

Os efeitos da arquitetura dirigida a eventos na modularidade de software: Um estudo exploratório

Luan Lazzari

Universidade do Vale do Rio do Sinos (Unisinos)
São Leopoldo, Rio Grande do Sul, Brasil
luanlazzari@hotmail.com

Kleinner Farias

PPGCA, Universidade do Vale do Rio do Sinos (Unisinos)
São Leopoldo, Rio Grande do Sul, Brasil
kleinnerfarias@unisinos.br

ABSTRACT

Este trabalho apresenta um estudo exploratório sobre os efeitos da arquitetura dirigida a eventos na modularização de *software*. Ele é um estudo inicial do qual se busca compreender os efeitos da adoção de arquitetura orientada a eventos na separação de interesses, complexidade, acoplamento, coesão e tamanho, em comparação com o estilo de arquitetura REST. Uma aplicação foi desenvolvida usando a arquitetura dirigida a eventos e a arquitetura tradicional REST através de cinco cenários de evolução. Em cada cenário, uma funcionalidade foi adicionada. As versões geradas foram comparadas usando 10 métricas. Até onde sabemos, os resultados relatados são os primeiros a descrever os benefícios da arquitetura dirigida a eventos, em termos de modularidade de *software* em cenários reais de evolução. Nesse caso, o estudo pode ser visto como o primeiro passo em uma agenda mais ambiciosa para avaliar os benefícios da arquitetura empiricamente orientada a eventos.

KEYWORDS

Arquitetura dirigida a eventos; Modularização; Estudo empírico; Kafka

1 INTRODUÇÃO

A arquitetura dirigida a eventos é uma solução promissora para o desenvolvimento de sistemas distribuídos que entrega modularização, escalabilidade e concorrência [12]. O KAFKA¹ seria um exemplo de tecnologia que suporta arquitetura dirigida a eventos, propondo uma série de componentes que se comunicam através de eventos [27]. Logo, o processamento procura compor serviços, não por meio de cadeias de comandos e consultas, mas sim pelo fluxo de eventos [25]. De tal forma que cada componente pode executar a sua tarefa independente, pois são acionados por gatilhos, que por sua vez são eventos [27]. Ao ponto em que há a separação entre computação e comunicação, tornando fácil a integração entre componentes heterogêneos em sistemas complexos que são fáceis de evoluir e escalar [12], como por exemplo, ambientes inteligentes [22].

Já em modelos de sistemas baseados em requisição/resposta, os dados são obtidos em diferentes fontes via requisições, por exemplo, HTTP, podendo gerar possíveis congestionamentos [12]. A literatura atual [25] aponta que a arquitetura dirigida a eventos promove o fraco acoplamento, essencial para a modularização de software, porém pode aumentar a complexidade do projeto e o entendimento do sistema [12]. Entre as arquiteturas tradicionais para implementação de sistemas orientados a serviços, destaca-se a arquitetura REST [13].

Os proponentes da arquitetura dirigida a eventos advogam que projetar aplicações fortemente baseada em eventos favorece à modularização das funcionalidades, bem como facilita atividades de manutenção e evolução. Neste sentido, projetar software adotando uma arquitetura dirigida a eventos pode implicar em uma forma mais sistemática de promover uma melhor modularização de software. Atualmente, conjectura-se que o uso de tecnologias como o KAFKA gerará aplicações com uma melhor separação de interesses, melhor acoplamento e coesão, menor complexidade e tamanho. No entanto, há poucas evidências para confirmar se essa expectativa se confirma ou não. Hoje, a literatura ainda carece de estudo exploratórios que investiguem os efeitos de uma arquitetura dirigida a eventos em aspectos de modularidade de software. Além disso, não se tem conhecimento se esses efeitos são melhores ou piores que os causados por arquiteturas tradicionais e amplamente utilizadas, tal com o estilo arquitetura REST. Consequentemente, os desenvolvedores acabam adotando arquitetura dirigida a eventos sem nenhuma evidência empírica sobre seus efeitos na modularidade de software. Alguns estudos na literatura apontam para alguns benefícios da adoção de arquitetura dirigida a eventos. Laigner et al. [17] reporta um estudo empírico no qual se constatou que a adoção da arquitetura dirigida a eventos melhorou a manutenção e o isolamento de falhas em um sistema que foi refatorado após passar anos evoluindo, dando origem a um código grande, complexo e obsoleto, exigindo um processo de manutenção caro. Urdangarin et al. [28] destaca a importância da decomposição de aplicações monolíticas, principalmente para reduzir o esforço de manutenção de software [5].

Este artigo, portanto, apresenta um estudo empírico exploratório sobre os efeitos de arquitetura dirigida a eventos na modularidade de software. Trata-se de um estudo inicial através do qual busca-se compreender os efeitos da adoção arquitetura dirigida a eventos na separação de interesses, complexidade, acoplamento, coesão e tamanho, em comparação com o estilo arquitetura REST. Investiga-se uma faceta particular da arquitetura dirigida a eventos em relação aos benefícios na evolução de uma aplicação alvo real através da adição de novas funcionalidades. Uma aplicação foi desenvolvida utilizando uma arquitetura dirigida a eventos e uma arquitetura tradicional REST através de 5 cenários de evolução. Em cada cenário, uma funcionalidade foi adicionada. As versões geradas foram comparadas usando 10 métricas. Os resultados reportados são os primeiros a relatar as vantagens potenciais da arquitetura dirigida a eventos, em termos de modularidade de software em cenários reais de evolução. Neste sentido, este artigo pode ser visto como um primeiro passo em uma agenda mais ambiciosa para avaliar os benefícios da arquitetura dirigida a eventos empiricamente.

¹KAFKA: <https://www.confluent.io/>

O estudo está dividido conforme a seguinte estrutura: a Seção 2 conterà o referencial teórico, com os principais conceitos para entendimento do estudo proposto; a Seção 3 abordará os trabalhos relacionados, explorando o processo de seleção utilizado e também realizando um comparativo destes com o presente; a Seção 4 abordará a descrição da metodologia para desenvolvimento do estudo; a Seção 5 traz os resultados obtidos; e, por fim, a Seção 6 traça algumas conclusões e trabalhos futuros.

2 REFERENCIAL TEÓRICO

Esta seção aborda os conceitos teóricos usados durante a construção e desenvolvimento do estudo.

2.1 Modularização de interesses

Modularização é considerado um conceito essencial no *design* de *software* moderno [21]. Definido pelo IEEE como o grau em que um programa de sistema é composto de componentes discretos, de forma que uma mudança em um componente tenha impacto mínimo em outros [21]. Enquanto um interesse é qualquer propriedade importante ou área de interesse de um sistema que se deseja tratar de forma modular [21]. Portanto, um interesse de *software* pode ser uma *feature*, regra de negócio, requisito não funcional, um padrão de arquitetura ou padrões de projeto [21].

Software com alto grau de modularização traz vários benefícios como compreensão, extensibilidade, adaptabilidade, reuso, entre outros [19]. Por consequência a modularização pode ser aplicada em várias etapas do *design*, variando da especificação da arquitetura ao detalhamento do *design* e níveis de abstração de código [21]. O principal objetivo da arquitetura de *software* é definir em quais componentes o sistema deve consistir, como esses componentes irão se comunicar uns com os outros e como eles devem ser implantados para cumprir os requisitos [6]. Portanto, a arquitetura de *software* desempenha um papel importante na formulação e desenvolvimento de *software* [16].

Todavia a decomposição arquitetural ainda é um grande gargalo para o processo de *design* de *software* segundo [21], principalmente pela necessidade da modularização simultânea de uma série de questões de amplo escopo. Consequentemente, a inadequada modularização de interesses pode gerar complexidade no *design* de *software* [21]. A avaliação de diferentes arquiteturas requer técnicas para mensurar quantitativamente [21]. De modo que as métricas de *software* são poderosos indicadores de modularização no *design* de *software* [21]. Logo, a comunidade tem utilizado consistentemente noções de acoplamento de módulo, coesão e tamanho de interfaces para mensurar a modularização [21]. Por isso, neste estudo foi definido um conjunto de métricas abordadas na Seção 4.2.

2.2 Arquitetura dirigida a eventos

No modo baseado em eventos, os componentes apenas publicam dados, sem conhecer os demais componentes, muitos menos quem irá consumir e reagir aos dados, promovendo a separação da computação e publicação de eventos de qualquer processamento subsequente [12]. Além disso, sua comunicação é assíncrona, no modelo produtor/consumidor, ambos são independentes um do outro [6].

Por consequência, promovendo o fraco acoplamento entre os componentes, motivo pelo qual a arquitetura dirigida a eventos tornou-se predominante em aplicações distribuídas em grande escala [12].

O sistema de mensagens nos permite construir serviços fracamente acoplados, pois ele move os dados puros a um local altamente acoplado (o produtor) e coloca em um local fracamente acoplado (o consumidor) [25]. Portanto, quaisquer operações que precisem ser executadas nesses dados não são feitas no produtor, mas em cada consumidor [25]. Isto é, serviços podem ser facilmente adicionados ao sistema, no modo *plug and play* (plugável), em que ele se conecta aos fluxos de eventos e executando quando seus critérios são atendidos [25].

Não só promove o baixo acoplamento mas também o *Kafka Stream* é capaz de armazenar os eventos e dados, não necessitando um banco de dados externo, e, mantendo os eventos “próximos” dos serviços [25]. Além disso, todos os eventos são armazenados na ordem em que chegaram, permitindo que os eventos sejam reproduzidos em ordem [25]. Por isso o desempenho de aplicações baseadas em eventos também se mostra melhor, garantindo estabilidade e alto desempenho para grandes quantidades de eventos.

2.3 Arquitetura REST

Dentre as arquiteturas tradicionais, destaca-se a *REST* no modelo requisição/resposta síncrono. Embora, também seja possível operar no modelo assíncrono seu uso é menos difundido. No modelo síncrono o cliente realiza uma requisição e aguarda pela resposta, enquanto está sendo processado pelo serviço responsável. De forma que é amplamente utilizada em aplicações distribuídas em rede [29]. Promovendo, portanto, o fraco acoplamento entre seus interesses, um dos motivos para sua popularidade [29]. Tanto que dentre os modelos arquitetônicos mais utilizados está o modelo MVC (Model-View-Controller), amplamente difundido [23]. Pois, seu principal conceito é a separação em camadas, separando a persistência de dados, interface do usuário e controle da aplicação [20]. Tal separação em camadas tem como objetivo promover a separação de interesses e consequentemente promover a modularização.

Diferentemente de arquiteturas dirigidas a eventos, adicionar novos serviços na arquitetura *REST*, em geral, implica em introduzir um novo método e realizar chamadas pelos serviços que precisam dele [25]. Entretanto, a arquitetura *REST* possui algumas vantagens consideráveis como a simplicidade em implementar, dados (ou estado) reside em apenas um lugar e controle centralizado [25]. De forma que neste estudo o principal ponto de comparação entre a arquitetura *REST* e a orientada a eventos será a modularização. A fim de mensurar a modularização, alguns conceitos de qualidade de software e métricas serão abordados para suporte da análise quantitativa.

3 TRABALHOS RELACIONADOS

A pesquisa pelos trabalhos relacionados foi realizada em repositórios digitais como *Google Scholar* e *Scopus (Elsevier)*. Grande parte dos trabalhos selecionados foram resultados de pesquisas pelo termo “event-driven architecture AND microservice”.

3.1 Análise dos trabalhos relacionados

Figueiredo et al. [14]. Apresenta um estudo quantitativo sobre duas *Software Product Lines (SPLs)*, a fim de avaliar várias facetas de estabilidade de *design*, considerando métricas como SoC (*Separation of Concerns*), acoplamento, e coesão. As SPLs foram implementadas usando programação orientada a aspectos (AO) e orientada a objetos (OO). As versões AO e OO SPLs foram comparadas, buscando entender os benefícios de AO em questões de qualidade de software. O artigo reporta os benefícios de uma arquitetura orientada a aspectos de SPLs.

Garcia et al. [15]. Apresentam um estudo quantitativo, comparando implementações em Java e AspectJ dos 23 padrões de projeto da *Gang-of-Four (GoF)*. Para tal foi utilizada a programação orientada a objetos (OO) e programação orientada a aspectos (AOP). A fim de comparação, as métricas utilizadas foram o acoplamento dos objetos e o SoC, mas também a coesão e tamanho. Considerando as características das implementações em cada padrão. Após cada alteração eram coletadas as métricas, comparando sempre com a versão anterior, antes das alterações. Por fim, reporta em quais pontos AOP se destacou positivamente e negativamente em comparação ao OO.

Fiege et al. [12]. Apresenta um estudo qualitativo sobre o *design* modular e a implementação de um sistema de eventos, capaz de suportar escopos e mapeamentos de eventos. Dentre os conceitos, são especificados o papel do produtor e consumidor, formas de comunicação entre eles e gatilhos entre eventos. Bem como os benefícios, os quais pode-se citar a modularização do sistema, baixo acoplamento, abstração e ocultamento de informações. Além dos componentes que compõem uma arquitetura dirigida a eventos, como o *subscribe/unsubscribe* dos eventos, necessário para garantir a distribuição de mensagens. Bem como o ponto central responsável por gerenciar parte do sistema, como passar um gatilho a um ou mais eventos.

Falatiuk et al. [6]. Apresentam um estudo qualitativo, descrevendo os principais conceitos arquiteturais e seleção de tecnologias para implementação de um sistema de gerenciamento de documentos, o *e-archive*. Para isso, as arquiteturas escolhidas foram a dirigida a eventos e microsserviços, de acordo com os requisitos levantados. Como vantagens encontradas, estão a escalabilidade horizontal, modularização, perda de acoplamento entre componentes, facilidade de modificações e manipulação de grande quantidade de dados. Contudo, requer maior conhecimento de padrões de arquitetura na nuvem e cultura *DevOps*, para garantir o escalonamento e o devido monitoramento. Assim, o *design* se mostra bastante custoso em seu início, mas oferece futuras manutenções, modificações e atualizações mais baratos à medida que o sistema evoluir.

Alaasam et al. [1]. Propõe um estudo de caso sobre a viabilidade de uso do *Apache Kafka Stream API (Kafka stream DSL)* no desenvolvimento do *Digital Twin (DT)*. Um sistema de processamento de fluxos de dados em tempo real, capaz de monitorar, controlar e prever estados a partir de dados, coletados de diversos sensores. Nele foi realizado um estudo paramétrico de latência e tempo de resposta, levando em consideração tolerância a falhas, escalabilidade e facilidade de implementação. Como conclusão, o *Kafka* se mostrou adequado ao sistema proposto, fornecendo um bom gerenciamento de estados e latência aceitável. Entretanto, percebeu-se

uma perda na eficiência, enquanto há muito tráfego de dados entre os tópicos intermediários.

Schipor et al. [22]. Introduce o *Euphoria*, uma nova arquitetura de *software* orientada a eventos, voltado aos ambientes inteligentes. Composto por uma enorme gama de dispositivos heterogêneos, cada um com seu sistema operacional, protocolos de comunicação, forma de interação entre outros. Tais ambientes possuem alguns critérios de *design* como a modularidade, escalabilidade e assincronicidade para produzir, processar e transmitir mensagens e eventos. Portanto, o *Euphoria* foi projetado adotando várias técnicas e propriedades de qualidade seguindo aos padrões do (*SQuaRE*) *ISO/IEC 25000*. Além disso, foi conduzido uma avaliação técnica sobre as capacidades do *Euphoria*, foram quantificados tamanho das mensagens, diferentes dispositivos e complexidade do ambiente (quantidade de dispositivos). Seu resultado foi satisfatório, alcançando um baixo tempo de resposta mesmo em um ambiente composto por um número grande de produtores e consumidores.

Laigner et al. [17]. Realizou a troca de um sistema de *Big Data* legado (BDS) para uma arquitetura orientada a eventos baseada em microsserviços. Tal BDS está localizado no Instituto Tecgraf da PUC-Rio, que fornece soluções para parceiros industriais. Uma das soluções desenvolvidas para um cliente do setor de Óleo e Gás em 2008, diz respeito a um BDS que monitora objetos em movimento e detecta de forma proativa eventos que geram riscos à operação, como desvios de rota de veículos. Motivados pelo difícil processo de manutenção do sistema e o advento de um novo parceiro, desenhou-se a reescrita completa do BDS legado para então tecnologias atuais de *Big Data*. Como conclusão, observou-se o suporte a microsserviços para manutenção mais fácil e isolamento de falhas como benefícios. No entanto, o fluxo de dados complexo gerado pelo número de microsserviços, bem como a miríade de tecnologias, como desvantagens.

3.2 Análise comparativa dos trabalhos relacionados

Critério de Comparação. Foram definidos cinco Critérios de Comparação (CC) para realizar a comparação de similaridades e diferenças entre o trabalho proposto e os artigos selecionados. Sua comparação é crucial para auxiliar na identificação de oportunidades de pesquisa utilizando critérios objetivos, ao invés de subjetivos. Os critérios são descritos a seguir:

- **Estudo Empírico (CC1):** pode ser entendida como aquela em que é necessária comprovação prática de algo, especialmente por meio de experimentos ou observação de determinado contexto para coleta de dados em campo;
- **Análise de Modularização (CC2):** compreende a capacidade de decomposição da aplicação em programas menores com interfaces padronizadas, oferecendo maior capacidade de gerenciamento no desenvolvimento da aplicação;
- **Arquitetura dirigida a eventos (CC3):** composta por diversos componentes onde a comunicação entre produtores e consumidores ocorre por meio de eventos;
- **Arquitetura de microsserviço (CC4):** cada módulo da aplicação é totalmente independente de outros (*standalone*), cada contém uma suíte de serviços autônomos, comunicando-se por meio de uma API;

- **Contexto de aplicação (CC5):** define se o estudo aborda a implementação de uma aplicação real.

Oportunidades de pesquisa. A Tabela 1 apresenta a comparação dos estudos selecionados, evidenciando o não atendimento de todos os critérios. A seguir, as oportunidades observadas a partir das comparações: (1) apenas o trabalho proposto atende todos os critérios de comparação definidos; (2) nenhum estudo empírico referente à arquitetura dirigida a eventos; e (3) assim como nenhum, sobre modularização de *software* em arquitetura dirigida a eventos e microsserviços. Portanto, a seguinte oportunidade de pesquisa foi identificada: análise sobre os efeitos da arquitetura dirigida a eventos na modularização de *software*. À medida que será explorada nas próximas seções.

Table 1: Análise comparativa dos Trabalhos Relacionados selecionados

Trabalho Relacionado	Critério de Comparação				
	CC1	CC2	CC3	CC4	CC5
Trabalho Proposto	●	●	●	●	●
Figueiredo <i>et al.</i> [14]	●	●	○	○	●
Garcia <i>et al.</i> [15]	●	●	○	○	●
Fiege <i>et al.</i> [12]	●	●	●	○	○
Falatiuk <i>et al.</i> [6]	○	●	●	●	○
Alaasam <i>et al.</i> [1]	●	○	●	●	○
Schipor <i>et al.</i> [22]	●	○	●	○	●
Laigner <i>et al.</i> [17]	●	○	●	●	●

● Atende Completamente ● Atende Parcialmente ○ Não Atende

4 METODOLOGIA

Esta seção descreve a metodologia seguida para executar o estudo empírico. A Seção 4.1 apresenta o objetivo do estudo e a questão de pesquisa investigada. A Seção 4.2 descreve as métricas utilizadas. A Seção 4.3 detalha o processo experimental seguido, descrevendo as fases do estudo e as atividades executadas. A Seção 4.4 traz os procedimentos de análise dos dados. A Seção 4.5 descreve a aplicação alvo escolhida, detalhando suas funcionalidades e características. Por fim, a Seção 4.6 detalha os cenários de evolução da aplicação alvo. A metodologia deste trabalho é inspirada a partir de estudos empíricos anteriores previamente publicados [4, 7–11, 18].

4.1 Objetivo e Questão de Pesquisa

O objetivo deste estudo é essencialmente avaliar os efeitos da arquitetura dirigida a eventos na modularidade de software. Em particular, busca-se investigar os efeitos sobre cinco diferentes variáveis envolvidas com modularidade [14]: separação de interesses, acoplamento, complexidade, coesão e tamanho. Esses efeitos são investigados no contexto de cenários reais de evolução (Seção 4.6) de uma aplicação alvo (Seção 4.5), a qual foi implementada usando a arquitetura dirigida a eventos e usando a arquitetura REST. As duas implementações geradas foram necessárias para viabilizar a comparação. Portanto, o objetivo deste estudo é estabelecido com base no modelo GQM [3] da seguinte forma:

Analisar estilos arquiteturais
com o propósito de investigar seus efeitos
em relação à modularidade de software
através da perspectiva de desenvolvedores
no contexto de evolução de software.

Em particular, este artigo tenta revelar os efeitos da arquitetura dirigida a eventos na modularidade de software durante a evolução de software. Neste sentido, uma questão de pesquisa (QP) foi formulada:

- **QP:** Arquitetura dirigida a eventos promove uma maior modularização quando comparada ao estilo arquitetura REST?

Parnas [19] aponta que, caso a modularização de uma aplicação seja alta, alguns benefícios serão obtidos como, por exemplo, maior facilidade de alteração, maior poder de adaptação e compreensão de código. Além disso, a modularização pode proporcionar o desenvolvimento de cada módulo independentemente, permitindo o desenvolvimento paralelo, redução do tempo de desenvolvimento e o melhor gerenciamento do impacto de alterações [2]. Parnas [19] reforça que um módulo pode ser definido como um conjunto de decisões de projeto independente de outros módulos e a interação entre os módulos deve ser totalmente por meio de suas *interfaces* [19] – promovendo, assim, a separação de interesses e delegando funções isoladas a cada módulo. Portanto, a separação precisa dos interesses da aplicação leva a modularização, que pode tornar a aplicação customizável, permitindo seu uso em diferentes contextos. Além disso, a modularização permite que o desenvolvedor foque em um módulo por vez, facilitando o entendimento, para depois combinar todos e entender a aplicação pelo todo [2]. A arquitetura dirigida a eventos [22, 25] tenta contemplar tais benefícios citados, destacando a importância da execução de um estudo empírico para verificar os benefícios deste novo estilo arquitetural.

4.2 Métricas

Tabela 2 apresenta as métricas utilizadas para quantificar cinco variáveis de modularidade, incluindo separação de interesses (SoC), acoplamento, coesão, complexidade e tamanho. Tais métricas foram utilizadas, pois estudos empíricos anteriores [14, 15] já mostraram a validade delas em investigações sobre modularidade de software.

Separação de interesses. Neste estudo, esta métrica busca medir o grau de modularização das funcionalidades implementadas usando a arquitetura dirigida a eventos e o estilo arquitetural REST. A SoC utilizará três métricas: (i) componentes (ou classes) baseado no *Concern Diffusion over components (CDC)*, (ii) operações (ou funções) baseado no *Concern Diffusion over operations (CDO)*, e (iii) linhas de código baseado no *Concern Diffusion over Lines of Code (CDLOC)* [14, 15]. Estas métricas ajudam a revelar o grau de espalhamento e entrelaçamento das funcionalidades (*features*) implementadas nos módulos da aplicação alvo. Quanto menor o número de módulos necessários para a implementação de uma funcionalidade, menor será o seu grau de espalhamento. Quanto maior o número de funcionalidades existentes em um determinado módulo, maior será o grau de entrelaçamento entre elas. As métricas do SoC foram quantificadas manualmente [14, 15] através do “sombreamento” manual do código que identifica quais trechos do código fonte contribuem para a implementação de uma determinada funcionalidade [14].

Acoplamento. Esta variável busca quantificar, através de duas métricas *Dep_Out* e *Dep_In*, o quão os elementos de *design* (pacotes, classes e métodos) estão acoplados. Quanto maior o grau de dependência entre eles, mais elementos tendem a sofrer com propagações indesejadas de modificações. *Dep_In* quantifica o número de

Table 2: Conjunto de métricas utilizadas no estudo (fonte [15]).

Variável	Métrica	Definição
Separação de interesses (SoC)	<i>Concern Diffusion over components (CDC)</i>	Conta o número de classes cujo objetivo principal é contribuir para a implementação do cenário e o número de classes que os acessam.
	<i>Concern Diffusion over operations (CDO)</i>	Conta o número de métodos cujo objetivo é contribuir para a implementação do cenário e o número de métodos que os acessam.
	<i>Concern Diffusion over LOC (CDLOC)</i>	Conta o número de pontos de transição para implementação do cenário nas linhas de código. Os pontos de transição são pontos no código onde há uma "mudança de preocupação".
Acoplamento	<i>Coupling between components (Dep_Out)</i>	Número de dependências em que o módulo é cliente.
	<i>Coupling between components (Dep_In)</i>	Número de dependências em que o módulo é fornecedor.
Coesão	Relational cohesion (H)	Mensura o número médio de relacionamentos internos por classe/interface. É calculada como a razão de $R + 1$ para o número de classes e interfaces por pacote.
		Mensura o número de relacionamentos entre classes e interfaces por pacote.
Tamanho	Number of relationships (R)	Mensura o número de relacionamentos entre classes e interfaces por pacote.
	<i>Lines of code (LOC)</i>	Conta as linhas de código cujo objetivo é contribuir para a implementação do cenário.
	<i>Number of attributes (NumAttr)</i>	Conta o número de atributos cujo objetivo é contribuir para a implementação do cenário.
	<i>Weighted operations per component (NumOps)</i>	Conta o número de operações cujo objetivo é contribuir para a implementação do cenário.

classes fora de um pacote que depende das classes dentro deste pacote. *Dep_Out* quantifica o número de classes dentro de um pacote que dependem de classes fora deste pacote.

Complexidade e coesão. A complexidade mede o grau de conectividade entre os elementos por pacote. Assim, para seu cálculo foram somados os valores dos pacotes do projeto. Não obstante, a coesão mensura o grau em que os elementos estão logicamente relacionados ou “pertencem um ao outro”. Por consequência, quanto maior a conectividade entre os elementos maior a coesão. Assim como a complexidade, seus valores são por pacote, neste caso foram somados os valores e divididos pela quantidade de pacotes. Tanto que ambas métricas estão um tanto relacionadas ao tamanho.

Tamanho. As métricas referente ao tamanho são bastante objetivas, incluindo linhas de código (LOC), número de atributos (NumAttr) e operações (NumOps). Tanto que as linhas de código foram coletadas manualmente, desconsiderando linhas em branco ou comentários. Já o número de atributos e operações foi coletado em partes manualmente, e, em outras através do *SDMetrics*². Todavia, para todas métricas referente ao tamanho, foram contabilizadas apenas aquelas que contribuíram para o cenário de evolução que estava sendo avaliado 4.6.

4.3 Processo experimental

Figura 1 apresenta o processo experimental adotado, o qual é formado por três etapas, incluindo a (1) identificação de aplicação alvo, (2) implementação e coleta de dados e (3) análise dos resultados. Cada etapa é discutida a seguir.

Etapa 1: Identificação de aplicação alvo. A primeira etapa se concentrou em buscar na indústria, uma aplicação alvo realística que utilizasse arquitetura dirigida a eventos. Neste sentido, identificou-se a aplicação desenvolvida por Ben Stopford [24, 25] como sendo a aplicação alvo (descrita em Seção 4.5). Esta aplicação foi desenvolvida utilizando boas práticas, trata-se de uma aplicação real que utiliza tecnologias amplamente utilizadas pela a indústria, tais como o Apache Kafka.

Etapa 2: Implementação e coleta de dados. As funcionalidades da aplicação alvo foram identificadas e organizadas em cenários de evolução, de tal forma que permitissem a implementação de uma aplicação similar utilizando o estilo arquitetural REST.

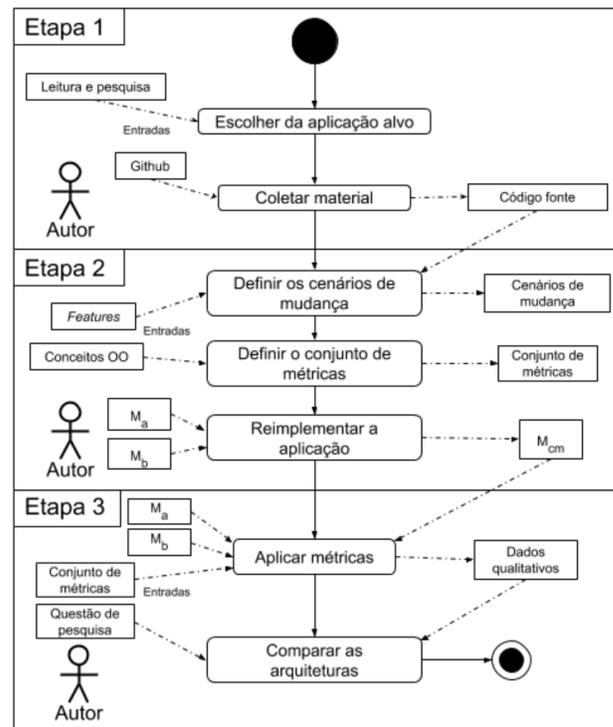


Figure 1: Processo experimental

Destaca-se que as funcionalidades da aplicação alvo estão bem documentadas em [24, 25]. Após a identificação, os funcionalidades foram implementadas utilizando o *framework Spring Boot* e *Spring Web*. Inevitavelmente, por se tratar de arquiteturas distintas, certas diferenças e refatorações são esperadas, a fim de alinhar as aplicações com suas evoluções. Após a implementação de cada cenário de mudança, será coletado um conjunto de métricas definido na Seção 4.2, utilizando a ferramenta *SDMetrics* para coletar parte das métricas. A segunda etapa foi finalizada após a implementação da aplicação alvo utilizando o estilo arquitetural REST.

Etapa 3: Análise dos resultados. Conforme citado na etapa anterior, a partir das métricas obtidas pela ferramenta *SDMetrics*,

²SDMetrics: <https://www.sdmetrics.com/>

foi possível realizar comparações entre as aplicações — as quais serão essencialmente entre as aplicações após cada implementação, permitindo observar suas evoluções. Entretanto, outras comparações podem ser realizadas como, por exemplo, a comparação entre as versões da mesma aplicação, a fim de analisar sua evolução. Assim, para auxílio da análise, serão utilizadas tabelas que permitem identificar visualmente mudanças, além de classificar os elementos por uma métrica e destacar os elementos por percentuais.

4.4 Procedimento da análise

Gráficos de linha são usados para fornecer uma visão geral dos dados coletados no processo de medição. Esses gráficos nos permitem analisar o impacto da arquitetura dirigida a eventos nas métricas de modularização definidas. Cada gráfico enfoca os dados coletados em relação à uma métrica específica. O eixo X especifica os cenários de evolução. O eixo Y apresenta os valores coletados para uma métrica específica. Para trazer uma análise da distribuição dos dados, métodos estatísticos foram utilizados, incluindo desvio padrão, mediana e a média. Além disso, a diferença entre as médias foi contabilizada.

A análise quantitativa dos dados será através das métricas coletadas pela ferramenta SDMetrics, o qual contabiliza automaticamente as métricas Dep_Out, Dep_In, H, R, NumAttr e NumOps. As métricas de separação de interesses (CDC, CDO e CDLOC) foram contabilizadas de forma manual.

4.5 Aplicação alvo

Figura 2 apresenta uma ilustração esquemática da aplicação alvo. Trata-se um sistema de gerenciamento de pedidos composto por vários componentes. A aplicação alvo utilizada foi obtida a partir de um exemplo disponibilizado pela Confluent em [24]. Sendo assim, as principais razões para sua escolha foram o cuidadoso detalhamento da aplicação em [24], a disponibilidade da aplicação e a adoção de boas práticas de implementação. Pode-se considerar uma aplicação complexa, devido os recursos utilizados. A arquitetura abordada é a dirigida a eventos em conjunto com a arquitetura de microsserviços, em uma implementação utilizada a linguagem Java. O Kafka é o *middleware* responsável por gerenciar a aplicação, como armazenamento dos dados, mapeamento de entidades, produção e consumo dos eventos. No site da Confluent, detalhes dos recursos utilizados e responsabilidades de cada componente são apresentados, além de disponibilizar o código fonte da implementação no repositório do Github³.

4.6 Cenários de evolução

Tabela 3 apresenta os cenários de evolução considerados. No total, cinco cenários foram identificados, cada qual contendo funcionalidades relativas à aplicação alvo. Cada cenário incorpora uma funcionalidade na versão anterior. O serviço de pedidos representa o ponto de entrada, uma interface REST provê os métodos *POST* e *GET* [24]. Ao realizar um *POST*, criará um evento no Kafka, que será consumido por outros três serviços de validação (validação do pedido, identificação de fraude e reserva de estoque). Sendo que o pedido será validado em paralelo, emitindo *PASS* (válido) ou *FAIL*

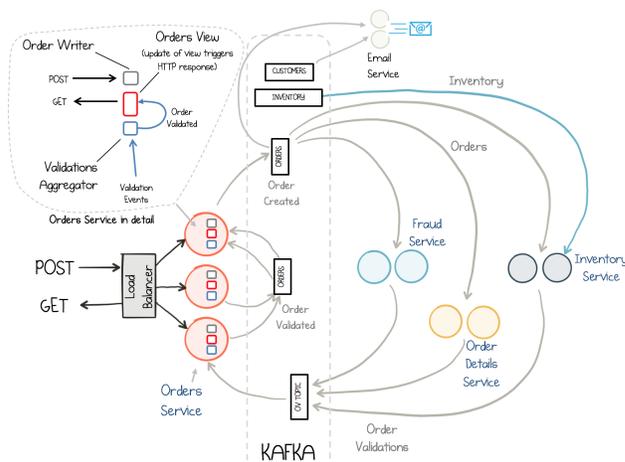


Figure 2: Ilustração esquemática da aplicação utilizada (fonte [24]).

(inválido) baseado no sucesso de cada serviço [24]. Como as validações ocorrem em paralelo, o resultado de cada uma é enviado por meio de seu tópico, separado das demais [24]. Por fim, os resultados são agregados no serviço de pedidos aonde os pedidos são movidos para o status de *PASS* ou *FAIL*, baseado na combinação de resultados [24].

Para que o usuário consulte algum pedido a partir do *GET*, no serviço de pedidos foi criado uma *view* materializada consultável (“Orders view” na Figura 2), usando um armazenamento de estado em cada instância do serviço, de forma que qualquer pedido pode ser requisitado historicamente [24]. Dentre as validações, têm-se três: (i) validação do pedido confere os elementos básicos, como quantidade e preço do próprio pedido; (ii) identificação de fraude rastreia o valor total dos pedidos de cada cliente em uma janela de uma hora, alertando se o limite que configura uma fraude for alcançado; e (iii) reserva de estoque verifica se há unidades disponíveis para esse pedido [24], primeiramente precisa consultar quantas unidades disponíveis há em estoque, após, a quantidade requisitada será reservada pelo tempo necessário até que o pagamento seja concluído.

Table 3: Descrição dos cenários de mudança

Versão	Descrição
V1	Serviço de pedidos
V2	Serviço de validação do pedido
V3	Serviço de identificação de fraude
V4	Serviço de reserva de estoque
V5	Serviço de envio de e-mail ao cliente

5 RESULTADOS

Esta seção apresenta os resultados coletados após a execução da metodologia definida na Seção 4. A Figura 3, Figura 4 e Figura 5 apresentam os resultados obtidos nos cenários de evolução. A Tabela 4 traz indicadores estatísticos sobre os resultados, incluindo o desvio padrão, mediana, média e a diferença entre as médias.

³<https://github.com/confluentinc/kafka-streams-examples/tree/6.0.0-post/src/main/java/io/confluent/examples/streams/microservices>

Table 4: Resultados obtidos

Atributos	Métricas	Arquitetura	Desvio padrão	Mediana	Média	Diferença
SoC	CDC	KAFKA	1.72	3	3.2	40.74%
		REST	2.50	4	5.4	
	CDO	KAFKA	13.11	10	15.2	22.37%
		REST	9.02	10	11.8	
	CDLOC	KAFKA	2.19	4	4	48.72%
		REST	2.56	8	7.8	
Coupling	Dep_Out	KAFKA	3.83	4	5.4	70.37%
		REST	0.80	1	1.6	
	Dep_In	KAFKA	3.83	4	5.4	85.19%
		REST	0.75	1	0.8	
Cohesion	H	KAFKA	0.04	1.2075	1.207	20.94%
		REST	0.02	0.966	0.9543	
Complexity	R	KAFKA	11.43	56	57.8	85.12%
		REST	5.46	6	8.6	
Size	LOC	KAFKA	116.17	174	233	43.35%
		REST	93.11	100	132	
	NumAttr	KAFKA	4.83	8	9.2	41.30%
		REST	3.50	4	5.4	
	NumOps	KAFKA	13.50	10	15.4	12.99%
		REST	8.73	11	13.4	

5.1 Separação de Interesses e Acoplamento

A Tabela 4 mostra os resultados dos efeitos da arquitetura-dirigida a eventos na separação de interesses através da perspectiva de três métricas: CDC, CDO e CDLOC. O KAFKA apresentou resultados menores em comparação à arquitetura REST, considerando as métricas CDC e CDLOC. Isso pode ser notado através das diferenças entre as médias computadas 40,74% e 48,72%, respectivamente. Esse resultado indicava que menos classes (CDC) são afetadas em cada cenário de evolução, já que os serviços são independentes, mas compartilham classes auxiliares (*utils*). Também observam-se números menores na quantidade de transições de interesses sob as linhas (CDLOC). Isso significa que o Kafka promoveu uma melhor modularização de *concerns* considerando componentes e linhas de código. Na Figura 3, observa-se que em todas as versões, ambas métricas apresentaram valores menores para o KAFKA. Porém, o KAFKA apresentou resultados superiores para a métrica CDO, tendo uma diferença entre as médias de 22,37%. A mediana da CDO, por sua vez, não registrou diferença. Na maioria dos cenários foram necessárias mais operações (ou funções) para implementar o serviço de cada cenário na arquitetura dirigida a eventos. Na Figura 3, observa-se que apenas na versão 4, o REST apresentou um valor superior ao Kafka. Logo, essa necessidade foi refletida na métrica CDO.

Considerando as variáveis de acoplamento da Tabela 4, observa-se que a arquitetura REST entregou menor acoplamento do que o KAFKA. Ambas métricas Dep_out e Dep_in, apresentam diferenças entre as médias computadas de 70,37% e 85,19%, respectivamente. Contudo, ao analisar a Figura 4 percebe-se valores bastante elevados para ambas métricas no KAFKA na primeira versão. Mesmo que nos demais cenários os valores sejam superiores, a diferença não é tão alta quanto as diferenças entre as médias. Alias, não só um menor acoplamento na arquitetura REST, mas também uma menor variação em cada cenário de evolução, conforme observado na Figura 4.

A maior separação de interesses está em linha com a característica de microsserviços [6, 17], na qual cada serviço é independente. Isso promove uma maior modularização da aplicação, a qual se beneficia em cenários onde há alterações de comportamento ou evolução da aplicação, como a adição de novas *features* [21]. Além disso, também evita-se a degradação da qualidade da aplicação, garantindo ao usuário acesso ao serviço sem interrupções. Ou ainda, traz benefícios a performance do projeto [26]. Contudo, a independência dos serviços tem seu preço, resultando em mais funções, atributos e classes auxiliares. O maior acoplamento na arquitetura dirigida a eventos pode ser explicado pela necessidade de classes auxiliares. Indispensáveis para não gerar duplicação de código, já que as funções atendem diferentes contextos, caso não existissem gerariam mais código em cada serviço, consequentemente complexidade [21].

Resultados observados 1: As médias de separação de interesses do KAFKA, das métricas CDC e CDLOC apresentaram diferenças de 40,74% e 48,72%, respectivamente. Além de valores menores em todas as versões. Evidenciando a maior separação de interesses se comparado ao REST. Entretanto, as médias de acoplamento mostram diferenças acima dos 70% para o KAFKA. Isso mostra que as dependências, tanto internas quanto externas são maiores no KAFKA.

5.2 Complexidade, Coesão e Tamanho

A Tabela 4 traz os resultados dos efeitos da arquitetura-dirigida a eventos em relação à complexidade, coesão e tamanho ao longo dos cenários de evolução. Os resultados apontam que o KAFKA obteve resultados superiores em comparação à arquitetura REST, observando as variáveis de complexidade e coesão computadas através das métricas H e R, respectivamente. Os números obtidos apontam

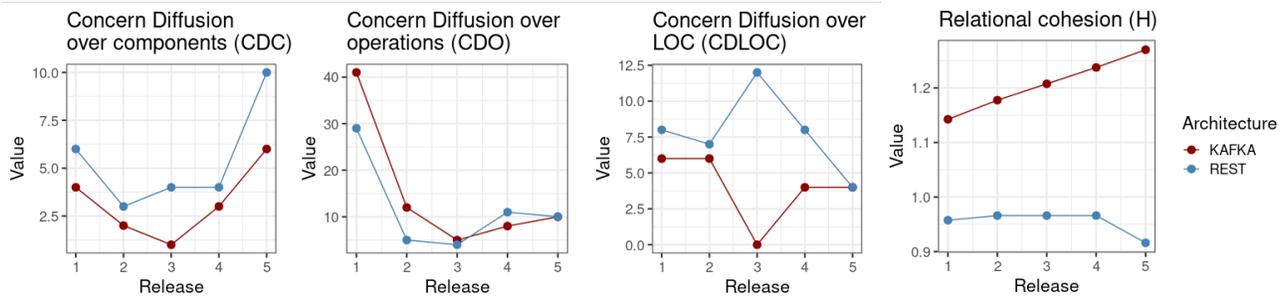


Figure 3: Resultados coletados considerando as métricas CDC, CDO, CDLOC e H.

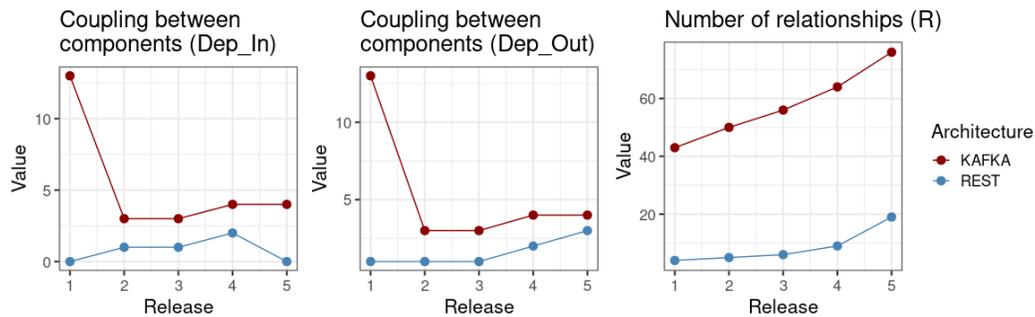


Figure 4: Resultados coletados considerando as métricas Dep_In, Dep_Out e R.

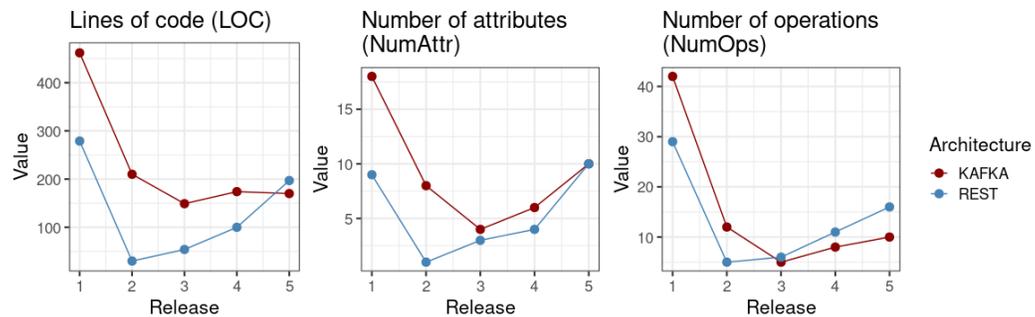


Figure 5: Resultados coletados considerando as métricas LOC, NumAttr e NumOps.

que as diferenças entre as médias computadas das métricas H e R foram 20,70% e 85,12%, respectivamente. Embora os resultados da coesão relacional (H) tenham favorecido ao Kafka, a complexidade deve ser um fator a ser avaliado.

Analisando os resultados através da perspectiva da variável tamanho, eles indicam que o KAFKA obteve resultados superiores em comparação à arquitetura REST. Esse achado foi quantificado utilizando as métricas LOC, NumAttr e NumOps, as diferenças entre as médias dessas métricas foram 43,34%, 41,30% e 12,99% respectivamente. Por outro lado, analisando a Figura 5, observa-se valores bastante elevados nos três primeiros cenários para as métricas NumAttr e NumOps. Após, a métrica NumAttr apresenta valores elevados, mas bem próximos, e, a métrica NumOps a partir do terceiro cenário valores inferiores.

Tais resultados eram esperados após a implementação dos cenários, pois a quantidade de linhas e operações no KAFKA foram bastante elevadas, principalmente nos primeiros cenários. Isso pode ser explicado pelo fato de que na arquitetura REST precisa-se de uma quantidade menor de elementos (classes) para a construção da aplicação. Por outro lado, no KAFKA observou-se a necessidade de criar várias classes auxiliares aos serviços. Bem como funções e atributos para configurações de cada serviço, tanto na sua inicialização como operação. Por consequência, essa necessidade refletiu em maiores valores na complexidade e coesão — quais estão de alguma forma ligados ao aumento do tamanho da aplicação sem um gerenciamento adequado, por exemplo, entre o número de relacionamentos entre as classes e interfaces por pacote. Em suma, aplicações maiores tendem a também serem mais complexas, além

de que outros estudos também perceberam maior complexidade na arquitetura dirigida a eventos [6, 17].

Resultados observados 2: *As médias de complexidade e tamanho favoreceram o REST, destacando-se as métricas LOC e NumAttr com diferenças de 43,45% e 41,30%, respectivamente. Oriundos em sua maior fração de classes auxiliares, e, funções/atributos necessários ao funcionamento de cada serviço do KAFKA. Diretamente ligado a complexidade, medido pela métrica R que apresentou uma diferença de 85,12% a favor do REST. Por outro lado, a média de coesão (H), favoreceu o KAFKA em 20,94%.*

5.3 Discussão

A arquitetura dirigida a eventos obteve bons resultados na separação de interesses. Por outro lado, ao analisar os resultados obtidos pelo conjunto de métricas definidos, percebeu-se que a arquitetura REST obteve melhores resultados de uma maneira geral. Uma melhor separação de interesses viabiliza uma melhor modularização das funcionalidades, algo altamente recomendável quando busca-se ter serviços independentes e que evitem concatenação de modificações ao serem alterados [21]. A seguir, são abordadas algumas das características da arquitetura dirigida a eventos, por consequência do KAFKA também.

Livre de coordenação por design. O controle das consistências fica comprometido, ao enviar dados para muitos serviços diferentes. Uma forma de mitigar isso é adotar o princípio de *single writer*, pois consiste em delegar a tarefa de propagar eventos de um tipo específico, um único serviço [24]. Tal controle é mais evidente, ao replicar os eventos entre replicas (ou instâncias), algo normal e comum por questão de garantia. Assim, ao centralizar no *single writer* criasse um “túnel” de consistências, validações e outros escritores por meio de um único fluxo. Tanto que, uma das características da arquitetura é mover os dados (ou eventos) enquanto opera neles [12], no KAFKA através do *Kafka Streams* e KSQL — pontos centrais para processar dados dentro de programas clientes.

Eventos também são uma ferramenta útil ao *design* do sistema, fornecendo notificação, transferência de estado e desacoplamento [12]. Embora o KAFKA apresentou maior acoplamento, segundo as métricas *Dep_in* e *Dep_out* da Figura 3. Visto que, há várias dependências por serviços auxiliares, comuns a todos os serviços. Contudo, no desenvolvimento dos cenários as dependências se limitaram a classes auxiliares, e, não aos outros serviços. Inclusive, no Euphoria [22] foi identificado o desacoplamento, pois mudanças em um módulo não determinou mudanças em módulos adjacentes, desde que a interface permanecesse a mesma. Portanto, apesar das métricas reportarem maior acoplamento, não observou-se a concatenação de modificações nos serviços.

Source of truth. Caso os eventos sejam armazenados na ordem de criação e não sofrerem alterações. O *log* fornece uma linha de tempo do que exatamente aconteceu [25]. Logo, esse comportamento torna o fluxo de eventos a *source of truth*, já em sistemas tradicionais é o banco de dados. Assim como, a aplicação explorada consiste em vários serviços cooperando em um único fluxo de negócio, onde cada um faz seu trabalho de processar eventos

e criar novos — eventos são o ponto central do sistema [25]. Pois, o *log* disponibiliza os dados centralmente como a *shared source of truth* aos serviços, porém com um contrato simples, o que mantém os serviços fracamente acoplados. Desta forma, o CQRS é um dos pontos chave, pois ele separa o caminho de gravação do caminho de leitura e os vincula a um canal assíncrono. Seu funcionamento é no estilo *log write-ahead*, as inserções e atualizações são imediatamente registradas sequencialmente no disco. Desse modo, torna o processo bastante rápido pois não precisa esperar pelo lento processo de atualizar várias estruturas como tabelas, índices e assim por diante [25].

A garantia do devido funcionamento da aplicação, requer alguns cuidados. Primeiro, a ordem dos eventos deve ser preservada, ao serem enviados sempre a mesma partição isso garante a correta ordenação. Já para garantias de ordem global, deve-se utilizar um tópico de partição única, isso vai limitar a taxa de transferência, se bem que é o suficiente para a maioria dos casos [25]. Segundo, o reenvio de eventos, causado por alguma falha de rede, *garbage-collector* demorado, falha ou algo semelhante. Como as mensagens são enviadas em lotes, deve-se ter cuidado de enviar os lotes um por vez, por máquina de destino, a fim de evitar a reordenação dos eventos [25]. Em vista disso, o Kafka fornece poderosas funcionalidades, mas que precisam ser devidamente configuradas, e, cuidados no desenvolvimento dos serviços a fim de evitar efeitos indesejáveis e difíceis de rastrear. Inclusive, em [17] percebeu-se como desvantagem, o complexo fluxo de eventos acarretado pelo número de microsserviços. Portanto, a arquitetura dirigida a eventos apresenta maior complexidade, assim, seu desenvolvido e manutenções podem vir a requerer maior esforço e devido gerenciamento conforme evolução.

Stream processing. Há bastante tempo os sistemas de mensagens são utilizados para trocar eventos entre sistemas, porém apenas recentemente eles começaram a serem utilizados na camada de armazenamento. Isso cria um estilo arquitetônico interessante. Ao analisar a estrutura de um banco de dados, encontram-se uma série de componentes, análogo a uma caixa preta. Sendo assim, essa estrutura pode ser decomposta utilizando processamento de fluxo (*streaming*) e esses componentes existirem em locais diferentes, unidos pelo *log* [25]. Logo, surgiram plataformas de *streaming*, como o KAFKA que processa o fluxo de eventos, armazenar os eventos em estrutura de *log* e disparar uma cascata de serviços inscritos a tópicos. Isso permite aplicativos e serviços incorporem lógica diretamente sobre o fluxos de eventos. Além de disponibilizar os recursos de processamento do banco de dados na camada de aplicação, por meio de uma API.

O *Kafka Streams* é a principal API para processamento de *stream* na JVM. Baseado em uma DSL (*domain-specific language*), que fornece uma interface estilo declarativo onde os fluxos podem ser unidos, filtrados, agrupados ou agregados [25]. Bem como, fornece mecanismos de estilo funcional como *map*, *flatMap*, *transform*, *peek*, entre outros [25]. Diante disso, podem ser criados índices ou visualizações, e essas visualizações se comportam como uma forma de cache continuamente atualizado, dentro ou perto de seu aplicativo. A aplicação utilizada faz uso do *Kafka Streams*. Como resultado, em geral as consultas e processamento de eventos na DSL, requerem mais linhas de código, quantificado na métrica LOC da Figura 5. Ao comparar a DSL com *querys SQL*, percebe-se que a DSL é mais

complexa, entretanto, mais poderosa também. Portanto, o uso do *Kafka Streams* gera mais linhas de código, enquanto também entrega mecanismos mais poderosos.

Escalabilidade. Entre as características da arquitetura dirigida a eventos, a altamente escalabilidade demonstra ser a mais desejada [22]. Sendo capaz de suportar diferentes tamanhos de *clusters*, por exemplo 5, 100, 200 ou quantas máquinas forem necessárias [25]. A adição de máquinas se resume em adicionar novas máquinas e rebalancear. Entretanto, podem aparecer problemas de negação de serviços, administráveis com o *cotas*, um serviço para definir a largura de banda a cada serviço. Sua grande vantagem está no seu armazenamento (estrutura de *logs*), pois não utilizam índices que são bastante custosos para se manter. Portanto, sua complexidade é $O(1)$ ao ler e gravar mensagem em uma partição. Além de que, ao ler ou escrever em um tópico é capaz de replicar os eventos para todas instancias definidas. Suas consultas, abordadas anteriormente podem ser em paralelo, dividindo as mensagens de um tópico entre diversos consumidores.

Serviços orientados a eventos devem sempre ser executados com *highly available (HA)*, a menos que não haja requisitos para HA. A principal razão para isso é essencialmente um ambiente autônomo [25]. Se há apenas uma instancia do serviço, ao adicionar uma segunda, a carga será rebalanceada naturalmente. O mesmo acontece caso um nó falhe. Portanto, os serviços herdam a alta disponibilidade e balanceamento de carga naturalmente, o que significa que eles podem escalar horizontalmente lidar com interrupções não planejadas ou realizar reinicializações contínuas sem perder a operabilidade do serviço [25]. Seu maior impacto pode ser observado na Figura 5, através da métrica LOC com valores consideravelmente superiores se comparado ao REST, e, NumOps que também apresentou valores superiores, mas com menor diferença. Isso, se dá pelas várias configurações (LOC) e funções (NumOps) em cada serviço para conexão com a plataforma e configuração dos tópicos utilizados por ele. Tamanhos maiores são um dos fatores que contribuem com a complexidade da aplicação, refletido na métrica R da Figura 4, também observados em [17]. Portanto, a complexidade da arquitetura dirigida a eventos pode ser verificada em diferentes perspectivas.

5.4 Limitações

O estudo exploratório reportado trata-se de um estudo inicial que explora uma área ainda pouco investigada na literatura. Neste sentido, o estudo possui algumas limitações que precisam ser consideradas. Apenas uma aplicação foi considerada no estudo. Priorizou-se o uso de aplicações da indústria e de código aberto que fizessem uso da arquitetura dirigida a eventos. Outras aplicações foram encontradas, além da aplicação alvo, porém não foram consideradas devido à algumas restrições, tais como tamanho pequeno, não eram de código aberto, não adotaram boas práticas de implementação, e não possuem documentação. Assim como argumentado no estudo, aplicações dirigidas a eventos mostram uma complexidade elevada, que também está presente no desenvolvimento, pela sua curva de aprendizado percebida. Sendo esse um dos motivos pelo qual o estudo explora apenas uma aplicação, desenvolver uma além de inviável poderia prejudicar a avaliação dos resultados. Tal dificuldade, pode ser compreendida pela grande diferença se comparado

a arquiteturas convencionais. Conforme explorado anteriormente, não há banco de dados, eventos são gatilhos para serviços e deve-se construir um fluxo de eventos.

6 CONCLUSÃO E TRABALHOS FUTUROS

Arquitetura dirigida a eventos vem sendo adotada na indústria e algumas tecnologias foram propostas para viabilizá-las, tais como o KAFKA. Ela surge como uma alternativa à arquitetura REST para a implementação de sistemas orientados a serviços. Este trabalho reportou um estudo empírico inicial com o propósito de comparar a arquitetura dirigida a eventos e a arquitetura REST — através da perspectiva de separação de interesses, acoplamento, coesão, complexidade e tamanho — ao longo de 5 cenários de evolução de uma aplicação real.

A arquitetura dirigida a eventos, representada pelo KAFKA, apresentou bons resultados quanto à separação de interesses. Por outro lado, as demais métricas não apresentaram resultados melhores do que a arquitetura REST tradicional. Ao proporcionar uma melhor modularidade dos interesses, algumas características podem ser afetadas. Portanto, ao adotar uma arquitetura dirigida a eventos é preciso analisar tais vantagens e desvantagens apontadas neste estudo. Como trabalhos futuros, pretende-se realizar: (1) aumentar o número de métricas contabilizadas, visando aumentar a perspectiva de análise; (2) considerar mais aplicações para replicar o estudo realizado; e (3) coletar mais dados para viabilizar uma análise estatística rigorosa. Esse trabalho pode ser visto como um primeiro passo para uma agenda mais robusta de estudos experimentais relacionados aos efeitos de arquiteturas dirigidas a eventos na modularidade de software.

REFERENCES

- [1] Ameer B. A. Alaasam, G. Radchenko, and A. Tchernykh. 2019. Stateful Stream Processing for Digital Twins: Microservice-Based Kafka Stream DSL. *2019 International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON)* (2019), 0804–0809.
- [2] Eduardo Almentero, Julio Cesar Sampaio do Prado Leite, and C. Lucena. 2014. Towards software modularization from requirements. *Proceedings of the 29th Annual ACM Symposium on Applied Computing* (2014).
- [3] Victor R Basili. 1992. *Software modeling and measurement: the Goal/Question/Metric paradigm*. Technical Report.
- [4] Leandro Ferreira D'Avila, Kleinner Farias, and Jorge Luis Victória Barbosa. 2020. Effects of contextual information on maintenance effort: a controlled experiment. *Journal of Systems and Software* 159 (2020), 110443.
- [5] Carlos Eduardo Carbonera, Kleinner Farias, and Vinicius Bischoff. 2020. Software development effort estimation: A systematic mapping study. *IET Software* 14, 4 (2020), 328–344.
- [6] Hennadii Falatiuk, Mariya Shirokopetleva, and Zoia Dudar. 2019. Investigation of Architecture and Technology Stack for e-Archive System. *2019 IEEE International Scientific-Practical Conference Problems of Infocommunications, Science and Technology (PIC S&T)* (2019), 229–235.
- [7] Kleinner Farias. 2016. Empirical evaluation of effort on composing design models. *arXiv preprint arXiv:1610.09012* (2016).
- [8] Kleinner Farias, Alessandro Garcia, and Carlos Lucena. 2012. Evaluating the impact of aspects on inconsistency detection effort: a controlled experiment. In *Int. Conf. on Model Driven Engineering Languages and Systems*. Springer, 219–234.
- [9] Kleinner Farias, Alessandro Garcia, and Carlos Lucena. 2014. Effects of stability on model composition effort: an exploratory study. *Software & Systems Modeling* 13, 4 (2014), 1473–1494.
- [10] Kleinner Farias, Alessandro Garcia, and Jon Whittle. 2010. Assessing the impact of aspects on model composition effort. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*. 73–84.
- [11] Kleinner Farias, Alessandro Garcia, Jon Whittle, Christina von Flach Garcia Chavez, and Carlos Lucena. 2015. Evaluating the effort of composing design models: a controlled experiment. *Software & Systems Modeling* 14, 4 (2015), 1349–1365.

- [12] L. Fiege, Gero Mühl, and F. Freiling. 2002. Modular event-based systems. *The Knowledge Engineering Review* 17 (2002), 359 – 388.
- [13] R. Fielding. 2000. Architectural Styles and the Design of Network-based Software Architectures"; Doctoral dissertation, Vol. 7. University of California, Irvine Irvine.
- [14] Eduardo Figueiredo, Nelio Cacho, Claudio Sant'Anna, Mario Monteiro, Uira Kulesza, Alessandro Garcia, Sergio Soares, Fabiano Ferrari, Safoora Khan, Fernando Castor Filho, and Francisco Dantas. 2008. Evolving software product lines with aspects. In *2008 ACM/IEEE 30th International Conference on Software Engineering*. 261–270. <https://doi.org/10.1145/1368088.1368124>
- [15] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and Arndt von Staa. 2005. Modularizing design patterns with aspects: a quantitative study. *LNCS Trans. Aspect Oriented Softw. Dev.* 1 (2005), 36–74.
- [16] Syeda Uzma Gardazi and A. A. Shahid. 2009. Survey of software architecture description and usage in software industry of Pakistan. *2009 International Conference on Emerging Technologies* (2009), 395–402.
- [17] Rodrigo Laigner, M. Kalinowski, P. Diniz, Leonardo Barros, Carlos Cassino, Melissa Lemos, Darlan Arruda, Sérgio Lifschitz, and Yongluan Zhou. 2020. From a Monolithic Big Data System to a Microservices Event-Driven Architecture. *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)* (2020), 213–220.
- [18] Kleinner S Farias Oliveira and Toacy Cavalcante. 2007. A guidance for model composition. In *Int. Conf. on Software Engineering Advances (ICSEA 2007)*. IEEE, 27–27.
- [19] D. Parnas. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM* 15 (1972), 1053–1058.
- [20] Yu Ping, K. Kontogiannis, and Terence C. Lau. 2003. Transforming legacy Web applications to the MVC architecture. *Eleventh Annual International Workshop on Software Technology and Engineering Practice* (2003), 133–142.
- [21] C. Sant'Anna, C. Lucena, and A. Garcia. 2008. On the Modularity of Aspect-Oriented Design: A Concern-Driven Measurement Approach.
- [22] O. Schipor, Radu-Daniel Vatavu, and J. Vanderdonckt. 2019. Euphoria: A Scalable, event-driven architecture for designing interactions across heterogeneous devices in smart environments. *Inf. Softw. Technol.* 109 (2019), 43–59.
- [23] Stefano Spaccapietra, Salvatore March, and Yahiko Kambayashi. 2003. *21st Int. Conf. on Conceptual Modeling Tampere, Finland, October 7–11*.
- [24] B. Stopford. 2017. Building a Microservices Ecosystem with Kafka Streams and KSQL. Disponível em: <<https://www.confluent.io/blog/building-a-microservices-ecosystem-with-kafka-streams-and-ksql/>>. Acesso em: 25 outubro 2020.
- [25] Ben Stopford. 2018. *Designing Event-Driven Systems*. O'Reilly Media, Incorporated.
- [26] Girish H. Subramanian, J. Jiang, and Gary Klein. 2007. Software quality and IS project performance improvements from software development process maturity and IS implementation strategies. *J. Syst. Softw.* 80 (2007), 616–627.
- [27] Simon Tragatschnig and U. Zdun. 2015. Modeling Change Patterns for Impact and Conflict Analysis in Event-Driven Architectures. *2015 IEEE 24th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises* (2015), 44–46.
- [28] Roger Gonçalves Urdangarin, Kleinner Farias, and Jorge Barbosa. 2021. Mon4Aware: A multi-objective and context-aware approach to decompose monolithic applications. In *XVII Brazilian Symposium on Information Systems*. 1–9.
- [29] W. Zhou, L. Li, M. Luo, and W. Chou. 2014. REST API Design Patterns for SDN Northbound API. *2014 28th International Conference on Advanced Information Networking and Applications Workshops* (2014), 358–365.