UNIVERSIDADE FEDERAL RURAL DE PERNAMBUCO

DEPARTAMENTO DE ESTATÍSTICA E INFORMÁTICA

PROGRAMA DE PÓS–GRADUAÇÃO EM INFORMÁTICA APLICADA

ALEX CORDEIRO CUNHA

AVALIAÇÃO DE DEPENDABILIDADE DE AMBIENTES DE NUVENS PRIVADAS BASEADA EM CLASSIFICAÇÃO DE DEFEITOS E INJEÇÃO DE FALHAS

RECIFE – PE

2023

ALEX CORDEIRO CUNHA

AVALIAÇÃO DE DEPENDABILIDADE DE AMBIENTES DE NUVENS PRIVADAS BASEADA EM CLASSIFICAÇÃO DE DEFEITOS E INJEÇÃO DE FALHAS

Dissertação submetida à Coordenação do Programa de Pós-Graduação em Informática Aplicada da Universidade Federal Rural de Pernambuco, como parte dos requisitos necessários para obtenção do grau de Mestre.

ORIENTADOR: Erica Teixeira Gomes de Sousa

RECIFE – PE

2023

Dados Internacionais de Catalogação na Publicação
Universidade Federal Rural de Pernambuco
Sistema Integrado de Bibliotecas
Gerada automaticamente, mediante os dados fornecidos pelo(a) autor(a)

A374a Cunha, Alex Cordeiro
Avaliação de Dependabilidade de Ambientes de Nuvens Privadas Baseada em Classificação de Defeitos e Injeção de Falhas / Alex Cordeiro Cunha. - 2023.
79 f. : il.

Orientadora: Erica Teixeira Gomes de Sousa.
Inclui referências.

Dissertação (Mestrado) - Universidade Federal Rural de Pernambuco, Programa de Pós-Graduação em Informática Aplicada, Recife, 2023.

1. Avaliação de Dependabilidade. 2. Computação em Nuvem. 3. Injeção de Falhas. 4. Classificação de Defeitos. I. Sousa, Erica Teixeira Gomes de, orient. II. Título

CDD 004

ALEX CORDEIRO CUNHA

AVALIAÇÃO DE DEPENDABILIDADE DE AMBIENTES DE NUVENS PRIVADAS BASEADA EM CLASSIFICAÇÃO DE DEFEITOS E INJEÇÃO DE FALHAS

Dissertação submetida à Coordenação do Programa de Pós-Graduação em Informática Aplicada da Universidade Federal Rural de Pernambuco, como parte dos requisitos necessários para obtenção do grau de Mestre.

Aprovada em: 13 de março de 2023.

BANCA EXAMINADORA

Erica Teixeira Gomes de Sousa (Orientadora)
Universidade Federal Rural de Pernambuco
Departamento de Computação

Fernando Antônio Aires Lins
Universidade Federal Rural de Pernambuco
Departamento de Computação

Almir Pereira Guimarães
Universidade Federal de Alagoas
Instituto de Computação

Agradecimentos

A Deus pela oportunidade concedida e ao Universo pela energia correspondida. A Minha Família por estarem sempre na torcida. À Universidade Federal Rural de Pernambuco pela acolhida e a FACEPE, pela bolsa concedida, que contribuiu para a concretização da pesquisa. Ao Eduardo, assistente administrativo do PPGIA, que várias vezes atendeu às minhas solicitações. Aos professores das disciplinas ministradas durante o curso, em especial o Professor Dr. Gustavo Callou, que ministrou a disciplina Avaliação de Desempenho e Dependabilidade de Sistemas na qual serviu como base para escrita do artigo Modelagem Hierárquica e Heterogênea para Avaliação de Disponibilidade de Aplicações Big Data na Nuvem Privada, onde fomos agraciados com o prêmio BEST PAPER no Workshop em Desempenho de Sistemas Computacionais e de Comunicação (WPerformance 2021), em que tivemos a valiosíssima contribuição do Professor Dr. Eduardo Tavares da UFPE, a quem sou muito grato pela colaboração. E especialmente à professora Dra. Érica Sousa, por ter me aceitado como seu orientando, por remover os obstáculos em momento atípico no mundo ao sermos acometidos pela pandemia da COVID-19, onde ficamos impossibilidade de ir até a Universidade para orientação, desenvolvimento e realização de experimentos relativos ao trabalho desenvolvido, pela orientação, os ensinamentos, indicação e a dedicação ao seu trabalho. Meu sincero obrigado!

Resumo

A nuvem privada vem se consolidando como uma alternativa viável às nuvens públicas, motivada por preocupações como custo, segurança e armazenamento de dados. Porém, manter os níveis de dependabilidade de ambientes de nuvem conforme os acordos de nível de serviço é um desafio para os provedores desse ambiente, dado o nível de complexidade em relação à sua arquitetura e a diversidade de componentes de software e hardware envolvidos. Defeitos nestes ambientes são uma ameaça operacional permanente. Compreender as características, os efeitos e estimar o impacto de falhas no ambiente de nuvem privada é uma abordagem importante para identificar fatores que podem afetar seus serviços. Diante disso, a análise do impacto de defeitos nos serviços por meio da injeção de falhas, considerando métricas como carga de trabalho e utilização de recursos computacionais, como memória e processador, é essencial para avaliar a dependabilidade de nuvens privadas. Este trabalho apresenta uma estratégia baseada em uma metodologia de classificação de defeitos em nuvens privadas e uma solução para injeção de falhas. Uma metodologia foi proposta para classificação de defeitos, aplicando a técnica de Classificação Ortogonal de Defeitos (*ODC*). Essa metodologia contempla atividades como estudo comparativo entre plataformas de computação de nuvens privadas, escolha de repositório, aplicação de *activity*, escolha da *trigger* e definição do *defect type*. Uma solução *Web* para injeção de falhas foi desenvolvida para emular defeitos reportados por usuários da nuvem privada. A metodologia teve como objetivo avaliar a dependabilidade de nuvens privadas, compreendendo as características, efeitos e estimando o impacto de falhas nesse ambiente. O resultado prático gerado foi a demonstração da aplicabilidade da metodologia por meio de estudos de caso. A análise foi realizada em um *dataset* com 395 defeitos reportados ao longo de três anos (2018 a 2021), e a eficiência da ferramenta foi verificada ao emular 39 defeitos injetados nos componentes Nova, Neutron, Swift e Cinder da plataforma *OpenStack*. Esses defeitos afetaram a conectividade externa com as instâncias criadas e causaram falhas no uso da *API*.

Palavras-chave: Avaliação de Dependabilidade, Computação em Nuvem, Injeção de Falhas, Classificação de Defeitos.

Abstract

The private cloud has been consolidating as a viable alternative to public clouds, driven by concerns such as cost, security, and data storage. However, maintaining the levels of dependability of cloud environments according to service level agreements is a challenge for providers of this environment, given the level of complexity in relation to its architecture and the diversity of software and hardware components involved. Defects in these environments are a permanent operational threat. Understanding the characteristics, the effects, and estimating the impact of failures in the private cloud environment is an important approach to identify factors that can affect their services. In light of this, the analysis of the impact of defects on services through fault injection, considering metrics such as workload and the use of computational resources, such as memory and processor, is essential to assess the dependability of private clouds. This work presents a strategy based on a defect classification methodology in private clouds and a solution for fault injection. A methodology was proposed for defect classification, applying the Orthogonal Defect Classification (ODC) technique. This methodology includes activities such as comparative study between private cloud computing platforms, choice of repository, application of activity, choice of trigger, and definition of defect type. A Web solution for fault injection was developed to emulate defects reported by private cloud users. The methodology aimed to evaluate the dependability of private clouds, understanding the characteristics, effects, and estimating the impact of failures in this environment. The practical result generated was the demonstration of the applicability of the methodology through case studies. The analysis was carried out on a dataset with 395 defects reported over three years (2018 to 2021), and the efficiency of the application was verified by emulating 39 defects injected into the Nova, Neutron, Swift, and Cinder components of the OpenStack platform. These defects affected external connectivity with the instances created and caused failures in the use of the API.

Keywords: Dependability Assessment, Cloud Computing, Fault Injection, Defect Classification.

Lista de Figuras

Lista de tabelas

Lista de Siglas

ACM	*Association for Computing Machinery.*
AMQP	*Advanced Message Queuing Protocol.*
API	*Application Programming Interface.*
AWS	*Amazon Web Service.*
BSD	*Berkeley Software Distribution.*
CDN	*Content Delivery Network.*
CLI	*Command-line Interface.*
CMMI	*Capability Maturity Model Integration.*
CSV	*Comma-Separated-Values.*
EC2	*Elastic Compute Cloud.*
IBM	*International Business Machines Corporation.*
IaaS	*Infraestructure as a Service.*
IEEE	*Institute of Electrical and Electronic Engineers.*
KVM	*Kernel-based Virtual Machine.*
MIT	*Massachusetts Institute of Technology.*
NIST	*National Institute of Standards and Technology.*
ODC	*Orthogonal Defect Classification.*
PaaS	*Platafor as a Service.*
RDO	*RPM Distribution of OpenStack.*
RPC	*Remote Procedure Call.*
SaaS	*Software as a Service.*
SDK	*Software Development Kit.*
SDLC	*Software Development Life Cycle.*
SWIFI	*Software Implemented Fault Injection.*
TI	*Information Technology.*
URL	*Uniform Resource Locator.*

VM	*Virtual Machine.*
VPN	*Virtual Private Network.*
XML	*Extensible Markup Language.*

Sumário

1 Introdução

A computação em nuvem é uma das técnicas de computação amplamente utilizadas na indústria, abrangendo uma ampla gama de serviços, como armazenamento de dados, processamento em larga escala, hospedagem de websites, *machine learning*, análise de dados, *streaming* de vídeos, infraestrutura como serviço (IaaS), plataforma como serviço (PaaS) e software como serviço (SaaS). Os serviços em nuvem oferecem diversas vantagens, tais como agilidade, resposta rápida, redução dos custos operacionais e uma ampla gama de outros benefícios. Esses serviços são úteis para atender aos requisitos cada vez mais complexos dos clientes desse ambiente. A indústria de computação em nuvem está crescendo rapidamente e os fornecedores estão investindo uma quantidade significativa de dinheiro para desenvolver software como serviço (SaaS) e aproveitar os benefícios da computação em nuvem em várias atividades de negócios (ELMORSHIDY, 2019).

No entanto, o crescente uso da computação em nuvem em vários segmentos torna a dependabilidade uma grande preocupação tanto na indústria quanto na academia, especialmente para aplicações em tempo real (ABOHAMAMA et al., 2018).

Nesse contexto, a categorização de defeitos reportados por usuários de ambientes de computação em nuvem pode ser feita com base no seu impacto no ambiente em execução. A classificação pode ser aplicada de acordo com a taxonomia *Orthogonal Defect Classification* (*ODC*). A taxonomia *ODC* possui treze categorias que permitem aos desenvolvedores classificar os defeitos dependendo de seu impacto no serviço oferecido ao cliente (HERNáNDEZ-GONZáLEZ et al., 2018).

Sistemas distribuídos modernos atingiram um nível de complexidade em que defeitos de software e falhas de hardware não são mais excepcionais, mas uma ameaça operacional permanente. Isso vale especialmente para infraestruturas de computação em nuvem que precisam fornecer recursos a seus clientes sob acordos de nível de serviço bem definidos. A injeção de falhas contribui para verificar a dependabilidade de plataformas de computação em nuvens ao aplicar cargas de trabalho emuladas com características *fail-stop* que interrompem o funcionamento das máquinas virtuais ou *fail-silent* causando travamento (FEINBUBE et al., 2017).

1.1 Motivação e Justificativa

A computação em nuvem pode ajudar as organizações a criar valor comercial, fornecendo flexibilidade e versatilidade. Dados de pesquisa coletados de 174 empresas revelaram que 59% consideram-se satisfeitas com o nível de segurança na nuvem, enquanto 34% consideram a nuvem um recurso confiável no gerenciamento de informações. Além disso, 55% das pequenas empresas concordaram que a computação em nuvem reduz custos (ALIJANI et al., 2014). No entanto, sistemas de computação em nuvem estão propensos a vários problemas em tempo de execução, causados por falhas de hardware e software. A garantia da dependabilidade, que se refere à confiabilidade e resiliência dos sistemas em fornecer serviços corretos e confiáveis ao longo do tempo, é crucial para a criação de serviços sustentáveis em ambientes de computação em nuvem (GUAN et al., 2012).

Criar uma nuvem privada, viável economicamente e disponível é um desafio. Para melhorar a dependabilidade, administradores de sistema geralmente empregam hardware redundante, o que aumenta o custo de aquisição. Há muitas decisões envolvidas neste processo, como: a quantidade de servidores que serão empregados e a função de cada um deles na infraestrutura (OLIVEIRA et al., 2019). À medida que a complexidade do ambiente de computação em nuvem aumenta, várias falhas podem causar problemas, como inatividade de máquinas virtuais, e esses eventos degradam seriamente a dependabilidade deste ambiente (WANG et al., 2015). Além disso, a virtualização em ambientes de computação em nuvem podem gerar problemas técnicos, como ataques, despejos de memória e várias falhas.

A injeção de falhas pode contribuir para o aumento da dependabilidade em ambientes de nuvens privadas, criando situações de falha e observando seus efeitos nos serviços ofertados (XIAOYONG et al., 2014).

1.2 Problema de Pesquisa

A computação em nuvem oferece serviços sob demanda que são pagos conforme sua utilização. O cliente de nuvem pode escolher entre várias ofertas de provedores de nuvem que atendam às suas necessidades.

A eficiência de ambientes em nuvem, permite acesso onipresente e provimento de recursos computacionais configuráveis, que podem ser rapidamente provisionados e liberados com mínimo de esforço de gerenciamento ou intervenção do provedor de serviços, associado a

custos acessíveis, foram determinantes para a adoção das nuvens públicas pelas empresas. Porém, preocupações relacionadas a dependabilidade, segurança, armazenamento de dados, motivou o aumento da migração para o uso de arquiteturas baseadas em nuvem privada. Conforme sugerido por Alhaisoni (ALHAISONI, 2015), a nuvem privada poderia ser considerada uma alternativa potencialmente mais econômica para a aquisição e manutenção da infraestrutura de sistemas de organizações. Os estudos daquela época apontavam que, em um horizonte de 5 anos após a migração, poderia haver uma redução significativa nos custos, com economias estimadas em 74% nos custos operacionais, 8% em gerenciamento e 63% na aquisição de equipamentos.

A computação em nuvem revolucionou o setor empresarial ao permitir que serviços e recursos fossem disponibilizados sob demanda para as empresas. A continuidade e a constância no fornecimento desses serviços são cruciais, exigindo que os ambientes de computação em nuvem operem não apenas com alta precisão, mas também com tolerância a falhas (PATTANAYAK et al., 2020). Na esfera das nuvens privadas, a dependabilidade é frequentemente comprometida por defeitos associados a componentes de hardware e software. Assim, qualquer alegação relativa à dependabilidade desses ambientes deve ser fundamentada em uma análise detalhada das causas subjacentes, quer se originem no próprio sistema ou no ambiente em que ele opera (FEINBUBE et al., 2017).

Portanto, a dependabilidade dos ambientes de nuvens privadas é de extrema importância devido às preocupações relacionadas à segurança, armazenamento de dados e tolerância a falhas. A aplicação de métodos personalizados de teste, como a injeção de falhas, revela-se uma abordagem versátil para a avaliação da dependabilidade desses ambientes. Compreender as causas das falhas e seus impactos no sistema investigado ou em seu ambiente de execução é essencial para assegurar a entrega de serviços ininterruptos e confiáveis. O problema de pesquisa desta dissertação é descrito da seguinte forma: “Como avaliar a dependabilidade de ambientes de nuvens privadas?”.

1.3 Objetivos

O objetivo central deste trabalho é realizar uma avaliação aprofundada da dependabilidade em ambientes de computação em nuvem privada, utilizando a técnica de injeção de falhas para simular condições adversas e medir a resiliência do sistema. Para alcançar este fim, o trabalho está estruturado em torno dos seguintes objetivos específicos:

- Desenvolver e detalhar uma metodologia para a classificação sistemática de defeitos em ambientes de nuvem privada, empregando a abordagem da Classificação Ortogonal de Defeitos (*ODC - Orthogonal Defect Classification*), com o intuito de facilitar a identificação e o entendimento das falhas potenciais e suas origens;
- Projetar e implementar uma ferramenta dedicada à injeção de falhas, especificamente adaptada para o contexto de nuvens privadas, que possa emular uma variedade de cenários de defeitos reais e hipotéticos, proporcionando um meio robusto para testar e melhorar a dependabilidade desses sistemas.

Estes objetivos visam contribuir para a compreensão teórica e aplicada sobre como falhas impactam a operação e a manutenção de infraestruturas de nuvem privada, bem como fornecer estratégias práticas para o fortalecimento da confiabilidade e disponibilidade dos serviços de nuvem.

1.4 Estrutura do Documento

Este documento está dividido em sete capítulos. Após o Capítulo 1, o Capítulo 2 apresenta a fundamentação teórica, com os principais conceitos e terminologias relacionados ao tema deste trabalho. Em seguida, o Capítulo 3 aborda os trabalhos relacionados à abordagem deste documento e faz uma comparação das contribuições. O Capítulo 4 abrange a metodologia utilizada para classificação de defeitos em plataformas de nuvens privadas. No Capítulo 5, é apresentada a solução FIES. Em seguida, no Capítulo 6, são apresentados os estudos de caso nos quais a metodologia proposta é aplicada e a ferramenta de injeção de falhas é utilizada para avaliar a dependabilidade de um ambiente de nuvem privada. Por fim, o Capítulo 7 traz as conclusões, contribuições, limitações e sugestões para trabalhos futuros.

2 Fundamentação Teórica

Este capítulo apresenta uma visão geral dos conceitos mais relevantes para o entendimento deste trabalho. Este capítulo está organizado da seguinte maneira: inicialmente são abordados os conceitos básicos sobre Computação em Nuvem. Posteriormente, noções básicas sobre dependabilidade. Em seguida são apresentados os modelos para classificação de defeitos. Por fim, são abordadas as principais técnicas de injeção de falhas.

2.1 Computação em Nuvem

A computação em nuvem é um modelo de tecnologia que oferece acesso fácil e sob demanda a um conjunto de recursos de computação compartilhados, como redes, servidores, armazenamento e aplicativos. Esses recursos podem ser disponibilizados rapidamente e com o mínimo de gestão ou interação necessária por parte do usuário, simplificando processos e reduzindo esforços de manutenção (NIST, 2021). De acordo com (SUN, 2009) e (BAUER E.; ADAMS, 2012), este modelo envolve acessar tais recursos computacionais através de uma rede, o que inclui, mas não se limita a, capacidades de rede, armazenamento, servidores e serviços variados. Características comuns das nuvens incluem a alocação flexível de recursos baseada na necessidade do usuário, a possibilidade de acesso através de qualquer dispositivo conectado à internet, e um sistema de cobrança que calcula custos com base na quantidade de recursos consumidos.

De acordo com (MARINESCU, 2018), cinco atributos essenciais definem a computação em nuvem:

- Atendimento sob demanda, que permite aos usuários obter recursos de TI instantaneamente quando necessário, sem atrasos;
- Amplo acesso à rede, garantindo que os serviços estejam disponíveis através de qualquer dispositivo conectado à Internet, como computadores e dispositivos móveis;
- Compartilhamento de recursos, que consiste na utilização de um pool de recursos computacionais distribuídos entre vários usuários;
- Escalabilidade, permitindo que os recursos possam ser facilmente escalados para cima ou para baixo conforme as necessidades dos usuários;
- Serviços sob medida, que oferecem a possibilidade de personalizar recursos e serviços de

acordo com as exigências específicas de cada cliente

2.1.1 Classificação dos Modelos de Computação em Nuvem

Segundo Marinescu (MARINESCU, 2018), os tipos de computação em nuvem em relação à infraestrutura e gestão são a nuvem privada, a nuvem pública, a nuvem híbrida e a nuvem comunitária, os quais são descritas a seguir:

Nuvem privada: a infraestrutura de computação em nuvem é provisionada para utilização exclusiva por uma única organização composta de diversos consumidores (como unidades de negócios). A sua propriedade, gerenciamento e operação podem ser da organização, de terceiros ou de uma combinação mista, que pode está dentro ou fora das instalações da organização (MARINESCU, 2018; BAUER E.; ADAMS, 2012).

Nuvem pública: a infraestrutura de nuvem computacional é provisionada para uso aberto ao público em geral. A sua propriedade, gerenciamento e operação podem ser de uma empresa, uma instituição acadêmica, uma organização do governo, ou de uma combinação mista. Ela fica nas instalações do fornecedor (MARINESCU, 2018; BAUER E.; ADAMS, 2012).

Nuvem híbrida: a infraestrutura de nuvem computacional é uma composição de duas ou mais infraestruturas na nuvem (como as nuvens privadas, comunitárias ou públicas) que permanecem entidades distintas, mas são interligadas por tecnologia padronizada ou proprietária que permite a comunicação de dados e portabilidade de aplicação (como a transferência de processamento para balancemento de carga entre nuvens) (MARINESCU, 2018; BAUER E.; ADAMS, 2012).

Nuvem comunitária: a infraestrutura de computação em nuvem é provisionada para utilização exclusiva por uma determinada comunidade de consumidores de organizações que têm interesses em comum (de missão, requisitos de segurança, políticas, observância de regulamentações). A sua propriedade, gerenciamento e operação podem ser de uma ou mais organizações da comunidade, de terceiros ou de uma combinação mista, e pode estar dentro ou fora das instalações das organizações participantes (MARINESCU, 2018; BAUER E.; ADAMS, 2012).

2.1.2 Classificação dos Serviços Oferecidos pela Computação em Nuvem

Segundo o NIST (NIST, 2021) a computação em nuvem fornece serviços em três níveis de abstração: infraestrutura como serviços (IaaS), a plataforma como serviço (PaaS) e o *software* como serviço (SaaS).

Infraestrutura como Serviço (*Infrastructure as a Service - IaaS*): Este modelo oferece acesso sob demanda a recursos computacionais virtualizados, tais como capacidade de processamento, armazenamento e redes. Os usuários têm a liberdade de implantar e gerenciar softwares diversos, incluindo sistemas operacionais e aplicações, com a flexibilidade de escolher configurações personalizadas para atender às suas necessidades específicas (MARINESCU, 2018; RITTINGHOUSE; RANSOME, 2009).

Plataforma como Serviço (*Platform as a Service - PaaS*): PaaS provê um ambiente de desenvolvimento e hospedagem na nuvem que permite aos desenvolvedores criar, lançar e gerenciar aplicações sem a complexidade de construir e manter a infraestrutura típica associada ao desenvolvimento e lançamento de software. Os desenvolvedores mantêm o gerenciamento de aplicações, com certa capacidade de personalização do ambiente de execução, porém sem se envolverem com as questões de hardware e camadas de software de baixo nível (MARINESCU, 2018; RITTINGHOUSE; RANSOME, 2009).

Software como Serviço (*Software as a Service - SaaS*): No modelo SaaS, os usuários acessam aplicações fornecidas e gerenciadas pelo provedor de serviços na nuvem. Estas aplicações são disponibilizadas através da internet, permitindo o uso em uma variedade de dispositivos com conectividade à rede, tipicamente por meio de uma interface de usuário baseada em navegador web. A infraestrutura e as plataformas subjacentes são totalmente administradas pelo provedor, simplificando a experiência do usuário ao focar somente na utilização do software (MARINESCU, 2018; RITTINGHOUSE; RANSOME, 2009).

2.1.3 Plataforma de Nuvem OpenStack

OpenStack é um conjunto abrangente de projetos de software livre, projetado para criar e gerenciar plataformas de nuvem de grande escala, tanto públicas quanto privadas. Com uma arquitetura modular e um design distribuído, o OpenStack oferece uma solução flexível e escalável para gerenciar recursos computacionais, de armazenamento e de rede, facilitando o gerenciamento de infraestrutura de TI através de interfaces de programação de aplicações (*APIs*)

robustas e mecanismos de autenticação complexos (OPENSTACK, 2021).

2.1.3.1 Arquitetura OpenStack

A arquitetura do OpenStack é composta por uma série de serviços interconectados que funcionam em conjunto para fornecer uma infraestrutura de nuvem versátil. Estes serviços são organizados em módulos que cobrem diversas funções, desde o provisionamento de instâncias de máquina virtual até o gerenciamento de redes e armazenamento. Esta estrutura permite aos administradores e usuários finais controlarem recursos de nuvem complexos de maneira eficiente e em tempo real, através de uma painel de controle unificado ou via programação direta usando as *APIs* do OpenStack (OPENSTACK, 2021).

A Figura 1 mostra a arquitetura mais comum, mas não a única possível, para uma nuvem *OpenStack*.

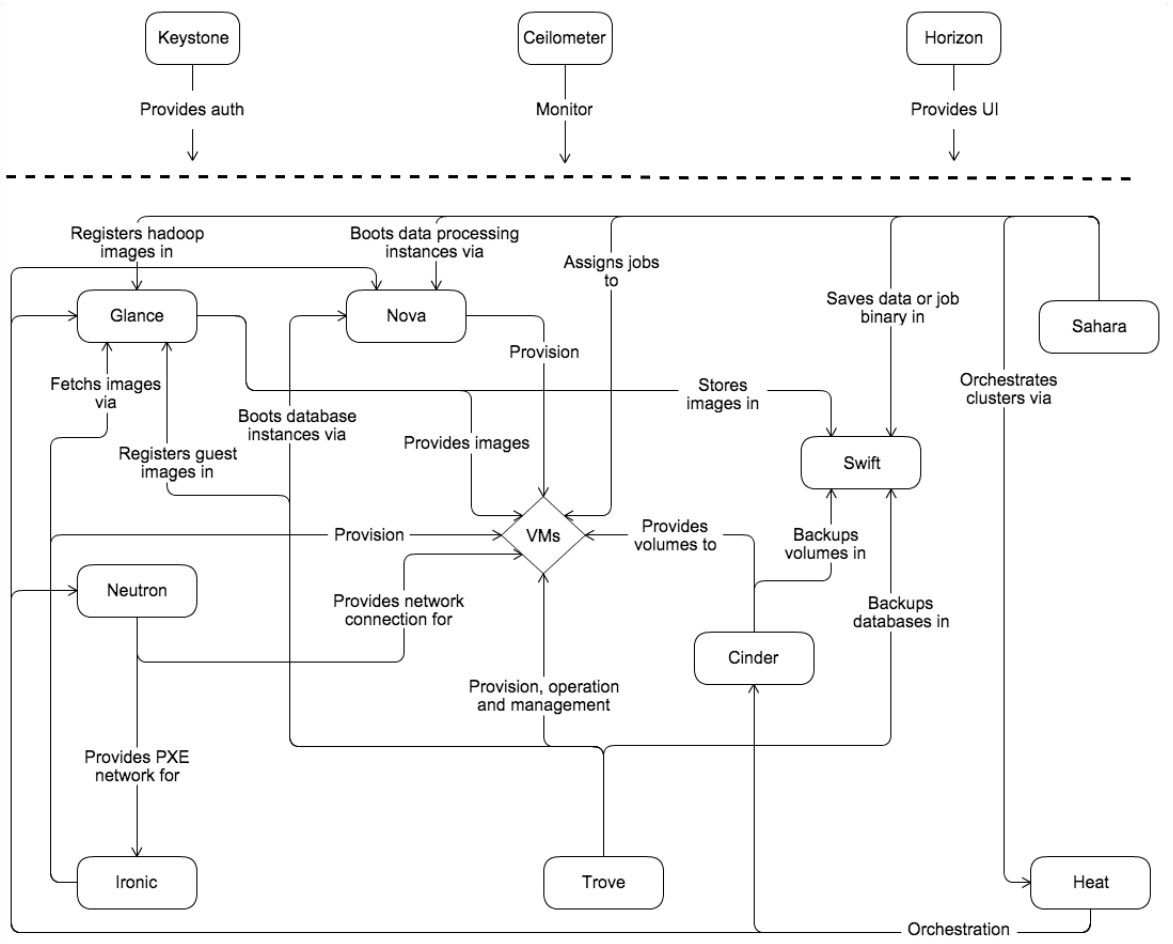

Figura 1 – Arquitetura Conceitual *OpenStack* (OPENSTACK, 2021)

Uma implementação do *OpenStack* pode prover vários serviços por meio de componentes

que fornecem APIs para acessar recursos da infraestrutura. Os componentes podem ser agrupos de acordo com os seguintes serviços: *Compute*, *Hardware Lifecycle*, *Storage*, *Networking*, *Shared Services*, *Orchestration*, *Workload Provisioning*, *Application Lifecycle*, *Telemetry*, *API Proxies* e *Web Frontend* (OPENSTACK, 2021).

- **Compute**
  - **Nova:** é o módulo responsável por prover instâncias de computação (servidores virtuais) (OPENSTACK, 2021).
- **Hardware Lifecycle**
  - **Ironic:** fornecer acesso de autoatendimento sob demanda e altamente escalável para recursos de computação (OPENSTACK, 2021).
- **Storage**
  - **Swift:** é o sistema de armazenamento de objetos e arquivos (OPENSTACK, 2021).
  - **Cinder:** virtualiza o gerenciamento de dispositivos de armazenamento em bloco e fornece aos usuários finais uma *API* de autoatendimento para solicitar e consumir esses recursos (OPENSTACK, 2021).
- **Networking**
  - **Neutron:** é o componente que gerencia a infraestrutura virtual de rede e aspectos da infraestrutura física de rede no ambiente *OpenStack* (OPENSTACK, 2021).
- **Shared Services**
  - **Keystone:** é o componente reponsável pela autenticação e autorização de alto nível para acesso aos serviços da plataforma *OpenStack* (OPENSTACK, 2021).
  - **Glance:** providencia serviços de pesquisa, registro e entrega para imagens de disco e servidor. Ele inclui uma *API REST* que permite consulta de metadados de imagem *VM*, bem como a recuperação da imagem real (OPENSTACK, 2021).
  - **Ceilometer:** é um componente que coleta, normaliza e transforma dados de todos os componentes do *OpenStack*. Seus dados podem ser usados para mensurar uso, rastreamento de recursos e alerta de problemas dos principais componentes da plaforma (OPENSTACK, 2021).
- **Orchestration**
  - **Heat:** orquestra os recursos de infraestrutura para aplicações de nuvem utilizando templates. Ele fornece uma *API REST* nativa do *OpenStack* e uma *API* de consulta compatível com *AWS CloudFormation* (OPENSTACK, 2021).

- **Workload Provisioning**
  - **Trove:** provisiona bancos de dados relacionais e não-relacionais (OPENSTACK, 2021).
  - **Sahara:** provê o provisionamento de infraestruturas de processamento de dados (como *Hadoop*, *Spark* e *Storm*) (OPENSTACK, 2021).
- **Application Lifecycle**
  - **Masakari:** é um componente que provê instâncias para nuvens *OpenStack* com alta disponibilidade e capacidade de recuperação automática em caso de falha (OPENSTACK, 2021).
- **API Proxies**
  - **EC2API:** é o componente que fornece uma *API* compatível com o EC2 (OPENSTACK, 2021).
- **Web Frontend**
  - **Horizon:** provê uma interface gráfica baseada na *web* para os usuários acessarem, provisionarem e automatizarem recursos da nuvem (OPENSTACK, 2021).

2.2 Dependabilidade

Conforme descrito por Boulanger (BOULANGER, 2016), dependabilidade é entendida como a qualidade do serviço que é consistentemente entregue pelo sistema ao usuário, de maneira confiável. Em um contexto complementar, Avizienis et al. (AVIZIENIS et al., 2004) caracterizam dependabilidade como a habilidade de um sistema computacional em prover serviços de forma justa e sem falhas. Para ilustrar esses conceitos, a Figura 2 detalha os componentes que constituem a dependabilidade de um sistema, conforme identificado em (BOULANGER, 2016).

2.2.1 Atributos

Os atributos possibilitam a obtenção de medidas quantitativas, que muitas vezes são cruciais para a análise dos serviços oferecidos.

A disponibilidade é a probabilidade que o sistema esteja operacional durante um determinado intervalo de tempo (MACIEL et al., 2012). Esta probabilidade pode ser dada através

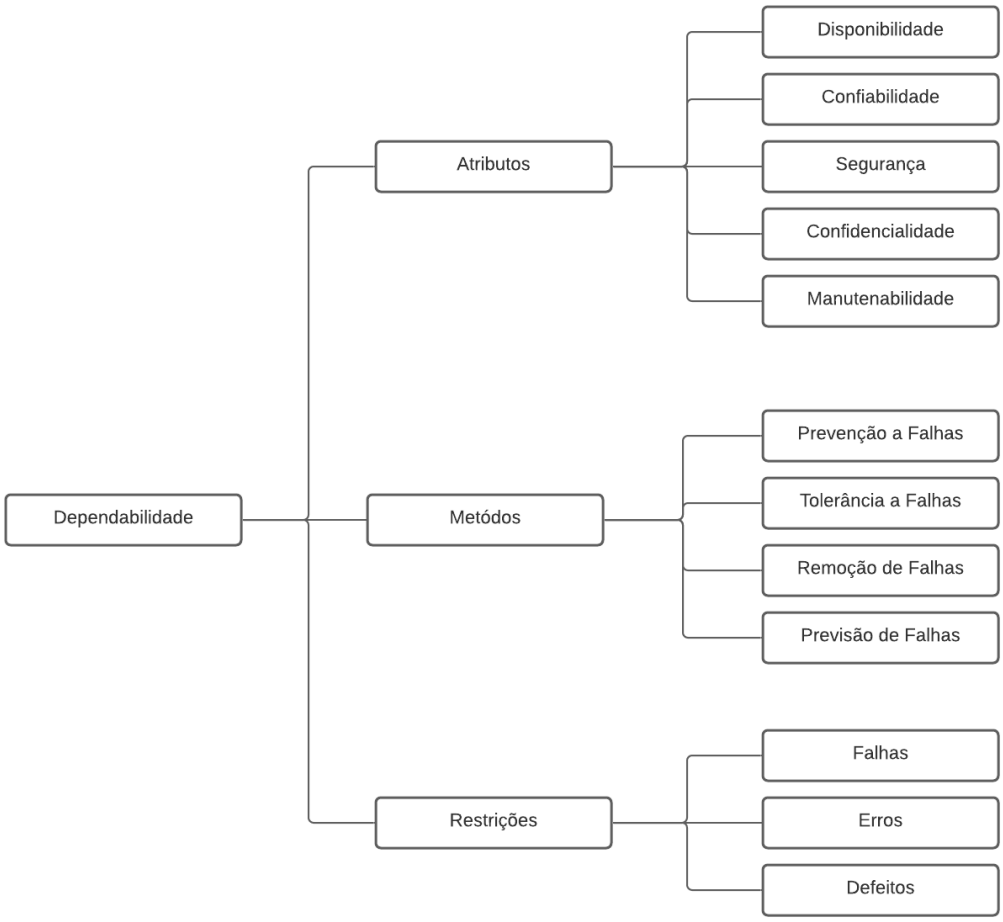

Figura 2 – Árvore de Dependabilidade - Adaptado de (BOULANGER, 2016)

da razão entre o tempo de funcionamento do sistema e a soma entre o tempo de funcionamento e o tempo de falha (MACIEL et al., 2012). A disponibilidade pode ser obtida através da Equação 2.1:

$$A = \frac{E[Uptime]}{E[Uptime] + E[Downtime]} \tag{2.1}$$

onde, **A** é a disponibilidade do sistema, **E[Uptime]** define o tempo em que o sistema está disponível, **E[Downtime]** é o tempo em que o sistema está indisponível.

A Confiabilidade de um sistema é a probabilidade (P) de que esse sistema execute a sua função, de modo satisfatório, sem a ocorrência de defeitos, por um determinado período de tempo (T). A confiabilidade é representada pela Equação 2.2, onde T é uma variável aleatória que representa o tempo para ocorrência de defeitos no sistema (KUO; ZUO, 2003).

$$R(t) = P\{T > t\}, t \geq 0 \tag{2.2}$$

A probabilidade da ocorrência de defeitos até um instante t, é representada pela Equação 2.3 , onde T é uma variável aleatória que representa o tempo para ocorrência de defeitos no sistema (KUO; ZUO, 2003).

$$F(t) = 1 - R(t) = P\{T \leq t\} \tag{2.3}$$

A Equação 2.4 representa a confiabilidade, considerando a função de densidade F(t) do tempo para ocorrência de defeitos (T) no sistema (KUO; ZUO, 2003).

$$R(t) = P\{T > t\} = \int_t^{\infty} F(t)dt \tag{2.4}$$

O tempo médio para falha (*Mean Time to Failure* - MTTF) representa o período médio esperado antes que uma falha ocorra em um sistema. Especificamente, quando a probabilidade de falha do sistema segue uma distribuição exponencial com parâmetro λ, o valor do MTTF pode ser matematicamente expresso pela Equação 2.5, conforme descrito por (KUO; ZUO, 2003).

$$MTTF = \int_0^{\infty} R(t)dt = \int_0^{\infty} exp^{(-\lambda)t} = \frac{1}{\lambda} \tag{2.5}$$

2.2.2 Métodos

Segundo Avizienis et. al. (AVIZIENIS et al., 2004), apesar de o trabalho ter sido publicado há 19 anos, a ideia de que a dependabilidade é influenciada pelo manejo de defeitos e falhas no sistema continua sendo um princípio fundamental. A ocorrência de defeitos e falhas pode afetar adversamente os aspectos de dependabilidade de um sistema. Por isso, as técnicas de prevenção, predição, remoção e tolerância a falhas são essenciais para manter os níveis de dependabilidade, especialmente em sistemas que são críticos para a segurança ou o funcionamento contínuo de serviços essenciais.

Para (AVIZIENIS et al., 2004), a dependabilidade está diretamente ligada ao estudo do efeito de defeitos e falhas no sistema, visto que a ocorrência destes ocasionam um impacto negativo nos atributos de dependabilidade. Téncnicas de prevenção, predição, remoção e tolerância a falhas contribuem para manter níveis desejados de dependabilidade, sobretudo em sistemas críticos.

2.2.3 Restrições

As restrições na dependabilidade são frequentemente caracterizadas por falhas, erros e defeitos, que, apesar de descritos em literatura clássica, mantêm sua relevância.

Um **defeito** refere-se a uma imperfeição no código ou na lógica de um programa de computador, que pode levar o sistema a comportar-se de maneira inesperada ou indesejada (JOHNSON, 1988). Este conceito é fundamental na compreensão de que, mesmo com a evolução tecnológica, o desenvolvimento de software ainda está sujeito a imperfeições inerentes ao processo de sua criação.

Erro, por sua vez, é a manifestação de um defeito no sistema, sendo o estado incorreto gerado quando o sistema executa alguma operação baseada em um defeito existente. Um erro pode ser refletido em resultados imprecisos, processamento incorreto de dados, ou comportamento errático do sistema (JOHNSON, 1988).

A falha é a consequência de um erro que é externamente visível, onde o sistema deixa de cumprir o seu serviço esperado, falhando em entregar o resultado correto ou comportar-se conforme especificado. A falha é, portanto, a interrupção do serviço devido à propagação não mitigada de um erro (JOHNSON, 1988).

2.3 Métodos para Classificação de Defeitos

A classificação de defeitos é um pilar central na garantia da qualidade de *software*. Métodos sistemáticos de identificação e categorização de defeitos fornecem insights valiosos para o aprimoramento do processo de desenvolvimento, contribuindo assim para a produção de sistemas mais confiáveis e robustos.

2.3.1 Classificação Ortogonal de Defeitos (ODC)

A Classificação Ortogonal de Defeitos (*Orthogonal Defect Classification* - ODC) emerge como uma metodologia estruturada que integra tanto a avaliação quantitativa quanto a análise qualitativa de defeitos de *software*. Desenvolvida com o intuito de estabelecer um mecanismo de diagnóstico para defeitos de programação, a ODC proporciona uma abordagem sistemática para a classificação de defeitos, que é essencial para o diagnóstico preciso e a correção eficiente dos mesmos.

Conforme descrito por (CHILLAREGE et al., 1992), a ODC não apenas categoriza defeitos segundo critérios específicos relacionados à natureza do erro, mas também vincula essas categorias ao ciclo de vida do desenvolvimento de *software*. Isso permite que as organizações monitorem a evolução da qualidade do produto ao longo do tempo e identifiquem padrões que podem sinalizar problemas sistêmicos no processo de desenvolvimento.

A aplicação da ODC no contexto do desenvolvimento de *software* possibilita uma compreensão mais profunda sobre as causas raiz dos defeitos, facilitando assim o desenvolvimento de estratégias mais eficazes para a prevenção de falhas futuras. Com a utilização desta metodologia, os times de desenvolvimento podem alocar recursos de maneira mais eficiente, direcionando esforços para as áreas que mais impactam a qualidade e a dependabilidade do sistema (BRIDGE; MILLER, 1998).

Para Ali (ALI, 2019), a *ODC* nos permite categorizar o defeito em classes que apontam para a parte do processo que requer mais atenção. Os defeitos não refletem somente na qualidade do produto, mas também nas deficiências do processo que foi usado no desenvolvimento. Para uma processo *ODC* assertivo, a medição ortogonal é necessária, o que significa dizer que os defeitos coletados não são redundantes e usam atributos para sua classificação. Não obstante, para uma medição eficaz do sistema, é necessário que se atenda a todas as fases do *Software Development Life Cycle* (SDLC).

2.3.1.1 Atributos ODC para Classificação de Defeitos

O desafio para qualquer metodologia de medição de defeito de *software* é identificar um conjunto mínimo de atributos de defeito, a fim de manter a classificação simples, enquanto mapeia completamente todas as atividades do processo de desenvolvimento (BRIDGE; MILLER, 1998).

Segundo (BUTCHER et al., 2002) a metodologia *ODC* está estruturada nas classes, abertas e classes fechadas; as classes por sua vez possuem atributos que denotam a gravidade dos defeitos; associados aos atributos tem-se ações necessárias para representar os defeitos no caso de pertencerem à classe aberta, ou ações tomadas na correção do defeito, nos casos que pertecerem à defeitos da classe fechada. Os atributos e ações associados estão descritos a seguir:

1. **Aberta:** são atributos usados ao se encontrar um defeito. As circustâncias que levaram à exposição de um defeito e o provável impacto para os usuários são em geral conhecidos,

neste caso os defeitos estão relacionados ao *design* e código, classificados em:

- ***Activities:*** usado para identificar a atividade real que estava sendo executada no momento em que o defeito foi descoberto (por exemplo capturar atividades distintas na correção de um defeito como durante a fase de teste usando, *Function Test*, pode-se decidir fazer uma *Code Inspection*).

  Ações: *Design Review*, *Code Inspection*, *Unit Test*, *Function Test* e *System Test*.

- ***Triggers:*** representa o ambiente ou condição que deveria existir para o defeito se materializar.

  Ações: *Design Conformance*, *Logic/Flow*, *Backward Compatibility*, *Lateral Compatibility*, *Concurrency*, *Internal Document*, *Language Dependency*, *Side Effect*, *Rare Situations*, *Complex Path*, *Test Coverage*, *Test Variation*, *Test Sequencing*, *Test Interaction*, *Workload/Stress*, *Recovery/Exception*, *Start up/Restart*, *Hardware Configuration*, *Software Configuration* e *Blocked Test*.

- ***Impact:*** representa o impacto da falha de *software* no serviço prestado ao cliente.

  Ações: *Installability*, *Serviceability*, *Standards*, *Integrity/Security*, *Migration*, *Reliability*, *Performance*, *Documentation*, *Requirements*, *Maintenance*, *Usability*, *Accessibility* e *Capability*.

2. **Fechada:** após a aplicação da correção, a natureza exata do defeito e o escopo da correção são conhecidos.

   - ***Target*:** representa a identidade em alto nível da entidade que foi corrigida. **Ações:** *Design/Code*.

   - ***Defect Type*:** representa a correção real que foi feita.

     Ações: *Assignment/Initialization*, *Checking*, *Algorithm/Method*, *Function/Class/Object*, *Timing/Serialization*, *Interface/O-O Messages* e *Relationship*.

   - ***Qualifier*:** é o qualificador aplicado ao *Defect Type*, e captura o elemento de uma implementação inexistente, errada ou irrelevante.

     Ações: *Missing*, *Incorrect* e *Extraneous*.

   - ***Age*:** refere-se ao estágio (*release*) em que se encontra a correção.

     Ações: *Base*, *New*, *Rewritten* e *ReFixed*.

   - ***Source*:** usado para escolher o perfil que melhor define sua experiência em desenvolvimento, análise de requisitos e projeto.

     Ações: *Developed In House*, *Reused From Library*, *Outsourced* e *Ported*.

2.4 Injeção de Falhas

Segundo Benso e Prinetto (BENSO; PRINETTO, 2010) a injeção de falhas é definida como uma técnica para validar a confiabilidade por meio de experimentos controlados onde se observa o comportamento resultante do sistema após a injeção de falhas artificiais. Essa técnica pode acelerar a ocorrência e a propagação de falhas no sistema para observar os efeitos no desempenho e disponibilidade.

Para Mei-Chen Hsueh et. al. (MEI-CHEN HSUEH et al., 1997) a injeção de falhas é importante para avaliar a confiabilidade dos sistemas de computador, seja por *hardware* ou *software*. O contraste entre os métodos de *hardware* e *software* reside principalmente nos pontos de injeção de falha que eles podem acessar, o custo e o nível de perturbação. Os métodos de *hardware* podem injetar falhas nos pinos do chip e componentes internos, como circuitos combinacionais e registros que não são endereçáveis por *software*. Por outro lado, os métodos de *software* são convenientes para produzir mudanças diretamente no nível do estado do *software*.

A Figura 3 mostra a arquitetura de um injetor de falhas (*fault injector*). Esse sistema é composto por um injetor de falhas que injeta falhas em um sistemas de destino (*target system*), executa comandos do gerador de carga de trabalho (*workload generator*). O monitor rastreia a execução dos comandos e inicia a coleta de dados (*data collector*) sempre que necessário. O coletor de dados (*data collector*) executa a coleta de dados online e o analisador de dados (*data analyzer*), que pode estar offline, executa o processamento e a análise dos dados. O controlador (*controller*) é um programa que pode ser executado no sistema de destino ou em um computador separado. O injetor de falhas (*fault injector*) pode ser um hardware ou software personalizado. O injetor de falhas (*fault injector*) em si pode suportar diferentes tipos de falhas, localizações de falhas, tempos de falhas e semântica de hardware ou estrutura de software apropriadas, cujos valores são extraídos de uma biblioteca de falhas (*fault library*). A biblioteca de falhas na Figura 3 é um componente separado, o que permite maior flexibilidade e portabilidade. O gerador de carga de trabalho (*workload generator*) simular a demanda normal de um sistema ou aplicativo e monitor coleta e analisa métricas e estatísticas durante a execução da carga de trabalho gerada.

2.4.1 Injeção de Falhas Baseadas em *Software*

Nos últimos anos, os pesquisadores têm demonstrado mais interesse no desenvolvimento de ferramentas de injeção por falhas implementadas por *software*. As técnicas de injeção de

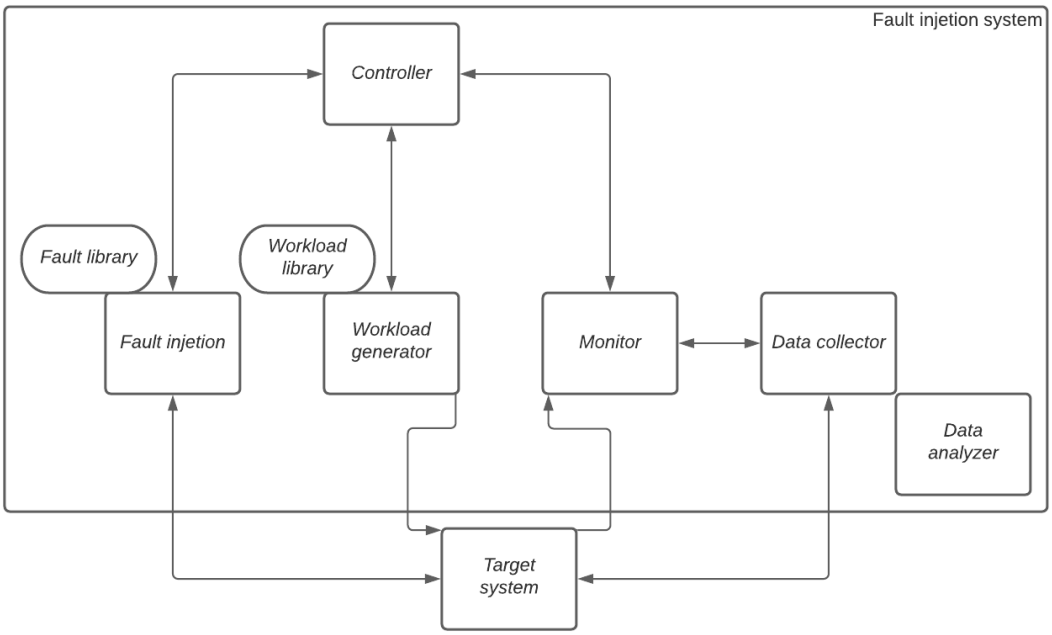

Figura 3 – Arquitetura de um Injetor de Falhas - Adaptado de (MEI-CHEN HSUEH et al., 1997)

falha de *software* são atraentes porque não requerem *hardware* caro. Além disso, elas podem ser usados para direcionar aplicativos e sistemas operacionais, o que é difícil de fazer com a injeção de falha por *hardware* (MEI-CHEN HSUEH et al., 1997).

A injeção de falhas por *software* podem ser categorizadas com base no período em que as falhas são injetadas: durante o tempo de compilação ou durante o tempo de execução (MEI-CHEN HSUEH et al., 1997).

Tempo de compilação: para injetar falhas em tempo de compilação, a instrução do programa deve ser modificada antes que a imagem do programa seja carregada e executada. Esse método injeta erros no código-fonte ou no código do programa de destino para emular o efeito de *hardware*, *software* e falhas temporárias. O código modificado altera as instruções do programa de destino. A injeção gera uma imagem de *software* incorreta e, quando o sistema executa a imagem de falha, ele ativa a falha (MEI-CHEN HSUEH et al., 1997).

Tempo de execução: não exigem a modificação do código fonte do sistema. Em vez disso, este tipo de injetor utiliza algum mecanismo que serve de gatilho para a injeção de falhas. Existem três técnicas para implementar o gatilho que irá disparar as falhas (MEI-CHEN HSUEH et al., 1997):

- ***Time-out*:** esta técnica utiliza um *timer* (que pode ser de *hardware* ou de *software*) para controlar a emissão de falhas. Quando o contador do *timer* chega até zero, uma falha é injetada no sistema;

- **Exceção/trap:** nesta técnica, quando uma certa exceção de *software* ou interrupção de *hardware* ocorre, o controle é transferido para o injetor de falhas. Esta técnica permite a injeção de falhas após a ocorrência de eventos específicos, ação que a injeção por *time-out* não permite.
- **Inserção de código:** nesta técnica, as instruções são adicionadas ao programa de destino que permite que a injeção de falhas ocorra antes de instruções específicas, bem como o método de modificação de código. Ao contrário da modificação de código, a inserção de código executa a injeção de falha durante o tempo de execução e adiciona instruções em vez de alterar as instruções originais.

2.4.2 Injeção de Falhas Baseadas em *Hardware*

A injeção de falha baseada em *hardware* envolve um sistema alvo a ser analisado com *hardware* de teste, especialmente projetado para permitir a injeção de falhas no sistema para examinar os efeitos. Tradicionalmente, essas falhas são injetadas no nível do pino do Circuito Integrado (IC), porque esses circuitos são complexos o suficiente para garantir a caracterização por meio da injeção de falha, e porque essas são as falhas básicas mais bem compreendidas em tais circuitos (BENSO; PRINETTO, 2010).

Segundo(MEI-CHEN HSUEH et al., 1997), dependendo da falha e da localidade onde deve ser inserida, o método de injeção de falhas por hardware pode ser dividido em duas categorias:

Com contato, o injetor tem contato físico com o sistema alvo introduzindo variação de corrente externamente ao chip do sistema em teste.

Sem contato, neste caso o injetor não possui contato físico direto com o sistema em teste. Nesta categoria, falhas são introduzidas por meio de fontes externas que produzem alguns fenômenos físicos como, por exemplo, interferência eletromagnética e radiação de íons.

Independente da categoria utilizada, alguns benefícios em utilizar a técnica de injeção por falhas baseadas em *hardware* são: o acesso a locais que não podem ser acessados por outras técnicas; os experimentos são considerados rápidos; e os testes podem ser executados próximo ao tempo real. O uso desta técnica também acarreta alguns inconvenientes tais como: alto risco de dano no sistema em teste; baixa portabilidade e controle sobre falhas; além de reduzida observação do comportamento do sistema durante testes.

3 Trabalhos Relacionados

Este capítulo oferece uma análise criteriosa dos estudos anteriores que são pertinentes à temática central desta dissertação: a classificação e categorização de defeitos em computação em nuvem e as estratégias de injeção de falhas em tais ambientes. A discussão aqui empreendida está fundamentada em uma revisão sistemática da literatura, destinada a identificar os avanços recentes, as lacunas existentes e as oportunidades para pesquisa futura.

3.1 Metodologia de Pesquisa

A seleção de trabalhos foi conduzida através de uma busca abrangente nas três principais bases de dados científicas do campo de TI: *IEEE Xplore*, *ACM Digital Library* e *SpringerLink*. A pesquisa utilizou combinações de palavras-chave estratégicas, especificamente duas strings de busca: *"orthogonal defect classification"AND "bug classification"AND "software error"* e *"cloud computing"AND "fault injection"AND "dependability evaluation"*. Com o intuito de assegurar uma revisão inclusiva, optamos pela modalidade de busca *Full Text* nos metadados dos artigos.

3.2 Critérios de Seleção

A relevância dos artigos foi aferida com base em critérios de inclusão claramente definidos:

- Publicações em periódicos de alto impacto ou conferências de prestígio na área de tecnologia da informação.
- Trabalhos publicados a partir de 2017, para garantir a atualidade e a relevância das tecnologias e metodologias discutidas.

3.3 Organização do Capítulo

O capítulo está organizado da seguinte forma: Inicialmente, apresentamos uma discussão sobre a Classificação e Categorização de Defeitos na computação em nuvem, seguida de uma seção dedicada à Injeção de Falhas em Ambientes de Nuvem. Posteriormente, fornecemos uma Comparação dos Trabalhos Relacionados, onde as contribuições e limitações dos estudos

selecionados são analisadas de forma comparativa. Esta estrutura visa não apenas a catalogar o conhecimento existente, mas também a estabelecer um diálogo com as pesquisas anteriores, destacando como este estudo contribui para o corpo de conhecimento existente.

A compreensão aprofundada fornecida por esta revisão literária estabelece uma base sólida para a investigação proposta, alinhando-a com o estado da arte e evidenciando sua relevância científica e prática.

3.4 Classificação e Categorização de Defeitos na Computação em Nuvem

Tradicionalmente, a melhoria da qualidade de software tem focado em modelos de desenvolvimento e em práticas de testes extensivos. Contudo, a análise semântica dos defeitos - isto é, o estudo do significado e das consequências dos defeitos - muitas vezes não recebe a atenção devida. A técnica de Classificação Ortogonal de Defeitos (*ODC*), no entanto, destaca-se por sua eficiência e precisão na identificação de defeitos ao focar na semântica das falhas.

Nesta seção, examinamos diversos estudos que aplicaram a metodologia ODC em diferentes cenários de desenvolvimento e manutenção de software. O uso do ODC transcende a simples detecção de defeitos, mergulhando nas causas subjacentes e promovendo um entendimento profundo das raízes dos problemas. Tal abordagem não só identifica defeitos de forma eficaz, mas também fornece insights valiosos para a prevenção de falhas futuras, contribuindo substancialmente para o aprimoramento da robustez do software.

Através dos trabalhos analisados, fica evidente que a aplicação da ODC é versátil, podendo ser integrada tanto nas etapas iniciais de desenvolvimento quanto nas fases de produção. Esta seção detalha como o ODC tem sido utilizado para diagnosticar e remediar falhas, demonstrando sua aplicabilidade e benefícios na engenharia de software.

O trabalho (CHILLAREGE, 2017) apresenta um estudo de defeitos com base na *ODC*, analisando distribuições da *Trigger* e *Impact* em cinco estágios de testes em desenvolvimento e testes com software em produção. A análise dos dados possibilitou a detecção de falhas e suas consequências. Os dados utilizados no trabalho são provenientes do desenvolvimento de um produto de software de alta confiabilidade. O processo utilizado no desenvolvimento do software envolve metodologias ágeis e clássicas. O foco de alta confiabilidade tem camadas adicionais de testes, aumentando assim a contagem de estágios de teste para o propósito do estudo. Isso permitiu analisar características das distribuições *Trigger* e *Impact* em função dos Estágios que

avançam. A classificação completa também inclui alguns atributos adicionais: Atividade, Fase, Idade, Gravidade, Alvo. As falhas que causam alterações no código devido a falhas foram, categorizadas pelo *ODC*.

No estudo conduzido por (AGNELO et al., 2020), foi realizada uma análise de defeitos em bancos de dados *NoSQL* para discernir os mais recorrentes e suas repercussões. O intuito era auxiliar os desenvolvedores na criação de sistemas *NoSQL* mais sólidos e confiáveis. Utilizando a técnica *ODC* para a categorização de defeitos, o procedimento adotado incluiu as seguintes etapas:

- Seleção dos bancos de dados *NoSQL* — *MongoDB*, *Cassandra*, e *HBase* — com base em sua popularidade, conforme rankings de 2017 disponíveis em *DB-Engines*[1], *StackOverkill*[2] e *KDNuggets*[3].
- Capacitação de um pesquisador no uso da metodologia *ODC*, aplicando-a a descrições reais de defeitos retiradas de plataformas de rastreamento de defeitos públicas.
- Análise de 4096 defeitos já resolvidos para avaliar a aplicabilidade dos atributos *ODC* na identificação de padrões de defeitos.

Os achados do estudo revelaram uma ampla gama na distribuição dos tipos de defeitos, sugerindo que a compreensão desta distribuição é fundamental e deveria ser realizada regularmente para determinar quais defeitos são mais representativos. Notou-se também uma recorrência de padrões de defeitos entre os bancos de dados, com uma maior incidência de atividades de teste em defeitos ligados à confiabilidade. Distinções significativas na distribuição dos tipos de defeitos foram evidenciadas entre os diferentes componentes dos sistemas, muitas vezes relacionadas à função específica do componente. Concluiu-se que determinados tipos de defeitos estão consistentemente relacionados a períodos de reparo mais extensos nos três bancos de dados analisados.

Em (KUMAR et al., 2021), os autores propuseram uma metodologia para classificação de defeitos baseada na *ODC* adotando memória de longo prazo (*Long Short Term Memory - LSTM*), que é um tipo de Redes Neurais muito utilizada em tarefas de classificação. A proposta consiste na utilização de algoritmos de Processamento de Linguagem Natural (PNL) fazendo pré-processamento de texto aplicado à descrição dos relatórios de defeitos, removendo a pontuação, caracteres especiais e *stop words*. O método proposto supera a abordagem clássica, como

[1] https://dbengines.com/
[2] https://stackoverkill.com/
[3] https://kdnuggets.com/

bag of words e modelos de classificação baseados em *Term Frequency – Inverse Document Frequency* (TF-IDF). A abordagem proposta baseada em LSTM obteve melhores resultados que a abordagem baseada em TF-IDF na tokenização em 34% , 24,7% para precisão na representação de palavras semelhantes e 22,4% para recall.

No trabalho proposto por (HU; LIU, 2021), é apresentada a concepção de um sistema de medição multidimensional baseado na Classificação Ortogonal de Defeitos (*ODC*). O estudo aponta limitações da *ODC* devido ao seu foco em aspectos semânticos abstratos, argumentando que isto restringe sua aplicabilidade em Sistemas Intensivos de Software (*SIS*). A pesquisa propõe o desenvolvimento de uma ontologia para:

- Analisar a duração e as características dos erros de software com um olhar da engenharia de software.
- Propor um modelo de geração de erros, o qual revisita e expande os conceitos de padrões de erros de software (*SEP*) e de padrões de erros de requisitos de software (*SREP*) através da lente da *ODC*.
- Contemplar os tipos de erros que ocorrem na interação entre software e hardware (*SHIEP*), essenciais na fase de requisitos.

Neste modelo, são introduzidos quatro tipos de *SHIEP* durante a fase de requisitos, acompanhados da representação ontológica correspondente. Os resultados indicaram que o conhecimento prévio dos *SHIEPs* é eficaz para identificar potenciais defeitos e falhas que podem comprometer a função ou desempenho dos *SISs*. Dessa forma, a abordagem *SHIEP* proposta é de significativa relevância para aprimorar a qualidade do desenvolvimento e da verificação de software. No entanto, o alcance de aplicação dessa metodologia ainda é considerado restrito e os dados experimentais disponíveis são limitados.

3.5 Injeção de Falhas em Ambiente de Nuvem

A injeção de falhas emerge como uma técnica fundamental para avaliar a resiliência e a robustez de softwares distribuídos, especialmente em sistemas complexos hospedados em ambientes de nuvem. Esta seção realiza um exame criterioso dos avanços recentes na literatura concernente a esta prática. Destaca-se uma tendência entre os pesquisadores de criar injetores de falhas que são meticulosamente adaptados para atender às necessidades específicas de seus estudos. Este enfoque personalizado reflete a diversidade e a especificidade dos cenários

operacionais na nuvem. A revisão dos métodos de injeção de falhas apresentada a seguir fornece uma perspectiva sobre os diferentes objetivos de estudo, as técnicas utilizadas e os insights obtidos, delineando um panorama dos desafios e soluções propostos na atualidade para a engenharia de software na nuvem.

Em (TIAN et al., 2018), é introduzida a ferramenta **ESIFT** (*Efficient System for Error Injection*), uma ferramenta avançada projetada para injeção de erros em sistemas de software. Construída sobre a base do *GNU Project Debugger* (GDB) e implementada em Python, a **ESIFT** se beneficia de uma notável portabilidade, podendo ser utilizada em uma grande variedade de sistemas operacionais. Um dos diferenciais do **ESIFT** reside em sua capacidade de operar em sincronia com o sistema operacional, permitindo, assim, uma avaliação precisa da confiabilidade em sistemas de grande escala.

Diferentemente das ferramentas tradicionais de injeção de falhas, que geralmente se limitam a verificar o comportamento de sistemas após a ocorrência de falhas, o **ESIFT** é capaz de avaliar tanto o comportamento padrão do sistema quanto sua resposta durante e após a ocorrência de erros. Essa característica dual é especialmente útil para medir a latência na detecção de falhas e a extensão de sua propagação dentro do sistema.

Graças à integração com o GDB, a **ESIFT** também é capaz de realizar análises em cenários complexos, como aplicações que operam com múltiplas threads ou em ambientes distribuídos. Experimentos realizados utilizando a ferramenta com quatro *benchmarks SPEC2000* mostraram a aplicabilidade do **ESIFT** em condições reais de teste.

No entanto, a ferramenta possui algumas limitações inerentes à sua dependência do GDB. As localizações de injeção de falhas que o **ESIFT** pode acessar estão confinadas aos registros e áreas de memória que são visíveis ao GDB, isto é, áreas acessíveis pelo usuário. Isso significa que determinados tipos de falhas de hardware, que requerem acesso a outros níveis do sistema, não podem ser simulados pelo **ESIFT**. Além disso, a dependência em relação ao GDB pode representar um obstáculo para a realização de experimentos de injeção de falhas em plataformas que não suportam o GDB, limitando assim o escopo de aplicação da ferramenta.

A ferramenta *FailViz*, detalhada no trabalho (COTRONEO et al., 2019), representa um avanço significativo na área de análise de falhas em sistemas distribuídos, oferecendo uma plataforma poderosa para visualizar e entender as falhas injetadas em tempo de execução. Esta ferramenta é projetada para ser instalada no sistema antes do início dos experimentos de injeção de falhas. Sua funcionalidade principal é capturar e registrar as mensagens trocadas entre os

componentes do sistema durante a injeção de falhas.

O processo de análise começa com o *FailViz* monitorando as mensagens em tempo real, identificando e registrando seus atributos significativos. Quando uma falha é injetada, a ferramenta compara dinamicamente as mensagens atuais com aquelas de um sistema que opera corretamente, sem falhas. Esta comparação permite que o *FailViz* destaque discrepâncias e padrões anormais, apresentando ao analista eventos que possam indicar causas ou consequências da falha.

Para rastrear a comunicação dentro do sistema, o *FailViz* utiliza técnicas avançadas de rastreamento distribuído, que são capazes de interceptar e registrar chamadas de *APIs* sem necessitar alterações diretas no código-fonte ou nos binários. A instalação dos chamados probes — pontos de rastreamento — é feita de maneira não intrusiva, seguindo o princípio do "*black-box tracing*". Isso significa que não é necessário ter conhecimento prévio sobre as *APIs* de comunicação específicas do sistema em análise, facilitando a utilização da ferramenta em uma ampla gama de aplicações distribuídas.

Desta forma, o *FailViz* se destaca como uma solução eficiente e versátil para os especialistas em confiabilidade de sistemas que buscam compreender melhor o comportamento do sistema sob condições de falha, contribuindo assim para o desenvolvimento de sistemas mais robustos e confiáveis.

ProFIPy é uma solução inovadora no campo da engenharia de software, projetada especificamente para o teste de resiliência de aplicações por meio da injeção de falhas. Esta ferramenta, que é descrita em detalhe em (COTRONEO et al., 2020), emprega uma linguagem de domínio específico (DSL) para criar uma série de cenários experimentais. Esses experimentos são convenientemente armazenados como arquivos *JSON*, facilitando o manuseio e a reutilização de dados.

O cerne da ferramenta é um mecanismo que converte especificações DSL em um injetor de falhas automatizado. Isso permite que os usuários não só salvem seus modelos de falhas personalizados para uso futuro, mas também importem modelos já existentes. Com esses modelos, o ProFIPy executa o injetor contra o software alvo, gerando variantes mutantes do programa original. Essas variantes são então submetidas a testes em um ambiente de execução controlado para coleta de dados analíticos.

Para simplificar ainda mais o processo de teste para os usuários, ProFIPy é disponibilizado como um serviço baseado em nuvem (SaaS). Ele oferece um suporte abrangente, incluindo

a configuração de carga de falhas, a análise de dados resultantes e a automação total dos experimentos. A ferramenta utiliza tecnologias de virtualização avançadas, como contêineres Docker, para paralelizar as tarefas e maximizar a eficiência.

O fluxo de trabalho começa com a criação de uma imagem Docker personalizada, que contém o código-fonte enviado pelo usuário. Para cada falha que precisa ser injetada, o ProFIPy inicia um novo contêiner Docker, dentro do qual o código mutado é testado contra uma carga de trabalho específica. Este processo é governado por parâmetros definidos pelo usuário, incluindo um tempo limite para cada teste. Após a conclusão dos testes ou o término do tempo estipulado, o ProFIPy automaticamente desmonta o ambiente de teste, garantindo uma limpeza eficaz e deixando o sistema pronto para a próxima série de experimentos.

ProFIPy se destaca como uma ferramenta altamente flexível e funcional, projetada para auxiliar profissionais na avaliação precisa da robustez do software, conduzindo-os através de um ciclo de testes abrangente e automatizado, destinado a reforçar a qualidade e a confiabilidade das aplicações de software.

Particionamentos de rede são problemas comuns e desafiadores em sistemas de nuvem de grande escala. Eles ocorrem quando falhas na rede causam a fragmentação do sistema, impedindo a comunicação entre grupos distintos de nós. Isso pode levar a inconsistências e falhas no sistema. A ferramenta **CoFI**, discutida em (CHEN et al., 2020), oferece uma abordagem sistemática para injeção de falhas em partições de rede, visando simular e estudar os efeitos desses tipos de falhas de maneira mais eficaz.

O método inovador por trás do **CoFI** envolve a injeção de falhas de forma controlada e inteligente em sistemas de nuvem distribuídos. O foco é reproduzir cenários onde ocorrem estados consistentes no sistema, o que significa que a ferramenta tem a capacidade de reconhecer e marcar automaticamente esses estados em diferentes plataformas de nuvem.

Uma vez identificados esses estados consistentes, **CoFI** monitora ativamente o sistema em tempo real. Quando detecta um estado que poderia levar a uma inconsistência, a ferramenta desencadeia propositadamente um particionamento de rede. Essa atuação criteriosa permite analisar o comportamento do sistema frente a falhas específicas.

Mais do que simplesmente criar particionamentos aleatórios, o **CoFI** leva em consideração os tipos de mensagens que estão sendo transmitidas pelo sistema. Ao se concentrar em mensagens críticas e em pontos de falha específicos, a ferramenta pode explorar de maneira mais eficiente os momentos e condições em que a partição deve ser introduzida.

Essa abordagem metódica torna o teste de sistemas em nuvem substancialmente mais eficiente do que técnicas que podem ser excessivamente amplas ou aleatórias. Ao evitar interrupções desnecessárias para cada tipo de mensagem, o **CoFI** permite uma avaliação mais precisa e focada dos mecanismos de recuperação e tolerância a falhas do sistema.

Em suma, o **CoFI** é uma ferramenta projetada para enriquecer o processo de teste de sistemas em nuvem, oferecendo uma estratégia sofisticada para a injeção de falhas em partições de rede e permitindo aos desenvolvedores e pesquisadores identificar e corrigir vulnerabilidades de maneira eficiente e sistemática.

3.6 Comparação dos Trabalhos Relacionados

A Tabela 1 a seguir resumem as principais contribuições desta pesquisa as relacionam aos trabalhos apresentados.

Os trabalhos relacionados *Trigger and impact profiles enable outcome focused defect discovery to manage software quality* ((CHILLAREGE, 2017)) (A1), *Using orthogonal defect classification to characterize nosql database defects* ((AGNELO et al., 2020)) (A2), *Bug report classification into orthogonal defect classification defect type using long short term memory* ((KUMAR et al., 2021)) (A3), *Orthogonal defect classification-based ontology construction and application of software-hardware integrated error pattern of software-intensive systems* ((HU; LIU, 2021)) (A4), *Efficient system for error injection* ((TIAN et al., 2018)) (A5), *Failviz: A tool for visualizing fault injection experiments in distributed systems* ((COTRONEO et al., 2019)) (A6), *Profipy: Programmable software fault injection as-a-service* ((COTRONEO et al., 2020)) (A7) e *Cofi: Consistency-guided fault injection for cloud systems* ((CHEN et al., 2020)) (A8) foram analisados em relação a Esta Dissertação, considerando as características P1 até P5.

As características escolhidas para comparar os trabalhos são cruciais para entender a evolução e a eficácia das metodologias de identificação e gestão de defeitos em sistemas de software. A Classificação Ortogonal de Defeitos (*ODC*) (P1) é uma dimensão fundamental pois oferece uma taxonomia robusta para categorizar e analisar os defeitos, o que é vital para a melhoria contínua da qualidade do software. A criação de uma metodologia para classificação de defeitos (P2) é essencial para estabelecer um processo sistemático e replicável, garantindo consistência e precisão na identificação de defeitos. A análise de defeitos ainda abertos (P3) é um fator crítico que influencia diretamente a estabilidade e confiabilidade do software, destacando

áreas que exigem atenção imediata e melhorias. O desenvolvimento de uma ferramenta para injeção de falhas (P4) permite a simulação de condições adversas em um ambiente controlado, o que é crucial para testar a robustez e a resiliência do software. Por fim, a injeção de falhas na nuvem (P5) é particularmente relevante no contexto atual, onde serviços em nuvem dominam a indústria, e a capacidade de manter a qualidade do serviço em face de falhas distribuídas é um diferencial competitivo. Estas características foram escolhidas por sua relevância direta para a garantia da qualidade e para a capacidade de um sistema resistir e se recuperar de falhas, refletindo as demandas e desafios enfrentados pela indústria de software moderna.

Tabela 1 – Principais contribuições dos trabalhos relacionados

Trabalho	P1	P2	P3	P4	P5
Trigger and impact profiles enable outcome focused defect discovery to manage software quality (CHILLAREGE, 2017) (A1)	✓	✓	✗	✗	✗
Using orthogonal defect classification to characterize nosql database defects (AGNELO et al., 2020) (A2)	✓	✗	✗	✗	✗
Bug report classification into orthogonal defect classification defect type using long short term memory (KUMAR et al., 2021) (A3)	✓	✗	✗	✗	✗
Orthogonal defect classification-based ontology construction and application of software-hardware integrated error pattern of software-intensive systems (HU; LIU, 2021) (A4)	✓	✓	✗	✗	✗
Efficient system for error injection (TIAN et al., 2018) (A5)	✗	✗	✗	✓	✗
Failviz: A tool for visualizing fault injection experiments in distributed systems (COTRONEO et al., 2019) (A6)	✗	✗	✗	✓	✗
Profipy: Programmable software fault injection as-a-service (COTRONEO et al., 2020) (A7)	✗	✗	✗	✓	✗
Cofi: Consistency-guided fault injection for cloud systems (CHEN et al., 2020) (A8)	✗	✗	✗	✓	✓
Este trabalho	✓	✓	✓	✓	✓

Dentre os estudos analisados, observa-se que apenas dois apresentam uma metodologia estruturada para a classificação de defeitos, uma lacuna significativa no contexto da gestão de

qualidade de software. Surpreendentemente, não se identificou nenhum trabalho anterior que abordasse a categorização de defeitos ainda não solucionados, os chamados defeitos abertos, revelando uma oportunidade crítica de pesquisa e aplicação. Ademais, enquanto quatro estudos propuseram ferramentas de injeção de falhas para testar a robustez de sistemas de software, apenas uma dessas ferramentas foi especificamente adaptada para ambientes de nuvem privada.

Essa dissertação pretende avançar na pesquisa e prática da garantia de qualidade, propondo uma nova metodologia para classificar defeitos abertos em ambientes de nuvem privada. Esta metodologia será embasada na Classificação Ortogonal de Defeitos (*ODC*), oferecendo uma abordagem sistemática e replicável. Além disso, este trabalho incluirá o desenvolvimento de uma ferramenta inovadora destinada à injeção de falhas, projetada especificamente para ambientes de nuvem privada. Esta ferramenta não só permitirá a simulação de falhas para testar a resiliência de sistemas em nuvem, mas também contribuirá para a detecção precoce e a correção de defeitos, melhorando a confiabilidade e a segurança dos sistemas de software na nuvem.

4 Metodologia para Classificação de Defeitos em Nuvens Privadas

Este capítulo descreve em detalhe a metodologia desenvolvida para a classificação sistemática de defeitos em ambientes de nuvem privada. A estrutura do capítulo é composta pelas seguintes seções: visão geral, análise exploratória, aplicação do modelo *ODC*, síntese conclusiva e considerações finais.

4.1 Visão Geral

A metodologia aqui apresentada tem como objetivo otimizar o entendimento e a correção de defeitos em plataformas de nuvem privada, empregando os princípios da Classificação Ortogonal de Defeitos (*ODC*). Esta dissertação foca nos defeitos ainda não resolvidos — conhecidos como defeitos abertos — cujas condições de ocorrência e os impactos percebidos pelos usuários são previamente identificados. A metodologia adota os seguintes atributos do *ODC*: *activity*, *trigger* e *impact*. Estruturada em três grandes fases: **Análise Exploratória**, **Aplicação do Modelo *ODC***, e **Síntese Conclusiva**, cada uma produzindo artefatos relevantes como resultado das atividades realizadas.

A fase de **Análise Exploratória** envolve quatro etapas principais: a comparação das plataformas de computação em nuvem, a seleção de repositórios de defeitos, o levantamento de defeitos por componente e a criação de um *dataset* inicial. Seguindo, a fase de **Aplicação do Modelo *ODC*** é dividida nas etapas de aplicação do atributo *activity*, determinação do *trigger* e avaliação do *impact*. Finalmente, a fase de **Síntese Conclusiva** é dedicada à definição do tipo de defeito (*defect type*).

As Figuras 4 e 5 ilustram a estrutura e os componentes da metodologia proposta para a classificação de defeitos em plataformas de nuvens privadas. Nelas, cada atividade da metodologia de classificação é representada, seguindo a sequência lógica indicada pelas conexões. Os artefatos ilustram os resultados alcançados ao final de cada atividade, e os elementos de decisão direcionam a progressão das etapas. As conexões de fluxo interligam as atividades e decisões, proporcionando uma visão coesa do processo de classificação.

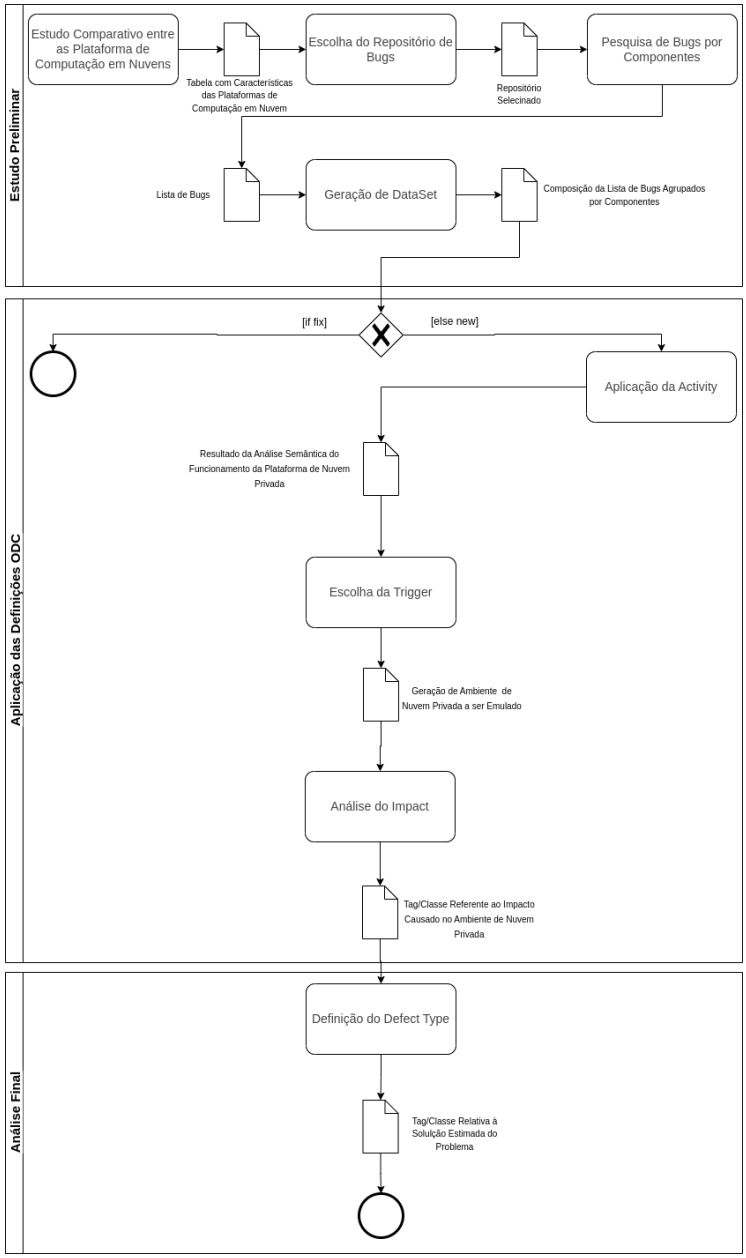

Figura 4 – Metodologia para Classificação de Defeitos em Plataformas de Nuvens Privadas, representada utilizando a notação BPMN (Business Process Model and Notation) (FONTE: Próprio Autor).

Figura 5 – Elementos da Metodologia Proposta

4.2 Análise Exploratória

Esta seção apresenta uma sequência de quatro atividades e quatro artefatos que constituem a **análise exploratória**, são elas: a atividade estudo comparativo entre as plataformas de computação em nuvem que implica na criação do artefato tabela com características das

plataformas de computação em nuvem; em seguida a atividade de escolha do repositório de *bugs* onde o artefato repositório selecionado é definido; após, é executada a atividade de pesquisa de defeitos por componentes a qual gera uma lista de defeitos como o artefato resultante; por fim a atividade de geração do *dataset*, onde uma composição da lista de defeitos agrupados por componentes, criando o *dataset*. Detalhes sobre cada atividade serão apresentados nas subseções a seguir. O objetivo é apresentar insumos necessários para a **aplicação do modelo *ODC***.

4.2.1 Estudo Comparativo Entre as Plataformas de Computação em Nuvem

Para fundamentar a análise atual das plataformas de nuvem privada, é essencial reconhecer os desenvolvimentos passados do setor. Uma década atrás, as plataformas que dominavam o mercado devido ao seu alto nível de maturidade e aceitação incluíam *OpenStack*, *CloudStack*, *OpenNebula* e *Eucalyptus* (BROCKMEIER, 2012). Essas soluções se estabeleceram como players significativos no ecossistema de nuvem de código aberto.

Um estudo da época, representado graficamente na Figura 6 e delineado por (LLORENTE, 2013), ilustrava um quadrante que avaliava esses principais players. A avaliação era baseada em duas dimensões críticas: o *Modelo Cloud*, que abordava a abrangência e a abordagem de serviço de cada plataforma, e a *Flexibility*, relacionada à capacidade de adaptação dos produtos aos serviços de data centers e à customização para fornecer serviços de cloud distintos. A dimensão de flexibilidade, em particular, destacava a importância da adaptabilidade do produto, variando de baixo a alto.

É importante notar que, enquanto essas plataformas foram pioneiras e formaram a espinha dorsal da infraestrutura de nuvem privada na época, a paisagem tecnológica evoluiu significativamente. As considerações atuais sobre as plataformas de nuvem privada podem ser diferentes, levando em conta os avanços tecnológicos, as mudanças no mercado e as novas exigências dos usuários. A análise contemporânea deve, portanto, ser contextualizada com as tendências atuais e futuras, ajustando o entendimento das capacidades atuais das plataformas em um mercado que continua a se expandir e a inovar.

Uma das principais características da plataforma *Eucalyptus* é a implementação de *API's* que permitem a interoperabilidade com serviços baseados na AWS (MATOS et al., 2012).

OpenNebula é uma solução de nuvem que se destaca particularmente em ambientes acadêmicos e de pesquisa, assim como em organizações corporativas, incluindo aquelas que

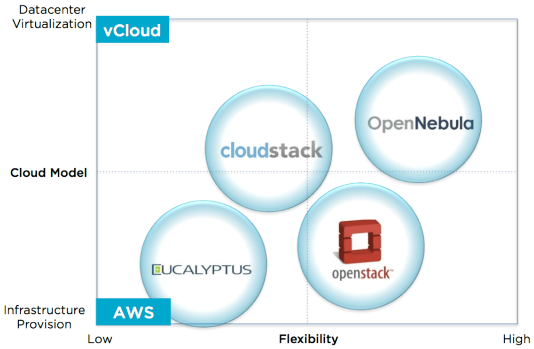

Figura 6 – Classificação das plataformas de Computação em Nuvem de acordo com suas características (LLORENTE, 2013)

gerenciam grandes data centers. Esta plataforma é reconhecida por sua capacidade de oferecer uma infraestrutura de nuvem que não apenas é aberta e adaptável, mas também escalável, atendendo às necessidades de crescimento e expansão de instituições que realizam pesquisas intensivas em dados. Além disso, o *OpenNebula* permite que essas organizações tirem proveito da virtualização de máquinas (VM) e otimizem sua infraestrutura de TI já existente. Este aproveitamento maximiza o retorno sobre os investimentos existentes em TI e proporciona uma proteção significativa contra o risco de ficar dependente de um único fornecedor de tecnologia, uma preocupação conhecida como 'vendor lock-in'. Portanto, o *OpenNebula* é especialmente adequado para instituições que buscam uma solução de nuvem que alie flexibilidade, eficiência e autonomia (ZHAO et al., 2014).

O *CloudStack* não possui o mesmo grau de integração com os serviços da *AWS* como a plataforma *Eucalyptus*. É possível integrar a plataforma com provedores de armazenamento de terceiros por meio de *plug-ins* nativos para *Amazon Simple Storage Service* (S3) ou *OpenStack Object Storage* (Swift).

O *OpenStack* possui o maior número de recursos (ver Tabela 2), e uma solução integrada de armazenamento baseado em objetos e módulos específicos para cada função.

A Tabela 2 oferece uma análise comparativa detalhada entre as principais plataformas de computação em nuvem privada, focando nos tipos de recursos que são essenciais nesses ambientes. Entre as opções comparadas, o OpenStack se destaca por oferecer um conjunto de serviços mais completo. Ele abrange soluções extensivas para computação, redes e armazenamento. Isso significa que, além de fornecer capacidades robustas de processamento e gerenciamento de dados, o OpenStack também entrega uma infraestrutura de rede virtualizada e opções de armazenamento versáteis, que são críticas para suportar uma variedade de cargas de trabalho e aplicações em ambientes de nuvem privada (JOSHI; SHAH, 2019).

Tabela 2 – Comparação Entre Plataformas de Computação em Nuvem (JOSHI; SHAH, 2019)

Resource type	***OpenStack***	***CloudStask***	***OpenNebula***	***Eucalyptus***
Virtualization	KVM, vCenter, XenServer	XenServer, KVM, vCenter, Hyper-V	KVM, vCenter, Xen	KVM, vCenter, Xen
Containerization	✓	✓	✓	✓
Bare Metal	✓	✗	✗	✗
Preemptive VM	✗	✗	✗	✗
Migration suport	✓	✓	✓	✓
Block storage	✓	✓	✓	✓
Cold storage	✓	✗	✗	✗
Object storage	✓	✓	✓	✓
VPN	✓	✓	✓	✓
CDN	✓	✗	✗	✗

4.2.2 Escolha do Repositório de Bugs

A escolha do repositório de bugs para este estudo é uma decisão estratégica que depende diretamente da plataforma de gerenciamento de nuvem escolhida e da distribuição Linux que serve de base para a configuração. Entre as opções disponíveis, o *Launchpad*[1], administrado pela Canonical Ltd e em operação desde 2004, é notável por sua ampla coleção de bugs relacionados ao OpenStack em ambientes Ubuntu Linux. Esta característica o torna uma fonte primária para análise, devido à sua relevância e ao volume significativo de dados sobre defeitos na configuração especificada.

Por outro lado, o *Bugzilla*[2], gerido pela Mozilla Foundation desde 1998, possui um repositório rico em registros de defeitos para plataformas OpenStack implantadas em distribuições Linux derivadas do Red Hat. A extensa história e a diversidade de defeitos reportados em Bugzilla fornecem um contexto valioso para compreender as falhas em ambientes com configurações distintas.

Adicionalmente, o *Atlassian Jira*, da Apache Software Foundation, é reconhecido por

[1] https://launchpad.net/
[2] https://www.bugzilla.org/

hospedar uma gama de defeitos associados ao CloudStack[3]. O detalhamento e a organização dos bugs neste repositório são especialmente úteis para analisar plataformas gerenciadas sob o CloudStack.

Levando em consideração a sinergia entre os repositórios de bugs e as plataformas e configurações de interesse para esta pesquisa, a escolha será orientada por aquele que oferecer a mais rica fonte de dados alinhados com os critérios de seleção definidos para o estudo em questão (ANBALAGAN; VOUK, 2009).

4.2.3 Pesquisa de Defeitos por Componentes

Nos repositórios de bugs, a interface geralmente inclui uma função de pesquisa que permite aos usuários inserir a informação desejada para encontrar problemas específicos. Devido à vasta quantidade de projetos que esses repositórios abrangem, a pesquisa é frequentemente estruturada com o uso de um formato padronizado: ***XXXX-NomeDoComponente***, em que ***XXXX*** representa o nome da plataforma ou software em questão, seguido pelo nome específico do componente em análise, conforme ilustrado na Tabela 3.

O resultado de uma busca é comumente apresentado em um formato tabular, proporcionando uma visualização clara dos dados. Esta tabela contém várias colunas com etiquetas informativas que são cruciais para o agrupamento e análise dos defeitos. Os campos incluem o *bug id*, que é um identificador único geralmente composto por uma sequência numérica inteira, e o *summary*, um campo textual que fornece uma descrição concisa do problema. Adicionalmente, a coluna *changed* registra a data de submissão ou a última alteração do bug, enquanto a *priority* indica a urgência com que o defeito deve ser tratado. Esses e outros rótulos desempenham um papel fundamental na organização e priorização dos defeitos para análises subsequentes.

4.2.4 Geração do *DataSet*

Uma vez completada a coleta e o processamento dos dados para isolar os defeitos de acordo com as informações pertinentes, é viável exportar esses dados consolidados em diversos formatos digitais. Formatos como XML e CSV são comuns e facilitam a transição dos dados

[3] https://issues.apache.org/jira/secure/Dashboard.jspa

Tabela 3 – Exemplo de *TAG's*

Componente	***TAG de Pesquisa***
Nova	openstack-nova
Neutron	openstack-neutron
Keystone	openstack-keystone
Glace	openstack-glace
Cinder	openstack-cinder

para a composição de um *DataSet* estruturado. Esta etapa é crucial para preparar os dados para análises futuras e garantir sua fácil manipulação e acessibilidade em diferentes plataformas de processamento de dados.

4.3 Aplicação do Modelo *ODC*

Esta seção está dividida em três atividades: aplicação da *activity*, escolha da *trigger* e análise do *impact*. Objetiva-se utilizar todo conhecimento adquirido na seção anterior para **aplicação das definições *ODC***.

4.3.1 Aplicação da *Activity*

Com base no entendimento da plataforma apresentada anteriormente, a atividade de **aplicação da *activity*** equivale a associar uma classe relativa ao atributo *Activity* da *ODC* que represente a ação necessária para melhoria, verificação e/ou correção de algum *bug* identificado, tais como: *Design Review*, *Code Inspection*, *Unit Test* e *Function Test* (CHILLAREGE et al., 1992). É importante ressaltar que a aplicação da *Activity* na metodologia proposta se refere a defeitos não solucionados, ou seja, definidos como *new* nos repositórios de *bugs*. De acordo com a *ODC*, são quatro as classes possíveis que podem ser atribuídas:

- *Design Review:* para a revisão do design ou comparar o design documentado com os requisitos conhecidos.
- *Code Inspection:* no exame do código ou comparando o código com o design documentado.
- *Unit Test:* para testes caixa branca ou execução com base no conhecimento detalhado dos componentes internos do código.

- *Function Test:* na execução de testes caixa preta com base em especificações externas de funcionalidade.
- *System Test:* em testes ou execução do sistema completo, em ambiente real, exigindo todos os recursos.

4.3.2 Escolha da *Trigger*

Após a aplicação da *Actvity*, tem-se um subconjunto de atributos denominados de *Trigger*, associados exclusivamente a uma *Activity*, de acordo com a Tabela 4. A *Trigger* tem a finalidade de representar o ambiente ou condição que levou à exposição do *bug*, o que possibilita a recriação de cenários. Por exemplo, na definição da ***Activity***, *Design review / Code Inspection*, escolhe-se a *Trigger* que melhor representa o defeito. Para os demais casos, é ideal que se combine a descrição do defeito com o ambiente ou condição no qual ocorreu o defeito.

A seleção da *Trigger* é feita equiparando às caracteristicas do *bug* com descrição da *Trigger*. Conforme *ODC*, suas descrições são (CHILLAREGE et al., 1992):

- *Design Conformance:* Detecta o defeito ao comparar o elemento de design ou segmento de código que está sendo inspecionado com sua especificação. Isso inclui documentos de design, código, práticas de desenvolvimento e padrões, ou para garantir que os requisitos de design não estejam ausentes ou sejam ambíguos.
- *Logic/Flow:* Usa o conhecimento das práticas e padrões básicos de programação para examinar o fluxo da lógica ou dos dados para garantir que estejam corretos e completos.
- *Backward Compatibility:* O inspetor/aplicador usa ampla experiência em produto/componente para identificar uma incompatibilidade entre a função descrita pelo documento de design ou o código e as versões anteriores do mesmo produto ou componente.
- *Internal Document:* Há informações incorretas, inconsistência ou incompletude na documentação interna. Prólogos e comentários de código representam alguns exemplos de documentação que cairia nesta categoria.
- *Lateral Compatibility:* O inspetor/aplicador com ampla experiência detecta uma incompatibilidade entre a função descrita pelo documento de design ou o código e os outros sistemas, produtos, serviços, componentes ou módulos com os quais deve fazer interface.

Tabela 4 – Relação *Activity/Trigger*

Activity	***Triggers***
Design review / Code Inspection	*Design Conformance* *Logic/Flow* *Backward Compatibility* *Lateral Compatibility* *Concurrency* *Internal Document* *Language Dependency* *Side Effect* *Rare Situations*
Unit Test	*Simple Path* *Complex Path*
Function Test	*Test Coverage* *Test Variation* *Test Sequencing* *Test Interaction*
System Test	*Workload/Stress* *Recovery/Exception* *Start up/Restart* *Hardware Configuration* *Software Configuration* *Blocked Test (Previously called Normal Mode)*

- *Concurrency:* O inspetor/aplicador está considerando a serialização necessária para controlar um recurso compartilhado quando o defeito é descoberto. Isso incluiria a serialização de várias funções, threads, processos ou contextos de kernel, bem como obter e liberar bloqueios.
- *Language Dependency:* O desenvolvedor detecta o defeito enquanto verifica os detalhes específicos da linguagem da implementação de um componente ou função. Padrões de

linguagem, questões de compilação e eficiências específicas de linguagem são exemplos de áreas potenciais de preocupação.

- *Side Effects:* O inspetor/aplicador usa ampla experiência ou conhecimento do produto para prever algum sistema, produto, função ou comportamento do componente que pode resultar *Design review*. Os efeitos colaterais seriam caracterizados como resultado do uso comum ou configurações, mas fora do escopo do componente ou função com a qual o design ou código em revisão está associado.
- *Rare Situation:* O inspetor/aplicador usa ampla experiência ou conhecimento do produto para prever algum comportamento do sistema que não é considerado ou abordado pelo projeto documentado ou *Design review* e normalmente estaria associado a configurações ou uso incomuns. A recuperação de erro ausente ou incompleta NÃO seria, em geral, classificada como *Rare Situation*, mas provavelmente poderia ser enquadrada em *Design Conformance* se detectada durante a Revisão / Inspeção.
- *Simple Path:* O caso de teste foi motivado pelo conhecimento de ramificações específicas do código e não pelo conhecimento externo da funcionalidade. Este gatilho normalmente não seria selecionado para defeitos relatados em campo, a menos que o cliente tenha muito conhecimento do código e do design interno e esteja invocando especificamente um caminho específico (como às vezes é o caso quando o cliente é um parceiro de negócios ou fornecedor).
- *Complex Path:* No teste de caixa branca, o caso de teste que encontrou o defeito estava executando algumas combinações condicionais de cobertura de código. Em outras palavras, o testador tentou invocar a execução de vários condicionais sob várias condições diferentes. Este gatilho só seria selecionado para defeitos relatados em campo nas mesmas circunstâncias descritas em *Simple Path.*
- *Coverage:* Durante o teste da caixa preta, o caso de teste que encontrou o defeito foi uma tentativa direta de exercitar o código para uma única função, sem usar parâmetros ou com um único conjunto de parâmetros.
- *Variation:* Durante o teste da caixa preta, o caso de teste que encontrou o defeito foi uma tentativa direta de exercitar o código para uma única função, mas usando uma variedade de entradas e parâmetros. Este pode incluir parâmetros inválidos, valores extremos, condições de limite e combinações de parâmetros.
- *Sequencing:* Durante o teste da caixa preta, o caso de teste que encontrou o defeito executou

várias funções em uma sequência muito específica. Esta ação só é escolhido quando cada função executa com sucesso, de forma independente, mas falha nesta sequência específica. Também pode ser possível executar uma sequência diferente com sucesso.

- *Interaction:* Durante o teste da caixa preta, o caso de teste que encontrou o defeito iniciou uma interação entre dois ou mais campos de código. Esta ação só é escolhido quando cada função executa com sucesso, de forma independente, mas falha nesta combinação específica. A interação era mais abrangente do que uma simples sequência serial das execuções.
- *Workload/Stress:* O sistema está operando no limite ou próximo a algum limite de recursos, superior ou inferior. Esses limites de recursos podem ser criados por meio de uma variedade de mecanismos, incluindo execução pequena ou grandes cargas, executando alguns ou muitos produtos ao mesmo tempo, permitindo que o sistema funcione por um longo período de tempo.
- *Recovery/Exception:* O sistema está sendo testado com a intenção de invocar um manipulador de exceção ou algum tipo de código de recuperação. O defeito não teria surgido se alguma exceção anterior não tivesse feito com que a exceção ou o processamento de recuperação fosse invocado. De uma perspectiva de campo, esta ação seria selecionada se o sistema ou produto, possuísse a capacidade de se recuperar de falhas.
- *Startup/Restart:* O sistema ou subsistema estava sendo inicializado ou reiniciado após algum desligamento anterior ou alterar o grau de disponibilidade do serviço
- *Hardware Configuration:* O sistema está sendo testado para garantir que as funções sejam executadas corretamente em configurações de hardware específicas.
- *Software Configuration:* O sistema está sendo testado para garantir que as funções sejam executadas corretamente em configurações de software específicas.
- *Blocked Test:* O produto está operando bem dentro dos limites de recursos e o defeito veio à tona ao tentar executar um cenário de teste do sistema. Esta ação seria usado quando os cenários não puderem ser executados porque existem problemas básicos que impedem sua execução. Esta ação não deve ser usado em defeitos relatados pelo cliente.

4.3.3 Análise do *Impact*

Essa atividade consiste em atribuir um rótulo referente ao *Impact*, ou seja, o efeito causado pelo *bug* na plataforma de computação em nuvem o qual foi percebido pelo usuário. De acordo com a *ODC*, as descrições do *Impact* são (CHILLAREGE et al., 1992):

- *Installability*: A capacidade do cliente de preparar e colocar o software em posição de uso.
- *Integrity/Security*: A proteção de sistemas, programas e dados contra destruição, alteração ou divulgação inadvertida ou maliciosa.
- *Performance*: A velocidade do software percebida pelo cliente e pelos usuários finais, em termos de capacidade de execução de suas tarefas.
- *Maintenance*: A facilidade de aplicar correções preventivas ou corretivas ao software.
- *Serviceability*: A capacidade de diagnosticar falhas com facilidade e rapidez, com impacto mínimo para o cliente
- *Migration*: A facilidade de atualização para uma versão atual, especialmente em termos de impacto nos dados e operações existentes do cliente.
- *Documentation*: O grau de correção e completude das informações e publicações auxiliares fornecidas para a compreensão da estrutura e uso pretendido do software.
- *Usability*: O grau de informações e publicações acerca do software permitem que o produto seja facilmente compreendido e convenientemente empregado por seu usuário final.
- *Standards*: O grau em que o software está em conformidade com os padrões estabelecidos.
- *Reliability*: A capacidade do software executar de forma consistente a função pretendida sem interrupções não planejadas.
- *Requirements*: Uma expectativa do cliente, em relação à capacidade, que não era conhecida, compreendida ou priorizada como um requisito para o produto ou lançamento atual.
- *Accessibility*: Garantir que o acesso à informação e o uso da tecnologia da informação sejam bem-sucedidos às pessoas com deficiência.
- *Capability*: A capacidade do software de executar as funções pretendidas e satisfazer os requisitos, onde o cliente não é afetado em nenhuma das categorias anteriores.

4.4 Síntese Conclusiva

Nesta seção, aprofundamos a discussão sobre a atividade de determinação do *defect type*, integrando os insights adquiridos através do emprego das *tags* durante a etapa crítica

de **aplicação das definições *ODC***. A análise aqui apresentada busca sintetizar o processo de classificação de defeitos, culminando na atribuição de um tipo de defeito específico que reflete as características identificadas ao longo da metodologia aplicada.

4.4.1 Definição do *Defect Type*

Esta atividade consiste na análise das atribuições feitas pela macro atividade Aplicações das Definições *ODC*, para propor possíveis soluções para o *bug*, isso é, para aqueles que estão com *status new*, ou seja, não foi solucionado. Para defeitos com *status* definidos como *Fix*, ou seja, defeitos que de alguma forma já foram corrigidos, seleciona-se o *Defect Type* que se equipare com a solução encontrada. Conforme *ODC*, as descrições do *Defect Type* são (CHILLAREGE et al., 1992):

- *Assignment/Initialization:* Valor (es) atribuído (s) incorretamente ou não atribuído (s); uma correção, envolvendo várias correções de atribuição em algoritmo.
- *Checking:* Erros causados por validação ausente ou incorreta de parâmetros ou dados em declarações condicionais. Pode-se esperar que uma consequência da verificação de um valor exija código adicional, como um loop do while ou branch. Se a verificação ausente ou incorreta for o erro crítico, a verificação ainda seria o tipo escolhido.
- *Algorithm/Method:* Problemas de eficiência ou correção que afetam a tarefa e podem ser corrigidos pela (re) implementação de um algoritmo ou estrutura de dados local, sem a necessidade de solicitar uma alteração no projeto. Problema no procedimento, modelo ou função sobrecarregada que descreve um serviço oferecido por um objeto.
- *Function/Class/Object:* O erro deve exigir uma mudança formal de design, pois afeta a capacidade significativa, interfaces de usuário final, interfaces de produto, interface com arquitetura de hardware ou dados globais estrutura (s); O erro ocorreu ao implementar o estado e as capacidades de uma entidade real ou abstrata.
- *Timing/Serialization:* A serialização necessária do recurso compartilhado estava faltando, o recurso errado foi serializado ou a técnica de serialização errada foi empregada.
- *Interface/O-O Messages:* Problemas de comunicação entre módulos, componentes, dispositivos, funções.
- *Relationship:* Problemas relacionados a associações entre procedimentos, estruturas de dados e objetos. Essas associações podem ser condicionais.

4.5 Considerações Finais

Este capítulo detalhou uma metodologia robusta e replicável para a classificação de defeitos em plataformas de computação em nuvem privada, fundamentada nas definições do modelo *Orthogonal Defect Classification* (ODC). A aplicação desta metodologia transcende a mera correção de falhas, abrindo caminho para um entendimento mais profundo dos defeitos, o que é crucial tanto para a evolução das plataformas quanto para a sustentação de sua confiabilidade e disponibilidade.

Para demonstrar a eficácia e aplicabilidade prática da metodologia proposta, foi desenvolvida e integrada ao estudo uma ferramenta específica, denominada **FIES**, desenhada para simular defeitos de forma controlada, baseando-se nos princípios e diretrizes do ODC. A existência desta ferramenta não apenas valida nossa metodologia, mas também serve como um recurso adicional para a comunidade, auxiliando na gestão de qualidade e na avaliação da resiliência das plataformas de nuvem privada.

No capítulo subsequente, dedicaremos uma análise detalhada à ferramenta **FIES**, explorando suas capacidades de emulação de defeitos e o impacto de sua utilização na classificação e compreensão dos bugs, corroborando a viabilidade e eficiência da nossa abordagem metodológica.

5 FIES - Ferramenta para Injeção de Falhas na Nuvem Privada

Este capítulo apresenta a ferramenta **FIES** (***F**ault **I**njection in **E**mulated **S**cenarios*), projetada para dar suporte à injeção de falhas artificiais em um ambiente real de computação em nuvem privada.

5.1 Visão Geral

A motivação para o desenvolvimento da **FIES** se deu em função da necessidade de usar uma ferramenta em diferentes plataformas para injetar falhas artificiais em ambientes de nuvem privada, de forma a emular cenários semelhantes ao momento da ocorrência/percepção do *bug*. Os repositórios de *bugs* ou *Bugtrakers* como são conhecidos, são sistemas com capacidade de rastrear defeitos em *softwares* e pedido de novos recursos; para além desta finalidade, pode-se usá-lo como biblioteca para entendimento da ocorrência de *bugs*, seus efeitos e soluções encontradas pelo desenvolvedor e/ou pela comunidade.

O desenvolvimento e utilização da ferramenta **FIES** leva em consideração a descrição da ocorrência do *bug*. A ferramenta injeta falhas de acordo com a descrição feita, emulando o cenário da sua ocorrência, em *bugs* não solucionados, ou seja, classificado no repositório com *status new*. A **FIES** atua na macro atividade de **aplicação das definições *ODC*** da metodologia proposta neste trabalho (Capítulo 4); primeiro, emulado o cenário no momento que o *bug* foi percebido, essa etapa compreende a *activity*, para em seguida definir a *trigger* e verificar o impacto e propagação de falhas nos serviços e componentes da nuvem após a injeção da falha. A Figura 7 apresenta em detalhes a interação do usuário com a ferramenta que consiste nos seguintes passos:

1. Acesso a **Interface da Ferramenta** que é executada em *browser* ao ser passado na barra de endereços a *URL* padrão (*https://127.0.0.1:8000*), após inicializar servidor local *virtualenv*;
2. A **Configuração dos Parâmetros da Nuvem privada** consiste no preenchimento de formulário, onde é passado o nome do componente e seu respectivo endereço *IP*, feitos no momento da instalação/configuração do ambiente;
3. Na **Criação do repositório de *bugs***, é selecionada uma ***Activity***, uma ***Trigger*** a ela associada, o componente na qual a falha será disparada e por fim é inserido em formulário

a linha de comando reponsável por emular o *bug*.

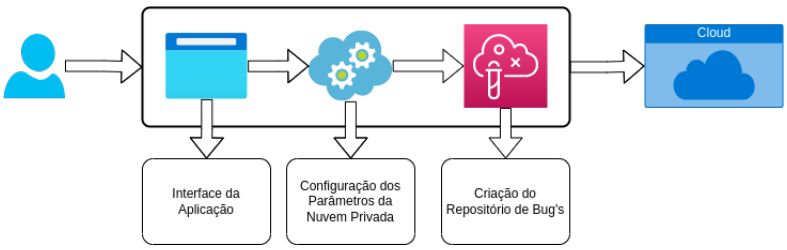

Figura 7 – Interação do Usuário com a Ferramenta FIES (FONTE: Próprio Autor)

A ferramenta concentra-se em componentes de alto nível, englobando tanto a infraestrutura física quanto as máquinas virtuais. É crucial enfatizar que o repositório de bugs está intrinsecamente ligado à plataforma específica; isto é, comandos particularizados que emulam falhas variam de acordo com cada sistema de nuvem privada. Contudo, a ferramenta foi projetada para interagir de maneira eficiente com os mais reconhecidos ambientes de computação em nuvem privada disponíveis no mercado.

5.2 FIES - Descrição

A ferramenta de injeção de falhas **FIES** foi desenvolvida para avaliar a dependabilidade dos serviços de computação em nuvem privada. A Figura 8 ilustra uma visão geral da arquitetura da ferramenta **FIES**. O *Controller* foi projetado para controlar todos os fluxos de trabalho, empregando o *Experiment Actuator* e *Fault Repository*. Em primeiro lugar, ele carrega as falhas armazenadas no banco de dados, que podem ser do tipo: cargas de trabalho, falhas, linha de comando (*CLI*) e arquivos de scripts escritos em *Python*. O banco de dados é representado como *Fault Repository* no diagrama. O *Experiment Actuator* controla o ambiente de nuvem por meio do adaptador nativo da *Cloud Platform*, tendo acesso a todo ambiente e *VM's*, de modo que o injetor de falhas, parte integrante do *Experiment Actuator*, alcance todos os componentes da plataforma. O *Controller* detecta a falha por meio do *Error Detect* disparando uma mensagem ao *Experiment Reporter*, responsável pela coleta, análise e geração de relatórios.

5.3 Desenvolvimento

A ferramenta FIES foi meticulosamente concebida para assegurar versatilidade no seu emprego, adaptando-se eficientemente à variedade de plataformas de nuvem existentes. A

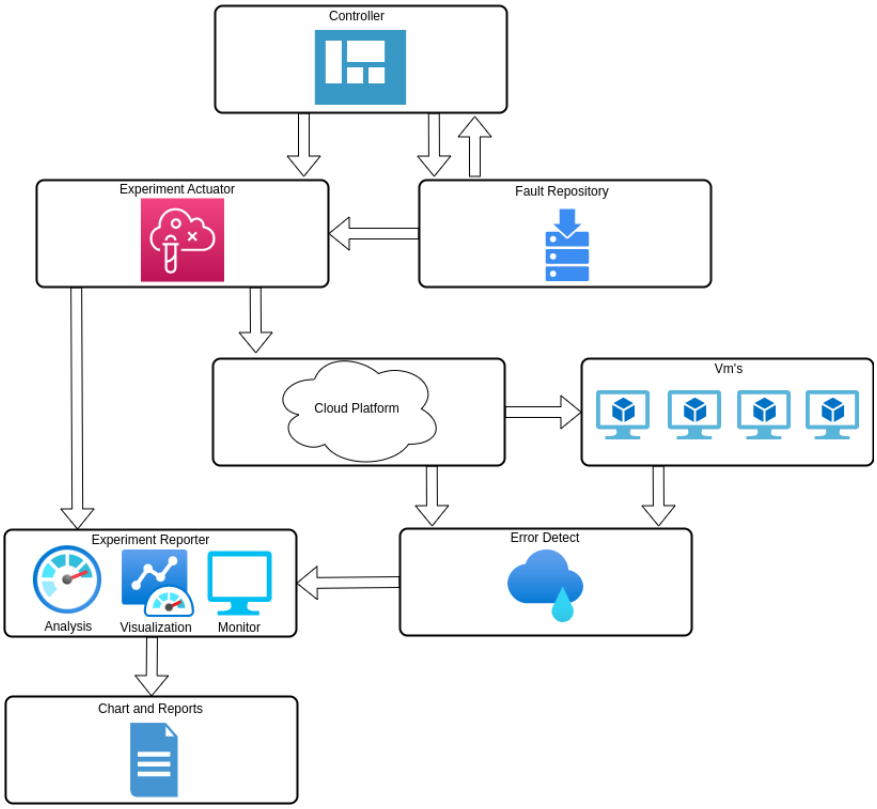

Figura 8 – FIES - Arquitetura da Ferramenta Para Injeção de Falhas (FONTE: Próprio Autor)

integração do protocolo SSH como mecanismo de comunicação é um ponto-chave nesse aspecto, já que permite a execução de comandos específicos do sistema operacional e das distintas plataformas de nuvem. Esta escolha estratégica viabiliza a operacionalidade da ferramenta FIES em diversas infraestruturas, especialmente por ser uma tecnologia amplamente suportada pelas principais distribuições Linux. Detalhes adicionais sobre a concepção e operação da ferramenta serão abordados nas seções subsequentes. A Figura 9 esquematiza os procedimentos executados pela ferramenta para cumprir os objetivos propostos neste estudo.

5.3.1 Linguagem de Programação

A escolha da linguagem de programação foi a primeira atividade considerada para a implementação da ferramenta. Tendo em vista sua abrangência, flexibilidade e levando em consideração sua presença na codificação total ou parcial nos projetos das principais plataformas de nuvem privada do mercado, optou-se por utilizar a linguagem *Python*. Outra razão para sua escolha, foi sua vasta biblioteca que pode ser importada para o projeto, como geradores de gráficos, relatórios e comunicação com plataformas de nuvem.

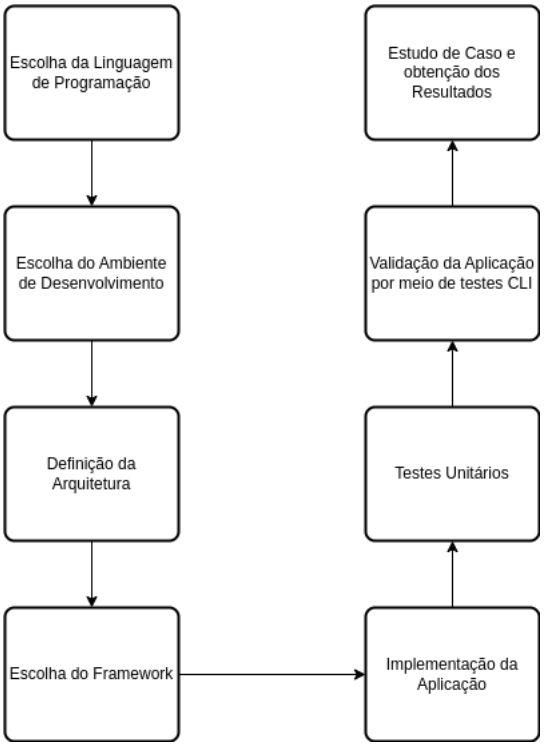

Figura 9 – Atividades Adotadas Durante a Implementação da FIES (FONTE: Próprio Autor)

5.3.2 Escolha do Ambiente de Desenvolvimento

Na sequência do processo de desenvolvimento, a seleção de um ambiente de programação adequado é crucial. Nesse contexto, a escolha recaiu sobre a IDE PyCharm versão 2022.3, uma decisão que se alinha harmoniosamente com a linguagem Python, previamente selecionada pela sua versatilidade e eficácia. A PyCharm se destaca pela sua integração profunda com Python, oferecendo um conjunto robusto de ferramentas especializadas que potencializam a produtividade e a eficiência do desenvolvimento. Entre essas ferramentas, encontram-se funcionalidades como análise de código em tempo real, gerenciamento de sistemas de controle de versão e um sistema de depuração avançado, todos adaptados especificamente para a linguagem Python. Estas características tornam a PyCharm uma escolha estratégica para a codificação da ferramenta FIES, garantindo um fluxo de trabalho otimizado e a qualidade do código produzido.

5.3.3 Definição da Arquitetura

A terceira etapa do desenvolvimento focaliza na definição da arquitetura da ferramenta, um passo decisivo para assegurar a flexibilidade e escalabilidade da solução. Para atingir esse objetivo, optou-se pela implementação da arquitetura Modelo-Visão-Controlador (MVC). Essa

arquitetura tripartida permite a segregação das responsabilidades em três componentes principais:

- Modelo (Model): É a camada responsável pela gestão dos dados, lógica de negócios e regras da aplicação. Isola o domínio da aplicação da interface do usuário e a lógica de apresentação, facilitando a sua gestão e manutenção.
- Visão (View): Refere-se à camada de apresentação, responsável por exibir os dados ao usuário. Qualquer representação visual dos dados, como gráficos ou tabelas, é manipulada aqui.
- Controlador (Controller): Atua como intermediário entre o modelo e a visão, recebendo os inputs dos usuários e convertendo-os em comandos para o modelo ou visão.

A adoção do MVC traz múltiplas vantagens. O baixo acoplamento entre as camadas promove uma maior facilidade de teste e manutenção do código. Além disso, a clara separação de responsabilidades facilita a reutilização do código e permite que equipes de desenvolvimento trabalhem em componentes distintos sem interferências. Finalmente, essa abordagem de design suporta a evolução do projeto ao longo do tempo com adaptabilidade, permitindo que alterações em uma camada sejam feitas com impacto mínimo nas outras.

5.3.4 Escolha do Framework

A etapa seguinte é a escolha do *framework*, optou-se pelo *Django*. O *Django* é uma estrutura da *web* de *back-end* em *Python* gratuita e de código aberto que fornecendo uma estrtura coesa para o desenvolvimento de aplicações. O *Django* lida com a maior parte do ciclo de requisição e resposta *HTTP*, o restante é tratado pelo servidor *Django* (PINKHAM, 2016).

A estrutura do projeto *Django* é frequentemente descrita de acordo com a arquitetura *Model-View-Controller* (MVC).

5.3.5 Implementação da Ferramenta

Na fase de implementação da ferramena **FIES**, procedeu-se com a codificação efetiva do projeto utilizando o ambiente de programação *PyCharm*. Durante este processo, estabeleceu-se a estrutura fundamental do projeto, que abrange a definição dos *models* (modelos), que são as representações dos dados da ferramenta; *views* (visões), que controlam a forma como os dados são apresentados aos usuários; *urls*, que determinam os esquemas de endereçamento dentro

da ferramenta; e *apis* (interfaces de programação de aplicações), que permitem a interação e comunicação com outros softwares.

É importante salientar que, embora o desenvolvimento da ferramenta **FIES** seja meticuloso e seguindo as melhores práticas de engenharia de software, o código-fonte não será disponibilizado publicamente. Esta decisão se baseia no intuito de proteger a propriedade intelectual e permitir o devido processo de registro da ferramenta, garantindo assim a salvaguarda dos direitos associados e permitindo o controle sobre a distribuição e uso da mesma.

5.3.6 Testes Unitários

A sexta etapa, corresponde ao Teste Unitário, na qual todos os métodos que compõem a ferramenta foram testados, com objetivo de verificar seu desempenho. Foi utilizada a biblioteca *PyUnit* para testar alguns métodos específicos da ferramenta.

5.3.7 Validação da Ferramenta por Meio de Teste *CLI*

Após os testes unitários, foram feitos vários testes de comunicação da ferramenta em máquinas virtuais via *SSH*, executando comandos nativo do sistema operacional.

5.3.8 Estudo de Caso e Obtenção de Resultados

Por fim, a última etapa é voltada para analisar a eficiência da ferramenta através dos resultados obtidos no estudo de caso, onde envolve a avaliação da dependabilidade de componentes da nuvem privada frente a ocorrência da falha injetada. O Capítulo 6 descreve com maiores detalhes esta etapa.

5.4 Considerações Finais

Este capítulo apresentou conceitos e os aspectos que motivaram o desenvolvimento da ferramenta para injeção de falhas **FIES**, além da metodologia adotada para o desenvolvimento do projeto. O próximo capítulo apresentará estudos de caso no qual a ferramenta será validada em ambiente real de computação de nuvem privada.

6 Estudo de caso

Este capítulo tem como foco principal a avaliação da dependabilidade de ambientes de nuvens privadas, utilizando para isso uma metodologia fundamentada na classificação de defeitos, bem como na injeção de falhas, por meio da ferramenta **FIES**. A eficácia desta abordagem metodológica é posta à prova em dois estudos de caso distintos.

No Estudo de Caso 1, exploramos a aplicação da metodologia proposta na classificação de defeitos de componentes específicos da plataforma de computação em nuvem *OpenStack*. O propósito é entender em que medida a classificação contribui para aperfeiçoar a confiabilidade do ambiente de nuvem.

Em seguida, o Estudo de Caso 2 foca na determinação do impacto da injeção de falhas nos serviços da plataforma *OpenStack*. Utilizando a ferramenta **FIES** para simular defeitos, avaliamos como as falhas afetam a estabilidade e o desempenho dos serviços de nuvem.

Ambos os estudos têm a intenção de validar a metodologia e a ferramenta desenvolvida, confirmando a sua utilidade prática na melhoria da dependabilidade de sistemas de nuvem privada.

6.1 Estudo de Caso 1 - Classificação de Defeitos da Plataforma de Nuvem Privada OpenStack

Esta seção apresenta um estudo cujo objetivo é avaliar a metodologia proposta (ver Seção 4.1) que pode ser adaptada para classificação de defeitos em diferentes plataformas de computação em nuvem, levando em consideração o sistema descrito. Esse estudo de caso avalia a aplicabilidade da metodologia proposta com base na ODC. Os resultados dessa avaliação proporcionam um entendimento mais sólido dos defeitos abertos (não solucionados). As próximas seções descrevem as atividades executadas e resultados obtidos.

6.1.1 Estudo Comparativo Entre as Plataformas de Nuvem Privada

Conforme discutido na Seção 4.2.1, onde foram exploradas as características das principais plataformas de nuvem privada disponíveis no mercado, a escolha recaiu sobre o *OpenStack*. Esta decisão foi embasada em evidências concretas, tais como seu superior

desempenho sistêmico pós-implantação, o que é comprovado por uma série de *benchmarks* em que o *OpenStack* demonstrou ser preponderante. Além do mais, a adesão crescente tanto por parte da comunidade open-source quanto de empresas líderes de mercado sinaliza uma tendência de convergência para o *OpenStack* como solução de preferência para a implementação de nuvens privadas em ambientes organizacionais (MULLERIKKAL; SASTRI, 2015).

A escolha do *OpenStack* é também estratégica, considerando o alinhamento com as principais iniciativas tecnológicas e o potencial de integração com uma gama diversificada de soluções e serviços. Tal flexibilidade e abertura à inovação posicionam o *OpenStack* não apenas como uma solução atualmente robusta, mas também como uma plataforma que promete escalabilidade e adaptabilidade às futuras demandas do setor de computação em nuvem privada.

6.1.2 Escolha do Repositório de Defeitos

Após a escolha da plataforma de nuvem, a definição do repositório de defeitos é feita com base na distribuição Linux no qual o *OpenStack* foi instalado. Optamos pelo repositório *Bugzilla*[1] do *Mozilla Foundation*, por concentrar um grande número de requisições e informações acerca de instalação e uso do *OpenStack* configurado na distribuição Linux baseada no *Red Hat*. Outro fator a ser considerado é que a ferramenta permite filtros de pesquisa individualizado por componentes e composição do relatório de defeitos com as informações definidas pelo usuário além de criar gráfico de *burndown* a partir do filtro criado, *RPC* em *Pearl* e *JSON*, expande árvore de dependência dos *bugs* e exporta os relatórios nos formatos *CSV* e *XML*.

6.1.3 Pesquisa de Defeitos por Componentes

O *Bugzilla* mantém um repositório altamente organizado. Cada repositório contém problemas relacionados a desenvolvimento e implantação que são enviados por desenvolvedores e usuários de uma comunidade. Para cada defeito, o repositório armazena vários rótulos que podem ser utilizados na pesquisa (Por exemplo *bug id*, *summary*, *changed*, *priority, severity, hardware, keywords, component, type, status, internal whiteboard* e OS) . Nosso filtro foi feito com base nos componentes da plataforma: *nova, neutron, swift, keystone, glance e cinder*. As TAG's usadas na pesquisa (Ver Tabela 3), são termos chave aplicado ao rótulo *component*, usados

[1] https://bugzilla.redhat.com/

na busca informações sobre um determinado componente alvo.

O resultado obtido após a execução dos filtros, foi um total de 395 defeitos, divididos por componentes conforme a Tabela 5, submetidos ao longo de três anos (01/2018 - 01/2021).

Tabela 5 – Total de Defeitos por Componente

Componente	***Bugs***
Nova	99
Swift	22
Neutron	100
Keystone	38
Glace	35
Cinder	101

6.1.4 Geração de DataSet

Cada filtro resultou em informações sobre um componente específico da plataforma *OpenStack*. As informações dos componentes foram exportadas no formato *CSV*, agrupadas de forma tabular para comporem um *DataSet*[2]. Algumas alterações foram feitas nos cabeçalhos das colunas com objetivo de tornar mais intuitivo, como o *summary* foi alterado para *title*. Foram adicionados os rótulos *web link* para informações referentes a *URL*, *activity* e *trigger*.

6.1.5 Aplicação da Activity

Na Seção 4.3.1, as características dos atributos de *Activity* foram descritas conforme segue:

- *Design Review*: Avaliação de design ou comparação do design documentado com os requisitos estabelecidos.
- *Code Inspection*: Inspeção de código ou comparação do código fonte com o design documentado.

[2] https://drive.google.com/file/d/1-84zILeBmaSKXu3YLgIOoPpQHMXTLkDD/view?usp=sharing

- *Unit Test*: Teste de unidade ou caixa branca, realizando execução baseada no conhecimento detalhado dos componentes internos do código.
- *Function Test*: Teste funcional ou caixa preta, executado com base nas especificações externas de funcionalidade.
- *System Test*: Teste do sistema completo ou execução em um ambiente de produção, exigindo a funcionalidade de todos os recursos.

A atribuição manual das classes de *Activity* foi guiada por uma análise minuciosa do *summary* e *description* de cada defeito. As decisões foram baseadas na interpretação semântica fornecida por estas descrições.

Os resultados dessa classificação são evidenciados na Figura 10, que apresenta a distribuição percentual das classes por componente no *OpenStack*. Notadamente, a classe *System Test* predomina no componente Nova, correspondendo a 55% dos casos. Conforme as diretrizes da *Orthogonal Defect Classification (ODC)*, a classificação como *System Test* é aplicável apenas quando o ambiente completo está operacional, refletindo a funcionalidade crítica do componente Nova no gerenciamento do ciclo de vida das instâncias da nuvem e na administração dos recursos computacionais, de rede, autorização e escalabilidade. Os 45% restantes englobam defeitos que afetam diretamente os serviços prestados pelo componente.

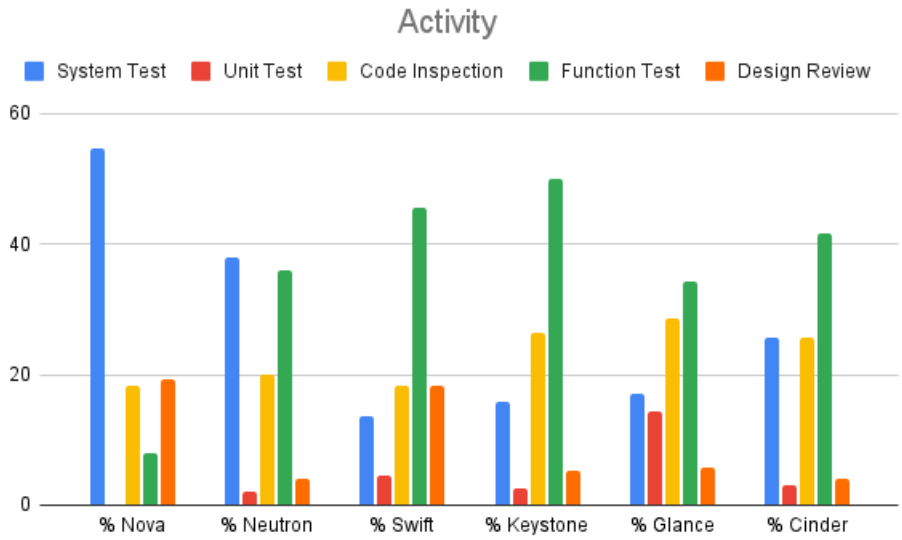

Figura 10 – Distribuição das Classes do Atributo *Activity* Aplicados aos Componentes da Plataforma *OpenStack*

A Tabela 6 mostra o percentual de classes atribuída aos componentes da plataforma de nuvem *OpenStack*, em relação ao atributo *Activity*.

Tabela 6 – Total de Defeitos por Componente

Activity	*% Nova*	*% Neutron*	*% Swift*	*% Keystone*	*% Glance*	*% Cinder*
System Test	55	38	14	16	17	26
Unit Test	✗	2	5	3	14	3
Code Inspection	18	20	18	26	29	26
Function Test	8	36	45	50	34	42
Design Review	19	4	18	5	6	4

6.1.6 Escolha da Trigger

A *Trigger* representa o ambiente ou condição que deveria existir para a percepção do defeito, ou, o que seria necessário para reproduzir/emular o ambiente do *bug* (CHILLAREGE et al., 1992). Os atributos *Trigger* estão diretamente associados à escolha da *Activity* conforme Tabela 4. Ao atribuir uma *Activity* a um defeito, passamos a ter um subconjunto de atributos relacionados à *Trigger*. Por exemplo, ao aplicar uma *Activity Review* e *Inspection*, escolhe-se a Trigger que melhor representa o defeito. Para os demais casos, é ideal que se combine a descrição do defeito com o ambiente ou condição no qual ocorreu o defeito.

A Figura 11 mostra o percentual de classes atribuída aos componentes da plataforma de nuvem *OpenStack*, em relação ao atributo *Trigger*.

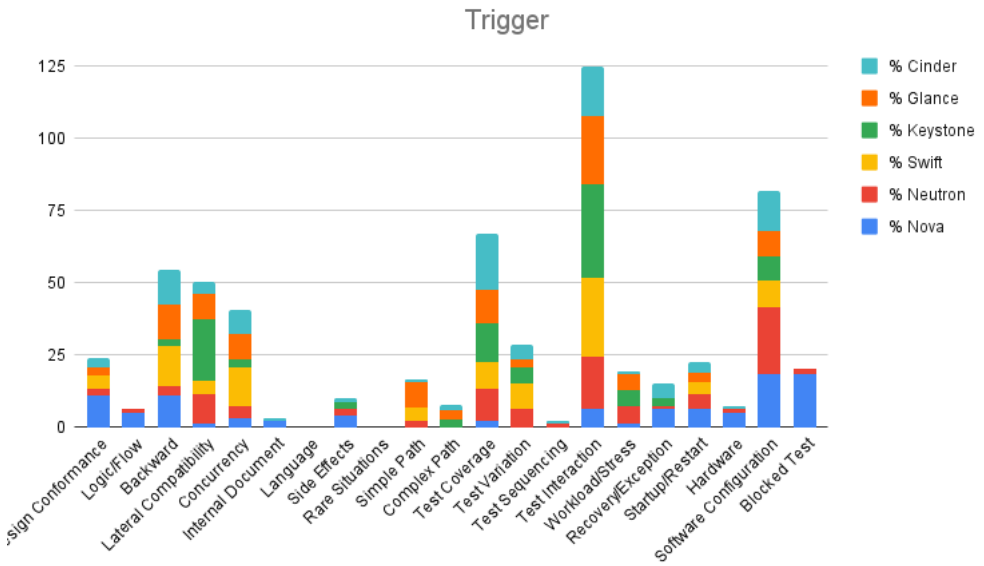

Figura 11 – Atributos *Trigger* Aplicados aos Componentes da Plataforma *OpenStack*

A Tabela 7 mostra o percentual de classes atribuída aos componentes da plataforma de

nuvem *OpenStack*, em relação ao atributo *Trigger*.

Tabela 7 – Classes *Trigger*

Trigger	***% Nova***	***% Neutron***	***% Swift***	***% Keystone***	***% Glance***	***% Cinder***
Design Conformance	11	2	5	✗	3	3
Logic/Flow	5	1	✗	✗	✗	✗
Backward Compatibility	11	3	14	3	12	12
Lateral Compatibility	1	10	5	22	9	4
Concurrency	3	4	14	3	9	8
Internal Document	2	✗	✗	✗	✗	1
Language Dependency	✗	✗	✗	✗	✗	✗
Side Effects	4	2	✗	3	✗	1
Rare Situations	✗	✗	✗	✗	✗	✗
Simple Path	✗	2	5	✗	9	1
Complex Path	✗	✗	✗	3	3	2
Test Coverage	2	11	9	14	12	19
Test Variation	✗	6	9	5	3	5
Test Sequencing	✗	1	✗	✗	✗	1
Test Interaction	6	18	27	32	24	17
Workload/Stress	1	6	✗	5	6	1
Recovery/Exception	6	1	✗	3	✗	5
Startup/Restart	6	5	5	✗	3	4
Hardware Configuration	5	1	✗	✗	✗	1
Software Configuration	18	23	9	8	9	14
Blocked Test	18	2	✗	✗	✗	✗

6.1.7 Análise do Impact

A classe *impact* não foi utilizada para esse estudo de caso, tendo em vista que o DataSet extraído e analisado se trata de defeitos com *Status* definidos como *New*, abertos. A escolha por defeitos abertos se deve ao fato de existirem em maior número no período da pesquisa nas

plataformas de *bugtrakers*. O atributo *impact* é atribuído a defeitos considerados concluídos *Fix*.

6.1.8 Análise do Defect Type

O *Defect Type* não foi levado em consideração para esta classificação, por se tratar de um atributo associado à análise do processo da resolução do defeitos com *status* definidos como *Fix* ou seja, defeitos já foram corrigidos.

6.1.9 Reavaliar Recursivamente a Fase de Definição ODC

A técnica *ODC* tem sido usada com sucesso como uma ferramenta para melhorar o processo de design de software e fornecer uma base importante para entender e classificar defeitos de software. No entanto, sua aplicação relaciona defeitos à maneira como os mesmos foram corrigidos (DURAES; MADEIRA, 2006). Deste modo, esta atividade consiste em reavaliar a fase de definição da *ODC* sob a perspectiva de um segundo avaliador, considerando que este tenha conhecimento em relação a técnica *ODC* e compreensão prévia das falhas reportadas, sendo possível fazer um paralelo entre as classificações sob perspectivas distintas.

6.2 Estudo de Caso 2 - Validação da Ferramenta de Injeção de Falhas

O propósito deste estudo de caso é validar a ferramenta **FIES**, emulado condições necessárias para que o defeito ocorra por meio de injeção de falhas em nuvem privada configurada com OpenStack e avaliar a dependabilidade dos módulos da nuvem frente às falhas injetadas. Para os experimentos, serão considerados defeitos classificadas de acordo com as classes *Workload/Stress*, *Startup/Restart* e *Software Configuration* do atributo Trigger da ODC. A injeção de falhas se concentra nos módulos Nova e Neutron considerando que ambos concentram maior volume de bugs relatados.

6.2.1 Ambiente de Teste

O ambiente de teste na qual a nuvem privada foi implantada é inteiramente baseada no *OpenStack* em máquinas virtuais (*VM's*) utilizado *hypervisor KVM*, instanciados em dois

computadores físicos denominados *Host1* e *Host2*. A Tabela 8 mostra as especificações dos computadores físicos e *VM's* hospedadas. A nuvem foi configurada com quatro nós composto de um nó denominado *Controller Node (CLN)*, um nó *Compute Node (CN)*, um nó *Network Node (NN)* e um *Storage Node (SN)*. Cada nó é uma máquina virtual (VM) dedicada, configurada com recursos computacionais mostrados na Tabela 9.

Tabela 8 – Especificação dos *hosts* hospedeiros e VM's

Hardware	**Especificação**	**Node**
Host1	*CPU Intel(R) Core(R) i7-2600 @ 3.40 GHz (8 core)* *RAM 2 x DDR3 8193 MB @ 1333 MHZ* *HD 1TB SATA*	*Controller (CLN)* *Network (NN)*
Host2	*CPU Intel(R) Core(R) i7-2600 @ 3.40 GHz (8 core)* *RAM 2 x DDR3 8193 MB @ 1333 MHZ* *HD 1TB SATA*	*Compute (CN)* *Storage (SN)*

Tabela 9 – Especificação dos nós da nuvem privada

Node	***Memória (GB)***	***Cores***	***Espaço em Disco (GB)***
Controller (CLN)	*4*	*3*	*300*
Compute (CN)	*4*	*4*	*300*
Network (NN)	*2*	*2*	*30*
Storage (SN)	*8*	*3*	*300*

Para os sistemas operacionais dos hosts físicos e das instâncias virtuais na infraestrutura de nuvem privada, optou-se pelo *Linux CentOS 7 minimal*. Essa escolha foi pautada não apenas pela menor demanda de recursos computacionais devido à ausência de interface gráfica, o que favorece a minimização de *overheads*, mas também pela sua sólida base no *Red Hat Linux*—uma distribuição conhecida por sua estabilidade e segurança (CENTOS, 2019). Tal característica é fundamental em ambientes de nuvem onde a confiabilidade é crucial. A versão do *OpenStack* escolhida foi a *Train*, que se integra harmoniosamente com o *CentOS* através da ferramenta *Packstack*, facilitando a implantação de ambientes compactos e eficientes (OPENSTACK, 2019).

6.2.2 Arquitetura da Nuvem

Neste trabalho, será utilizado a arquitetura *multinode* do *OpenStack*. A Figura 12 mostra a arquitetura implantada para este trabalho. A rede *service network* atende a comunicação das *APIs* do *OpenStack* e storage network é dedicada ao tráfego de armazenamento entre os servidores *Swift* e demais servidores *OpenStack*.

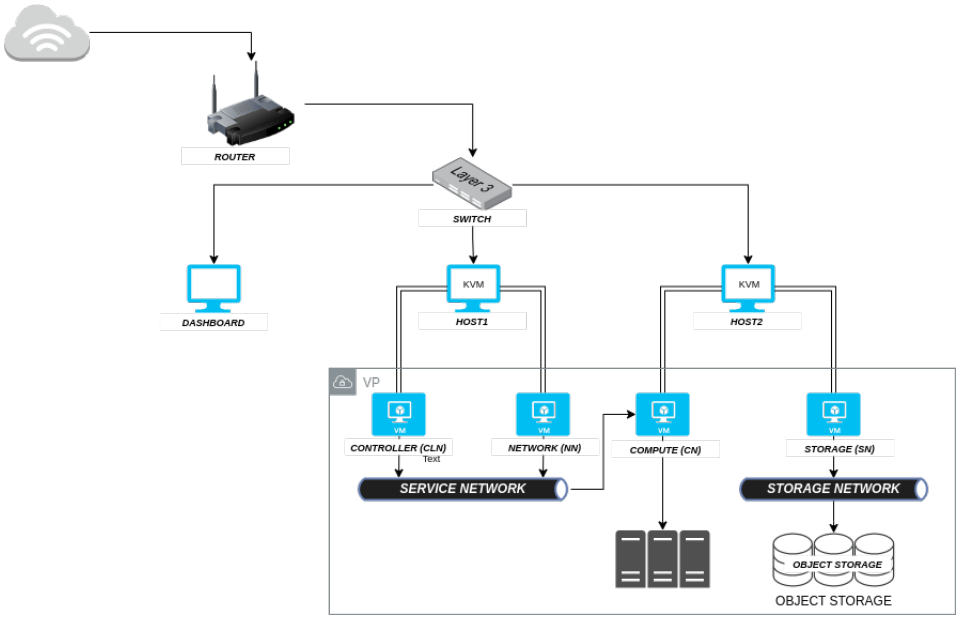

Figura 12 – Arquitetura Nuvem Privada (FONTE: Próprio Autor)

6.2.3 Injeção de Falhas

Escolhemos emular falhas rotuladas com as classes *Workload/Stress*, *Startup/Restart* e *Software Configuration* que compõem o atributo *Trigger*, ambas fazem parte da *ODC*. Foram implementadas 34 de 92.

A injeção de falhas foi executada nos componentes *Nova* e *Neutron*, considerando a gravidade do defeito reportado por usuários da plataforma *OpenStack* como mostrado na Tabela 10. Os **IDs** das falhas são mnemônicos formados pelos acrônimos da *Trigger* atribuída à falha e *Severity* reportada pelos usuários da plataforma.

Tabela 10 – Total de Falhas Injetadas Considerando sua Gravidade

Trigger	ID	Severity	Nova	Neutron
Workload/Stress	WSU	*Unspecified*	1	1
	WSL	*Low*	1	1
	WSM	*Medium*	2	1
	WSH	*High*	1	1
	WSR	*Urgent*	1	0
Startup/Restart	SRU	*Unspecified*	1	2
	SRL	*Low*	1	1
	SRM	*Medium*	2	2
	SRH	*High*	3	3
	SRR	*Urgent*	2	3
Software Configuration	SCU	*Unspecified*	0	0
	SCL	*Low*	1	1
	SCM	*Medium*	1	0
	SCH	*High*	0	1
	SCR	*Urgent*	0	0

6.2.4 Avaliação da Dependabilidade

A avaliação da dependabilidade foi meticulosamente realizada considerando quatro categorias distintas de comportamento do sistema frente às falhas injetadas. Estas categorias são descritas a seguir e são essenciais para compreender a reação do sistema à introdução de uma falha, ou seja, a causa, e suas respectivas repercussões, ou efeitos, sobre os módulos em questão:

1. **Hang**: Esta categoria é caracterizada por uma resposta suspensa do sistema, onde as operações pendentes não são concluídas, resultando em uma inércia operacional sem recuperação automática.
2. **Crash**: Representa uma falha abrupta e completa do sistema ou componente, resultando no término inesperado da execução do serviço.
3. **Silent Crash**: Refere-se a uma falha onde o sistema ou componente deixa de responder ou funcionar corretamente, porém, sem alertar o usuário ou os sistemas de monitoramento, podendo passar despercebido.
4. **Incorrect Functionality**: Denota um cenário onde o sistema continua operacional, mas

executa suas funções de maneira incorreta ou imprecisa.

Os efeitos resultantes das injeções de falhas nos componentes *Nova* e *Neutron* foram catalogados e analisados. A frequência e características de cada tipo de comportamento de efeito – *hang*, *crash*, *silent crash*, e *incorrect functionality* – são detalhadas nas Tabelas 11 e 12, permitindo uma visão abrangente do impacto das falhas sobre a infraestrutura testada.

Tabela 11 – Impacto de Falhas Injetada no Módulo Nova

Impact	***Falha ID***	***Módulo(s) Afetado***
Hang	*SRM, SRH, SRR*	*Glance*
Crash	*SRU, SRL*	*Glance*
Silent Crash	*WSU, WSL, WSH, WSR*	*Neutron, Keystone*
Incorrect Funcionality	*SCL, ACM*	*Glance*

Tabela 12 – Impacto de Falhas Injetada no Módulo Neutron

Impact	***Falha ID***	***Módulo(s) Afetado***
Hang	*SRM, SRH, SRR*	*Nova*
Crash	*SRU, SRL*	*Nova*
Silent Crash	*SCL, SCM, SCH*	*Keystone*
Incorrect Funcionality	*WSU, WSL, WSM, WSH, WSR*	*Nova*

6.3 Considerações Finais

Este capítulo detalhou dois estudos de caso fundamentais, enfocando a avaliação da dependabilidade de ambientes de nuvem privada e a classificação efetiva de bugs utilizando a ferramenta **FIES**. Os estudos de caso foram desenhados para simular falhas realistas em nuvens privadas, com o intuito de testar a metodologia proposta e a ferramenta desenvolvida para injeção de falhas. Os resultados alcançados endossam a validade da metodologia na classificação de bugs e confirmam que a ferramenta **FIES** satisfaz os propósitos para os quais foi criada, bem como proporciona insights valiosos sobre a dependabilidade dos sistemas em nuvem diante das falhas inseridas.

6.3.1 Classificação de Bugs e Avaliação da Metodologia

No primeiro estudo de caso, implementou-se a metodologia proposta sobre uma base de 395 bugs do *OpenStack*, reportados ao longo de três anos (2018-2021). Utilizando a abordagem *Orthogonal Defect Classification (ODC)*, a metodologia provou ser eficiente na categorização dos bugs, auxiliando na compreensão das falhas mais recorrentes e do impacto destas nos serviços de nuvem.

6.3.2 Avaliação de Dependabilidade com a Ferramenta FIES

No segundo estudo de caso, a ferramenta **FIES** foi empregada na emulação de 34 bugs selecionados, o que possibilitou uma análise aprofundada da dependabilidade. As observações revelaram que o componente *Nova* é suscetível a comportamentos de *hang*, *crash* e *incorrect functionality* como resultado das falhas injetadas. Notavelmente, o componente *Neutron* permaneceu resiliente frente às falhas *SCL*, *SCM* e *SCH*, enquanto o *Keystone* manifestou comportamentos de *silent crash* sob as mesmas condições. Ademais, falhas no *Nova* impactaram o *Glance*, que exibiu *hang*, *crash* e *incorrect functionality*. Tanto o *Neutron* quanto o *Keystone* demonstraram *incorrect functionality* quando confrontados com falhas originadas no *Nova*.

6.3.3 Implicações e Aplicações Práticas

Os resultados advindos destes estudos de caso sugerem que a metodologia desenvolvida pode ser uma ferramenta valiosa para pesquisadores e profissionais do mercado que buscam um método sistemático para a classificação de bugs em ambientes de nuvem privada. Além disso, a ferramenta **FIES** para a injeção de falhas mostrou-se eficaz em ambientes operacionais, permitindo a avaliação detalhada da dependabilidade dos módulos de nuvem.

A robustez da metodologia e a eficiência da ferramenta **FIES** na simulação de condições adversas são de grande relevância para o avanço das práticas de segurança e confiabilidade em ambientes de nuvem privada, reforçando a importância da preparação contra falhas potenciais e da manutenção da integridade do sistema em condições desafiadoras.

7 Conclusão

A avaliação da dependabilidade em ambientes de nuvem privada representa um desafio emergente, especialmente diante da expansão acelerada desses sistemas em diversos setores. A presente dissertação abordou essa questão propondo uma metodologia inovadora e aplicando uma ferramenta robusta para classificar defeitos e injetar falhas.

O método *Orthogonal Defect Classification (ODC)*, adotado neste trabalho, demonstrou ser uma estratégia de classificação de defeitos extremamente eficaz, alinhando-se perfeitamente com as necessidades de ambientes de nuvem privada. A utilização do *ODC* forneceu uma base sólida para a identificação e o entendimento das falhas, estabelecendo uma ponte entre as diversas fases de desenvolvimento de software e a realidade operacional dos ambientes em nuvem.

A ferramenta **FIES** representou um salto qualitativo para a injeção de falhas, superando as limitações das ferramentas existentes que tendem a ser restritas a tecnologias específicas e a um conjunto limitado de falhas. Com sua flexibilidade de comunicação via protocolos *HTTP*, *SSH*, e *IP*, e a capacidade de realizar injeções de falhas dinâmicas, o **FIES** se mostrou capaz de emular falhas de forma realista, permitindo uma avaliação precisa da dependabilidade.

Este estudo também identificou a lacuna existente entre a classificação de bugs e a avaliação da dependabilidade devido à prevalência de defeitos não resolvidos. O desenvolvimento da ferramenta **FIES** veio preencher essa lacuna, possibilitando a emulação de defeitos previamente classificados para avaliar seu impacto real nos sistemas em nuvem.

A metodologia foi posta à prova através de estudos de caso meticulosamente desenhados. No âmbito do *OpenStack*, a classificação dos defeitos proporcionou uma visão clara das vulnerabilidades nos módulos e serviços. Com a injeção de falhas utilizando o **FIES**, foi possível transcender a teoria e observar o comportamento da nuvem sob condições adversas, conduzindo a uma avaliação concreta da dependabilidade.

Os resultados alcançados confirmam que a metodologia proposta, junto com a ferramenta **FIES**, oferecem recursos valiosos para pesquisadores e operadores de nuvens privadas. As descobertas deste trabalho contribuem significativamente para a compreensão das dinâmicas de falhas e para o desenvolvimento de estratégias mais robustas de manutenção e garantia da dependabilidade em ambientes de nuvem privada.

7.1 Contribuições

As contribuições desta dissertação estendem-se em duas dimensões principais: o desenvolvimento de uma metodologia inédita de classificação de defeitos específica para nuvens privadas e a criação de uma ferramenta web avançada para a injeção e análise de falhas em tempo real.

7.1.1 Metodologia de Classificação de Defeitos para Nuvens Privadas

A primeira e mais significativa contribuição deste estudo é a proposição e validação de uma metodologia de classificação de defeitos adaptada para as complexidades e especificidades dos ambientes de nuvem privada. Esta metodologia não só aprimora o entendimento semântico dos relatórios de erros fornecidos pelos usuários, mas também estabelece um protocolo para desvendar as inter-relações e dependências intrínsecas dos serviços de nuvem. Ela oferece um quadro analítico para a categorização precisa de defeitos, o que, por sua vez, informa estratégias de mitigação e reforça a robustez do ambiente de nuvem.

7.1.2 Ferramenta FIES para Injeção e Análise de Falhas

A segunda contribuição substancial é o desenvolvimento da ferramenta **FIES**, uma plataforma web inovadora para emular e monitorar falhas em ambientes de nuvem privada. A ferramenta excede as funcionalidades convencionais de softwares semelhantes ao permitir a injeção dinâmica de falhas, junto com a capacidade de capturar e reportar mensagens de erro em tempo real. A **FIES** possibilita uma análise detalhada do impacto das falhas, permitindo aos administradores de sistemas e desenvolvedores rastrear e mapear a cascata de efeitos adversos nos diversos componentes e serviços da nuvem. Este nível de introspecção é fundamental para o fortalecimento da dependabilidade em nuvens privadas, pois facilita a identificação de pontos vulneráveis e auxilia na prevenção de interrupções de serviço.

Em conjunto, essas contribuições delineiam um avanço metodológico e prático na gestão de falhas e na manutenção da integridade de sistemas de nuvem privada, alinhando-se com a necessidade emergente de soluções mais resilientes e confiáveis nesse domínio em crescimento.

7.2 Limitações

A principal limitação enfrentada neste estudo decorre do mecanismo de geração de falhas empregado pela ferramenta **FIES**. Atualmente, **FIES** depende de comandos nativos do shell (*bash*) para simular falhas no sistema operacional. Esta abordagem, embora eficaz em replicar uma variedade de condições de falha, não abrange a totalidade do espectro de defeitos possíveis, particularmente aqueles que poderiam surgir de mutações no código-fonte do software.

Idealmente, a ferramenta poderia ser aprimorada para incorporar técnicas de teste de mutação automatizadas. Isso permitiria que **FIES** simulasse um conjunto mais amplo e complexo de falhas sem a necessidade de intervenção manual para alterar o código, proporcionando uma avaliação mais abrangente da dependabilidade do sistema. A inclusão de testes de mutação automatizados poderia oferecer insights adicionais sobre a resiliência do sistema frente a defeitos mais sutis e variações de falhas, resultando em uma compreensão mais profunda da estabilidade do sistema em diversos cenários operacionais.

7.3 Trabalhos Futuros

Embora esta dissertação tenha contribuído com uma ferramenta robusta de injeção de falhas e uma metodologia inovadora para a classificação de *bugs* em ambientes de computação em nuvem privada, as possibilidades de extensão deste trabalho são amplas e promissoras. Entre elas, destacam-se:

- A integração de funcionalidades de teste de mutação na ferramenta **FIES**, que permitiria automatizar a avaliação da resistência de plataformas de computação em nuvem privada a variações de falhas. Essa automação poderia analisar a estrutura sintática do código-fonte das aplicações alvo, identificar e classificar blocos de código com semântica similar para evitar a geração de mutantes redundantes ou equivalentes.
- A aplicação de algoritmos avançados de decisão e seleção de operadores de mutação, visando otimizar a cobertura de testes e o impacto no desempenho computacional, será fundamental para a eficiência do processo de criação de mutantes.
- A automação do processo de construção de datasets, atualmente realizado por meio da metodologia proposta, poderia ser significativamente aprimorada. Utilizando ferramentas de extração de dados, informações relevantes poderiam ser coletadas diretamente dos repositórios de *bugs*, através da interpretação de URLs.

- Por fim, a tarefa de análise e rotulação das descrições de defeitos, que atualmente depende da intervenção humana, poderia ser automatizada. Empregando técnicas de Processamento de Linguagem Natural (PLN) e algoritmos de classificação baseados em inteligência artificial, a precisão e a eficiência desse processo seriam notavelmente incrementadas.

Essas extensões não apenas aprofundariam o entendimento dos defeitos em sistemas de computação em nuvem, mas também melhorariam a agilidade e a precisão da avaliação de dependabilidade desses sistemas críticos.

Referências

ABOHAMAMA, A.; ALRAHMAWY, M.; ELSOUD, M. A. Improving the dependability of cloud environment for hosting real time applications. **Ain Shams Engineering Journal**, v. 9, n. 4, p. 3335–3346, 2018. ISSN 2090-4479. Disponível em: <https://www.sciencedirect.com/science/article/pii/S2090447917301466>.

AGNELO, J.; LARANJEIRO, N.; BERNARDINO, J. Using orthogonal defect classification to characterize nosql database defects. **Journal of Systems and Software**, v. 159, p. 110451, 2020. ISSN 0164-1212. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121219302250>.

ALHAISONI, M. Migrating in-house data center to private cloud: Acase study. 2015.

ALI, S. R. Software reliability analysis. In: ____. **Next Generation and Advanced Network Reliability Analysis: Using Markov Models and Software Reliability Engineering**. Cham: Springer International Publishing, 2019. p. 59–104. ISBN 978-3-030-01647-0. Disponível em: <https://doi.org/10.1007/978-3-030-01647-0_3>.

ALIJANI, G. S.; FULK, H. K.; OMAR, A.; TULSI, R. Cloud computing effects on small business. **The Entrepreneurial Executive**, Jordan Whitney Enterprises, Inc, v. 19, p. 35, 2014.

ANBALAGAN, P.; VOUK, M. On mining data across software repositories. In: **2009 6th IEEE International Working Conference on Mining Software Repositories**. [S.l.: s.n.], 2009. p. 171–174.

AVIZIENIS, A.; LAPRIE, J. .; RANDELL, B.; LANDWEHR, C. Basic concepts and taxonomy of dependable and secure computing. **IEEE Transactions on Dependable and Secure Computing**, v. 1, n. 1, p. 11–33, 2004.

BAUER E.; ADAMS, R. **Reliability and Availability of Cloud Computing**. [S.l.]: Wiley-IEEE Press, Hoboken. ISBN 1118177010, 2012.

BENSO, A.; PRINETTO, P. **Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation**. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2010. ISBN 1441953914.

BOULANGER, J.-L. 3 - principle of dependability. In: BOULANGER, J.-L. (Ed.). **Certifiable Software Applications 1**. Elsevier, 2016. p. 43–64. ISBN 978-1-78548-117-8. Disponível em: <https://www.sciencedirect.com/science/article/pii/B9781785481178500039>.

BRIDGE, N.; MILLER, C. Orthogonal defect classification using defect data to improve software development. In: . [S.l.: s.n.], 1998.

BROCKMEIER, J. **It's Not Highlander: There Can Be More Than One Open Source Cloud.** 2012. <http://readwrite.com/2012/04/06/its-not-highlander-there-can-b/>. Online; Acessado em: 06/03/2021.

BUTCHER, M.; MUNRO, H.; KRATSCHMER, T. Improving software testing via odc: Three case studies. **IBM Systems Journal**, v. 41, n. 1, p. 31–44, 2002.

CENTOS. The centos project. In: ____. [s.n.], 2019. p. Acesso em: 20 de fev. 2023. Disponível em: <https://www.centos.org>.

CHEN, H.; DOU, W.; WANG, D.; QIN, F. Cofi: Consistency-guided fault injection for cloud systems. In: **2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)**. [S.l.: s.n.], 2020. p. 536–547.

CHILLAREGE, R. Trigger and impact profiles enable outcome focused defect discovery to manage software quality. In: **2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)**. [S.l.: s.n.], 2017. p. 245–251.

CHILLAREGE, R.; BHANDARI, I. S.; Chaar, J. K.; Halliday, M. J.; Moebus, D. S.; Ray, B. K.; Wong, M. . Orthogonal defect classification-a concept for in-process measurements. **IEEE Transactions on Software Engineering**, v. 18, n. 11, p. 943–956, 1992.

COTRONEO, D.; SIMONE, L. D.; LIGUORI, P.; NATELLA, R.; BIDOKHTI, N. Failviz: A tool for visualizing fault injection experiments in distributed systems. In: **2019 15th European Dependable Computing Conference (EDCC)**. [S.l.: s.n.], 2019. p. 145–148.

COTRONEO, D.; SIMONE, L. D.; LIGUORI, P.; NATELLA, R. Profipy: Programmable software fault injection as-a-service. In: **2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)**. [S.l.: s.n.], 2020. p. 364–372.

DURAES, J. A.; MADEIRA, H. S. Emulation of software faults: A field data study and a practical approach. **IEEE Transactions on Software Engineering**, v. 32, n. 11, p. 849–867, 2006.

ELMORSHIDY, A. Cloud computing: A new success model. In: **2019 7th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)**. [S.l.: s.n.], 2019. p. 7–12.

FEINBUBE, L.; PIRL, L.; TROGER, P.; POLZE, A. Dependability stress testing of cloud infrastructures. In: **2017 18th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)**. [S.l.: s.n.], 2017. p. 453–460.

GUAN, Q.; CHIU, C.-C.; FU, S. Cda: A cloud dependability analysis framework for characterizing system dependability in cloud computing infrastructures. In: **2012 IEEE 18th Pacific Rim International Symposium on Dependable Computing**. [S.l.: s.n.], 2012. p. 11–20.

HERNáNDEZ-GONZáLEZ, J.; RODRIGUEZ, D.; INZA, I.; HARRISON, R.; LOZANO, J. A. Learning to classify software defects from crowds: A novel approach. **Applied Soft Computing**, v. 62, p. 579–591, 2018. ISSN 1568-4946. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1568494617306610>.

HU, X.; LIU, J. Orthogonal defect classification-based ontology construction and application of software-hardware integrated error pattern of software-intensive systems. In: **2021 23rd International Conference on Advanced Communication Technology (ICACT)**. [S.l.: s.n.], 2021. p. 1286–1298.

JOHNSON, B. W. **Design amp; Analysis of Fault Tolerant Digital Systems**. USA: Addison-Wesley Longman Publishing Co., Inc., 1988. ISBN 0201075709.

JOSHI, N.; SHAH, S. A comprehensive survey of services provided by prevalent cloud computing environments. In: SATAPATHY, S. C.; BHATEJA, V.; DAS, S. (Ed.). **Smart Intelligent Computing and Applications**. Singapore: Springer Singapore, 2019. p. 413–424.

KUMAR, S.; SHARMA, M.; SINGH, V. B.; MUTTOO, S. K. Bug report classification into orthogonal defect classification defect type using long short term memory. In: **2021 3rd International Conference on Advances in Computing, Communication Control and Networking (ICAC3N)**. [S.l.: s.n.], 2021. p. 285–287.

KUO, W.; ZUO, M. **Optimal Reliability Modeling: Principles and Applications. John Wiley Sons**. [S.l.: s.n.], 2003. 560 p.

LLORENTE, I. M. **Eucalyptus, CloudStack, OpenStack and OpenNebula: A Tale of Two Cloud Models.** 2013. <https://opennebula.io/eucalyptus-cloudstack-openstack-and-opennebula-a-tale-of-two-cloud-models/>. Online; Acessado em: 11/03/2021.

MACIEL, P.; TRIVEDI, K.; MATIAS, R.; KIM, D. S. Dependability modeling. In: ____. **Performance and Dependability in Service Computing: Concepts, Techniques and Research Directions**. Hershey, PA, USA: IGI Global, 2012. p. 53–97.

MARINESCU, C. D. **Cloud Computing Theory and Practice**. 2. ed. Cambridge, MA, USA: Morgan Kaufmann, 2018.

MATOS, R.; ARAUJO, J.; ALVES, V.; MACIEL, P. Experimental evaluation of software aging effects in the eucalyptus elastic block storage. In: **2012 IEEE International Conference on Systems, Man, and Cybernetics (SMC)**. [S.l.: s.n.], 2012. p. 1103–1108.

MEI-CHEN HSUEH; TSAI, T. K.; IYER, R. K. Fault injection techniques and tools. **Computer**, v. 30, n. 4, p. 75–82, 1997.

MULLERIKKAL, J. P.; SASTRI, Y. A comparative study of openstack and cloudstack. In: **2015 Fifth International Conference on Advances in Computing and Communications (ICACC)**. [S.l.: s.n.], 2015. p. 81–84.

NIST. **The nist definition of cloud computing. Computer Security Division, Informartion Technology Laboratory, National Institute of Standards and Technology Gaithersburg**. [S.l.: s.n.], 2021. Online; Acessado em: 17/04/2021.

OLIVEIRA, D.; DANTAS, J.; ROSA, N.; MACIEL, P.; MATOS, R.; BRINKMANN, A. A dependability and cost optimisation method for private cloud infrastructures. **International Journal of Web and Grid Services**, v. 15, p. 367, 01 2019.

OPENSTACK. Openstack train release extends security and data protection, adds new ai and machine learning support. In: ____. [s.n.], 2019. p. Acesso em: 20 de fev. 2023. Disponível em: <https://www.openstack.org/software/train/>.

____. **What is OpenStack?** 2021. <https://www.openstack.org/software/>. Online; Acessado em: 06/03/2021.

PATTANAYAK, B. K.; HOTA, N.; MISHRA, J. P. A systematic overview of fault tolerance in cloud computing. **Intelligent and Cloud Computing: Proceedings of ICICC 2019, Volume 2**, Springer, p. 13–21, 2020.

PINKHAM, A. In: **Django Unleashed**. [S.l.: s.n.], 2016. p. 8.

RITTINGHOUSE, J.; RANSOME, J. **Cloud Computing: Implementation, Management, and Security**. 1st. ed. USA: CRC Press, Inc., 2009. ISBN 1439806802.

SUN. **Introduction to Cloud Computing Architecture**. [S.l.]: Sun Microsystems, Inc, 2009.

TIAN, N.; SAAB, D.; ABRAHAM, J. A. Esift: Efficient system for error injection. In: **2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS)**. [S.l.: s.n.], 2018. p. 201–206.

WANG, G.; CHEN, S.; LIU, J. An environment-aware anomaly detection framework of cloud platform for improving its dependability. In: THE STEERING COMMITTEE OF THE WORLD CONGRESS IN COMPUTER SCIENCE, COMPUTER .... **Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)**. [S.l.], 2015. p. 431.

XIAOYONG, Y.; YING, L.; ZHONGHAI, W.; TIANCHENG, L. Dependability analysis on open stack iaas cloud: Bug anaysis and fault injection. In: **2014 IEEE 6th International Conference on Cloud Computing Technology and Science**. [S.l.: s.n.], 2014. p. 18–25.

ZHAO, K.; JIN, H.; ZOU, D.; DAI, W. Onhelp: Components in building secure cloud based on opennebula. In: **2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications**. [S.l.: s.n.], 2014. p. 320–327.