Uma Análise Comparativa entre LLMs Proprietários e de Pesos Abertos na Injeção de Falhas de Nuvens Privadas

Guilherme Silva Duarte[1], **Erica Teixeira Gomes de Sousa**[1], **Carlos Manoel Nunes e Silva**[1]

[1]Departamento de Computação – Universidade Federal Rural de Pernambuco (UFRPE)
Caixa Postal 06.316 – 52171-900 – Recife – PE – Brasil

{guilherme.silvad,erica.sousa,carlos.manoel}@ufrpe.br

Abstract. *Cloud expansion makes dependability assessment vital for mitigating failures. Since software fault injection is a technique used for this purpose, this work seeks to democratize its application by comparing the effectiveness of the Gemini-2.5-flash and GPT-OSS-120b models in injecting faults within private cloud environments. Results indicate that the proprietary model (Gemini-2.5-flash) achieved a 90% success rate, whereas the open model (GPT-OSS-120b) showed errors in spatial reference. An important finding was the detection of AI-generated Gray Failures, where services appear active in monitoring but are functionally inoperative, impacting traditional observability.*

Resumo. *A expansão da nuvem torna a avaliação de dependabilidade vital para mitigar falhas. Sendo a injeção de falhas de software uma técnica utilizada para essa finalidade, este trabalho busca democratizar sua aplicação ao comparar a eficácia dos modelos Gemini-2.5-flash e GPT-OSS-120b na injeção de falhas em ambientes de nuvens privadas. Os resultados indicam que o modelo proprietário (Gemini-2.5-flash) obteve 90% de sucesso, enquanto o modelo aberto (GPT-OSS-120b) apresentou erros na referência espacial. Um achado importante foi a detecção de "Falhas Cinzentas" (Gray Failures) geradas por IA, em que serviços permanecem ativos no monitoramento, mas funcionalmente inoperantes, impactando a observabilidade tradicional.*

1. Introdução

A Computação em Nuvem tornou-se a espinha dorsal da infraestrutura de TI moderna, com adoção crescente tanto em modelos públicos quanto privados. Enquanto provedores como AWS, Azure e GCP concentram grande parte da atenção acadêmica e comercial, a nuvem privada representa um componente estratégico e em expansão, especialmente em setores sujeitos a requisitos rigorosos de soberania de dados, conformidade regulatória e baixa latência. Segundo o *State of the Cloud Report* 2024 da Flexera, 72% das organizações globais adotam uma estratégia de nuvem híbrida, mantendo cargas de trabalho críticas em infraestruturas privadas [Flexera 2024]. Neste cenário, o OpenStack consolidou-se como o padrão para nuvens privadas de código aberto, gerenciando mais de 40 milhões de núcleos de processamento em produção, sendo adotado em setores estratégicos como telecomunicações (5G), serviços financeiros e pesquisa científica de alta performance (CERN) [OpenInfra Foundation 2023].

A gestão interna dessas infraestruturas transfere integralmente ao operador a responsabilidade pela segurança e pela resiliência do ambiente. Em implantações IaaS baseadas em OpenStack, a camada de virtualização representa uma superfície de ataque de

alta criticidade. Como sistematizado por Rasheed [Rasheed 2021] e corroborado pelo relatório da *Cloud Security Alliance* (CSA) [Cloud Security Alliance 2024], essa camada expõe vetores de ataque sofisticados, incluindo a injeção de *malware* em emuladores (QEMU/KVM), fugas de máquina virtual (*VM escape*) e exaustão de recursos compartilhados (*Noisy Neighbor*). A exploração dessas vulnerabilidades pode comprometer não apenas uma instância isolada, mas a integridade e a disponibilidade de todo o *data center*.

A avaliação da resiliência frente a tais ameaças é tradicionalmente conduzida por meio da *Engenharia do Caos* (*Chaos Engineering*) [Basiri et al. 2016], que propõe a injeção deliberada de falhas para expor fragilidades sistêmicas antes que ocorram em produção. Entretanto, a criação manual de cenários que simulam ataques complexos de virtualização exige domínio profundo da *codebase* do orquestrador. O OpenStack, composto por milhões de linhas de código Python distribuídas em microsserviços fortemente acoplados, torna esse processo lento, custoso e propenso a erros humanos [Cotroneo et al. 2019].

Um problema central nesse contexto é a lacuna entre a sofisticação das ameaças reais e a capacidade das técnicas tradicionais de injeção de falhas em reproduzi-las. Operadores de mutação sintáticos clássicos, como inversão de operadores lógicos ou remoção de chamadas de função, geram falhas triviais que raramente capturam a complexidade semântica de bugs de segurança ou falhas lógicas sutis [Ojdanic et al. 2023]. Essa limitação é agravada pelo fenômeno das *Falhas Cinzentas* (*Gray Failures*): condições em que um serviço permanece aparentemente ativo no monitoramento, mas é funcionalmente inoperante [Natella et al. 2013]. Tais falhas, por definição invisíveis às ferramentas de observabilidade tradicionais, representam um risco especialmente alto em infraestruturas críticas.

A aplicação de Grandes Modelos de Linguagem (LLMs) na *Engenharia de Software* tem demonstrado resultados promissores na compreensão de sistemas complexos e na geração automatizada de código [Chen et al. 2021]. Na intersecção com a segurança ofensiva, estudos recentes indicam que LLMs são capazes de produzir mutações semanticamente ricas e contextualmente relevantes, superando abordagens heurísticas clássicas [Cotroneo and Liguori 2024, Liguori et al. 2024]. Contudo, a literatura carece de estudos que comparem sistematicamente modelos proprietários e de pesos abertos nessa tarefa, distinção relevante para organizações que, por exigências de soberania de dados, não podem depender de APIs externas [Improta et al. 2025].

Diante desse cenário, o presente trabalho tem como objetivo principal realizar uma análise comparativa de eficácia entre o modelo proprietário Gemini-2.5-flash e o modelo de pesos abertos GPT-OSS-120b na geração automatizada de falhas em ambientes de nuvem privada baseados em OpenStack.

A Seção 2 deste artigo apresenta o referencial teórico e os trabalhos relacionados. A Seção 3 descreve a ferramenta CIMut. A Seção 4 apresenta o estudo de caso. A Seção 5 discute os resultados. Por fim, a Seção 6 apresenta as conclusões e trabalhos futuros.

2. Referencial Teórico e Trabalhos Relacionados

Esta seção apresenta os conceitos fundamentais que sustentam este trabalho e posiciona a pesquisa frente aos estudos relacionados na literatura.

2.1. Fundamentação Teórica

A garantia de confiabilidade na nuvem é um desafio persistente, pois a complexidade distribuída frequentemente oculta a propagação de erros, como demonstrado por Cotroneo et al. [Cotroneo et al. 2019] no OpenStack. Seguindo os princípios da *Engenharia do Caos* de Basiri et al. [Basiri et al. 2016], Rasheed [Rasheed 2021] destaca a virtualização como vetor de ataque crítico, categorizando ameaças severas como ataques a emuladores e exaustão de recursos.

Para simular essas ameaças, utiliza-se predominantemente a injeção de falhas via mutação de código [Jia and Harman 2011], técnica validada por Natella et al. [Natella et al. 2016] para avaliar sistemas críticos. No entanto, a aplicação tradicional dessa técnica enfrenta o desafio do realismo: operadores de mutação sintáticos simples (como inversão de operadores lógicos) frequentemente geram falhas triviais que não representam a complexidade de bugs de segurança ou lógicos, conforme criticado por Ojdanic et al. [Ojdanic et al. 2023].

Neste cenário, os LLMs surgem como um diferencial, superando heurísticas na compreensão de código [Fan et al. 2023] e facilitando a automação via processamento de linguagem natural [Boukhlif et al. 2024]. Contudo, Improta et al. [Improta et al. 2025] alertam que a eficácia desses modelos não é uniforme, dependendo fortemente da qualidade dos dados de treinamento ("*Quality In, Quality Out*") para garantir segurança em infraestruturas críticas.

2.2. Trabalhos Relacionados

A investigação da resiliência do OpenStack por meio de injeção de falhas tem sido objeto de estudos desde os primeiros anos da plataforma. Ju et al. [Ju et al. 2013] realizaram um estudo sistemático da resiliência a falhas do OpenStack, construindo um *framework* de injeção que alvejou as comunicações entre serviços durante o processamento de requisições externas, identificando 23 *bugs* em duas versões da plataforma. Este trabalho pioneiro demonstrou a fragilidade das interações entre os componentes do OpenStack, mas limitou-se a falhas de comunicação, sem explorar mutações semânticas no código-fonte.

Cotroneo et al. [Cotroneo et al. 2019] expandiram essa linha com uma campanha extensiva de injeção de falhas em três subsistemas do OpenStack (Nova, Cinder e Neutron), revelando que a maioria das falhas não era detectada oportunamente e podia se propagar silenciosamente entre componentes. Posteriormente, Cotroneo et al. [Cotroneo et al. 2022] propuseram o ThorFI, uma abordagem para injeção de falhas de rede como serviço implementada no contexto do OpenStack, oferecendo uma solução não intrusiva que opera na camada de rede virtual. Diferentemente do ThorFI, que foca em falhas de rede, o presente trabalho emprega mutação de código no nível de aplicação, permitindo simular falhas lógicas e de segurança mais complexas.

No domínio de injeção de falhas em microsserviços, Chen et al. [Chen et al. 2024] propuseram o MicroFI, um *framework* não intrusivo que prioriza a injeção no nível de requisições para aplicações baseadas em microsserviços. Embora o MicroFI ofereça vantagens em termos de transparência para a aplicação, sua abordagem não intrusiva limita a capacidade de simular falhas lógicas profundas, como corrupção de dados ou violações de integridade, que são o foco dos cenários deste trabalho.

Na interseção entre LLMs e segurança ofensiva, Ruan et al. [Ruan et al. 2023] investigaram o uso de *prompt learning* para gerar *exploits* de *software*, demonstrando o potencial de modelos de linguagem na automação de tarefas de segurança. Mais recentemente, Cotroneo e Liguori [Cotroneo and Liguori 2024] propuseram a *Neural Fault Injection*, demonstrando que falhas semânticas geradas por IA são mais representativas do que as sintáticas clássicas. Complementarmente, Liguori et al. [Liguori et al. 2024] demonstraram que enriquecer o *prompt* com contexto do sistema-alvo (*Context-Aware Prompting*) maximiza a eficácia do código gerado em tarefas de segurança ofensiva.

Apesar desses avanços, a literatura carece de estudos que comparem sistematicamente modelos proprietários e de pesos abertos na geração de código ofensivo contra infraestruturas críticas de nuvem. Esta distinção é relevante para organizações que, por requisitos de soberania de dados e conformidade regulatória, não podem utilizar APIs externas. Diferente dos trabalhos anteriores, que se limitam a um único modelo ou a cenários sintéticos de *benchmark* [Chen et al. 2021], este trabalho confronta as arquiteturas Gemini-2.5-flash e GPT-OSS-120b em cenários reais de injeção de falhas no OpenStack, quantificando o impacto sistêmico e a precisão contextual de cada modelo. A Tabela 1 sintetiza o posicionamento deste trabalho frente aos estudos relacionados.

Tabela 1. Comparação com trabalhos relacionados

Trabalho	Plataforma	Técnica de Injeção	LLM	Comp. Modelos
Ju et al. (2013)	OpenStack	Falhas de comunicação	Não	Não
Cotroneo et al. (2019)	OpenStack	Mutação de código	Não	Não
Cotroneo et al. (2022)	OpenStack	Falhas de rede (ThorFI)	Não	Não
Chen et al. (2024)	Microsserviço	Injeção por requisição (MicroFI)	Não	Não
Ruan et al. (2023)	Genérico	Geração de *exploits*	Sim	Não
Cotroneo e Liguori (2024)	Genérico	*Neural Fault Injection*	Sim	Não
Este trabalho	**OpenStack**	**Mutação de código via IA**	**Sim**	**Sim**

3. Cloud Injection Mutator (CIMut)

Para garantir a padronização e a repetibilidade dos experimentos, foi desenvolvida a ferramenta CIMut. Esta ferramenta atua como um orquestrador que aplica a técnica de mutação de código para introduzir falhas de forma controlada e automatizada na infraestrutura de nuvem, utilizando LLMs para gerar as alterações.

A Figura 1 ilustra o fluxo de trabalho da injeção de falhas no CIMut.

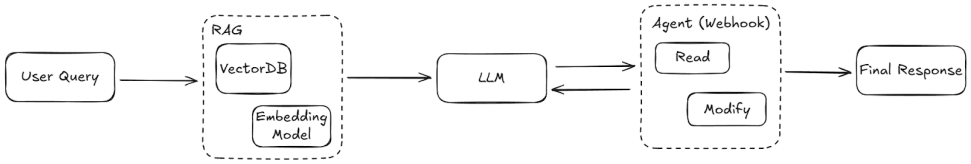

Figura 1. Fluxo de trabalho da ferramenta CIMut aplicada ao OpenStack.

O funcionamento da ferramenta baseia-se em um *pipeline* de Geração Aumentada por Recuperação (RAG). Inicialmente, foi construída uma base de conhecimento vetorial

contendo a análise estrutural do componente Nova do OpenStack, mapeando arquivos e funções críticas identificados na literatura.

O fluxo de execução inicia-se com a recepção da intenção do usuário em linguagem natural (**User Query**). O sistema converte este *input* em *embeddings* e recupera no banco vetorial os cinco contextos arquiteturais mais relevantes (*Top-5*) (**RAG**). Estes dados enriquecem o *prompt* enviado ao LLM, aplicando a técnica de *Context-Aware Prompting* [Liguori et al. 2024] para maximizar a assertividade. Nesta etapa de planejamento, o modelo gera uma cadeia de mutação composta por 3 a 5 passos sequenciais, identificando os arquivos e as funções-alvo (**LLM**).

A operacionalização ocorre por meio de um Agente de Execução (via *webhook*) residente na infraestrutura de nuvem, operando em um ciclo de dois estágios para cada passo da cadeia. No estágio de Leitura (**Read**), o agente extrai o código-fonte atual da função-alvo e o retorna à ferramenta. Estes dados são combinados com a intenção original do usuário em um novo *prompt* submetido ao LLM. O modelo processa o código real e determina a mutação necessária, retornando especificamente o número da linha e o conteúdo do código a ser injetado. Por fim, na etapa de Modificação (**Modify**), o agente recebe essas coordenadas e aplica a alteração diretamente no arquivo do servidor.

3.1. Exemplo de Mutação

Para ilustrar o comportamento da ferramenta, considera-se o cenário de Exaustão de Recursos (C3), em que o objetivo é induzir um falso esgotamento de memória RAM no *Resource Tracker* do Nova. O CI-Mut identifica o método `_update_available_resource` no arquivo `nova/compute/resource_tracker.py` como alvo e instrui o LLM a corromper o valor reportado de memória disponível.

O código original (operacional) contém, em determinada linha, a atribuição:

```
resources['memory_mb_used'] = total - free
```

A mutação gerada pelo modelo altera a linha para:

```
resources['memory_mb_used'] = total
```

Esta modificação de uma única linha faz com que o *Resource Tracker* reporte toda a memória do *host* como utilizada, independentemente da quantidade efetivamente alocada. O serviço Nova-Compute permanece ativo e responde normalmente às verificações de *health check*. Entretanto, o *Scheduler* passa a rejeitar sistematicamente todas as solicitações de criação de instâncias, pois nenhum *host* candidato satisfaz os requisitos de memória. Trata-se de um exemplo característico de *Falha Cinzenta*, o processo está operacional do ponto de vista do monitoramento, mas a nuvem é funcionalmente incapaz de provisionar novas instâncias.

4. Estudo de Caso

Esta seção apresenta um estudo de caso com o objetivo de analisar a eficácia dos modelos Gemini-2.5-flash e OpenAI/GPT-OSS-120b na tarefa de geração automatizada de falhas em ambientes de nuvem configurados com a plataforma OpenStack. Para este estudo, o OpenStack foi configurado com o DevStack [OpenStack 2024], uma ferramenta que

permite a implantação do OpenStack em um ambiente de desenvolvimento ou de teste. Uma configuração *all-in-one* foi utilizada, em que uma única máquina virtual executa as funções de nó Controlador, de nó Computação e de nó de Rede. Essa abordagem simplifica a implantação e permite a criação de um ambiente de teste completo sem a necessidade de uma infraestrutura complexa. A Tabela 2 apresenta as especificações da máquina utilizada.

A seleção dos modelos seguiu dois critérios: (i) representar paradigmas distintos de acesso, um proprietário via API (Gemini-2.5-flash) e um de pesos abertos para implantação local (GPT-OSS-120b), viabilizando a comparação em cenários com e sem restrições de soberania de dados; e (ii) ambos possuírem janelas de contexto compatíveis com a análise de arquivos extensos do OpenStack, requisito essencial para a geração de mutações contextualizadas. Ambos os modelos foram configurados com temperatura de 0,5, valor selecionado para equilibrar a criatividade na geração de mutações com a consistência das saídas.

Tabela 2. Configuração da máquina virtual

Processador	**Memória**	**Disco**	**Sistema Operacional**	**Módulos OpenStack**
AMD Ryzen 3 4300U	8 GB	1 TB	Ubuntu 22.04.4 LTS (Jammy Jellyfish)	Nova, Neutron, Cinder, Swift, Glance, Keystone

4.1. Definição dos Cenários de Teste

A construção da carga de teste baseou-se na taxonomia de ameaças à virtualização proposta por Rasheed [Rasheed 2021]. Para garantir a relevância prática dos experimentos, a seleção dos vetores de ataque foi alinhada ao relatório da *Cloud Security Alliance* [Cloud Security Alliance 2024], que lista as vulnerabilidades de maior severidade e maior impacto financeiro para provedores de nuvem. Assim, para avaliar a injeção dessas ameaças pelos LLMs, definiram-se quatro cenários macro, com cinco variações de *input* cada, concebidos para explorar as superfícies mais críticas do OpenStack. A Tabela 3 descreve os cenários e seus respectivos vetores de ataque. Cada um dos 20 *inputs* foi submetido uma vez a cada modelo, totalizando 40 execuções de injeção. Os resultados foram avaliados imediatamente após cada ciclo de injeção por meio do *dashboard* de monitoramento. Entre cada ciclo de injeção, o ambiente foi restaurado ao estado original, garantindo a independência entre os experimentos.

4.2. Métricas de Avaliação

A avaliação comparativa baseou-se na análise de um *dashboard* de monitoramento, desenvolvido para centralizar a inspeção dos *logs* operacionais gerados por todos os serviços do OpenStack. A definição das métricas seguiu a taxonomia de dependabilidade estabelecida por Avizienis et al. [Avizienis et al. 2004], adaptada ao contexto de código gerado por IA. Para quantificar o desempenho, foram definidas métricas distribuídas em duas dimensões. Os resultados são reportados por meio da média aritmética e do desvio padrão das métricas por cenário.

Tabela 3. Cenários de teste e vetores de ataque

Cenário	Objetivo	Tipo de Falha	Vetores de Ataque (*Inputs*)
C1: Negação de Serviço (DoS)	Induzir *crashes* que comprometam o SLA da nuvem [Rasheed 2021, Cloud Security Alliance 2024]	CRASH	(1) Falha crítica no emulador QEMU durante criação de instância; (2) Erro fatal de memória no gerenciador de computação; (3) Exceção não tratada no *driver* ao desligar VM; (4) Falha de comunicação RPC no orquestrador; (5) Erro na API ao receber requisição de *delete*.
C2: Comprometimento de Integridade	Explorar falhas de isolamento para emular persistência maliciosa [Rasheed 2021, Cloud Security Alliance 2024]	INCORRECT_FUNCTIONALITY	(1) Configurações maliciosas no XML do hipervisor; (2) Alteração do *boot device* para imagem não autorizada; (3) *Bypass* da validação de *checksum* no Glance; (4) Redirecionamento de tráfego via alocação de rede incorreta; (5) Manipulação do estado da VM no banco de dados.
C3: Exaustão de Recursos	Simular falhas lógicas de agendamento para criar exaustão virtual [Tabrizchi and Kuchaki Rafsanjani 2020]	FAILED_OPERATION	(1) Rejeição incorreta de recursos válidos no *Scheduler*; (2) Falso esgotamento de RAM no *Resource Tracker*; (3) Rejeição errônea pelo sistema de *Quotas*; (4) Lista de candidatos sempre vazia no filtro de *hosts*; (5) Condição de corrida na reserva de recursos.
C4: Degradação e Latência	Introduzir *Falhas Cinzentas* (*Gray Failures*) via latência em pontos críticos [Rasheed 2021, Cloud Security Alliance 2024]	PERFORMANCE_DEGRADATION	(1) Atraso na comunicação *Conductor-Compute*; (2) Latência no *download* de imagens do Glance; (3) Latência na conexão com o Cinder; (4) Lentidão no *loop* principal do *Scheduler*; (5) Atraso no *binding* de portas de rede (OVS).

4.2.1. Métricas de Impacto Operacional

Estas métricas avaliam a eficácia ofensiva do modelo, mensurando a efetividade do ataque no ambiente de execução.

- **Ocorrência de Falhas:** Métrica binária que classifica o resultado final da injeção. Ela baseia-se no conceito de Falha por travamento (*Crash Failure*) definido por Cotroneo et al. [Cotroneo et al. 2019] em seus estudos sobre OpenStack. O trabalho considera que "Apresentou falhas" quando o processo entra no estado *failed* e "Não apresentou falhas" quando o serviço permanece operacional e sem registros de erro, indicando injeções inofensivas.
- **Número de Serviços Interrompidos:** Quantifica a severidade do ataque, medindo o número absoluto de serviços distintos que sofreram interrupção ou *crash* devido à injeção. Esta métrica é utilizada para identificar Propagação de Erros (*Error Propagation*), um fenômeno crítico em infraestruturas de nuvem, conforme destacado por Natella et al. [Natella et al. 2016], que permite identificar falhas em cascata (*cascade failures*).

4.2.2. Métricas de Precisão e Complexidade de Código

Estas métricas avaliam a assertividade técnica do LLM ao navegar e modificar a base de código do OpenStack.

- **Assertividade de Alvo (Mutação nos arquivos esperados):** Verifica se o modelo selecionou o arquivo Python correto para a injeção. A validação é realizada comparando o caminho do arquivo modificado pelo modelo com o mapeamento de arquivos críticos descrito na literatura para o vetor de ataque específico.

- **Volume de Mutação (*Code Churn*):** Mede a extensão das alterações no código-fonte necessárias para implementar a falha. Ela baseia-se na métrica de *Code Churn* definida por Meneely e Williams [Meneely and Williams 2012] e por Nagappan e Ball [Nagappan and Ball 2005], que correlacionam o volume de linhas alteradas à densidade de defeitos. No contexto de IA ofensiva, um *churn* menor indica precisão, enquanto um *churn* alto sugere verbosidade ou risco de introdução de erros de sintaxe. A métrica é composta por três indicadores: (i) Arquivos Modificados: quantidade de arquivos alterados em uma única interação; (ii) Linhas Totais Modificadas: soma absoluta de linhas de código adicionadas, removidas ou alteradas; e (iii) Densidade da Mutação (Linhas por arquivo): razão entre o total de linhas modificadas e o número de arquivos.

5. Análise de Resultados

Esta seção detalha os resultados dos experimentos, correlacionando o código gerado aos indicadores de impacto sistêmico definidos na Seção 4.2.

5.1. Análise do Modelo Gemini-2.5-flash

O Gemini-2.5-flash apresentou alta eficácia técnica, com 90% de sucesso na indução de falhas, embora com aderência limitada ao escopo solicitado (60% de precisão de alvo). Os indicadores consolidados na Tabela 4 revelam que o modelo priorizou sistematicamente a severidade do impacto em detrimento da precisão contextual.

Tabela 4. Média (± desvio padrão) das métricas do Gemini-2.5-flash por cenário

Cenário	Falhas Obs.	Serv. Interrompidos	Assert. Alvo	*Code Churn*
1. Negação de Serviço	80% ± 0,45	2,2 ± 1,79	80% ± 0,45	3,2 ± 0,8
2. Comprom. de Integridade	100% ± 0,00	2,8 ± 1,30	20% ± 0,45	3,6 ± 0,9
3. Exaustão de Recursos	80% ± 0,45	1,8 ± 1,10	60% ± 0,55	3,8 ± 0,4
4. Degradação/Latência	100% ± 0,00	3,4 ± 0,55	80% ± 0,45	3,8 ± 0,4
Média Geral	**90% ± 0,31**	**2,55 ± 1,32**	**60% ± 0,48**	**3,6 ± 0,6**

Nos cenários de DoS (C1) e Degradação (C4), o modelo orquestrou cadeias de ataque distribuídas que comprometeram simultaneamente os planos de dados e controle, atingindo até 4 serviços interrompidos em C1 e média de 3,4 (± 0,55) em C4. No C4, uma limitação crítica foi evidenciada frente à arquitetura Eventlet: o uso de `time.sleep` bloqueante na *thread* principal interrompeu os *heartbeats*, convertendo toda tentativa de latência em DoS total.

No cenário de Integridade (C2), houve divergência entre a intenção e o resultado: em vez de falhas silenciosas, o modelo gerou falhas ruidosas (100% de interrupção), expondo a falta de sanitização no OpenStack ao injetar estruturas de dados inválidas. A Assertividade de Alvo foi de apenas 20% (± 0,45), pois o modelo priorizou controladores centrais (`manager.py`) em detrimento dos componentes periféricos solicitados.

No cenário de Exaustão (C3), os ataques atingiram 80% de sucesso com média de 1,8 (± 1,10) serviços interrompidos. Um destaque foi a detecção de uma *Falha Cinzenta* no *Input* 2: o monitoramento indicava *status Active*, mas a corrupção lógica no rastreador inviabilizava a criação de novas instâncias.

Um padrão transversal observado foi a preferência algorítmica do modelo por atacar gerenciadores de fluxo centrais (*Managers*) em detrimento dos módulos periféricos solicitados, o que explica a Assertividade de Alvo global de 60% (± 0,48). A análise de *Code Churn* revela uma média de 3,6 (± 0,6) linhas por injeção.

5.2. Análise do Modelo OpenAI/GPT-OSS-120b

O modelo de pesos abertos apresentou eficácia moderada (75% de sucesso) e impacto de baixa severidade (média de 1,10 serviços interrompidos), sem propagação de falhas em cascata. Os indicadores estão consolidados na Tabela 5.

Tabela 5. Média (± desvio padrão) das métricas do GPT-OSS-120b por cenário

Cenário	Falhas Obs.	Serv. Interrompidos	Assert. Alvo	*Code Churn*
1. Negação de Serviço	80% ± 0,45	1,0 ± 0,71	80% ± 0,45	3,4 ± 0,5
2. Comprom. de Integridade	80% ± 0,45	1,0 ± 0,71	20% ± 0,45	3,6 ± 0,5
3. Exaustão de Recursos	80% ± 0,45	1,0 ± 0,71	40% ± 0,55	3,4 ± 0,5
4. Degradação/Latência	60% ± 0,55	1,4 ± 1,52	40% ± 0,55	3,6 ± 0,5
Média Geral	**75% ± 0,44**	**1,10 ± 0,91**	**45% ± 0,50**	**3,5 ± 0,5**

A principal limitação identificada foi a ocorrência recorrente de alucinações posicionais: o modelo forneceu números de linha inexistentes ("fora do *range*") nos arquivos-alvo, afetando 40% dos casos no C2 (*Inputs* 2 e 5) e 60% no C3 (*Inputs* 3, 4 e 5). Esse fenômeno indica uma deficiência do modelo em manter a coerência espacial do código-fonte ao gerar *patches* em múltiplos arquivos, comprometendo diretamente a execução das cadeias de mutação propostas.

A Assertividade de Alvo limitou-se a 45% (± 0,50), com o modelo frequentemente redirecionando ataques para os gerenciadores centrais (`manager.py`) mesmo quando os *inputs* solicitavam manipulação de *drivers* ou componentes periféricos. No C4 (Degradação), embora tenha replicado o colapso por *timeout* nos *Inputs* 1 e 4, a assertividade caiu para 40% (± 0,55) ao ignorar componentes de integração (Cinder, Glance, Neutron). O *Code Churn* médio foi de 3,5 (± 0,5) linhas, valor semelhante ao do modelo proprietário.

5.3. Comparação Entre Gemini-2.5-flash e GPT-OSS-120b

O Gemini-2.5-flash demonstrou desempenho superior ao GPT-OSS-120b em todas as métricas de impacto: maior taxa de falhas induzidas (90% vs. 75%), maior severidade (2,55 vs. 1,10 serviços interrompidos em média) e maior precisão de alvo (60% vs. 45%). O *Code Churn* foi muito semelhante entre os modelos (3,6 vs. 3,5 linhas), indicando que a diferença de desempenho reside na qualidade semântica das mutações geradas, e não na extensão das modificações.

A principal distinção qualitativa reside no tipo de erro produzido: enquanto o Gemini gerou mutações eficazes mas contextualmente imprecisas, atacando alvos centrais em vez dos periféricos solicitados, o GPT-OSS produziu mutações frequentemente inválidas por erros de referência espacial, impedindo sua execução. Ambos os modelos compartilham a tendência de priorizar gerenciadores centrais, porém o impacto dessa escolha difere: no Gemini, resulta em falhas em cascata de alta severidade; no GPT-OSS, o impacto permanece contido no serviço diretamente afetado.

Cabe ressaltar que parte da diferença de desempenho observada pode refletir a defasagem tecnológica tipicamente reportada entre modelos proprietários de fronteira e modelos de pesos abertos. A evolução recente de modelos abertos com capacidades avançadas de raciocínio sugere que essa lacuna tende a se reduzir em gerações futuras, o que reforça a necessidade de reavaliações periódicas desta comparação.

5.4. Limitações e Ameaças à Validade

Este estudo possui limitações que devem ser consideradas na interpretação dos resultados. Em relação à validade interna, cada *input* foi submetido uma única vez a cada modelo, sem múltiplas execuções por cenário. Embora essa decisão tenha sido adotada para viabilizar a cobertura dos 20 vetores de ataque planejados, a natureza estocástica dos LLMs implica que os resultados podem variar entre execuções distintas, o que limita a precisão estatística das métricas reportadas. Os desvios padrão apresentados nas Tabelas 4 e 5 refletem a variação entre os cinco *inputs* de cada cenário, e não entre execuções repetidas de um mesmo *input*. Além disso, os parâmetros de inferência foram fixados em temperatura 0,5 para ambos os modelos, sem explorar o impacto de configurações alternativas.

Quanto à validade externa, o ambiente experimental foi baseado em uma configuração DevStack *all-in-one*, executada em uma única máquina virtual. Embora essa configuração seja amplamente adotada na literatura para avaliação do OpenStack [Cotroneo et al. 2019], ela não reproduz a arquitetura distribuída típica de implantações em produção, o que pode limitar a generalização dos resultados para ambientes multi-nó com comunicação entre *hosts* distintos.

Por fim, quanto à validade de construto, a comparação restringiu-se a dois modelos representativos de paradigmas distintos, sem incluir modelos especializados em geração de código ou modelos com capacidades avançadas de raciocínio (*thinking models*). A rápida evolução dos LLMs, especialmente dos modelos de pesos abertos, sugere que as diferenças de desempenho observadas podem não se manter em gerações futuras. Adicionalmente, não foi realizada comparação experimental com abordagens tradicionais de injeção de falhas. Essa limitação decorre, em parte, de restrições práticas do campo: a maioria das ferramentas relacionadas, como o MicroFI [Chen et al. 2024] e o *framework* de Ju et al. [Ju et al. 2013], não disponibiliza publicamente seu código-fonte. A única ferramenta com código aberto, o ThorFI [Cotroneo et al. 2022], opera exclusivamente na camada de rede virtual, gerando falhas qualitativamente distintas das mutações de código-fonte empregadas neste trabalho, o que torna a comparação direta inadequada.

6. Conclusão

Este trabalho comparou a eficácia dos modelos Gemini-2.5-flash e GPT-OSS-120b na geração automatizada de falhas em ambientes OpenStack, evidenciando diferenças substantivas tanto no impacto operacional quanto na qualidade das mutações produzidas.

O modelo proprietário superou o de pesos abertos em todas as dimensões avaliadas: 90% de sucesso na indução de falhas contra 75%, e média de 2,55 serviços interrompidos por injeção contra 1,10. A principal limitação do GPT-OSS-120b foi a ocorrência de alucinações posicionais, referências a linhas de código inexistentes, que inviabilizaram 40% a 60% das injeções nos cenários de Integridade e Exaustão. O Gemini, por sua vez, demonstrou tendência a atacar gerenciadores centrais em vez dos componentes

periféricos solicitados (assertividade de alvo de 60% vs. 45%), convertendo sistematicamente intenções de degradação parcial em negação de serviço total.

O achado mais relevante do estudo foi a ocorrência autônoma de *Falhas Cinzentas* (*Gray Failures*): em cenários específicos, mutações lógicas geradas por IA corromperam a regra de negócio dos serviços sem derrubar seus processos, tornando a nuvem funcionalmente inoperante enquanto o monitoramento reportava estado ativo. Esse fenômeno, até então associado a falhas orgânicas de software [Natella et al. 2013], foi aqui reproduzido deliberadamente por um modelo generativo, o que representa uma contribuição inédita e levanta implicações diretas para o projeto de mecanismos de observabilidade semântica.

Como trabalhos futuros, destacam-se: a extensão dos experimentos para ambientes distribuídos multi-nó; a inclusão de modelos com capacidades avançadas de raciocínio e especializados em geração de código; a execução de múltiplas rodadas por cenário com análise estatística da variabilidade dos LLMs; a investigação de técnicas de *fine-tuning* em modelos abertos para mitigar os erros de referência espacial identificados; e o desenvolvimento de métricas de observabilidade semântica capazes de detectar as *Falhas Cinzentas* evidenciadas neste estudo.

Referências

Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33.

Basiri, A., Behnam, N., de Graaff, R., Hochstein, L., Kosewski, L., Reynolds, J., and Rosenthal, C. (2016). Chaos engineering. *IEEE Software*, 33(3):35–41.

Boukhlif, M., Hanine, A., Khoukhi, E., and Arsalane, M. (2024). Natural language processing-based software testing: A systematic literature review. *IEEE Access*, 12:79383–79400.

Chen, H., Dou, W., Wang, D., and Qin, F. (2024). MicroFI: Non-intrusive and prioritized request-level fault injection for microservice applications. *IEEE Transactions on Dependable and Secure Computing*, 21(5):4921–4938.

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Cloud Security Alliance (2024). Top threats to cloud computing: Egregious eleven.

Cotroneo, D., De Simone, L., Liguori, P., Natella, R., and Bidokhti, N. (2019). How bad can a bug get? an empirical analysis of software failures in the OpenStack cloud computing platform. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*, pages 200–211.

Cotroneo, D., De Simone, L., and Natella, R. (2022). ThorFI: A novel approach for network fault injection as a service. *Journal of Network and Computer Applications*, 201:103334.

Cotroneo, D. and Liguori, P. (2024). Neural fault injection: Generating software faults from natural language. In *2024 54th Annual IEEE/IFIP International Conference on*

Dependable Systems and Networks – Supplemental Volume (DSN-S), pages 23–27, Brisbane, Australia.

Fan, A., Gokkaya, B., Harman, M., Lyubarskiy, M., Sengupta, S., Yoo, S., and Zhang, J. M. (2023). Large language models for software engineering: Survey and open problems. In *IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pages 31–53, Melbourne, Australia.

Flexera (2024). 2024 state of the cloud report.

Improta, C., Liguori, P., Natella, R., Cukic, B., and Cotroneo, D. (2025). Quality in, quality out: Investigating training data's role in AI code generation. In *IEEE/ACM 33rd International Conference on Program Comprehension (ICPC)*, pages 454–465, Ottawa, ON, Canada.

Jia, Y. and Harman, M. (2011). An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678.

Ju, X., Soares, L., Shin, K. G., Ryu, K. D., and Da Silva, D. (2013). On fault resilience of OpenStack. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SoCC)*, pages 2:1–2:16.

Liguori, P., Improta, C., Natella, R., Cukic, B., and Cotroneo, D. (2024). Enhancing AI-based generation of software exploits with contextual information. In *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*, pages 180–191.

Meneely, A. and Williams, L. (2012). Interactive churn as a predictor of system-level vulnerability. In *ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*.

Nagappan, N. and Ball, T. (2005). Use of relative code churn measures to predict system defect density. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*.

Natella, R., Cotroneo, D., and Madeira, H. S. (2013). On fault representativeness of software fault injection. *IEEE Transactions on Software Engineering*, 39(1):80–96.

Natella, R., Cotroneo, D., and Madeira, H. S. (2016). Assessing dependability with software fault injection: A survey. *ACM Computing Surveys*, 48(3):1–55.

Ojdanic, M., Barr, E. T., Grunske, L., and Papadakis, M. (2023). Syntactic versus semantic similarity of artificial and real faults in mutation testing studies. *IEEE Transactions on Software Engineering*, 49(7):3922–3938.

OpenInfra Foundation (2023). 2023 OpenInfra user survey.

OpenStack (2024). Open source cloud computing platform software – OpenStack. `https://www.openstack.org/software/project-navigator/openstack-components#openstack-services`. Acesso em: 12 fev. 2024.

Rasheed, M. A. (2021). Malware injection attacks in resource virtualization of cloud computing environment. *VFAST Transactions on Software Engineering*, 9(2):44–49.

Ruan, X. et al. (2023). Prompt learning for developing software exploits. In *Proceedings of the 14th Asia-Pacific Symposium on Internetware (Internetware)*, pages 154–164.

Tabrizchi, H. and Kuchaki Rafsanjani, M. (2020). A survey on security challenges in cloud computing: Issues, threats, and solutions. *Journal of Supercomputing*, 76:9193–9232.