UNIVERSIDADE FEDERAL DO CEARÁ

CENTRO DE TECNOLOGIA

DEPARTAMENTO DE ENGENHARIA DE TELEINFORMÁTICA

BRENO DE SOUZA ALMEIDA BARROSO

EXPLORAÇÃO DAS ETAPAS DE INTEGRAÇÃO DE SISTEMAS HETEROGÊNEOS EM PROJETOS DE SOFTWARE

FORTALEZA

2023

BRENO DE SOUZA ALMEIDA BARROSO

EXPLORAÇÃO DAS ETAPAS DE INTEGRAÇÃO DE SISTEMAS HETEROGÊNEOS EM PROJETOS DE SOFTWARE

Monografia apresentada ao Curso de Graduação em Engenharia de Computação do Centro de Tecnologia da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Computação.

Orientador: Prof. Dr. José Marques Soares.

FORTALEZA

2023

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

B285e Barroso, Breno de Souza Almeida.
Exploração das etapas de integração de sistemas heterogêneos em projetos de software / Breno de Souza Almeida Barroso. – 2023.
75 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Centro de Tecnologia, Curso de Engenharia de Computação, Fortaleza, 2023.
Orientação: Prof. Dr. José Marques Soares.

1. Integração de sistemas. 2. Arquitetura de software. 3. Projetos de software. I. Título.
CDD 621.39

BRENO DE SOUZA ALMEIDA BARROSO

EXPLORAÇÃO DAS ETAPAS DE INTEGRAÇÃO DE SISTEMAS HETEROGÊNEOS EM PROJETOS DE SOFTWARE

Monografia apresentada ao Curso de Graduação em Engenharia de Computação do Centro de Tecnologia da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Computação.

Aprovada em: / / .

BANCA EXAMINADORA

_____

Prof. Dr. José Marques Soares (Orientador)
Universidade Federal do Ceará (UFC)

_____

Prof. Dr. Giovanni Cordeiro Barroso (DEE/UFC)
Universidade Federal do Ceará (UFC)

_____

Me. Luís César Marques de Vasconcelos (Secretário de Governança/UFC)
Universidade Federal do Ceará (UFC)

A Deus.

Aos meus pais, Marcelo e Ana Paula.

AGRADECIMENTOS

Primeiramente, a Deus, pelo dom da minha vida e da minha família, pelos talentos que de graça me foram ofertados e pelas infinitas graças em minha vida.

Aos meus pais, Marcelo e Ana Paula, pelo apoio incondicional em todas as etapas de minha vida, principalmente pela educação e direcionamento desde a infância, que se perpetua até hoje. Muito obrigado.

À minha namorada, Larissa Sampaio, que esteve ao meu lado, oferecendo todo o suporte, especialmente o emocional, para que eu fosse capaz de suportar as adversidades enfrentadas na formação acadêmica, na formação pessoal e também profissional. A ela, quero deixar a minha imensa gratidão e recíproco apoio pelo resto da minha vida.

Aos professores que me acompanharam durante a trajetória, com o conhecimento passado, lições apresentadas e empatia ofertada de bom grado, especialmente aos professores Dr. Nicolas Araujo, Dr. Helano Castro, Dr. Michela Mulas e, em especial, ao Dr. José Marques por me orientar em meu trabalho de conclusão de curso.

Aos meus colegas, Guilherme, Maycon e Rodrigo, que transformaram, na medida do possível, a formação em engenharia em um caminho mais leve, com aprendizados tão ricos e, principalmente, amizades para a vida.

À UFC, por me proporcionar ricas experiências de aprendizado e resiliência.

“Não há lugar para a sabedoria onde não há paciência.” (Santo Agostinho).

RESUMO

A acelerada evolução dos sistemas computacionais e tecnologias demanda uma abordagem eficiente para a integração de sistemas heterogêneos, onde aplicativos interagem para atender às necessidades dos usuários. Este trabalho aborda os fatores impactantes em projetos de software nesse contexto, destacando a análise do gerenciamento de dados, padrões de projeto e arquitetura de software. Além disso, explora as principais etapas do ciclo de vida da integração, propondo um guia para projetos de integração, em complemento com a sua análise aplicada em uma experiência de projeto no mercado de software. A experimentação mostrou-se eficiente na aplicação das etapas propostas, adicionando algumas complexidades típicas da implementação da fundamentação teórica em um ambiente não controlado. Isso pode servir como aprendizados relevantes ao avaliar o grande volume de dados, a inexperiência da equipe e a concorrência enfrentada em um ecossistema diverso de sistemas. O trabalho conclui que a análise prévia, a colaboração entre os times responsáveis pelos sistemas envolvidos e a avaliação crítica das etapas propostas são fatores contributivos para o desenvolvimento de um projeto de integração de sistemas com menos incertezas e, portanto, mais próximo do objetivo esperado.

Palavras-chave: integração; sistemas heterogêneos; ciclo de vida; gerenciamento de dados; arquitetura de software.

ABSTRACT

The quickened evolution of computer systems and technologies demands an efficient approach to the integration of heterogeneous systems, where applications interact to meet the users' needs. This work addresses the impacting factors in software projects in this context, highlighting the analysis of data management, design patterns, and software architecture. Additionally, it explores the key stages of the integration life cycle, proposing a guide for integration projects, complemented by its application analysis in a software market project experience. The experimentation proved effective in implementing the proposed stages, adding some typical complexities of implementing the theoretical foundation in an uncontrolled environment. This can serve as relevant insights when evaluating the large volume of data, team inexperience, and competition faced in a diverse ecosystem of systems. The work concludes that preliminary analysis, collaboration among the teams responsible for the involved systems, and critical evaluation of the proposed stages are contributing factors to the development of a system integration project with fewer uncertainties and, therefore, closer to the expected goal.

Keywords: integration; heterogeneous systems; life cycle; data management; software architecture.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

SQL	Structure Query Language
SGBD	Sistema de Gerenciamento de Banco de Dados
JSON	JavaScript Object Notation
API	Application Programming Interface
REST	Representational State Transfer
SOAP	Simple Object Access Protocol
gRPC	Google Remote Procedure Call
WSDL	Web Services Description Language
DDD	Domain Driven Design
HTTP	Hypertext Transfer Protocol

SUMÁRIO

1 INTRODUÇÃO

Nas primeiras décadas do séc. XXI, as pessoas são sobrecarregadas com informações que se propagam em alta velocidade, direcionadas às suas necessidades individuais e integradas ao ecossistema da internet. Isto só é possível devido a uma população conectada à rede de forma contínua e quase em período integral. Para atender aos desafios da rápida e frequente evolução dos sistemas computacionais e de suas tecnologias, é preciso administrar um contexto diverso de aplicativos em que recursos integrados à infraestrutura da web se articulam para atender às necessidades dos usuários.

Neste modelo de integração, o indivíduo possui, por exemplo, um aplicativo para agendamento de reuniões, outro para gestão de cronogramas, um terceiro para marcação de consultas para os filhos e um último para tarefas de trabalho. Todos esses sistemas estão conectados dentro de uma grande estrutura dominante no mundo ocidental no início da década de 2020: o Google. Ferramentas desta plataforma, por exemplo, possibilitam visualizar todas as atividades e dados dos usuários na agenda do Google acessíveis via web em seus computadores pessoais ou no calendário de aplicativos que executam em seus smartphones. Com frequência, os diferentes dispositivos estão em execução em aplicações e sistemas operacionais distintos. Para que isto seja possível, é preciso uma integração entre diversos sistemas que permita a troca de informações entre eles, possibilitando, para este caso, a sensação de centralização das informações que, por sua vez, contribui para a facilitação da dinâmica desse indivíduo na rede.

Pensando no cenário corporativo, lidamos a todo momento com troca de informações entre os sistemas da companhia e até mesmo com os sistemas de parceiros comerciais. Integrar aplicações empresariais é fazer com que diferentes aplicações que não foram concebidas tendo em mente sua integração, possam colaborar e dar suporte a processos de negócio (HOHPE; WOOLF, 2003).

Peguemos como exemplo o processo de gestão de compras. Para o funcionamento da empresa, é necessária a utilização de produtos como móveis, papel, caneta, produtos de limpeza, bobina e etc, chamados de produtos de uso e consumo. Por isso, é preciso abrir um processo de compra em um fornecedor, seguindo as etapas de: confirmação do pedido de compra, emissão da nota fiscal, faturamento da nota, atualização do *status* em sistemas *Enterprise Resource Planning* (ERP) da companhia, atualização do *status* do pedido até a entrega do material. Portanto, para cada etapa desse processo, é necessária a integração desses sistemas, permitindo a automatização desse fluxo de compra. Esses exemplos mostram que a

demanda atual exige que nos adaptemos em relação à transmissão de dados, velocidade de resposta, qualidade das informações (dados que refletem a última amostragem possível), entre outros aspectos. A integração de sistemas é a espinha dorsal da arquitetura empresarial moderna. É a capacidade de mover dados e informações de maneira eficiente que impulsiona o sucesso dos negócios (Rosen 2008).

Para compreender a integração de sistemas heterogêneos, é necessário discutir alguns dos fatores de impacto para um projeto de software nessa perspectiva. Para tanto, torna-se essencial a análise das estruturas de gerenciamento de dados a serem utilizadas, a escolha da arquitetura, ou seja, como as aplicações interagem entre si dentro do projeto de integração, além de avaliar o impacto dessas escolhas considerando alguns fatores cruciais, como escalabilidade, desempenho, segurança, entre outros. Para isso, se faz necessário avaliar o impacto de tais tecnologias a fim de fornecer aos envolvidos no projeto informações sobre o comportamento delas e quais são melhores e mais adequadas de acordo com as características da aplicação a ser desenvolvida (JAIN, 1991).

Portanto, este trabalho justifica-se do ponto de vista acadêmico devido à importância que a troca constante de dados entre sistemas impacta o cotidiano nas relações interpessoais e em todo o ecossistema corporativo. Neste sentido, a integração que ocorre nesse arcabouço tecnológico permite a interação entre sistemas heterogêneos, que, por sua vez, possibilita uma maior facilidade de acesso aos dados, descentralização das informações e automatização das tarefas e processos. É fundamental analisar os impactos das novas tecnologias de mercado que surgem para complementar um mesmo produto de software, além de articular sistemas independentes para um mesmo propósito de entrega de valor. Por fim, é importante avaliar o arcabouço tecnológico e de infraestrutura que serve de suporte para esse conjunto de integrações.

1.1 Objetivos

Pretende-se, com este trabalho, desenvolver uma análise da interação entre sistemas de informação heterogêneos e o conjunto de tecnologias que os compõem, trazendo exemplos das diversas facetas da integração de tecnologias, incluindo gestão, arquitetura, modelagem, interfaces, padrões de desenvolvimento e infraestruturas diversas. O propósito é utilizar a experiência no mercado de desenvolvimento e análise de software, juntamente com o conhecimento e as valências adquiridas ao longo do curso de engenharia.

Serão analisados, também, os aspectos relacionados à eficácia na troca de informações entre os sistemas, à interoperabilidade entre tecnologias heterogêneas, à facilidade de manutenção e evolução do sistema integrado, bem como à viabilidade de implementação em ambientes reais.

Ao final deste trabalho, será apresentado um exemplo concreto de integração de tecnologias, abordando aspectos discutidos na fundamentação teórica. A avaliação do projeto desenvolvido considerará critérios como escalabilidade, desempenho, segurança, modularidade e reutilização de componentes. Além disso, será feita uma análise detalhada sobre as etapas que compõem o ciclo de vida de um projeto de integração. A análise visa fornecer insights valiosos para a área de engenharia de computação e contribuir para a compreensão e aprimoramento da integração de sistemas heterogêneos.

1.2 Estrutura do trabalho

O trabalho está dividido em 5 capítulos. No capítulo 2 é apresentada a fundamentação teórica, referenciando os principais autores que abordam os conceitos de integração de sistemas sob a perspectiva da análise das abordagens utilizadas no mercado. Em especial, discute-se sobre a arquitetura de software e as frentes que orientam a tomada de decisão em um projeto de software.

No Capítulo 3 são apresentadas as etapas que compõem o ciclo de vida em um projeto de integração de sistemas. Exploraremos como as decisões tomadas nas fases iniciais, desde a concepção até o design, têm impacto direto na eficácia da operação do sistema integrado. Além disso, examinaremos os processos de desenvolvimento e implementação, destacando estratégias eficazes para lidar com a diversidade tecnológica presente em ambientes heterogêneos. Compreenderemos como as fases que sucedem a entrega em produção, como monitoramento contínuo e manutenção, desempenham um papel crucial na garantia da longevidade e no desempenho otimizado do sistema integrado.

No Capítulo 4 é descrito um projeto prático, sob a ótica dos fundamentos discutidos nos capítulos 2 e 3. Com o intuito de aprimorar a compreensão dos leitores sobre o ciclo de vida, busca elucidar não apenas as nuances técnicas, mas também as considerações práticas e os desafios enfrentados no mundo real. Por fim, no Capítulo 5 são feitas as considerações finais acerca do trabalho desenvolvido.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Definição

Podemos definir integração entre sistemas heterogêneos como a troca de informações entre dois sistemas independentes que se comunicam devido a uma necessidade de negócio. Ou seja, quando há a necessidade de que a informação, que está sob o domínio do sistema A, chegue até o sistema B para que seja executado algum processamento ou tomada de decisão. É necessário que ocorra uma comunicação entre as partes para que esses dados cheguem até o destino esperado. Portanto, o desafio consiste em criar um modelo de integração que abstraia o sistema que irá integrar a solução, podendo ter sido desenvolvido em qualquer linguagem, sendo importante atender aos critérios estabelecidos por uma interface, como por exemplo uma *Application Programming Interface* (API), para que haja a comunicação entre os sistemas de forma consistente.

A integração entre sistemas heterogêneos é definida por diversos autores consagrados no meio tecnológico. Pressman & Maxim (2021) a definem como “o processo de conectar diferentes sistemas e software de forma que eles funcionem como um único sistema coordenado”. Fowler (2002), especialista em design de software, a define como “o ato de levar informações de um sistema a outro e torná-los operar como um sistema único”. Enquanto a Red Hat a conceitua como “a capacidade de conectar sistemas e aplicações heterogêneas para que possam funcionar juntos de maneira eficaz”. Essas definições destacam a importância da integração de sistemas como um processo que visa unir diferentes sistemas, aplicativos e tecnologias para operar de forma coordenada, proporcionando maior eficiência e capacidade de atender às necessidades de negócios de uma organização.

Essa integração pode ocorrer em diferentes níveis, desde a integração de aplicativos individuais até a integração de sistemas em toda a empresa. Estamos imersos em um ambiente cada vez mais interconectado, e no mundo corporativo, não seria diferente. Vamos tomar como exemplo o ERP (*Enterprise Resource Planning*), que desempenha um papel fundamental na gestão eficiente dos recursos e processos de uma organização. Ele funciona como um sistema centralizado que integra várias funções e departamentos, permitindo que as informações fluam de maneira mais eficaz entre os diferentes sistemas. De maneira geral, em um contexto corporativo, a integração de sistemas é a união de diversos sistemas, ou apenas um, e softwares que a empresa já usa, seja para gestão de estoque,

pagamentos, atendimento ao cliente, etc, de forma automatizada, centralizando dados que irão circular de maneira mais rápida e eficiente entre os setores.

2.2 Análise de Dados

Na perspectiva de analisar as partes que compõem um projeto de integração de sistemas de informação, os dados representam um valor central. Segundo Elmasri e Navathe (2011), “os bancos de dados e os sistemas de bancos de dados se tornaram componentes essenciais no cotidiano da sociedade moderna. No decorrer do dia, a maioria de nós se depara com atividades que envolvem alguma interação com os bancos de dados”.

Por isso, devemos entender o contexto atual sobre o armazenamento, processamento e integração desses dados, que, hoje, recebem protagonismo não só no ecossistema de tecnologia da informação, mas também no domínio do mercado de capital. Segundo LOPES (2008), "a produção de riqueza passa a depender do conhecimento e informações produzidos". Dessa forma, com o grande volume de dados produzidos de maneira exponencial na contemporaneidade, cabe à integração de sistemas empregar técnicas que suportem as atuais necessidades e desafios.

A integração corporativa deve tornar possível aos gerentes monitorarem a operação de negócios, identificarem problemas e oportunidades, definirem alterações e melhorias desejadas e avaliarem resultados. Para acessar informações de nível gerencial, que incluem vários segmentos da empresa, é necessário coletar dados de diferentes departamentos, a fim de compará-los, unificá-los, conciliá-los e realizar outras ações que respaldam a tomada de decisões em um contexto estratégico.

No entanto, para que a integração desses dados de sistemas diversos ocorra de maneira eficaz, é imperativo que as informações estejam consistentes em todas as bases de dados que alimentam esse ecossistema de sistemas integrados. Isso envolve a aplicação de princípios fundamentais da ciência de dados, exigindo uma análise aprofundada sobre a qualidade, consistência e acessibilidade dos dados. Essa análise é essencial para orientar a tomada de decisões por meio do conjunto de integrações e para manter a integridade dos bancos de dados, *data warehouses* e *data lakes* que sustentam a estrutura tecnológica subjacente.

A modelagem de dados em um ambiente de integração corporativa depende das necessidades da organização e do tipo de dados que estão sendo tratados. *Databases* são ideais para transações, *data warehouse* para análise de negócios e *data lakes* para

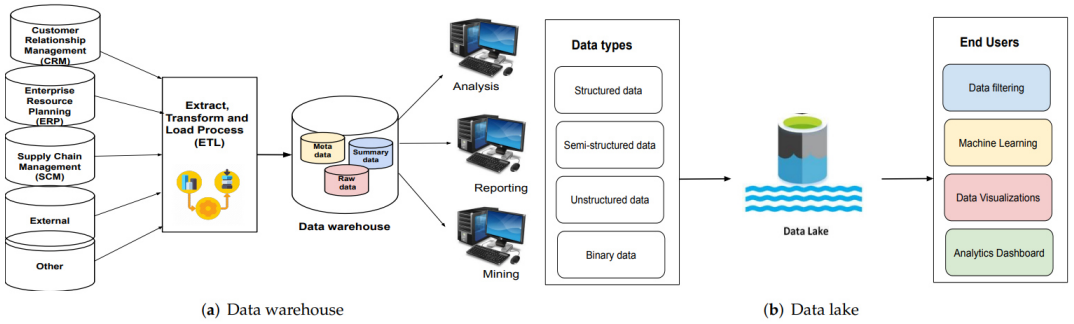
armazenamento flexível de dados brutos. Muitas empresas também adotam uma abordagem híbrida, integrando essas soluções para atender às suas demandas específicas de dados.

1. *Database*: Sistema que armazena, organiza e gerencia dados. Esses dados podem ser estruturados de diversas maneiras e são gerenciados para que possam ser facilmente acessados, gerenciados e atualizados. É usado para aplicações transacionais, onde a prioridade é a inserção, atualização e recuperação de dados de maneira rápida e consistente. No modelo de banco de dados relacional, se caracteriza pelo esquema muito bem definido. Utiliza modelagem de dados rigorosa, com tabelas, chaves primárias e estrangeiras para garantir a integridade dos dados. Geralmente, é altamente normalizado para evitar a redundância de dados. Outra abordagem é a de banco de dados NoSQL, projetados para lidar com tipos de dados e cenários que podem não se encaixar no modelo relacional. Usam estruturas como chave-valor, grafos, colunas ou documentos como forma de organização dos dados. A lógica não relacional se preocupa mais com os dados, estabelecendo como prioridade o tratamento deles.

2. *Data Warehouse*: Repositório centralizado de dados históricos de várias fontes dentro de uma organização, projetado para análise de negócios e consulta, permitindo que as empresas tomem decisões informadas. Utiliza modelagem dimensional, com tabelas de fatos e dimensões, para simplificar consultas complexas e agregações. Os dados são extraídos, transformados e carregados (ETL) de fontes operacionais para o data warehouse em intervalos regulares.

3. *Data Lake*: Repositório de dados brutos e não processados, que podem incluir dados estruturados, semi-estruturados e não estruturados. Ele oferece flexibilidade para armazenar grandes volumes de dados em sua forma original, sem a necessidade de uma estrutura rígida. A modelagem de dados em um *data lake* é mais orientada para o esquema, permitindo que os dados sejam moldados conforme necessário no momento da análise. É adequado para casos em que a variedade e o volume de dados são altos.

Figura 1 – Arquitetura data warehouse vs. arquitetura data lake



(a) Data warehouse (b) Data lake

Fonte: https://images.app.goo.gl/AiWHiiKBX6R6E8YU9.

As arquiteturas, e suas diferenças, empregadas pelas duas abordagens, são ilustradas na Figura 1. Para Woods (2011), “A diferença entre um *data lake* e um *data warehouse* é que, em um *data warehouse*, os dados são pré-categorizados no ponto de entrada, o que pode ditar como serão analisados.”, e acrescenta “no mundo do *big data*, não sabemos realmente qual o valor que os dados têm quando são inicialmente aceitos a partir do conjunto de fontes disponíveis”, trazendo uma avaliação positiva quanto aos mecanismos de *Data Lake*, os quais foram criados para atender à necessidade de lidar com conjuntos de dados cada vez maiores, percebendo-se que a análise torna-se bastante custosa, especialmente no processamento online.

Uma excelente estratégia para a manipulação de dados, quando lidamos com alta escalabilidade e volume de informações, são os bancos de dados não relacionais. Um SGBD não relacional busca atender à demanda atual de disponibilidade em detrimento da consistência dos dados, adotando com maior ênfase arquiteturas de dados altamente escaláveis e acessíveis.

O banco de dados relacional, apesar das necessidades do *Big Data*, mostra-se uma tecnologia muito eficaz para a grande maioria dos projetos de software destinados ao armazenamento de dados estruturados. Segundo Elmasri e Navathe (2011), “o modelo relacional representa o banco de dados como uma coleção de relações. Informalmente, cada relação é semelhante a uma tabela de valores”.

A aplicação dos modelos de dados e a sua disponibilidade na rede exigem a adequação da disponibilidade aliada à consistência dos dados. Para isso, os SGBDs relacionais devem atender às propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade):

- Atomicidade: transações são processadas como uma entidade única, retornando ao estado inicial caso haja falha em uma delas.
- Consistência: os dados gravados no banco devem sempre ser válidos, garantindo a estabilidade.
- Isolamento: a execução concorrente permite deixar o banco de dados no mesmo estado em que ele estaria caso as transações fossem executadas em sequência.
- Durabilidade: transações concluídas são gravadas em dispositivos de memória (não volátil), como discos rígidos, garantindo que os dados estejam sempre disponíveis, mesmo que a instância do BD seja reiniciada.

Em síntese, a integração de dados, para atender às necessidades de negócio corporativas, vai além da conexão entre sistemas, sendo também um esforço conjunto que visa garantir que os processos operem com dados cada vez mais confiáveis, consistentes e acessíveis. Isso possibilita que os gestores tomem decisões estratégicas fundamentadas com base em informações precisas e de alta qualidade.

2.3 Padrões de Projeto

Para que essa integração ocorra, utilizamos um conjunto de padrões de projetos que orientam o design dentro do contexto e escopo definido, interpretando as necessidades de um projeto a um arcabouço de projetos já catalogados que facilitam a tomada de decisão sobre a modelagem de uma integração entre sistemas. Esses padrões, catalogados pelo GOF (Gang of Four) são um conjunto de padrões de design de software amplamente reconhecidos e documentados por Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides em seu livro "Design Patterns: Elements of Reusable Object-Oriented Software.", são amplamente utilizados na programação orientada a objetos para criar software mais flexível, extensível e fácil de manter. Eles representam soluções testadas e comprovadas para problemas comuns de design de software.

"Um padrão descreve um problema que ocorre inúmeras vezes em determinado contexto, e descreve ainda a solução para esse problema, de modo que essa solução possa ser utilizada sistematicamente em distintas situações. Cada padrão tem uma característica diferente para ajudar em algum lugar onde se precisa de mais flexibilidade ou precisa encapsular uma abstração ou de se fazer um código menos casado.” (GAMMA,HELM, et al., 2000)

Os padrões de projeto desempenham um papel essencial no design de integração de software. Em um contexto de integração de sistemas, é crucial estabelecer um contrato de interface que permita a comunicação fluida entre os sistemas, seguindo protocolos definidos e uma arquitetura bem planejada. Um design de projeto bem orientado é fundamental, e uma abordagem eficaz é basear-se em soluções previamente testadas, comprovadas e amplamente adotadas.

A utilização de padrões de projeto oferece uma abordagem segura e coerente para conceber uma integração de software. Segundo Gamma, (et al., 2000), “os padrões de projeto representam um avanço considerável para a área de orientação a objeto, uma vez que

disponibiliza um catálogo de planos de projeto que admite a reutilização dessas soluções que foram testadas e provaram ser eficientes para a resolução de problemas semelhantes".

Esses padrões representam melhores práticas consolidadas e diretrizes que foram desenvolvidas ao longo do tempo por profissionais experientes. Ao aplicá-los, os desenvolvedores podem enfrentar desafios de integração com mais confiança, alinhando-se com as expectativas do domínio do projeto e reduzindo o risco de erros.

Os padrões de projeto fornecem um alicerce sólido para o design de integração de software, permitindo que as equipes de desenvolvimento construam sistemas interconectados de forma mais eficaz e confiável, seguindo as melhores práticas estabelecidas pela comunidade de desenvolvedores.

2.4 Escalabilidade

A escalabilidade se refere à capacidade de um sistema crescer ou encolher de forma eficiente em resposta às mudanças na carga de trabalho. Segundo Bondi (2000), "O conceito conota a habilidade de um sistema de acomodar um número crescente de elementos ou objetos, para processar crescentes cargas de trabalho graciosamente, e/ou ser suscetível a ser ampliado."

Esse ajuste pode ser feito de duas maneiras principais: escalabilidade horizontal, na qual novos recursos, como servidores adicionais, são acrescentados para aprimorar a disponibilidade do sistema e distribuir melhor a carga recebida, ou escalabilidade vertical, que envolve aumentar a capacidade das máquinas existentes, seja por meio da adição de mais memória, ou pelo upgrade das CPUs, ampliando assim a capacidade de processamento. A escalabilidade é fundamental para garantir que um sistema possa lidar com picos de tráfego, mantendo o desempenho e a disponibilidade. Ela é sentida, normalmente, como um problema de desempenho causado pela limitada capacidade de servidores e rede.

Quando projetamos uma aplicação com foco na escalabilidade, nosso objetivo principal é assegurar um gerenciamento eficiente dos recursos. Esse tipo de design não impõe limites a nenhuma camada ou componente da aplicação. A escalabilidade deve ser cuidadosamente considerada em todos os níveis, desde a interface com o usuário até a base de dados. Ao adotar essa abordagem, estamos preparando a aplicação para crescer e se adaptar às demandas em constante evolução, garantindo a eficiência no uso dos recursos disponíveis em todos os aspectos do sistema.

Em um produto de integração, é fundamental avaliar os aspectos que envolvem a escalabilidade e a disponibilidade dos sistemas envolvidos. Portanto, ao projetá-lo, é crucial garantir que nenhum processo aguarde mais tempo do que o necessário. Cada momento em que um processo utiliza um recurso é um momento em que outro processo aguarda para ser executado. Logo, é fundamental segregar nossos processos em duas principais categorias: síncronos e assíncronos.

Há momentos em que a aplicação deve executar operações de forma síncrona, como quando aguardamos o resultado de uma ação para determinar se foi bem-sucedida ou não. Nesse caso, todas as operações dependentes desse resultado devem esperar. Para melhorar a escalabilidade, recorremos a operações assíncronas, especialmente ao manipular tarefas de longa duração. Essas tarefas são adicionadas a uma fila para serem executadas posteriormente por um processo separado, evitando a espera ociosa e maximizando a utilização de recursos.

Além disso, a contenção de recursos é uma das principais causas de problemas de escalabilidade. Em qualquer design, uma aplicação tem recursos limitados. Além de acelerar tarefas de longa duração, existem outras medidas para evitar a contenção de recursos. Ao executar operações que envolvem recursos escassos sujeitos a contenção, devemos adiar a utilização desses recursos o máximo possível e liberá-los o mais rápido possível. Quanto menos tempo um processo monopoliza recursos escassos, mais rapidamente esses recursos estarão disponíveis para outros processos. Essa estratégia é fundamental para garantir que os recursos sejam utilizados de forma eficaz, evitando gargalos de recursos e melhorando a escalabilidade do sistema.

Como prática recomendada, é essencial adotar a estratégia de particionar recursos e atividades, visando minimizar as interdependências entre eles e, assim, mitigar o risco de criar gargalos. Ao particionar atividades, aliviamos a carga imposta sobre recursos de alto custo. No entanto, é importante exercer cautela, pois o particionamento nem sempre é a escolha ideal. Em alguns casos, ele pode tornar o sistema mais complexo, especialmente quando dividimos recursos com dependências interligadas, o que pode introduzir sobrecarga significativa em certas operações. Portanto, ao aplicar a estratégia de particionamento, é essencial considerar cuidadosamente as implicações e os benefícios específicos a cada situação, a fim de garantir a escalabilidade eficaz do sistema.

- Use tecnologias de *clustering*: Envolve a distribuição de carga e recursos em múltiplas instâncias ou nós. Isso não apenas melhora a disponibilidade e a redundância do

sistema, mas também aumenta a capacidade de processamento para atender a um maior número de solicitações.

- Considere camadas lógicas contra camadas físicas: O código deve ser projetado de forma que a lógica do negócio seja independente da infraestrutura física subjacente. Dessa forma, as alterações na infraestrutura (por exemplo, migração para a nuvem) não devem exigir mudanças significativas no código de negócios. Isso simplifica as operações de escalabilidade e manutenção.
- Isole métodos transacionais: Métodos transacionais, que envolvem operações de escrita no banco de dados, podem ser críticos para a consistência dos dados. Isolá-los e garantir que sejam tratados de maneira adequada é fundamental.
- Elimine o estado da camada de negócios quando possível: Sempre que possível, busque eliminar o estado da camada de negócios e opte por uma abordagem sem estado (stateless). Dessa forma, as solicitações podem ser tratadas de forma independente, o que facilita a distribuição de carga e a escalabilidade horizontal.
- Reveja as melhores práticas para performance: Por fim, essa atividade compreende otimizações de código, monitoramento, ajustes de configuração e a incorporação de técnicas eficazes para melhorar a velocidade e a eficiência do software. A busca constante por melhorias de desempenho é essencial para garantir que o sistema possa expandir e atender às demandas crescentes de forma eficaz.

Com o intuito de ilustrar a capacidade de escalabilidade, podemos analisar um caso prático comum ao ecossistema de software nos dias de hoje. Sabe-se que um *e-commerce* precisa lidar com um aumento significativo no tráfego durante eventos importantes (festas de fim de ano e *black friday*, por exemplo). Para tanto, a escalabilidade deve permitir que a infraestrutura do sistema se expanda automaticamente para acomodar o aumento no número de visitantes, garantindo que o site não fique lento ou saia do ar.

Em cenários como esse, é comum adotar uma arquitetura serverless baseada em nuvem (esse é um paradigma que se popularizou com o AWS Lambda em 2014), onde os servidores respondem dinamicamente aos eventos acionados. Isso significa que a disponibilidade das máquinas é ajustada de acordo com a demanda real, garantindo uma experiência ininterrupta para os usuários. Além disso, enfatiza a elasticidade e a ausência de custos fixos associados à infraestrutura.

Outro cenário extremamente relevante nos dias de hoje, intrínseco à integração de componentes de software, é o cenário do *Big Data*, no qual a necessidade de armazenar petabytes de informações tornou-se uma demanda comum. À medida que acumulamos cada

vez mais dados, aprimoramos a precisão das análises e expandimos significativamente a riqueza da amostragem disponível. Entretanto, o gerenciamento dessa vasta quantidade de dados deve ser considerado desde a fase de concepção do projeto, pois o volume de dados é uma métrica fundamental para a escalabilidade. A capacidade de lidar eficazmente com esse acúmulo de informações torna-se essencial para garantir que as análises sejam efetivas e que a infraestrutura seja escalável para atender às crescentes demandas de processamento e armazenamento.

2.5 Desempenho

O desempenho refere-se à capacidade de um sistema executar suas funções dentro de limites de tempo aceitáveis. Isso envolve otimizar algoritmos, hardware e software para garantir que as operações sejam concluídas de maneira eficiente.

No contexto de integração de sistemas, é fundamental assegurar que os recursos sejam disponibilizados dentro de um limite aceitável de tempo de resposta. Em outras palavras, o desempenho é fundamentalmente um requisito não funcional que precisa ser bem definido para que todo o conjunto de negócios possa atingir sua finalidade proposta com eficácia.

Existem algumas métricas essenciais para avaliar o desempenho de software e, principalmente, aplicações distribuídas. Entre elas, destacam-se o tempo médio de resposta, a taxa de erros e a vazão, que fornecem insights valiosos sobre a eficácia de um sistema.

O tempo médio de resposta é um indicador fundamental que mede o intervalo desde o momento em que o usuário envia uma solicitação até o momento em que recebe uma resposta do servidor, seguindo o modelo cliente-servidor. Para calcular o tempo médio de resposta, somamos os tempos de todas as amostras coletadas e dividimos pela quantidade total de amostras.

A vazão (throughput), por sua vez, representa a quantidade de requisições bem-sucedidas por unidade de tempo. O ideal é que a vazão seja capaz de acompanhar o aumento na carga de acessos. Uma vazão alta é essencial para manter baixos tempos de resposta e garantir um bom desempenho. Se a vazão não acompanha a crescente demanda, problemas como aumento na taxa de resposta e erros em requisições podem surgir, juntamente com gargalos na infraestrutura.

A taxa de erros (error rate) é um indicador crucial que calcula a porcentagem de erros em relação ao número total de requisições. Monitorar a taxa de erros é essencial, pois

ajuda a identificar o ponto em que uma aplicação começa a não suportar o volume de acessos, indicando a necessidade de ajustes ou melhorias.

Essas métricas desempenham um papel vital na avaliação e na otimização do desempenho das aplicações envolvidas na integração de sistemas. Elas fornecem informações valiosas para o desenvolvimento e a manutenção de sistemas, permitindo ajustes mais eficientes, capazes de lidar eficazmente com o crescimento da demanda e fornecer uma experiência de usuário.

Portanto, o desempenho é um fator crítico para o sucesso da integração, exigindo a otimização dos processos, que, para esse contexto, se referem aos processos de pagamento, e a alocação de recursos adequados para atender às demandas dos clientes durante picos de tráfego, como promoções sazonais (no cenário de e-commerce). Isso garante que as operações de pagamento sejam realizadas dentro de limites de tempo aceitáveis, proporcionando uma experiência de compra positiva aos clientes.

2.6 Segurança da Informação

A segurança engloba um conjunto de medidas para proteger os recursos e dados do sistema contra ameaças, com o objetivo de garantir a confidencialidade, integridade e disponibilidade dos dados. Isso inclui práticas como autenticação, autorização, criptografia e monitoramento. Segundo o Guia de Segurança da Informação do Instituto Nacional de Padrões e Tecnologia dos Estados Unidos (NIST), "a implementação eficaz de medidas de segurança, como autenticação forte e criptografia robusta, é crucial para proteger os sistemas de informação contra ameaças cada vez mais sofisticadas. A segurança da informação é uma jornada contínua que exige vigilância e adaptação constantes".

Considerando o contexto de integração de sistemas, a segurança da informação deve ser uma prioridade desde as fases iniciais do projeto. Isso implica na implementação de medidas de segurança robustas, incluindo práticas de autenticação e autorização eficazes, bem como o uso de protocolos de criptografia fortes para proteger a integridade dos dados durante o trânsito e no repouso.

Nesta abordagem, a cultura *DevOps*, que pode ser definida como uma metodologia de desenvolvimento de software responsável por unificar as equipes de desenvolvimento e operações, representa uma prática de engenharia de software que visa à integração das áreas de desenvolvimento e operações. O propósito é alcançar uma qualidade superior nas entregas. Uma das principais contribuições dessa cultura é a integração dos times

de segurança da informação, banco de dados e governança de TI. Isso ocorre ao longo de todo o ciclo de desenvolvimento, desde o planejamento do produto de software até o lançamento da versão em produção, visando priorizar, também, a segurança da informação.

Adicionalmente, a implementação de protocolos de monitoramento contínuo é essencial para identificar e mitigar possíveis ameaças à segurança, garantindo uma resposta rápida a incidentes de segurança e minimizando potenciais danos.

2.7 Modularidade

A modularidade compreende a prática de separar um sistema em módulos independentes, nos quais cada módulo desempenha uma função específica e pode ser atualizado ou substituído sem afetar o sistema como um todo. Essa abordagem facilita a manutenção e evolução do sistema.

A abordagem da modularidade é fortemente endossada por Fowler (2002), que afirma: "A modularidade é a chave para uma arquitetura de software flexível e escalável. Ao dividir um sistema em módulos bem definidos, é mais fácil manter e estender a funcionalidade sem comprometer a estabilidade do sistema."

Ao conceber uma integração de sistemas heterogêneos, é fundamental projetar uma arquitetura com baixo nível de acoplamento. Isso começa com a definição coesa dos processos que serão executados para que, em seguida, seja possível dividir as responsabilidades entre as partes do software, dessa forma, resultando em uma arquitetura coesa, desacoplada e com responsabilidades bem definidas. Cada serviço, ou API, pode ser visto como um módulo independente, o que permite a evolução e manutenção individuais sem afetar todo o sistema. Essa abordagem é especialmente valiosa em um cenário onde a evolução constante e a necessidade de se adaptar rapidamente a novos requisitos são comuns.

A arquitetura de microsserviços, que será melhor discutida nas seções seguintes, representa bem esse conceito, uma vez que promove a modularidade como princípio central. Cada microsserviço é uma unidade autônoma que pode ser desenvolvida, implantada e escalada independentemente dos outros. Essa abordagem oferece algumas vantagens quando aplicada ao projeto de integração de sistemas incluindo manutenção, escalabilidade, reutilização de componentes, flexibilidade, testes e controle de falhas.

A modularidade, segundo Martin (2002), permite que os sistemas sejam mais fáceis de entender, manter e evoluir. Ela promove a reutilização de código, a testabilidade e a escalabilidade. Além disso, a modularidade torna possível a substituição de partes do sistema

sem afetar o restante, o que é essencial para manter a flexibilidade e a adaptabilidade do software. Ainda segundo ele, a ideia é que a arquitetura do software seja uma coleção de módulos independentes, todos juntos na compreensão do negócio. As funcionalidades não devem ser misturadas; ao contrário, cada módulo deve ser uma cápsula com uma única responsabilidade.

Em síntese, a modularidade proporciona às organizações um conceito essencial para enfrentar os desafios dinâmicos do ambiente empresarial contemporâneo. Ao adotar abordagens modulares, as empresas aumentam a capacidade de adaptar-se mais rapidamente às mudanças, possibilitando uma evolução contínua na integração de sistemas e recursos. Essa flexibilidade resulta não apenas em uma resposta mais eficaz às demandas do mercado, mas também em uma maior adaptabilidade a avanços tecnológicos, permitindo com que as interconexões entre sistemas permaneçam eficientes e alinhadas aos objetivos estratégicos da organização. Dessa forma, a modularidade emerge como um alicerce fundamental para a construção e manutenção de sistemas interconectados que se destacam pela agilidade, durabilidade e eficácia em um cenário de tecnologia em constante transformação.

2.8 Monitoramento e Rastreabilidade

Quando trabalhamos, especialmente, com a integração de sistemas, o objeto de valor entregue aos clientes, usuários e/ou partes interessadas envolvidas só pode ser considerado "pronto" se houver viabilidade para o acompanhamento dos processos. Em outras palavras, um requisito indispensável para esse modelo de software é a transparência dos processos e recursos envolvidos, bem como a rastreabilidade de seus estados. Idealmente, isso deve permitir o acompanhamento em tempo real do ambiente e dos sistemas presentes no contexto da integração em questão.

O monitoramento envolve a observação em tempo real do desempenho e estado do sistema, proporcionando visibilidade do processo após a sua implantação em ambiente produtivo. Por outro lado, a rastreabilidade consiste no registro de eventos e logs com o propósito de auditoria e diagnóstico, permitindo a rápida identificação dos problemas e, consequentemente, uma melhor capacidade de resposta.

De acordo com Kim (2020), "o monitoramento e a rastreabilidade são fundamentais para a prática eficaz da cultura *DevOps*. Eles fornecem a visibilidade necessária para identificar problemas rapidamente, colaborar de forma eficaz entre equipes e melhorar continuamente a entrega de software".

Em cenários nos quais os sistemas operam em segundo plano, frequentemente chamados de *jobs*, ou seja, quando não há um usuário humano executando uma ação que ative o processo, o monitoramento é ainda mais crucial. Isso permite que as partes envolvidas identifiquem cenários de falhas na integração, além de garantir visibilidade sobre as integrações bem-sucedidas. Para os processos automatizados, a visibilidade do monitoramento é uma salvaguarda crítica. Ao monitorar jobs e processos em segundo plano, as equipes de operações podem detectar problemas, como falhas na execução ou atrasos, antes que eles causem impacto nos processos subsequentes ou nos resultados esperados.

Em tais cenários, o monitoramento não apenas ajuda a identificar problemas em tempo real, mas também a fornecer uma trilha de auditoria completa.

Além disso, a visibilidade sobre as integrações bem-sucedidas é valiosa para avaliar o desempenho do sistema e para fins de otimização contínua. O monitoramento permite que as equipes identifiquem padrões de sucesso e eficiência, possibilitando ajustes e melhorias nos processos automatizados.

2.9 Arquiteturas de Software

A arquitetura define a estrutura e o design geral do sistema, incluindo a forma como os componentes se conectam e interagem. A definição clássica dessa disciplina diz que arquitetura de software define o que é o sistema em termos de componentes computacionais e os relacionamentos entre estes componentes (Shaw, 96).

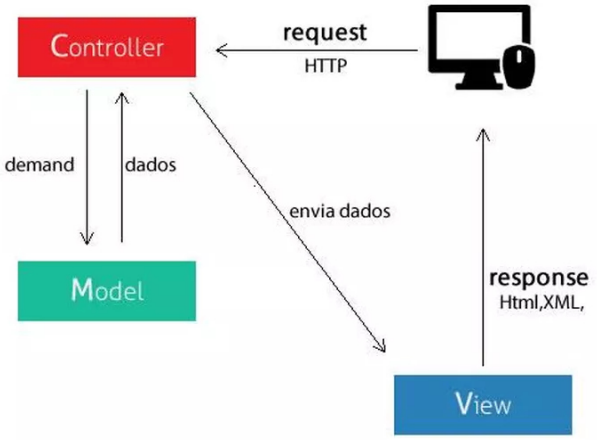
Uma das primeiras etapas executadas em um projeto de integração, após compreender as necessidades de negócio e estabelecer com clareza os objetivos do projeto, é a definição de uma arquitetura que atenda de forma otimizada a essas necessidades. Isso envolve uma análise prévia dos padrões de software consolidados no mercado e a identificação de analogias e associações que possam ser aplicadas a uma arquitetura capaz de solucionar eficazmente às demandas do contexto empresarial.

A seguir, discutiremos de forma mais detalhada algumas das arquiteturas de software mais debatidas na comunidade de desenvolvedores e arquitetos de software na atualidade, sob a perspectiva da integração de sistemas.

2.9.1. Arquitetura em Camadas

A arquitetura em camadas separa um sistema em camadas distintas, como a camada de apresentação, a camada de lógica de negócios e a camada de dados. Essa abordagem promove a modularização e a manutenção do sistema. Um padrão de projeto que materializa essa solução é o MVC (*Model View Controller*), que divide a aplicação em três camadas: a camada de interação do usuário (*view*), a camada de manipulação dos dados (*model*) e a camada de controle (*controller*), como ilustrado na Figura 3.

Figura 2 – Diagrama Arquitetural do Paradigma em Camadas



Fonte: Aplicativo de diagramação Lucidchart (2023).

Através da adoção do padrão MVC, as solicitações do usuário seguem um fluxo organizado, direcionando-se inicialmente ao Controlador. Este componente assume a responsabilidade de interagir com o Modelo, executando as ações solicitadas pelo usuário e/ou recuperando os resultados de consultas necessárias. Posteriormente, o Controlador seleciona a Exibição apropriada, fornecendo-a com os dados solicitados do Modelo.

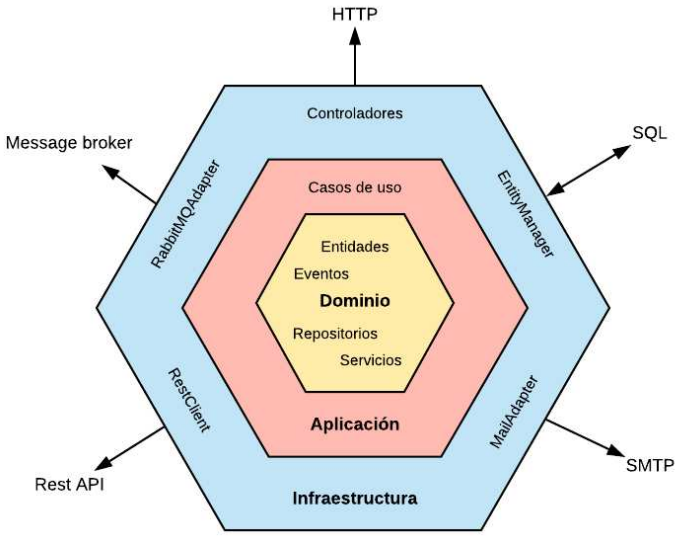
Essa descrição das responsabilidades visa contribuir de forma positiva para o dimensionamento do aplicativo em termos de complexidade, visto que é mais fácil codificar, depurar e testar algo (modelo, exibição ou controlador) que possui apenas uma única tarefa.

Por fim, o acoplamento flexível entre os três componentes principais de um aplicativo MVC também promove o desenvolvimento paralelo. Enquanto um desenvolvedor pode trabalhar na exibição, um segundo desenvolvedor pode trabalhar na lógica do controlador e um terceiro desenvolvedor pode se concentrar na lógica de negócios do modelo.

2.9.2 Arquitetura em Hexágono (Port and Adapter)

Também conhecida como "*Port and Adapter*" ou "*Hexagonal Architecture*", essa arquitetura enfatiza a separação das partes do sistema em portas (interfaces) e adaptadores (implementações). Ela facilita a substituição de componentes e o teste unitário. A arquitetura Hexagonal divide o sistema em dois grupos: as classes de domínio, ou seja, aquelas diretamente relacionadas com o negócio do sistema, e as classes relacionadas à infraestrutura, tecnologias e responsáveis pela integração com sistemas externos, como bancos de dados, serviços REST, mensagerias, etc.

Figura 3 – Diagrama Arquitetural em Hexágono



Fonte: Aplicativo de diagramação Lucidchart (2023).

Além disso, em uma Arquitetura Hexagonal, classes de domínio não devem depender de classes relacionadas com infraestrutura, tecnologias ou sistemas externos. A vantagem dessa divisão é desacoplar esses dois tipos de classes, conforme arquitetura modelada na Figura 3.

Em 2005, Cockburn comenta sobre a arquitetura proposta: “Eu finalmente tive esse momento aha! (junho de 2005) no qual vi que as faces do hexágono [interno] são portas… e os objetos entre os dois hexágonos são adaptadores e, portanto, o padrão arquitetural consiste em uma Arquitetura Baseada em Portas e Adaptadores. Esse nome admite uma explicação melhor do que aquela que eu propus com o nome Arquitetura Hexagonal”.

Portanto, o fraco acoplamento entre as partes do sistema proposto pela arquitetura em hexágono possibilita uma melhor adaptação na criação e/ou substituição de adaptadores. Além disso, proporciona facilidade na troca ou atualização de tecnologias e viabiliza uma implementação mais fácil da cobertura de testes no sistema.

2.9.3. Arquitetura de Microsserviços

A arquitetura de microsserviços envolve a divisão de um aplicativo em serviços independentes e autônomos, cada um focado em uma função específica. Pondo em prática o primeiro princípio do SOLID, que é o de responsabilidade única. Esse modelo permite escalabilidade, flexibilidade e a capacidade de desenvolver e implantar serviços de forma independente.

A arquitetura de microsserviços (Newman, 2015), evoluiu com o objetivo de aumentar a agilidade ao entregar o software como serviços altamente escaláveis, independentes e através de entregas contínuas.

Esse modelo arquitetural ganhou destaque no mercado de desenvolvimento de software a partir das melhores práticas de uma série de organizações líderes e o seu esforço de construir sistemas empresariais grandes e continuamente crescentes, sustentáveis, adaptáveis, e capazes de reagir rapidamente aos requisitos de software sempre em mudança. (de la Torre, et al., 2017)

Essa proposta visou abordar diretamente o desafio crescente da escalabilidade, uma questão que o modelo tradicional de desenvolvimento de softwares corporativos vinha enfrentando. Este aumento trouxe alguns obstáculos, como o aumento da complexidade dos sistemas, o crescimento do tempo de desenvolvimento, a dependência e o alto acoplamento entre as partes da aplicação, a dificuldade em realizar mudanças, a complexidade na implantação de melhorias e, consequentemente, a ampliação do tempo necessário para a entrega de valor ao cliente.

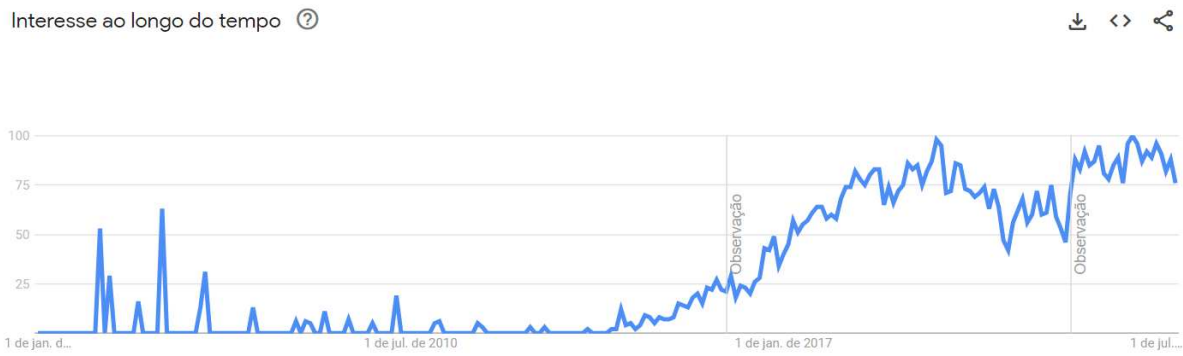
Os microsserviços são ideais para realizar múltiplas funções empresariais, com cada microsserviço focado na realização de uma única função comercial ou subfunção. As equipes são livres para inovar a seu próprio ritmo. (de la Torre, et al., 2017)

A forma como os microsserviços contribuem para mitigar essas problemáticas no mercado de desenvolvimento de aplicativos, mencionadas anteriormente, é propondo componentes altamente coesos e interconectados, podendo ser gerenciados por equipes multidisciplinares capazes de desenvolver e manter sistemas de software complexos e

robustos. Essa abordagem possibilita o aumento da velocidade e da qualidade necessárias para atender às demandas dos negócios digitais atuais.

Há alguns anos, a arquitetura baseada em microsserviços tem ganhado destaque e um crescente interesse da comunidade tecnológica, como é possível observar na Figura 4. Isso ocorre em grande parte devido ao aumento das tecnologias disponíveis em aplicações de computação em nuvem, assim como ao crescimento dos dispositivos computacionais móveis (por exemplo, IoT) (Newman, 2015).

Figura 4 – Pesquisa sobre o crescimento do interesse em Microsserviços entre 2004-2023



Fonte: Google 2023
(https://trends.google.pt/trends/explore?date=all&q=microservice%20architecture).

Alguns fatores importantes para a análise desse modelo de arquitetura orientam-se pelo estudo de sua abordagem em relação às necessidades inerentes à integração de sistemas. O primeiro tópico relevante e de impacto nas integrações em geral é a escalabilidade, que, para os microsserviços, é percebida como uma melhoria, especialmente quando comparada às aplicações monolíticas. Estas, embora possam ter requisitos muito diversos, geralmente possuem uma única base de dados (Abbott & Fisher, 2009).

Ao optarmos por trabalhar com módulos independentes, obtemos a capacidade de sustentar ou até mesmo melhorar o desempenho frente ao crescimento do volume dos processos. Em geral, isso ocorre devido ao fato de pequenos serviços serem responsáveis por um escopo bem definido e enxuto de negócio, permitindo, em primeira medida, um maior controle das ações e recursos. Isso possibilita garantir, de maneira mais assertiva, a sustentação das performances individuais diante do aumento no volume da operação.

Outro fator de destaque para esta análise é a autonomia dos módulos de software. A atualização independente é uma característica essencial dos microsserviços, permitindo que cada serviço seja implantado de forma autônoma. Isso proporciona uma maior flexibilidade,

uma vez que qualquer alteração em um serviço pode ser realizada sem a necessidade de coordenação com outras equipes, simplificando a vida do programador e facilitando a implementação de CI/CD.

Além disso, a facilidade de manutenção é uma vantagem distintiva dos microsserviços. O código restrito a uma capacidade específica torna-se mais compreensível em comparação com arquiteturas monolíticas. Os IDEs conseguem lidar facilmente com pequenas quantidades de código, resultando em builds mais leves. Trabalhar com bases de código menores não apenas acelera o desenvolvimento, mas também proporciona uma compreensão mais clara dos efeitos colaterais das alterações realizadas pelos programadores.

Outro benefício notável dos microsserviços é a melhoria na comunicação entre as equipes. Geralmente, esses serviços são construídos por equipes *Full-Stack*, ou multidisciplinares (como orienta a cultura DevOps), que compartilham o mesmo domínio. Essa colaboração estreita aprimora significativamente a comunicação entre os membros, pois todos trabalham juntos em uma única equipe. Essa abordagem não só promove a coesão e os objetivos comuns, mas também contribui para que o serviço final seja verdadeiramente o produto da equipe, destacando a importância da sinergia durante todo o processo de desenvolvimento.

2.9.4. Arquitetura em Eventos

Nesta arquitetura, os componentes do sistema se comunicam por meio de eventos assíncronos. Ela consiste em serviços produtores de eventos, que o fazem com o objetivo de acionar alguma ação devido ao acontecimento (chamado de evento), e em serviços consumidores, que escutam os eventos para que seja efetuada uma tomada de decisão dentro do contexto de negócio.

Um evento pode ser a adição de um novo produto ao carrinho de compras em um *e-commerce*, pode ser uma aprovação dada pelo gestor em uma solicitação de compra, ou ainda a confirmação, por parte do cartão de crédito, da compra. Todos esses cenários ilustram eventos que requerem alguma ação do sistema diante de sua consolidação.

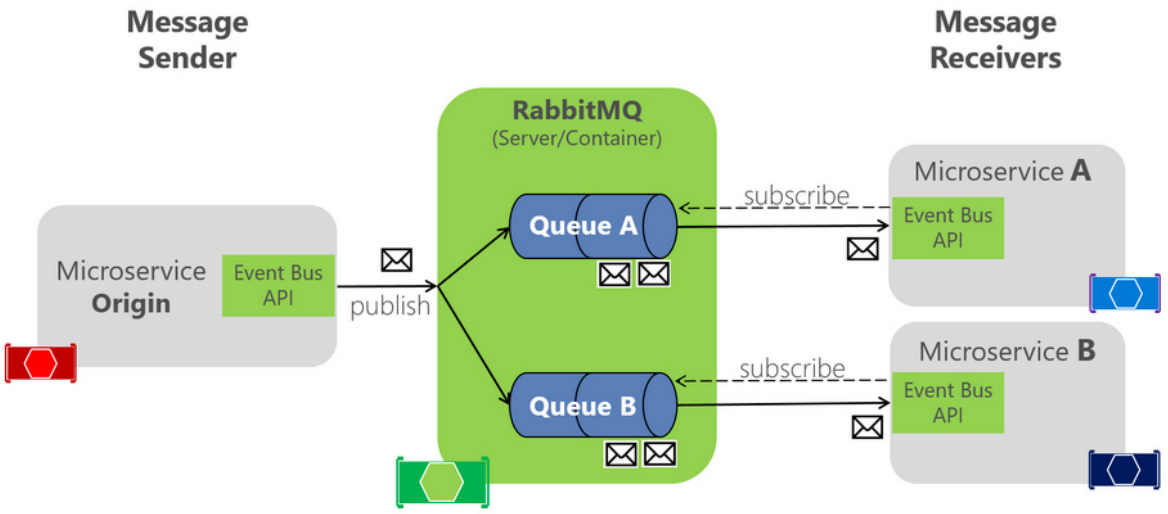
Essa abordagem significa colocar em prática os benefícios de integrações verdadeiramente assíncronas, ou seja, nas quais uma operação não precisa aguardar que a outra seja finalizada. Isso possibilita a concorrência e o paralelismo entre os processos e, desse modo, libera os recursos para os demais processos. Logo, essa é especialmente adequada para sistemas altamente distribuídos e escaláveis.

2.9.4.1 Mensageria RabbitMQ

Um dos modelos mais utilizados para a construção de serviços orientados a eventos é o padrão de mensagens. Nele, os sistemas se comunicam por meio de mensagens assíncronas, geralmente utilizando sistemas de mensagens ou filas. Em 2003, Gregor Hohpe e Bobby Woolf, no livro "*Enterprise Integration Patterns*", deram maior destaque a um dos temas centrais no estudo de sistemas distribuídos: lidar com serviços assíncronos, como afirmaram: “As mensagens assíncronas são fundamentalmente uma reação pragmática aos problemas dos sistemas distribuídos. O envio de uma mensagem não exige que ambos os sistemas estejam ativos e prontos ao mesmo tempo. Além disso, pensar na comunicação de forma assíncrona obriga os desenvolvedores a reconhecer que trabalhar com uma aplicação remota é mais lento, o que incentiva o design de componentes com alta coesão (muito trabalho local) e baixa adesão (trabalho seletivo remotamente)”.

O RabbitMQ é um software de mensagens de código aberto desenvolvido em Erlang que suporta diversos protocolos, tais como: *Advanced Message Queuing Protocol* (AMQP) e *Message Queuing Telemetry Transport* (MQTT). Essa solução permite que aplicações sejam desenvolvidas utilizando o paradigma de comunicação indireta, muito utilizada em sistemas distribuídos que se apoiam na arquitetura baseada em eventos, em que o destinatário pode ser desconhecido, uma vez que ele lida com o tráfego de mensagens entre diferentes processos, como ilustrado na Figura 5.

Figura 5 – Design do modelo de arquitetura em mensageria RabbitMQ

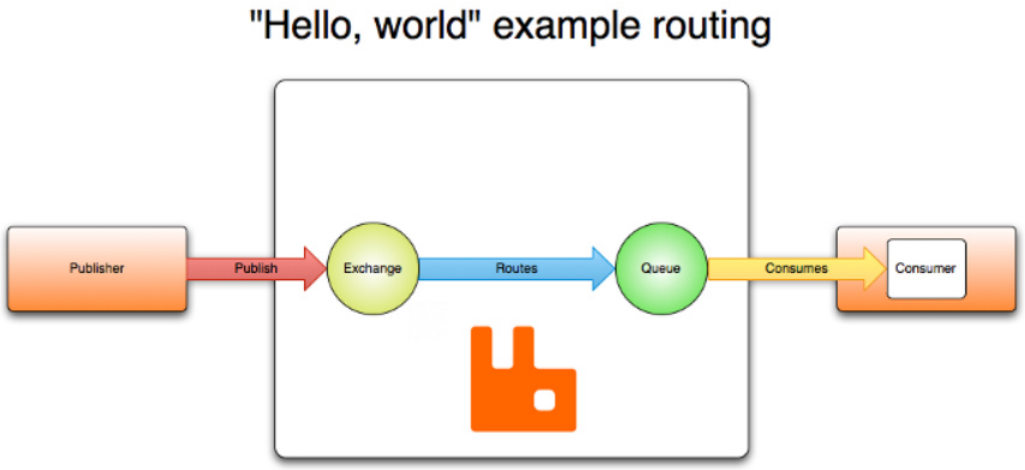


Fonte: Documentação da Microsoft (https://learn.microsoft.com/pt-br/dotnet/architecture/microservices/multi-container-microservice-net-applications/rabbitmq-event-bus-development-test-environment).

Existem alguns conceitos chaves que fazem parte da infraestrutura do modelo de arquitetura de mensageria e que serão definidos a seguir e ilustrados na Figura 6:

- Fila: Onde as mensagens enviadas são armazenadas até que um processo as retire.
- Mensagem: Objeto enviado por um processo para uma ou mais filas com o propósito de ser recebido por outro processo. Além disso, uma mensagem AMQP tem cabeçalho (semelhante aos *headers* http) e *payload* (o *body* efetivamente). Os Cabeçalhos, por sua vez, são utilizados para algumas finalidades, como o roteamento.
- Publicador/*Publisher*/Produtor: Processo que insere mensagens na fila.
- Assinante/*Subscriber*/Consumidor: Processo que retira mensagens da fila.
- *Channels*: Canal de comunicação necessário para trabalhar com filas.
- *Exchange*: É um objeto programável de roteamento. Com ele, pode-se definir um conjunto de regras de roteamento. Essas regras podem fazer uma mensagem ir direto para uma fila, ou mesmo ser distribuída em diversas filas.

Figura 6 – Modelo de interação entre os processos do rabbitmq



Fonte: Documentação rabbitmq (https://www.rabbitmq.com/tutorials/amqp-concepts.html).

Em síntese, a interconexão entre os componentes, a gestão de filas e a comunicação assíncrona revelaram-se elementos-chave na construção de uma integração de sistemas eficiente e escalável. Portanto, o RabbitMQ é amplamente empregado no contexto de integração assíncrona com o modelo de comunicação por mensageria.

2.9.4.2 Apache Kafka

O Apache Kafka é definido pela própria empresa responsável pela tecnologia como sendo “uma plataforma de streaming de eventos distribuída de código aberto usada por milhares de empresas para pipelines de dados de alto desempenho, análise de streaming, integração de dados e aplicativos de missão crítica”.

Segundo o NARKHEDE (2017), "O Kafka é uma plataforma que permite que você desenvolva aplicativos em tempo real que podem consumir e reagir a fluxos de eventos de qualquer fonte.". Uma terceira definição, também interessante, é a do co-fundador do Confluent e co-criador do Apache Kafka, Jay Kreps, que disse: "O Kafka é uma plataforma de streaming de eventos que soluciona o problema de como mover dados entre aplicativos ou microsserviços".

Portanto, podemos descrevê-la como uma tecnologia desenvolvida para lidar com a transmissão de eventos em integrações assíncronas e, especialmente, com grandes volumes de dados, sendo muito eficaz em cenários de streaming em tempo real. Nele, as mensagens são organizadas em tópicos e retidas por um período configurável, o que possibilita que várias partes interessadas acessem e leiam essas mensagens quando necessário.

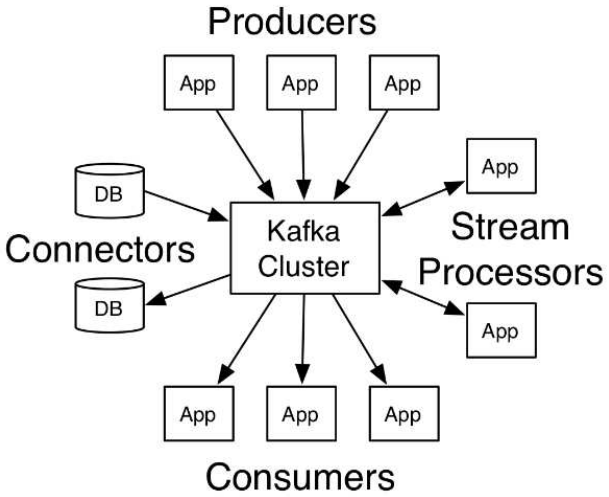
Além disso, o Kafka é altamente escalável, sendo projetado para lidar com volumes massivos de dados, graças ao suporte a particionamento e replicação, garantindo assim alta disponibilidade. Um outro fator relevante e de notável impacto, é a capacidade de manter mensagens por um período configurável, o que permite a recuperação de informações, independentemente de terem sido consumidos ou não. Dessa forma, o Kafka se torna uma opção muito viável para cenários de streaming em tempo real, como análise de dados em tempo real, processamento de eventos, integração de sistemas e registros de alterações.

O projeto do kafka foi desenvolvido para fornecer uma solução de alta throughput, tolerante a falha e horizontalmente escalável para lidar com streaming de dados em tempo real. Pode ser visualizado como uma fila de mensagens durável e distribuída que pode lidar com altos volumes de dados, permitindo que as aplicações produzam e consumam os eventos em fluxos de registros.

O kafka surgiu devido a limitação técnica das ferramentas que desenvolvidas sob a infraestrutura dos padrões de mensageria, na qual a partir de um problema prático de se trabalhar com um conjunto grande de dados (o problema do linkedin). Tentou-se utilizar o rabbitmq para performar na captura das informações, porém, o tempo de resposta era muito

alto, frente a um crescimento no volume de informações, principalmente os dados de comportamento do usuário inestimáveis para a compreensão das necessidades dos usuários na perspectiva do LinkedIn.

Figura 7 – Design do modelo de arquitetura kafka

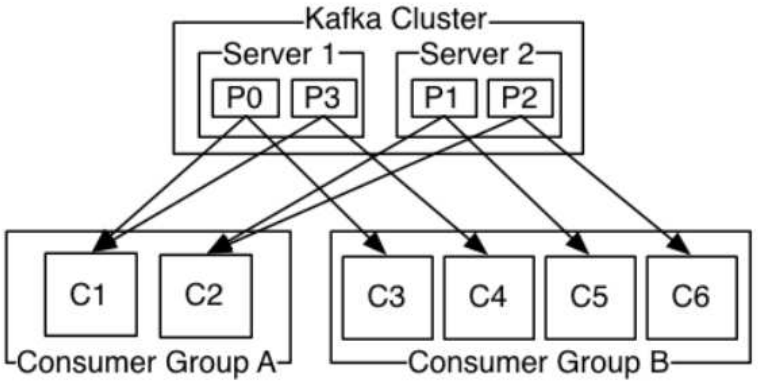


Fonte: Documentação kafka (https://kafka.apache.org/20/documentation.html).

A infraestrutura que sustenta a tecnologia do kafka possui conceitos essenciais para a integração de sistemas heterogêneos baseada em eventos, como ilustrado na Figura 7. Este alicerce garante a robustez e a eficiência da plataforma.

A Figura 8 apresenta o tópico, uma abstração utilizada pelo kafka para se referir ao fluxo de registros (dados) que são armazenados no *cluster* em categorias. O tópico é uma categoria no qual os registros são publicados, cada um composto por uma chave, um valor e um carimbo de data/hora, permitindo uma organização precisa dos dados. Além disso, eles são multi-assinantes; isto é, um tópico pode ter zero, um ou vários consumidores que assinam os dados registrados nele. Cada tópico é associado a um log particionado, e cada partição representa uma sequência ordenada e imutável de registros que é continuamente anexada a um log de commit estruturado.

Figura 8 – Topologia Consumer-*Cluster* para o kafka



Fonte: Documentação kafka (https://kafka.apache.org/20/documentation.html).

O segundo aspecto consiste na distribuição dos dados. As partições do log são distribuídas pelos servidores no *cluster*, com cada servidor manipulando dados e solicitações de compartilhamento das partições. Cada partição é replicada em servidores com o objetivo de garantir tolerância a falhas.

Avaliando a infraestrutura de rede que apoia essa integração de dados baseada no streaming de dados, temos que o kafka opera por meio de um protocolo binário que é transmitido sobre TCP, proporcionando uma comunicação eficiente e confiável. Esse protocolo é projetado de forma que todas as interações entre o cliente e o *cluster* Kafka ocorram como pares de mensagens de solicitação e resposta. Cada mensagem é delimitada por tamanho, o que requer uma avaliação cuidadosa da rede para determinar a melhor abordagem no equilíbrio entre throughput e latência de rede. A infraestrutura oferecida pelo Kafka permite a escolha de ambas as abordagens. Portanto, é possível trabalhar com um buffer maior, o que reduz o número de requisições na rede, mas pode afetar a latência, ou optar por diminuir o tamanho das mensagens, afetando, assim, mais significativamente o *throughput* da rede.

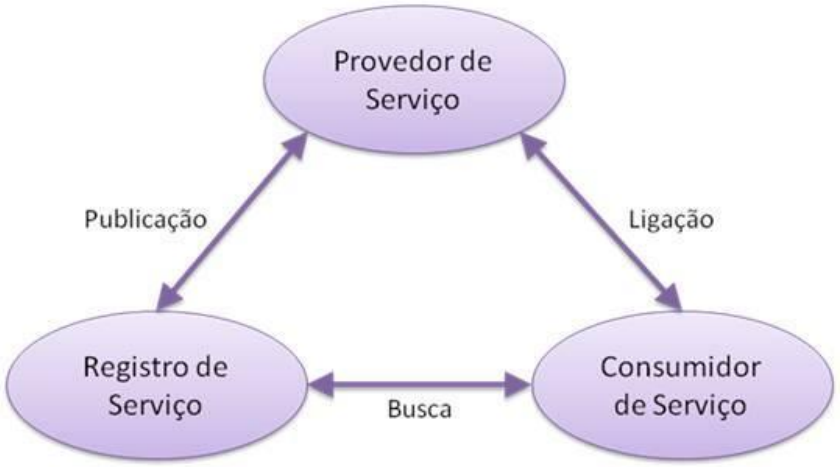
O processo de comunicação inicia com o cliente estabelecendo uma conexão de soquete e, em seguida, enviando uma sequência de mensagens de solicitação ao servidor Kafka. Este, por sua vez, processa as solicitações e envia as mensagens de resposta correspondentes. Notavelmente, o Kafka não requer um aperto de mão (*handshake*) na conexão ou desconexão, o que simplifica o processo. Embora o TCP beneficie da manutenção de conexões persistentes para amortizar o custo do *handshake* TCP, a conexão em si é considerada economicamente eficaz, tornando o Kafka uma solução prática para cenários que exigem troca rápida de mensagens em tempo real.

2.9.5. Arquitetura Orientada a Serviços (SOA)

Esse modelo de arquitetura baseado em serviços propõe um design no qual componentes pequenos são projetados para se comunicar por meio de contratos de interfaces definidos e estruturados, como serviços, promovendo um maior desacoplamento e, consequentemente, o reuso dos recursos disponíveis.

A arquitetura SOA se caracteriza, basicamente, pela interação de três tipos de agentes de software: os Provedores de Serviço, os Consumidores (ou Clientes) de Serviço e o Registro de Serviço, conforme a Figura 9:

Figura 9 – Design do modelo de arquitetura SOA



Fonte: Aplicativo de diagramação Lucidchart (2023).

Dentro do design geral de uma arquitetura, o Provedor é o responsável por disponibilizar o serviço e publicá-lo em um WSDL (linguagem de interface, baseada em XML, para descrever o serviço), quando tratamos dos serviços web. Além disso, após criar o WSDL, o Provedor poderá publicá-lo no Registro de Serviço, que funciona como um diretório de serviços, para que as aplicações interessadas em consumir este serviço possam identificar a sua disponibilidade e o provedor capaz de fornecê-lo. O Solicitante, então, irá propor, de maneira ativa, uma ligação com o Provedor, de maneira que, utilizando o protocolo SOAP (protocolo para comunicação estruturada entre aplicações em um ambiente distribuído), seja possível consumir os serviços assim disponibilizados.

No contexto da web, são definidas algumas estruturas (algumas já mencionadas) para que haja uma comunicação estruturada e consistente entre os sistemas. Esses padrões são baseados na tecnologia XML, como a camada de suporte que utiliza o *WS-Security*, por

exemplo, baseado em XML, assim como a definição dos serviços em WSDL e, também, o serviço de mensagem com o protocolo SOAP.

Essa abordagem remonta a vários princípios fundamentais no domínio da integração de aplicações distribuídas. A necessidade de particionar as responsabilidades, ou seja, a granularidade. Para isso, outro aspecto fundamental se faz necessário, que é o entendimento prévio do contexto de negócio, para que, tendo o entendimento do domínio, seja possível separar bem as responsabilidades, desacoplando as regras de negócio nos provedores de serviços corretos.

Vale analisar a correlação entre essa arquitetura tão mais antiga, remontando ao início dos anos 2000, quando a arquitetura SOA ganhou força com o surgimento da tecnologia XML, que é utilizado nas tecnologias SOAP (*Simple Object Access Protocol*) e WSDL (*Web Services Description Language*), para facilitar a comunicação entre os sistemas, e conceitos mais atuais como o DDD (*Domain Driven Design*), que trazem em sua essência a modelagem do software que é desenvolvido e distribuído de forma desacoplada e, quase que, independente.

Portanto, a orientação ao serviço se mostra uma abordagem que propõe contratos de interface estruturados, para possibilitar a comunicação distribuída entre partes de software desacopladas e com responsabilidades bem definidas, e entre sistemas externos, propondo a utilização de protocolos simples, documentados e que interagem por meio de mecanismos leves, geralmente uma API de recursos HTTP. Embora tenha perdido parte de sua popularidade para o microservices, ainda é relevante em alguns contextos.

2.9.6. Arquitetura REST

A arquitetura REST (*Representational State Transfer*) é um modelo criado para o desenvolvimento de sistemas distribuídos na web. Ela propõe um conjunto de restrições que devem ser aplicadas aos elementos da arquitetura. Roy Fielding, um dos principais arquitetos por trás do desenvolvimento do protocolo HTTP e uma figura central na conceituação da arquitetura REST, define em sua tese de doutorado o REST como "um conjunto de restrições arquiteturais que, quando aplicado como um todo, enfatiza a escalabilidade das interações de componentes, generalidade de interfaces, implantação independente de componentes e o uso de componentes intermediários para reduzir a latência de interação, reforçar a segurança e encapsular sistemas legados." As restrições que definem o estilo REST são:

Arquitetura cliente-servidor: A principal função dessa restrição é a separação das responsabilidades, um aspecto essencial quando avaliamos sob a ótica do Clean Code, conforme destaca MARTIN (2012). Isso indica uma arquitetura baseada em clientes, servidores e recursos, em que as solicitações são feitas via protocolo HTTP. Como enfatiza Fielding (2000), "talvez o mais significativo para a Web seja o fato de a separação permitir que os componentes evoluam de forma independente, apoiando assim os requisitos à escala da Internet de múltiplos domínios organizacionais."

Comunicação *Stateless*: A comunicação deve ocorrer de forma isolada e independente, sem armazenamento de nenhuma informação sobre transações antigas. A solicitação contém todos os dados necessários para ser atendida. O estado da sessão é, portanto, mantido inteiramente no cliente. Isso destaca a característica da aplicação REST de processar as solicitações a curto prazo.

Cache: Essa restrição requer que a resposta a uma solicitação ocorra de maneira implícita ou explicitamente rotulada como armazenável em cache ou não. Com o intuito de melhorar a eficiência na rede, ou seja, quando uma informação fica armazenada em cache, a comunicação entre cliente e servidor é otimizada.

Interface uniforme: A interface uniforme permite o desenvolvimento independente da aplicação entre usuário e servidor, proporcionando uma capacidade de evolução de forma desacoplada e com maior eficiência. Conforme Fielding (2000) destaca, "Ao aplicar o princípio de generalidade da engenharia de software à interface do componente, a arquitetura geral do sistema é simplificada e a visibilidade das interações é melhorada."

Sistema de camadas: As camadas do sistema devem possuir uma funcionalidade específica, sendo responsáveis por etapas diferentes dos processos de requisição do usuário e de resposta do servidor. A arquitetura é composta por camadas hierárquicas, restringindo a visibilidade de cada uma delas, para que as interações ocorram de forma organizada e coordenada. O objetivo é limitar a complexidade geral do sistema e promover a independência do substrato.

De maneira geral, essa abordagem adicionou um caráter de simplicidade à lógica do design por trás da arquitetura Web. Devido à adoção da interface padronizada e à aplicação ao protocolo de comunicação *Hypertext Transfer Protocol* (HTTP), este padrão simplifica o desenvolvimento dos componentes, reduz a verbosidade dos conectores, ou seja, melhora a semântica da comunicação, amplia a eficácia do ajuste no desempenho e promove o aumento da escalabilidade dos componentes no servidor.

Para a criação de uma API Web, são fornecidos alguns serviços dentro do modelo proposto pelo REST para as ações (verbos) expressas pelos métodos do HTTP (GET, POST, PUT, DELETE, etc). Para documentar e testar a Web API REST, uma ferramenta muito utilizada no mercado é o Swagger, um conjunto de ferramentas de código aberto construídas em torno da especificação *OpenAPI*, que ajuda a projetar, construir, documentar e consumir APIs REST.

Na Figura 10, é possível verificar um exemplo da interface gerada na criação do API REST para a compra e os respectivos métodos gerados.

Figura 10 – Documentação da API *ASP.NET-Core3.1-WEB-API-BASE*

Fonte: Ferramenta para documentação de api 's Swagger (2023).

A definição das interfaces de forma padronizada e pouco verbosa introduz uma maior facilidade no consumo dos serviços disponíveis. Pensando na arquitetura cliente-servidor, um dos requisitos propostos pelo criador do método de arquitetura REST, na qual o cliente solicita ao servidor as informações de interesse ou fornece os dados necessários ao servidor a fim de persistir dados nas bases, essa abordagem favorece uma comunicação mais leve em comparação a outros modelos anteriores, como o SOAP.

Além disso, torna ainda mais fácil a evolução dos produtos de maneira desacoplada, já que a API REST não precisa visualizar os clientes que irão consumir os serviços. Isso acontece de forma transparente, abstraindo assim as entidades que necessitam requisitar os recursos.

Segundo Fielding (2000), “Como o REST é direcionado especificamente para sistemas de informação distribuídos, ele vê uma aplicação como uma estrutura coesa de

informações e alternativas de controle por meio da qual um usuário pode executar uma tarefa desejada".

Dessa forma, a abordagem REST revela-se não apenas uma escolha prática, mas também alinhada com os objetivos de eficiência, simplicidade e evolução sustentável de sistemas distribuídos na era web. Em concordância com a visão de Fielding sobre sistemas distribuídos, a aplicação REST se delineou como uma estrutura coesa para o gerenciamento de informações e controle.

2.9.7. Arquitetura Monolítica

Essa abordagem está entre as mais antigas quando se trata do desenvolvimento de software para aplicações, em sua maioria, web, e ainda se mostra uma escolha eficiente, especialmente nas fases iniciais da evolução da tecnologia da informação. Isso é evidenciado pelo fato de que algumas das grandes empresas atualmente mantêm uma série de aplicações monolíticas, tais como *Stack Overflow, Shopify e GitHub*.

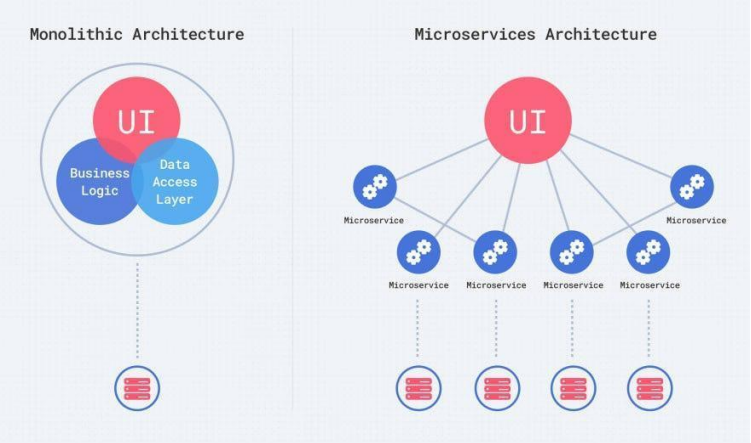
A primeira perspectiva que se pode analisar na definição de aplicações denominadas monolíticas é a forma como ocorre o deploy, ou seja, como é feita a publicação desse sistema. A arquitetura monolítica se caracteriza pelo empacotamento de toda a sua estrutura de código e dependências em um único pacote. Em outras palavras, para colocar a aplicação em produção, é criado apenas um único pacote que é adicionado ao servidor para a implantação do sistema.

Sob a ótica da arquitetura, observamos que essa abordagem de pacote único em produção permite a realização do deploy de uma vez só, eliminando a necessidade de coordenar deploys de múltiplas aplicações e/ou componentes, como ocorre nas aplicações distribuídas.

A arquitetura monolítica é constituída por uma única entidade de sistema na qual todos os módulos disponibilizados pela aplicação estão incluídos. Esse núcleo único é responsável por executar todas as tarefas necessárias, instanciando as classes, aplicando as regras de negócio e invocando serviços parceiros, englobando tudo que for necessário dentro do contexto de negócio. Em termos de arquitetura de software, de maneira geral, trata-se de uma abordagem que atende às necessidades quando as aplicações envolvidas são de tamanho reduzido. Isso ocorre porque o desenvolvimento, testes e implantação em sistemas menores são, em geral, mais simples.

No entanto, também é possível identificar desvantagens significativas, sobretudo ao considerarmos o crescimento acelerado das necessidades do mercado em relação a sistemas e produtos de software. Essas necessidades se traduzem no rápido acréscimo de novas funcionalidades e na demanda por uma maior velocidade nas entregas de código. Com isso, observa-se que os serviços dos sistemas monolíticos apresentam alto acoplamento, no qual as camadas do software dependem umas das outras, formando um único bloco que pode ser executado em um único processo.

Figura 11 – Esquema de arquitetura monolítica vs microsserviços



Fonte: https://medium.com/startlovingyourself/microservices-vs-monolithic-architecture-c8df91f16bb4

Essa característica impede a definição precisa das partes do sistema, uma vez que tudo está interconectado. Uma aplicação monolítica é aquela onde toda base de código está centralizada, isto é, a lógica de negócio, a conectividade com os bancos de dados e as interfaces formam um único bloco. Altamente vertical, essa abordagem faz com que a atuação frente às falhas do sistema se torne complexa e de difícil manutenção, já que, se uma camada falhar, toda a aplicação irá falhar.

Contudo, sua simplicidade traz consigo algumas vantagens na modelagem de software que a posicionam como uma opção viável e interessante para alguns contextos de projeto. Além disso, ela permite a agregação de tecnologias em um único processo executando todas as tarefas, permite, também, uma implantação menos robusta, em comparação com a arquitetura de microsserviços (conforme ilustrado na Figura 11), por exemplo, e, por fim, tende a ser um desenvolvimento mais rápido, devido ao fato de não haver um conjunto de repositórios distintos para se trabalhar.

3 CICLO DE VIDA DA INTEGRAÇÃO

O ciclo de vida refere-se às fases nas quais se divide o período de vida de uma integração de sistemas. Quando tratamos de sistemas, essa definição se aplica ao processo completo de desenvolvimento, implantação, operação e manutenção dos sistemas envolvidos, especialmente em ambientes que abrangem diversas tecnologias, plataformas e/ou linguagens.

Neste contexto, exploraremos as principais etapas, ou pelo menos aquelas que consideramos fundamentais, seguindo as principais referências em engenharia de software, arquitetura de sistemas e ciência da computação. Essa exploração é complementada por uma experiência de mercado adquirida em ambientes de desenvolvimento de software no setor de tecnologia voltado para integração de sistemas heterogêneos. As fases abordadas a seguir abrangem desde a concepção do produto de software, passando pelo design, desenvolvimento, implantação até as fases que sucedem a entrega do produto em produção, com o intuito de proporcionar uma abordagem abrangente e estratégica para a gestão eficaz da integração de sistemas heterogêneos.

Ao integrar as melhores práticas identificadas na literatura especializada com insights provenientes da experiência prática de mercado, almejamos fornecer um guia robusto que possa orientar profissionais e gestores de TI no planejamento, execução e manutenção de projetos de integração. Ao integrar as melhores práticas, almejamos fornecer um guia importante que possa orientar profissionais e gestores de TI no planejamento, execução e manutenção de projetos de integração.

3.1 Entendimento do processo de negócio

Antes de iniciar a construção de qualquer componente tecnológico, é fundamental que a equipe compreenda o contexto de negócios no qual a necessidade de integração está inserida. Isso contribuirá para a tomada de decisões quanto aos sistemas envolvidos, para uma modelagem mais precisa dos software e também para uma estimativa inicial do capacity necessário.

Ao iniciarmos o processo de design de uma integração é necessário um entendimento claro e objetivo do escopo de negócio. A construção de um design de produto juntamente com os envolvidos no projeto é fundamental para que essa atividade, desenvolvida em conjunto, esteja em alinhamento com as reais necessidades que serviram de iniciativa para o projeto.

Em consonância com a visão de estrategista de negócios, o autor e consultor Porter (2008) observa que "o entendimento profundo dos processos de negócios é a base para o desenvolvimento de estratégias de sucesso. Sem esse entendimento, a integração tecnológica é uma solução em busca de um problema, em vez de um meio de aprimorar processos de negócios essenciais". Kerzner (2022) enfatiza que "o entendimento detalhado dos processos de negócios é uma pedra angular para o sucesso dos projetos. É uma base sólida para o planejamento e a execução de projetos de integração eficazes."

O maior benefício alcançado durante esta fase, que é o entendimento do processo de negócios, está na construção de uma compreensão mútua entre os gerentes de projeto, responsáveis pelo planejamento e acompanhamento do projeto e seus entregáveis, e os clientes, usuários e *stakeholders*.

Conforme a abordagem do GAP (Gestão Ágil de Projetos), na falta de uma visão clara do produto, a natureza exploratória de um projeto ágil pode levá-lo a experimentações intermináveis. Por isso, as metas de negócio e a visão de produto devem ser suficientemente consistentes, ao passo que as incertezas serão reduzidas ao longo do projeto. Ainda segundo o GAP, a visão do produto promove a coesão da equipe, como explicado acima, exige o alinhamento entre a área de negócio e o time de projetos. “Sem uma visão consistente, corre-se o risco de desenvolver um projeto oscilante, com perspectiva desviada dos valores reais. Assim, o foco termina fixado em elementos de menor granularidade e relevância” Highsmith (2021).

Em conclusão, uma visão de produto clara é o farol que orienta a equipe na jornada de desenvolvimento. Ela assegura que todos entendam o 'porquê' por trás de suas tarefas e promove um ambiente de trabalho colaborativo e produtivo (Beck e Cohn 2004). O entendimento do processo de negócios é o alicerce sobre o qual o sucesso da integração de sistemas é construído. Essa compreensão não apenas alinha as partes envolvidas no projeto, mas também fornece uma base sólida para a definição de metas de negócio claras e a visão de produto, garantindo que a integração tecnológica esteja estrategicamente alinhada com as reais necessidades de negócio.

3.2 Identificação dos sistemas envolvidos

Depois de uma compreensão definida a respeito da visão de produto, ou seja, do entendimento sobre onde queremos chegar, faremos uma análise sobre os sistemas envolvidos na integração para que se possa modelar a melhor arquitetura dentro do nosso contexto de

sistema. Nesta etapa, avaliamos todo o ecossistema com o qual iremos trabalhar, definindo os sistemas, APIs, ambientes e infraestruturas que compõem o nosso contexto de projeto.

Nas palavras de, Lines e Ambler (2012), "a identificação dos sistemas envolvidos é o primeiro passo para uma integração de sucesso. É a fundação sobre a qual toda a estrutura de integração é construída, permitindo que as diferentes partes do sistema trabalhem em conjunto de maneira harmoniosa."

A integração de sistemas é um processo complexo que frequentemente envolve múltiplos atores e componentes. Portanto, é essencial identificar e mapear todos os sistemas que participam direta ou indiretamente na troca de informações. Isso inclui tanto sistemas internos da organização como sistemas externos de parceiros ou fornecedores. A integração de sistemas é uma disciplina interdisciplinar que exige uma visão abrangente de todas as partes envolvidas. A identificação precisa dos sistemas é o primeiro passo na orquestração de um ambiente de integração eficiente (BERNSTEIN, 2009).

Portanto, é necessário identificar quais sistemas ou interfaces possibilitam o acesso a esses dados dentro desse conjunto de repositórios de informações, ou seja, quais mecanismos sistêmicos nos permitem obter as informações necessárias para o nosso contexto de negócios. Em resumo, para realizar essa identificação de forma eficaz, é fundamental responder a duas perguntas-chave:

- *Quais sistemas devem interagir para que o requisito de negócio seja alcançado?*
  Essa pergunta ajuda a estabelecer a relação entre os sistemas e a compreender quais deles são essenciais para atender às necessidades de negócio. Isso envolve a identificação de sistemas que são fontes de dados, sistemas de processamento, sistemas de armazenamento, entre outros.
- *Quais são os sistemas especialistas na informação que se deseja obter, processar ou armazenar?*

  Aqui, concentramos nossa atenção nos sistemas que contêm as informações críticas para o negócio. Isso inclui a identificação de bancos de dados, APIs ou outras interfaces que fornecem acesso às informações necessárias para a integração.

É importante notar que a identificação dos sistemas envolvidos vai além de apenas listar seus nomes. Também envolve a compreensão dos padrões de comunicação, segurança, protocolos e requisitos de desempenho associados a cada sistema. Essas informações são fundamentais para a definição de uma arquitetura de integração robusta e eficiente. A integração de sistemas bem-sucedida requer um profundo entendimento dos

sistemas envolvidos e a capacidade de projetar soluções que garantam uma comunicação segura e eficaz (O'Neill 2012).

Portanto, essa etapa requer o conhecimento das aplicações, serviços ou módulos de sistemas capazes de fornecer os recursos necessários. Essa ação se caracteriza, dentro do ciclo de vida da integração, pelo envolvimento dos analistas de negócio e sistemas para o levantamento das soluções que farão parte do projeto de integração.

3.3 Desenho do fluxo de dados entre os sistemas

Tendo feito o levantamento dos sistemas envolvidos na integração, precisaremos, então, definir o tráfego dos dados necessários para que se obtenha o produto de software. Ao desenhar este fluxo, precisaremos mapear a origem e o destino dos dados para cada uma das entidades de sistema envolvidos. Em outras palavras, é necessário identificar de onde esses dados serão extraídos, como serão transformados ou processados e para onde serão direcionados.

Nesta fase do processo de integração, não apenas delineia o movimento de informações, mas também estabelece as bases para uma comunicação transparente entre sistemas heterogêneos. Este estágio também compreende a determinação da sequência em que cada fluxo de dados deve ocorrer. Isso assegura que os sistemas interajam de maneira ordenada e eficiente, evitando conflitos e garantindo que as informações sejam transmitidas no momento certo e no formato adequado.

Conforme indicado por Fowler (2002), a sequência do fluxo de dados é crucial para evitar problemas de concorrência e garantir que a informação seja processada em uma ordem lógica. Ele argumenta que "a definição clara da sequência do fluxo de dados é a chave para evitar inconsistências e falhas na integração."

Durante essa fase, não é necessário adentrar na definição dos campos de dados específicos. Em vez disso, o foco está na concepção da estrutura geral do fluxo de dados e na compreensão da dinâmica do processo de integração. Essa abordagem ajuda a estabelecer as bases necessárias para um projeto de integração, antes de entrar em detalhes mais específicos, como a formatação dos dados ou as regras de validação.

O objetivo desta etapa consiste em orientar a forma como os dados fluem entre os diversos componentes do sistema. Ele define a arquitetura de comunicação que possibilita a coordenação eficiente das ações entre sistemas distintos. Além disso, é importante destacar que o desenho do fluxo de dados pode revelar oportunidades para otimização e melhoria do

desempenho da integração. À medida que os detalhes do fluxo são delineados, podem surgir insights sobre a possibilidade de paralelizar tarefas, reduzir latências ou simplificar processos.

Como afirmado por Seroter (2009), "O desenho cuidadoso do fluxo de dados é o alicerce para uma integração bem-sucedida e a realização de projetos de software de alto desempenho." Portanto, o desenho do fluxo de dados é uma etapa crítica na integração de sistemas, pois define como as informações são trocadas e coordenadas entre os sistemas envolvidos. A clareza na concepção do fluxo de dados estabelece as bases para uma integração eficiente e bem-sucedida, permitindo que os sistemas funcionem em harmonia e atinjam os objetivos da integração de maneira eficaz.

3.4 Definição dos contratos de interface

É necessário definir as interfaces e/ou APIs que serão desenvolvidas para a integração, incluindo o seu propósito, nomenclatura e os dados de entrada e saída. Nesta etapa, também realizamos o detalhamento dos campos envolvidos na integração, estabelecendo efetivamente um contrato que permite a comunicação entre os sistemas por meio de uma interface específica. Além disso, é crucial especificar o tipo de dado que será transmitido por essa interface, que pode incluir informações em formato de texto, data, arquivos e outros.

A definição dos contratos de interface é uma parte fundamental do processo de integração de sistemas, pois serve como o ponto de contato e intercâmbio de informações entre sistemas heterogêneos. Essa definição estabelece as diretrizes que governam como os sistemas vão se comunicar e compartilhar dados, garantindo a consistência e a interoperabilidade necessárias.

Além do propósito e da estrutura da interface, é vital documentar a semântica e a sintaxe dos dados que serão trocados. Isso implica na descrição detalhada dos campos, estruturas de dados e padrões de comunicação que devem ser seguidos. Essa documentação precisa ser clara e precisa, a fim de evitar mal-entendidos e garantir uma integração bem-sucedida.

Outro aspecto relevante é a especificação dos protocolos de comunicação a serem utilizados, como REST, SOAP, gRPC, entre outros. Isso assegura que os sistemas envolvidos possam compreender a forma como as solicitações e respostas serão formatadas e transmitidas.

Portanto, a definição dos contratos de interface estabelece as regras, os formatos e os protocolos necessários para que diferentes sistemas possam cooperar de maneira eficiente e confiável. A qualidade e a clareza desses contratos desempenham um papel fundamental no sucesso da integração e na criação de sistemas interconectados que funcionam em harmonia.

3.5 Definição dos serviços

O foco principal é a compreensão detalhada dos serviços que serão responsáveis por implementar os contratos de interface previamente delineados. Isso implica em uma análise das funcionalidades específicas que cada serviço executará dentro do contexto da integração. Para atingir esse objetivo, é essencial realizar um levantamento abrangente dos fluxos de orquestração, mapeando cuidadosamente como os serviços serão acionados, coordenados e interconectados ao longo do processo de integração. Além disso, é fundamental identificar possíveis agregações de informações que podem ocorrer durante a execução dos serviços, garantindo que todos os dados necessários sejam corretamente reunidos e processados.

Outro aspecto relevante é a consideração das rotas alternativas que podem surgir durante a integração. Isso envolve a antecipação de cenários não triviais, como situações de exceção ou erros potenciais, e a definição de como os serviços lidarão com essas condições alternativas de maneira eficaz.

- Análise Detalhada das Funcionalidades dos Serviços: Inclui a identificação de entradas e saídas, a descrição das operações que cada serviço pode realizar e a definição clara das suas responsabilidades na troca de informações.
- Levantamento Abrangente dos Fluxos de Orquestração: É fundamental mapear como os serviços serão acionados, coordenados e interconectados ao longo do processo de integração. Isso permite uma visão geral do fluxo de trabalho da integração e a identificação de possíveis gargalos e ineficiências.
- Identificação de Agregações de Informações: Identificar possíveis agregações de informações que podem ocorrer durante a execução dos serviços permite que todos os dados necessários sejam corretamente reunidos, processados e entregues aos sistemas ou processos finais, evitando lacunas ou duplicações de dados.
- Rotas Alternativas e Exceções: Exige a definição de como os serviços lidarão com as condições alternativas de maneira eficaz. É fundamental elaborar estratégias para o

tratamento de erros, o controle de transações e a recuperação de falhas, garantindo a robustez e a confiabilidade da integração.

- Segurança e Controle de Acesso: A definição dos serviços deve considerar os mecanismos de autenticação, autorização e criptografia necessários para proteger os dados sensíveis que estão sendo compartilhados entre sistemas. Além disso, o controle de acesso e a gestão de identidades desempenham um papel crucial na garantia da integridade e da confidencialidade dos dados.

3.6 Configuração dos Ambientes

Essa fase, pretende estabelecer as bases para a comunicação eficaz e segura entre os sistemas envolvidos. Para isso, uma série de etapas é realizada para garantir que os recursos estejam prontos para atender às demandas do projeto e que a integração ocorra sem problemas.

Além disso, a configuração dos ambientes inclui a definição de rotas entre os *pipelines* da plataforma de integração e os recursos, como máquinas, endereços IP e portas, que estão sendo utilizados pelos serviços. Essas rotas garantem que os dados fluam de maneira eficiente entre os sistemas, evitando gargalos de comunicação e assegurando que os recursos certos sejam acessados nos momentos apropriados. Algumas tarefas essenciais estão envolvidas na preparação do ambiente, como: Configuração de *Firewalls* e Segurança de Rede, Gerenciamento de Certificados Digitais, Configuração de Servidores e Recursos de Banco de Dados, Controle de Acesso e Autenticação, Monitoramento e Registro de Atividades, *Backup* e Recuperação de Dados, Documentação e Políticas de Segurança.

Em resumo, a configuração dos ambientes é uma etapa crítica na preparação para a integração de sistemas. Garantir que a comunicação entre os sistemas ocorra de forma eficaz e segura requer uma abordagem detalhada que envolve medidas de segurança, gerenciamento de recursos e políticas de acesso. Essa fase estabelece as bases para o sucesso da integração e a proteção dos dados e sistemas envolvidos.

3.7 De-Para / Diagrama de Sequência

Nesta fase, avançamos para a segunda iteração do diagrama de sequência, agora com os endpoints das APIs/serviços claramente definidos. Isso permite criar um mapeamento preciso de como os diferentes componentes se comunicam durante o processo de integração.

Além disso, é o momento de definir se a integração será síncrona ou assíncrona, o que influenciará a maneira como os dados são transmitidos entre os sistemas envolvidos. A planilha de De-Para também é preenchida com detalhes específicos, incluindo a definição campo a campo das origens e destinos, bem como as transformações necessárias para garantir a compatibilidade e a consistência dos dados.

- Diagrama de Sequência Real: Define uma representação visual das interações entre os sistemas e componentes envolvidos. Ele mostra a ordem e o fluxo das mensagens, solicitações e respostas entre os sistemas, destacando como os dados fluem através da integração.
- Síncrono ou Assíncrono: Uma integração síncrona envolve a comunicação em tempo real, onde um sistema aguarda uma resposta imediata do outro antes de continuar. Por outro lado, uma integração assíncrona permite que os sistemas continuem operando de forma independente, trocando informações quando estiverem disponíveis. A escolha entre síncrono e assíncrono depende das necessidades de negócios e dos requisitos de desempenho.
- Planilha de De-Para: Descreve a correspondência entre os campos de origem e destino, indicando como os dados são mapeados de um sistema para o outro. Além disso, a planilha inclui informações sobre as transformações necessárias para garantir a compatibilidade e a consistência dos dados. Isso pode envolver a conversão de formatos de dados, a validação de dados, a agregação de informações e outros processos de transformação.
- Escalabilidade e Desempenho: Visa garantir que a integração seja capaz de lidar com cargas de trabalho crescentes e que o sistema seja dimensionado adequadamente para suportar a demanda esperada.

3.8 Construção

Com todos os detalhes definidos, a construção do *pipeline* dentro da plataforma de integração é iniciada. Esse processo envolve a implementação dos componentes e lógicas definidos no diagrama de sequência, bem como a configuração das APIs e serviços necessários para garantir a integração eficaz entre os sistemas. Durante esse estágio, os detalhes técnicos e os componentes previamente definidos são transformados em realidade.

A construção envolve a implementação de todos os componentes e lógicas previamente definidos no diagrama de sequência ou no design do *pipeline* de integração. Isso

inclui a escrita de código, a configuração de regras de negócios, a criação de transformações de dados e a programação das interações com sistemas externos. Além disso, é preciso incluir a definição de *endpoints* de API, a configuração de autenticação e outras configurações técnicas que permitem a comunicação e a troca de informações entre os sistemas.

3.9 Testes

Essa fase envolve a verificação das funcionalidades, identificação e correção de erros, bem como a validação do desempenho e da qualidade do software. A preocupação aqui é com os aspectos que garantirão uma entrega mais confiável do produto.

É fundamental a escrita dos testes unitários (testes de "caixa branca") que se concentram na verificação detalhada do funcionamento de cada componente ou módulo do pipeline de integração. Eles são projetados para identificar possíveis problemas ou erros em nível de código, permitindo uma verificação minuciosa do fluxo do pipeline, desde a entrada até a saída. Os testes unitários são executados em ambientes controlados e podem ser automatizados, o que agiliza a detecção de problemas e contribui para uma abordagem de desenvolvimento ágil.

Por outro lado, os testes de integração envolvem a validação da integração entre mais de um pipeline. Eles simulam cenários de negócios reais, permitindo verificar se a integração entre os diferentes componentes funciona de maneira harmoniosa. Esses testes são projetados para garantir que os pipelines se comportem conforme o esperado quando trabalham juntos para atender aos requisitos de negócios. Em suma, seu objetivo consiste na validação dos módulos, que já passaram pela validação unitária, e agora são testados em grupo para garantir que não haja nenhuma quebra naquilo que foi feito individualmente e que, agora, está sendo integrado.

Para garantir a abrangência dos testes integrados, é fundamental ter uma massa de dados mínima de cenários de teste já produzida. Essa massa de dados realistas ajuda a identificar problemas que podem não ser evidentes em testes isolados. Eles são projetados para simular situações do mundo real, garantindo que a integração funcione em condições próximas às que serão encontradas em produção.

Com a ampla adoção do modelo ágil na produção de sistemas e a crescente demanda por sistemas em larga escala, a construção e entrega de software ocorrem em uma velocidade cada vez maior. No entanto, é importante não confundir entrega rápida com falta

de qualidade. Ou seja, não é sustentável focar apenas na velocidade de entrega, negligenciando as rotinas de teste essenciais.

Uma abordagem que vem sendo bastante utilizada no mercado é o *Domain Driven Development* (DDD), que consiste em uma abordagem que oferece uma estrutura para a tomada de decisões no processo de design e desenvolvimento de software. Essa metodologia, centrada na lógica de negócios, ou no domínio, têm como objetivo principal melhorar todo o ciclo de desenvolvimento, permitindo a criação de aplicações que refletem um profundo entendimento dos processos e das regras do negócio. A história do DDD remonta ao livro escrito por Eric Evans em 2003.

O Domínio representa o coração do negócio em que estamos trabalhando, com todas as suas regras e peculiaridades que o DDD visa atacar. Esse aspecto do desenvolvimento orientado ao domínio está alinhado com a ideia de alcançar uma cobertura de testes mínima para a entrega de um projeto de software. Ao definir adequadamente a camada de domínio, que engloba as regras de negócio da aplicação de forma modularizada, cria-se um código altamente testável. Isso proporciona a capacidade de escrever testes unitários que isolam cada regra, conforme necessário, garantindo uma solução mais confiável e, acima de tudo, mais próxima do objetivo final de entrega de valor.

Outro aspecto que vem impactando significativamente os testes em integração de sistemas no cenário atual de desenvolvimento de software é a adoção das esteiras de integração e entrega contínua, conhecido como CI/CD. Essa abordagem visa automatizar os processos de implantação e entrega de código em produção de maneira eficiente e rápida.

O principal valor proporcionado por essa abordagem é a capacidade de incluir novas funcionalidades, corrigir erros descobertos em produção ou até mesmo adicionar novos módulos ao sistema de forma automatizada, sem a necessidade de passar por longos processos de aprovação, validações manuais, formulários de gestão de mudança, configurações complexas de servidores, e assim por diante. Essa abordagem é amplamente adotada no contexto da gestão ágil e é complementada por diversas ferramentas e tecnologias do mercado voltadas para a automação de testes. O que a torna ainda mais essencial é a integração das rotinas de testes, desde as validações realizadas no ambiente de desenvolvimento, graças às pipelines configuradas para executar esses testes a cada nova versão da aplicação.

3.10 Definição do Capacity

A definição do *Capacity* (capacidade) é uma etapa crítica na preparação para a implantação em produção de uma integração. Antes da implantação em produção, é essencial definir a capacidade que a integração deverá suportar. Isso inclui determinar quantas transações por segundo serão processadas, qual o tamanho dos payloads e qual é a criticidade para o negócio. Com base nessas informações e nos resultados dos testes realizados, serão configuradas a quantidade de réplicas do pipeline e seu tamanho para atender às demandas operacionais. Aqui estão os principais aspectos relacionados à definição do *capacity*:

- Transações por Segundo (TPS): Compreende determinar quantas operações ou solicitações a integração deve ser capaz de lidar em um intervalo de tempo específico. Essa métrica é fundamental para dimensionar a infraestrutura e a capacidade da integração.
- Tamanho dos Payloads: Além do número de transações, é importante considerar o tamanho dos payloads, ou seja, o volume de dados que será transferido em cada transação. Payloads maiores exigem mais recursos de processamento e largura de banda.
- Criticidade para o Negócio: Determinar o grau de criticidade ajuda a priorizar a alocação de recursos e a definir os acordos de nível de serviço (SLAs) apropriados. Integrações críticas para o negócio podem exigir configurações de alta disponibilidade e recuperação de desastres mais robustos.
- Testes de Desempenho: Os testes fornecem dados valiosos sobre como a integração se comporta sob carga. Com base nessas informações, é possível ajustar o número de réplicas do pipeline, o dimensionamento dos servidores e outros recursos para atender às demandas operacionais de forma eficaz.

Com base na definição do *capacity*, podem ser configuradas a quantidade de réplicas do pipeline e o tamanho de cada instância. Isso permite distribuir a carga de trabalho e melhorar a escalabilidade. O balanceamento de carga entre as réplicas do pipeline é crucial para garantir o desempenho e a alta disponibilidade da integração.

É importante destacar que essa definição não é um processo estático. À medida que as demandas operacionais evoluem e a carga de trabalho varia, é necessário realizar ajustes contínuos na capacidade para garantir que a integração continue atendendo às expectativas. Compreender as necessidades de carga de trabalho, a criticidade para o negócio

e os resultados dos testes de desempenho permite configurar adequadamente a infraestrutura e os recursos para garantir que a integração funcione de maneira eficaz e confiável.

3.11 Implantação e Governança

Após a validação dos testes, a integração entra na fase de implantação e governança. Com todas as verificações concluídas, o deploy em ambiente de produção é realizado. Durante essa fase, várias etapas são realizadas para garantir que a integração seja implementada com sucesso e que atenda aos requisitos e expectativas estabelecidos. Aqui estão os principais aspectos da fase de implantação:

- Validação Final dos Testes: Compreende garantir que todos os casos de teste tenham sido executados com êxito, que as anomalias e os problemas tenham sido resolvidos e que a integração esteja funcionando conforme o esperado.
- Sign-off da Área de Negócio: Indicação de que a solução atende às necessidades e expectativas dos stakeholders e está pronta para ser implantada em produção.
- Verificação da Capacidade de Carga: Compreende a análise de desempenho, a capacidade de dimensionamento e a garantia de que os recursos de hardware e software estão configurados adequadamente para atender às demandas da produção.
- Deploy em Ambiente de Produção: Inclui a implantação da integração em servidores de produção e a configuração dos ambientes para suportar a execução eficaz da solução. É crucial que o deploy seja feito com cuidado e monitorado de perto para evitar problemas de indisponibilidade ou degradação de desempenho.
- Monitoramento e Garantia de Qualidade Contínua: Envolve a rápida identificação e resolução de problemas, ajustes de desempenho e aprimoramentos conforme necessário.

Garantir uma implantação bem-sucedida exige uma abordagem cuidadosa, colaboração entre as equipes de desenvolvimento, operações e negócios, além de um acompanhamento constante para manter a integração em funcionamento com eficácia.

Por fim, algumas etapas, denominadas de governança, envolvem atividades que visam à conformidade, ao controle e ao desempenho do projeto de integração após sua implantação em produção. Para isso, é necessário monitorar de forma contínua, revisar periodicamente o código, adotar medidas para otimização de desempenho e implementar ações corretivas. Essas ações têm como objetivo agir de modo preventivo para corrigir falhas

no processo, garantindo que a integração continue funcionando de maneira eficaz e alinhada com as necessidades do negócio.

Aqui estão alguns aspectos-chave da governança em projetos de integração:

- Monitoramento Contínuo: Compreende a coleta de métricas e a análise de indicadores-chave de desempenho (KPIs). O monitoramento permite identificar possíveis problemas e tendências, fornecendo informações valiosas para a tomada de decisões e a otimização do sistema.
- Revisões de Código: Abrange a avaliação rigorosa do código fonte, identificando erros, garantindo a conformidade com padrões de codificação e promovendo boas práticas de desenvolvimento. As revisões de código ajudam a manter a qualidade do software e a evitar a introdução de vulnerabilidades.
- Otimização de Desempenho: Compreende a identificação e resolução de gargalos, a implementação de melhorias de código e a seleção de soluções de integração mais eficientes. A otimização contínua é fundamental para garantir que o sistema possa lidar com a crescente demanda.
- Ações Corretivas: A governança também envolve ações corretivas em resposta a problemas identificados durante o monitoramento, revisões de código ou avaliações de desempenho. Essas ações visam resolver questões rapidamente, minimizando o impacto nos usuários finais e garantindo a integridade do sistema.

Em resumo, a governança em projetos de integração é um processo holístico que garante a qualidade, a conformidade e o desempenho contínuo do sistema de integração. Ao adotar uma abordagem de governança sólida, as organizações podem garantir que suas integrações sejam bem-sucedidas e eficazes em um ambiente cada vez mais complexo e dinâmico, conforme avaliado anteriormente.

4 PROJETO DE INTEGRAÇÃO: APLICAÇÃO PARA O MERCADO DE TECNOLOGIA

É fundamental a compreensão da arquitetura de software em relação às necessidades de negócios e aos desafios relacionados encontrados na concepção de um fluxo capaz de atender eficazmente, levando em consideração os possíveis riscos, mas baseando-se nas estratégias adotadas, nas experiências vivenciadas, nas metodologias e padrões fornecidos pelo mercado, e, não menos importante, nos tópicos mencionados neste trabalho (os pré-requisitos para uma integração de software "saudável") que servem como orientação para a construção de uma visão mais sólida e fundamentada diante de tantas incertezas inerentes ao desenvolvimento de uma solução de integração.

Com o objetivo de aprofundar a análise sobre os aspectos que compõem uma integração de sistemas heterogêneos, vamos explorar um exemplo prático no mundo empresarial. O contexto, o problema a ser tratado e as soluções são detalhadamente descritos neste capítulo.

4.1. Contextualização

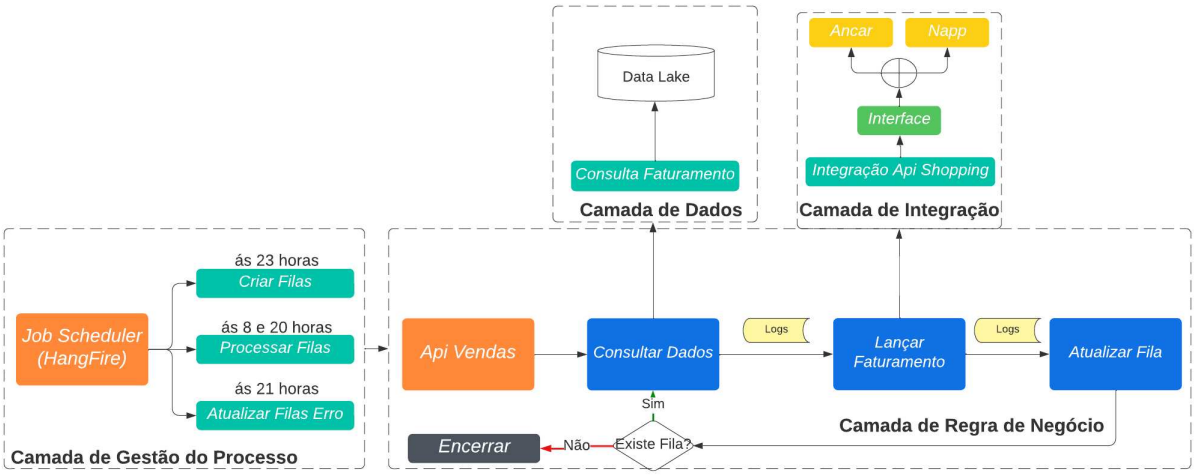
No âmbito de uma equipe de desenvolvimento voltado para o atendimento de uma grande empresa do setor varejista no Brasil, surgiu uma necessidade específica para as filiais localizadas em redes de *shopping centers* credenciados. Essas redes mantinham unidades distribuídas em todo o país (um aspecto de negócio relevante que abordaremos posteriormente, em relação à escalabilidade e desempenho) e possuíam uma política de aluguel baseada em uma porcentagem da receita mensal de cada filial. Isso demandava uma prestação de contas diária das vendas, chegando ao nível de detalhamento dos cupons fiscais, a fim de permitir comprovação e auditoria. Portanto, para enfrentar esse desafio e compreender completamente o escopo do negócio, foi necessário desenvolver uma solução de automação para realizar o registro diário das vendas. Essas ações são controladas e orquestradas por um *scheduler* (agendador) que coordena as ações necessárias.

4.2. Apresentação das aplicações envolvidas

A principal diretriz do desenho arquitetural, apresentado na Figura 12, consistiu na clara divisão das responsabilidades, com a separação das regras de negócio em uma API

independente. Esta API foi incumbida de definir os contratos de interface, efetuar o mapeamento das entidades de negócio e implementar todo o domínio da aplicação.

Figura 12 – Diagrama Arquitetural do Projeto de Integração de Vendas



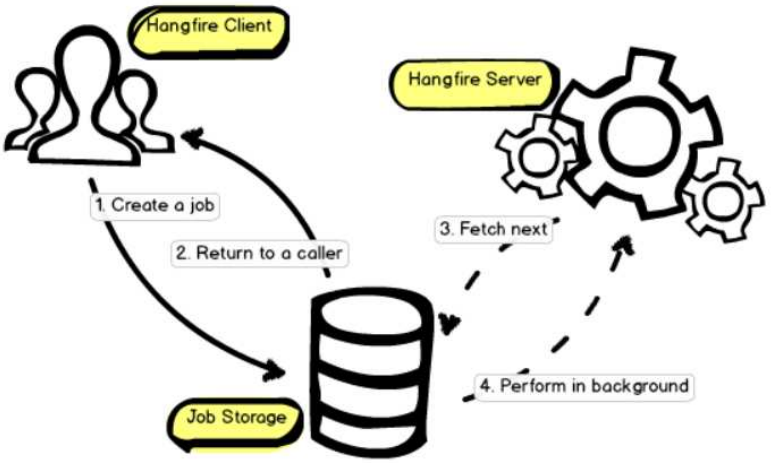
Fonte: Aplicativo de diagramação Lucidchart (2023).

Outra camada foi destinada ao gerenciamento de dados. Sua função primordial foi abstrair, na medida do possível, as bases que armazenavam as informações das vendas realizadas pelas filiais. Mesmo diante do considerável volume de transações no banco de dados transacional corporativo, onde todas as vendas de mais de 1500 lojas eram registradas, optou-se por executar as operações de leitura nessa base de dados relacional que detinha as informações de vendas.

A camada de integração era a responsável pela interação com os sistemas externos, no caso, as APIs das redes credenciadas de *shoppings*. Essas APIs foram desenvolvidas com interfaces predefinidas para padronizar a comunicação entre os dois sistemas: o Integrador de Vendas (desenvolvido pela empresa de varejo, com filiais nas redes) e a API Externa (desenvolvida pelas redes credenciadas de *shoppings*). O principal papel desse serviço era implementar o mapeamento de dados e configurar as chamadas aos endpoints das APIs/serviços, assegurando a compatibilidade e a consistência dos dados nas transações.

Por fim, uma camada de gestão de processos, um *Job Scheduler*, foi desenvolvida para orquestrar e ativar as integrações. Para isso, foi utilizado uma ferramenta desenvolvida em .NET chamada Hangfire, uma estrutura de código aberto que ajuda a criar, processar e gerenciar seus trabalhos em segundo plano, como apresentado na Figura 13.

Figura 13 – Modelo de integração com *Job Scheduler* Hangfire



Fonte: Aplicativo de diagramação Lucidchart (2023).

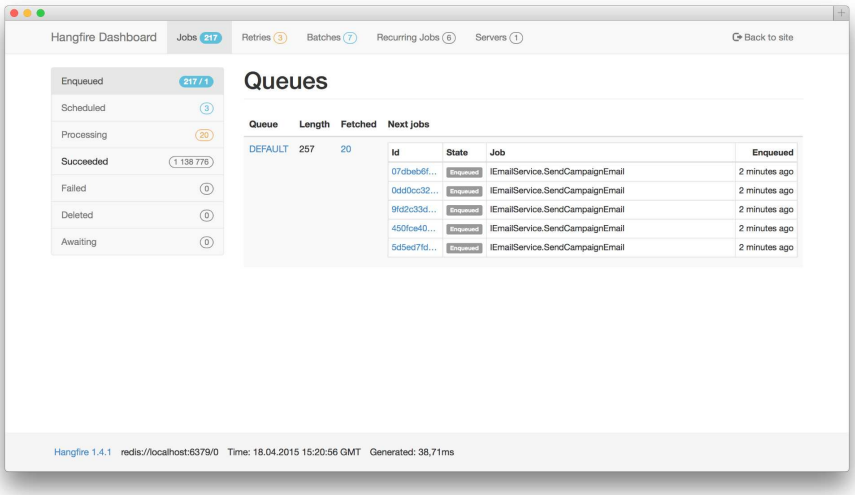
O objetivo aqui foi separar o controle das operações, concedendo à área de negócios maior capacidade de gerenciar e monitorar os processos no dia a dia. Isso reflete o princípio de transparência e autonomia para os responsáveis pelo negócio. Vale ressaltar que nesta camada foram adotadas algumas tecnologias para visualização das integrações. Além disso, foram incluídas opções para acionamento manual em caso de falhas na integração ou intermitência nos servidores ou na rede.

4.3. Avaliação na perspectiva da integração de sistemas

É notável a presença de, pelo menos, duas integrações fundamentais: entre o módulo de gestão de processos e o módulo de regras de negócio, e entre o módulo de integração e as APIs/serviços externos.

i) A gestão dos processos foi isolada em um módulo único devido à necessidade de modularização e à facilitação, em primeiro plano, da inclusão e/ou exclusão de serviços no processo atual. Além disso, ela visa aprimorar a manutenção de possíveis alterações nas regras de lançamento das integrações, bem como em relação aos agendamentos e ordenamentos das tarefas já estabelecidas, conforme representada na Figura 14. Adicionalmente, observamos que esse módulo, a gestão de processos, que funciona como um sistema de agendamento, integrando os serviços em segundo plano e disponibilizando painéis para o monitoramento do negócio, pode ser reaproveitado em qualquer outro contexto de negócio que demande um *Cron Job*, o qual é um gerenciador de tarefas repetitivas por meio de processos automatizados.

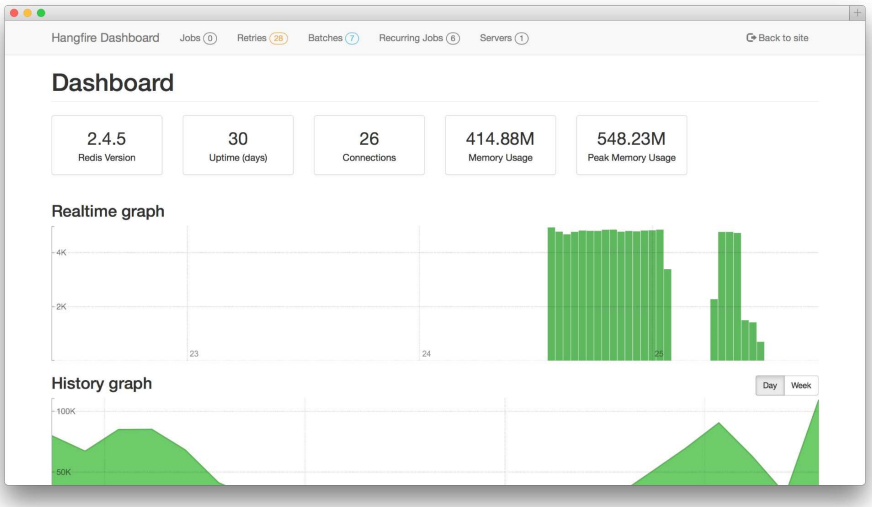
Figura 14 – Agendamento das filas de tarefas no Hangfire



Fonte: Aplicativo de diagramação Lucidchart (2023).

Assim, foi possível acompanhar as tarefas de criação das filas de processamento, além das requisições enviadas à API de vendas com o intuito de disparar o processo de integração com as interfaces dos serviços externos. A Figura 15 ilustra como o monitoramento desses processos foi feito mediante o *Dashboard* dos *jobs* disponibilizado pela plataforma do Hangfire.

Figura 15 – Hangfire Dashboard



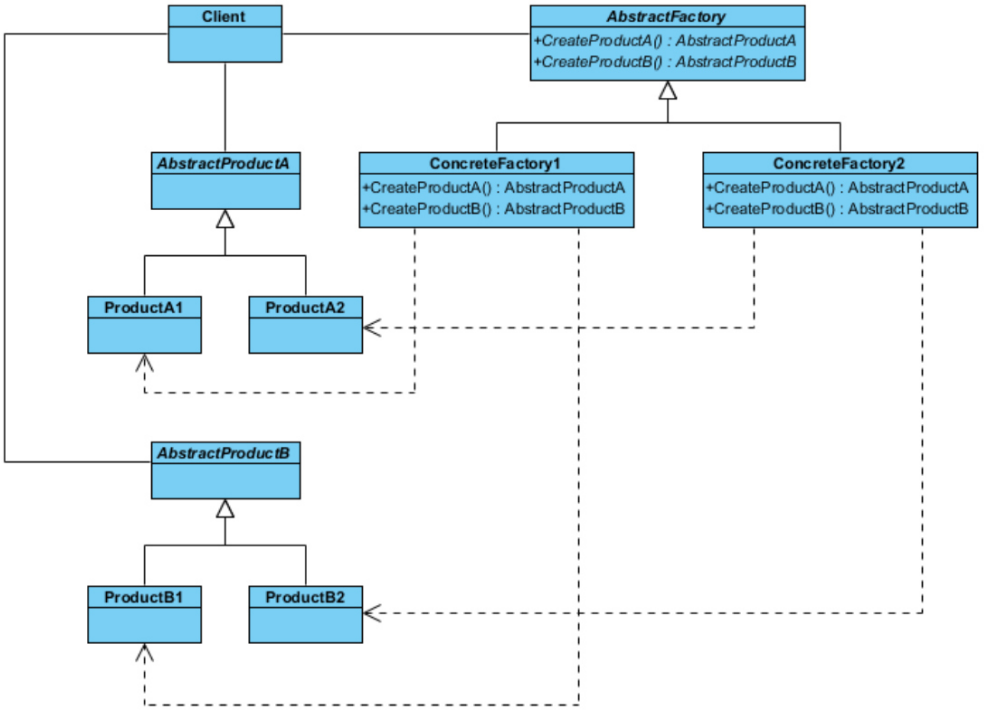
Fonte: Aplicativo de diagramação Lucidchart (2023).

ii) A camada de integração foi projetada com o objetivo de estabelecer uma divisão clara entre as especificidades de cada serviço externo, considerados, neste contexto, como

regras de negócio. Portanto, essas regras de negócio foram implementadas na camada de negócio, enquanto a camada de integração ficou responsável pela construção dos clientes REST para integrar os dados por meio de objetos JSON, bem como pela infraestrutura do protocolo de comunicação HTTP. Dessa forma, foi definida uma única interface que recebe como parâmetro objetos já mapeados do serviço externo correspondente, que, neste caso prático, consiste em dois serviços, representando duas redes de shoppings credenciadas e independentes, cada uma com APIs e interfaces de dados distintas.

Para tanto, a construção do módulo foi orientada pelo padrão de projeto Abstract Factory, Gamma, Helm, Dr e Vlissides (1994), que é um padrão de projeto dito criacional, fornecendo uma interface que delega as chamadas de criação a uma ou mais classes concretas para entregar objetos específicos, como ilustrado pela Figura 16.

Figura 16 – Diagrama de classes UML para o padrão de projeto *Abstract Factory*



Fonte: Visual Paradigm Community Circle (2023).

O objetivo do Abstract Factory é fornecer uma interface para a criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas, permitindo a criação de todas as variantes dos produtos de acordo com essas interfaces pré-definidas. Com isso, foi possível desenvolver cada objeto especificado e documentado por cada serviço externo, implementando uma interface única para as operações de venda.

4.4. Discussões sobre as implementações e resultados

Dessa maneira, além de fornecer uma solução viável para o atual contexto de negócios, dentro do escopo definido para este projeto, a arquitetura e o padrão de projeto implementados proporcionam uma modularização que pode ser facilmente incorporada a outros serviços de terceiros que possam ser adicionados no futuro. Isso visa a um compromisso não apenas com a manutenção, mas também com a evolução e a escalabilidade do produto. Logo, quando surgir uma nova rede de *shoppings* que cobra seus aluguéis de forma proporcional ao faturamento dessas lojas, será muito mais fácil incluir o serviço devido à forma como a solução foi desenvolvida.

4.5. Análise Teórica do Projeto: A Perspectiva das Etapas do Ciclo de Vida

Pretende-se realizar uma análise sob a perspectiva das etapas do ciclo de vida da integração, discutidas durante a fundamentação teórica, aplicadas em um projeto de integração de faturamentos para uma empresa do ramo de varejo no Brasil.

Com essa análise, busca-se efetivamente elencar as atividades desenvolvidas em cada etapa, destacar as lições aprendidas e apresentar os respectivos entregáveis. O objetivo é, após essa avaliação, identificar concretamente os impactos que cada fase pode provocar em um projeto de integração de software. Esse processo permitirá uma compreensão mais aprofundada das particularidades e desafios enfrentados durante cada etapa do ciclo de vida da integração, fornecendo insights valiosos para futuras iniciativas similares.

4.5.1. Análise de Dados

Analisando as estratégias para armazenamento e gerenciamento de dados à luz do que foi discutido na fundamentação teórica, introduzindo alguns processos mais modernos, principalmente quando se trabalha com um grande volume de dados, e trazendo para a realidade do projeto discutido aqui, vamos abordar algumas importantes lições aprendidas.

Como já especificado, estamos lidando com um varejo com mais de 1.500 lojas, o que implica em um alto volume de cupons de vendas diariamente. Tínhamos, portanto, uma grande quantidade de transações acontecendo de forma paralela, escrevendo novos registros em uma mesma base, além das operações de e-commerce. Isso resultou na primeira lição, que foi o problema da concorrência (com o sistema de PDV e Painel de Vendas, por exemplo) em

um banco de dados relacional centralizador, que processa transações de escrita e leitura. A concorrência resultava, em alguns casos, em uma intervenção manual por parte da equipe de redes e banco de dados para interromper os processos em execução que estavam extraindo os dados de vendas, afetando a operação mais crítica, que, neste caso, é a operação em loja.

Uma estratégia que poderia ter sido adotada aqui era a de criar um data lake, para que o banco de dados relacional permanecesse com todos os seus recursos dedicados aos processos de escrita. Enquanto isso, a consulta de relatórios de vendas poderia ser executada nessa outra base. A abordagem sugerida vai ao encontro do CQRS (*Command Query Responsibility Segregation*), padrão que separa as estruturas utilizadas para mutação de dados da parte de consultas, utilizando eventos ou triggers para a atualização da base de leitura, mantendo a consistência entre as duas. Esse padrão é valioso quando lidamos com operações envolvendo um grande volume de transações, pois adiciona uma complexidade importante ao sistema.

Adicionalmente, os relatórios extraídos no processamento de integração de faturamentos incluíram uma problemática de desempenho, principalmente devido ao esquema do banco de dados. Isso ocorreu porque seria necessária uma ação de junção em diversas tabelas para entrar nos detalhes do cupom fiscal, um requisito de negócio definido pela interface criada pela administradora dos shoppings. Essa abordagem conduziu a um alto consumo de recursos e memória no banco de dados de produção, um aspecto percebido somente após a implantação do sistema em produção, o que demonstra a falta de maturidade do time técnico responsável.

4.5.2. Aspectos de Arquitetura

Havia a possibilidade de adotar a estratégia de integração orientada a eventos, conforme discutido na fundamentação teórica (tópico de arquitetura). A principal contribuição dessa abordagem está no desacoplamento da gestão das filas em um processo orientado a eventos por meio de mensageria. Essa coordenação e manutenção estão a cargo da API REST de vendas. No entanto, com a remodelagem, a API seria encarregada apenas de extrair os dados de faturamento e enviá-los ao módulo de integração. Dessa forma, seria configurada uma fila no servidor RabbitMQ que faria o controle do acionamento da integração. Cada nova mensagem representaria uma entidade que conteria apenas duas chaves: código da filial e data de faturamento.

Essa proposta de melhoria visa promover uma maior segregação das responsabilidades, contribuindo com a escalabilidade, desempenho, modularidade e, principalmente, manutenção do projeto. No entanto, por falta de capacidade técnica no time de arquitetura e desenvolvimento na época, foi escolhida a solução de processar os envios orientados por fila, sendo armazenados no contexto de um banco de dados relacional. Isso adicionou gargalos desnecessários quanto ao desempenho, armazenamento e melhores práticas de projeto.

4.5.3. Aplicação dos Padrões de Projeto

Foi abordada uma estratégia de padrão de projeto Abstract Factory para construção do módulo de integração. O padrão Abstract Factory é um padrão de criação que sugere uma interface para criar famílias de objetos relacionados ou dependentes entre si, sem especificar suas classes concretas (GAMMA, HELM, et al., 2000).

Com esse padrão, todos os métodos de geração de cenários saem da classe principal e entram na classe abstrata que irá fabricar os cenários. O padrão também propõe uma classe genérica para a geração de cenários, que será uma classe concreta da fábrica abstrata. Esse padrão de projeto só foi adotado na segunda entrega do projeto. Em um primeiro momento, a demanda de negócio surgiu para a integração dos faturamentos para uma rede de shoppings (descrita como Ancar na Figura 15). Posteriormente, após a entrega do produto e o acompanhamento bem-sucedido em operação, surgiu um segundo grupo de shoppings que demandava o mesmo processo, mas com outras regras de negócio, modelos de dados e interfaces.

Com isso, surgiu a necessidade de refatorar o módulo de integração para que fosse possível incrementar não só essa nova integração, mas todas as que pudessem surgir posteriormente. Por esse motivo, optou-se pelo padrão que propõe uma abstração das entidades concretas, disponibilizando, dessa forma, um método de fábrica abstrata, instanciando os objetos concretos que implementam os métodos com as regras específicas para cada cenário.

4.5.4. Entendimento do processo de negócio

A necessidade de negócio surgiu a partir de uma norma jurídica, decorrente da necessidade de adaptação da empresa ao contrato estabelecido com um parceiro. Portanto, o

envolvimento dos *stakeholders* foi incrementado, incluindo analistas jurídicos, que provocaram a necessidade de integração entre as aplicações responsáveis pelos dados de faturamento das lojas e os grupos locadores dos pontos comerciais. Com isso, ocorreu alinhamento entre as três frentes: locador, locatário e jurídico, para o desenho do projeto e compreensão do processo atual, que ocorria manualmente. Esse processo consistia na prestação de contas das filiais por meio da extração diária de um relatório de vendas.

Além disso, foram definidas as ações de design do projeto, a metodologia aplicada, a equipe de desenvolvedores dedicada, bem como o planejamento macro do escopo, capacity, primeiros entregáveis e um primeiro roadmap do projeto, tudo isso à luz da gestão ágil de projetos.

4.5.5. Identificação dos sistemas envolvidos

Com o envolvimento da equipe de desenvolvimento dos parceiros responsáveis pelo sistema terceiro que precisaríamos integrar, foi possível realizar o levantamento dos componentes envolvidos no projeto de integração. Como esse processo de integração automatizada de faturamentos era uma exigência dos parceiros de negócio, eles já tinham o domínio do processo e, portanto, um serviço contendo as interfaces definidas, o que facilitou a identificação por parte do módulo de integração proposto.

A segunda fase consistiu no levantamento dos sistemas que detinham as informações dos faturamentos. Como a proposta era descentralizar esse processo do core de vendas e fiscal, dois dos principais pilares de negócio da companhia, identificou-se a necessidade da criação, para isso, de uma interface de aplicação em conformidade com as restrições da arquitetura REST.

Portanto, não havendo uma API pronta capaz de consumir os dados de vendas das filiais, a atividade, nesta etapa, foi voltada para a identificação e mapeamento das bases de dados, servidores, bancos e tabelas que continham as informações necessárias.

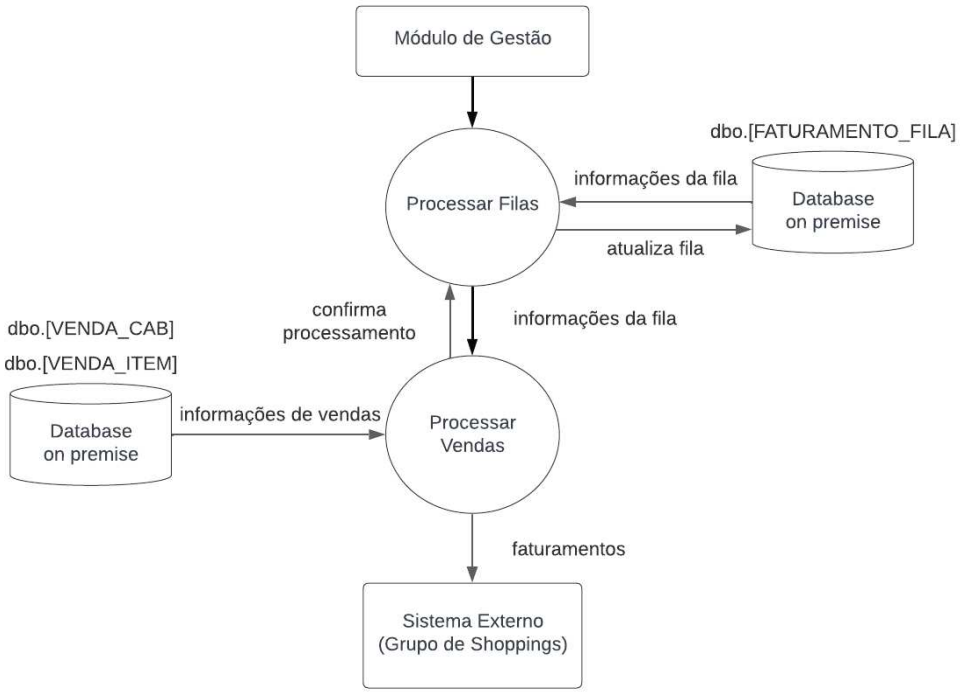
4.5.6. Desenho do fluxo de dados entre os sistemas

Nessa fase, ocorreu a elaboração dos fluxos de dados entre os sistemas envolvidos na integração, que foram levantados na fase anterior, juntamente com as interações entre os sistemas e as bases de dados. Mesmo compreendendo que o desenho do fluxo não deve entrar no nível de granularidade dos campos, foi necessária uma solicitação prévia da interface

disponibilizada pelo sistema externo para que fosse possível mapear os bancos e tabelas envolvidos na integração.

Como entregável, foi desenvolvido um modelo de fluxo de dados para o processo de integração de vendas, conforme apresentado na Figura 17.

Figura 17 – Desenho do fluxo de dados para a integração de vendas



Fonte: Aplicativo de diagramação Lucidchart (2023).

4.5.7. Definição dos contratos de interface

De maneira prática, o projeto possui duas integrações para que a comunicação entre os três módulos descritos ocorra. Por isso, foi preciso definir as interfaces da API de vendas para a disponibilização dos serviços ao módulo de gestão de processos e, para a consolidação do fluxo de dados, havia a necessidade da definição dos contratos de interface entregues pelo sistema do grupo de shoppings para a modelagem da comunicação com a API de vendas.

As interfaces da API, desenvolvida em arquitetura REST, atendem às especificações do protocolo HTTP, no qual foram desenvolvidos três serviços simples para que a aplicação de gestão pudesse requisitá-los com o intuito de acionar os processos de

integração, sendo eles: Criar Filas, Processar Filas e Atualizar Filas com Erro. Como as duas entidades de sistema que compõem a primeira fase da integração são de nosso domínio, optou-se por utilizar a abordagem de documentação por meio da ferramenta de código aberto Swagger, um framework já apresentado na seção de apresentação do domínio do projeto.

Devido ao domínio do processo por parte da empresa terceira, eles já disponibilizaram a documentação de especificação das interfaces, serviços, detalhamento e estrutura dos campos, com exemplos de integração, além da definição do modelo de comunicação e protocolo utilizado. Foi adotada uma abordagem de web service por meio de interfaces para a integração de arquivos, podendo ser JSON, XML, CSV, etc. O web service foi desenvolvido para receber e armazenar o arquivo para que, posteriormente, em um pós-processamento, fosse efetivada a integração das informações contidas nele, definindo, portanto, um modelo de integração assíncrona entre as entidades de sistema. Nossa escolha foi enviar os arquivos em formato JSON, por ser um modelo mais simples e pela facilidade em trabalhá-lo em APIs REST.

4.5.8. Definição dos serviços

Vamos analisar essa definição sob a perspectiva de duas situações dentro de nosso contexto, referentes ao mapeamento dos fluxos de orquestração, isto é, como os serviços serão acionados, coordenados e interconectados ao longo do processo de integração, e também sobre a definição de serviços desenvolvidos para lidar com as condições alternativas e recuperação de falhas.

Como detalhado na fundamentação teórica, essa é uma integração assíncrona, diária e processada em background, sem a necessidade de interação com um usuário de sistema. Portanto, o desafio consistia no acionamento e coordenação dos serviços desenvolvidos. Para isso, pensou-se na estratégia de filas de processamento, que funcionavam da seguinte forma:

1. Acionamento do serviço para criação das filas: havia uma consulta na base de filiais locadas em shoppings da rede, em seguida o cadastro na base de filas contendo o código da filial, a data referente ao processamento esperado (optou-se por integrar as filas em formato de D-1, ou seja, a rotina de processamento hoje integra os faturamentos de ontem), e o status inicial igual a 'Ativo'.
2. Ao acionar o serviço de processamento das filas, o sistema faz uma consulta na base de filas, consumindo as filas com status 'Ativo'.

3. Ao término do processamento, o mesmo serviço é encarregado de retornar na base de filas e atualizar o seu status para 'Processado', em caso de sucesso, e 'Erro' em caso de falha, para que a fila possa ser reprocessada.

Para lidar com as condições alternativas, um dos cenários analisados foi o caso de erro na integração com o serviço externo e, por consequência, com as filas em status de erro. Para isso, desenvolveu-se uma terceira interface para que o módulo de gestão dos processos pudesse acionar a API e reprocessar as filas com status de 'erro', prevendo uma falha de comunicação, rede ou até intermitência dos servidores envolvidos. Dessa maneira, uma nova tentativa de integração é efetuada. Como a responsabilidade de coordenação das ações está sob comando do módulo de gestão, nela é configurado um número máximo de tentativas para o reprocessamento das filas, que, se ainda permanecerem em falhas, por qualquer motivo, o status é atualizado para 'Erro', e os logs são armazenados em um campo da tabela, permitindo a consulta dessas falhas de integração pelo time de negócio para avaliar com mais refinamento esses casos alternativos.

4.5.9. Configuração dos Ambientes

Configurou-se um ambiente de homologação com o intuito de validar os processos desenvolvidos, buscando simular os aspectos críticos pelos quais a integração enfrentaria. Como entregável, foram criados os repositórios na plataforma de gestão de código, os arquivos de configuração para orquestração das aplicações em containers Docker e criação das novas tabelas no banco de dados.

4.5.10. De-Para / Diagrama de Sequência

A planilha foi desenvolvida para atender às especificações a nível de granularidade campo a campo. Visto que, identificando as informações necessárias para a integração e a estrutura das interfaces, é preciso mapear a origem de cada campo, com o entendimento da formatação dos dados, da junção entre tabelas para consumo dos detalhes de cada objeto e, posteriormente, criar uma documentação que servirá como base fundamental para a manutenção do projeto no pós-implantação. Como entregável, foi construída a planilha De-Para com a especificação dos campos.

4.5.11. Construção

As atividades já definidas, foi o momento de refinar as histórias de usuário, estimá-las e executar as disciplinas ágeis para a entrega do projeto dentro do prazo pré-definido. Houve o envolvimento de analistas de infraestrutura nessa fase, de forma a configurar previamente os ambientes no servidor e as pipelines de build, testes automatizados e deploy.

Isso foi feito para integrar com as ferramentas do Azure DevOps, com o objetivo de acelerar esse processo, evitando passar por rotinas burocráticas da gestão de mudanças em ambiente de qualidade, promovendo maior agilidade e autonomia do time de desenvolvimento. Além disso, nessa fase, passamos pelo time de segurança da informação para configurar os usuários de serviço e definir as regras de acesso para uma ação de autenticação e autorização dos acessos.

4.5.12. Testes

A cobertura de testes no projeto foi definida com o objetivo de atingir 80% do código. Para alcançar essa meta, foram adotadas algumas abordagens e estratégias:

1. A criação dos testes unitários foi estabelecida como um pré-requisito para qualquer nova versão a ser enviada ao ambiente de qualidade.
2. Foi adotado o software SonarQube para automatizar o processo de revisão do código. Essa ferramenta é amplamente utilizada na indústria e oferece uma análise inteligente do código desenvolvido, identificando possíveis bugs, duplicações e code smells. Essa avaliação é realizada de maneira prática, gerando uma nota final para a versão do software que foi implementada.
3. Por fim, a aplicação dessas duas rotinas (testes unitários e análise do SonarQube) nas pipelines automatizadas melhorou ainda mais o desempenho do time de desenvolvimento. Essas rotinas de teste foram integradas ao processo de deploy do código, garantindo maior eficiência e qualidade durante todo o ciclo de desenvolvimento.

4.5.13. Definição do Capacity

Como fase de preparação para a implantação, os esforços se voltaram para as definições de tamanho dos payloads, com uma configuração de 100 notas por requisição, a configuração dos pods mantidos pelo Kubernetes (grupo de um ou mais containers, com armazenamento compartilhado e recursos de rede, e uma especificação de como executar os contêineres) e, por fim, a aplicação das rotinas de teste de desempenho com a utilização da ferramenta JMETER.

A principal lição aprendida nessa fase, sobretudo durante os testes, consiste na falha ao tentar simular o ambiente de produção. Para isso, foi feita a transferência dos dados do servidor de produção para o ambiente de qualidade, e, dessa forma, as aplicações se comportaram da maneira esperada e com desempenho aceitável. A principal complicação adicionada a esse contexto está no ecossistema de aplicações, com suas instâncias de containers, concorrendo, em produção, pelos mesmos recursos, em um cenário que não foi mapeado nesta fase e que produziu um impacto significativo após a implantação.

4.5.14. Implantação e Governança

Nesta fase, os esforços se voltaram para a última rodada de testes, com a documentação dos casos de teste e cenários validados com as respectivas evidências de aprovação. Além disso, foi documentado também o aceite da área de negócio para comprovação da validação e autorização para subida em ambiente de produção. Em seguida, abrimos a rotina para gestão de mudanças, passando as evidências de testes e aceite da área responsável, para que o time de infraestrutura, aprovem a requisição e iniciem as pipelines para deploy. Por fim, o acompanhamento das pipelines no Azure DevOps, e verificação do estado das aplicações em forma de containers, agora em produção, por meio da plataforma do Rancher.

A principal lição aprendida está no *sign-off* da área de negócio que, por se tratar de um processo novo que estava sendo implantado, não havia um setor responsável por ele e essa identificação da área dona do processo deveria ter sido feita nas primeiras etapas do ciclo de vida da integração, de preferência no entendimento do processo de negócio.

4.5.15. Considerações finais

O ciclo de vida da integração destina-se a destacar as principais etapas do projeto de integração entre sistemas heterogêneos. Com a análise realizada sobre o impacto dessas etapas na construção do projeto chamado "Integração de Vendas", percebeu-se que elas nos auxiliam a orientar o processo de desenvolvimento, principalmente para que haja uma definição clara do que se deseja entregar em cada uma delas. Isso contribui significativamente para evitar atropelos nos processos e ressalta a importância de compreender que a construção de um projeto sólido depende de decisões fundamentadas desde o início do ciclo.

Na perspectiva mais prática, observou-se que grande parte do que está na fundamentação teórica foi aplicada como guia. No entanto, mesmo em uma equipe experiente, podem ocorrer falhas e até mesmo displicência durante o processo, principalmente em projetos de maior escala. Destaca-se, portanto, a importância das lições aprendidas elencadas em cada etapa, servindo como pontos de atenção relevantes para projetos dessa natureza. Isso pode agregar significativamente à contribuição desse guia proposto, denominado ciclo de vida da integração.

5 CONCLUSÕES

Neste trabalho, exploramos o universo da integração de sistemas heterogêneos, um tópico que se tornou de extrema relevância na era digital do século XXI. A sociedade contemporânea é alimentada por uma constante troca de informações, impulsionada por uma conectividade onipresente e uma variedade impressionante de aplicativos e recursos, todos interligados em uma grande rede de dados. Desde o agendamento de compromissos pessoais até os processos de gestão em ambientes corporativos, a integração de sistemas desempenha um papel vital na garantia da eficiência, acessibilidade e automação.

A pesquisa explorou as várias facetas da integração de tecnologias, incluindo gestão, arquitetura, modelagem, interfaces, padrões de desenvolvimento e infraestrutura. Esses são os pilares para entender os conceitos que fundamentam o desenvolvimento de um projeto de integração de sistemas heterogêneos, como a integração de vendas apresentada. Nesse projeto, foram analisadas e refletidas de forma mais aprofundada as etapas do seu ciclo de vida, com ações, alinhamentos, perspectivas, entregáveis e, principalmente, lições aprendidas. Em um mundo corporativo onde a agilidade e a adaptação constante são essenciais, é importante equilibrar a busca pela inovação com a estabilidade e a eficácia operacional. A análise cuidadosa, a colaboração e a avaliação crítica são as chaves para tomar decisões estratégicas que impulsionam o sucesso a longo prazo do projeto.

A integração de sistemas é uma área em constante transformação, e este trabalho serve como uma base para futuros estudos, ajudando a compreender melhor como enfrentar os desafios da integração em um mundo tecnologicamente diverso e em constante evolução. Ao mesmo tempo, reforça a importância da colaboração e do compartilhamento de conhecimento na comunidade de desenvolvedores, tornando-nos mais preparados para enfrentar as demandas da integração em um futuro cada vez mais interconectado.

Esperamos que este trabalho não seja apenas uma contribuição valiosa para a área de Engenharia de Computação, mas também uma fonte de inspiração para os que se aventuram no emocionante e desafiador campo da integração de sistemas.

REFERÊNCIAS

CUMMINS, Fred A.. **Integração De Sistemas: arquitetura para integração de sistemas e aplicações corporativas**. New York: Elsevier, 2002.

KATZ, Yehuda et al. **JSON:API A SPECIFICATION FOR BUILDING APIS IN JSON**. 2018. Disponível em: https://jsonapi.org/. Acesso em: 18 jul. 2023.

Nilsson, Jimmy. **Applying Domain-Driven Design and Patterns: With Examples in C# and .NET**, 2006

McCarthy, Tim. **.NET Domain-Driven Design with C#: Problem - Design – Solution**, 2008.

ALONSO; CASATI, Gustavo And; KUNO, Fabio And; MACHIRAJU, Harumi And; VIJAY. **Web services**. Springer, 2004.

MARTIN, Robert C.. **Código limpo: Habilidades práticas do Agile Software**. Alta Books, 2012.

Highsmith, J. (2011) **Gerenciamento Ágil de Projetos – Criando projetos inovadores** (Cap.6).

TANENBAUM, Andrew S. **Distributed Systems: Principles and Paradigms**. 2. ed. [S. l.: s. n.], 2016.

MARTIN, Robert C.. **Clean Architecture: a craftsman's guide to software structure and design**. Londres: Pearson, 2017.

NYGARD, Michael T.. **Release It!: design and deploy production-ready software**. 2. ed. Pragmatic Bookshelf, 2018.

KIM, Gene; BEHR, Kevin; SPAFFORD, George. **O projeto fênix: um romance sobre ti, devops e sobre ajudar o seu negócio a vencer**. Rio de Janeiro: Alta Books, 2020.

FOWLER, Martin. **Patterns of Enterprise Application Architecture**. Londres: Addison-Wesley Professional, 2002.

EVANS, Eric. **Domain-driven design: atacando as complexidades no coração do software**. 3. ed. Rio de Janeiro: Alta Books, 2016.

PRESSMAN, Roger S.; MAXIM, Bruce R.. **Engenharia de software: uma abordagem profissional**. 9. ed. Amgh, 2021.

GAMMA, Erich; HELM, Richard; DR, Ralph Johnson; VLISSIDES, John. **Design Patterns: elements of reusable object-oriented software**. Addison-Wesley Professional, 1994.

FIELDING, Roy T.. **Architectural Styles and the Design of Network-based Software Architectures**. 2000. 6 v. Dissertação (Mestrado) - Curso de Doctor Of Philosophy In

Information And Computer Science, University Of California, Irvine, California, 2000. Disponível em: https://ics.uci.edu/~fielding/pubs/dissertation/top.htm. Acesso em: 07 nov. 2023.

PORTER, Michael E.. **Competitive Advantage: creating and sustaining superior performance**. Free Press, 2008.

KERZNER, Harold. **Project Management: a systems approach to planning, scheduling, and controlling**. 13. ed. Wiley, 2022.

BECK, Kent; COHN, Mike. **User Stories Applied: for agile software development**. Addison-Wesley Professional, 2004.

LINES, Mark; AMBLER, Scott. **Disciplined Agile Delivery: a practitioner's guide to agile software delivery in the enterprise**. Ibm Press, 2012.

ROSEN, Michael; LUBLINSKY, Boris; SMITH, Kevin T.; BALCER, Marc J.. **Applied SOA: service oriented architecture and design strategies**. John Wiley & Sons, 2008.

A BERNSTEIN, Philip; NEWCOMER, Eric. **Principles of Transaction Processing**. 2. ed. Morgan Kaufmann Publishers, 2009.

O'NEILL, Mark. **Web Services Security: application development**. New York: McGraw Hill, 2012.

SEROTER, Richard. **SOA Patterns with BizTalk Server 2009**. Packt Publishing, 2009.

HOHPE, Gregor; WOOLF, Bobby. **Enterprise Integration Patterns - Designing, Building, and Deploying Messaging Solutions**. The Addison Wesley Signature Series. Addison Wesley, Boston, 2003.

BONDI, André B. **Characteristics of scalability and their impact on performance**. In: Proceedings of the 2nd international workshop on Software and performance. 2000. p. 195-203.

SHAW, Mary; GARLAN, David. **Software Architecture: perspectives on an emerging discipline**. Prentice Hall, 1996.

NARKHEDE, Neha; SHAPIRA, Gwen; PALINO, Todd. **Kafka: the definitive guide: real-time data and stream processing**. New York: O'Reilly Media, 2017