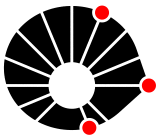

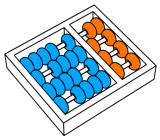

Arquitetura para Condução de Múltiplos Experimentos Online

R. G. Pimenta *B. B. N. França*

Relatório Técnico - IC-PFG-19-17
Projeto Final de Graduação
2019 - Julho

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Arquitetura para Condução de Múltiplos Experimentos Online

Rafael Gois Pimenta* rafaelgpimenta@gmail.com

Breno Bernard Nicolau de França* breno@ic.unicamp.br

Julho 2019

Resumo

Com avanços em temas relacionados à experimentação contínua como abordagem para desenvolvimento de *softwares*, conseguimos oferecer entrega de valor baseando-se em dados colhidos das experiências dos usuários. É com base nisso que este projeto tem como objetivo apresentar uma proposta de arquitetura completa capaz de conduzir múltiplos experimentos em paralelo. Para isso, foi feito um levantamento dos componentes que deveriam estar presentes na arquitetura, exploramos os principais meios utilizados atualmente para a implementação de experimentos e apresentamos um guia de como o Istio pode ser utilizado para direcionar os usuários para o experimento que eles devem participar. Concluindo, podemos reforçar a capacidade que arquiteturas baseadas em microsserviços têm para a condução dos experimentos e que um controle bem estabelecido sobre a rede desses serviços facilita a execução de experimentos em paralelo ao usarmos o paradigma de um *Service Mesh*, porém desafios estatísticos ainda devem ser aprofundados para tratar todo o desafio envolvendo esse paralelismo. Além disso, mostramos que aplicar uma estrutura como a proposta estabelece custos adicionais tanto financeiros como de performance e que devem ser temas abordados em pesquisas futuras.

Abstract

With advances in topics related to continuous experimentation as an approach to software development, we have been able to offer value delivery based on data collected from users' experiences. It is based on this that this project aims to present a complete architecture proposal capable of conducting multiple experiments in parallel. For this, we gathered the components that should be present in the architecture, explored the main methods currently used for implementing experiments and presented a guide of how Istio can be used for directing users to the experiment that they should participate in. In conclusion, we can reinforce the ability of microservices-based architectures to conduct experiments and that a well-established control over the network of these services contributes to the execution of experiments in parallel when using the Service Mesh paradigm, but statistical challenges still need to be deepened with the intention of addressing the whole challenge involving this parallelism. In addition, we show that applying a structure as the one proposed establishes additional costs, both financial and performance, that should be addressed in future research.

*Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP.

1 Introdução

O processo de experimentação remete a um dos instintos mais primitivos da humanidade, a curiosidade. Quando falamos em experimentos, podemos pensar desde de processos bem estruturados e controlados que tenham como objetivo encontrar um remédio super eficiente para curar uma doença grave até um bebê explorando por tentativa e erro a maneira mais eficiente de chamar a atenção de seus pais a fim de atenderem o seu desejo. Como não podia ser diferente, na computação esse tópico também se faz muito presente seja através do lançamento lento de uma funcionalidade para podermos analisar os efeitos que ela provoca no sistema (liberações Canário), como através da comparação de duas variantes no software para ver quem se sai melhor pelas métricas que foram decididas (teste A/B), e também outras situações.

O avanço de temas relacionados a engenharia de software, nos convida a buscar e implantar as melhores práticas referentes ao planejamento e gestão de um projeto com metodologias que se prezam por oferecer agilidade e a entrega constante de valor sobre um produto, como é o caso no *framework* SCRUM, que oferece a adaptabilidade necessária no meio do processo de desenvolvimento, e nas metodologias de *Lean software development*, que tem como base o sistema Toyota de produção. Além disso, temas recentes de estudo, que guiaram o desenvolvimento deste projeto, abordam métodos para uma experimentação contínua que está relacionada à cultura de desenvolver software fundamentando-se em dados e evidências, onde as modificações no seu produto final surgem a partir de hipóteses validadas com experimentos onde conseguimos mesurar com mais precisão o impacto dessas mudanças.

Para o desenvolvimento do projeto, identificamos os relatos bem sucedidos de empresas [3, 16] e os requisitos identificados como importantes através de estudos empíricos no assunto [6] com o intuito de propor uma arquitetura capaz de viabilizar o processo de experimentação contínua, trazendo produtos de alta qualidade para os consumidores, e que ao mesmo tempo se preocupa em agilizar a quantidade de valor entregue por disponibilizar a execução simultânea de múltiplos experimentos. Além disso, foi feita uma análise prática das principais formas de implementação de experimentos, segundo pesquisas realizadas por Shermann et al. [14, 15], para identificar boas práticas e tecnologias a serem embutidas na nossa arquitetura.

2 Justificativa

A área de experimentação contínua é identificada como um passo seguinte ao CI/CD [15], trazendo uma forma sistemática de entregar valor sobre um produto se baseando em dados e nos interesses dos consumidores, ao invés de puramente suposições [6]. Além disso, temos a situação atual do mercado onde temos resultados interessantes alcançados pela indústria sobre temas como a experimentação online [3, 16], mas sem respostas de como construir uma infraestrutura para alcançar de fato tais resultados e ao mesmo tempo temos o cenário em que, segundo respostas obtidas por pesquisas [14], o principal obstáculo para a condução de experimentos é a falta de uma arquitetura de software adequada.

Adicionalmente, o projeto busca viabilizar a execução de experimentos em paralelo, pensando na dificuldade ressaltada na literatura [6, 13] e que com isso alcançaremos uma evolução mais rápida dos produtos e uma melhor aceitação à cultura de experimentação no ponto de vista de negócios por oferecer um retorno sobre investimento também mais rápido.

3 Objetivos

Esse projeto pretende explorar arquiteturas de *software* e ferramentas existentes para experimentação online (por exemplo, testes A/B, liberações Canário e Blue-Green) com objetivo de propor uma arquitetura a ser integrada em uma aplicação-alvo para a condução de múltiplos experimentos com usuários reais em paralelo.

4 Revisão da Literatura

A exploração dos principais conteúdos para o desenvolvimento do projeto pode ser dividido em duas etapas: Buscar os resultados que uma arquitetura de experimentação pode fornecer dentro de experiências reais e estudar conceitos base para conduzir nosso caminho até os mesmos.

Referente aos resultados julgados relevantes temos os casos de duas empresas Uber [3, 7] e Grab [16] que demonstram, através de seus próprios blogs de tecnologia, o que atualmente elas são capazes de fazer nas respectivas plataformas de experimentação. Em ambas empresas vemos que o desenvolvimento dessas plataformas garantiu um salto no número de experimentos realizados, disponibilizando ferramentas de análises robustas para seus cientistas de dados.

Fagerholm et al. [6] destacam benefícios gerados pela experimentação contínua, identificam os requisitos necessários para uma entrega rápida e constante de valor através de experimentos e estabelecem um modelo cíclico com detalhamento das suas etapas de construção, medição e aprendizagem. Além disso tem-se o modelo aplicado em casos de estudos que suportam a ideia da necessidade de uma integração sistemática de todas as esferas de um produto e seu desenvolvimento.

Shermann et al. [14, 15] mostram as principais técnicas para implementação de experimentos de baseando-se em uma série de entrevistas realizadas com empresas de diferentes seguimentos, levantando também os obstáculos e pontos negativos para cada técnica. Também podemos observar que existe um ponto de vantagem para a arquitetura de microsserviços e aplicações Web.

Além disso, Shermann et al. [13] chama a atenção para o fato de que, apesar de empresas conduzirem experimentos nos seus produtos, muitas delas baseiam-se na intuição tanto quando vão especificar um experimento quanto quando vão analisar seus resultados e a partir disso eles se propõe à mostrar um caminho para as companhias, independente do tamanho ou área, migrarem para um ambiente de experimentação explícita, automatizada e com base em dados.

5 Metodologia

Nesta seção, apresentamos as atividades para desenvolver uma arquitetura de experimentação online capaz de operar experimentos em paralelo. Primeiramente, apresentamos um levantamento sobre alguns componentes considerados como essenciais para esta experimentação online. Em seguida, pegamos o componente sobre condução de usuários à participação de um experimento e detalhamos duas técnicas comuns de implementações: *feature toggle* e controle de tráfego [14, 15]. Tendo isso, observamos e destacamos as principais características dessas abordagens, levando também em consideração conclusões tiradas durante o estágio do autor na empresa Nexoos. Finalmente, descrevemos a nossa proposta de arquitetura para uma plataforma de experimentos com suporte a múltiplos experimentos em paralelo.

5.1 Levantamento de Requisitos para Múltiplos Experimentos Online

Para apresentarmos uma proposta de arquitetura voltada para experimentação que se aproximasse do cenário que pode ser vislumbrado através de blogs das empresas de tecnologia [3, 16] no tocante à simplificação em criar, manusear, avaliar e agilizar experimentos, tomamos como um meio para alcançar esse cenário a infraestrutura proposta por Fagerholm et al. [6].

Com base nos resultados apresentados em blogs [3, 16], uma plataforma de experimentação integrada a aplicação traz benefícios significativos para os *stakeholders* do produto, pois a proposta é que tal plataforma seja capaz de proporcionar um avanço significativo na agilidade com que funcionalidades são lançadas, com decisões fundamentadas em dados.

Dentro da infraestrutura que tomamos como base, o primeiro ponto de atenção é que vamos partir do pressuposto que a aplicação já deve estar organizada a ponto de utilizar as práticas de *continuous integration* (integrar frequentemente as modificações no código de um projeto com a construção e execução de testes automatizados), *continuous delivery* (uma prática de desenvolvimento onde o software é construído de forma a sempre estar disponível para ser lançado em produção) e *continuous deployment* (a prática de automação do lançamento de versões estáveis do software para produção, envolvendo inúmeras etapas de testes). Essas práticas pavimentam o caminho para a experimentação contínua [15].

Além dessas práticas, é necessário uma estrutura de armazenamento dos dados para os experimentos, tais como os dados coletados da aplicação, os planos do experimento e o conhecimento ou aprendizado obtido. Com essa organização, podemos desenvolver uma ferramenta de análise robusta e com possibilidade de realizar automaticamente pré-processamentos dos dados, por exemplo detectar irregularidades, reduzir a variância e corrigir dados tendenciosos, como realizado na Uber [3].

Outro ponto importante para a execução de experimentos é a forma de selecionar e direcionar os usuários para participar dos grupos de controle ou tratamento. Nesse ponto, é importante garantir a consistência na experiência do usuário, impedindo, por exemplo, que a experiência dele seja prejudicada por uma alternância constante entre experimentos toda vez que ele acessa a aplicação, um cenário que normalmente não deveria acontecer, além da confiabilidade do experimento, uma vez que a coleta de informações depende da alocação

correta e planejada dos usuários para os experimentos. Ainda, é importante o cuidado com a seleção dos participantes na execução de diferentes experimentos em paralelo, pois deve se garantir que a seleção de um grupo para um experimento não tenha um impacto negativo nos experimentos já em execução e dos planejados [13].

5.2 Técnicas para Experimentação Online

5.2.1 *Feature toggle*

Do ponto de vista da implantação dentro do sistema, essa técnica pode ser considerada a mais simples ao compararmos com a técnica discutida na próxima seção, isto porque ela se resume na adição de um bloco condicional embutido no código fonte. Ou seja, constrói-se o comportamento a ser testado junto com o comportamento presente, passando como parâmetro o usuário e, dessa forma, durante a execução do programa, a aplicação decidirá qual comportamento executar, de acordo com as regras definidas no banco de dados para esse usuário (ver Figura 1).

```
1 se estaHabilitado('feature', usuario)
2   # codigo da nova feature
3 senao
4   # codigo com o comportamento padrao
5 fim se
```

Figura 1: Representação do uso de *feature toggle*

Segundo o estudo realizado por Shermann et al. [15], o uso de *feature toggles* é a técnica mais utilizada nas empresas participantes do estudo (36%), o que pode ser explicado também pela quantidade de bibliotecas prontas em diversas linguagens de programação, como é o caso da biblioteca Flipper (desenvolvida para Ruby) utilizada dentro da *startup* Nexoos. Atualmente, na Nexoos, o uso de *feature toggles* tem como foco realizar a implantação Canário de uma funcionalidade o que permitiu também a realização mais frequente de liberações, mesmos antes da finalização de *sprints*, e com independência entre os diferentes times de desenvolvedores. Porém, apesar da simplicidade no uso de *feature toggles*, é possível identificar pontos negativos.

O primeiro ponto negativo observado no estágio dentro da Nexoos, foi a quantidade de código referente a experimentos antigos que por vezes fica esquecido na aplicação e também pela complexidade adicional no código que, apesar de ser apenas um bloco condicional a mais, atrapalha a eficiência no desenvolvimento, principalmente aos novos ingressantes. Ainda, existe a questão financeira que não é avaliada precisamente, como por exemplo o tempo extra gasto por desenvolvedores para realizar tarefas de manutenção no código [8]. Shermann et al. [15] também relatam a experiência de uma empresa que chegou a um nível insustentável, pois estavam tendo que manter e experimentar cerca de 150 features no código. Então, ao iniciar a implantação dessa técnica, é importante entender essa característica e se preocupar desde o começo com medidas para amenizá-la como não exagerar na utilização, revisar com frequência, manter um tempo de vida curto para o

experimento (o que pode ser um conflito no caso de um experimento mais robusto que requer um tempo maior para compilação dos dados) e contabilizar os custos gerados [1].

5.2.2 Controle de tráfego em tempo real

Essa técnica para experimentação é implementada em nível de rede de computadores, ou seja, não adiciona novas linhas de código na sua aplicação para executar experimentos. Dessa forma, permite que a administração referente a selecionar entre os diferentes experimentos rodando em paralelo aconteça fora da sua aplicação com uma configuração bem definida de um "roteador" que direciona as requisições dos usuários para um definido experimento.

De acordo com Shermann et al. [15], vemos que esse tipo de implementação está presente em 30% das empresas entrevistadas, menor que a técnica anterior, mas ao olharmos somente para empresas aplicações Web notamos que a técnica tem um alcance maior e com a mesma porcentagem da implementação envolvendo *feature toggles* (45%), o que é totalmente coerente, pensando por exemplo que aplicações voltadas para dispositivos móveis geralmente não conseguem tirar um bom proveito desse controle de tráfego.

Considerando as ferramentas disponíveis que implementam essa técnica, algumas utilizam soluções de mais "baixo nível" como HAProxy e outras utilizam soluções mais sofisticadas envolvendo *sevice mesh* como o Istio [14]. Escolhemos a última abordagem para realizar o estudo por se tratar de uma tecnologia que vem ganhando notoriedade recentemente [17].

5.2.3 *Service Mesh*

Quando falamos dessa malha de serviços (*service mesh*), devemos pensar como uma infraestrutura de redes para microsserviços voltada ao gerenciamento de comunicação da aplicação [17] que implementa tratamento de falhas, observabilidade da rede e um roteamento dinâmico [11]. Aqui estamos interessados justamente no último ponto para direcionar as chamadas de um serviço para o experimento correto.

A lógica de um *service mesh* é comumente dividida em duas partes o *control plane* e o *data plane*. O *data plane* fica responsável por interceptar as requisições e pacotes que chegam e saem de cada instância de um serviço, dessa forma, para cada serviço da aplicação teremos um *proxy* dedicado, dessa forma o *proxy* fica responsável pela descoberta do serviço na rede, verificação de integridade, roteamento, balanceamento de carga, autenticação e observabilidade. Por sua vez, o *control plane* trata de gerenciar e configurar todos os *data planes* da nossa malha criando um sistema distribuído. [10]

Dentre as diversas implementações de *service mesh*, escolhemos o Istio por se tratar de um projeto de código aberto e também por ter engajamento dentro de sua comunidade.

5.2.4 Istio

O Istio [9] implementa a infraestrutura de um *service mesh* independente de plataforma e é projetado para funcionar em diversos ambientes, incluindo ambientes da nuvem como é o caso do cluster Kubernetes que usamos no desenvolvimento do projeto. Dessa forma,

o administrador da infraestrutura descreve instruções em arquivos YAML, que serão carregados no Istio através da interface por linha de comando do próprio Kubernetes (*kubectl*). No Istio temos a capacidade de agrupar os experimentos de um serviço através de rótulos a serem descritos ao fazermos o *deployment* com Kubernetes como podemos ver na figura 2 abaixo.

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: my-service-experimentX
  labels:
    app: my-service
    version: experimentX
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: my-service
        version: experimentX
    spec:
      containers:
      - name: my-service
        image: docker-hub/my-service-experimentX
        imagePullPolicy: IfNotPresent
```

Figura 2: Representação de um *deployment* através do Kubernetes, usando rótulos

Dentro da lógica de divisão de um *service mesh*, o *data plane* do Istio utiliza uma versão extendida do Envoy, um *proxy open-source* e de alta performance desenvolvido em C++ que funciona dentro da camada de aplicação no modelo OSI [5], para criar o conjunto de *proxies* que vão intermediar toda a comunicação da rede entre os microsserviços. O modelo de anexar o *proxy* dentro de cada *pod* do Kubernetes permite adicionar as capacidades do Istio dentro de uma aplicação já implantada sem precisar mudar sua arquitetura ou código.

Já o *control plane* é dividido em quatro componentes:

- Galley centraliza as configurações de gerenciamento, isolando os outros componentes de detalhes da plataforma subjacente (Kubernetes, por exemplo). Ele recebe as configurações fornecidas pelo usuário e valida antes de distribuir nos *pods*.
- Mixer é o componente do Istio responsável pela aplicação de políticas e coleta de telemetria nos *proxies* e outros serviços.
- O Citadel permite uma forte autenticação entre os serviços e o usuário final com gerenciamento integrado de identidade e credenciais.
- Pilot é o principal componente de gerenciamento do controle de tráfego da nossa malha

de serviços, crucial para nossa arquitetura de experimentação, pois é responsável pelo ciclo de vida das instâncias do Envoy implementadas na malha de serviço do Istio.

Cada instância do Envoy mantém as informações de balanceamento de carga com base nas informações obtidas do Pilot e em verificações de integridade periódicas das outras instâncias em seu *pool* de balanceamento de carga, permitindo distribuir de maneira inteligente o tráfego entre instâncias de destino enquanto seguem suas regras de roteamento especificadas.

Para configurar o roteamento interno da nossa aplicação entre os *proxies* através da API que fornece as instruções ao Pilot, focamos em dois recursos do Istio: VirtualService e DestinationRules.

O VirtualService define as regras de roteamento entre os serviços e adicionalmente oferece algumas ações que podem ser executadas como, por exemplo, adicionar ou remover *headers* na comunicação e reescrever a URL. Na configuração do VirtualService, podemos definir as mais variadas regras para roteamento como por exemplo definir pesos para cada subconjunto de um serviço sem se preocupar com a escalabilidade dos *pods* [2] (interessante para quem quer fazer liberações canário), espelhar as requisições recebidas por um serviço em outro, mas aqui vamos focar na capacidade de decidir o destino das requisições através de informações no *header* da aplicação. Importante dizer que, no arquivo de configuração correspondente ao VirtualService, a precedência em que as regras aparecem é importante. Dessa forma, a primeira regra na lista terá prioridade. No exemplo da figura 3 a seguir, temos um típico arquivo de configuração de um VirtualService correspondente ao serviço *my-service*, podemos notar que primeiramente tem-se prioridade para fazer a correspondência da regra que busca um *header Experiment* na aplicação contendo o valor X para direcionar a requisição ao ao subconjunto *experimentX*, caso essa regra não seja correspondida a requisição será direcionada por padrão ao subconjunto *default*.

O DestinationRules configura políticas a serem aplicadas às instâncias de um serviço após o roteamento ter sido computado, como políticas de balanceamento, e também define se um subconjunto do serviço receberá o tráfego da nossa malha. Na figura 4 podemos observar primeiramente um arquivo de configuração de DestinationRules para o serviço *my-service* que somente os *pods* marcados com rótulos *default*, *experimentX* e *experimentY* são identificáveis na rede e depois podemos ver como as instâncias dos seus *pods* recebem a distribuição de carga, no caso dos *pods* dos subconjuntos *default* e *experimentY* a distribuição é feita de forma aleatória e para o subconjunto *experimentX* é aplicado o algoritmo *round robin*.

5.3 Arquitetura Proposta

Consideramos como base a arquitetura de microsserviços, estilo em que a experimentação contínua é mais promissora tendo em vista a possibilidade de entregar componentes de forma independente [15], embora precisemos de mais pesquisas sobre essa arquitetura, possibilitando uma análise mais profunda sobre as vantagens e desvantagens. Além disso, considera-se primordial as práticas de CI/CD (*continuous integration*, *continuous delivery* e *continuous deployment*). Dentro da nossa proposta, cada experimento é representado

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: my-service
spec:
  hosts:
  - my-service
  http:
  - match:
    - headers:
        Experiment:
          exact: X
    route:
    - destination:
        host: my-service
        subset: experimentX
  - route:
    - destination:
        host: my-service
        subset: default
```

Figura 3: configuração de um VirtualService no Istio usando *header* da aplicação

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: my-destination-rule
spec:
  host: my-service
  trafficPolicy:
    loadBalancer:
      simple: RANDOM
  subsets:
  - name: default
    labels:
      version: default
  - name: experimentX
    labels:
      version: experimentX
    trafficPolicy:
      loadBalancer:
        simple: ROUND_ROBIN
  - name: experimentY
    labels:
      version: experimentY
```

Figura 4: configuração de um DestinationRule no Istio

por uma *branch* no código dentro do seu repositório onde posteriormente será usada para construção dos respectivos contêineres (orquestrados usando Kubernetes) que executarão tais experimentos.

5.3.1 Armazenamento de dados do experimento

Partindo disso e dos levantamentos feitos anteriormente, desenhamos um banco de dados de experimentos para cada serviço da nossa aplicação (onde todos os experimentos referentes ao serviço em questão devem guardar as informações), pois atualmente este é o padrão (cada serviço ter seu próprio banco de dados) recomendado se tratando de banco de dados da aplicação dentro do ambiente de microsserviços, mas fica a recomendação para um estudo mais aprofundado na área com o objetivo de trazer mais contribuições para a comunidade. Este banco de dados deve, além de reunir todos os dados brutos específicos do experimento, ter uma relação fraca com o banco de dados da aplicação pensando em trazer mais poder ao seu serviço de análise e dar mais suporte a como direcionamos o usuário para o experimento correto, onde falaremos mais detalhadamente na Seção 5.3.3. Importante notar que na nossa proposta, o banco da aplicação não tem a necessidade de saber da existência do banco de experimentos, ou seja, implementar essa estrutura voltada para experimentação não trará mudanças no modelo de sua aplicação.

Sobre os dados enviados aos banco de dados, deixamos essa parte por ser ativamente trabalhada no código fonte do experimento. Claramente, existem os mesmos problemas que envolvem certo grau de complexidade extra no código sendo adicionado, assim como foi mencionado no caso das *feature toggles*, mas, no caso específico destas escritas no banco de dados de experimento, não estaremos adicionando pontos de desvio no fluxo de execução e o código é facilmente identificável para ser eliminado futuramente com a finalização do experimento, seja identificando a diferença com o código original, seja colocando as chamadas ao banco de dados num único *commit*.

5.3.2 Análise e acompanhamento dos experimentos

Referente ao acompanhamento dos experimentos e tomada de decisões, temos basicamente duas abordagens para construir um serviço próprio para análise dos dados nos bancos de dados do experimento. A primeira delas envolve ter vários serviços de análise atrelados a um respectivo serviço da aplicação principal. A segunda opção é compartilhar todos os bancos de experimento com um único serviço de análise. É intuitivo pensar que a segunda opção oferece custos mais baixos por precisar de menos recursos computacionais. Por outro lado, a primeira opção respeita o princípio de isolamento de microsserviços e garante a independência entre os times, e por isso é a opção adotada neste projeto.

O serviço em questão se propõe a ser a principal ferramenta do cientista de dados da organização. Por meio dele, o cientista deverá ser capaz de definir os usuários que ele quer que participe de cada experimento e obter conclusões, tudo através dos dados no banco de experimentos.

5.3.3 Direcionando o usuário para um experimento

Uma importante função que a nossa infraestrutura deve oferecer a fim de possibilitar a realização de múltiplos experimentos é o direcionamento inteligente do usuário para o experimento correto. Ainda, existem serviços da aplicação que dependem de uma consistência referente a sessão do usuário e existem experimentos que desejam analisar o comportamento de um grupo específico de usuários ao longo do tempo, independente do momento que o usuário interagir com o software, por exemplo, em um experimento com um serviço *front-end* da aplicação. Nesse cenário, dificilmente se quer que toda vez que o usuário interaja com o programa, seja apresentado um visual diferente. Para que isso não ocorra, é preciso guardar no banco de dados de experimentos desse serviço a informação de que o usuário participa do experimento ou não para depois direcionarmos para a visualização correta. Sendo assim, levamos em consideração as duas principais abordagens para implementação de experimentos e propomos o uso das duas abordagens.

Primeiro escolhemos a técnica de controle de tráfego como o cerne da nossa arquitetura para construção de uma infraestrutura robusta que simplifique a execução de múltiplos experimentos em paralelo, buscando minimizar os eventuais ajustes no código e aparecimento de débitos técnicos que apareceriam se fossemos implementar vários experimentos dentro do mesmo código. Assim, recomendamos o uso do Istio para controlar a rede de seus microsserviços por permitir a independência entre os diferentes experimentos em execução, além de ser uma tecnologia em crescente adoção e se tratar de uma tecnologia de código aberto. Além disso, propomos o uso de *feature toggles* quando queremos testar várias versões de um experimento, no caso de pequenas variações dentro do mesmo experimento, pensando que, por se tratar de um mesmo contexto experimental e de pequenas variações, não vale a pena criar uma instância para cada versão de um mesmo experimento.

Anteriormente, mostramos a capacidade que o Istio possui para rotear requisições das mais diversas formas possíveis, seja usando métodos puramente aleatórios até especificar diretamente por meio do *header* da aplicação que queremos acionar um subconjunto de um serviço. Mas, de fato, não temos uma forma direta de, através de informações do banco de dados, realizar o direcionamento correto para um experimento. Para solução desse problema, propomos o desenvolvimento de um serviço próprio de coleta das informações do usuário. Dessa forma, no primeiro contato do usuário com a aplicação, esse serviço deve ser acionado a fim de buscar informação nos bancos de dados de experimento sobre quais experimentos ele está participando e repassar esses dados no *header* de acordo com as regras de roteamento criadas através VirtualService do Istio.

Por exemplo, supondo que um dos serviços é alvo de diferentes experimentos simultaneamente e que dependem da faixa de idade do usuário, então, assim que o usuário interagir com a nossa aplicação, o serviço de reconhecimento aciona todos os demais serviços e coleta os dados necessários para identificar os experimentos que o usuário está participando e repassar essa informação no *header*, conforme as regras que definimos no VirtualService. No caso do exemplo, digamos que tenhamos as regras de tal forma que a presença do *header* *experimentAge* com valor *under40* joga o usuário para um experimento e esse mesmo *header* com o valor *over40* joga para outro. Dessa forma, nosso serviço de reconhecimento deve buscar a informação da idade do usuário e, dependendo do resultado, acrescentar os *headers*

requeridos para o roteamento.

Por fim, apresentamos o elemento básico da nossa arquitetura, denominado "célula", voltado para múltiplos experimentos na Figura 5. Nesta, temos um serviço denominado por **A**, no qual estamos rodando ao mesmo tempo dois experimentos diferentes (**X** e **Y**) além do seu código padrão. Note que o subconjunto que não está sob efeito de experimentação não possui ligação com o banco de experimentos e consequentemente não deve existir em seu código nenhum débito técnico referente aos experimentos presentes e passados. Para comparar os experimentos com a execução padrão do serviço deve-se subir um **subconjunto de controle** a partir do código original do serviço, mas com o incremento no código de chamadas ao banco de experimentos para armazenamento de informações julgadas relevantes. Temos na imagem representado em azul e vermelho o fluxo de transferência de dados dentro da "célula". Na imagem ainda temos em laranja as linhas que representam o fluxo da nossa rede de microsserviços, a ser controlada pelo Istio, por onde um usuário tem acesso aos serviços da aplicação ou o cientista de dados tem acesso aos serviços de análise. É através desse serviço de análise que o serviço de reconhecimento proposto anteriormente pode ter acesso à informações sobre qual experimento um determinado usuário participa nessa "célula" e transmitir essa informação pelo *header* da aplicação, garantindo um direcionamento para o subconjunto direto com o Istio.

6 Resultados e discussões

Nessa seção vamos mostrar experiências no controle de tráfego do Istio, trazendo um passo a passo para reprodução, e também levantaremos reflexões a respeito da arquitetura proposta pensando em como pesquisas futuras podem ajudar a trazer valor ao projeto.

6.1 Experimentos com Istio

Durante o projeto também foram feitas algumas experiências dentro do Google Cloud Platform com o Istio, explorando os tutoriais na própria documentação do Istio e com isso resumimos no Apêndice A um passo a passo para a instalação da aplicação **BookInfo**, trazendo uma melhor visualização de como funciona o controle de tráfego do Istio. Para isso, você precisa em primeiro lugar entrar na sua conta no Google Cloud Platform, criar um projeto e guardar o ID dele e exportar no seu terminal na variável **PROJECT_ID**, além disso, exporte em **CLUSTER_NAME** o nome que você deseja dar para seu cluster Kubernetes e em **ZONE_NAME** a zona que ficará seu cluster (e.g. southamerica-east1-a). Por fim instale no terminal a interface de linha de comando do Google Cloud (gcloud).

A aplicação BookInfo é dividida em quatro serviços *productpage*, *details*, *reviews* e *ratings*, sendo que *reviews* apresenta 3 diferentes versões. A versão v1 é a única que não interage com o serviço *ratings*, as versões v2 e v3 faz com que o *rating* do livro aparece com até cinco estrelas sendo que em v2 são estrelas pretas e em v3 vermelhas. No apêndice, temos no passo 26 a aplicação rodando com todo o tráfego sendo direcionado para a versão v1 de cada serviço graças às regras definidas pelo arquivo que configura um VirtualService, assim toda vez que você acessar o endereço de sua aplicação, o mesmo endereço que aparece no passo 23, em seu navegador deve aparecer o resultado como na figura 6.

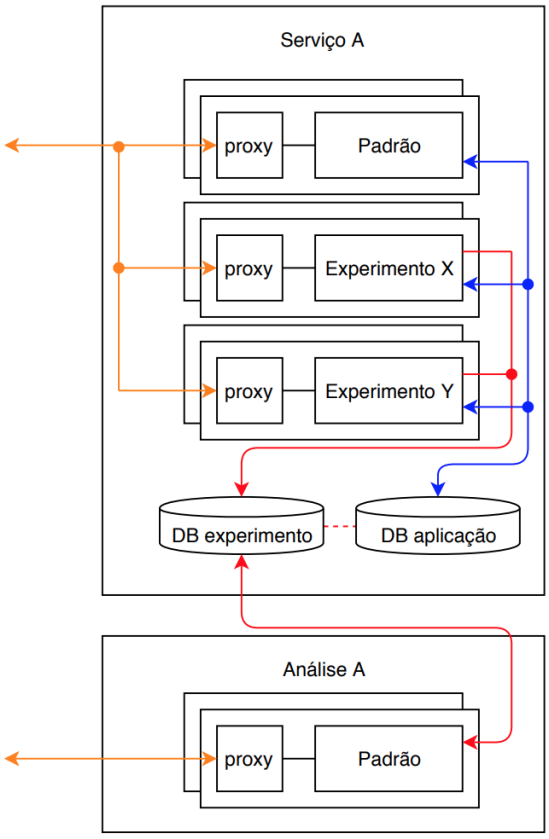

Figura 5: "Célula" da arquitetura para múltiplos experimentos online

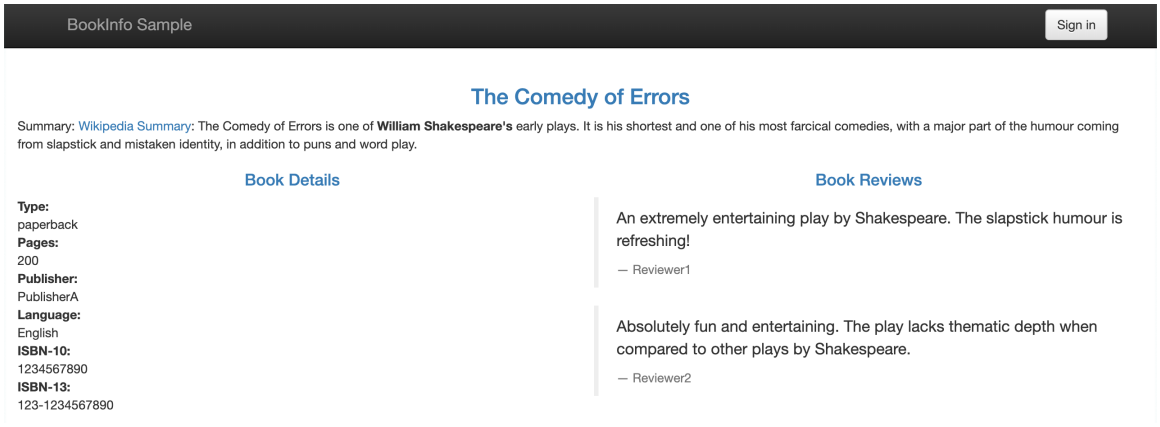

Figura 6: serviço *reviews* sem mostrar nota

No passo 27, configuramos os *proxies* e direcionamos as requisições que chegam no serviço *reviews* para a versão v2 ou v3 com 50% de chances cada. Dessa forma, toda vez que você recarregar a sua aplicação no navegador, você deve observar o aparecimento de estrelas vermelhas ou pretas na área referente à revisão do livro, semelhante às figuras 7 e 8.

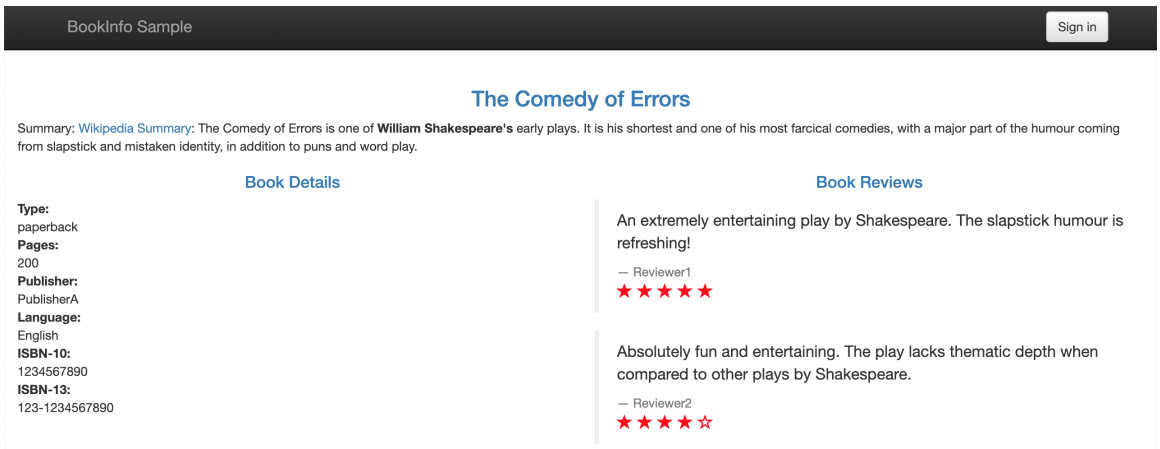

Figura 7: serviço *reviews* mostrando a nota destacada por estrelas vermelhas

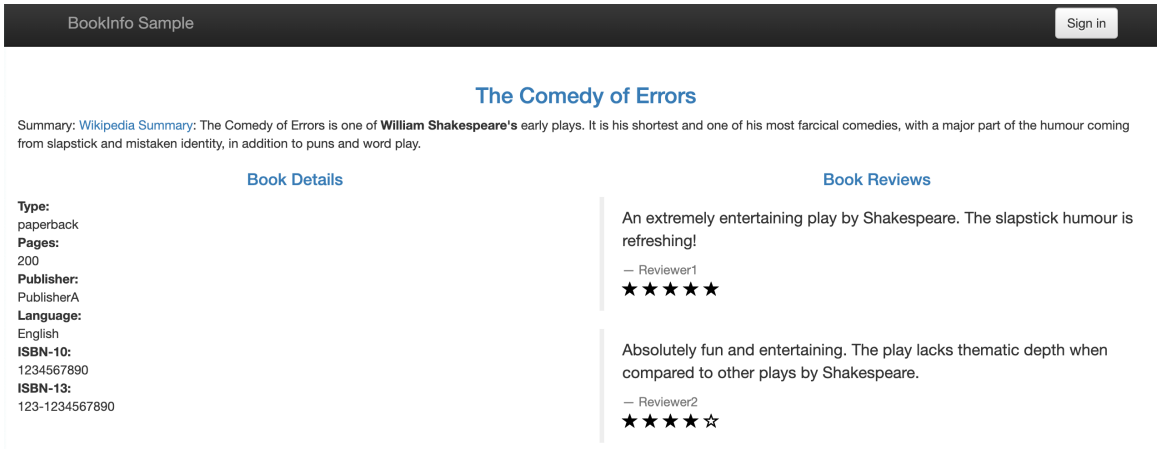

Figura 8: serviço *reviews* mostrando a nota destacada por estrelas pretas

6.2 Discussões a cerca da arquitetura proposta

Primeiramente, sabemos que dentro de um ambiente de mercado tem-se o interesse comum no retorno sobre investimento, dessa forma, uma barreira é levantada quando profissionais na área de negócios não estão interessados em deslocar grande quantidade de recursos, sejam financeiros ou até mesmo o tempo que um desenvolvedor terá que dispor, para o desenvolvimento de experimentos no seu produto com resultados incertos. Sendo assim, antes de se preocupar em desenvolver uma aplicação dentro de uma arquitetura voltada à experimentos, temos que alinhar a cultura da empresa sem fugir da realidade de que o retorno também é importante. Para isso, temos dois artigos que trabalham a favor dessa direção. Em "The RIGHT model for Continuos Experimentation" [6], temos dentro do processo de seu modelo um destaque para a necessidade de identificar e priorizar várias hipóteses para eliminar o máximo possível de incertezas antes de executar o experimento de maior prioridade. Já em "Software Engineering Needs Agile Experimentation" [12] traz no seu manifesto a regra de praticar experimentos seguindo o que é usado por pesquisadores na área de saúde para evitar gastos que um experimento já em larga escala poderia trazer. Além disso, acreditamos que a capacidade em realizar experimentos em paralelo oferece mais velocidade para o retorno sobre o investimento, mas deve-se tomar o cuidado com possíveis interações estatísticas que um experimento, principalmente se mal projetado, pode provocar nos demais.

Pensando na arquitetura em si, devemos nos atentar ao custo gerado em manter uma estrutura desse porte, pois vários serviços extras são adicionados à sua aplicação como os serviços do Istio, os *proxies* que são agregados em todos os *pods*, um serviço de análise para cada serviço da aplicação e um serviço voltado para identificação e busca de informações do usuário. Por isso é interessante a realização de mais pesquisas no assunto, buscando examinar o impacto gerado seja em pequenas *startup* descobrindo seu modelo de negócio ou em grandes empresas com uma grande estrutura de serviços rodando e comparar com soluções envolvendo somente alterações no código fonte, embora essa última solução tenha custos mais subjetivos como o impacto de débitos técnicos. Além disso, devemos estar atentos no impacto correspondente à performance que hoje mostra-se como uma preocupação do Istio [4] no sentido que estão sempre procurando amenizar esse impacto, porém dentro da arquitetura foram introduzidos conceitos mais abstratos no tocante à análise dos experimentos e identificação do usuário onde devemos pensar na forma mais eficiente de desenvolver, testar seu desempenho e ver se existem formas alternativas de organizar tais conceitos.

7 Conclusão

Com a arquitetura proposta, temos que um sistema baseado em microsserviços para aplicação de experimentos ganha força por conta do crescente desenvolvimento tecnológico em torno dessa abordagem, como é o caso do aparecimento de *frameworks* de *Service Mesh* (e.g. Istio) que oferecem um controle de tráfego inteligente dentro da nossa rede de serviços, guiando o usuário para o experimento correto.

Do ponto de vista da viabilidade da arquitetura, sabemos que de fato toda essa infraestrutura agregada na aplicação gera custos financeiros e perda de performance. Por

deixarmos alguns pontos mais subjetivos no projeto, como é o caso do serviço para análise de experimentos e do serviço de identificação do usuário, não temos como dar uma precisão sobre o *tradeoff* entre a variação de custo ou performance que uma arquitetura nos parâmetros do projeto traz e os benefícios em ter toda essa infraestrutura para facilitar a entrega de valor com base em dados e interesses do consumidor. Ou seja, precisamos primeiro de mais pesquisas na área, seja para tirar as subjetividades aqui introduzidas como também para dar mais maturidade para os *frameworks* de *Service Mesh*.

Por fim, tratando do assunto de experimentos em paralelo, chegamos a conclusão que, apesar da literatura [6, 13] mencionar que existe dificuldade para utilizar essa prática, vemos que conseguimos direcionar corretamente os usuários para o experimento que ele participa em cada serviço, ao construirmos uma rede de serviços bem controlada e programada conforme proposta. Dessa forma, os esforços para execução de múltiplos experimentos transfere-se mais para a área estatística sobre quais usuários devem ser os escolhidos como participantes do experimento e sobre conseguir identificar se existe interferência entre os experimentos e como evitar.

Referências

[1] J. Bird. Feature toggles are one of the worst kinds of technical debt, Aug 2014. URL: `http://swreflections.blogspot.com/2014/08/feature-toggles-are-one-of-worst-kinds.html`.

[2] F. Budinsky. Canary deployments using istio, Jun 2017. URL: `https://istio.io/blog/2017/0.1-canary/`.

[3] A. Deb, S. Bhattacharya, J. Gu, T. Zhou, E. Feng, and M. Liu. Under the hood of uber's experimentation platform, Aug 2018. URL: `https://eng.uber.com/xp/`.

[4] S. V. Duggirala, M. Jog, and J. Nativo. Architecting istio 1.1 for performance, Mar 2019. URL: `https://istio.io/blog/2019/istio1.1_perf/`.

[5] Envoy Project Authors. Envoy. version 1.11.0-dev-18b8f1. URL: `https://www.envoyproxy.io/docs`.

[6] F. Fagerholm, A. S. Guinea, H. Mäenpää, and J. Münch. The right model for continuous experimentation. *Journal of Systems and Software*, 123:292 – 305, 2017. URL: `http://www.sciencedirect.com/science/article/pii/S0164121216300024`, `doi:https://doi.org/10.1016/j.jss.2016.03.034`.

[7] E. Fang. Building an intelligent experimentation platform with uber engineering, May 2017. URL: `https://eng.uber.com/experimentation-platform/`.

[8] P. Hodgson. Feature toggles (aka feature flags), Oct 2017. URL: `https://martinfowler.com/articles/feature-toggles.html`.

[9] Istio Authors. Istio. version 1.2.2. URL: `https://istio.io/docs/`.

[10] M. Klein. Service mesh data plane vs. control plane, Oct 2017. URL: https://blog.envoyproxy.io/service-mesh-data-plane-vs-control-plane-2774e720f7fc.

[11] R. Li. Service mesh: Promessa ou risco?, Out 2018. URL: https://www.infoq.com/br/articles/service-mesh-promise-peril/.

[12] L. Madeyski and M. Kawalerowicz. Software engineering needs agile experimentation: A new practice and supporting tool. In L. Madeyski, M. Śmiałek, B. Hnatkowska, and Z. Huzar, editors, *Software Engineering: Challenges and Solutions*, pages 149–162, Cham, 2017. Springer International Publishing.

[13] G. Schermann. Continuous experimentation for software developers. In *Proceedings of the 18th Doctoral Symposium of the 18th International Middleware Conference*, Middleware '17, pages 5–8, New York, NY, USA, 2017. ACM. URL: http://doi.acm.org/10.1145/3152688.3152691, doi:10.1145/3152688.3152691.

[14] G. Schermann, J. Cito, and P. Leitner. Continuous experimentation: Challenges, implementation techniques, and current research. *IEEE Software*, 35(2):26–31, March 2018. doi:10.1109/MS.2018.111094748.

[15] G. Schermann, J. Cito, P. Leitner, U. Zdun, and H. C. Gall. We're doing it live: A multi-method empirical study on continuous experimentation. *Information and Software Technology*, 99:41 – 57, 2018. URL: http://www.sciencedirect.com/science/article/pii/S0950584917302136, doi:https://doi.org/10.1016/j.infsof.2018.02.010.

[16] A. Thomas and R. Atachiants. Building grab's experimentation platform, Jul 2018. URL: https://engineering.grab.com/building-grab-s-experimentation-platform.

[17] J. Warren. Service mesh is super hot. why?, Dec 2018. URL: https://www.forbes.com/sites/justinwarren/2018/12/19/service-mesh-is-super-hot-why.

A Instalação do Istio com Google Kubernetes Engine

```
# Configurar plataforma do Google Kubernetes Engine
1. gcloud container clusters create $CLUSTER_NAME --cluster-version latest
    --num-nodes 4 --zone $ZONE_NAME --project $PROJECT_ID --enable-
    autorepair
2. gcloud container clusters get-credentials $CLUSTER_NAME --zone $ZONE_NAME
     --project $PROJECT_ID
3. kubectl create clusterrolebinding cluster-admin-binding --clusterrole=
    cluster-admin --user=$(gcloud config get-value core/account)

# Download
4. curl -L https://git.io/getLatestIstio | ISTIO_VERSION=1.2.2 sh -
5. cd istio-1.2.2
6. export PATH=$PWD/bin:$PATH
```

```
# Instalacao rapida do Istio
7. for i in install/kubernetes/helm/istio-init/files/crd*yaml; do kubectl
    apply -f $i; done
8. kubectl apply -f install/kubernetes/istio-demo.yaml
## Verificar a instalacao (aguardar inicializacao dos pods)
9. kubectl get svc -n istio-system
10. kubectl get pods -n istio-system

# Subindo a aplicacao
11. kubectl label namespace default istio-injection=enabled
12. kubectl apply -f samples/bookinfo/platform/kube/bookinfo.yaml
## Verificar pods e servico
13. kubectl get services
14. kubectl get pods
15. kubectl exec -it $(kubectl get pod -l app=ratings -o jsonpath='{.items
    [0].metadata.name}') -c ratings -- curl productpage:9080/productpage |
    grep -o "<title>.*</title>"
## Determinando o IP e porta de ingresso
16. kubectl apply -f samples/bookinfo/networking/bookinfo-gateway.yaml
17. kubectl get gateway
## Verificar se possui IP externo
18. kubectl get svc istio-ingressgateway -n istio-system
19. export INGRESS_HOST=$(kubectl -n istio-system get service istio-
    ingressgateway -o jsonpath='{.status.loadBalancer.ingress[0].ip}')
20. export INGRESS_PORT=$(kubectl -n istio-system get service istio-
    ingressgateway -o jsonpath='{.spec.ports[?(@.name=="http2")].port}')
21. export SECURE_INGRESS_PORT=$(kubectl -n istio-system get service istio-
    ingressgateway -o jsonpath='{.spec.ports[?(@.name=="https")].port}')
22. export GATEWAY_URL=$INGRESS_HOST:$INGRESS_PORT
## Confirme que a aplicacao e acessivel fora do cluster
23. curl -s http://${GATEWAY_URL}/productpage | grep -o "<title>.*</title>"
## Aplicando regras padroes de  DestinationRules
24. kubectl apply -f samples/bookinfo/networking/destination-rule-all.yaml
25. kubectl get destinationrules -o yaml
26. kubectl apply -f samples/bookinfo/networking/virtual-service-all-v1.yaml

# Alterar fluxo do trafego
27. kubectl apply -f samples/bookinfo/networking/virtual-service-reviews-v2-
    v3.yaml

# Metricas
28. kubectl -n istio-system port-forward $(kubectl -n istio-system get pod -
    l app=prometheus -o jsonpath='{.items[0].metadata.name}') 9090:9090 &
29. kubectl -n istio-system port-forward $(kubectl -n istio-system get pod -
    l app=grafana -o jsonpath='{.items[0].metadata.name}') 3000:3000 &
```