UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

VITOR DE OLIVEIRA FERNANDEZ ARAUJO

CHAOS ENGINEERING

História, evolução e tendências em uma metodologia de testes em sistemas distribuídos

RIO DE JANEIRO
2022

VITOR DE OLIVEIRA FERNANDEZ ARAUJO

CHAOS ENGINEERING

História, evolução e tendências em uma metodologia de testes em sistemas distribuídos

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientadora: Profa. Silvana Rossetto

RIO DE JANEIRO

2022

CIP - Catalogação na Publicação

A663c

Araujo, Vitor de Oliveira Fernandez
Chaos Engineering: história, evolução e tendências em uma metodologia de testes em sistemas distribuídos / Vitor de Oliveira Fernandez Araujo. - Rio de Janeiro, 2022.
79 f.

Orientadora: Silvana Rossetto.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Computação, Bacharel em Ciência da Computação, 2022.

1. chaos engineering. 2. sistemas distribuídos. 3. resiliência. 4. software. 5. testes. I. Rossetto, Silvana, orient. II. Título.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a), sob a responsabilidade de Miguel Romeu Amorim Neto - CRB-7/6283.

VITOR DE OLIVEIRA FERNANDEZ ARAUJO

CHAOS ENGINEERING
História, evolução e tendências em uma metodologia de testes em sistemas distribuídos

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 08 de Setembro de 2022

BANCA EXAMINADORA:

Silvana Rossetto
Silvana Rossetto
D.Sc. (UFRJ)

Daniel Sadoc Menasché
Daniel Sadoc Menasché
D.Sc. (UFRJ)

Felipe Fink Grael
M.Sc. (COPPE/UFRJ)

AGRADECIMENTOS

Gostaria de dar meus primeiros agradecimentos aos meus pais, Patricia e Jorge, por seu apoio, compreensão e suporte ao longo de todos os meus anos de vida e, sobretudo, nestes últimos anos de graduação. Tive a educação como prioridade durante meu crescimento por causa de vocês e, por isso, meu muito obrigado. Agradeço à minha parceira, Aline, pelo seu amor e, com ele, ter me aturado durante todo o tempo de produção desse trabalho. Da mesma maneira, agradeço aos meus amigos, que trouxe do colégio para a vida, por seu apoio nesta jornada. À minha orientadora, Silvana Rossetto, agradeço por sua compreensão e suporte, sem os quais este trabalho não seria possível. Obrigado à EJCM e ao GRIS, grupos de extensão dos quais participei na graduação e que me possibilitaram aprender muito. As experiências que tive em cada um deles certamente me tornaram um profissional melhor. Por fim, mas não menos importante, deixo minha gratidão à UFRJ, o Instituto de Computação e seus docentes, que contribuíram para a minha formação e sem dúvida representam o que a ciência brasileira tem de melhor.

"Nós precisamos saber. Nós iremos saber."

David Hilbert

RESUMO

Ao longo do início do século XXI, com a expansão da Internet, a sociedade tornou-se cada vez mais dependente de sistemas distribuídos. Esta dependência — por muitas vezes invisível — cria a responsabilidade para as empresas que desenvolvem estes sistemas de testá-los adequadamente e garantir que seu funcionamento seja estável e resiliente. Num contexto de sistemas de larga escala, com grande quantidade de usuários e componentes internos, essa tarefa pode ser um grande desafio, pela característica caótica e imprevisível que tais sistemas apresentam. Considerando esta problemática, este trabalho tem por objetivo prover uma visão geral sobre *Chaos Engineering*, uma nova metodologia de testes de sistemas distribuídos de larga escala, criada no mercado. Esta metodologia propõe uma estratégia rigorosa de testes em sistemas distribuídos por meio de técnicas de injeção de falhas, com o intuito de revelar fraquezas ocultas e intrínsecas do sistema. No trabalho, são analisados os passos propostos pela metodologia, as premissas consideradas e um recorte de sua história de origem. A metodologia também é abordada pelo prisma prático, estudando os experimentos e técnicas que algumas empresas selecionadas utilizaram em sua trajetória. Apresentamos, ainda, uma coleção de ferramentas e bibliotecas relevantes para a prática de *Chaos Engineering*, baseada em listas de recursos produzidas pela comunidade. Por fim, o trabalho fornece um breve resumo do contexto atual da metodologia, desafios para sua adoção e uma análise de seu futuro esperado.

Palavras-chave: chaos engineering; sistemas distribuídos; resiliência; confiabilidade; software; devops; computação em nuvem; testes; garantia de qualidade.

ABSTRACT

Over the course of the early 21st century, as the Internet expanded, society became further dependent on distributed systems. This dependence — often invisible — creates the responsibility that the companies who develop these systems test them adequately and ensure their stable and resilient operation. In the context of large-scale systems, having an extended amount of users and internal components, this task can prove to be a big challenge, given the chaotic and unpredictable characteristic these systems present. Considering this issue, this work aims to provide an overview of *Chaos Engineering*, a new testing methodology on large-scale distributed systems, created in the industry. This methodology proposes a rigorous testing strategy for distributed systems using failure injection techniques, aimed at revealing a system's hidden and intrinsic weaknesses. In this work, we analyze the steps proposed by the method, its assumptions and a piece of its origin history. The methodology is also seen through a practical lens, studying the experiments and techniques some selected companies have used in their evolution. Furthermore, we gather a collection of tools and libraries that are relevant to the practice of *Chaos Engineering*, based on resource lists built by the community. Finally, the work provides a short summary of the current context of the methodology, challenges to its adoption and an analysis of its foreseeable future.

Keywords: chaos engineering; distributed systems; resilience; reliability; software; devops; cloud computing; testing; quality assurance.

LISTA DE ILUSTRAÇÕES

LISTA DE CÓDIGOS

LISTA DE ABREVIATURAS E SIGLAS

TI	Tecnologia da Informação
RPC	Remote Procedure Call
CPU	Central Processing Unit
AWS	Amazon Web Services
GCP	Google Cloud Platform
REST	Representational State Transfer
API	Application Programming Interface
RAM	Random Access Memory
MTTR	Mean Time to Recovery/Repair
TLS	Transport Layer Security
JSON	JavaScript Object Notation
YAML	YAML Ain't Markup Language
CLI	Command Line Interface
SLA	Service Level Agreement
SRE	Site Reliability Engineering

SUMÁRIO

1 INTRODUÇÃO

Nos primeiros anos do século XXI, vimos a Internet crescer em relevância, assim como a quantidade de computadores e outros dispositivos conectados usados pela população. Uma tecnologia que, antes, tinha sua maior aplicação em ambientes acadêmicos e empresariais, agora se difundia globalmente, pressionando a demanda por bens e serviços *on-line*. A fim de comportar este aumento de escala, os sistemas de *software* cresceram e tornaram-se mais distribuídos. Para dar suporte ao desenvolvimento desses sistemas, alguns estilos arquiteturais se destacaram, entre eles, a arquitetura orientada a serviços (SOA[1]) (LASKEY; LASKEY, 2009) e, mais recentemente, sua evolução para arquitetura de microsserviços (FOWLER, 2014). Tal transformação veio com o objetivo tanto de simplificar o desenvolvimento, separando o código em componentes discretos que trocam mensagens entre si, como de utilizar múltiplos servidores de maneira mais eficiente, distribuindo as requisições pela rede e possibilitando atender maior número de usuários.

Partindo do amadurecimento das tecnologias de virtualização de *hardware* que ocorreu no final dos anos 90, o contexto de crescimento da Internet também proporcionou o surgimento de soluções de nuvem pública, que têm como proposta facilitar o investimento em infraestrutura de tecnologia por parte de empresas, de forma mais escalável. Por meio deste modelo, os clientes podiam expandir e reduzir o uso de recursos sob demanda, incorrendo em menos custos. Isto causou uma mudança de paradigma na indústria, que começou a migrar do uso de infraestrutura *on-premises*[2] para a nuvem pública. Com esta transformação, acelerou-se ainda mais o uso de arquiteturas de *software* distribuídas, pois agora era possível ajustar o custo do *hardware* de forma mais precisa. No entanto, neste cenário de maior tráfego entre aplicações, problemas inerentes a sistemas distribuídos tornam-se mais críticos.

Steen e Tanenbaum (2018, p. 2) definem, brevemente, um sistema distribuído como: "[...] uma coleção de elementos computacionais autônomos, que se mostra aos seus usuários como um único sistema coerente". Com base nesta definição, podemos perceber desafios associados à confiabilidade, capacidade e latência do meio de comunicação entre componentes, bem como às configurações e condições de manutenção de componentes remotos, que muitas vezes podem não ser administrados pela mesma entidade, mas possuem interdependências. Esta problemática foi e é estudada por engenheiros de *software* e pesquisadores, que buscam maneiras eficientes de obter a maior confiança possível na

[1] Do original: *Service Oriented Architecture*.

[2] Refere-se a um modelo de hospedagem de sistemas, onde a infraestrutura é de propriedade e mantida pela própria empresa que utiliza os sistemas.

resiliência e correto funcionamento dos sistemas que suportam o modo de vida de nossa sociedade moderna.

1.1 MOTIVAÇÃO

Neste contexto, surgem na indústria, de forma difusa, práticas de teste orientadas a detectar e mitigar estas fragilidades características de sistemas distribuídos, que culminam para o que hoje se tornou uma metodologia chamada de *Chaos Engineering*[3] (CHAOS COMMUNITY, 2019). Esta disciplina consiste numa abordagem controlada para a injeção de falhas e exposição de fraquezas em sistemas distribuídos. O tema surgiu, sob este nome, a partir dos esforços de experimentação da empresa de *streaming* Netflix, em 2008, quando iniciaram uma grande migração de seus sistemas para a nuvem pública da AWS. Para garantir a confiabilidade de seus sistemas perante a falha de instâncias de máquinas virtuais, criaram a ferramenta *Chaos Monkey* (NETFLIX, 2019b). Ao longo dos anos, mais empresas foram adotando práticas similares, e o corpo de práticas entendido como *Chaos Engineering* se solidificou e ganhou popularidade no mercado de tecnologia, como evidenciado pelo surgimento de diversas conferências e grupos de discussão do tema, como a Chaos Conf (GREMLIN, 2020b) e a Chaos Carnival (CHAOSNATIVE, 2022), além de outras, presentes em listas como a que se encontra em Ratis (2022).

1.2 OBJETIVO

O objetivo deste trabalho é prover uma base de conhecimento sobre o tema de *Chaos Engineering*, apresentando em detalhes a história de sua criação, e realizando uma análise de suas práticas, impactos organizacionais, e de ferramentas associadas à sua implantação. Com isto, busca-se iniciar o preenchimento de uma lacuna observada na produção acadêmica lusófona com relação a este tema, que vem ganhando relevância crescente no mercado mundial de tecnologia.

1.3 METODOLOGIA

A coleta de informações para a produção deste texto foi orientada fortemente pelo conteúdo que se originou na Netflix, na forma de artigos e livros publicados ao longo dos últimos anos por membros da empresa. Em especial, destacam-se o livro publicado em 2017 (ROSENTHAL et al., 2017) e o primeiro artigo onde são apresentados os princípios da disciplina (ROSENTHAL et al., 2016). Para compor a base teórica e histórica do trabalho, foram utilizados artigos de fontes reconhecidas, como o IEEE Xplore e a ACM

[3] Ao longo de todo o trabalho, optaremos por não traduzir o termo "*Chaos Engineering*", devido à popularidade da versão anglicizada e possível confusão com o uso do termo "Engenharia do Caos", aplicado a outros contextos.

Digital Library, tanto quanto possível. No entanto, pela natureza incipiente do tema e o perfil ágil da indústria, parte das informações também foi encontrada em artigos de *blogs* de tecnologia de empresas da área: notadamente, da própria Netflix, e da Gremlin, empresa que surgiu para prover consultoria e ferramentas especializadas de *Chaos Engineering*.

1.4 ESTRUTURA DO TRABALHO

O presente trabalho se organiza da seguinte maneira: no Capítulo 2, apresentamos os fundamentos sobre alguns tópicos importantes para entender o contexto no qual se insere a disciplina de *Chaos Engineering*. No Capítulo 3, apresentamos uma introdução sobre as origens do tema e abordamos seus conceitos e princípios. No Capítulo 4, nos debruçamos em mais detalhes sobre práticas e relatos da aplicação do método no mercado, discutindo mudanças culturais nas empresas e a criação de ferramentas dentro de grandes organizações, como Netflix, Amazon, Google e LinkedIn. No Capítulo 5, destacamos exemplos de ferramentas abertas e proprietárias, criadas ao longo dos últimos anos para auxiliar a implementação e automação de *Chaos Engineering* em diferentes contextos organizacionais. Por fim, no Capítulo 6, concluímos o trabalho, sintetizando as informações encontradas e fazendo uma breve discussão do cenário atual da prática de *Chaos Engineering*, bem como expectativas futuras e desafios na implementação e na pesquisa sobre o tema.

2 FUNDAMENTOS

Neste capítulo, tratamos de temas que circundam a temática de *Chaos Engineering*, sobre a qual nos aprofundaremos nos próximos capítulos deste trabalho. A disciplina de *Chaos Engineering* surge num contexto de rápida evolução das tecnologias computacionais, em termos de escala de consumo. A necessidade das empresas de criar produtos de alcance global, com altíssima disponibilidade e escalabilidade, fez com que surgissem soluções flexíveis de computação para atender esta demanda, na forma de modelos de nuvem pública, que abordamos na seção 2.2. Da mesma maneira, tal crescimento de escala traz algumas dificuldades, previstas e analisadas pela disciplina de Sistemas Distribuídos, que se torna cada vez mais relevante dado o cenário tecnológico atual. Para abordar este tema, reservamos a seção 2.1. Por fim, este novo contexto de desenvolvimento de *software*, com muitas partes independentes e integradas para prover um serviço, traz consigo novos desafios de gestão de entrega de *software* e garantia de qualidade do código. Estes são assuntos centrais dos quais trata o movimento DevOps, através de um conjunto de práticas, ferramentas e, sobretudo, mudanças culturais sobre as quais discutimos na seção 2.3.

2.1 SISTEMAS DISTRIBUÍDOS

Neste trabalho, faremos frequentes referências ao tópico de sistemas distribuídos. Precisamos, portanto, debruçar-nos sobre algumas definições e conceitos envolvidos neste tema. A área de sistemas distribuídos é fonte de ampla discussão e pesquisa, de forma que, nesta seção, nos restringiremos a abordar definições gerais dentro do tema e discussões relacionadas aos desafios envolvidos na implementação de sistemas distribuídos. Seguindo esta temática, apresentaremos também, uma visão do que a literatura traz de conhecimento a respeito do tratamento de falhas nestes sistemas, um assunto que terá grande correlação com o tema central deste trabalho.

2.1.1 Definição

De maneira direta, utilizamos a seguinte definição de sistema distribuído, dada por Steen e Tanenbaum (2018): "Um sistema distribuído é uma coleção de elementos autônomos de computação, que se mostra aos seus usuários como um único sistema coeso". O conceito de computação distribuída se difunde a partir da consolidação das tecnologias de comunicação em rede, por volta da década de 1980. Os programas, que antes executavam em apenas um computador, poderiam ser particionados de maneira a utilizar o poder computacional de diversas máquinas e permitir a interação entre usuários em localizações geográficas distintas.

2.1.2 Vantagens e desafios

O desenvolvimento de *software* num modelo de sistema distribuído pode apresentar vantagens tanto financeiras quanto em termos de flexibilidade para a operação do mesmo em produção. Destas, ressaltamos os seguintes aspectos, sintetizados a partir dos objetivos de projeto de sistemas distribuídos, conforme citados por Steen e Tanenbaum (2018):

- **Otimização de custos**: Por permitir o uso de *hardware* heterogêneo, é possível ter um sistema com grande capacidade de processamento paralelo sem necessitar de investimento em um único elemento poderoso de computação, como um mainframe, ou máquinas com muitos núcleos.
- **Tolerância a falhas**: A depender da arquitetura do sistema projetado, permite que os serviços possam continuar a ser providos mesmo com a perda de um ou mais componentes.
- **Escalabilidade**: Recursos de computação podem ser adicionados e removidos de um sistema com facilidade, possibilitando a adaptação a uma carga inesperada e, em cenários de computação em nuvem (como discutiremos na seção 2.2), permitindo também otimizar custos.
- **Otimização de latência**: Por permitir uma distribuição geográfica de recursos, é possível servir conteúdo para usuários em locais diferentes do mundo a partir de servidores mais próximos, minimizando a latência decorrente da distância física.

Estas vantagens, em maioria, se relacionam com o fato de que, em um sistema distribuído, podemos ter múltiplas réplicas de partes diferentes do sistema, espalhadas de forma heterogênea e pouco acoplada. Mostraremos que estas propriedades também apresentam desafios, que precisam ser levados em consideração para o desenvolvimento e operação bem sucedidos de um sistema distribuído.

A presença da comunicação em rede como um fator relevante dentro do sistema introduz uma série de desafios, que, por vezes, podem ser ignorados por desenvolvedores de *software*. Com estes desafios em mente, Peter Deutsch e outros engenheiros da Sun Microsystems idealizaram, em 1997, uma compilação de oito falácias, que podem enganar desenvolvedores de sistemas distribuídos e gerar problemas na manutenção desses sistemas. Listamos essas falácias abaixo, explicando brevemente seu significado, conforme o trabalho feito por Rotem-Gal-Oz (2008):

- "A rede é confiável": A comunicação através da rede pode ser suscetível a falhas por diversos motivos, e é responsabilidade do *software* tratá-las para garantir a entrega de mensagens.

- "A latência é zero": Principalmente em ambientes que envolvem o uso de chamadas de função remotas (RPC), torna-se importante levar em consideração a latência introduzida pela comunicação em rede, visto que a mesma é ordens de magnitude maior que a de uma chamada de função local, ou uma comunicação interprocessos.

- "A largura de banda é infinita": Apesar de, para o contexto dos tempos atuais, esta limitação não ser muito relevante (não é incomum termos redes locais com capacidade de dezenas de Gigabits por segundo, em contextos empresariais), ainda é interessante limitar, tanto quanto possível, o tamanho em *bytes* das mensagens trocadas pelos componentes dos sistemas.

- "A rede é segura": Ao construir *software* que lida com dados sensíveis, tanto para os clientes quanto para os negócios, é necessário levar em consideração o uso de protocolos de criptografia na comunicação entre os componentes mais expostos do sistema, bem como medidas de segurança de rede.

- "A topologia não muda": A topologia da maioria dos sistemas distribuídos da atualidade é extremamente dinâmica, com usuários se conectando e desconectando frequentemente. Além disso, também é possível que haja mudanças decorrentes de mecanismos de escalabilidade do próprio sistema, que podem inserir ou remover nós do mesmo em resposta à demanda.

- "Existe apenas um administrador": Na grande maioria dos sistemas distribuídos da atualidade, existe diversas dependências externas, inclusive a própria Internet, que faz com que os mesmos estejam suscetíveis a múltiplos domínios administrativos e, portanto, possíveis limitações de tráfego e compatibilidade de interfaces.

- "O custo de transporte é zero": Nesta afirmação, o custo pode ter tanto um significado econômico, relacionado aos custos que podem ser aplicados ao trafegar dados numa rede gerenciada, como também ao custo computacional envolvido na serialização/desserialização de dados.

- "A rede é homogênea": A rede que conecta componentes de um sistema distribuído pode ser muito heterogênea, com equipamentos de diferentes fabricantes, sujeitos a condições variadas. Além da heterogeneidade física, sistemas distribuídos também estão sujeitos à heterogeneidade lógica de outros subsistemas nos quais dependam, e, por este motivo, precisam implementar interfaces e tecnologias de fácil portabilidade tanto quanto possível.

2.1.3 Modos de Falha

Como pudemos ver, apesar de suas vantagens, também existem diversas maneiras em que um sistema distribuído apresenta dificuldades no seu desenvolvimento e manutenção.

Estas dificuldades se traduzem em aberturas para falhas que, muitas vezes, são difíceis de diagnosticar e prever, pela alta quantidade de interações e dependências entre componentes. Por este motivo, é interessante categorizarmos os modos de falha de sistemas distribuídos, com o objetivo de entender se e como é possível mitigá-las.

Cristian (1991) destaca os seguintes modos de falha:

- **Falha de parada**[1]: Consiste na interrupção repentina de um servidor, caracterizada pela sua incapacidade de responder requisições, até que o mesmo seja reiniciado. Considera-se, também, o que acontece com o estado original do servidor após sua reinicialização, visto que ele pode reter completamente o estado anterior à falha, reter parte dele, ou voltar para um estado padrão.
- **Falha de omissão**: Uma ausência transiente de comunicação com clientes que se conectam ao servidor. Pode se manifestar de duas formas:
  - Omissão no recebimento: Não houve comunicação por que o servidor nunca recebeu uma requisição;
  - Omissão no envio: A requisição foi recebida e processada, mas uma resposta não consegue alcançar o cliente por quaisquer motivos.
- **Falha de *timing***: Refere-se à situação em que a resposta a uma requisição é correta, mas foi enviada mais cedo ou mais tarde que o esperado. No caso de um envio adiantado (mais raro), problemas podem ser causados se não houve tempo para alocação de memória ou alguma outra ação necessária para processar a mensagem. No caso tardio, isso pode indicar um problema de desempenho por parte do servidor.
- **Falha de resposta**: Neste caso, uma resposta incorreta foi gerada pelo servidor. Manifesta-se de duas formas:
  - Falha no valor: O conteúdo da resposta está incorreto (e.g., informação corrompida, faltante, de tamanho inesperado);
  - Falha na transição de estados: O servidor não sabe como tratar uma determinada requisição e, como consequência da mesma, transita para algum estado que não é esperado.
- **Falha arbitrária (bizantina)**: Abrange situações em que o servidor pode produzir respostas inesperadas em momentos inesperados. Usualmente, tem relação com dependências que o servidor possa ter de outros componentes e sistemas. É o modelo de falha mais difícil de lidar, por sua imprevisibilidade.

[1] Traduzido do original "*crash failure*"

2.1.3.1 Técnicas de tolerância a falhas

A propriedade de tolerância a falhas se refere à capacidade de um sistema de ocultar a ocorrência de falhas para eventuais usuários externos. Sistemas tolerantes a falha são capazes de, com certa velocidade, detectar falhas em um ou mais componentes internos e alterar seu estado para mascará-las, de acordo com suas especificações de projeto.

Temos, como principal estratégia para a implementação de mecanismos de tolerância a falhas, a noção de **redundância** (STEEN; TANENBAUM, 2018). Esta redundância pode se apresentar de diversas formas, a depender do(s) modo(s) de falha que se deseja ocultar. A mais comum no planejamento de sistemas modernos, e a que traz consigo os maiores impactos financeiros, é a **redundância física**. Notadamente, o uso de múltiplas máquinas, equipamentos ou processos lógicos para replicar o processamento e armazenamento de dados é uma estratégia amplamente utilizada em bancos de dados distribuídos e em aplicações que lidam com alto volume de dados. A implementação deste tipo de redundância traz consigo, no entanto, dificuldades relacionadas ao consenso entre as múltiplas réplicas, com ampla pesquisa científica tendo sido feita em algoritmos de escolha de líder e propagação de informação. Não nos aprofundaremos nos temas citados neste trabalho, referindo Cachin, Guerraoui e Rodrigues (2011), Lamport (2001) e Steen e Tanenbaum (2018) como fontes sobre o assunto.

O uso de redundância como método de tolerância a falhas se estende, também, a medidas como a **redundância informacional**, que consiste no uso de bits redundantes numa mensagem para que corrupções no envio da mesma não causem sua perda. Este método é utilizado em praticamente toda a comunicação de rede mundial atualmente. Por fim, temos a **redundância temporal**, que consiste na repetição de ações, com o intuito de recuperar o sistema de falhas transientes ou intermitentes. Um exemplo célebre de seu uso é na implementação de transações, muito comum em motores de bancos de dados modernos.

Ademais, com a evolução dos sistemas distribuídos (sobretudo, dos *Web Services* e, posteriormente, microsserviços), surgiram implementações de um conjunto de padrões de desenvolvimento de *software* e algoritmos orientados à resiliência, que se consolidaram e foram documentados nos últimos anos (NYGARD, 2007):

- *Retry*: Parte da premissa que falhas na comunicação entre serviços podem ser transientes, e implementa um tempo de espera variável ("*backoff*") antes de tentar fazer uma chamada remota novamente.
- *Fallback*: Define um resultado padrão ou uma segunda alternativa de dependência a chamar, em caso de falha. Tem o objetivo de mitigar impactos negativos da falha de um componente do sistema.

- *Timeout*: Especifica um tempo limite para a execução de uma chamada remota. Desta forma, evita-se que um sistema fique não responsivo por muito tempo no evento de uma falha no serviço remoto sendo chamado.

- *Caching*: Armazena o resultado anterior de uma requisição em alguma estrutura de dados chave-valor, utilizando os parâmetros da requisição como chave. Ao executar novamente a mesma requisição dentro de uma janela de tempo especificada, retorna o valor salvo imediatamente, sem fazer uma chamada remota.

- *Circuit Breaker*: Visa a impedir que a falha de um componente se propague para outros que dependam do mesmo, por meio do acúmulo de chamadas enfileiradas aguardando resposta. O método consiste em monitorar a quantidade recente de falhas em uma chamada remota e, ao superar um dado limite, impedir que mais chamadas ocorram, retornando um erro instantaneamente. O algoritmo age como uma máquina de estados que pode estar Fechado, Aberto ou Meio-Aberto. Em situação normal, o circuito está Fechado e as requisições fluem para o destino. Ao detectar falhas acima do limite, o algoritmo muda para o estado Aberto, iniciando a contagem de um relógio e retornando erros imediatamente durante o período, sempre que uma chamada remota é requisitada. Quando o relógio atinge um tempo limite predefinido, a máquina de estados varia para Meio-Aberto, e passa a encaminhar chamadas para o componente remoto normalmente. Se uma única falha for detectada, volta para o estado Aberto; do contrário, aguarda até que a quantidade de sucessos consecutivos atinja um limite de confiança para que possa voltar ao estado Fechado.

- *Bulkhead*: Consiste numa política de isolamento de dependências, onde os recursos de um serviço são particionados de tal forma que o aumento expressivo na carga sobre uma dependência não prive os recursos necessários para continuar a interação normal com as outras dependências do serviço. Essencialmente, a solução implementa *pools* de recursos que limitam a concorrência de forma independente para as diferentes chamadas remotas que o serviço faz, evitando que as mesmas roubem recursos umas das outras.

A aplicação de um conjunto destas técnicas é, atualmente, uma condição importante para a construção de sistemas distribuídos de larga escala. Isto se dá, entre outros motivos, pelo crescente uso da computação em nuvem como infraestrutura de base para estes sistemas, o que introduz novas preocupações e paradigmas de desenvolvimento de software, como apresentaremos na próxima seção.

2.2 COMPUTAÇÃO EM NUVEM

O modelo de computação em nuvem surgiu diante da inflexibilidade e alto custo proporcionados pelo uso de *data centers* próprios. Além disso, com a evolução da Internet possibilitando conexões mais rápidas e amplamente disponíveis, provedores de nuvem pública surgiram para tornar a abordagem um produto acessível e diverso, com recursos que visam a agilidade e simplicidade na hospedagem de *software* para a *web*.

Provedores de nuvem pública são empresas, cujo negócio se baseia na concessão de recursos computacionais para seus clientes. Isso se dá conforme um contrato que estabelece níveis de confiabilidade (SLA[2]), num modelo *pay-as-you-go*. Para prover este serviço de forma segura, a infraestrutura de nuvem utiliza fortemente o conceito de **virtualização**, que provê separação e isolamento dos recursos. Não nos aprofundaremos neste conceito, por fugir do escopo desejado, apontando Campbell e Jeronimo (2006) e Sareen (2013) como fontes de mais informações.

2.2.1 Modelos de computação em nuvem

Grance e Mell (2011), em relatório técnico do NIST[3], definem a computação em nuvem:

> A computação em nuvem é um modelo que permite o acesso por rede de forma ubíqua, conveniente e sob-demanda, a um *pool* de recursos computacionais configuráveis (como redes, servidores, armazenamento, aplicações e serviços), que podem ser rapidamente provisionados e liberados com esforço gerencial ou interação mínimos, por parte do provedor de serviços.

Nos moldes desta definição, com foco principal na característica de configurabilidade e baixo esforço gerencial, as implementações de nuvem atuais se apresentam organizadas nas seguintes camadas (STEEN; TANENBAUM, 2018):

- **Hardware**: Sendo a camada mais baixa, comporta todos os elementos elétricos e eletrônicos necessários para a computação, incluindo o *hardware* clássico, como processadores, memórias, roteadores, *switches*, mas também equipamentos de resfriamento e suprimento de energia. É uma camada com a qual os usuários finais não interagem, pois requer acesso físico ao *data center*.
- **Infraestrutura**: Esta camada é a responsável pela virtualização que faz com que os usuários finais vejam recursos de computação, armazenamento e rede virtuais, que podem utilizar conforme sua necessidade.

[2] Acrônimo para *Service Level Agreement*. Se trata de um nível de disponibilidade mínimo expresso em percentuais, no qual o provedor se compromete em manter a operação de seus serviços.

[3] Instituto Nacional de Padrões e Tecnologia dos Estados Unidos da América.

- **Plataforma**: A camada de plataforma provê uma interface que abstrai detalhes de implementação. Esta é feita de forma a permitir que um usuário execute programas e dê comandos para a infraestrutura subjacente, além de prover funcionalidades específicas, como armazenamento, troca de mensagens, entre outros.

- **Aplicação**: Aqui se localizam as aplicações desenvolvidas pelos provedores, que são disponibilizadas já prontas para os usuários, num modelo cliente-servidor, onde os mesmos podem realizar algum tipo de atividade e manter os resultados dessas atividades armazenados na nuvem, por exemplo.

Esta organização propicia um modelo de responsabilidade compartilhada, por meio do uso dos serviços de nuvem conforme as seguintes abordagens básicas (GRANCE; MELL, 2011):

- **Infrastructure as a Service (IaaS)**: O usuário tem a capacidade de provisionar poder de processamento, armazenamento, memória, rede, e executar qualquer tipo de *software* nesta infraestrutura, tendo controle de administrador sobre estes recursos virtuais de forma remota. Neste modelo, o usuário é responsável pela segurança e operação do software que executa nas máquinas virtuais, assim como qualquer configuração e dados ali presentes. O provedor apenas tem gerência sobre o *hardware* e a infraestrutura.

- **Platform as a Service (PaaS)**: Neste segundo modelo, abstrai-se as camadas mais baixas de *software* (e.g. sistema operacional, bibliotecas de código), permitindo ao usuário executar código próprio, sujeito às limitações e praticidades proporcionadas pela plataforma. Neste cenário, o usuário tem controle apenas sobre os programas que serão executados e possíveis parâmetros de configuração.

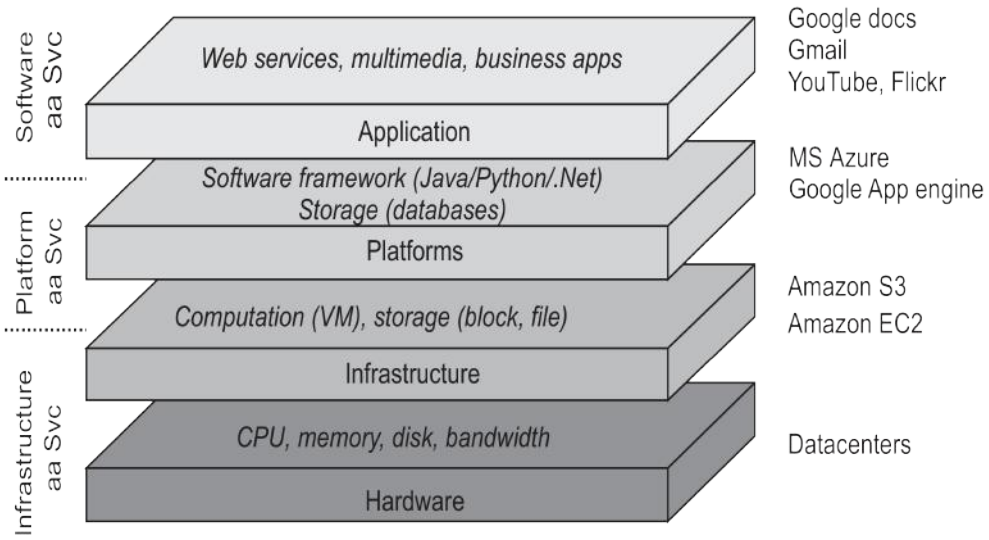
- **Software as a Service (SaaS)**: Por último, abstrai-se ao máximo a responsabilidade do usuário, provendo um *software* pronto, para um propósito específico. Neste caso, o usuário tem a liberdade de utilizar o *software* como desejar, tendo controle sobre os dados e configurações que serão feitas nele, mas sujeito às limitações impostas pelo provedor, que desenvolve e mantém o *software*.

A Figura 1 ilustra a organização hierárquica das camadas e modelos de serviço, incluindo, no lado direito, uma relação entre alguns exemplos de produtos e serviços e o modelo no qual se encaixam.

2.2.2 Impactos no desenvolvimento de sistemas

Tal configuração do modelo de computação em nuvem faz com que surjam características inerentes ao mesmo, que modificam a forma na qual *software* desenvolvido para

Figura 1 – Modelo hierárquico de camadas da computação em nuvem



Fonte: Steen e Tanenbaum (2018)

ser implantado num ambiente de nuvem é pensado. Dentre estas características, figuram, segundo Grance e Mell (2011) e Boutaba, Cheng e Zhang (2010):

- **Compartilhamento de recursos**: Em decorrência do modelo de virtualização, os recursos de computação utilizados são compartilhados, ainda que exista uma camada de abstração separando as diferentes processos que podem estar sendo executados numa mesma máquina física. Isso faz com que seja necessária a confiança nos *softwares* de virtualização para balancear de forma eficiente as cargas de trabalho, e, apesar da maturidade do estudo nesta área, vulnerabilidades nesses mecanismos podem representar um risco.

- **Falta de controle sobre o *hardware***: Pela definição das camadas do modelo de computação em nuvem, determina-se que o *hardware* não está sob poder do usuário, ou seja, do desenvolvedor que produz *software* para a nuvem. Desta forma, o desenvolvimento deve ser feito levando em consideração o paradigma de não tratar o *hardware* como relevante, devendo manter o código agnóstico em relação ao mesmo.

- **Escalabilidade sob-demanda**: Um dos principais diferenciais da computação em nuvem é seu potencial de escalabilidade, que traz benefícios tanto em termos de desempenho, quanto econômicos. Por outro lado, uma importante consequência é que, para que uma aplicação ou serviço possa ser escalado horizontalmente, i.e., ser replicado em múltiplas instâncias para balanceamento de carga, é necessário que o mesmo não guarde nenhum tipo de estado ou tenha configurações locais que dependam do ambiente virtual em que foi criado. Este conceito é conhecido como *stateless* (REDHAT, 2020).

- **Telemetria**: O ambiente virtualizado de um provedor de nuvem conta com mecanismos de monitoramento de métricas relevantes para cada tipo de serviço. Estas métricas, além de proverem controle para a gestão do próprio provedor, também representam informações valiosas que podem ser utilizadas no desenvolvimento e sustentação de *software* para a nuvem.

- **Disponibilidade geográfica**: Para prover maior resiliência e ter um alcance abrangente para seus negócios, a grande maioria dos provedores de nuvem pública contam com *data centers* em diversas partes do globo. Para algumas aplicações, a latência pode ser um fator sensível. Por este motivo, uma das componentes da latência, a distância física, pode ser controlada através do uso de regiões mais próximas da origem do tráfego. Aproveitamos para introduzir conceitos importantes da configuração geográfica dos provedores de nuvem (AWS, 2022):

  - **Regiões**: No contexto de computação em nuvem, as regiões são locais físicos ao redor do mundo onde o provedor posiciona *data centers*. Dentro das regiões, os *data centers* são agrupados logicamente e têm seus recursos como energia elétrica, refrigeração e segurança física mantidos de forma independente.
  - **Zonas de Disponibilidade (AZ)**: Os grupos lógicos de *data centers* dentro das regiões representam as Zonas de Disponibilidade (referidas como AZ, de *Availability Zone*). As AZs são configuradas dentro das regiões, de forma que todas tenham uma distância geográfica mínima entre si, para prover isolamento no caso de desastres naturais e outros problemas de infraestrutura. Apesar da distância, as AZs têm uma comunicação de alta capacidade entre si, com canais redundantes e criptografados.

  As características das AZs podem ser usadas por desenvolvedores para implementar modelos de infraestrutura multi-AZ (XU et al., 2013), que consistem na replicação de sistemas e bancos de dados entre múltiplas zonas de disponibilidade para atingir maior resiliência no caso de eventuais falhas em um *data center* do provedor.

Estes fatores fazem com que o desenvolvimento de *software* para a nuvem exija a adaptação das equipes de desenvolvimento e de testes, pois introduz pontos de falha e configurações extras que precisam ser validados.

2.2.3 Novos paradigmas

Nos últimos anos, vimos também, o surgimento de novos paradigmas de execução e encapsulamento de componentes de *software* no contexto de sistemas distribuídos. A arquitetura de microsserviços se tornou bastante relevante neste âmbito (FOWLER, 2014), com a influência de grandes empresas de tecnologia empregando seu uso para ter mais agilidade no desenvolvimento de seus produtos, além de uma maior capacidade de lidar com

grande carga de requisições de forma mais flexível. Este modelo arquitetural permite que se possa escalar partes específicas do sistema conforme a demanda momentânea, com alta granularidade. Naturalmente, para que tal feito seja possível na camada lógica, é necessário que a camada física também acompanhe, por meio do provisionamento automático de recursos.

Atualmente, uma das principais ferramentas utilizadas para facilitar esta atividade é o uso do conceito de contêineres Linux, tendo como principal produto associado, o Docker (DOCKER, 2022). O uso de contêineres permite um encapsulamento ideal de dependências de cada aplicação, tipicamente usando-se um contêiner por instância de serviço desejada. Por ter uma estrutura mais simples, a inicialização e destruição de um contêiner é mais ágil do que a de uma máquina virtual, por exemplo, possibilitando o surgimento de ferramentas e produtos focados na orquestração destes contêineres, fazendo o provisionamento dos mesmos de forma adaptativa. Destas, a principal atualmente é o Kubernetes (THE LINUX FOUNDATION, 2022a), que inclui também ferramentas de configuração para definir mecanismos de balanceamento de carga, estratégias de *release* e limitações de recursos dos contêineres.

Neste contexto, também surgiu, alguns anos depois, produtos dos provedores de nuvem que introduziriam o modelo de computação "*Serverless*". Este consiste em fazer uso de uma infraestrutura de contêineres, entre outros detalhes que não ficam claros por motivos de segredo industrial, para prover recursos de computação sob demanda para cada requisição ou evento recebido por um serviço. Desta forma, a alocação de recursos ocorre de forma momentânea, utilizando a plataforma e incorrendo em custos apenas pela duração do processamento necessário. Esta flexibilidade financeira traz consigo um modelo de precificação que, por cobrar por tempo de processamento, desencoraja operações de longa duração em plataformas *Serverless*, fazendo com que as aplicações construídas utilizando as mesmas necessitem ser parte de uma arquitetura orientada a eventos ou de processamento em lotes, com comunicação assíncrona e computação distribuída.

2.3 DEVOPS

O objetivo desta seção é elucidar o conceito de DevOps e prover uma conexão do mesmo com a ideia de *Chaos Engineering*, tema central deste trabalho. Para isto, vamos nos ater a definir conceitos fundamentais dentro do movimento DevOps e apresentar algumas práticas relacionadas a automação de testes de *software*, que provêm um contexto ideal para a implantação do método de *Chaos Engineering*.

2.3.1 Cultura e princípios

DevOps é um conceito amplo, que, em linhas gerais, se trata de um movimento que busca trazer uma mudança cultural nas equipes de TI, através da maior integração entre desenvolvedores (Dev) — que projetam a arquitetura do sistema e implementam as linhas de código — e profissionais de operações (Ops), responsáveis pela implantação de sistemas e a manutenção da infraestrutura que os sustenta. Para isso, o movimento DevOps defende práticas que visam a otimizar o fluxo de entrega de *software*, utilizando-se de ideias presentes em outras metodologias de gestão, como o Lean e o Agile, cujas premissas também defendem a colaboração, experimentação contínua e criação de valor por meio de entregas constantes (KIM et al., 2016).

Na prática, estes objetivos são atingidos por meio de ferramentas, processos e práticas de desenvolvimento que permitem o trabalho das equipes de forma independente, com entregas pequenas e constantes, que são testáveis e monitoráveis em produção. Para englobar estes objetivos, Jez Humble, um dos autores do *The DevOps Handbook* (KIM et al., 2016), criou o acrônimo CALMS, que consiste nos princípios (BUCHANAN, 2022):

- **Cultura**: Este princípio visa a garantir o alinhamento de todos os integrantes de uma equipe com o objetivo final de entrega de um produto. Isto é atingido através da colaboração no planejamento e desenvolvimento, bem como pela resolução de incidentes focada no aprimoramento de sistemas e processos, sem atribuição de culpa a membros da equipe.

- **Automação**: Para acelerar o fluxo de entrega, a automação de processos repetitivos é fundamental. Candidatos frequentes a tais automações são as rotinas de compilação de pacotes (conhecida como *build*), testes lógicos e implantação do *software*. Atualmente, com o conceito de infraestrutura elástica introduzido pelo modelo de computação em nuvem pública, é possível também falarmos de automação na camada de *hardware*, por meio de ferramentas como Infraestrutura como Código (IaC) e do provisionamento escalável por meio de APIs dos provedores de nuvem.

- **Lean**: A metodologia Lean tem como objetivo otimizar o fluxo de produção e estabelecer um ciclo de melhoria constante. No contexto de DevOps, isto se mostra na noção da entrega de pequenos incrementos ao produto de forma contínua e equipes enxutas, nas quais é possível comunicar-se e colaborar com fluidez.

- **Medição**: Para suportar o objetivo de melhoria constante, a metodologia DevOps também defende a medição e coleta de dados em diversos pontos. Isso se aplica tanto aos processos organizacionais, como, principalmente, aos sistemas e os resultados das interações de usuários com os mesmos. Neste senso, ter métricas estabelecidas

e ferramentas eficientes para capturá-las e prover *feedback* sobre as mesmas em produção, é essencial.

- **Compartilhamento** (*Sharing*): Ressaltando a necessidade de colaboração, o modelo CALMS vai além, defendendo também o compartilhamento livre de informações entre equipes, de forma a mostrar o valor das práticas e aumentar o escopo do impacto organizacional. Não só informações, mas também, de forma mais direta, é importante o compartilhamento de responsabilidades, a fim de quebrar os isolamentos entre equipes e fomentar a comunicação aberta, à que visa a cultura DevOps.

Nas implementações de DevOps encontradas na indústria, vemos práticas comuns, que visam a incorporar os princípios apresentados no ciclo de vida do desenvolvimento de *software* e, assim, garantir a entrega contínua de valor para o usuário final. Daremos uma visão geral destas práticas nas próximas seções.

2.3.2 Integração Contínua

O termo "integração contínua"[4] foi originado na descrição da metodologia de desenvolvimento ágil XP, por Kent Beck, no final dos anos 90 (FOWLER, 2013). A prática era uma peça importante para a metodologia, e se mostraria igualmente relevante no futuro, com o amadurecimento das metodologias ágeis, incluindo o DevOps. Ela consiste na atividade de integrar o trabalho, produzido de forma independente por cada desenvolvedor, com frequência (FOWLER; FOEMMEL, 2006). Isto é possível por meio de boas práticas de controle de versionamento de *software*, aliadas de técnicas de teste automatizado e servidores de *build* dedicados a validar testes unitários de maneira rápida e alertar falhas de integração. É importante que a etapa de integração contínua seja curta dentro do processo de entrega do *software*, por isso é comum que a mesma se atenha apenas à execução de testes unitários e análise estática do código. Esta prática faz com que o tempo gasto pela equipe de desenvolvimento com *bugs* de integração de código e planejamento de pacotes seja reduzido, pois evidencia estes problemas cedo no processo de desenvolvimento, conforme a filosofia *shift-left*[5], que permeia diversos conceitos da cultura DevOps (DEVOPEDIA, 2021).

[4] Do original: *continuous integration* (CI).

[5] Denota o ato de adiantar ações e atividades dentro do processo de desenvolvimento, partindo da ilustração teórica de um quadro de tarefas, onde os itens de trabalho a serem feitos estão listados da esquerda para a direita. Neste sentido, a ação de "trazer para a esquerda" significaria adiantar a execução de algo.

2.3.3 Entrega Contínua

A evolução seguinte no processo de aceleração da entrega de *software* é a implementação de uma esteira (*pipeline*) de "entrega contínua"[6] (FARLEY; HUMBLE, 2010). Por meio desta, é possível utilizar os artefatos validados, que foram produzidos pela fase de integração contínua, e executar uma carga maior de testes automatizados. Além disso, por meio desta esteira de entrega, profissionais responsáveis por testes funcionais podem solicitar o provisionamento e construção de ambientes de testes com facilidade.

Após todas as aprovações da fase final de testes, tanto por equipes de teste, quanto por usuários (se for o caso), a fase de implantação em produção pode ser automatizada, necessitando ou não de interação humana para ter seu início. Aqui, vemos o uso de ferramentas de infraestrutura como código, *scripts* e as capacidades de provedores de nuvem para fazer a implantação da nova versão, com quaisquer ajustes de configuração necessários. Para que isto seja feito de forma segura, é comum que sejam adotadas estratégias de lançamento (*release*) e reversão (*rollback*), que fogem ao escopo desta seção e são melhor discutidas por Farley e Humble (2010).

2.3.4 Monitoramento

Supondo uma implantação bem sucedida, é necessário ter mecanismos para monitorar a saúde e desempenho do sistema em produção. Aqui, retomamos a noção de medição dentro do DevOps, e a necessidade de *feedback* contínuo. No contexto de sistemas distribuídos, nos quais focamos ao longo deste trabalho, o monitoramento se prova ainda mais importante, em dimensões como: latência, taxa de tráfego, taxa de erros de aplicação e utilização de *hardware* (BEYER et al., 2016a). Com base em métricas como estas, combinadas com informações de *logs* e alertas da aplicação, é possível ter alguma confiança de que uma nova versão do *software* não causou problemas. Veremos, no entanto, que apenas métricas isoladas não apresentam a perspectiva total da resiliência de um sistema.

Conforme indicado quando abordamos o assunto de sistemas distribuídos, as interdependências inerentes aos mesmos fazem com que, muitas vezes, possíveis falhas possam ser imprevisíveis. Desta maneira, podemos lançar mão da capacidade de monitoramento e da infraestrutura de provisionamento automático para incorporar, no ciclo de entrega contínua de *software*, práticas de experimentação conhecidas como *Chaos Engineering*, que explicaremos e estudaremos nos próximos capítulos.

[6] Do original: *continuous delivery* (CD).

3 CHAOS ENGINEERING

Neste capítulo abordaremos o objeto principal deste trabalho, que é a compreensão do conceito de **Chaos Engineering**. Na primeira seção, apresentamos brevemente a história dos eventos que levaram à criação do termo e como este veio a se tornar relevante no mercado. Na segunda seção, dissertamos a respeito do que se trata *Chaos Engineering* em uma perspectiva ampla. Nas seções seguintes, nos debruçamos, por fim, sobre temas mais aprofundados dentro da disciplina de *Chaos Engineering*, discutindo requisitos técnicos e organizacionais, bem como desafios e benefícios de sua implementação em sistemas distribuídos.

3.1 ORIGENS

O conceito de *Chaos Engineering* surgiu a partir da experiência de engenheiros da empresa de *streaming* de vídeos Netflix que, a partir de 2008, começaram a ter problemas frequentes com sua infraestrutura e, consequentemente, com a disponibilidade de seus serviços (VLAOVIC et al., 2016). À época, todo o *hardware* que hospedava suas aplicações era administrado localmente (conhecido como modelo *on-premises*) e a crescente demanda dos serviços, combinada com a recente evolução do mercado de computação em nuvem, fez com que a equipe da empresa iniciasse um replanejamento da arquitetura de seus sistemas para migrar toda sua infraestrutura para o provedor Amazon Web Services (AWS). Com a nova arquitetura de microsserviços, combinada à flexibilidade da computação em nuvem, seria possível ter maior escalabilidade para atender à demanda. No entanto, a complexidade de gerenciar e testar todos os serviços aumentaria, já que o sistema estava se tornando mais distribuído, com maior número de partes independentes.

Foi neste cenário que a cultura de testes de resiliência e injeção de falhas se intensificou na empresa, para garantir que a migração estava ocorrendo sem problemas. Isto levou ao lançamento, em 2010, da ferramenta de código aberto *Chaos Monkey* (NETFLIX, 2010), cujo propósito é de desligar aleatoriamente máquinas virtuais em execução no ambiente de produção, com o intuito de forçar os desenvolvedores de software a desenvolver seus sistemas com resiliência a este tipo de evento. Pouco tempo depois, em 2011, a ferramenta evoluiu para o que é conhecido hoje como *Simian Army* (NETFLIX, 2011), que é uma suíte de ferramentas construídas com o objetivo de facilitar a automação de injeção de falhas, tais como: perda de conexão com máquinas, aumento de latência em comunicações, desligamento de máquinas, simulação de falha em zonas de disponibilidade do provedor de nuvem, entre outros.

Com o tempo, conforme as ferramentas e práticas foram ganhando robustez e relevân-

cia, a disciplina foi melhor formalizada em Rosenthal et al. (2016), um artigo publicado por engenheiros da Netflix. Ademais, outras empresas do mercado passaram a adotar práticas similares e o termo *Chaos Engineering* foi ganhando notoriedade no mercado ao final da última década, de acordo com levantamento publicado em InfoQ (2019), que coloca *Chaos Engineering* como uma prática em fase inicial de adoção pela indústria.

3.2 DEFINIÇÃO

O arcabouço de métodos e práticas chamado de *Chaos Engineering* tem como propósito criar nos desenvolvedores de sistemas distribuídos um senso maior de preocupação com resiliência, e foi definido por Rosenthal et al. (2016, p. 2):

> *Chaos Engineering* é a disciplina de experimentação em um sistema distribuído, com o objetivo de construir confiança em sua capacidade de suportar condições turbulentas em ambiente de produção.

O objetivo de experimentos de *Chaos Engineering* é trazer à tona falhas que derivam inerentemente do caos associado a sistemas distribuídos complexos; falhas estas que muitas vezes não são detectadas por testes comuns, por terem modelos de falha altamente imprevisíveis. Ressaltamos que, apesar da alusão ao "caos" presente no nome, *Chaos Engineering* não se trata de provocar falhas de forma desordenada nos sistemas, mas sim, de observar controladamente a reação de um conjunto de componentes quando sujeitos a condições adversas e o efeitos colaterais destas reações, potencialmente caóticas, no restante do sistema (ROSENTHAL et al., 2017) (JERNBERG, 2020).

Desta forma, diferenciamos a disciplina de *Chaos Engineering* da técnica pura de testes por injeção de falhas. Embora ambas tenham uma interseção prática, o conceito do teste propriamente dito pressupõe uma condição de sucesso conhecida e uma saída esperada, enquanto a prática de *Chaos Engineering* se aproxima muito mais de uma análise exploratória, que se utiliza de injeção de falhas como ferramenta (ROSENTHAL et al., 2017). Outro ponto chave onde *Chaos Engineering* se diferencia é na prática de experimentos em ambiente de produção. Em um contexto normal de desenvolvimento de *software*, a execução de testes (dentre eles: testes unitários, de integração, de carga, funcionais, etc.) é normalmente feita antes de colocar uma aplicação em produção, onde receberá tráfego e carga de trabalho críticos. No entanto, como o objetivo de *Chaos Engineering* é expor problemas desconhecidos, o ambiente sujeito aos experimentos deve ser o mais próximo possível de produção (com o objetivo de ser o próprio, eventualmente), já que há parâmetros de configuração, versões de software, estados de sistemas externos e diversas outras variáveis que podem ser facilmente ignoradas e afetariam a validade dos testes realizados (ROSENTHAL et al., 2017).

Em essência, experimentos de *Chaos Engineering* podem ser entendidos como uma prova por contradição. Iniciamos experimentos partindo da hipótese de que o sistema

é resiliente a um determinado tipo de falha e, caso um problema seja encontrado na aplicação, consideramos esta hipótese como anulada. Com base neste novo conhecimento, podemos investigar a causa das falhas. Isso permite ganhar, progressivamente, mais confiança na resiliência do sistema como um todo, já que a complexidade do mesmo não permite medir essa de forma analítica, então precisamos experimentar de forma iterativa para nos aproximarmos de um resultado satisfatório (JERNBERG, 2020).

Considerando tais definições, notamos que a adoção de *Chaos Engineering* não substitui a execução de nenhum tipo de teste de software, já que seu propósito é de revelar falhas não previstas, enquanto a premissa de um teste envolve conhecer o objetivo final do *software* e testar pontos específicos do código para garantir que este objetivo seja atingido. Por este mesmo motivo, ressaltamos que as injeções de falhas preconizadas por *Chaos Engineering* não devem ser aplicadas em partes do sistema com problemas conhecidos, onde se sabe que o sistema vai falhar e causar problemas para o negócio. Nestes casos, a falha deve ser testada e corrigida seguindo uma metodologia convencional de desenvolvimento de *software* (ROSENTHAL et al., 2017) (JERNBERG, 2020).

3.3 PRÉ-REQUISITOS DE IMPLEMENTAÇÃO

A aplicação de *Chaos Engineering* dentro de uma organização vem com alguns requisitos de maturidade técnica e cultural. Naturalmente, é um desafio cultural fazer com que equipes de operações e gestores aceitem ter falhas injetadas propositalmente em seus sistemas expostos ao cliente final. Da mesma maneira, é um desafio técnico colocar ferramentas e controles apropriados em posição para monitorar com precisão toda a infraestrutura e estatísticas da saúde do sistema.

Para que se possa começar a aplicar qualquer prática de *Chaos Engineering*, pressupõe-se que os componentes sobre os quais se deseja experimentar estejam devidamente mapeados e que se tenha um conhecimento preliminar sobre suas interdependências e possíveis pontos de falha. Como dito anteriormente, também é imprescindível ter ferramentas de monitoramento configuradas para prover um fluxo constante e atualizado de informações acerca do estado interno de cada serviço. Tentar aplicar *Chaos Engineering* sem cumprir estes requisitos seria uma tarefa, além de arriscada (supondo a aplicação em ambiente de produção), pouco proveitosa, já que seria um trabalho às cegas, sem nenhum fruto em termos de conhecimento novo sobre o comportamento da aplicação perante eventos imprevisíveis (ROSENTHAL et al., 2017).

Chaos Engineering é, por todos os motivos já citados, uma prática mais apropriada para organizações que já têm métodos e processos de TI mais robustos. Desta forma, além de poder aplicar os experimentos em produção com mais segurança, seu potencial

também poderá ser mais explorado, já que o propósito final das práticas de *Chaos Engineering* é de, necessariamente, prover conhecimento novo sobre interações intercomponentes e fraquezas desconhecidas do sistema, complementando outros testes de validação funcional (ROSENTHAL et al., 2017) (SAULITIS et al., 2017). Isto não impede, no entanto, que organizações de menor porte iniciem seus passos na adoção da metodologia, podendo implementar ferramentas e experimentos mais simples em ambientes de menor criticidade, dado que tenham recursos de monitoramento suficientes para tirarem proveito dos resultados.

3.4 PRINCÍPIOS DO CAOS

Com a evolução das técnicas e métodos para a aplicação de *Chaos Engineering* no contexto das organizações, os engenheiros que criaram o conceito desenvolveram, com a colaboração da comunidade, um arcabouço de princípios, entitulados "Princípios do Caos" (CHAOS COMMUNITY, 2019), que buscam trazer uma padronização e uma formalização para as experimentações feitas. Trata-se de um conjunto de cinco atividades, que devem ser realizadas progressivamente, aumentando a confiabilidade trazida pelos experimentos.

Ao longo das próximas seções, elaboramos sobre cada um dos princípios, que consistem em:

- Criar hipóteses sobre um estado de equilíbrio;
- Variar eventos do mundo real;
- Fazer experimentos em ambiente de produção;
- Automatizar a execução dos experimentos;
- Minimizar o "raio de explosão".

3.4.1 Criar hipóteses sobre um estado de equilíbrio

No contexto de *Chaos Engineering*, um "estado de equilíbrio" se refere a uma métrica (ou conjunto de métricas) que, ao ser observada no sistema ao longo de um intervalo razoável de tempo, permite identificar um comportamento esperado do sistema como um todo; um ponto de equilíbrio, em que temos confiança de que o sistema está se comportando de forma adequada.

A princípio, poderíamos ser levados a considerar, principalmente, métricas de *hardware*, que já são fornecidas pelas ferramentas de monitoramento disponíveis no mercado. No entanto, tais métricas podem ter alta variabilidade e, quando consideradas de forma isolada, podem não contar a história completa sobre o estado do sistema. No contexto

de *Chaos Engineering*, não devemos olhar apenas para as métricas técnicas do sistema (como uso de memória, CPU, etc.), mas sim, atrelar o conceito de estado de equilíbrio a métricas que permitam medir o quão bem o sistema está realizando seu objetivo final, ou seja, precisamos também de métricas de negócio. Desta forma, a situação ideal é utilizar um conjunto de métricas técnicas e de negócio para estabelecer uma linha de base para o comportamento do sistema como um todo (JERNBERG, 2020).

Em geral, o estado de equilíbrio pode ser flutuante e periódico, mas a ideia gira em torno de observar e estabelecer um comportamento de base desejado, como função do tempo, e continuamente monitorar e atuar nos sistemas com o objetivo de manter as métricas dentro do intervalo esperado. Determinar os limites aceitáveis desse intervalo é um dos desafios envolvidos nos experimentos de *Chaos Engineering* e é fundamental para a validade dos testes sobre as hipóteses definidas.

Tendo este conhecimento, podemos partir para as hipóteses propriamente ditas. Esta parte é o fundamento de *Chaos Engineering* como disciplina, pois é o que separa experimentação rigorosa de observação desordenada de falhas. De posse de métricas confiáveis, podemos iniciar experimentos nos perguntando como a falha injetada (ou conjunto de falhas) pode perturbar nossas métricas. Se entendemos bem o estado de equilíbrio e a responsabilidade de um determinado componente do sistema, certamente somos capazes de, ao menos, ter uma ideia ou palpite sobre como uma falha no mesmo pode fazer com que as métricas se afastem da zona de equilíbrio. Quando executamos *Chaos Engineering*, a premissa é de que o sistema será resiliente à falha injetada, pois o propósito do método de experimentação é ir de encontro à essa hipótese, pondo, assim, a resiliência do sistema à prova.

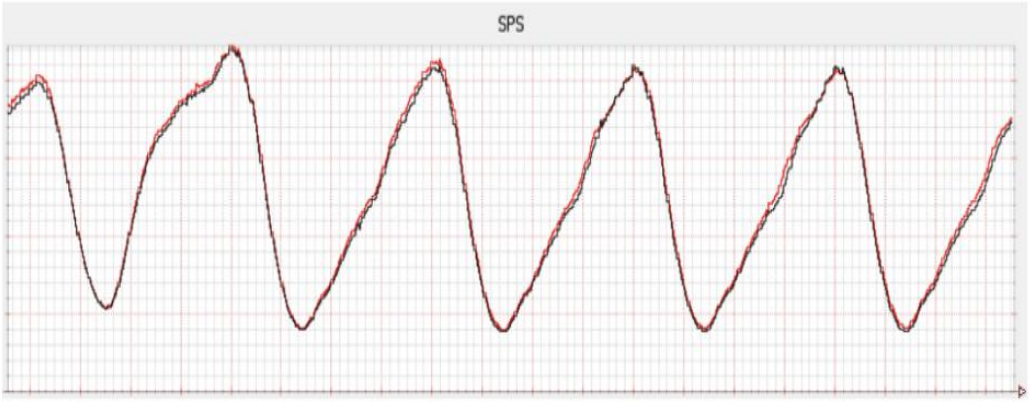
Podemos tomar como um exemplo concreto, as experiências relatadas por engenheiros da Netflix, que utilizam como métrica principal o que eles chamam de "inícios de *stream* por segundo" (na sigla em inglês, SPS). Ela os permite ter um "sinal vital" — como definido por eles — do funcionamento dos sistemas da empresa (NETFLIX, 2015). Esta é uma métrica fundamentalmente de negócio, que reflete o comportamento do usuário final interagindo com o sistema, e provê uma forma de avaliar diretamente o grau de impacto de incidentes e falhas em seus usuários. Em conjunto, métricas de *hardware* podem ser utilizadas para avaliar impactos no desempenho, que, num cenário de estresse no sistema, também podem acarretar em impacto no negócio (ROSENTHAL et al., 2016).

A Figura 2 exibe um exemplo do comportamento da métrica de SPS no tempo, apresentando uma linha vermelha, que representa o histórico da métrica na semana em que foi medida, e uma preta, representando o sinal da semana anterior. O movimento oscilatório é indicativo do uso natural da plataforma, onde os usuários são mais propensos a iniciar transmissões no horário noturno, e, conforme o dia acaba, a frequência de vídeos iniciados

diminui. Os rótulos indicativos dos eixos x e y foram suprimidos intencionalmente pela companhia, a fim de preservar eventuais segredos industriais.

Os ciclos naturais deste sinal são monitorados periodicamente, e os engenheiros da empresa desenvolveram algoritmos refinados de análise de dados para reconhecer variações anormais (NETFLIX, 2015).

Figura 2 – Amostra gráfica da métrica de SPS ao longo de duas semanas



Fonte: Netflix (2015)

3.4.2 Variar eventos do mundo real

Em sistemas suficientemente distribuídos e complexos, existe uma gama muito ampla de falhas que podem ocorrer, desde falhas de hardware e comunicação em rede, até problemas causados por entradas de dados inválidas. Em Rosenthal et al. (2017), os engenheiros criadores da disciplina de *Chaos Engineering* listam as categorias mais comuns de falhas:

- Falhas de *hardware*;
- *Bugs* funcionais;
- Erros na divulgação de estados entre componentes;
- Latência e particionamento de redes;
- Flutuação da frequência de entrada de dados;
- Exaustão de recursos;
- Combinações incomuns ou imprevisíveis de comunicação interserviços;

- Falhas bizantinas;
- Condições de corrida;
- Falhas em dependências externas.

Diante desta alta variabilidade de possíveis falhas, se torna necessário estabelecer uma ordem de prioridade para gerenciar o caos e ser possível executar experimentos de maneira controlada. É um trabalho para as equipes responsáveis pela implementação dos testes, estimar a relação custo-benefício em escolher trabalhar cada tipo de falha em um experimento de *Chaos Engineering*. Torna-se pouco interessante testar um determinado modo de falha se ele for altamente improvável, dado o contexto de utilização do sistema. No entanto, se houver uma possibilidade razoável de ocorrência, recomenda-se medir o grau dos danos causados no evento da ocorrência desta falha e os custos envolvidos em sistematizar a injeção da mesma no sistema (TUCKER et al., 2018).

Vale citar que, quando falamos de custos, há também custos culturais envolvidos, além dos financeiros. Se a resiliência do sistema for superestimada, ou se o monitoramento não for feito de maneira adequada, um experimento pode causar impactos indesejados ao negócio, e a existência deste risco pode criar dificuldades ao se tentar difundir a cultura de *Chaos Engineering* nas organizações (TUCKER et al., 2018). Por este motivo, os modos de falha trabalhados devem ser muito bem analisados e as dependências internas entre serviços devem ser levadas em consideração, para limitar a ocorrência de falhas em cadeia. No entanto, esta possibilidade também é mais um fator de motivação para que sejam feitos tais experimentos, pois eles podem ajudar a expor estes domínios de falha. As interações entre diferentes tipos de falha podem fazer surgir dependências que não estavam claras entre componentes do sistema, e evidenciar estas interdependências pode contribuir muito para o aumento progressivo da resiliência do sistema.

Quando falamos de *Chaos Engineering*, apesar do objetivo final ser a execução dos testes em ambiente de produção, tratamos a injeção de falhas como um processo que deve ser evolutivo, começando em domínios pequenos do sistema, num ambiente isolado onde se tem um bom entendimento das interações e dependências entre os componentes. Nesta etapa, inicia-se com falhas mais simples, como simulações de desligamento de máquinas, aumento de latência, falhas em chamadas RPC específicas, entre outras. Conforme a resiliência a estas falhas aumenta, juntamente com o nível de confiança no processo, e mais controles estão posicionados para conter danos (como abordaremos na subseção 3.4.5), é interessante abrir mais o leque de possibilidades, aumentando o conjunto de partes do sistema sujeitas a testes, e movendo os testes para o ambiente de produção, onde o comportamento real do sistema possa ser posto à prova (BUTOW; SALINAS, 2018).

3.4.3 Fazer experimentos em ambiente de produção

Neste ponto, *Chaos Engineering* mostra mais evidentemente sua diferença com relação à prática de testes automatizados. Geralmente, testes são executados antes de colocar o sistema em produção, com o objetivo de não observar falhas em ambiente crítico. No entanto, o objetivo final de um experimento de *Chaos Engineering* é ser executado em produção, pelo motivo de que o comportamento do sistema só será mostrado em sua real natureza, no ambiente onde ele é utilizado pelo usuário final (ROSENTHAL et al., 2017). Características como aleatoriedade, periodicidade e carga do tráfego recebido pelo sistema são inerentes ao ambiente de produção e a tentativa de replicar o ambiente de alguma forma diminuiria a validade do mesmo, já que inevitavelmente introduziria diferenças entre o sistema cuja resiliência queremos validar, e o sistema efetivamente testado.

Apesar de encorajar experimentos com certo nível de risco em produção, o intuito de *Chaos Engineering* não é de colocar à prova falhas conhecidas apenas com o intuito de ver o sistema falhando em produção. Como dito anteriormente, a hipótese sempre deve ser de que o sistema consegue resistir à injeção de falhas, do contrário, é muito menos custoso simplesmente depurar o código e consertar a falha. Além disso, a decisão de executar experimentos em produção tem uma resistência cultural muito forte nas equipes de TI e deve ser discutida e ter sua relação risco-retorno bem avaliada pelas partes envolvidas. Uma prática muito comum para construir confiança e avançar gradualmente a prática de experimentações é utilizar *Game Days* (ROBBINS, 2011), onde equipes de desenvolvimento e operações se reúnem periodicamente, por um tempo determinado, para executar um experimento contido e monitorado de perto.

3.4.4 Automatizar a execução dos experimentos

Ao conquistar confiança nos controles e processos ao redor dos experimentos de *Chaos Engineering*, o próximo passo para aumentar a maturidade da prática dentro da organização é o de automatizar a execução dos testes. Esta tarefa vem com o desafio de refinar mais ainda as métricas de saúde do sistema e de ter ferramentas que permitam controlar e interromper experimentos que saiam de controle, a qualquer momento.

O ato de automatizar experimentos possibilita manter a confiança na resiliência do sistema renovada, já que as dificuldades da mobilização de capital humano em reuniões para testes manuais pode fazer com que o intervalo entre experimentos seja grande. Com isto, é possível que muitas alterações sejam feitas no sistema dentro do intervalo de tempo entre testes, o que diminui a confiabilidade e aumenta o trabalho de investigação quando um problema é, eventualmente, encontrado (ROSENTHAL et al., 2017). Este conceito remete e se conecta com a noção de *feedback* contínuo, que é um dos pilares da cultura DevOps, cuja fundamentação pode ser encontrada na seção 2.3.

Por meio da automação, consegue-se extrair o maior potencial do método de *Chaos Engineering*, já que, com falhas sendo constantemente simuladas no sistema, os desenvolvedores são naturalmente compelidos a construir o código utilizando boas práticas de desenvolvimento orientado a resiliência. Num cenário de desenvolvimento ágil, onde as entregas de código são feitas com frequência, *bugs* que potencialmente ferem a resiliência do sistema podem ser introduzidos, e sem experimentos automáticos sendo executados rotineiramente, estes problemas podem se manifestar espontaneamente em produção, impactando o usuário final.

3.4.5 Minimizar o "raio de explosão"

Ao longo da discussão e fundamentação destes "Princípios do Caos", salientamos a necessidade de controlar e limitar danos ao negócio durante a execução dos experimentos. O conceito utilizado pelos autores para tratar desta temática é o de "raio de explosão"[1]. Se trata do conjunto de componentes e funcionalidades de um sistema afetadas por uma eventual indisponibilidade em uma dada parte do mesmo. Pela complexidade dos sistemas e o caos inerente às interações entre os mesmos e o mundo externo, a injeção de perturbações pode ser algo extremamente volátil e ter efeitos colaterais pouco previsíveis. Por este motivo, a adoção de recursos de mitigação de riscos às ferramentas e plataformas de *Chaos Engineering* é fundamental. Tais recursos de mitigação de riscos incluem:

- Habilidade de interromper o experimento de forma razoavelmente rápida e a qualquer momento;
- Limitar o percentual do volume de tráfego (de produção) direcionado a máquinas ou contêineres onde esteja havendo injeção de falhas;
- Configurar a automação dos experimentos para executar preferencialmente em horários onde seja possível uma resposta humana rápida, em caso de problemas.

Jernberg (2020) ressalta que o raio de explosão que buscamos minimizar é do ponto de vista de métricas de negócio, as mesmas estabelecidas no início da jornada de implementação do método de *Chaos Engineering*. Isso não significa, necessariamente, uma minimização dos impactos colaterais nas áreas de tecnologia. Na realidade, a ocorrência destes impactos seria uma evidência contrária à hipótese de resiliência estabelecida, e é parte do que consiste o valor deste método de experimentação.

[1] Do original: *blast radius*.

4 CHAOS ENGINEERING NO MERCADO

Nos últimos anos, a adoção de práticas de *Chaos Engineering* vem crescendo de forma acelerada no mercado de tecnologia, conforme sugerem os relatórios InfoQ (2019) e Gremlin (2021b). Esses estudos ressaltam um aumento expressivo nas pesquisas por termos relacionados ao tema (com base em uma análise dos últimos quatro anos), além de um percentual de 60% das empresas avaliadas no estudo tendo executado pelo menos um experimento de *Chaos Engineering* e 34%, com um processo maduro de execução de experimentos em produção.

Tal ganho em relevância pode ser explicado pelo fortalecimento da cultura DevOps nas organizações, trazendo a filosofia chamada de *shift-left*, que consiste na automação de processos de teste e implementação de software, com o intuito de evidenciar e corrigir falhas com mais antecedência dentro do processo de desenvolvimento de software (DEVOPEDIA, 2021) (INFOQ, 2019). Outro fator que contribui para o aumento na adoção de práticas de *Chaos Engineering*, este mais contemporâneo, são as dificuldades de manter sistemas cada vez mais complexos num cenário em que a comunicação das equipes de desenvolvimento e de sustentação é prejudicada pela distância e assincronismo. Nos referimos, especialmente, ao período da pandemia de COVID-19, nos anos de 2020 e 2021, que pôs à prova a resiliência de muitos sistemas, como foi possível ver em notícias que ecoaram nas comunidades de tecnologia ao redor do mundo nos últimos meses (FACEBOOK, 2021) (TECHCRUNCH, 2020). Tanto o novo paradigma de trabalho, quanto o aumento expressivo no tráfego e na dependência dos usuários de certos produtos, evidenciaram a necessidade de assegurar a resiliência dos sistemas desenvolvidos (CENTER, 2020).

Ao longo deste capítulo, apresentamos exemplos de ferramentas e implementações de *Chaos Engineering* de forma mais concreta, abrangendo diversas empresas de onde foi possível encontrar relatos, tais como: Netflix, Amazon, Google e LinkedIn. Em decorrência da sua grande relevância no mercado, os exemplos de sucesso destas empresas impulsionam outros atores da indústria a experimentar com métodos similares, como foi o exemplo da Netflix, onde foi cunhado o termo *Chaos Engineering* e cuja equipe foi ativamente responsável pela produção de conhecimento e ferramentas para a comunidade de tecnologia de forma mais abrangente.

4.1 ESTUDOS DE CASO EM EMPRESAS

Nesta seção, buscamos agregar e organizar alguns dos relatos de experiências com *Chaos Engineering* por parte da comunidade que circunda o tema, formada principalmente por grandes empresas de tecnologia. A mais eminente destas, em termos de ex-

periência relatada com a metodologia, é a Netflix, que produziu diversos artigos sobre experimentos e ferramentas em seu *blog* de tecnologia, além de ecoar suas experiências em conferências e outros canais de comunicação. Ademais, abordaremos os casos de outras grandes empresas do cenário de tecnologia atual (Amazon, Google e LinkedIn), que também podem ser consideradas pioneiras no tema de *Chaos Engineering*, tendo contribuído direta e indiretamente para a construção das técnicas e práticas associadas à metodologia. Destas, faremos uma breve exposição de algumas ferramentas e práticas utilizadas, na medida do que é possível resgatar na *web* na forma de conteúdo aberto, até o momento.

É importante ressaltar que os exemplos mostrados não necessariamente representam um relato fidedigno da média do mercado, uma vez que existe um percentual expressivo de empresas utilizando *Chaos Engineering*, conforme estudo em Gremlin (2021b), mas acreditamos que muitas não relatam de forma oficial suas experiências por questões de competição de mercado, segredos industriais, ou pela falta de uma cultura de produção de conteúdo livre por parte da empresa.

4.1.1 Netflix

Sendo a pioneira na prática de *Chaos Engineering*, a empresa desenvolveu, ao longo dos anos, diversas ferramentas internas para automatizar e sistematizar o processo de testes, tendo também publicado algumas destas de forma *open source*.

A história de *Chaos Engineering* na Netflix começa com sua migração para o provedor de nuvem pública AWS iniciada em 2008. Posteriormente, em 2010, foi lançada a ferramenta *Chaos Monkey* (NETFLIX, 2020), cujo propósito era forçar a parada definitiva de instâncias virtuais de computação na nuvem e de serviços internos, de forma pseudo-aleatória. Isto visava a garantir que todos os sistemas se manteriam operacionais como desejado, mesmo em cenários de perda de algumas instâncias, o que poderia acontecer no novo ambiente, devido ao compartilhamento de recursos e o funcionamento geral da nuvem pública (NETFLIX, 2010). A ferramenta era usada em horário comercial, de maneira que as equipes responsáveis poderiam responder a eventuais incidentes e aprender sobre as fraquezas dos sistemas no processo.

Como a ferramenta *Chaos Monkey* permitia apenas testes que envolvessem o desligamento de máquinas, eventualmente surgiu o desejo de expandir as possibilidades de injeção de falhas. Com isto, o universo de ferramentas foi ampliado, para o que foi chamado de *The Simian Army* (NETFLIX, 2011). Este "exército" consiste de um conjunto de ferramentas para tipos específicos de teste, a saber:

- **Janitor Monkey**: busca por recursos não utilizados no ambiente de nuvem (exclusivamente AWS) e decide se deve apagar os mesmos baseado num conjunto configurável de regras, como tempo sem uso, presença de dependências entre recursos (ex.:

uma unidade de armazenamento sendo utilizada por uma instância de computação), entre outros;

- **Conformity Monkey**: permite ao usuário definir um conjunto de regras de conformidade e vasculha o ambiente de nuvem, excluindo recursos que não se encaixem nas regras;
- **Security Monkey**: varre as definições de recursos de contas de um provedor de nuvem pública (AWS ou GCP) e provê alertas em casos de potenciais vulnerabilidades ou violações de políticas de segurança, como configurações de permissões, acessos de rede pouco restritos, e certificados de criptografia prestes a expirar. Atualmente, seu desenvolvimento foi descontinuado em favor de ferramentas nativas dos provedores de nuvem, mais robustas e apropriadas para este tipo de tarefa (NETFLIX, 2021c);
- **Chaos Monkey**: implementa um agente que escolhe, periodicamente e de forma pseudoaleatória, uma instância virtual no ambiente de nuvem para ser desligada;
- **Latency Monkey**: injeta latência em chamadas de rede entre serviços, de forma aleatória;
- **Doctor Monkey**: monitora e avalia métricas de máquinas virtuais no ambiente de nuvem, como carga de CPU, e uso de memória, deletando instâncias que sejam consideradas fora dos parâmetros desejados pelo usuário e notificando-o nestes eventos;
- **10-18 Monkey**: sendo seu nome derivado da sigla l10n-i18n (*Localization and Internationalization*), esta ferramenta tem como propósito detectar problemas na configuração de instâncias que estejam executando em locais geográficos diferentes, que possam estar usando diferentes idiomas e conjuntos de caracteres diferentes;
- **Chaos Gorilla**: simula, de forma similar ao *Chaos Monkey*, a indisponibilidade de uma "zona de disponibilidade"[1] da AWS;
- **Chaos Kong**: também específico para o ambiente AWS, esta ferramenta simula a indisponibilidade de uma "região"[1] inteira do provedor de nuvem.

Destes, apenas os três primeiros tiveram versões *open source* disponibilizadas, além do *Chaos Monkey*, que foi lançado separadamente em primeiro lugar, e teve uma versão 2.0 publicada em 2016 (GREMLIN, 2018b). Alguns não foram disponibilizados devido à especificidade de sua solução aos casos de uso internos da Netflix, e outros, como o *Chaos Kong* e o *Latency Monkey*, não o foram por terem potencial para causar problemas

[1] Esclarecemos o significado destes termos na subseção 2.2.2.

grandes com facilidade, conforme a experiência obtida pelos desenvolvedores da empresa durante o tempo em que estas ferramentas foram utilizadas (NETFLIX, 2014).

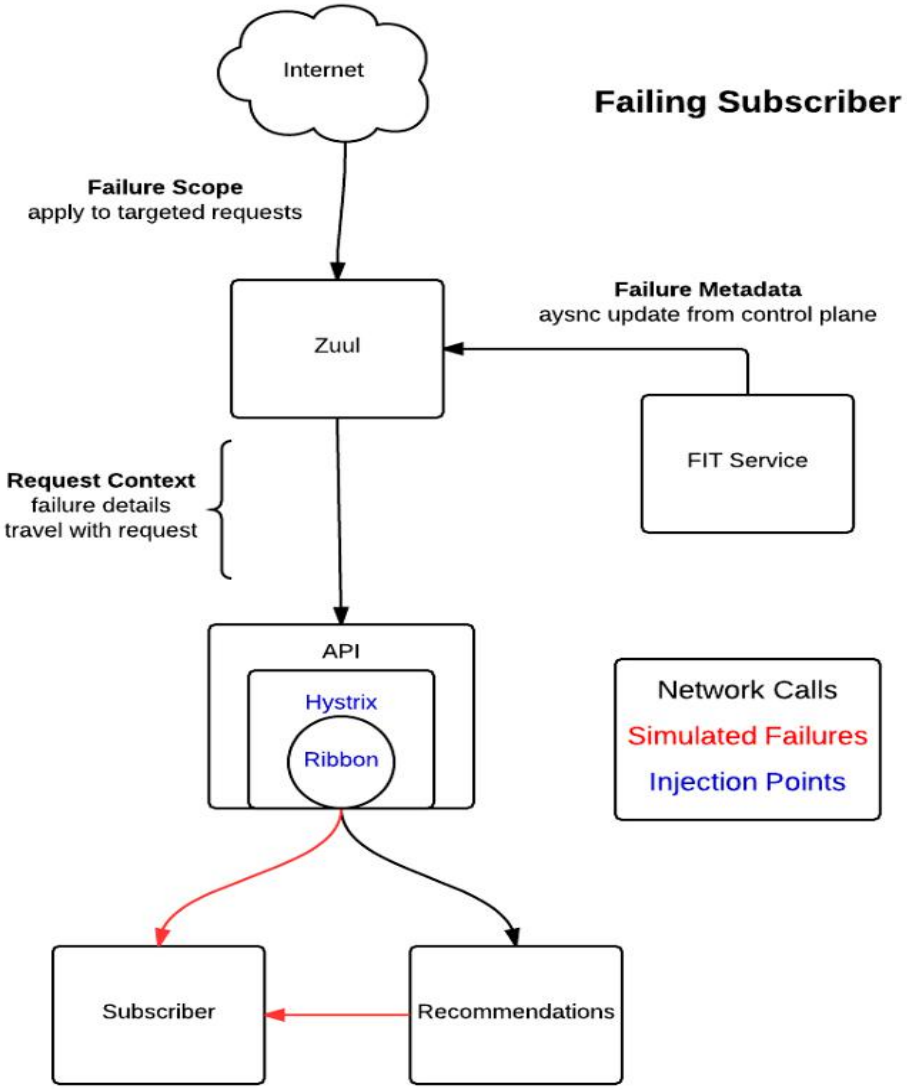
Com as ferramentas desenvolvidas até então, a companhia tinha meios para executar testes de forma parcialmente automatizada, mas ainda contava com um grande esforço de comunicação e coordenação entre as equipes de desenvolvimento e operações para limitar o raio de explosão[2] de falhas. Neste contexto, é criada a plataforma FIT (*Failure Injection Testing*) (NETFLIX, 2014). Seu propósito principal é permitir a difusão da prática de *Chaos Engineering* na organização, por meio da simplificação do processo de criação de experimentos e com uma arquitetura que permite limitar o impacto das falhas com precisão. A plataforma trabalha com outras ferramentas e bibliotecas internas, possibilitando a injeção de tipos diferentes de falhas em conjuntos restritos de requisições às APIs internas do sistema.

A Figura 3 detalha o fluxo de chamadas envolvidas na simulação de falha de um serviço chamado "*Subscriber*". O componente chamado *Zuul* é o gateway de todas as APIs internas da Netflix, sendo o responsável por receber as requisições e, portanto, ponto chave para a injeção de falhas (NETFLIX, 2021d) (NETFLIX, 2014). O serviço FIT gera metadados relacionados às falhas que devem ser inseridas, e comanda o *Zuul* a inserí-los em um percentual definido do tráfego, ou apenas em chamadas de uma origem específica. Os metadados são, então, interpretados e as respectivas falhas simuladas nos serviços internos pelas bibliotecas *Hystrix* e *Ribbon*, que também foram desenvolvidas dentro da empresa (NETFLIX, 2021a) (NETFLIX, 2021b). Estas bibliotecas implementam mecanismos e algoritmos orientados à resiliência em sistemas distribuídos, sobre os quais comentamos em mais detalhes na subseção 2.1.3.1. Pelo fato de ambas bibliotecas lidarem diretamente com comunicação inter-serviços, elas foram estendidas internamente para interagir com os metadados inseridos pelo FIT e servem como ponto de injeção das falhas na ponta final do fluxo.

A criação desta plataforma possibilitou a adesão de mais desenvolvedores da organização às práticas de *Chaos Engineering*, pois um dos fatores que suprimia o potencial de difusão da metodologia dentro da empresa era a dificuldade de limitar o raio de explosão das falhas (NETFLIX, 2014). Isso trouxe um impacto positivo para a maturidade da disciplina e dos sistemas da Netflix, e fez com que mais esforços fossem empregados na automação dos experimentos e otimização dos resultados encontrados. Estes esforços culminaram no sistema usado atualmente, chamado ChAP (*Chaos Automation Platform*), que utiliza amplamente a plataforma FIT para operacionalizar a injeção de falhas, mas automatiza o processo e integra com outros sistemas de infraestrutura e monitoramento da Netflix para prover métricas mais isoladas (NETFLIX, 2017a).

[2] Conforme definido na subseção 3.4.5

Figura 3 – Simulação de uma falha utilizando a plataforma FIT



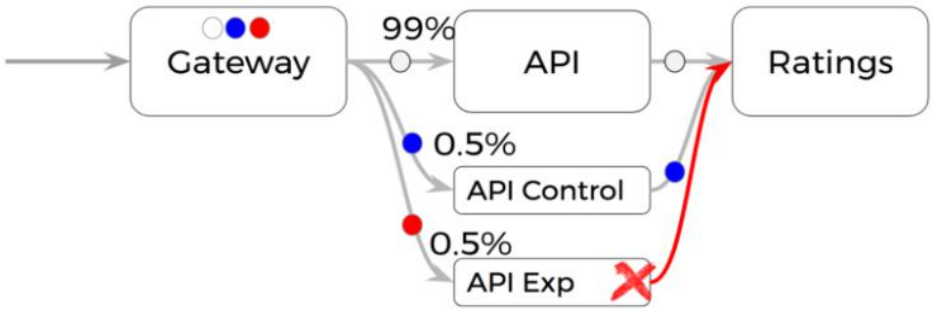
Fonte: Netflix (2014)

Essencialmente, a plataforma ChAP consiste num orquestrador, que se comunica com a ferramenta de entrega contínua de *software* interna da empresa e outros serviços para provisionar recursos extras de infraestrutura, proporcionais ao tamanho do experimento desejado, de forma a isolar e ter controle granular sobre os impactos de um experimento. O diferencial está na criação de dois subgrupos dentro dos servidores recém provisionados: um onde será induzida a falha e outro que servirá como controle, e ambos recebem a mesma quantidade de tráfego. A Figura 4 mostra a dinâmica de particionamento de tráfego aplicada a um experimento de simulação de falha do serviço "Ratings", com 1% do tráfego direcionado ao teste (0,5% controle e 0,5% com falha).

Este método faz com que falhas de resiliência sejam mais visíveis, pois o impacto nas métricas consegue ser muito mais relevante quando comparado ao grupo de controle do

que se comparado às métricas geradas pelos outros 99% de volume de tráfego, como evidenciado na comparação entre os gráficos das métricas de "inícios de *streams* por segundo" (SPS), utilizada como referência para as análises na Netflix.

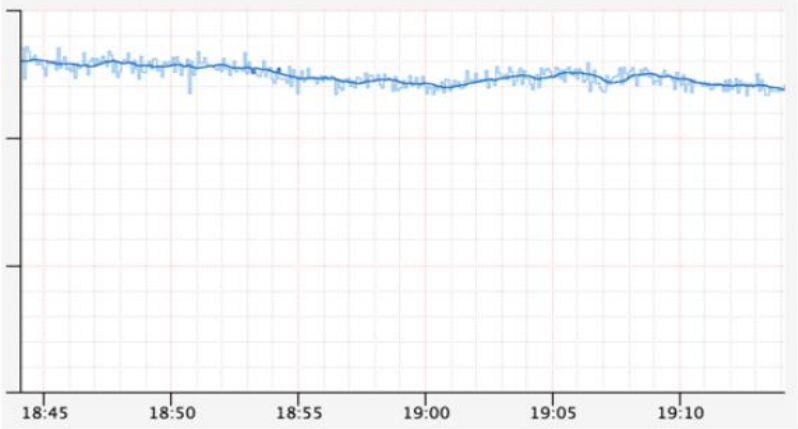
Figura 4 – Fluxo de tráfego numa simulação de falha utilizando o ChAP



Fonte: Netflix (2017a)

Para exemplificar este efeito, vemos na Figura 5 um exemplo provido pela Netflix em seu *blog*, onde usou-se a plataforma FIT para induzir uma falha em um percentual pequeno do tráfego. No gráfico, o eixo x representa o tempo, onde a falha foi inserida entre as marcas 19:05 e 19:10, e o eixo y, que representa a métrica de SPS, para a qual os autores não proveram valores por questões de segredo industrial. Ao analisar o gráfico, percebe-se que, por volta da marca de tempo de início da falha, não é possível notar nenhuma variação expressiva do sinal em relação ao estado de equilíbrio.

Figura 5 – Histórico de SPS medido em experimento feito com a plataforma FIT



Fonte: Netflix (2017a)

Em contraste, a Figura 6 mostra que, ao isolar o mesmo percentual de tráfego do experimento anterior para dois servidores de teste, um de controle e outro com a falha

injetada, é possível ver resultados muito mais expressivos, evidenciados na redução da métrica de SPS para o servidor submetido à falha.

Figura 6 – Históricos de SPS medidos pela plataforma ChAP para grupo de controle (azul) e experimental (vermelho)

Fonte: Netflix (2017a)

4.1.2 Amazon

Os exercícios de testes por injeção de falhas na Amazon começaram por volta do início dos anos 2000, com a introdução de dinâmicas de simulação de falhas. Motivados por eventos de indisponibilidade ocorridos em seus sistemas no passado recente (ROBBINS et al., 2012), a equipe começou a implementar eventos que vieram a ser chamados de *GameDays*, com o propósito de exercitar operações de resposta a falhas e incidentes nos sistemas. Este modelo de *Chaos Engineering*, na forma de eventos regulares, se tornaria uma prática comum na indústria, que abordaremos em maior detalhe na seção 4.2.

Não há muitos relatos disponíveis sobre a implementação de experimentos e ferramentas específicas dentro da Amazon, mas, em 2020, a empresa divulgou um novo serviço para automação e experimentos de *Chaos Engineering*, chamado AWS Fault Injection Simulator (AMAZON, 2021). Por meio deste, é possível implementar diversos tipos de injeção de falhas em recursos de infraestrutura da nuvem AWS, utilizando arquivos de configuração que descrevem os experimentos. Abordamos o funcionamento da ferramenta na seção 5.3.

4.1.3 Google

A principal prática de *Chaos Engineering* implementada pela empresa é a execução periódica de eventos de falha, executados por uma equipe designada para testar os proces-

sos internos de recuperação de desastres. Isto significa que os testes, além de envolverem a injeção de falhas nos sistemas de *software* e *hardware* da companhia, também envolvem exercitar falhas de comunicação e a resposta a incidentes de equipes responsáveis por sistemas críticos em cenários de pressão (ROBBINS et al., 2012).

O foco destes exercícios é de colocar à prova eventos e situações que têm baixa probabilidade de acontecer, mas com impactos críticos ao negócio da empresa. Isto é possível pois existe um esforço conjunto de diversas áreas para que os casos de falha mais triviais sejam cobertos por uma estratégia de testagem contínua (KRISHNAN, 2012), permitindo que experimentos de maior escala possam se concentrar em exercitar falhas sistêmicas, envolvendo mais equipes e componentes de *software*.

A organização que a Google utiliza para sua equipe DiRT (Disaster Recovery Testing), responsável pela execução dos processos de *Chaos Engineering* na companhia, divide a mesma em duas subequipes: técnica e de coordenação (KRISHNAN, 2012). A equipe técnica tem como função preparar e monitorar experimentos, assim como responder a eventos fora do planejado causados pelos mesmos. A equipe de coordenação, por outro lado, tem a atribuição de planejar e organizar os calendários de testes e garantir que todos os envolvidos sejam notificados com antecedência e que eventuais ambientes virtuais de testes estejam prontos para receber carga.

4.1.4 LinkedIn

Historicamente, o LinkedIn tinha sua plataforma de *software* implementada numa arquitetura monolítica, com todos os componentes integrados numa mesma base de código. Ao longo do tempo, com o ganho de notoriedade do produto, a empresa teve de mudar sua arquitetura para um modelo distribuído, para lidar melhor com a alta demanda e tamanho crescente das equipes de desenvolvimento. Com isto, adotaram um modelo de microsserviços, que os permitia um ganho de escala e velocidade de crescimento, mas que também trouxe o problema do alto grau de interdependência dos serviços (INFOQ, 2018).

A iniciativa criada dentro da empresa para controlar este problema e obter maior confiança na resiliência destes sistemas foi o "Project Waterbear" (DEVARAJ; LI, 2017). O projeto tem 3 principais componentes, que atuam em diferentes camadas dos sistemas da companhia. O primeiro componente do projeto atua no contexto de *software*, e consiste na ferramenta LinkedOut (ROSEN, 2018), que é um módulo integrado ao *framework* interno de gerenciamento da comunicação HTTP entre serviços. Este módulo permite filtrar requisições e injetar falhas apenas no contexto de usuários e rotas definidas, limitando a possibilidade de impactos colaterais dos testes. O segundo componente trata da configuração deste mesmo *framework*, permitindo monitorar as chamadas executadas pelo mesmo e sugerindo modificações em parâmetros de desempenho, como limites de *timeout*

e latência. Por fim, o terceiro componente, FireDrill (DEVARAJ; LI, 2017), está relacionado a testes de infraestrutura, permitindo injetar falhas parciais, como latência de rede, falha de disco e falhas de DNS, ou falhas totais, como queda de conexão, desligamento de máquinas, e sobrecarga de memória.

Com a implementação deste projeto, o LinkedIn pôde iniciar mudanças culturais na empresa (DEVARAJ; LI, 2017), automatizando e melhorando a experiência de testes, além de obter maior confiança na capacidade de seus sistemas de se recuperar de falhas de forma automática, provendo mecanismos de "degradação elegante"[3].

4.2 GAMEDAYS

Uma das atividades mais difundidas na prática de *Chaos Engineering* é a execução de dinâmicas chamadas *GameDays*. Estas podem levar uma quantidade variável de tempo, tendo versões curtas, de não mais de 4 horas (HABER, 2019), até exemplos de empresas que levam dias na execução da atividade (ROBBINS et al., 2012). O exercício tem como foco simular um tipo de falha ou conjunto predefinido de falhas em um sistema num ambiente controlado. Sua origem remete à prática comum em forças de segurança de simular incidentes, em particular, às simulações de incêndios executadas por bombeiros (ROBBINS, 2011).

A realização destes *GameDays* é muito recomendada para o início da implantação de uma cultura de *Chaos Engineering* nas empresas (BUTOW; SALINAS, 2018), pois possibilita um maior engajamento das diversas partes que serão envolvidas nos experimentos futuros. Além do ganho no aspecto cultural, também permite experimentar como seria a rotina de diagnóstico de problemas e obtenção de informações sobre o sistema que a prática continuada de *Chaos Engineering* proporciona.

Mesmo em organizações que já têm uma maior maturidade com a prática de *Chaos Engineering*, ainda encontramos a prevalência da execução de *GameDays* como uma rotina (ROBBINS et al., 2012). Retomando a temática de simulações de incêndio, a ideia a longo prazo destes exercícios é de que eles sejam realizados com frequência e em escalas gradualmente maiores, visando a manter as equipes de desenvolvimento e operações com prática na resposta aos incidentes, mas em um ambiente controlado, o que permite a aplicação de experimentos mais ousados e que ponham à prova as ferramentas de monitoramento do *software*, o *hardware* e, sobretudo, as pessoas e processos envolvidos.

A estrutura e aplicação dos *GameDays* varia de acordo com os diversos contextos culturais, organizacionais e tecnológicos das empresas, mas ao revisar a literatura é possível extrair pontos em comum, que exploraremos nas seções seguintes.

[3] Do termo original *graceful degradation*, denota a capacidade de um sistema de operar em um estado de falha parcial, sem afetar drasticamente a funcionalidade para o usuário final.

4.2.1 Planejamento

Os eventos de *GameDay* necessitam de um planejamento e organização delicados. Por este motivo, se faz necessário ter uma equipe responsável por planejar e observar a atividade. No caso de empresas mais maduras, estas equipes são construídas especialmente com o propósito de fomentar e facilitar a implementação da cultura de *Chaos Engineering*. Em empresas menores, que estão dando seus primeiros passos nesta disciplina, o planejamento é geralmente executado por parte de uma equipe que deseja começar a testar a resiliência de seus sistemas de formas diferentes e se dispõe a tomar a frente da organização de um evento assim.

A fase de planejamento de um *GameDay* envolve a definição do alvo (ou alvos) dos testes. Este alvo pode ser o sistema como um todo, para testes mais simples e exploratórios, ou apenas um conjunto restrito de componentes envolvidos em um determinado fluxo lógico do *software*. A partir deste alvo, é possível fechar o escopo dos testes, detalhando **o que** falhará, **como** e por **quanto tempo**. Possibilidades de testes a executar em um *GameDay* envolvem simular os passos de uma falha conhecida para validar uma correção, garantir que o monitoramento de novos sistemas estão bem configurados para casos de falha mais delicados, entre outros testes de injeção de falhas que são comuns no escopo de *Chaos Engineering* (GREMLIN, 2018a). Em resumo, é importante que todas estas definições sejam feitas levando em consideração as seguintes perguntas (SAULITIS et al., 2017):

- "O que se quer aprender?"
- "Quais são os riscos?"
- "Qual o objetivo dos testes?"
- "Quais ações de mitigação existem em caso de problemas?"

Tendo estas questões respondidas, um grupo que se proponha a realizar um *GameDay* deve escolher uma data (ou conjunto de datas possíveis) para o evento, que permita às partes envolvidas um tempo hábil para planejamento próprio e aplicação de mitigações, caso necessárias (ROBBINS et al., 2012). É essencial que estas partes sejam mapeadas e notificadas com clareza sobre a simulação, e que meios de comunicação eficientes (aplicativos de mensagens, videoconferências, telefones, etc.) estejam estabelecidos entre todos os envolvidos e a equipe organizadora do *GameDay*, para que a execução do mesmo transcorra bem (GREMLIN, 2018a).

É necessário ressaltar, no entanto, que no contexto de organizações que estão dando seus primeiros passos com a disciplina de *Chaos Engineering*, as experiências mostram que

não é interessante começar com exercícios muito complexos e de grande escala. O intuito inicial é de construir confiança com as equipes técnicas e a gestão através de aprimoramentos incrementais e escalar as simulações gradualmente (BURNS, 2019) (BUTOW; SALINAS, 2018).

4.2.2 Execução

Apesar da divergência da literatura e das experiências individuais com relação aos detalhes da implementação de *GameDays*, um tópico invariante é a importância da comunicação constante durante o experimento. O objetivo final da prática de *GameDays* é, sobretudo, o de fortalecer a confiança da organização na resiliência de seus sistemas e processos (ROBBINS, 2011). Desta forma, todas as partes envolvidas — desenvolvedores, equipes de infraestrutura, gestores e até mesmo executivos — devem ter ciência do objetivo do *GameDay*, dos impactos esperados e das condições de parada. Também é importante garantir que todos os que vão participar ativamente do *GameDay* estejam alinhados com relação ao funcionamento e arquitetura do sistema envolvido, para que o diagnóstico das falhas e coleta de informações sobre efeitos indesejados seja efetivo (GREMLIN, 2018a).

Ao supervisionar a execução da atividade, os organizadores da mesma devem monitorar o sistema que está sofrendo a injeção de falhas e os sistemas dependentes que foram previamente mapeados (e estão fora do escopo dos testes), a fim de detectar efeitos colaterais e avaliar constantemente a necessidade de abortar o experimento (mais importante em *GameDays* executados no ambiente de produção) (GREMLIN, 2018a). No entanto, é necessário separar falhas normais de falhas inesperadas, afinal, a premissa do exercício é de injetar falhas. Seguindo a filosofia dos princípios de *Chaos Engineering* (CHAOS COMMUNITY, 2019), este limiar costuma se definir em torno de falhas que afetam as métricas de negócio, ou seja, que teriam impactos para clientes externos.

Para que o exercício seja produtivo, a equipe que está exercendo o papel de responder ao incidente deve observar as ferramentas de monitoramento e se comunicar com as outras equipes envolvidas para diagnosticar e mitigar o problema. A ideia é que a equipe organizadora vá aumentando a intensidade da falha gradualmente até que ela se faça visível nas métricas monitoradas e, a partir deste ponto, a equipe de resposta deve atuar de forma independente, identificando e documentando todas as ações de correção executadas (BURNS, 2019).

Durante todo o experimento, é necessário se atentar e manter um registro de informações e acontecimentos durante a simulação de cada falha programada para o experimento, visando a responder às seguintes perguntas (GREMLIN, 2018a):

- As informações disponíveis são suficientes para entender todos os impactos desta falha?

- O comportamento observado era o esperado durante o planejamento?
- Como esta falha se mostraria para um cliente externo?
- Como os sistemas que se comunicam com o alvo da falha estão se comportando diante destas condições?

Estes questionamentos servem como um guia para nortear a obtenção de informações do experimento de forma estruturada, o que facilitará muito o próximo passo de um *GameDay*: a reunião de retrospectiva (HABER, 2019), também chamada na indústria de "*postmortem*" (BURNS, 2019).

4.2.3 Reuniões "*Postmortem*"

Após a execução dos testes planejados, é necessário analisar as informações observadas e coletadas. Esta análise é o que dá fechamento ao processo de aprendizado que os *GameDays* têm como objetivo. Estas reuniões costumam ser conduzidas alguns dias após a execução do *GameDay* (GREMLIN, 2018a), para que os envolvidos tenham tempo suficiente para descansar do exercício, mas não o suficiente para esquecer os detalhes do que aconteceu e o que foi observado.

Para direcionar a produção de conhecimento, um dos objetivos da reunião pode ser o de responder perguntas como (HABER, 2019):

- O que a equipe aprendeu? O entendimento geral do sistema melhorou com o exercício?
- As soluções de monitoramento e alertas funcionaram como planejado?
- Quanto tempo foi necessário para o sistema se recuperar da falha (MTTR)? É possível melhorar esta métrica?
- É possível automatizar alguma das rotinas executadas pela equipe para acelerar a mitigação dos problemas?
- Foi descoberta alguma nova dependência ou componente do sistema que exija uma atualização dos diagramas de arquitetura?
- As equipes responsáveis pelos serviços dependentes foram alertadas com sucesso pelas ferramentas de monitoramento?
- É necessário refatorar algo na forma como estes serviços dependentes se conectam?

É necessário ressaltar que, apesar da relação direta com *GameDays* de um ponto de vista moderno, a prática de reuniões *postmortem* não é uma novidade no mundo do desenvolvimento de *software* (COLLIER; DEMARCO; FEAREY, 1996), já sendo comum e fundamental no processo de resposta a incidentes de maneira geral, principalmente em casos de falhas consideradas críticas (BEYER et al., 2016b). O objetivo final da reunião é criar um relatório ou qualquer forma de conhecimento documentado, para que uma falha seja compreendida e ações corretivas possam ser tomadas. A forma com que este objetivo é atingido varia amplamente de acordo com as diferentes empresas e equipes, mas as informações costumam ser levantadas através de *logs* dos sistemas, registros de métricas monitoradas, históricos de conversas em *chats* utilizados durante o incidente, relatos falados de profissionais que atuaram na detecção do incidente, entre outros (WOODS, 2017). Com base nestas informações, é traçado o esboço de uma linha do tempo dos acontecimentos durante o incidente e busca-se determinar uma causa-raiz para o ocorrido através da análise destes acontecimentos. A quantidade de pessoas envolvidas na reunião pode variar conforme a criticidade da falha (em escala menor para falhas simuladas, no caso de *GameDays*), e gira em torno de 5 a até 50 pessoas, dentre as quais vão figurar profissionais técnicos, envolvidos com os sistemas afetados, mas também podem incluir pessoal fora da área de TI, que tenham sido impactados pela falha ou que tenham algum interesse específico nas ações de mitigação (WOODS, 2017).

Após o processo da reunião, as conclusões tiradas e dados levantados serão organizados no formato de um relatório[4], cujas informações e formato variam de acordo com cada implementação. Este servirá como guia para o desenvolvimento de correções, além de (usualmente) ser disponibilizado para o resto da organização por meio de alguma forma de repositório central (COLLIER; DEMARCO; FEAREY, 1996), onde outras equipes possam consultar falhas passadas e, idealmente, suas respectivas ações corretivas.

Como forma de potencializar o compartilhamento de conhecimento e a velocidade do aprendizado por parte da organização como um todo, é possível, também, fazer eventos e grupos de discussão internos, com o objetivo de estudar relatórios de incidentes específicos, ou até mesmo simular incidentes de *postmortems* antigos como *GameDays*, como relatado pela Google em seu livro sobre práticas SRE[5] (BEYER et al., 2016b).

Retomando o contexto específico dos *GameDays*, é comum que uma falha que já foi experimentada em um exercício anterior, e apresentou problemas, tenha alguma ação preventiva feita sobre a mesma e seja simulada novamente. Nestas situações, espera-se, com mais confiança, que o problema tenha sido corrigido, e isto deve ser validado e analisado

[4] Na literatura, este relatório é muitas vezes também chamado de *postmortem*, em confusão com o nome da reunião (conforme evidenciado por Fernández (2021)). Por clareza, trataremos da reunião com o nome *postmortem* e faremos referência ao relatório de forma genérica.

[5] Sigla para "*Site Reliability Engineering*", se trata de um conjunto de práticas que aliam a engenharia de software à operação de sistemas e infraestrutura em produção.

na reunião *postmortem*. Ao constatar, através do experimento, que a falha foi corrigida, é interessante reconhecer o trabalho feito (GREMLIN, 2018a), fortalecendo a cultura ao evidenciar um impacto positivo da prática de *Chaos Engineering*. Além disto, visando a extrair o maior benefício para a resiliência, devem ser discutidas formas de automatizar a injeção deste modo de falha, para validar a correção de forma contínua e impedir a regressão do problema em novas versões do código, alinhando-se à visão de *Chaos Engineering* aplicado de forma automatizada e em produção (CHAOS COMMUNITY, 2019).

Nesta fase de um *GameDay*, também identificamos uma das maiores mudanças culturais que a disciplina de *Chaos Engineering* proporciona, quando inserida em uma cultura DevOps, com colaboração aberta e comunicação entre as equipes. Dentro desta cultura organizacional, os colaboradores passam a tratar falhas nos sistemas não com uma visão punitiva e reativa do que fazer **se** uma falha acontecer, mas sim de como se preparar para **quando** ela acontecer (JONES, 2017). Neste contexto, entram os *blameless postmortems*, *postmortems* sem atribuição de culpa (ALLSPAW, 2012). Esta prática, com origens em serviços de alto risco, como saúde e aviação (BEYER et al., 2016b), trata-se de uma mudança cultural que tem o objetivo de melhorar a comunicação através da transparência e, com isso, expor e combater as verdadeiras falhas nos processos e sistemas. Tal abordagem vem de um entendimento que falhas, sejam elas operacionais ou de projeto, são sintomas de um problema sistêmico, e que são situações, não pessoas, que causam erros (ALLSPAW, 2012) (PAGERDUTY, 2021).

O cultivo desta cultura demanda mudanças na forma como as informações são levantadas durante o processo de um *postmortem*. O foco da retrospectiva deve ser em levantar acontecimentos e a sequência de eventos que levaram a eles, sem passar por questões relacionadas a pessoas específicas e justificativas de suas ações ao lidar com o incidente (PAGERDUTY, 2021). Desta maneira, todo o processo parte da premissa de que todos os envolvidos fizeram o que pensaram ser correto no contexto em que estavam e quaisquer problemas apenas ocorreram porque os sistemas e procedimentos foram desenvolvidos de maneira a permitir que estes ocorressem (BEYER et al., 2016b). Esta perspectiva permite evitar vieses cognitivos[6] e faz com que os colaboradores possam relatar suas experiências de forma mais honesta, o que contribui para o aprendizado da equipe e o aprimoramento da organização.

[6] Termo usado pela área da psicologia para se referir a padrões de julgamento humano que levam a distorções da realidade ou interpretações desprovidas de lógica (WIKIPEDIA, 2019). No contexto que estamos abordando, a literatura faz referência especialmente ao viés de retrospecto (do original: *hindsight bias*), que consiste em analisar acontecimentos passados como sendo previsíveis, muitas vezes atribuindo o acontecimento de uma falha à distração humana em não prever algo que pareceria óbvio (WIKIPEDIA, 2021).

4.3 CONFERÊNCIAS E EVENTOS RELEVANTES

Por se tratar de uma área muito movida pela experimentação, comunidades de profissionais foram se formando em torno do tema, para trocar experiências e desenvolver ferramentas em conjunto. Partindo disto, surgem conferências e grupos de discussão, com foco em *Chaos Engineering*. Estes eventos são, em grande maioria, patrocinados e promovidos por empresas envolvidas direta e indiretamente no processo de evolução da área de *Chaos Engineering*, como Amazon, Netflix e Gremlin.

Exemplos notáveis destes eventos são a Chaos Conf (GREMLIN, 2020b), a maior conferência orientada apenas ao tema de *Chaos Engineering*, seguida pela Chaos Carnival (CHAOSNATIVE, 2022) e a conferência anual da AWS, a Re:Invent (AMAZON, 2022b), que tem trazido apresentações e *workshops* de *Chaos Engineering* de forma mais frequente em suas últimas edições. Nestes eventos, profissionais de diversas empresas apresentam soluções de *Chaos Engineering* que aplicaram em seus projetos, ou experiências que tiveram com a implementação do método e seus impactos culturais — temas que são muito relevantes para a adoção do método por outras equipes.

Outra fonte de conhecimento sobre o tema são os grupos de discussão *on-line*, que estão presentes em diversas plataformas. Existe uma lista de *e-mail* bastante ativa no Google chamada Chaos Community (KLEMENTIEV, 2015), onde muitos pesquisadores e profissionais da indústria interagem trocando recursos sobre a disciplina. Além desta, também surgiram grupos em plataformas como LinkedIn, Slack e GitHub, onde praticantes de *Chaos Engineering* podem relatar experiências e buscar ajuda no uso de ferramentas e na implementação em seus respectivos sistemas.

5 FERRAMENTAS E PLATAFORMAS PARA CHAOS ENGINEERING

A prática de *Chaos Engineering* foi surgindo e tomando forma gradualmente, movida pelas experiências individuais nas grandes empresas que tinham problemas de larga escala em seus sistemas distribuídos. Estas experiências, combinadas pelo compartilhamento das mesmas em conferências, encontros de desenvolvedores e outros eventos sociais onde os profissionais de tecnologia envolvidos nestas práticas se juntavam, possibilitaram a difusão da ideia e a criação de diversas perspectivas sobre práticas efetivas para a implementação do método de *Chaos Engineering*.

Desta forma, foi criada uma gama de ferramentas e plataformas para a implementação de *Chaos Engineering*, variando conforme a realidade de cada organização onde elas foram utilizadas. A lista encontrada em Ratis (2022), disponibilizada por meio da plataforma GitHub, é uma das que tentam compilar o universo de ferramentas existentes. Destas, temos ferramentas de código aberto, ferramentas internas de diversas empresas grandes (cujos relatos de sua existência e implementação costumam ser publicados nos respectivos *blogs* de tecnologia das empresas), além de plataformas na forma de produtos, ofertados com a proposta de facilitar a implementação de *Chaos Engineering* dentro de empresas que não tenham o interesse de investir tempo para construir ferramentas customizadas.

Ao longo das próximas seções, cobriremos, de forma não extensiva, algumas das ferramentas de destaque na literatura encontrada, a saber: Chaos Monkey, Gremlin, AWS FIS, Chaos Toolkit, Hystrix e Simmy. O critério de definição das ferramentas selecionadas foi puramente baseado na análise subjetiva da frequência com a qual referências eram encontradas nos textos estudados. Também houve uma predileção por ferramentas sobre as quais houvesse uma documentação mais extensa e guias mais detalhados, o que é realidade para ferramentas que tiveram maior utilização na comunidade, e, portanto, suscitaram interesse por conteúdo em maior quantidade de pessoas.

5.1 CHAOS MONKEY

Já mencionado anteriormente, o *Chaos Monkey* foi a primeira ferramenta de *Chaos Engineering* amplamente conhecida, e seu nome é frequentemente associado pela comunidade à prática desta metodologia. A ferramenta foi desenvolvida pela Netflix e é atualmente utilizada e suportada dentro da mesma e, por este motivo, é disponibilizada tendo como dependência direta outra ferramenta que foi desenvolvida na empresa, para gerenciamento de entrega contínua, chamada *Spinnaker* (THE LINUX FOUNDATION, 2022b).

Este segundo projeto também nasceu na Netflix e, atualmente, é mantido de forma independente e com código aberto por uma comunidade de desenvolvedores, dentre os quais

figuram muitos colaboradores de grandes empresas do mercado que utilizam a ferramenta em suas operações, como Netflix, Google e outras. Desta forma, a principal desvantagem no uso do *Chaos Monkey* vem da necessidade de utilizar o *Spinnaker* como gerenciador de entrega contínua das aplicações existentes (GREMLIN, 2022c), o que pode ser um fator limitante para equipes que já tenham uma ferramenta diferente implementada para esta finalidade.

O *Chaos Monkey* é uma ferramenta básica, no sentido de que tudo o que entrega é uma forma de automatizar a injeção de um único modo de falha nos sistemas, que é o de perda de nós de forma aleatória. Ele o faz através da interface que o *Spinnaker* provê com diversos provedores de nuvem como AWS, GCP e Azure, programando o término de instâncias de computação ou contêineres (no caso da integração com *Kubernetes*) em intervalos aleatórios, dentro de uma faixa limite configurável. Este limite pode ser configurado em termos do tempo médio entre falhas e do tempo mínimo entre falhas, para garantir que as falhas sejam tão frequentes quanto desejado (NETFLIX, 2019b). É possível, também, selecionar a granularidade com a qual o *Chaos Monkey* irá inserir falhas, de forma a limitar seu uso apenas para uma aplicação específica, para um grupo de recursos de nuvem, ou para um *cluster* por inteiro, onde a ferramenta irá, na frequência configurada, terminar instâncias periodicamente.

Por sua simplicidade, o uso do *Chaos Monkey* é mais recomendado para passos iniciais em *Chaos Engineering*, ou para organizações que já estejam usando o *Spinnaker* e/ou não tenham problemas em implementar seu uso. Adicionalmente, o *Chaos Monkey*, por si só, não apresenta muitas capacidades extras de segurança, ficando a cargo do usuário implementar mecanismos de controle para interromper experimentos e integrações para monitoramento da saúde dos sistemas (GREMLIN, 2022c). Por ser uma ferramenta antiga e com muito tempo de uso, tem uma documentação bastante extensa (NETFLIX, 2020) (NETFLIX, 2019b), bem como uma grande riqueza de conteúdo *online* produzido sobre ela, na forma de guias de implementação e artigos que falam sobre seu funcionamento (RATIS, 2022) (GREMLIN, 2018c).

5.2 GREMLIN

Gremlin consiste numa plataforma para automação de experimentos de *Chaos Engineering*, criada e oferecida como serviço pela empresa de mesmo nome. O produto teve sua origem em 2016, criado em parte por um ex-funcionário da Netflix e Amazon, Kolton Andrus, que foi responsável pelo desenvolvimento de soluções de *Chaos Engineering* em ambas as empresas (BUTOW, 2021) (GREMLIN, 2022a). A plataforma foi criada com o propósito de ser um produto de "*Failure as a Service*" (como uma adaptação do termo existente "*Software as a Service*"), que permite a configuração simplificada de diversos

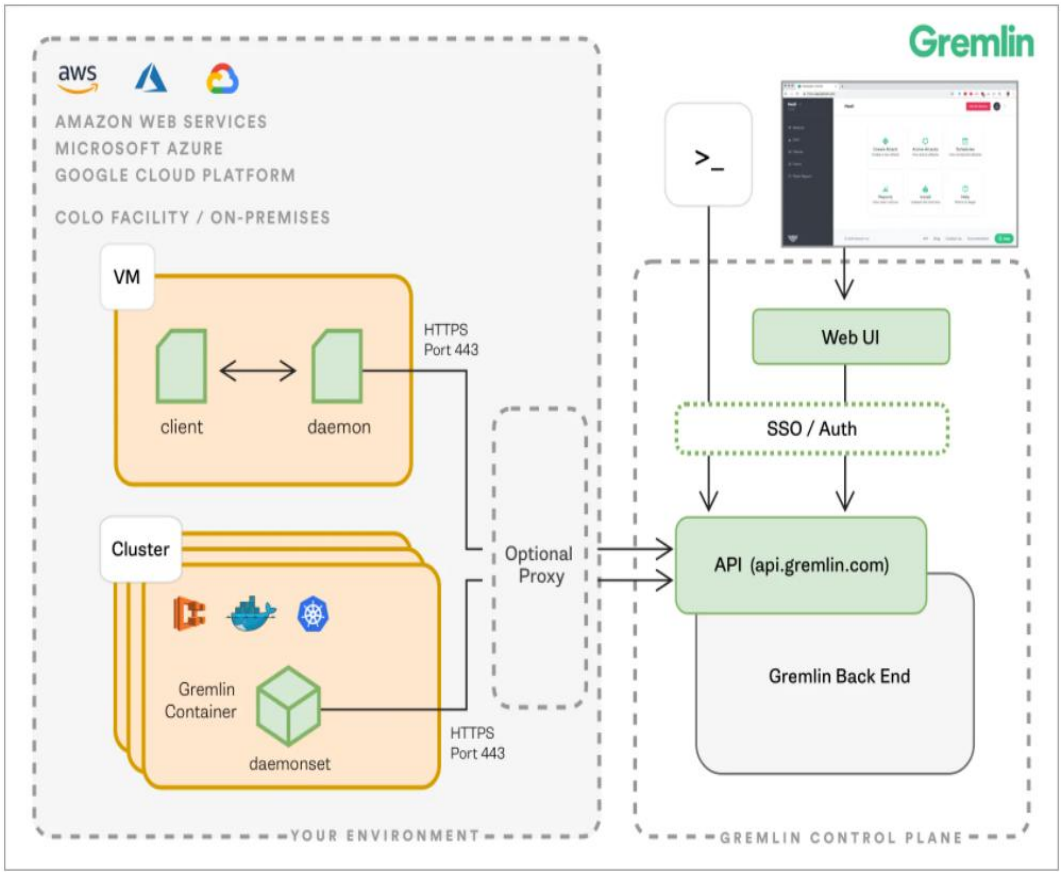
tipos de falha e sobre uma grande variedade de infraestruturas, tendo compatibilidade com os principais provedores de nuvem da atualidade.

A empresa, hoje, é a maior referência do mercado em *Chaos Engineering*, tendo liderado a comunidade, nos últimos anos, em torno de atividades relacionadas à disciplina, como as conferências Chaos Conf (GREMLIN, 2020a) e Failover Conf (GREMLIN, 2021a), assim como grupos de discussão online (GREMLIN, 2022b). Adicionalmente, a Gremlin também tem, em sua página da Web, uma abundância de conteúdos sobre *Chaos Engineering* em geral e guias de como implementar a prática utilizando seu produto. Além da plataforma de *Failure as a Service*, a companhia também provê treinamentos para indivíduos e empresas na aplicação de *Chaos Engineering* e, recentemente, criaram uma certificação (GREMLIN, 2022d) para profissionais que desejam se especializar na disciplina de *Chaos Engineering*.

O funcionamento da ferramenta se dá por meio de um agente, na forma de um *daemon*, que é um arquivo binário a ser instalado nas máquinas ou contêineres onde se deseja executar experimentos. Quando o mesmo está instalado, a máquina passa a se comunicar periodicamente com os servidores da Gremlin (através de uma conexão protegida por TLS) para divulgar seu status de saúde. Torna-se possível, também, a realização de ações remotas no nó a partir da interface *web* da plataforma ou via API REST. O agente do Gremlin é compativel com sistemas operacionais Linux e Windows, bem como ambientes de orquestração de contêineres, como o *Kubernetes* (THE LINUX FOUNDATION, 2022a). A Figura 7 mostra um diagrama da arquitetura de um ambiente integrado ao Gremlin, com os agentes instalados nas máquinas (VM) ou nós do *cluster* em um ambiente de nuvem ou infraestrutura própria, e a comunicação destes com os servidores externos da Gremlin, de onde as ações de controle são emitidas, a partir de comandos via interface *web*, utilitário de terminal ou chamadas de API.

O Gremlin tem como foco a segurança dos experimentos, através de mecanismos de interrupção dos mesmos em caso de perda de controle dos efeitos. Além de um botão para interromper manualmente qualquer ataque em execução, quando um ataque está em curso, o agente que o executa aumenta a frequência de comunicação com o servidor de controle, e, caso esta comunicação falhe e gere um erro, o próprio agente interrompe o ataque de maneira independente (GREMLIN, 2022g). No contexto de experimentos agendados e executados de forma automática, também é possível configurar condições para interrupção do experimento, extraídas por meio de integrações com ferramentas externas de monitoramento, ou por alguma resposta esperada da aplicação (por meio de requisição HTTP). Estas condições serão avaliadas ao início de um conjunto de ataques — garantindo que o estado do sistema é saudável antes de injetar falhas — bem como durante a execução dos mesmos e ao final, para verificar que o sistema retornou a um estado estável.

Figura 7 – Diagrama de arquitetura da ferramenta Gremlin



Fonte: Gremlin (2022e)

Em complemento à segurança nos experimentos, a ferramenta também conta com uma solução robusta de segurança no acesso à infraestrutura e rastreabilidade dos ataques, provendo diversas formas de acesso autenticado, por meio de integrações com ferramentas de autenticação por multi fatores e federação de credenciais de autenticação. Além disso, o uso das interfaces de controle pode ser administrado por perfis de acesso e a execução de ataques é rastreável por meio de *logs* de auditoria (GREMLIN, 2022i).

A plataforma tem um conjunto extenso de tipos de ataque, e permite selecionar alvos para os mesmos de acordo com diferentes níveis de granularidade, agrupando recursos por *hosts*, rótulos, tipo de instância, faixa de IP e outros (GREMLIN, 2022h). Os ataques podem ser executados de forma manual, monitorando sua execução pela interface, ou de forma agendada, onde é possível configurar dias e períodos específicos para os experimentos. Os modelos de ataque da plataforma podem ser divididos em 3 categorias, a saber (GREMLIN, 2022f):

- Ataques em **recursos**:
  - Sobrecarga de CPU;

  - Sobrecarga de memória;
  - Sobrecarga de operações de E/S;
  - Consumo de espaço de armazenamento;

- Ataques de **estado**:
  - Desligamento de nó;
  - Mudança na configuração de tempo do sistema;
  - Interrupção de processos;

- Ataques de **rede**:
  - Particionamento de rede;
  - Injeção de latência no tráfego de saída;
  - Perda aleatória de pacotes no tráfego de saída;
  - Disrupção de consultas DNS.

Por meio da interface *web*, é possível visualizar a execução dos ataques, observando métricas de hardware pertinentes ao ataque específico. No entanto, para o monitoramento de métricas relevantes ao negócio e outras métricas de interesse (previamente mapeadas, conforme preconiza a disciplina de *Chaos Engineering*), é recomendado utilizar outras ferramentas de monitoramento disponíveis no mercado e integrar o Gremlin a elas, permitindo visualizar eventos de ataques através das mesmas e utilizar métricas customizadas para as verificações de saúde feitas pelo Gremlin.

Por ser um serviço, o Gremlin conta uma uma desvantagem óbvia que é seu modelo de precificação. Apesar de ter uma versão gratuita, esta é limitada em termos da quantidade de ataques que podem ser executados e dos recursos de segurança e rastreabilidade, em relação à versão por assinatura (GREMLIN, 2022j). Outra restrição que pode ser um fator limitante a alguns casos de uso é a impossibilidade de implementar e automatizar ataques personalizados, permitindo apenas o uso dos modelos já implementados na ferramenta, com seus respectivos parâmetros personalizáveis.

5.3 AWS FAULT INJECTION SIMULATOR

Anunciado em março de 2021 pela Amazon, o *Fault Injection Simulator* (FIS) é uma plataforma que o provedor de nuvem oferece para seus usuários, onde é possível criar experimentos de *Chaos Engineering* por meio de uma interface simples e com integração completa com os principais serviços e recursos de infraestrutura da AWS (AMAZON, 2021).

A plataforma tem como objetivo prover uma ampla gama de modelos de falha, que podem ser aplicados sobre recursos da AWS, como máquinas virtuais, bancos de dados, *clusters* de contêineres, e aplicações hospedadas em outros serviços especializados. Dentre os possíveis modos de falha, destacam-se (AMAZON, 2022a):

- Falhas de API (por meio de códigos de erro HTTP);
- Interrupção de máquina virtual;
- Interrupção de contêineres gerenciados;
- Falha de servidor primário de banco de dados;
- Estresse de CPU, E/S e memória;
- Interrupção de processos em máquinas;
- Falha de comunicação por rede;
- Aumento de latência e perda de pacotes.

Os experimentos são configurados por meio da interface *web* de gerenciamento da AWS, ou por meio de arquivos de texto nos formatos JSON[1] ou YAML[2], para os quais a Amazon provê um formato padrão. Esta possibilidade faz com que os experimentos possam ser compartilhados entre equipes com facilidade, e também permite que os mesmos sejam controlados por meio de alguma solução de versionamento de código, como Git, SVN, Mercurial, e outras (HORNSBY, 2020).

A configuração de cada uma das falhas do experimento conta com diversos parâmetros, como: duração do ataque, conjunto de alvos (recursos de nuvem que sofrerão as falhas), pré-condições e parâmetros específicos de cada falha, como percentual de requisições afetadas por uma falha de API, por exemplo. É possível selecionar os alvos do experimento de forma específica ou aleatória dentro de um grupo de recursos, que pode ser definido por meio de rótulos (*tags*) — uma ferramenta de uso comum no ambiente da AWS. O início de cada ataque pode ser configurado para acontecer de forma sequencial (simulando uma falha gradual da infraestrutura) ou de forma paralela, caso haja o desejo de testar a resiliência a uma falha generalizada e crítica (HORNSBY, 2020). No Código 1, presente nos itens anexos, apresentamos um exemplo de um *template* de experimento, que configura

[1] Acrônimo para *JavaScript Object Notation*, é um formato para transmissão e representação de dados estruturados via texto. É popular no desenvolvimento de API's HTTP por sua legibilidade por humanos e fácil interpretação por máquinas (ECMA INTERNATIONAL, 2022).

[2] Acrônimo recursivo para *YAML Ain't Markup Language*, também é um formato para transmissão e representação de dados estruturados. Diferencia-se do JSON por sua sintaxe mais focada na legibilidade por humanos, fazendo com que seu uso seja mais popular em arquivos de configuração e armazenamento de dados estruturados (WIKIPEDIA, 2022).

o FIS para atuar sobre 3 máquinas virtuais que contenham o rótulo "prod" (descrito na seção "*targets*" do arquivo), interrompendo-as por 2 minutos (descrito na seção "*actions*").

A execução dos experimentos é feita por interface de linha de comando ou por meio da interface *web*, por onde é possível iniciar e acompanhar o progresso dos mesmos. Este acompanhamento é proporcionado tanto pela plataforma FIS, quanto por sua integração com a solução de *logs* e monitoramento da AWS chamada *CloudWatch*, por meio da qual é possível observar métricas dos sistemas e configurar alertas de segurança que podem ser usados para interromper experimentos do FIS de forma automática, caso algum limite seja violado (HORNSBY, 2020). No entanto, este monitoramento de métricas e configuração de gatilhos com base em experimentos também podem ser feitos por meio de outras plataformas de monitoramento, por meio da integração do FIS com o serviço de eventos da AWS, o *EventBridge*. Desta forma, torna-se possível observar métricas mais complexas e próximas das necessidades de negócio, além de permitir aos usuários usufruirem de outras ferramentas de monitoramento presentes no mercado.

Dentre as desvantagens no uso da plataforma FIS, figuram a incapacidade, até o momento, de configurar experimentos para serem executados automaticamente com periodicidade (conforme os princípios de *Chaos Engineering*), assim como a especificidade da plataforma aos produtos AWS. Isto faz com que não seja possível portar os experimentos e configurações para fora do ambiente do provedor de maneira fácil, o que poderia gerar problemas para equipes que queiram mover sua infraestrutura para fora da AWS.

5.4 CHAOS TOOLKIT

Criado para prover versatilidade em experimentos de *Chaos Engineering*, o *Chaos Toolkit* (CHAOS TOOLKIT, 2022a) é um utilitário de linha de comando de código aberto, desenvolvido com a linguagem Python, que provê um conjunto de funcionalidades simples para a especificação e execução de experimentos em diversos contextos de infraestrutura. Esta flexibilidade é suportada por meio de um grande conjunto de extensões, permitindo integrar a ferramenta com diversos provedores de nuvem, ferramentas de monitoramento e outras aplicações. Sua extensibilidade se dá graças a sua documentação e especificação de interface, que permite que desenvolvedores independentes possam criar extensões que se integram à ferramenta de forma fácil e implementam funcionalidades comuns.

O uso da ferramenta é feito apenas através de interface de linha de comando, e os experimentos são configurados por meio de arquivos YAML ou JSON, em formato especificado na documentação do *Chaos Toolkit* (CHAOS TOOLKIT, 2022b). Na especificação, são definidas três seções importantes para a descrição completa de um experimento seguro: a declaração da hipótese de estado de equilíbrio (*steady-state-hypothesis*), a declaração dos métodos de injeção de falhas a serem utilizados (*method*) e uma seção onde podem ser

declaradas funções para reverter quaisquer alterações de estado feitas pelo experimento (*rollback*). No Anexo B, adaptamos um exemplo retirado da documentação da ferramenta, onde é demonstrado o formato de declaração das seções, em um experimento fictício de validação do impacto da expiração de um certificado SSL em uma aplicação. Separamos as três seções em códigos diferentes, onde podemos ver que são declarados objetos do tipo "*action*" ou "*probe*". As *actions* são comandos a executar no sistema que está sendo alvo do experimento e os *probes* são formas de observar a condição atual do sistema, por meio de algum comando a ser executado na máquina, chamada HTTP ou até mesmo alguma interação com a API de um provedor de nuvem. No exemplo dado, a seção de *steady-state-hypothesis* (Código 2) valida se a aplicação está sendo executada e pode ser acessada com sucesso via HTTPS. Em seguida, na seção *method* (Código 3), as ações de injeção de falha são declaradas, descrevendo comandos para modificar o certificado da aplicação por um expirado e forçar o carregamento do novo certificado pelo servidor. Por último, a seção *rollback* (Código 4) desfaz as ações de injeção de falha, restaurando o certificado correto e reiniciando o servidor novamente.

Em complemento à definição dos experimentos, o CLI (*Command Line Interface*) da ferramenta conta com argumentos de execução que permitem definir parâmetros do experimento, como a estratégia de recuperação, que define em que situações o programa irá executar as ações definidas na seção *rollback* do arquivo de definição. Outros parâmetros configuráveis são a frequência e a sensibilidade da validação de hipóteses, que definirá a periodicidade na qual a ferramenta executa as ações de verificação do estado de equilíbrio e, caso haja desvio, se uma única ocorrência do mesmo deve interromper ou não o experimento (CHAOS TOOLKIT, 2022c). Desta maneira, o *Chaos Toolkit* provê uma estrutura para a declaração de experimentos de *Chaos Engineering* na forma de código, além de um sistema básico para orquestrar a execução dos mesmos. É importante ressaltar que, ao contrário das demais ferramentas apresentadas neste texto, o *Chaos Toolkit*, por si só, não dispõe de nenhum tipo de ataque ou estratégia de falha implementada. Todas as ações executadas pela ferramenta utilizam chamadas remotas ou outros utilitários de linhas de comando para interagir com o sistema sendo testado.

O uso do *Chaos Toolkit* se mostra muito flexível, porém, por este mesmo motivo, parte da suposição que o desenvolvedor que está implementando a ferramenta irá construir utilitários de suporte ou utilizar outras ferramentas *open-source* para executar os ataques específicos. Além disso, também será necessário tomar os cuidados apropriados com o monitoramento, agendamento de experimentos automáticos e relatórios de auditoria por meio de soluções externas.

5.5 BIBLIOTECAS DE CÓDIGO

Nas seções anteriores, discutimos ferramentas que atuam, principalmente, com falhas em camadas mais baixas da infraestrutura, como falhas de rede, memória, CPU e outras. Nesta seção, abordaremos algumas bibliotecas de programação relevantes, que permitem atuar na camada da aplicação, ou seja, injetar falhas de forma controlada dentro do código da aplicação em tempo de execução. Estas bibliotecas usualmente incluem, também, módulos e algoritmos utilizados na mitigação das falhas, implementando padrões de desenvolvimento de *software* orientados à resiliência de sistemas distribuídos, que listamos brevemente na subseção 2.1.3.1.

5.5.1 Polly e Simmy

Estas duas bibliotecas são de código aberto e disponíveis na forma de pacotes para a linguagem C#. A biblioteca *Polly* (THE POLLY PROJECT, 2022a) foi criada primeiro, com o propósito de prover implementações robustas dos padrões de resiliência apresentados para a plataforma .NET. Ela o faz por meio de políticas ("*policies*"), que são objetos que encapsulam um dos padrões de resiliência suportados, e permitem a composição dos mesmos, por meio da aglutinação de políticas.

A biblioteca *Polly* provê uma API consistente, onde cada política pode ter um conjunto de exceções que irá tratar e aceita uma coleção de parâmetros, dentre os quais se inclui, obrigatoriamente, uma função a ser encapsulada pela política, que representa a chamada remota (ou outra ação qualquer) que se deseja executar de forma resiliente. Toda a documentação da biblioteca, assim como seu código-fonte e exemplos de uso, estão presentes em seu repositório principal no GitHub (THE POLLY PROJECT, 2022a).

Em extensão à *Polly*, foi lançada em 2019 a biblioteca *Simmy* (THE POLLY PROJECT, 2022b), que visa a permitir a injeção de falhas no nível da aplicação de maneira mais fácil. A *Simmy* também opera por meio de políticas, chamadas "*Monkey Policies*", que se integram ao fluxo de trabalho da *Polly* e permitem compor as políticas de tratamento com as de injeção de falhas, possibilitando validar seu funcionamento correto de forma simples para o desenvolvedor. As injeções de falha do *Simmy* funcionam de forma probabilística, por meio de um parâmetro que determina a probabilidade de aplicar uma falha a cada chamada da função. Além disso, naturalmente, também é possível desabilitar a injeção de uma determinada falha por completo por meio de um parâmetro booleano. As falhas injetáveis através da biblioteca se manifestam de quatro maneiras: i) lançamento de exceções ao executar uma chamada; ii) substituição do valor de retorno por um valor padrão configurável; iii) injeção de latência na execução da chamada; iv) qualquer outro comportamento definido pelo usuário, por meio de uma função de *callback*.

Uma restrição importante a ressaltar é a de que os parâmetros de injeção de falha são todos configurados em memória. Desta forma, é necessária uma solução por parte do desenvolvedor para armazenar estes parâmetros de forma organizada e facilmente modificável. É possível fazê-lo na forma de um arquivo de configuração, ou criando uma interface com um banco de dados ou qualquer outra forma de armazenamento dinâmico, através da qual o código com o *Simmy* possa carregar os parâmetros em tempo de execução. Esta solução seria mais robusta na perspectiva de uma aplicação de *Chaos Engineering* em produção.

5.5.2 Hystrix

Criada pela Netflix em 2011, a biblioteca *Hystrix* tem foco em facilitar a implementação de padrões de resiliência e o monitoramento e controle de interações entre serviços no contexto de aplicações Java. A biblioteca tem seu código aberto e disponível no GitHub, porém, atualmente, se encontra em versão de manutenção, sem aceitar novas contribuições para o código. Isto se dá, de acordo com a nota escrita pela equipe mantenedora do repositório (NETFLIX, 2021a), devido ao fato de que a biblioteca, em seu estado atual, já atende a todas as necessidades que a Netflix tinha quando imaginou sua criação. Desta forma, a *Hystrix* permanece disponível para uso e é ativamente utilizada de forma interna pela Netflix, mas os mesmos recomendam o uso da biblioteca *resilience4j* (WINKLER et al., 2022) para novos projetos, por ter uma grande inspiração na *Hystrix*, mas com uma estrutura e recursos mais modernos da linguagem Java. Nesta seção, abordaremos em mais detalhes a *Hystrix* por motivos históricos e de disponibilidade de conteúdo.

A *Hystrix* implementa suas soluções de resiliência por meio de duas classes que encapsulam comandos, com o intuito de que cada comando seja mapeado para uma chamada remota. A biblioteca classifica estes comandos em dois: comandos simples e comandos "observáveis" (NETFLIX, 2017c). Os comandos simples podem ser executados de maneira síncrona ou assíncrona, enquanto os comandos observáveis implementam o padrão *Observable* e são inerentemente assíncronos, podendo retornar uma sequência de resultados em diferentes momentos ao longo do tempo.

Ao contrário da modularidade presente na biblioteca *Polly*, apresentada na seção anterior, os comandos da *Hystrix* combinam, por padrão, todos os padrões de resiliência relevantes que citamos, restando ao desenvolvedor implementar as funções que determinam as ações a serem feitas para os casos de *fallback* e *caching*, por exemplo, bem como os parâmetros e limites aceitáveis para *timeouts* e *retries* (NETFLIX, 2017b). Por padrão, os comandos executados pelo *Hystrix* são isolados em *thread pools* baseados em chaves ou por meio de semáforos para limitação de concorrência, de forma que a falha excessiva dos comandos com uma determinada chave (supostamente mapeada para as chamadas de um único serviço remoto) não sequestre os recursos utilizados para outras chamadas (padrão

Bulkhead). Analogamente, os comandos também fazem uso de um contador de falhas, que pode ser divulgado para ferramentas de monitoramento e serve para implementar *circuit breakers*, que guardam todas as execuções de comandos para que a falha de uma dependência não cause um enfileiramento excessivo de requisições no serviço (NETFLIX, 2017b).

A configuração dos diversos parâmetros de cada algoritmo foi feita, originalmente, por meio de um projeto de gerenciamento dinâmico de parâmetros desenvolvido também pela Netflix, chamado *Archaius* (NETFLIX, 2019a). Este projeto também está arquivado em modo de manutenção. Por este motivo, o *Hystrix* também permite a reimplementação de sua estratégia de obtenção de parâmetros, para possibilitar a utilização de outras soluções de gerenciamento de configuração.

Por fim, ressaltamos que a *Hystrix* não se apresenta como uma biblioteca com um propósito específico de *Chaos Engineering*, mas sim como uma ferramenta útil para a implementação de sistemas resilientes e para a observabilidade de falhas; fatores que apresentam grande sinergia com a proposta de *Chaos Engineering*. A biblioteca também foi um trabalho pioneiro que inspirou o desenvolvimento pela comunidade da *Polly*, *resilience4j* e outras bibliotecas de código aberto orientadas a resiliência.

5.6 DISCUSSÃO

As ferramentas aqui apresentadas representam apenas uma fração do universo de implementações produzidas, na forma de código aberto ou proprietário, para dar suporte à disciplina de *Chaos Engineering*. As listas presentes em Ratis (2022) e Novegil (2022) são listas colaborativas que buscam catalogar parte do conteúdo existente na *web* sobre *Chaos Engineering*.

A realidade é que a quantidade de ferramentas aumenta conforme a popularidade da disciplina aumenta e, por consequência, a quantidade de praticantes. No entanto, muitas destas ferramentas carregam em si detalhes de implementação específicos ao caso de uso dos desenvolvedores da mesma, e buscar por uma ferramenta que se encaixe perfeitamente em um caso de uso pessoal pode ser uma tarefa árdua. Neste cenário, ferramentas mais genéricas e polidas, com gerenciamento de terceiros, parecem ser uma alternativa interessante para o contexto de implementações robustas em produção, ainda que isso possa demandar um custo maior com licenças e contratos de suporte.

Em geral, o conjunto de ferramentas disponíveis de forma aberta provê um excelente ponto de partida para a implementação de soluções próprias, se o caso de uso não for atendido bem por nenhuma das disponíveis. O fato de terem o código aberto facilita a extensão ou adaptação das ferramentas, possivelmente melhorando o cenário colaborativo da área de *Chaos Engineering*.

6 CONCLUSÃO

Nos últimos anos, o tema de *Chaos Engineering* ganhou grande relevância, como sugerido por estudos de mercado recentes InfoQ (2019) e Gremlin (2021b). Este crescimento se justifica, dentre outros fatores, pelo aumento de complexidade dos sistemas de informação, que precisam atender a um número cada vez mais elevado de usuários e em cenários mais críticos. Para aumentar a confiança nestes sistemas, a indústria sempre buscou por métodos de teste para validar o funcionamento dos mesmos. *Chaos Engineering*, no entanto, se mostra como um método capaz de revelar problemas em pontos não previstos e até já testados por outros métodos, pois seu objetivo consiste em encontrar falhas no funcionamento de sistemas como um todo, colocando seus componentes e interações internas à prova.

O presente trabalho forneceu ao leitor uma visão sobre a disciplina de *Chaos Engineering*, em múltiplas perspectivas. Inicialmente, relata-se a história da origem da área e princípios fundamentais, no contexto de um grande crescimento de sistemas distribuídos na expansão da Internet. Em seguida, detalha-se casos de implementação e práticas aplicadas em grandes empresas do mercado de tecnologia atual. Por fim, é apresentado, em detalhes, um recorte de uma lista de ferramentas e recursos úteis (RATIS, 2022) para uma possível implementação de *Chaos Engineering*, de forma a dar insumos iniciais para o leitor interessado em experimentar o método.

6.1 DESAFIOS

A adoção de *Chaos Engineering* traz consigo alguns desafios relevantes. Em organizações que não têm experiência com esta metodologia, a primeira barreira consiste na insegurança envolvida em executar experimentos que provocam falhas propositais em sistemas críticos. A superação deste impedimento depende de uma mudança cultural gradual e profunda nas equipes de desenvolvimento e operação dos sistemas (TUCKER et al., 2018). Além desta, existem dificuldades mais objetivas, como: (i) a escolha de quais experimentos priorizar e em que parte do sistema aplicá-los, (ii) implementar mecanismos de monitoramento eficientes para detectar problemas, (iii) definir um estado de equilíbrio que represente bem o comportamento saudável do sistema, (iv) a escolha e desenvolvimento de ferramentas seguras para executar os testes.

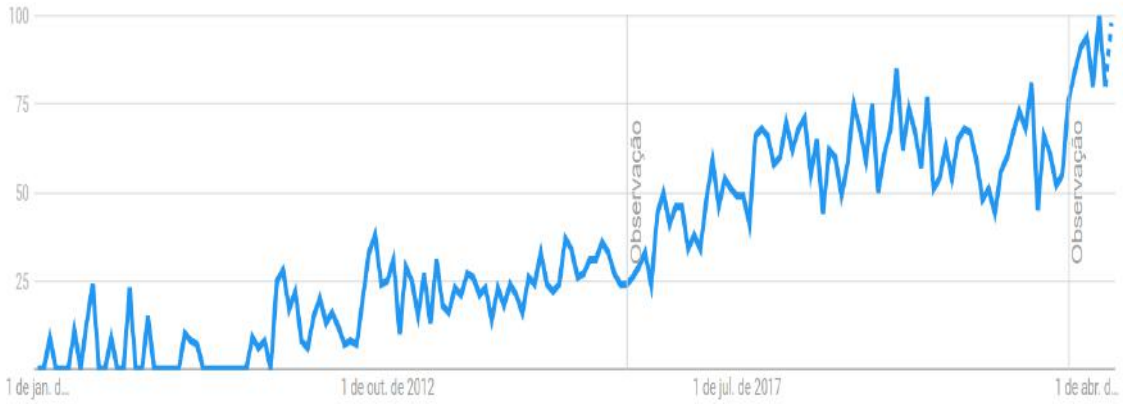
Ademais, a própria pesquisa sobre o tema também impõe desafios particulares, por conta da difusão do conteúdo sobre *Chaos Engineering* em *blogs* de tecnologia, artigos *online* de empresas, entre outros. Em decorrência desta diversidade, a tarefa de conectar as diferentes informações sobre práticas e sobre a evolução da disciplina torna-se um trabalho

dispendioso, auxiliado principalmente pelos conteúdos publicados pela equipe de *Chaos Engineering* da Netflix e por artigos e trabalhos científicos desenvolvidos em torno do tema, como os presentes em Rosenthal et al. (2017), Rosenthal et al. (2016), Basiri et al. (2019), Alvaro et al. (2016) e Jernberg (2020).

6.2 EXPECTATIVAS FUTURAS

Apesar dos desafios, de acordo com o último levantamento feito pela Gremlin (2021b), 60% das mais de 400 empresas analisadas já fizeram algum experimento de *Chaos Engineering* e 34% delas aplicam alguma forma de injeção controlada de falhas em produção. É possível perceber uma tendência crescente, tanto na adoção, quanto na discussão das ferramentas e métodos. É visível, também, um aumento nas buscas feitas pelo termo em motores de busca, nos últimos anos, como sugere o histórico do Google Trends, exibido na Figura 8, que utiliza uma métrica de relevância de 0 a 100, proporcional à frequência de pesquisas feitas pelo termo na janela de amostragem avaliada. A despeito da escassez de material acadêmico em língua portuguesa sobre o tema, artigos *on-line* e sites brasileiros que abordam o tema e falam sobre ferramentas têm se tornado mais frequentes.

Figura 8 – Tendência de popularidade das pesquisas mundiais pelo termo "chaos engineering" no Google, no período de 2008 a 2022



Fonte: Elaborado pelo autor[1]

Como indicado por InfoQ (2019), *Chaos Engineering* se encontra numa fase de "adoção recente" significando que empresas e profissionais com maior disposição para a inovação, estão demonstrando interesse pelo tema e passando a adotar práticas em diferentes níveis dentro de suas equipes. A proposta da metodologia de *Chaos Engineering* ainda encontra

[1] Resultado de pesquisa pelo termo "chaos engineering" na plataforma Google Trends.

resistência, principalmente, em níveis executivos mais elevados das organizações, mas a pressão exercida pela mídia — divulgando falhas em grandes sistemas de forma cada vez mais frequente — tende a fazer com que a implementação de rotinas mais rígidas de testes seja considerada. Esta necessidade se faz ainda mais evidente no que se refere a áreas da indústria de tecnologia com operações críticas, como os setores financeiro, varejo digital e serviços de saúde, onde interrupções de serviço podem ter alto impacto econômico e humano (ROSENTHAL et al., 2017).

A área de *Chaos Engineering*, como uma disciplina incipiente e nascida de forma majoritariamente distribuída e desestruturada, tem ainda um amplo conjunto de problemas não solucionados. Um dos problemas em aberto, que surge ao começar a estudar uma implementação prática, é o questionamento de por onde começar os testes em um sistema. Diversas abordagens existem, tais como priorizar testes de menor custo para implementação (TUCKER et al., 2018), ou a priorização de componentes críticos para o resultado final do sistema. Todas estas abordagens são, no limite, heurísticas que profissionais podem seguir ao tomar decisões de implementação de *Chaos Engineering*. No entanto, há trabalhos que buscam formalizar este processo de escolha, a princípio caótico, por meio de métodos formais e matemáticos, como os encontrados em Alvaro et al. (2016), Aleti et al. (2018) e Frank et al. (2020).

Outros problemas importantes, que surgem diretamente ao analisar os princípios de *Chaos Engineering* (CHAOS COMMUNITY, 2019), são o desenvolvimento de mecanismos de coleta e detecção de um comportamento padrão nos dados de monitoramento de sistemas, e métodos mais eficientes para tornar o processo de experimentação mais seguro em produção, afim de limitar impactos colaterais e riscos que freiam a maior adesão do método por parte da indústria. Futuramente, esperamos ver mais trabalhos que visem a sistematizar estes processos mais subjetivos, que estão envolvidos no cerne da disciplina de *Chaos Engineering*.

REFERÊNCIAS

ALETI, A. et al. Orcas: Efficient resilience benchmarking of microservice architectures. In: **2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)**. Memphis, TN, USA: [s.n.], 2018. p. 146–147.

ALLSPAW, J. **Blameless PostMortems and a Just Culture**. Etsy, 2012. Disponível em: https://codeascraft.com/2012/05/22/blameless-postmortems/. Acesso em: 10 fev. 2022.

ALVARO, P. et al. Automating failure testing research at internet scale. In: **Proceedings of the Seventh ACM Symposium on Cloud Computing**. New York, NY, USA: Association for Computing Machinery, 2016. (SoCC '16), p. 17–28.

AMAZON. **Announcing General Availability of AWS Fault Injection Simulator, a fully managed service to run controlled experiments**. 2021. Disponível em: https://aws.amazon.com/about-aws/whats-new/2021/03/aws-announces-service-aws-fault-injection-simulator/. Acesso em: 06 mar. 2022.

AMAZON. **Actions for AWS FIS**. 2022. Disponível em: https://docs.aws.amazon.com/fis/latest/userguide/actions.html. Acesso em: 06 mar. 2022.

AMAZON. **Homepage AWS re:Invent**. 2022. Disponível em: https://reinvent.awsevents.com/. Acesso em: 27 jun. 2022.

AWS. **Regions and Availability Zones**. 2022. Disponível em: https://aws.amazon.com/about-aws/global-infrastructure/regions_az/. Acesso em: 31 jul. 2022.

BASIRI, A. et al. Automating chaos experiments in production. In: **2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)**. Montreal, QC, Canada: IEEE, 2019. p. 31–40. Disponível em: https://doi.org/10.1109/ICSE-SEIP.2019.00012.

BEYER, B. et al. Monitoring distributed systems. In: BEYER, B. (Ed.). **Site Reliability Engineering: How Google Runs Production Systems**. O'Reilly Media, 2016. Disponível em: https://sre.google/sre-book/monitoring-distributed-systems/. Acesso em: 27 jul. 2022.

BEYER, B. et al. Postmortem culture: Learning from failure. In: O'CONNOR, G. (Ed.). **Site Reliability Engineering: How Google Runs Production Systems**. O'Reilly Media, 2016. Disponível em: https://sre.google/sre-book/postmortem-culture/. Acesso em: 6 fev. 2022.

BOUTABA, R.; CHENG, L.; ZHANG, Q. Cloud computing: state-of-the-art and research challenges. **Journal of Internet Services and Applications**, v. 1, n. 1, p. 7–18, 2010.

BUCHANAN, I. **CALMS Framework**. 2022. Disponível em: https://www.atlassian.com/devops/frameworks/calms-framework. Acesso em: 23 jul. 2022.

BURNS, J. **How to Get Started with Chaos: A Step-by-Step Guide to Gamedays**. Lightstep, 2019. Disponível em: https://lightstep.com/blog/get-started-with-chaos-guide-to-gamedays/. Acesso em: 12 jan. 2022.

BUTOW, T. **Chaos Engineering: the history, principles, and practice**. Gremlin, 2021. Disponível em: https://www.gremlin.com/community/tutorials/chaos-engineering-the-history-principles-and-practice/. Acesso em: 19 mar. 2022.

BUTOW, T.; SALINAS, E. Tammy bütow on chaos engineering. **IEEE Software**, v. 35, n. 5, p. 125–128, 2018.

CACHIN, C.; GUERRAOUI, R.; RODRIGUES, L. **Introduction to Reliable and Secure Distributed Programming**. 2. ed. Heidelberg: Springer, 2011.

CAMPBELL, S.; JERONIMO, M. **Applied Virtualization Technology**. [S.l.]: Intel Press, 2006.

CENTER, P. R. **Americans turn to technology during COVID-19 outbreak, say an outage would be a problem**. 2020. Disponível em: https://www.pewresearch.org/fact-tank/2020/03/31/americans-turn-to-technology-during-covid-19-outbreak-say-an-outage-would-be-a-problem/. Acesso em: 26 out. 2021.

CHAOS COMMUNITY. **Principles of Chaos Engineering**. 2019. Disponível em: https://principlesofchaos.org/. Acesso em: 19 jul. 2021.

CHAOS TOOLKIT. **Chaos Toolkit**. 2022. Disponível em: https://chaostoolkit.org/. Acesso em: 23 mar. 2022.

CHAOS TOOLKIT. **Chaos Toolkit**. 2022. Disponível em: https://chaostoolkit.org/reference/api/experiment/. Acesso em: 23 mar. 2022.

CHAOS TOOLKIT. **Chaos Toolkit**. 2022. Disponível em: https://chaostoolkit.org/reference/usage/run/. Acesso em: 23 mar. 2022.

CHAOSNATIVE. **Homepage Chaos Carnival**. 2022. Disponível em: https://chaoscarnival.io/. Acesso em: 22 mai. 2022.

COLLIER, B.; DEMARCO, T.; FEAREY, P. A defined process for project post mortem review. **IEEE Software**, v. 13, n. 4, p. 65–72, 1996.

CRISTIAN, F. Understanding fault-tolerant distributed systems. **Communications of the ACM**, Association for Computing Machinery, New York, NY, USA, v. 34, n. 2, p. 56–78, 1991. Disponível em: https://doi.org/10.1145/102792.102801.

DEVARAJ, B.; LI, X. **Resilience Engineering at LinkedIn with Project Waterbear**. 2017. Disponível em: https://engineering.linkedin.com/blog/2017/11/resilience-engineering-at-linkedin-with-project-waterbear. Acesso em: 11 jul. 2022.

DEVOPEDIA. **Shift Left**. 2021. Disponível em: https://devopedia.org/shift-left. Acesso em: 26 out. 2021.

DOCKER. **Why Docker**. 2022. Disponível em: https://www.docker.com/why-docker/. Acesso em: 30 jul. 2022.

ECMA INTERNATIONAL. **Introducing JSON**. 2022. Disponível em: https://www.json.org/json-en.html. Acesso em: 07 mar. 2022.

FACEBOOK. **More details about the October 4 outage**. 2021. Disponível em: https://engineering.fb.com/2021/10/05/networking-traffic/outage-details/. Acesso em: 26 out. 2021.

FARLEY, D.; HUMBLE, J. **Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation**. [S.l.]: Addison-Wesley, 2010.

FERNÁNDEZ, M. E. **Redefining Blameless Post-Mortem Terminology**. 2021. Disponível em: https://matiasfrndz.ch/2021/03/Redefining-Blameless-Post-Mortem-Terminology. Acesso em: 18 fev. 2022.

FOWLER, M. **ExtremeProgramming**. 2013. Disponível em: https://martinfowler.com/bliki/ExtremeProgramming.html. Acesso em: 24 jul. 2022.

FOWLER, M. **Microservices**. 2014. Disponível em: https://martinfowler.com/articles/microservices.html. Acesso em: 22 mai. 2022.

FOWLER, M.; FOEMMEL, M. **Continuous Integration**. 2006. Disponível em: https://martinfowler.com/articles/continuousIntegration.html. Acesso em: 24 jul. 2022.

FRANK, S. et al. Identifying and prioritizing chaos experiments by using established risk analysis techniques. In: **2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)**. Coimbra, Portugal: [s.n.], 2020. p. 229–240.

GRANCE, T.; MELL, P. **The NIST Definition of Cloud Computing**. Gaithersburg, MD, USA: National Institute of Standards and Technology, 2011. Disponível em: https://doi.org/10.6028/NIST.SP.800-145. Acesso em: 27 jul. 2022.

GREMLIN. **How to Run a Gameday**. 2018. Disponível em: https://www.gremlin.com/community/tutorials/how-to-run-a-gameday/. Acesso em: 8 jan. 2022.

GREMLIN. **The Simian Army**. 2018. Disponível em: https://www.gremlin.com/chaos-monkey/the-simian-army/. Acesso em: 30 out. 2021.

GREMLIN. **A Step-by-Step Guide to Creating Failure on AWS**. 2018. Disponível em: https://www.gremlin.com/chaos-monkey/chaos-monkey-tutorial/. Acesso em: 06 mar. 2022.

GREMLIN. **Chaos Conf 2020**. 2020. Disponível em: https://www.gremlin.com/chaos-conf/2020/. Acesso em: 19 mar. 2022.

GREMLIN. **Homepage Chaos Conf**. 2020. Disponível em: https://www.chaosconf.io/. Acesso em: 22 mai. 2022.

GREMLIN. **Failover Conf 2021**. 2021. Disponível em: https://www.gremlin.com/failoverconf/2021/. Acesso em: 19 mar. 2022.

GREMLIN. **State of Chaos Engineering 2021**. 2021. Disponível em: https://www.gremlin.com/state-of-chaos-engineering/2021/?soce2021=true. Acesso em: 25 out. 2021.

GREMLIN. **About Gremlin**. 2022. Disponível em: https://www.gremlin.com/leadership/. Acesso em: 20 mar. 2022.

GREMLIN. **Chaos Engineering Slack Channel**. 2022. Disponível em: https://www.gremlin.com/community/. Acesso em: 19 mar. 2022.

GREMLIN. **Chaos Monkey Guide for Engineers**. 2022. Disponível em: https://www.gremlin.com/chaos-monkey/. Acesso em: 06 mar. 2022.

GREMLIN. **Gremlin-certified Chaos Engineer**. 2022. Disponível em: https://www.gremlin.com/certification/. Acesso em: 19 mar. 2022.

GREMLIN. **Gremlin Docs: Clients - Overview**. 2022. Disponível em: https://www.gremlin.com/docs/clients/overview/. Acesso em: 20 mar. 2022.

GREMLIN. **Gremlin Docs: Infrastructure Layer - Attacks**. 2022. Disponível em: https://www.gremlin.com/docs/infrastructure-layer/attacks/. Acesso em: 20 mar. 2022.

GREMLIN. **Gremlin Docs: Infrastructure Layer - Common errors with solutions**. 2022. Disponível em: https://www.gremlin.com/docs/infrastructure-layer/common-errors-with-solutions/. Acesso em: 20 mar. 2022.

GREMLIN. **Gremlin Docs: Infrastructure Layer - Targets**. 2022. Disponível em: https://www.gremlin.com/docs/infrastructure-layer/targets/. Acesso em: 20 mar. 2022.

GREMLIN. **Gremlin Docs: Security - Overview**. 2022. Disponível em: https://www.gremlin.com/docs/security/overview/. Acesso em: 22 mar. 2022.

GREMLIN. **Gremlin Pricing**. 2022. Disponível em: https://www.gremlin.com/pricing/. Acesso em: 22 mar. 2022.

HABER, T. **How to Run an Adversarial Game Day**. New Relic, 2019. Disponível em: https://newrelic.com/blog/best-practices/how-to-run-a-game-day. Acesso em: 5 fev. 2022.

HORNSBY, A. **AWS re:Invent 2020 - AWS Fault Injection Simulator: Fully managed chaos engineering service**. Amazon Web Services, 2020. Disponível em: https://www.youtube.com/watch?v=yoNeMLj3CHc. Acesso em: 07 mar. 2022.

INFOQ. **The InfoQ eMag: Chaos Engineering**. 2018. Disponível em: https://www.infoq.com/minibooks/emag-chaos-engineering. Acesso em: 11 jul. 2022.

INFOQ. **DevOps and Cloud InfoQ Trends Report - February 2019**. 2019. Disponível em: https://www.infoq.com/articles/devops-cloud-trends-2019/. Acesso em: 01 ago. 2021.

JERNBERG, H. **Building a Framework for Chaos Engineering**. Dissertação (Mestrado) — Lund University, Suécia, 2020.

JONES, N. **AWS re:Invent 2017: Performing Chaos at Netflix Scale**. Las Vegas: Amazon Web Services, 2017. Disponível em: https://www.youtube.com/watch?v=LaKGx0dAUlo. Acesso em: 9 fev. 2022.

KIM, G. et al. **The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations**. [S.l.]: IT Revolution Press, 2016.

KLEMENTIEV, D. **Lista de discussões Chaos Community**. 2015. Disponível em: https://groups.google.com/g/chaos-community. Acesso em: 29 jun. 2022.

KRISHNAN, K. Weathering the unexpected: Failures happen, and resilience drills help organizations prepare for them. **ACM Queue**, Association for Computing Machinery, New York, NY, USA, v. 10, n. 9, p. 30–37, 2012.

LAMPORT, L. Paxos made simple. **ACM SIGACT News**, v. 32, n. 4, p. 18–25, 2001.

LASKEY, K. B.; LASKEY, K. Service oriented architecture. **WIREs Computational Statistics**, v. 1, n. 1, p. 101–105, 2009.

NETFLIX. **5 Lessons We've Learned Using AWS**. 2010. Disponível em: https://netflixtechblog.com/5-lessons-weve-learned-using-aws-1f2a28588e4c. Acesso em: 27 out. 2021.

NETFLIX. **The Netflix Simian Army**. 2011. Disponível em: https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116. Acesso em: 19 jul. 2021.

NETFLIX. **FIT: Failure Injection Testing**. 2014. Disponível em: https://netflixtechblog.com/fit-failure-injection-testing-35d8e2a9bb2. Acesso em: 30 out. 2021.

NETFLIX. **SPS: The Pulse of Netflix Streaming**. 2015. Disponível em: https://netflixtechblog.com/sps-the-pulse-of-netflix-streaming-ae4db0e05f8a. Acesso em: 09 nov. 2021.

NETFLIX. **ChAP: Chaos Automation Platform**. 2017. Disponível em: https://netflixtechblog.com/chap-chaos-automation-platform-53e6d528371f. Acesso em: 05 nov. 2021.

NETFLIX. **Hystrix Docs: How it Works**. GitHub, 2017. Disponível em: https://github.com/Netflix/Hystrix/wiki/How-it-Works. Acesso em: 27 mar. 2022.

NETFLIX. **Hystrix Docs: How to Use**. GitHub, 2017. Disponível em: https://github.com/Netflix/Hystrix/wiki/How-To-Use. Acesso em: 27 mar. 2022.

NETFLIX. **Archaius**. GitHub, 2019. Disponível em: https://github.com/Netflix/archaius. Acesso em: 28 mar. 2022.

NETFLIX. **Chaos Monkey Documentation**. 2019. Disponível em: https://netflix.github.io/chaosmonkey/. Acesso em: 06 mar. 2022.

NETFLIX. **Chaos Monkey**. GitHub, 2020. Disponível em: https://github.com/Netflix/chaosmonkey. Acesso em: 30 jan. 2022.

NETFLIX. **Hystrix: Latency and Fault Tolerance for Distributed Systems**. GitHub, 2021. Disponível em: https://github.com/Netflix/Hystrix. Acesso em: 01 nov. 2021.

NETFLIX. **Ribbon**. GitHub, 2021. Disponível em: https://github.com/Netflix/Ribbon. Acesso em: 01 nov. 2021.

NETFLIX. **Security Monkey**. GitHub, 2021. Disponível em: https://github.com/Netflix/security_monkey. Acesso em: 30 out. 2021.

NETFLIX. **Zuul**. GitHub, 2021. Disponível em: https://github.com/Netflix/zuul. Acesso em: 01 nov. 2021.

NOVEGIL, A. **Awesome Chaos Engineering**. GitHub, 2022. Disponível em: https://github.com/adriannovegil/awesome-chaos-engineering. Acesso em: 29 mar. 2022.

NYGARD, M. **Release It! Design and Deploy Production-Ready Software**. [S.l.]: Pragmatic Bookshelf, 2007.

PAGERDUTY. **The Blameless Postmortem**. 2021. Disponível em: https://postmortems.pagerduty.com/culture/blameless/. Acesso em: 19 fev. 2022.

RATIS, P. **Awesome Chaos Engineering**. GitHub, 2022. Disponível em: https://github.com/dastergon/awesome-chaos-engineering. Acesso em: 1 mar. 2022.

REDHAT. **Stateful vs Stateless**. 2020. Disponível em: https://www.redhat.com/en/topics/cloud-native-apps/stateful-vs-stateless. Acesso em: 31 jul. 2022.

ROBBINS, J. **GameDay: Creating Resiliency Through Destruction**. Boston: USENIX Association, 2011. Disponível em: https://www.youtube.com/watch?v=zoz0ZjfrQ9s. Acesso em: 8 jan. 2022.

ROBBINS, J. et al. **Resilience Engineering: Learning to Embrace Failure**. Association for Computing Machinery, 2012. Disponível em: https://queue.acm.org/detail.cfm?id=2371297. Acesso em: 04 jan. 2022.

ROSEN, L. **LinkedOut: A Request-Level Failure Injection Framework**. 2018. Disponível em: https://engineering.linkedin.com/blog/2018/05/linkedout--a-request-level-failure-injection-framework. Acesso em: 17 jul. 2022.

ROSENTHAL, C. et al. Chaos engineering. **IEEE Software**, v. 33, n. 3, 2016.

ROSENTHAL, C. et al. **Chaos Engineering: Building Confidence in System Behavior through Experiments**. 1. ed. [S.l.]: O'Reilly Media, 2017.

ROTEM-GAL-OZ, A. Fallacies of distributed computing explained. **Doctor Dobb's Journal**, Janeiro 2008. Disponível em: https://www.researchgate.net/publication/322500050_Fallacies_of_Distributed_Computing_Explained. Acesso em: 13 ago. 2022.

SAREEN, P. Cloud computing: types, architecture, applications, concerns, virtualization and role of it governance in cloud. **International Journal of Advanced Research in Computer Science and Software Engineering**, v. 3, n. 3, 2013.

SAULITIS, P. et al. **Chaos Engineering: A Lesson From the Experts**. 2017. Disponível em: https://dzone.com/articles/chaos-engineering-a-lesson-from-the-experts. Acesso em: 22 jan. 2022.

STEEN, M. van; TANENBAUM, A. S. **Distributed Systems**. 3. ed. Pearson Education, 2018. Disponível em: https://distributed-systems.net.

TECHCRUNCH. **Zoom meetings hit by outage**. 2020. Disponível em: https://techcrunch.com/2020/08/24/zoom-meetings-hit-by-outage/. Acesso em: 26 out. 2021.

THE LINUX FOUNDATION. **Kubernetes: Production-Grade Container Orchestration**. 2022. Disponível em: https://kubernetes.io/. Acesso em: 20 mar. 2022.

THE LINUX FOUNDATION. **Spinnaker**. 2022. Disponível em: https://spinnaker.io/. Acesso em: 05 mar. 2022.

THE POLLY PROJECT. **Polly**. GitHub, 2022. Disponível em: https://github.com/App-vNext/Polly. Acesso em: 28 mar. 2022.

THE POLLY PROJECT. **Simmy**. GitHub, 2022. Disponível em: https://github.com/Polly-Contrib/Simmy. Acesso em: 28 mar. 2022.

TUCKER, H. et al. The business case for chaos engineering. **IEEE Cloud Computing**, v. 5, n. 3, p. 45–54, 2018.

VLAOVIC, S. et al. **Completing the Netflix Cloud Migration**. 2016. Disponível em: https://about.netflix.com/en/news/completing-the-netflix-cloud-migration. Acesso em: 19 jul. 2021.

WIKIPEDIA. **Viés Cognitivo**. Wikipedia, a enciclopédia livre, 2019. Disponível em: https://pt.wikipedia.org/wiki/Vi%C3%A9s_cognitivo. Acesso em: 27 fev. 2022.

WIKIPEDIA. **Hindsight bias**. Wikipédia, a enciclopédia livre, 2021. Disponível em: https://en.wikipedia.org/wiki/Hindsight_bias. Acesso em: 27 fev. 2022.

WIKIPEDIA. **YAML**. 2022. Disponível em: https://en.wikipedia.org/wiki/YAML. Acesso em: 07 mar. 2022.

WINKLER, R. et al. **Resilience4j: Fault tolerance library designed for functional programming**. GitHub, 2022. Disponível em: https://github.com/resilience4j/resilience4j. Acesso em: 28 mar. 2022.

WOODS, D. **STELLA: Report from the SNAFUcatchers Workshop on Coping With Complexity**. The Ohio State University, 2017. Disponível em: https://snafucatchers.github.io/. Acesso em: 10 fev. 2022.

XU, X. et al. Availability analysis for deployment of in-cloud applications. In: **Proceedings of the 4th International ACM Sigsoft Symposium on Architecting Critical Systems**. New York, NY, USA: Association for Computing Machinery, 2013. (ISARCS '13), p. 11–16.

ANEXOS

ANEXO A – EXEMPLO DE TEMPLATE PARA EXPERIMENTO DO AWS FIS

Código 1 – Template do AWS FIS, em formato JSON

```
{
    "tags": {
        "Name": "StopEC2InstancesByCount"
    },
    "description": "Stop and restart three instances with the
       specified tag",
    "targets": {
        "myInstances": {
            "resourceType": "aws:ec2:instance",
            "resourceTags": {
                "env": "prod"
            },
            "selectionMode": "COUNT(3)"
        }
    },
    "actions": {
        "StopInstances": {
            "actionId": "aws:ec2:stop-instances",
            "description": "stop the instances",
            "parameters": {
                "startInstancesAfterDuration": "PT2M"
            },
            "targets": {
                "Instances": "myInstances"
            }
        }
    },
    "stopConditions": [
        {
            "source": "aws:cloudwatch:alarm",
            "value": "arn:aws:cloudwatch:us-east
              -1:111122223333:alarm:alarm-name"
        }
    ],
    "roleArn": "arn:aws:iam::111122223333:role/role-name"
}
```

ANEXO B – EXEMPLO DE DEFINIÇÃO DE UM EXPERIMENTO DO CHAOS TOOLKIT

Código 2 – Arquivo de definição do Chaos Toolkit, em formato JSON (parte 1)

```
{
    "title": "What is the impact of an expired certificate on
        our application chain?",
    "description": "If a certificate expires, we should
       gracefully deal with the issue.",
    "steady-state-hypothesis": {
        "title": "Application responds",
        "probes": [
            {
                "type": "probe",
                "name": "the-sunset-service-must-be-running",
                "tolerance": true,
                "provider": {
                    "type": "python",
                    "module": "os.path",
                    "func": "exists",
                    "arguments": {
                        "path": "sunset.pid"
                    }
                }
            },
            {
                "type": "probe",
                "name": "we-can-request-sunset",
                "tolerance": 200,
                "provider": {
                    "type": "http",
                    "timeout": 3,
                    "verify_tls": false,
                    "url": "https://localhost:8443/city/Paris
                       "
                }
            }
        ]
    },
```

Código 3 – Arquivo de definição do Chaos Toolkit, em formato JSON (parte 2)

```
    "method": [
        {
            "type": "action",
            "name": "swap-to-expired-cert",
            "provider": {
                "type": "process",
                "path": "cp",
                "arguments": "expired-cert.pem cert.pem"
            }
        },
        {
            "type": "probe",
            "name": "read-tls-cert-expiry-date",
            "provider": {
                "type": "process",
                "path": "openssl",
                "arguments": "x509 -enddate -noout -in cert.
                   pem"
            }
        },
        {
            "type": "action",
            "name": "restart-sunset-service-to-pick-up-
               certificate",
            "provider": {
                "type": "process",
                "path": "pkill",
                "arguments": "  echo   -HUP -F sunset.pid"
            },
            "pauses": {
                "after": 1
            }
        }
    ],
```

Código 4 – Arquivo de definição do Chaos Toolkit, em formato JSON (parte 3)

```
    "rollbacks": [
        {
            "type": "action",
            "name": "swap-to-valid-cert",
            "provider": {
                "type": "process",
                "path": "cp",
                "arguments": "valid-cert.pem cert.pem"
            }
        },
        {
            "ref": "restart-sunset-service-to-pick-up-
               certificate"
        }
    ]
}
```