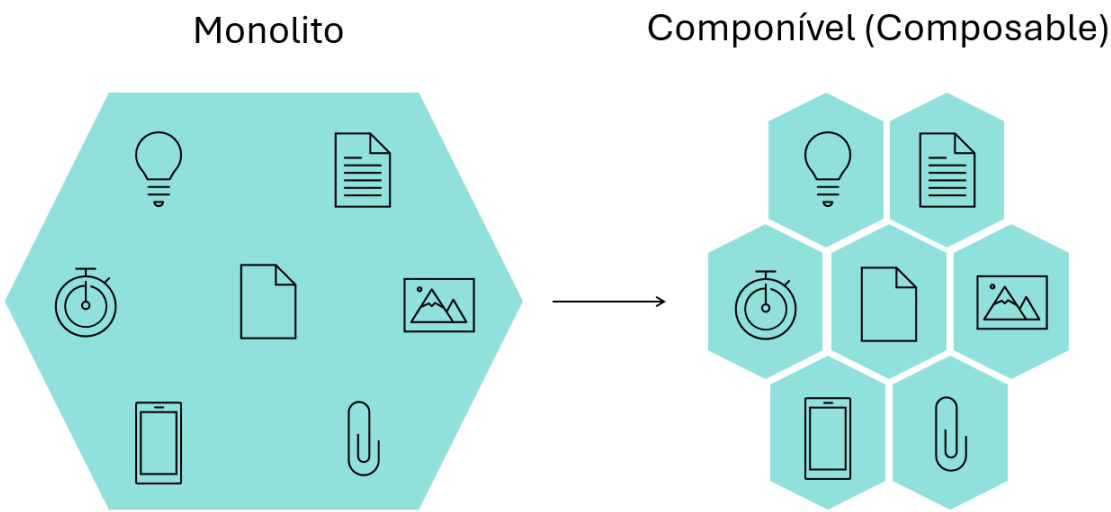
- API First (API Primeiro): A funcionalidade é exposta por meio de uma API, garantindo uma comunicação clara entre os componentes.

- Cloud-native (Nativo na Nuvem): A arquitetura aproveita ao máximo os recursos da nuvem, incluindo escalabilidade elástica e atualizações automáticas, indo além de simples armazenamento e hospedagem.

- Headless (Desacoplado): A apresentação da camada do usuário é separada da lógica do negócio, sendo agnóstica em relação a canais, linguagens de programação e frameworks.

Em sistemas componíveis os componentes funcionam como blocos de construção modulares, promovendo maior flexibilidade e escalabilidade no desenvolvimento de software. Ao contrário dos sistemas monolíticos, nos quais os componentes são fortemente acoplados, a arquitetura de microsserviços facilita a manutenção, atualização e evolução do software ao longo do tempo, permitindo que módulos sejam modificados ou substituídos sem a necessidade de reformular o sistema inteiro (RICHARDS & FORD, 2020). Em resumo, diferente de arquiteturas monolíticas onde o sistema é uma única peça, em uma arquitetura componível se constrói múltiplos apps independentes e os combina para formar o sistema (CRNKOVIC & LARSSON, 2002). Abaixo representação conforme Figura 2.

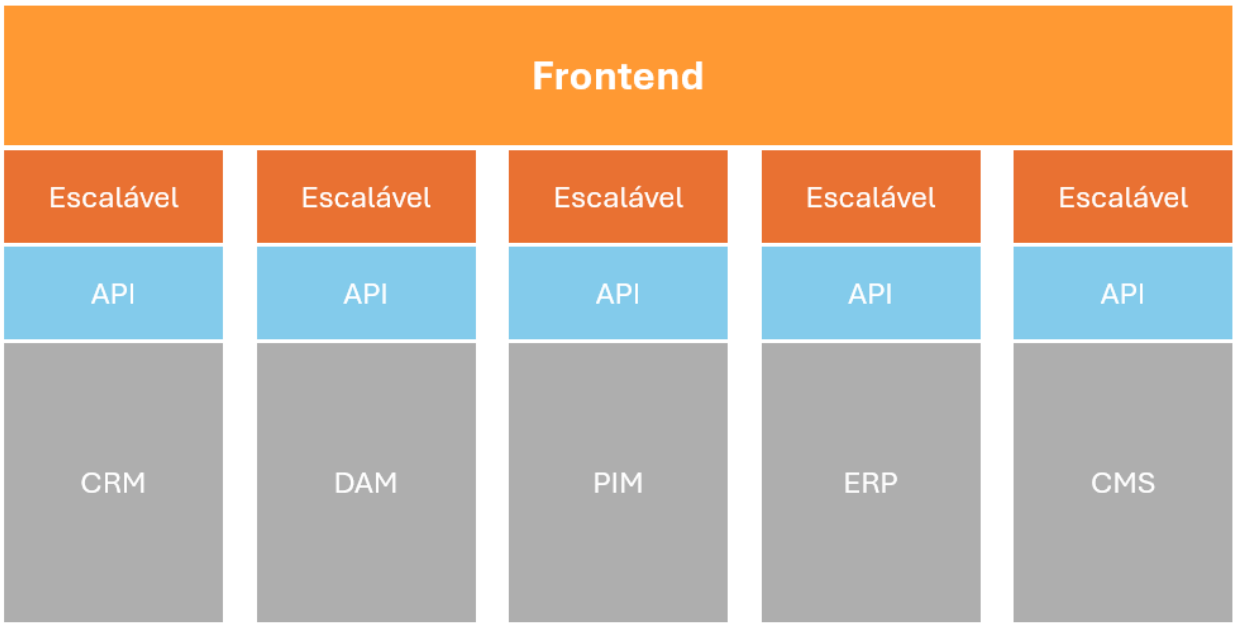
Figura 2 - Adaptação de sistema monolítico para componível



Fonte: Adaptado de Mays, 2023.

Apesar dos contextos de negócio serem tratados de maneira separada, necessário se faz, que os módulos sejam combináveis, para isso, arquiteturas componíveis se utilizam do modelo de API First (API Primeiro), a qual, é uma abordagem de desenvolvimento de software em que o design e a criação de APIs (Application Programming Interfaces) são tratados como o ponto central de todo o processo de desenvolvimento e não um produto secundário (MACH ALLIANCE, 2022). Nessa abordagem, as APIs são pensadas desde o início, garantindo que todos os serviços, módulos e funcionalidades de uma aplicação sejam expostos de forma coerente, consistente e acessível através dessas interfaces (JACOBSON et al., 2011). Isso facilita a integração com outros sistemas e permite que a aplicação seja facilmente escalável e adaptável a novos requisitos. Os eixos centrais de escalabilidade, modularidade e modelagem de APIs podem ser resumidos em conjunto arquitetural vide Figura 3, adaptada de Ländin.

Figura 3 - Modelo conceitual de arquiteturas componíveis



Fonte: Adaptado de Ländin, 2023

2.1.1 Microsserviços

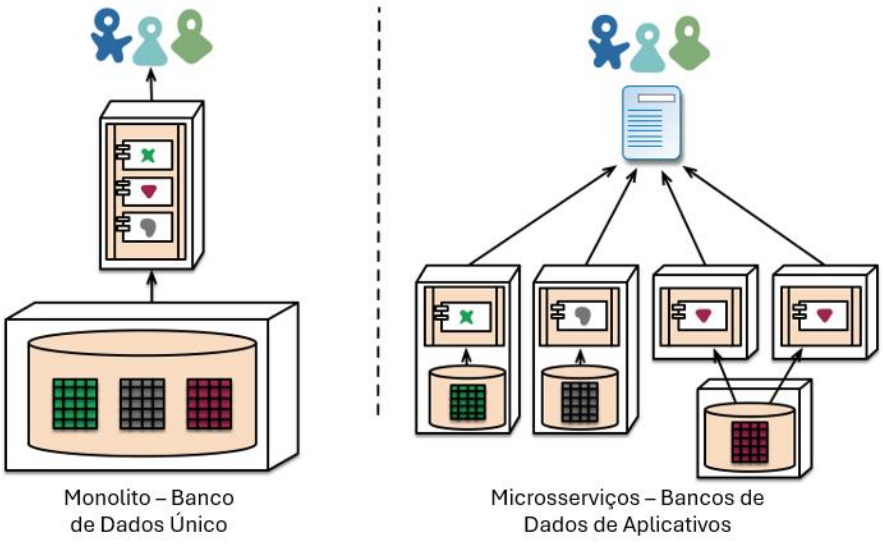
A crescente necessidade de criar sistemas de software mais flexíveis e escaláveis foi fator relevante para o surgimento da arquitetura de microsserviços como uma evolução das tradicionais arquiteturas monolíticas. Tendo os microsserviços ganhado popularidade à medida que as organizações passaram a

enfrentar desafios cada vez maiores em gerenciar aplicações complexas (NEWMAN, 2015; DRAGONI et al., 2017).

De acordo com Bruce e Pereira (2018), a principal vantagem da arquitetura de microsserviços reside na capacidade de isolar serviços de maneira autônoma, permitindo que sejam desenvolvidos, implantados e escalados independentemente. Cada microsserviço executa dentro de seu próprio processo e se comunica com outros serviços por meio de protocolos de comunicação padronizados, o que permite que sistemas sejam construídos de forma flexível, promovendo a adaptabilidade e facilitando a manutenção, haja visto que cada serviço pode ser atualizado ou modificado sem impactar diretamente os demais.

Segundo Lewis e Fowler (2014), a modularização em uma arquitetura de microsserviços refere-se à decomposição de uma aplicação em serviços pequenos, autônomos e especializados, cada um responsável por uma funcionalidade específica e além de descentralizar os serviços em módulos conceituais, os microsserviços também descentralizam as decisões sobre o armazenamento de dados. Enquanto aplicações monolíticas geralmente preferem um único banco de dados lógico para dados persistentes, os microsserviços incentivam que cada serviço gerencie seu próprio banco de dados, vide Figura 4.

Figura 4 – Paralelo entre arquiteturas monolíticas e microsserviços

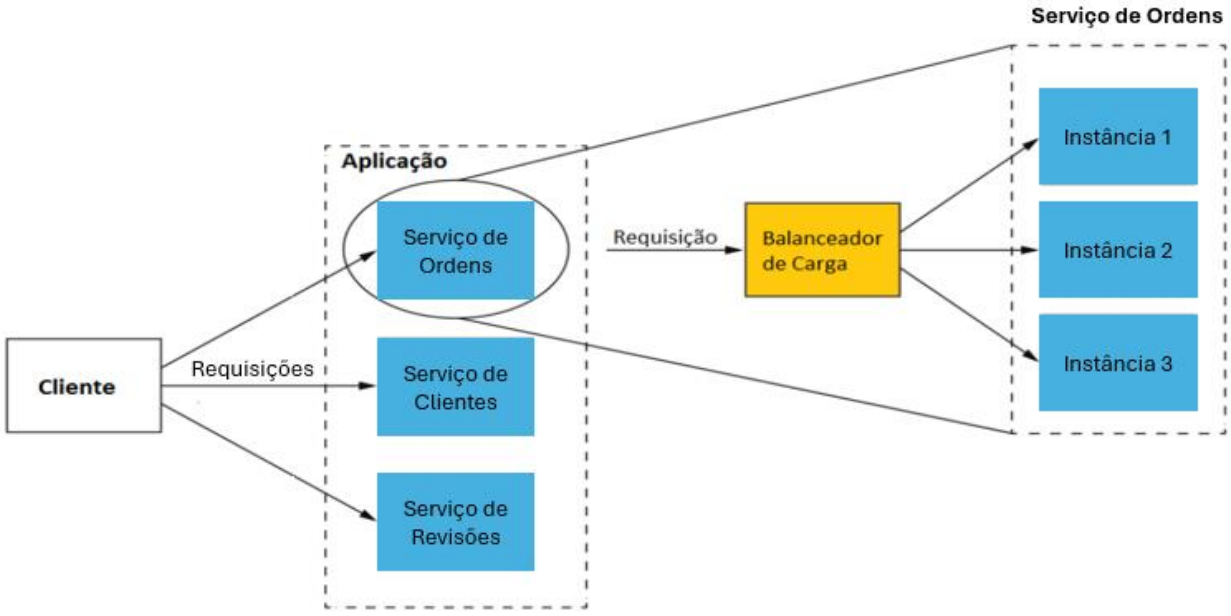


Fonte: Adaptado de Lewis e Fowler, 2014.

Essa abordagem traz diversas vantagens, como a possibilidade de utilizar diferentes tipos de bancos de dados otimizados para as necessidades específicas de cada serviço, o que pode resultar em melhor desempenho e escalabilidade. Ademais, a independência dos serviços permite que alterações na estrutura de dados de um serviço não afetem diretamente os outros, facilitando a manutenção e a evolução da aplicação. Por fim, a utilização de bancos de dados distintos pode aumentar a resiliência do sistema, pois falhas em um banco de dados específico não comprometem a operação dos demais serviços.

Além das vantagens mencionadas, Richardson (2018) destaca que a natureza modular dos microsserviços permite a escalabilidade independente de cada serviço, conforme a demanda. Diferente dos sistemas monolíticos, onde é necessário escalar toda a aplicação, os desenvolvedores podem aumentar a capacidade apenas dos serviços com maior carga, otimizando recursos e reduzindo custos operacionais. Essa abordagem melhora o desempenho do sistema e pode ser um fator importante na redução de custos, como ilustrado na Figura 5, adaptada de Richardson (2018), onde somente o serviço de ordens foi escalado em novas instâncias devido a uma demanda específica.

Figura 5 – Escalabilidade de instâncias em arquitetura de microsserviços.

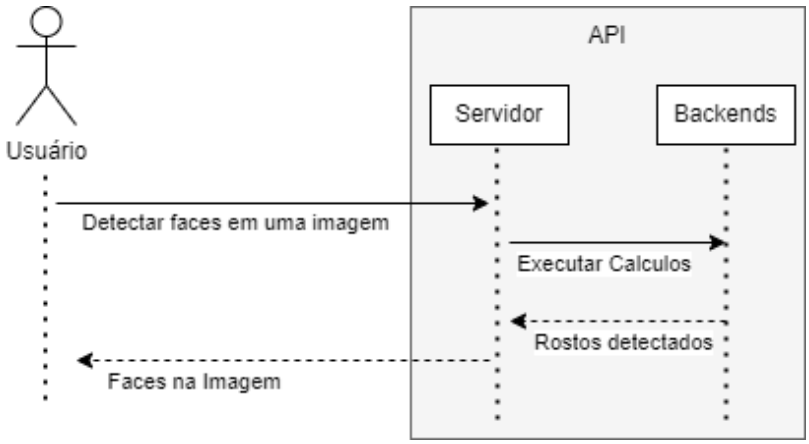


Fonte: Adaptado de Richardson, 2018.

2.1.2 API First

A API (Interface de Programação de Aplicativos) é um conjunto de definições e protocolos que permite a comunicação entre sistemas, possibilitando que diferentes softwares interajam entre si sem a necessidade de expor ao usuário questões superficiais, como a apresentação visual (GEEWAX, 2022). Através das APIs, é possível habilitar a troca de dados entre máquinas de forma transparente, sem a necessidade de interação direta com a interface de usuário (GEEWAX, 2022).

Figura 6 – APIs web como interface de comunicação



Fonte: Adaptado de Geewax, 2022

Ainda segundo Geewax (2022), o uso de APIs oferece uma série de vantagens no desenvolvimento de software, especialmente no contexto de sistemas distribuídos e integrações entre diferentes plataformas. Sendo uma das principais vantagens, conforme Figura 6, a capacidade de abstrair e proteger a lógica de negócios e os dados sensíveis por meio de uma interface, evitando assim a exposição direta dos serviços subjacentes. Isso garante que apenas as funcionalidades previamente autorizadas e controladas estejam acessíveis aos consumidores da API.

Tendo em vista a relevância do tema vinculado ao desenvolvimento de APIs, um conceito que vem ganhando muito destaque é o modelo de API First. Segundo Jin, Sahni e Shevat (2019), a abordagem API First coloca o design da API no centro do processo de desenvolvimento, garantindo que todos os serviços e funcionalidades do sistema sejam desenvolvidos com base nas necessidades da interface promovendo uma integração mais eficiente entre os sistemas, haja visto que, as APIs são planejadas e definidas antes do desenvolvimento de qualquer parte da

aplicação. Dentre as consequências de se modelar a API antes do desenvolvimento do sistema, observa-se uma melhora a consistência e reusabilidade das interfaces, já que a API serve como contrato para o desenvolvimento. Por fim, como destaca Geewax (2022), esse modelo é particularmente útil em arquiteturas de microsserviços, onde a comunicação entre componentes é fundamental para a escalabilidade e manutenção do sistema.

2.1.3 Nativo em Nuvem

Segundo Jesse Davis (2019) ir para a nuvem é mais sobre como você modela a sua aplicação do que onde propriamente onde ocorre o deploy. O conceito de Cloud-Native (Nativo em Nuvem) diz respeito a técnicas e métodos de modelagem de aplicações para serem disponibilizadas em ambiente na nuvem, e assim, aderirem da melhor forma as vantagens de escalabilidade, redundância, resiliência e capacidade de adaptação contínua. Por fim, Davis (2019), define que um software nativo em cloud deve ser altamente distribuído, deve operar em um ambiente em constante mudança e está, ele próprio, em constante transformação, sendo caracterizados pelas necessidades e soluções abaixo:

- Softwares nativos em nuvem precisam ser resiliente e capaz de se adaptar a falhas e mudanças na infraestrutura, sejam elas planejadas ou não. Por isso, indicasse projetos modulares, compostos por partes independentes que limitem os impactos de eventuais falhas.

- A necessidade de constantes lançamentos em produção é facilitada por softwares que adotem microsserviços, sendo essa arquitetura mais indicada do que arquitetura monolítica onde todo o modelo de release contempla todo o software.

- Softwares em que o volume de solicitações oscila significativamente são amplamente beneficiados pelas ferramentas de escalabilidade disponíveis em infraestruturas em nuvem, contudo, para melhor uso é importante que as peças sejam modularizadas de forma a escalar apenas as mais impactadas pela demanda.

- A arquitetura distribuída é uma necessidade em softwares redundantes. Distribuir cópias redundantes de componentes evita que falhas locais causem grandes impactos. Dessa forma, é essencial utilizar recursos distribuídos, incluindo serviços em nuvem.

Considerando os pontos destacados por Davis (2019), referência em arquiteturas nativas em nuvem, um software que visa integrar-se de maneira eficaz ao ambiente de nuvem deve ser modular, com componentes independentes e baseado em microsserviços. Além disso, deve adotar uma arquitetura distribuída, o que facilita o deploy em múltiplas regiões, reduzindo o risco de falhas geograficamente localizadas. É essencial que o sistema seja escalável dinamicamente, permitindo o ajuste automático para lidar com flutuações de demanda, e seja capaz de executar múltiplas instâncias do mesmo módulo ou componente. Abaixo alguns dos conceitos vinculados a aplicações nativas em nuvem (PAI; KUMAR, 2021; KRATZE ,2018):

- Baseados em Microsserviços: Os microsserviços dividem a aplicação em módulos independentes, cada um focado em uma função de negócio específica e com seus próprios dados. Esses módulos se comunicam entre si via APIs, permitindo que cada serviço seja desenvolvido, testado e escalado independentemente.

- Baseados em Containers: Os containers isolam logicamente as aplicações, permitindo que rodem de forma independente dos recursos físicos. Essa abordagem evita que os microsserviços interfiram uns nos outros, além de otimizar o uso de recursos do servidor e permitir múltiplas instâncias do mesmo serviço.

- Baseados em APIs: As APIs conectam microsserviços e containers, facilitando a comunicação entre os módulos de forma simplificada. Elas também garantem a segurança e a manutenção dos serviços, atuando como a "cola" que une os serviços desacoplados.

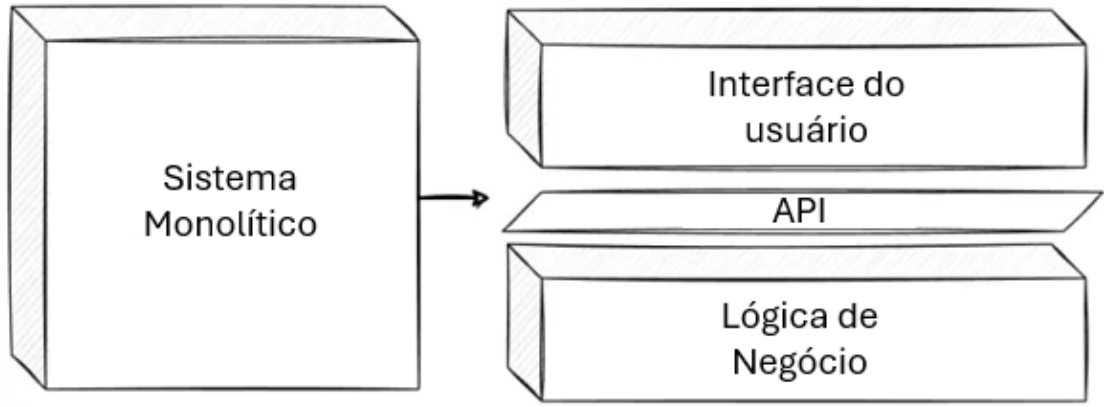
- Orquestrados Dinamicamente: Ferramentas de orquestração de containers, como Kubernetes, são utilizadas para gerenciar o ciclo de vida dos containers. Elas realizam a alocação de recursos, balanceamento de carga, agendamento de reinicializações após falhas e o provisionamento e

deploy de containers em nós de clusters de servidores, assegurando a eficiência do sistema

2.1.4 Headless (Desacoplamento)

Arquitetura headless, por vezes traduzida como desacoplada, é um conceito de desenvolvimento de software baseado na separação da camada de interface do usuário da camada de lógicas de negócio, respectivamente conhecidas como frontend e backend (OZKAYA, 2023). Tendo como base o conceito de desacoplamento entre as interfaces de apresentação e de negócios, ambas as tecnologias podem ser desenvolvidas e mantidas independentemente, sendo a comunicação entre elas comumente realizada por meio de APIs.

Figura 7 – Comparação sistema monolítico e sistema desacoplado



Fonte: Adaptado de Ozkaya, 2023

Conforme a Figura 7, em arquiteturas desacopladas, a interface do usuário se situa como uma aplicação independente da lógica de negócio, sendo o consumo de informações realizado por meio de APIs de serviço, as quais atuam como uma camada de comunicação. Segundo Ozkaya (2023), nesse tipo de arquitetura, as tecnologias utilizadas em cada uma das aplicações podem ser diferentes, visando a absorção do melhor ferramental de software para cada um dos contextos. Ou seja, a interface do usuário pode ser desenvolvida em uma linguagem totalmente desatrelada da camada de lógica de negócios, o que permite que ambos os times trabalhem independentemente nas suas funcionalidades, desde que respeitem os contratos de API.

Por fim, além da flexibilidade tecnológica inferida pela separação das camadas do software, a arquitetura desacoplada também oferece benefícios em termos de desenvolvimento ágil, haja vista que possibilita a criação de grupos de trabalho distintos para a evolução de cada uma das partes. Sendo assim, é possível evoluir a camada de apresentação ao usuário sem impacto direto na camada de serviços de negócio. Os benefícios acima citados tornaram o desacoplamento um tema cada vez mais recorrente no âmbito de desenvolvimento de software e, por conta disso, cada vez mais incorporado em modelos arquiteturais, como, por exemplo, o modelo de arquitetura componível.

2.2 Teoria moderna do portfólio

De maneira geral, ao investir capital, o objetivo principal do investidor é maximizar os retornos de um portfólio de ativos. No entanto, a busca por altas rentabilidades está intrinsecamente ligada ao risco de perdas significativas do capital principal (GUNTHER, 1980).

Em 1952, o economista Harry Markowitz desenvolveu um estudo focado nas relações entre risco e retorno dos capitais investidos, propondo metodologias para maximizar o retorno médio de um portfólio, respeitando um grau de risco aceitável. Em seu artigo *Portfolio Selection*, Markowitz destaca a diversificação como o princípio central da teoria, ou seja, a estratégia de manter posições em diferentes ativos para alcançar uma combinação mais eficiente em termos de risco e retorno (MARKOWITZ, 1952). A partir dessa premissa, a construção de um portfólio de investimentos não deve avaliar cada ativo isoladamente, mas considerar seu valor e função dentro da composição da carteira. Com base em medidas de variância e correlação, o desempenho de um ativo individual é menos relevante do que seu impacto no portfólio como um todo. Assim, a escolha de ativos para a carteira deve considerar sua sinergia e diferentes respostas a cenários de mercado. Segundo Markowitz (1952), o retorno do portfólio é obtido pela soma dos retornos individuais dos ativos, ponderados pelos seus respectivos pesos no portfólio.

Embora o modelo de Markowitz seja amplamente aceito e utilizado para a otimização de carteiras de investimentos, é importante destacar que o modelo se baseia em dados passados de rentabilidade para estimar o risco e o retorno. Isso significa que, embora o modelo considere a relação histórica entre ativos, ele não

projeta diretamente o retorno futuro. A projeção de rentabilidades futuras é uma prática comum no mercado financeiro, especialmente para ajustar as expectativas dos investidores conforme as condições econômicas e os fatores de risco futuros (BRIGHAM; EHRHARDT, 2017). Modelos mais avançados, como o Capital Asset Pricing Model (CAPM) ou abordagens de previsão por séries temporais, podem ser usados para estimar retornos futuros, levando em conta variáveis macroeconômicas e de mercado (ROSS et al., 2007).

Diversas instituições, como o Banco Central do Brasil, por meio do Boletim Focus, e bancos privados, produzem projeções periódicas de índices financeiros importantes, como inflação, taxa de juros e crescimento do PIB (BANCO CENTRAL DO BRASIL, 2023). Esses índices funcionam como parâmetros para modelar as expectativas de retorno dos ativos, proporcionando uma base para a gestão de portfólios de investimentos. No entanto, como afirmam Sharpe et al. (2014), é crucial que os gestores de portfólio não apenas considerem o histórico, mas também as expectativas de variabilidade dos ativos e sua correlação com as variáveis econômicas futuras a fim de modelar a rentabilidade esperada para uma carteira de investimentos.

2.3 Considerações do Capítulo

O capítulo apresentado objetiva fundamentar teoricamente dos principais conceitos abordados neste trabalho, contextualizando o modelo arquitetural componível, que integra diferentes modelos para formar uma estrutura única voltada para o reaproveitamento e integração de módulos. Foram explorados os pontos centrais do modelo MACH: microsserviços, API primeiro, desenvolvimento nativo em nuvem e arquitetura desacoplada. Por fim, foi introduzida a teoria moderna do portfólio de Harry Markowitz, estabelecendo uma base de conhecimento para o modelo de negócio do projeto, além de abordar os mecanismos de projeção de curvas de rentabilidade futuras, relevantes para o desenvolvimento da monografia.

3. PROCESSO DE ADAPTAÇÃO DE UM SOFTWARE DE INVESTIMENTOS PARA UM ARQUITETURA COMPONÍVEL

Este capítulo constitui a parte principal do trabalho, nele aplicaremos os conceitos de uma arquitetura componível a um software de otimização de portfolios e projeção de rentabilidades. Primeiramente será contextualizada a aplicação existente em um viés funcional e técnico e, posteriormente, aplicaremos os quatro pilares de uma arquitetura componível para adaptar nossa aplicação ao modelo proposto.

3.1 Contexto de Negócio

Nesse tópico abordaremos o discricionário da aplicação e suas funcionalidades tendo como objetivo principal nivelar o conhecimento frente a ferramenta. Sendo assim, faremos uma breve introdução ao nicho de mercado e, posteriormente, apresentaremos o domínio da aplicação e seus requisitos técnicos e funcionais.

3.1.1 Introdução ao Software

No contexto de investimentos, as projeções de rentabilidade são essenciais na seleção de produtos, permitindo ajustes na carteira de investimentos baseados em variáveis econômicas, como taxas de juros, inflação e crescimento econômico, elementos cruciais para o planejamento financeiro e a diversificação de portfólios.

Instituições financeiras utilizam modelos preditivos para calcular o retorno esperado de investimentos, auxiliando na tomada de decisões e estratégias de diversificação, entretanto esses modelos muitas vezes não são disponibilizados ao público, limitando o acesso dos pequenos investidores. Com isso, o software apresentado propõe um modelo matemático para cálculo de rentabilidade esperada com base em indicadores de mercado apresentados por meio de uma interface visual.

3.1.2 Domínio do software

O domínio da aplicação se concentra na funcionalidade de projetar rentabilidades futuras para produtos de investimento, atendendo à crescente demanda do mercado por ferramentas que forneçam previsões financeiras confiáveis.

Visando atender ao público geral que não possui acesso a ferramentas auxiliares para projeção de investimentos, a aplicação deverá possuir uma proposta interface de usuário intuitiva e desacoplada que permita aos clientes utilizarem a ferramenta. Nessa interface, será possível realizar autenticações, cadastrar produtos e realizar projeções de investimentos.

3.1.3 Requisitos do Software

Com o domínio do software devidamente contextualizado, antes de apresentarmos a arquitetura atual da aplicação e iniciarmos a transformação para um modelo componível, é importante esclarecer o escopo de negócios com base nos requisitos do software. Abaixo, os requisitos estão destacados e organizados em funcionais e não funcionais:

Tabela 1 - Requisitos do Software de Projeção de Rentabilidades

Requisitos de Software Funcionais	
RF01	Aplicação deve possuir uma interface web acessível em domínio público.
RF02	O usuário deve poder se cadastrar na aplicação informando um e-mail e uma senha.
RF03	Sistema deve possuir um serviço de recuperação de senha a partir do e-mail cadastrado.
RF04	Usuários devem ter a possibilidade de criar uma carteira de investimentos que ficará salva e atrelada ao seu perfil.
RF05	Usuários devem ter a possibilidade de adicionar seus produtos de investimentos a carteira.
RF06	O software tem aderência apenas a títulos de renda fixa, fundos de investimentos e dólar.
RF07	Sistema deve gerar um gráfico de projeção de rentabilidade para a carteira contemplando os próximos 12 meses.
RF08	O sistema deve permitir que o usuário visualize o histórico de rentabilidade da carteira nos últimos 12 meses.
RF09	Usuários devem poder acessar uma carteira uma vez salva.

Continua na próxima página

Tabela 1 - Requisitos do Software de Projeção de Rentabilidades

RF10	Quando o usuário for outra aplicação, o sistema deve possibilitar a integração via API REST com autenticação de par de chaves.
RF11	Aplicação deve ser capaz de extrair, transformar e salvar as informações de mercado dos principais órgãos brasileiros.
Requisitos não funcionais	
RNF01	Ferramenta deve ser construída com um modelo de arquitetura componível.
RNF02	Aplicação deve ser escalável.
RNF03	Interface gráfica deve ser responsiva a ações do usuário.

Fonte: Autor

Por fim, a distinção entre requisitos funcionais e não funcionais visa trazer maior clareza ao escopo da aplicação, garantindo maior confiabilidade no desenvolvimento do software. Segundo Sommerville (2011), os requisitos funcionais descrevem as funcionalidades que o sistema deve oferecer para atender às necessidades dos usuários, especificando os comportamentos que o sistema deve apresentar em resposta às interações dos usuários. Por outro lado, os requisitos não funcionais definem atributos de qualidade que o sistema deve possuir, como desempenho, segurança e usabilidade, determinando como o sistema deve operar.

3.2 Contexto de tecnologia atual do software

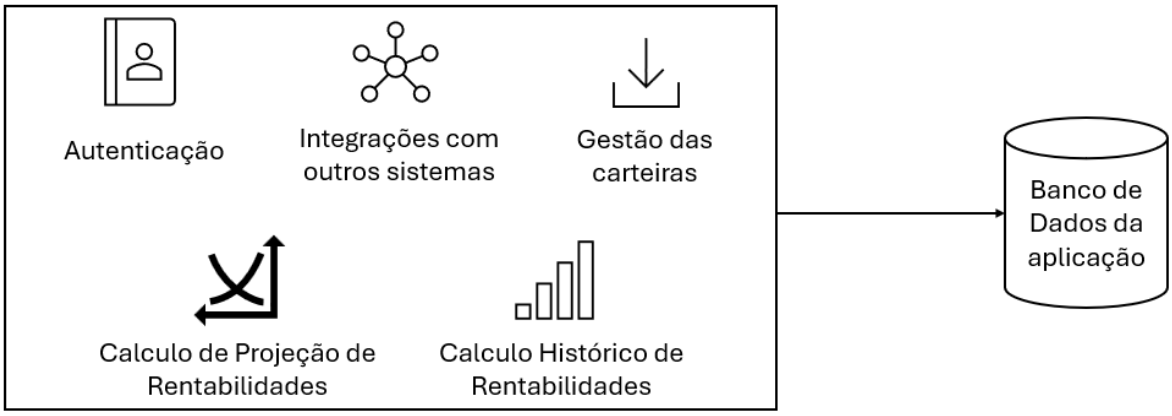
Este tópico examina a organização atual do sistema antes das adaptações necessárias para a transição ao modelo de arquitetura componível e aplicação dos princípios MACH. Com o objetivo de contextualizar o entendimento sobre a aplicação, serão detalhados os aspectos centrais de sua arquitetura monolítica atual, incluindo suas integrações existentes e as ferramentas de desenvolvimento.

3.2.1 Arquitetura atual

A aplicação utiliza uma arquitetura monolítica, na qual toda a lógica e funcionalidades estão agrupadas em um único módulo, sem separação explícita entre os diferentes componentes. Essa abordagem resulta em uma forte

interdependência entre as funcionalidades da aplicação, além disso, a aplicação não possui uma interface visual desenvolvida.

Figura 8 - Arquitetura atual do software de previsão de rentabilidades.



Fonte: Autor

Conforme Figura 8, todas as funcionalidades estão integradas em um único bloco de código. Ou seja, o mesmo conjunto de código integra todas as funcionalidades da ferramenta, sendo elas descritas abaixo:

- A funcionalidade de autenticação é responsável por garantir que apenas usuários autorizados tenham acesso à aplicação. Ela inclui todos os mecanismos de validação e controle de acesso, autenticando usuários com base nas credenciais fornecidas.

- A funcionalidade de gestão de carteiras está relacionada aos processos de criação, leitura, atualização e exclusão de carteiras de investimentos armazenas no banco de dados da aplicação.

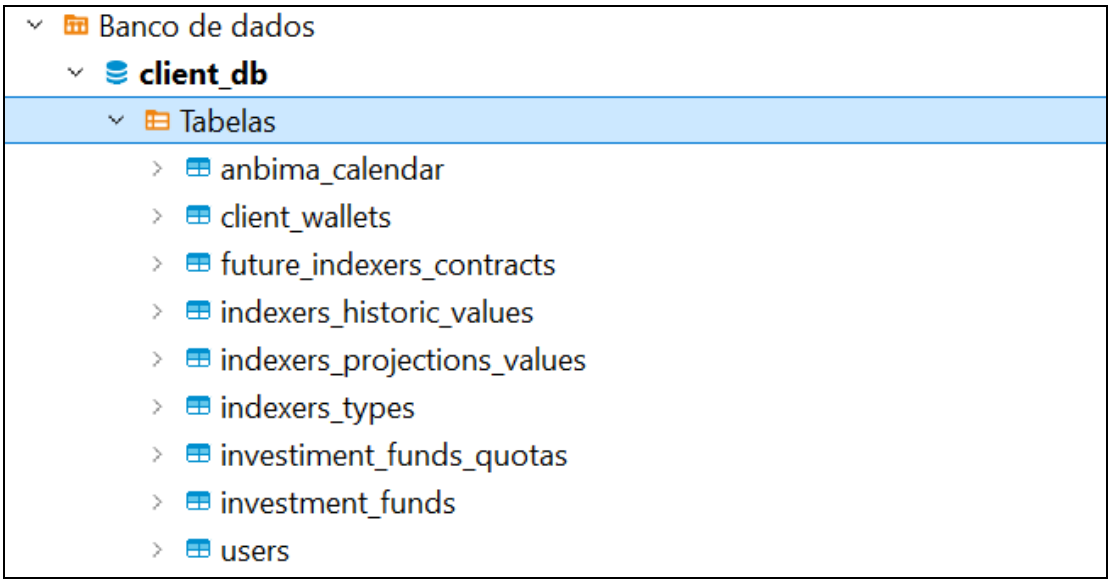
- O cálculo de projeção de rentabilidades inclui mecanismos para estimar retornos potenciais da carteira com base em indicadores futuros. Essa funcionalidade realiza os cálculos financeiros e projeções baseadas nos principais índices e dados de mercado.

- O mecanismo de cálculo de rentabilidade históricas permite simular a rentabilidade passados das carteiras. Os dados históricos dos produtos de investimentos são mantidos dentro do banco de dados e as análises são geradas com base nas informações pré-existentes.

- As integrações com outras ferramentas bases de mercado ocorrem por meio de ETL (Extração, Transformação e Carga), no qual, o monolito extrai dados dos principais relatórios econômicos de instituições do mercado financeiro, permitindo a integração dos dados externos ao banco de dados da aplicação.

Além disso, observa-se na Figura 8 que todo o sistema se comunica com um único banco de dados, o qual, centraliza o armazenamento de todas as informações da aplicação, desde dados de autenticação até os registros de rentabilidade e projeções de investimento. Na Figura 9, exemplificamos as tabelas presentes no banco de dados denominado “cliente_db”:

Figura 9 – Banco de dados do sistema atual



Fonte: Autor

As tabelas do monólito estão todas situadas em um único banco de dados, o que pode trazer diversos problemas, especialmente à medida que a aplicação cresce em complexidade e volume de dados, sendo ponto fundamental a ser tratado na adaptação para um modelo componível. Abaixo a descrição de cada uma das tabelas:

Tabela 2 - Descrição das tabelas no banco de dados atual

anbima_calendar	Calendário de feriados coletados da Anbima.
client_wallets	Tabela com id de referência do cliente, id da carteira e os produtos de investimento pertencentes aquela carteira.
future_indexers_contracts	Valor dos contratos futuros de índices de bolsa e S&P500.
indexers_historic_values	Histórico dos principais índices de mercado (IGP-M, CDI, Selic, IPCA e Dólar).
indexers_projections_values	Valor futuro dos principais índices de mercado (IGP-M, CDI, Selic, IPCA e Dólar).
indexers_types:	Id referencial para cada um dos índices presentes no sistema.
investment_funds_quotas	Valor histórico das cotas dos fundos de investimentos.
investment_funds	Dados dos fundos de investimento
users	Tabela com dados do usuário

Fonte: Autor

Com o objetivo de coletar dados do mercado de investimento o software busca informações de instituições financeiras. As instituições e os dados coletados estão apresentados na Tabela 3.

Tabela 3 - Plataformas integráveis e dados coletados

Instituição	Descrição
ANBIMA	Calendário de feriados do mercado financeiro
BACEN	Histórico da taxa Selic, IGPM, IPCA e cadernetas de poupança.
BACEN	Boletim focus com o futuro dos índices SELIC, IPCA e IGP-M e Dólar.
CETIP	Histórico da taxa CDI
CMV	Histórico de cotas de fundos de investimentos
Yahoo Finance	Histórico do ibovespa, S&P e Dólar
B3	Contratos futuros dos índices S&P e Ibovespa

Fonte: Autor

Atualmente todos os dados os dados são extraídos, transformados e carregamento no banco de dados da aplicação. Essa metodologia popularmente conhecida como ETL é processo de integração de dados usado para mover dados de várias fontes para outro banco de dados (KIMBALL; ROSS, 2013). Sendo as três fases do processo definidas como:

- Extração (Extract): Nesta fase, os dados são coletados de diversas fontes, que podem incluir bancos de dados, arquivos, APIs, ou sistemas de relatórios. O objetivo é capturar e centralizar os dados relevantes para análise.

- Transformação (Transform): Os dados extraídos são processados e convertidos para um formato adequado ao destino. Nessa etapa, podem ocorrer tarefas como limpeza de dados, remoção de duplicidades, normalização, agregação e aplicação de regras de negócio.

- Carregamento (Load): Os dados transformados são carregados no banco de dados final ou data warehouse, onde podem ser acessados para fins analíticos ou operacionais.

Esse processo permite integrar dados de diversas fontes e deixá-los prontos para análise, proporcionando maior confiabilidade e consistência para a tomada de decisões.

Por fim, destaca-se que o software não está disponível em um ambiente produtivo sendo executado apenas em uma máquina física sem disponibilização para os usuários finais.

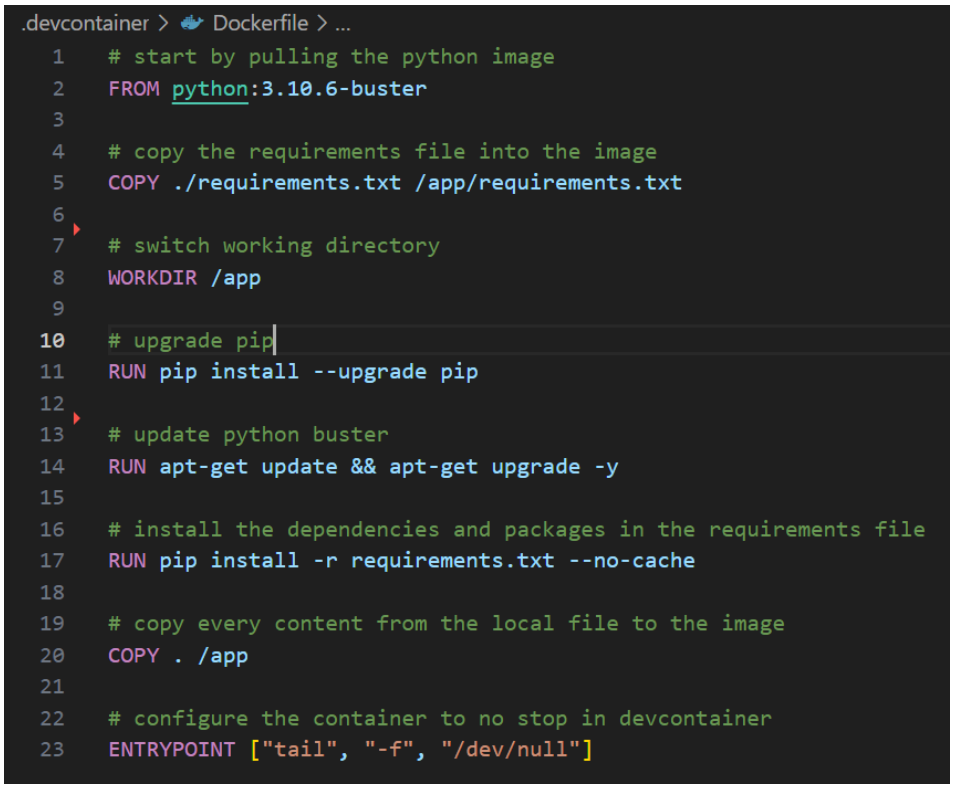
3.2.2 Ferramental de desenvolvimento atual

Este tópico apresenta a linguagem de desenvolvimento utilizada e informações relacionadas ao código da aplicação. Apesar de a aplicação ser estruturada como um monólito, ela foi submetida ao processo de conteinerização. Essa abordagem encapsula o código e suas dependências em um ambiente isolado, denominado contêiner, permitindo que a aplicação funcione de maneira consistente e previsível

em diferentes ambientes, abrangendo as etapas de desenvolvimento, teste e produção.

A aplicação é executada em um contêiner baseado em uma imagem Docker que instancia um ambiente Linux com Python 3.10.6-buster pré-instalado (DOCKER HUB, 2024), conforme apresentado na Figura 10.

Figura 10 – Configuração do Container pré-existente



Fonte: Autor

O código apresentado define um Dockerfile para a criação de uma imagem Docker com o Python pré-configurado. Essa configuração permite a instalação das dependências necessárias ao desenvolvimento, a preparação do contêiner e sua manutenção ativa para interações via API.

O código define um Dockerfile para a criação de uma imagem Docker personalizada com Python pré-configurado. Essa configuração inclui a instalação de dependências necessárias ao desenvolvimento, a preparação do contêiner e sua manutenção ativa para interações via API. Atualmente, essa configuração é utilizada exclusivamente no ambiente de desenvolvimento, conforme evidenciado pela

instrução na linha 23 do Dockerfile, que mantém o contêiner em execução contínua. Portanto, para a implantação em ambiente de produção, serão necessários ajustes.

Adicionalmente, os bancos de dados da aplicação estão sendo gerenciados em volumes locais, não havendo, até o momento, uma instância dedicada em ambiente produtivo. O arquivo docker-compose.yml configura esses volumes, define as variáveis de ambiente e gerencia a orquestração dos serviços, incluindo o banco de dados.

Figura 11 – Arquivo docker.compose para configuração local

Fonte: Autor

O arquivo docker-compose.yml orquestra dois serviços em uma rede personalizada compose-bridge: um serviço FastAPI e um banco de dados MySQL. O MySQL é configurado para expor a porta 3306, utilizando um volume nomeado db para persistir dados localmente e importando um script SQL (init.sql) no momento da inicialização. Ambos os serviços compartilham a mesma rede, garantindo comunicação direta entre os componentes.

Por fim, importante destacar que:

- Apesar de o código estar sendo executado localmente em um contêiner ainda não existe um modelo de Dockerfile produtivo capaz de executar a aplicação.

- Os bancos de dados e volumes estão todos sendo gerenciados em ambiente local ainda não possuindo infraestrutura em nuvem.

3.3 Proposta de solução adaptada para um contexto componível

Neste capítulo, objetiva-se realizar as adaptações necessárias na estrutura do projeto para alinhá-lo a um modelo componível, aproveitando, assim, os benefícios que essa arquitetura oferece. Para atingir esse objetivo, será aplicado o método de adaptação faseado, dividido nas seguintes etapas:

1. Adaptar o software para uma organização em microsserviços e promover o desacoplamento entre as camadas de apresentação e de negócios;
2. Remodelar as API’s, garantindo conformidade com boas práticas de design e integração;
3. Desenvolver uma proposta arquitetural em nuvem, com base nos microsserviços ajustados.

3.3.1 Adaptação a microsserviços e desacoplamento

Iremos apresentar uma adaptação da arquitetura monolítica atual do software para um modelo de microsserviços. Para isso, primeiramente, definiremos a aplicação em contextos delimitados com base nos requisitos do software.

Contextos delimitados (Bounded Contexts) são divisões estratégicas que delimitam fronteiras claras dentro de um domínio, garantindo que cada contexto possua um modelo de dados e regras de negócio específicos. Essa abordagem permite gerenciar a complexidade do sistema ao isolar subdomínios e reduzir interdependências, sendo um princípio essencial no Domain-Driven Design (FRITZSCHE, 2024). Com base nos requisitos do software enunciados no contexto de negócio, Tópico 3.1.3, Tabela 1.

Tabela 4 – Requisitos de Software Organizados por Contexto Delimitado

AUTENTICAÇÃO	
RF01	Aplicação deve possuir uma interface web acessível em domínio público.
RF02	O usuário deve poder se cadastrar na aplicação informando um e-mail e uma senha.
RF03	Sistema deve possuir um serviço de recuperação de senha a partir do e-mail cadastrado.
RF10	Quando o usuário for outra aplicação, o sistema deve possibilitar a integração via API REST com autenticação de par de chaves.
GESTÃO DE CARTEIRAS	
RF04	Usuários devem ter a possibilidade de criar uma carteira de investimentos que ficará salva e atrelada ao seu perfil.
RF05	Usuários devem ter a possibilidade de adicionar seus produtos de investimentos a carteira.
RF09	Usuários devem poder acessar uma carteira uma vez salva.
CORE DE RENTABILIDADE	
RF06	O software tem aderência apenas a títulos de renda fixa, fundos de investimentos e dólar.
RF07	Sistema deve gerar um gráfico de projeção de rentabilidade para a carteira contemplando os próximos 12 meses.
RF08	O sistema deve permitir que o usuário visualize o histórico de rentabilidade da carteira nos últimos 12 meses.
CRON DE COLETA DE DADOS	
RF11	Aplicação deve ser capaz de extrair, transformar e salvar as informações de mercado dos principais órgãos brasileiros.

Fonte: Autor

A separação dos requisitos funcionais e não funcionais em contextos de negócio permite transformar uma arquitetura monolítica em uma estrutura composta por serviços independentes. Cada serviço, ao ser atribuído a um contexto delimitado, pode ser gerenciado e modificado de forma autônoma. Os requisitos funcionais, ao

serem agrupados em contextos delimitados, garantem que cada módulo tenha responsabilidades específicas e bem definidas. Já os requisitos não funcionais, como segurança, escalabilidade e desempenho, são tratados de forma transversal, assegurando a consistência e o alinhamento do sistema como um todo (BASS, 2015; GARTNER, 2020).

Abaixo um breve descritivo das responsabilidades de cada um dos sistemas modelados a partir dos contextos delimitados:

- Serviço de Autenticação: Responsável por controlar o acesso dos usuários à aplicação. Ele fornece funcionalidades de registro de novos usuários, login e recuperação de senha. Desenvolvido em Python e com conexão a um banco de dados isolado para gestão de usuários. O sistema é acessado via API REST e possuirá uma camada anticorrupção.

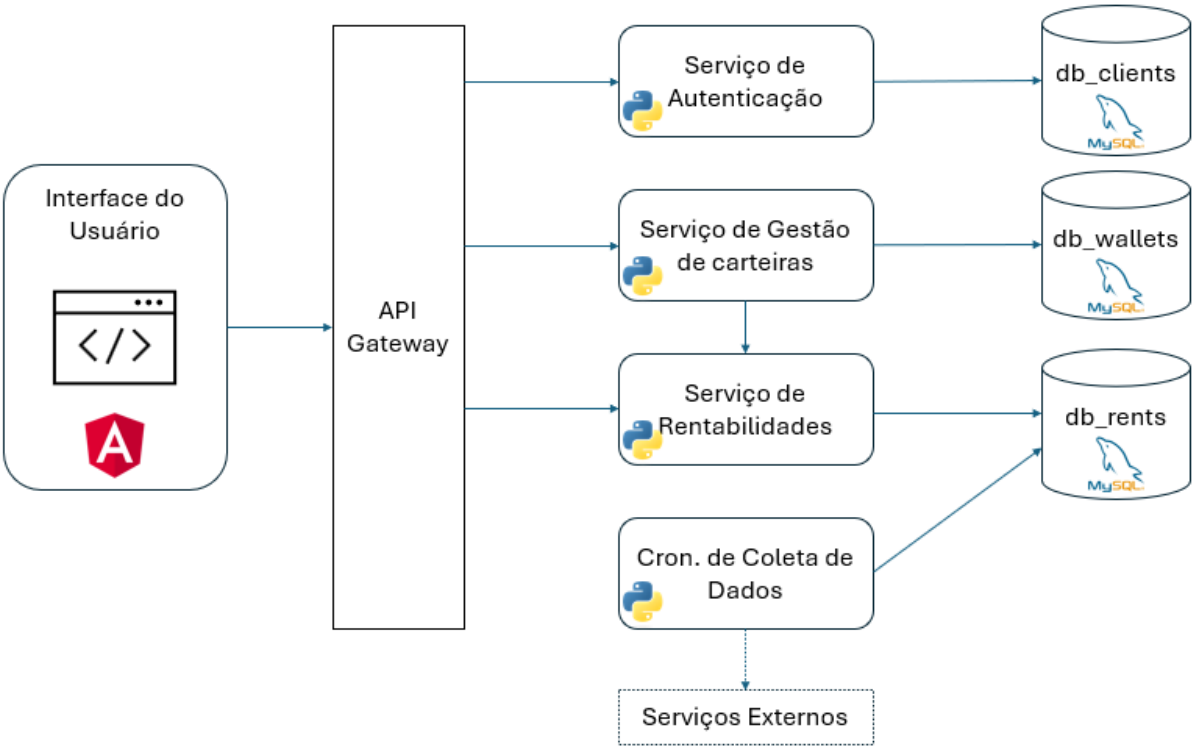
- Serviço de Gestão de Carteiras: Permite que os usuários criem e gerenciem suas carteiras de investimentos. Oferece funcionalidades para adicionar e remover produtos de investimento, salvar e remover carteiras. O sistema possui um mecanismo de decodificação de token para validar a autenticidade do usuário, sendo o token gerado pelo serviço de autenticação e encaminhado para a peça via API REST. Toda a peça de gestão de carteiras é desenvolvida em Python e possui um banco de dados isolado.

- Serviço de Rentabilidades: Tem como função principal o cálculo de rentabilidades, sejam essas passadas ou projetadas. Possui suporte para renda fixa, fundos de investimento e dólar, sendo os históricos e projeções de rentabilidades consultados em um banco de dados compartilhado com o serviço de armazenamento de rentabilidades.

- Serviço de Coleta de Dados: consiste em um cron de coleta com jobs agendados em Python, cuja finalidade é obter informações de mercado e alimentar o banco de dados por meio de um processo de operações de ETL (Extração, Transformação e Carregamento) programadas. Desempenha um papel fundamental na manutenção do sistema e possui camadas anticorrupção para todos os processos de obtenção de dados.

Com os contextos delimitados definidos, é possível modelar a aplicação em micros serviços. Nesse processo, destaca-se a importância do conceito de desacoplamento, que promove a separação entre a interface de apresentação e a interface de negócio por meio de um API Gateway. O API Gateway atua como um serviço intermediário, responsável por receber requisições dos clientes, processá-las e redirecioná-las para os serviços apropriados. Dentro desse contexto, sugere-se uma arquitetura desacoplada que também adota bancos de dados independentes para cada serviço, conforme evidenciado na Figura 12.

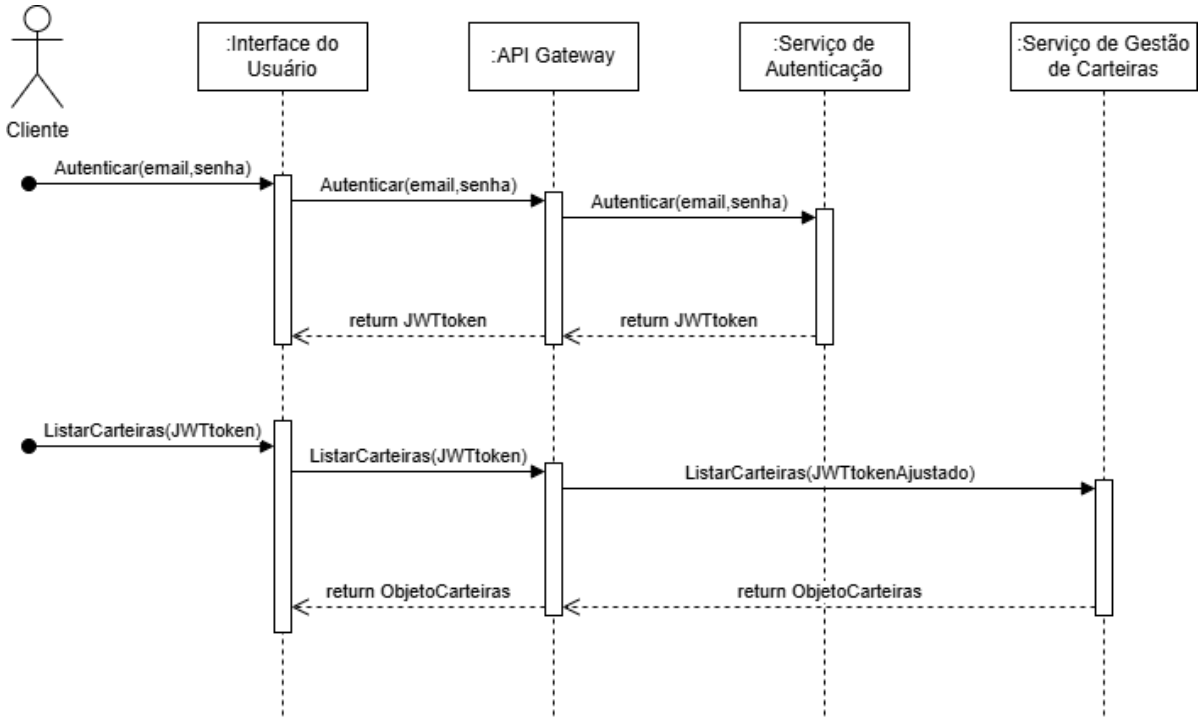
Figura 12 – Proposta Arquitetural da aplicação desacoplada



Fonte: Autor

A arquitetura proposta, que incorpora um API Gateway, oferece uma gestão aprimorada das requisições, permitindo que a comunicação com os microsserviços seja roteada conforme as tarefas executadas pelos usuários. Essa abordagem elimina a necessidade de interação direta da interface do usuário com os microsserviços, o que aumenta a segurança e simplifica a gestão da comunicação, o gateway pode ser utilizado como uma solução de mercado ou desenvolvimento local. A seguir, apresenta-se um exemplo de fluxo de comunicações:

Figura 13 - API Gateway e Fluxo de Chamadas



Fonte: Autor

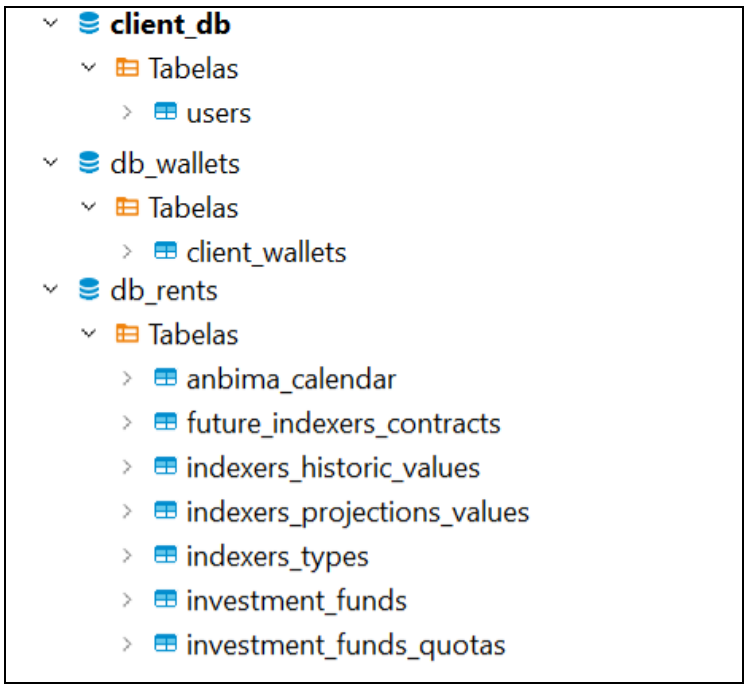
A Figura 13 ilustra uma interação no sistema utilizando o API Gateway como camada de comunicação entre a interface do usuário e os microsserviços. Nesse fluxo, ao autenticar-se na aplicação, o usuário fornece suas credenciais (e-mail e senha), que são validadas pelo serviço de autenticação. Em caso de sucesso, é gerado um token JWT, que é armazenado no contexto da aplicação e utilizado para autenticar futuras requisições aos serviços, no exemplo acima, caracterizado pela listagem de carteiras. Além disso, importante se faz, exemplificar que o processo de autenticação pode ter dupla validação complementar, onde a peça do API Gateway verifica alguns parâmetros e o microsserviço requisitado pode realizar validações adicionais.

Por fim, na abordagem proposta de solução arquitetural, enfatiza-se a necessidade de substituir o banco de dados único atualmente utilizado, conforme ilustrado na Figura 9, por um modelo desacoplado. Essa transformação visa melhorar a gestão da informação e promover maior aderência aos princípios da arquitetura de microsserviços. Nesse contexto, o banco de dados único foi dividido

em três bancos de dados independentes, cada um com gestão autônoma, descritos a seguir:

- db_clients: Responsável pelo armazenamento de informações vinculadas a gestão de usuário no contexto de autenticação.
- db_wallets: Banco de dados de gestão de carteiras de investimentos utilizado pelo serviço de gestão de carteiras.
- db_rents: Banco de dados responsável pelo armazenamento de informações de rentabilidade histórica e projeções de índice consumido pela peça de serviços de rentabilidade e abastecido pelo cron. de coleta de dados.

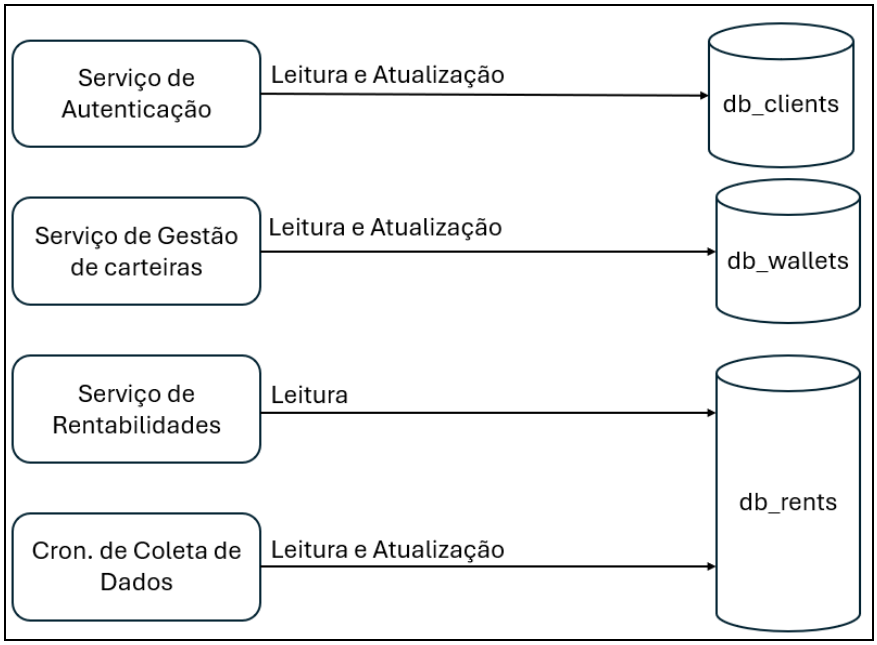
Figura 14 - Modelo de dados configurado para o contexto de microsserviços



Fonte: Autor

Ao segregar os bancos de dados e suas respectivas funções de armazenamento, é possível ter melhor gestão e planejarmos o desenvolvimento de novas funcionalidades de maneira mais sustentável. Abaixo mapa de alçadas de utilização dos bancos:

Figura 15 - Interações com banco de dados



Fonte: Autor

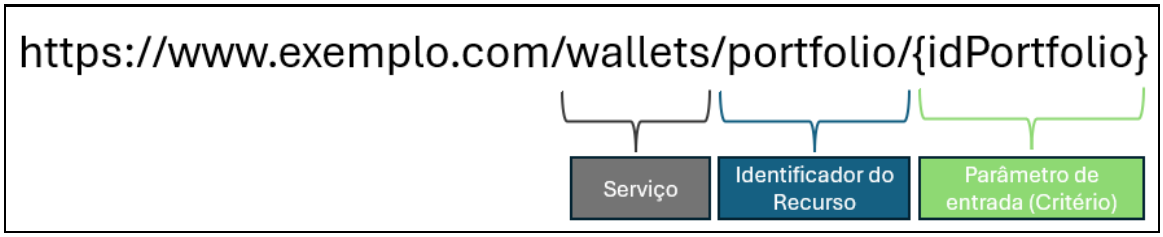
Ao segregar os bancos de dados e diferenciar as responsabilidades de atualização, é possível validar com maior precisão os contextos de mudança no banco de dados e identificar eventuais falhas de abastecimento. Conforme ilustrado na Figura 16, observa-se uma distinção no uso do banco de dados de rentabilidades: a parte responsável pela coleta de dados tem a função de alimentar o banco com novas informações, enquanto o componente de rentabilidades consome esses dados retornando ao usuário projeções e históricos de rentabilidade.

3.3.2 Modelagem de APIs

Neste tópico, não se pretende listar todas as APIs da aplicação, mas apresentar um processo de modelagem para construção de APIs que permita maior previsibilidade e independência aos times responsáveis pelas interfaces de apresentação e negócios. Para garantir uma gestão eficiente de APIs, foram estabelecidos padrões de construção integrados ao processo de desenvolvimento, baseados em RFCs relevantes ao protocolo HTTP. Esses padrões abordam métodos de requisição, códigos de status, cabeçalhos de requisição, entre outros aspectos. A adoção de boas práticas para APIs REST promove consistência e qualidade na comunicação entre os serviços, além de facilitar a criação de novas funcionalidades (FIELDING, 2022).

Segundo a RFC 1738 (1994), as URLs devem ser descritivas, semânticas e de fácil entendimento, utilizando preferencialmente substantivos em vez de verbos. Além disso, a URL deve identificar claramente o recurso associado. Por exemplo, para uma funcionalidade que recupera dados de uma carteira de investimentos, a URL poderia ser estruturada como mostrado na Figura 16.

Figura 16 - Proposta de URL de Serviço



Fonte: Autor

Na Figura 16, o recurso "wallets" é utilizado como rota de serviço para a aplicação de gestão de carteiras. O identificador de recurso "portfolio" representa funcionalidades vinculadas à gestão de portfólios de investimentos, e o parâmetro "idPortfolio" faz referência ao identificador do portfólio sendo acessado.

Em relação à semântica de conteúdos, a RFC-7231 (2014), de Fielding e Reschke, define padrões para métodos HTTP, os quais possuem objetivos distintos no sistema e, consequentemente, funcionalidades diferentes. Abaixo, estão descritos alguns dos métodos definidos na RFC-7231.

Tabela 5 - Métodos suportados RFC-7231

Verbo	**Descrição**
GET	Recupera um recurso
POST	Envia dados ao servidor
PUT	Armazena dados no servidor
PATCH	Modifica dados no servidor
DELETE	Apaga dados do servidor
OPTIONS	Obtém os métodos HTTP suportados pelo servidor

Fonte: AUTOR, dados baseados em FIELDING, R.; RESCHKE, J. RFC 7231: Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. 2014.

Avaliando o serviço de listagem de portfolios, a funcionalidade de recuperar uma carteira de investimentos seria tratado com base em um método “GET”. As figuras 17 e 18 exemplificam, respectivamente, um arquivo de rotas que contempla os métodos vinculados a serviços de portfólio e parte do código vinculado ao endpoint de listagem de portfolios.

Figura 17 – Arquivo de rotas no serviço de carteiras

```
6   api_router = APIRouter()
7
8   api_router.include_router(
9       healthcheck.router,
10      prefix="/healthcheck",
11      tags=["Healthcheck"]
12  )
13
14  api_router.include_router(
15      portfolio.router,
16      prefix="/portfolio",
17      tags=["Portfolio"]
18  )
```

Fonte: Autor

Figura 18 – Caso de uso método de listagem de dados do portfolio

```
4   router = APIRouter()
5
6   @router.get("/{idPortfolio}")
7   async def wallet(idPortfolio: str, response: Response):
8       try:
9           portfolio_data = business.get_portfolio(idPortfolio)
10
11          if not portfolio_data:
12              # Retorna 404 se o portfólio não for encontrado
13              status_code = status.HTTP_404_NOT_FOUND
14              message = "Portfolio not found"
15              raise HTTPException(status_code=status_code, detail={"status": "Error", "message": message})
16
17          # Sucesso explícito com status 200
18          response.status_code = status.HTTP_200_OK
19          return {"status": "Success", "data": portfolio_data}
20      except Exception as e:
21          # Erro interno com mensagem dinâmica
22          status_code = status.HTTP_500_INTERNAL_SERVER_ERROR
23          message = str(e)
24          raise HTTPException(status_code=status_code, detail={"status": "Error", "message": message})
```

Fonte: Autor

Ainda sobre o modelo de respostas e tratativa da API, importante situar que os padrões de autenticação foram validados no middleware da aplicação garantindo que todo o código vinculado a autenticação fique em um único lugar bem como alguns dos demais status de resposta, como por exemplo, o status 401.

Por fim, embora tenhamos definido um padrão de consultas à API e apresentado um modelo exemplificativo, o conceito de modelar APIs vai além disso, trata-se de um hábito fundamental de programação. Para que o software cresça de forma orgânica e sustentável é crucial que todos os desenvolvimentos, tanto passados quanto futuros, sejam adaptados e ajustados conforme os padrões estabelecidos. Isso garante consistência, escalabilidade e uma base sólida para evoluções contínuas.

3.3.3 Adaptação para execução na nuvem

A transição para uma arquitetura baseada em microsserviços permite a separação das bibliotecas conforme os módulos que possuem dependências específicas, reduzindo a importação de códigos de terceiros na aplicação. Esse benefício imediato da segregação de microsserviços impacta diretamente nos containers, tornando-os mais leves e adequados às necessidades de cada serviço. Inicialmente, possuíamos uma versão de container funcional para desenvolvimento local, porém a aplicação não tinha um arquivo Dockerfile voltado para o ambiente produtivo. A seguir, o código adaptado para o ambiente produtivo.

Figura 19 – Proposta de container produtivo

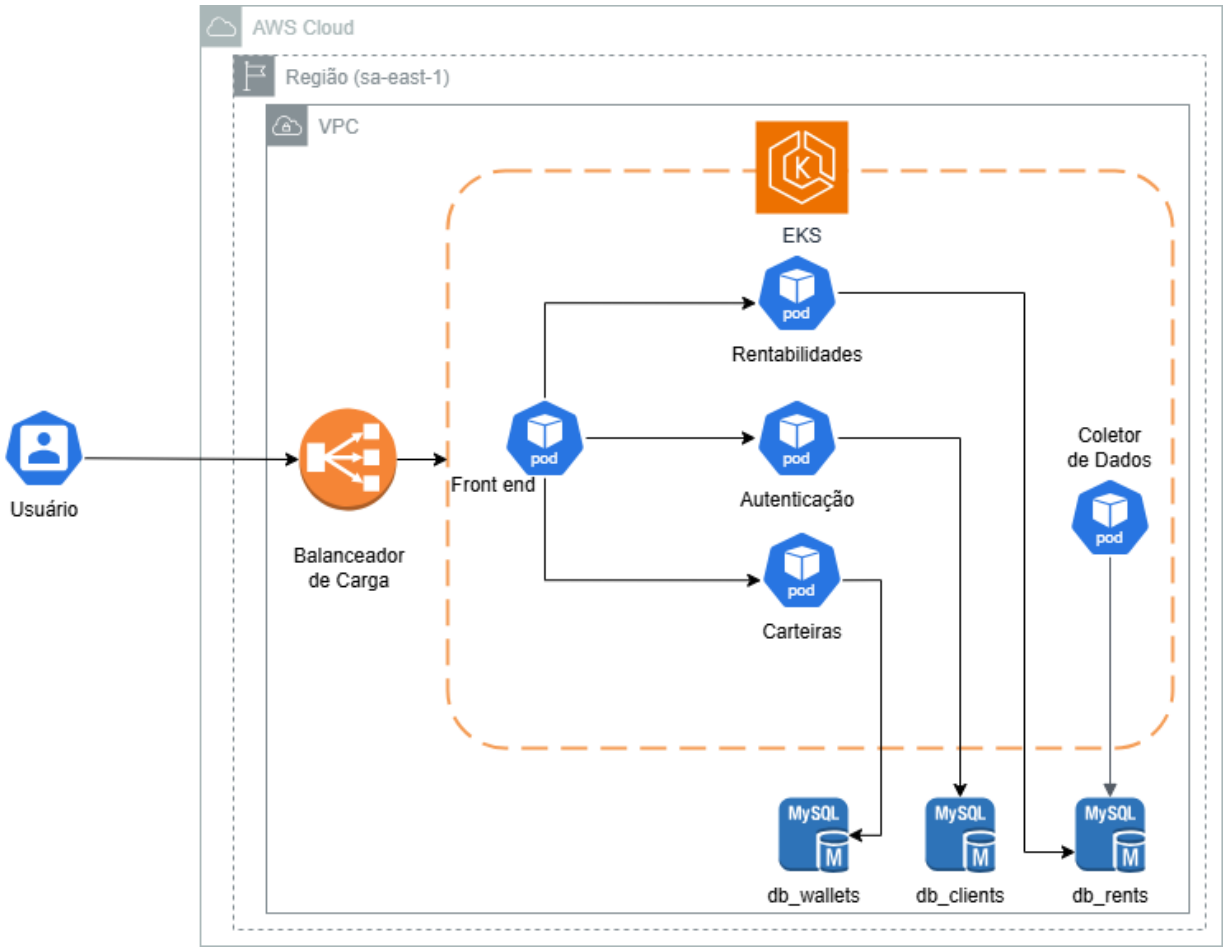
Dockerfile > ...

```
1  FROM python:3.10.6-buster
2  WORKDIR /app
3
4  # Copia o arquivo de dependências para o contêiner
5  COPY ./requirements.txt /app/requirements.txt
6
7  # Atualiza os pacotes do sistema e instala dependências
8  RUN apt-get update && \
9      apt-get install -y --no-install-recommends gcc && \
10     pip install --upgrade pip && \
11     pip install --no-cache-dir -r requirements.txt && \
12     apt-get clean && rm -rf /var/lib/apt/lists/*
13
14 # Copia o restante dos arquivos
15 COPY . /app
16
17 # Porta da aplicação
18 EXPOSE 5000
19
20 # Iniciar a aplicação
21 CMD ["python", "app.py"]
```

Fonte: Autor

Orquestradores de contêineres são ferramentas que apoiam no gerenciamento do ciclo de vida dos contêineres em ambientes de microsserviços, oferecendo funcionalidades como escalonamento, balanceamento de carga, e monitoramento de contêineres (PAI; KUMAR, 2021). Plataformas gerenciadoras garantem que cada serviço, encapsulado em contêineres independentes, funcione de forma integrada. Entre os orquestradores disponíveis, destaca-se o Amazon Elastic Kubernetes Service (EKS), que fornece uma solução robusta para a orquestração de contêineres na AWS, facilitando a integração com outros serviços da nuvem e aumentando a eficiência operacional de aplicações nativas em nuvem (AWS, 2024). Na Figura 20, arquitetura sugerida adaptada para execução em nuvem utilizando componentes da AWS.

Figura 20 – Proposta de arquitetura em nuvem



Fonte: Autor

Na Figura 20, temos uma arquitetura baseada na infraestrutura da Amazon Web Services (AWS) com implementação modular e escalável. A solução fica hospedada

na região de São Paulo, identificada pelo código sa-east, por ser a região mais próxima do público-alvo da aplicação (AWS, 2024).

A Virtual Private Cloud (VPC) garante o isolamento do ambiente e segurança de rede (AWS, 2024). Além disso, adicionamos um balanceador de carga responsável por gerenciar o tráfego de usuários, distribuindo requisições para diferentes pods de um cluster gerenciado pelo Amazon Elastic Kubernetes Service (EKS). Por fim, a decisão de ter instâncias diferentes de bancos de dados tem como objetivo garantir a independência entre as peças.

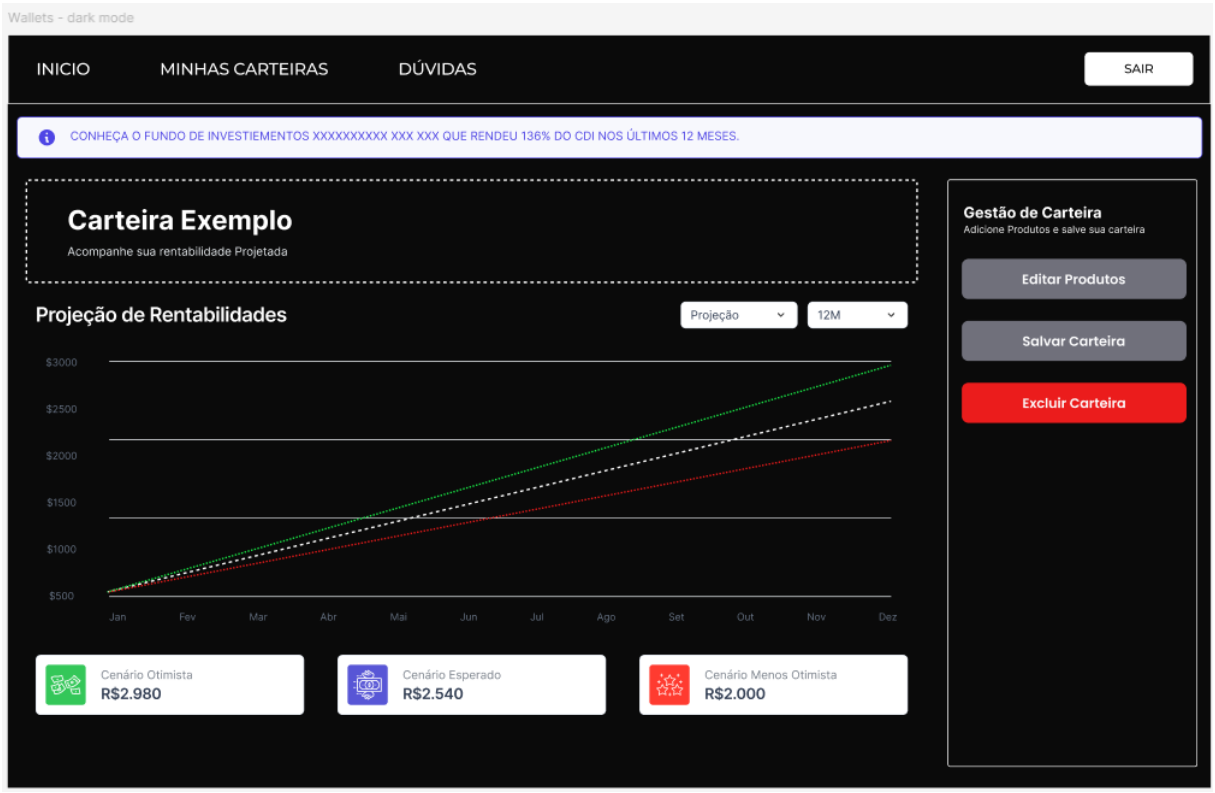
3.3.4 Proposta de front-end desacoplado

Com base no princípio de desacoplamento, abaixo algumas propostas de tela para a camada de usuário que podem ser desenvolvidas utilizando a linguagem mais adequada para o time de front-end. O desacoplamento garante a flexibilidade para que os desenvolvedores escolham as tecnologias que melhor atendem às suas necessidades, garantindo maior autonomia e adaptabilidade, sem comprometer a integração e funcionalidade do sistema como um todo

Figura 21 – Tela de Autenticação

Fonte: Autor

Figura 22 – Tela de Consulta de Rentabilidades



Fonte: Autor

Na Figura 21, apresenta-se uma tela de login proposta para o sistema, a qual segue os requisitos definidos na Tabela 4 – Requisitos de Software Organizados por Contexto Delimitado. Esta tela abrange o fluxo de login e oferece opções para cadastro de novos usuários e recuperação de senhas.

A Figura 22 ilustra a tela proposta para a apresentação de rentabilidades. Nessa tela, é possível aplicar filtros de período, tanto para projeções quanto para histórico, e filtros de prazo, com opções de 12, 24 e 36 meses. O gráfico exibe as rentabilidades, com intervalos de confiança para o cenário de projeção, denominados “Cenário Otimista” e “Cenário Menos Otimista”, além do “Cenário Esperado”. No painel lateral, o usuário pode acessar a ferramenta de edição de carteira, bem como salvar ou excluir a carteira criada.

3.4 Considerações do Capítulo

Esse Capítulo apresentou uma aplicação financeira e uma proposta de adaptação para um modelo componível.

Primeiramente foi apresentado o software em seus contextos de negócio e tecnologia, sendo apresentados, o domínio da aplicação, requisitos de software, arquitetura monolítica atual e as ferramentas de desenvolvimento utilizadas atualmente na aplicação. Posteriormente, com base na arquitetura atual e no contexto de aplicação, apresentamos propostas de adaptações para um modelo componível tendo incorporado arquitetura de microsserviços, API primeiro, adaptação para nuvem e com uma camada de apresentação desacoplada.

4. CONSIDERAÇÕES FINAIS

4.1 Conclusões

A adoção de uma arquitetura componível oferece uma série de benefícios às aplicações, como escalabilidade, reutilização de componentes e maior organização do código. No entanto, a implementação dessa abordagem exige o compromisso com boas práticas de engenharia e um planejamento criterioso para aplicar adequadamente os princípios, nessa monografia utilizamos o modelo MACH: Microsserviços, API First (API Primeiro), Cloud-Native (Nativo em Nuvem) e Headless (Desacoplado).

A aplicação dos conceitos associados a arquiteturas componíveis permite separarmos as responsabilidades dos serviços de um software, promovendo aplicações mais resilientes e reutilizáveis. Entre os fatores-chave para alcançar essas vantagens está o uso de microsserviços. Esses componentes, definidos por contextos bem delimitados, integram-se por meio de APIs.

Um aspecto central do modelo componível é o desacoplamento entre as camadas de apresentação e de negócios, viabilizado pelas comunicações mediadas por APIs. Esse desacoplamento, aliado à abordagem API First, reduz a interdependência entre as equipes responsáveis por diferentes camadas da aplicação, além de permitir o uso de tecnologias específicas e adequadas às demandas de cada contexto.

A adesão ao modelo MACH no contexto de uma arquitetura componível é potencializada pela adaptação do software para ambientes em nuvem. Essa transição oferece benefícios de elasticidade e escalabilidade, sendo esses benefícios sustentados pela modularidade proporcionada pelos microsserviços e na independência entre as camadas de software.

Por fim, o processo de transformação de uma aplicação monolítica para o modelo componível descrito nesta monografia evidencia a sinergia entre os princípios de microsserviços, API First, arquitetura baseada em nuvem e desacoplamento entre camadas. Essa abordagem viabiliza avanços significativos, como maior

escalabilidade, reutilização de módulos, desenvolvimento independente e maior resiliência do sistema como um todo.

4.2 Contribuições do Trabalho

Esse trabalho contribui para o campo de Engenharia de Software ao propor um arcabouço teórico e prático para aplicação de uma arquitetura componível, estando fundamento em microsserviços, desenvolvimento de APIs, software nativo em nuvem e desacoplamento

No âmbito acadêmico, a monografia amplia o entendimento sobre a aplicabilidade do modelo MACH, proposto pela MACH Alliance (2022), consolidando conceitos dispersos na literatura e estruturando-os de maneira acessível e lógica. O trabalho também apresenta uma fundamentação teórica abrangente que conecta os princípios da arquitetura componível a práticas consolidadas em Engenharia de Software, oferecendo uma base teórica para futuros estudos na área.

Adicionalmente, ao abordar a interrelação entre desacoplamento, modularidade, escalabilidade e comunicação entre serviços, o trabalho contribui para a discussão sobre o desenvolvimento de sistemas resilientes e adaptáveis. Essa análise se mostra relevante para o estudo de metodologias que suportam a evolução de sistemas legados em ambientes complexos e dinâmicos, como o mercado financeiro.

No âmbito técnico e prático monografia destaca um guia de implementação do MACH em um sistema existente, destacando práticas como o planejamento de APIs e decomposição em microsserviços.

As contribuições deste trabalho reforçam a importância da adoção de arquiteturas componíveis no cenário atual, onde a adaptabilidade e a escalabilidade são requisitos críticos para o sucesso de sistemas de software. Ao integrar teoria e prática, o trabalho oferece uma visão que pode servir como referência para trabalhos futuros.

4.3 Trabalhos Futuros

Os conceitos presentes nesse trabalho não se esgotam nessa pesquisa, haja visto a complexidade presente em uma arquitetura componível. Como trabalhos futuros, essa monografia pode suscitar os seguintes temas:

- Aplicação do modelo componível em diferentes contextos de negócios.
- Aplicação de métricas quantitativas e qualitativas para avaliar o sucesso de uma aplicação componível com o objetivo de mensurar a qualidade do modelo e seus benefícios.
- Adaptação de times e metodologias de desenvolvimento para um contexto de arquitetura componível.
- Aplicação de tecnologias emergentes como inteligência artificial para automação da adaptação de sistemas monolíticos para um modelo componível.

REFERÊNCIAS

AMAZON WEB SERVICES**. Amazon Elastic Kubernetes Service (Amazon EKS)**. Disponível em: https://docs.aws.amazon.com/eks/. Acesso em: 20 nov. 2024.

BANCO CENTRAL DO BRASIL. **Boletim Focus**. Brasília: BCB, 2023. Disponível em: https://www.bcb.gov.br. Acesso em: 28 out. 2024.

BASS, Len. **Software Architecture in Practice**. 3. ed. Boston: Addison-Wesley, 2015.

BRIGHAM, E. F.; EHRHARDT, M. C. **Financial Management: Theory & Practice**. 15. ed. Stamford: Cengage Learning, 2017.

BRUCE, K.; PEREIRA, M. **Arquitetura de Microsserviços: Princípios e Práticas**. São Paulo: Editora Tech, 2018.

CRNKOVIC, I.; LARSSON, M**. Building Reliable Component-Based Software Systems**, Artech, 2002.

DAVIS, J. **Cloud Native Patterns: Designing Change-Tolerant Software**. Manning Publications, 2019.

DRAGONI, N. et al. **Microservices: Yesterday, Today, and Tomorrow. In: Present and Ulterior Software Engineering**. Springer, 2017. p. 195–216.

DOCKER HUB. **Python 3.10.6 on Debian Buster**. Disponível em: https://hub.docker.com/layers/library/python/3.10.6-buster/images/sha256-25cfebf4cb288eefaa86ee886fb82d26a72fadea64b8c5a9287de2fb1c6ead72. Acesso em: 4 nov. 2024.

FIELDING, Roy T.; RESCHKE, Julian F. **HTTP/1.1: Semantics and Content**. RFC 9110 (obsoletes RFC 7231), Internet Engineering Task Force, [s.l.], jun. 2022. Disponível em: https://www.rfc-editor.org/info/rfc9110. Acesso em: 16 nov. 2024.

FRITZSCHE, R. **Bounded Contexts: Behavior Over Data Structures**. Disponível em: https://ricofritzsche.me/ddd-modularization-concepts-aggregates-part-ii/. Acessado em: 20 nov. 2024.

FIELDING, Roy T.; RESCHKE, Julian F. **HTTP/1.1: Semantics and Content**. RFC 9110 (obsoletes RFC 7231), Internet Engineering Task Force, [s.l.], jun. 2022. Disponível em: https://www.rfc-editor.org/info/rfc9110. Acesso em: 16 nov. 2024.

GARCÍA, D.; MARTÍNEZ, J.; LOPEZ, F. A.; RUIZ, P. **Composable Microservices Architecture: Strategies for Reusability and Flexibility**. Journal of Cloud Computing, v. 9, n. 3, p. 35-49, 2021.

GARTNER. **Predicts 2023: Composable Applications Accelerate Business Innovation**. Disponível em: https://www.gartner.com/en/doc/predicts-2023-composable-applications-accelerate-business-innovation. Acesso em: 21 set. 2024.

GARTNER. **Critical Capabilities for Cloud Infrastructure and Platform Services.** 2020.

GEEWAX, JJ. **API Design Patterns**. 1. ed. Sebastopol, CA: O'Reilly Media, 2022.

GROSSER, D.; SAHRAOUI, H. A.; VALCHEV, P. **Predicting Software Stability Using Case-Based Reasoning.** In: 17th IEEE International Conference on Automated Software Engineering, January, 2002. Proceedings [...]. DOI: 10.1109/ASE.2002.1115033. Disponível em: https://ieeexplore.ieee.org/document/1115033 . Acesso em: 21 de setembro de 2024.

GUNTHER, S. M**. Investing in Financial Markets**. New York: McGraw-Hill, 1980.

HYPERTEXT TRANSFER PROTOCOL (HTTP/1.1): Semantics and Content. RFC 7231. Disponível em: https://www.rfc-editor.org/info/rfc7231. Acesso em: 5 dez. 2024

JACOBSON, D.; BRAIL, G.; WOODS, D. **APIs: A Strategy Guide**. O'Reilly Media, 2011.

JIN, B; SAHNI, S; SHEVAT, A. **Designing Web APIs**. Sebastopol: O'Reilly Media, 2019.

LARRUCEA, X; SANTAMARRIA, I. R. C; EBERT, C. **Microservices**. IEEE Software vol. 35, no. 3, pp. 96-100, May/June 2018, doi: 10.1109/MS.2018.2141030

LÄNDIN, L. **A Pragmatic Approach to Composable**. Occtoo, 2023.

LEWIS, J.; FOWLER, M. **Microservices: A Definition of This New Architecture Term**. Mar. 2014.

MACH ALLIANCE. **Making the Case for MACH**. Disponível em: https://machalliance.org/making-the-case-for-mach. Acesso em: 1 dez. 2024.

MAHDY, A.; FAYAD, M. E. **A Software Stability Model Pattern**. In: 9th Conference on Pattern Language of Programs. Proceedings [...]. Agosto, 2002. p. 1-13. Disponível em: https://www.researchgate.net/publication/2559402_A_Software_Stability_Model_Pattern. Acesso em: 21 set. 2024.

MARKOWITZ, H. **Portfolio Selection**. Journal of Finance, v. 7, n. 1, p. 77–91, 1952.

MAYS, G. **Composable Architecture: What You Need to Know**. Set. 2023.

MEIRELLES, F. **Pesquisa de Uso da TI – Tecnologia de Informação nas Empresas**. Disponível em: https://eaesp.fgv.br/producao-intelectual/pesquisa-anual-uso-ti. Acessado em: 21 set. 2024, v. 35, Edição Anual, FGVcia 2024.

NASSER, A. A. M. **Competição e Concentração no Setor Bancário Brasileiro Atual: Estrutura e Evolução ao Longo do Tempo**. 3º Prêmio SEAE, 2008.

NEWMAN, S. **Building Microservices: Designing Fine-Grained Systems**. 1. ed. O'Reilly Media, 2015.

OZKAYA, M. **Headless architecture with separated UI for backend and frontend**. Publicado em Design Microservices Architecture with Patterns & Principles, 2023.

PAI, P; KUMAR, C. **Building cloud native application – analysis for multi-component application deployment**. 2021 International Conference on Computer Communication and Informatics (ICCCI-2021), 27-29 jan. 2021, Coimbatore, Índia.

PRESSMAN, R.; MAXIM, B. **Engenharia de Software - 8ª Edição**. [S.l.]: McGraw-Hill Brasil, 2016.

RICHARDS, M.; FORD, N. **Fundamentals of Software Architecture: An Engineering Approach**. O'Reilly Media, 2020.

RICHARDSON, C. **Microservices Patterns: With examples in Java**. Shelter Island: Manning Publications, 2018

ROSS, S. A., WESTERFIELD, R. W., & JAFFE, J. **Corporate Finance**. 8. ed. New York: McGraw-Hill Education, 2007.

SHARPE, W. F.; GORDON, J. M.; BLACK, F. **Investments**. 7. ed. Boston: Prentice Hall, 2014.

SOMMERVILLE, I. **Engenharia de Software**. 9. ed. São Paulo: Pearson Prentice Hall, 2011.

SEVERINO, A, J. **Metodologia do Trabalho Científico**. 23. ed. São Paulo: Cortez, 2007.

UNIFORM RESOURCE LOCATORS (URL). **RFC 1738**. Disponível em: https://www.rfc-editor.org/info/rfc1738. Acesso em: 5 dez. 2024.

KIMBALL, Ralph; ROSS, Margy. **The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling**. 3ª edição. Wiley, 2013.

WILLIAMS, Christopher L. et al. **The Growing Need for Microservices in Bioinformatics**. Journal of Pathology Informatics, v. 7, n. 1, p. 45, 2016.