Avaliação de Estratégias de Tolerância a Falhas em Middlewares MQTT para IoT

Tiago Luis Custódio[1] **e Luiz Antonio Rodrigues**[1]

[1] Programa de Pós-Graduação em Ciência da Computação (PPGComp)
Universidade Estadual do Oeste do Paraná (UNIOESTE)
Cascavel – PR, Brasil

{tiago.custodio, luiz.rodrigues}@unioeste.br

Abstract. *The Message Queuing Telemetry Transport (MQTT) is one of the leading communication protocols for the Internet of Things (IoT) applications due to its low resource consumption and the simplicity of the publish/subscribe model. However, operating MQTT middlewares in environments subject to network failures, traffic overload, and device heterogeneity requires fault-tolerance mechanisms that preserve reliability and availability under stress. This paper investigates resilience strategies for MQTT middleware by evaluating three approaches: Circuit Breaker, Active Replication, and a Staged Pipeline. The methodology employs scripts to simulate controlled fault scenarios, including intermittent instabilities, complete outages, and severe consumer slowdowns. Metrics such as message loss, duplication, latency, throughput, and CPU/memory usage are analyzed in an Edge-Cloud context. The results provide practical indicators to support architectural decisions in MQTT-based IoT systems, contributing to the design of more robust and fault-tolerant middleware.*

Resumo. *O Message Queuing Telemetry Transport (MQTT) é um dos principais protocolos de comunicação em aplicações de Internet das Coisas (IoT), devido ao baixo consumo de recursos e à simplicidade do modelo de publicação/assinatura. Entretanto, a operação de middlewares MQTT em cenários sujeitos a falhas de rede, sobrecarga de tráfego e heterogeneidade de dispositivos demanda mecanismos de tolerância a falhas que preservem a confiabilidade e a disponibilidade sob estresse. Este trabalho investiga estratégias de resiliência para middlewares MQTT, avaliando três abordagens: Circuit Breaker, Replicação Ativa e Pipeline por Estágios. A metodologia emprega scripts para simulação controlada de falhas, incluindo instabilidades intermitentes, quedas totais e degradação severa do consumidor. Foram analisadas métricas como perda, duplicação, latência, vazão e uso de recursos (CPU e memória), em um contexto arquitetural Edge-Cloud. Os resultados fornecem indicadores que apoiam decisões de projeto em sistemas IoT baseados em MQTT, contribuindo para o desenvolvimento de middlewares mais robustos e tolerantes a falhas.*

1. Introdução

O *Message Queuing Telemetry Transport* (MQTT) é um dos protocolos mais utilizados em Internet das Coisas (IoT) por sua eficiência no modelo *publish/subscribe*

[Banks e Gupta 2014, OASIS 2019]. Entretanto, em ambientes sujeitos a instabilidades de rede, variações de carga e heterogeneidade de dispositivos, o papel do *middleware* torna-se crítico, exigindo estratégias de tolerância a falhas que sustentem a comunicação de forma previsível [Falahah et al. 2021, Pastorio et al. 2020].

Este artigo avalia três abordagens de resiliência para *middlewares* MQTT: *Circuit Breaker*, Replicação Ativa e Pipeline por Estágios. As estratégias são testadas em três cenários típicos em IoT: *flapping* (instabilidade intermitente), *outage* (indisponibilidade total) e *slow consumer* (latência elevada). O objetivo é produzir indicadores para arquiteturas de microsserviços IoT, analisando o comportamento do *middleware* quanto a métricas de entrega, perda, duplicação, latência, vazão e consumo de recursos (CPU e memória).

A metodologia utiliza *scripts* Python para injeção controlada de falhas em um contexto *Edge-Cloud* [Ren et al. 2019]. Ao comparar padrões de resiliência consolidados sob condições equivalentes de estresse (em vez de propor um novo protocolo), este trabalho contribui para decisões arquiteturais mais informadas em sistemas que exigem disponibilidade e custo operacional mensuráveis.

O restante do texto apresenta fundamentação teórica (Seção 2), trabalhos relacionados (Seção 3), metodologia (Seção 4), resultados (Seção 5), discussão (Seção 6) e conclusões (Seção 7).

2. MQTT e Padrões de Projeto na Construção de Aplicações IoT

Esta seção apresenta os detalhes do protocolo MQTT e as estratégias de resiliência utilizadas na comparação prática do trabalho.

2.1. MQTT

O *Message Queuing Telemetry Transport* (MQTT) é um protocolo leve de mensagens, projetado para comunicação eficiente em ambientes com recursos limitados e redes instáveis, sendo amplamente utilizado em aplicações de Internet das Coisas (IoT) [Banks e Gupta 2014]. Seu modelo de comunicação baseia-se no paradigma publicação/assinatura (*publish/subscribe*), no qual clientes publicam mensagens em tópicos, e intermediários (*brokers*) distribuem esses dados aos assinantes, promovendo desacoplamento entre produtores e consumidores.

O MQTT foi originalmente desenvolvido pela IBM e pela Eurotech em 1999 para telemetria em conectividade limitada. Tornou-se padrão aberto com a especificação MQTT 3.1.1 pela OASIS, posteriormente adotada como ISO/IEC 20922 [ISO 2016]. Em 2019, a versão MQTT 5.0 [OASIS 2019] introduziu propriedades de mensagem, códigos de razão e melhorias de controle de fluxo, ampliando a capacidade de diagnóstico e interoperabilidade. Embora versões anteriores permaneçam comuns em dispositivos embarcados, recursos como *Reason Codes* permitem caracterizar falhas de forma mais granular, o que é relevante para estratégias de detecção e contenção no *middleware*.

O protocolo define três níveis de Qualidade de Serviço (QoS). Em *At most once* (QoS 0), a entrega ocorre por melhor esforço, sem confirmação, admitindo perdas. Em *At least once* (QoS 1), a entrega é garantida por retransmissões, mas duplicações podem ocorrer. Em *Exactly once* (QoS 2), garante-se entrega e processamento exatamente uma

vez, evitando perdas e duplicações, sendo indicado para aplicações críticas. Estes níveis são centrais na avaliação de estratégias que priorizam disponibilidade, durabilidade ou estabilidade sob falha.

A Figura 1 apresenta os principais componentes da arquitetura MQTT [OASIS 2019]. Os clientes publicam e/ou assinam tópicos (sensores, atuadores ou aplicações). O *broker* centraliza a mediação, sendo responsável pelo gerenciamento de tópicos, filtragem e roteamento de mensagens. Os tópicos organizam os fluxos e definem o escopo de assinatura, permitindo escalabilidade lógica do sistema.

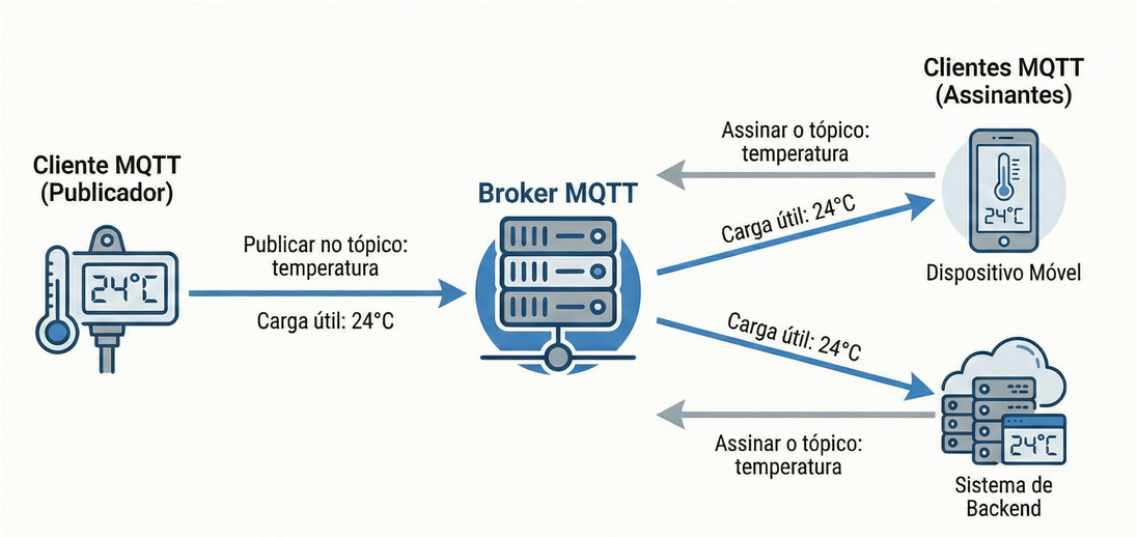

Figura 1. Arquitetura Publish/Subscribe do MQTT

O Mosquitto [Light 2017] é um *broker* MQTT leve e de código aberto, amplamente utilizado em aplicações de Internet das Coisas (IoT). O Mosquitto suporta múltiplos níveis de qualidade de serviço (QoS), mensagens retidas, autenticação, criptografia SSL/TLS e configuração de *bridges* entre *brokers*, tornando-o adequado tanto para ambientes experimentais quanto para sistemas industriais. Sua simplicidade, eficiência e compatibilidade com diversas plataformas o tornam uma escolha prática para desenvolvimento, teste e reprodutibilidade de experimentos.

2.2. Padrões para Tolerância a Falhas

Os *middlewares* implementados fundamentam-se em conceitos distintos de resiliência, definindo os parâmetros avaliados nos experimentos.

O *Circuit Breaker* (Disjuntor) baseia-se na transição entre três estados: *closed* (operação normal), *open* (falha detectada) e *half-open* (teste de recuperação), conforme a Figura 2. Seu propósito é reduzir falhas em cascata e limitar a propagação de instabilidades, permitindo recuperação controlada após eventos transitórios [Falahah et al. 2021], embora possa ser penalizado em interrupções prolongadas por descartar mensagens enquanto o circuito permanece aberto [Montesi e Weber 2018].

A Replicação Ativa fundamenta-se na redundância coordenada, em que múltiplas réplicas processam as mesmas requisições para garantir continuidade do serviço (Figura 3). A abordagem inspira-se em princípios de consenso, como o RAFT

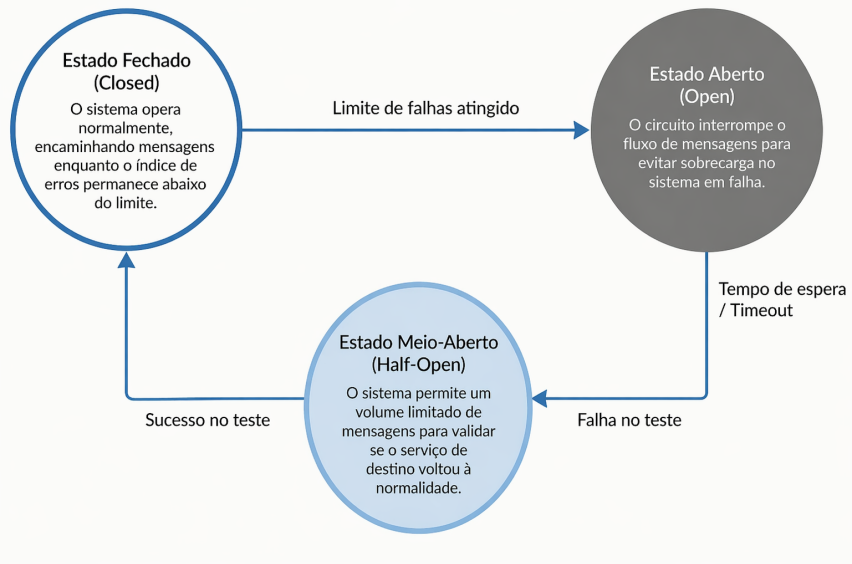

Figura 2. Exemplo de *Circuit Breaker*

[Ongaro e Ousterhout 2014], e em extensões como o MQTT-ST [Longo et al. 2020]. Contudo, a replicação contínua impõe overhead de coordenação e tráfego extra, podendo reduzir desempenho e eficiência em ambientes sem balanceamento adequado [Detti et al. 2020].

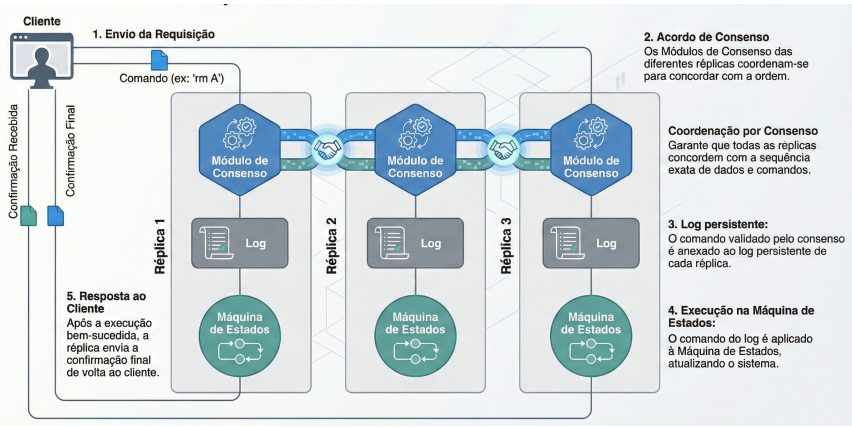

Figura 3. Exemplo de SMR

O Pipeline por Estágios adota uma arquitetura modular inspirada em *fog computing*, decompondo o fluxo em etapas independentes com desacoplamento temporal (ex: validação e transformação) [Mirampalli et al. 2023]. O modelo favorece resiliência localizada e reprocessamento controlado via filas, porém sua efetividade depende do controle de fluxo e da capacidade do consumidor, podendo elevar latência (p95) e demanda por armazenamento sob carga [Poojara et al. 2021].

Mecanismos complementares, como *Retry* com *Backoff* Exponencial e o padrão *Bulkhead* [Nygard 2007, Bass et al. 2012], oferecem camadas adicionais de proteção contra instabilidades e contenção de recursos, constituindo direções para trabalhos futuros.

3. Trabalhos Relacionados

Estudos recentes investigam limitações de desempenho e resiliência em *middlewares*, com ênfase em escalabilidade e confiabilidade sob falha. Detti et al. (2020) demonstraram que a expansão de *clusters* MQTT apresenta ganhos de vazão sub-lineares devido a gargalos na coordenação entre *brokers*. Para aumentar a robustez, Longo et al. (2020) propuseram o MQTT-ST, que utiliza mecanismos de consenso e quórum para garantir a entrega, ao custo de maior latência e processamento.

No campo de arquiteturas distribuídas, Naim et al. (2022) exploraram a cooperação entre *edge* e *cloud* para manter continuidade do serviço sob falhas parciais e detecção via monitoramento. Complementarmente, Hmissi e Ouni (2025) introduziram o TD-MQTT, conferindo transparência à localização de *brokers* em topologias dinâmicas, enquanto Chai et al. (2025) desenvolveram o DUA-MQTT para IoT, visando alta disponibilidade e interoperabilidade multiagente entre OPC UA, protocolo industrial, e MQTT.

Além disso, estudos sobre *middleware* tolerante a falhas destacam a existência de *trade-offs* intrínsecos entre desempenho e disponibilidade. Szentiványi (2005) demonstrou, por meio de avaliações empíricas, que mecanismos de tolerância a falhas incorporados ao *middleware* reduzem tempos de indisponibilidade e simplificam o desenvolvimento de aplicações, porém introduzem aumento de latência mesmo em condições sem falha. De forma complementar, Zhao et al. (2010) propuseram o *Low Latency Fault Tolerance* (LLFT), um *middleware* baseado em replicação líder-seguidor que busca manter consistência forte com baixa latência, evidenciando o custo associado à coordenação entre réplicas. Por fim, Dumitras e Narasimhan (2005) mostraram que uma pequena fração de eventos extremos pode ter domínio sobre o comportamento de latência e vazão em sistemas tolerantes a falhas, reforçando a importância de análises baseadas em percentis. Esses trabalhos fornecem evidências que a avaliação de *middlewares* tolerantes a falhas deve considerar simultaneamente métricas de desempenho e confiabilidade, motivando abordagens comparativas sob condições controladas, como a proposta neste estudo.

Diferente de abordagens que propõem novos protocolos ou arquiteturas específicas, este trabalho oferece uma análise comparativa sistemática entre *Circuit Breaker*, Replicação Ativa e Pipeline por Estágios. Avaliadas sob condições equivalentes e cenários de estresse idênticos, a contribuição reside em estabelecer uma base unificada de comparação que mensura simultaneamente ganhos de resiliência e custos operacionais, apoiando a escolha do padrão mais adequado ao perfil de falha esperado.

4. Metodologia

Foram implementados três *middlewares* independentes baseados em padrões consolidados de tolerância a falhas: *Circuit Breaker*, Replicação Ativa e Pipeline por Estágios. Cada middleware foi projetado como componente intermediário responsável por processar mensagens geradas por um produtor simulado e encaminhá-las a um consumidor final. O produtor foi implementado por meio de um *script* em Python que reproduz o comportamento de um dispositivo IoT, enviando uma sequência controlada de mensagens ao *middleware*. O sistema modela dispositivo, *middleware* e *backend* sob falhas de rede e temporização, abrangendo instabilidade, indisponibilidade e lentidão.

Diferentemente de uma abordagem centrada no produtor, as falhas foram injetadas no *downstream* do *middleware*, isto é, no lado do consumidor. Essa decisão visa

aproximar o experimento de cenários realistas em arquiteturas IoT, nas quais a instabilidade costuma se manifestar no *backend* (indisponível, intermitente ou degradado), e o *middleware* deve atuar como camada de resiliência ao proteger o fluxo de mensagens no sistema. Assim, cada solução reagiu a falhas simuladas no destino, permitindo comparar como diferentes estratégias impactam a continuidade do serviço. Falhas entre dispositivo e *middleware* podem ser consideradas em trabalhos futuros, mas a comparação atual é mais informativa no trecho Middleware→Backend.

Três cenários de falha reconhecidos em sistemas distribuídos foram considerados. O cenário *flapping* simula instabilidade intermitente do consumidor, alternando sucesso e falha de maneira periódica, representando reinicializações frequentes, perda momentânea de conectividade ou falhas curtas recorrentes. O cenário *outage* representa indisponibilidade completa e temporária do consumidor por um intervalo prolongado, avaliando a capacidade de absorção e recuperação de mensagens. O cenário *slow consumer* simula degradação severa de desempenho: o consumidor permanece acessível, porém responde de forma excessivamente lenta, gerando gargalos, acúmulo e *timeouts*.

A injeção de falhas foi realizada programaticamente no próprio *middleware*, garantindo reprodutibilidade e controle. No *flapping*, o consumidor alterna estados funcionais e falhos a cada conjunto fixo de mensagens. No *outage*, o consumidor torna-se indisponível após um determinado número de requisições e permanece inacessível até o final da execução. No *slow consumer*, atrasos artificiais de alta magnitude são introduzidos no processamento, representando um *backend* sobrecarregado. Em todos os casos, o fluxo de comunicação foi modelado com base em arquiteturas típicas de sistemas MQTT, de modo que o experimento avaliasse o comportamento do *middleware* frente à degradação do consumidor.

Para cada combinação de *middleware* e cenário, foram realizadas dez execuções independentes com carga idêntica, permitindo comparação direta entre as arquiteturas avaliadas. Durante cada execução, foram coletadas as seguintes métricas: número total de mensagens enviadas, número de mensagens entregues, taxa de perda, taxa de duplicação, número médio de cópias extras por mensagem, latência média e latência no percentil 95, vazão em mensagens por segundo, uso de CPU, pico de memória, tempo total de execução, tempo médio de recuperação do serviço e, no caso da replicação ativa, divergência entre réplicas. A coleta foi automatizada e registrada em arquivos CSV para posterior análise estatística. A repetição das execuções permitiu o cálculo de média e erro padrão das métricas, reduzindo a influência de variações ocasionais do sistema operacional durante os experimentos.

Os experimentos foram conduzidos em ambiente controlado composto por um computador Lenovo G460 com processador Intel Core i3 M370 de 2,4 GHz, 7,6 GB de memória RAM e sistema operacional Xubuntu 20.04.6 LTS. Todas as implementações foram desenvolvidas em Python utilizando bibliotecas padrão da linguagem para controle de execução e a biblioteca psutil para monitoramento de recursos computacionais.

A análise dos dados foi realizada utilizando as bibliotecas *Pandas* e *Matplotlib*, responsáveis pelo processamento dos arquivos CSV e pela geração de gráficos comparativos. Para cada cenário experimental, as métricas obtidas nas dez execuções foram agregadas e analisadas por meio de média e erro padrão. A partir desses dados, foram produzidos

três tipos de gráficos: (i) uso de CPU e memória, (ii) perda e duplicação de mensagens e (iii) vazão e latência. Essa organização permite analisar de forma sistemática os efeitos de cada estratégia de resiliência sobre custo operacional, desempenho e confiabilidade do sistema. Todo o código-fonte utilizado nos experimentos, bem como as instruções de execução e os conjuntos de resultados obtidos, encontram-se disponíveis publicamente no repositório do projeto: `https://github.com/tiago-lcustodio/AvaliacaoEstrategiasToleranciaFalhas`.

5. Resultados

Os experimentos permitiram comparar de forma objetiva o comportamento das três estratégias de *middleware* nos cenários de falha avaliados. Para cada combinação de cenário e arquitetura, foram analisadas métricas como perda, duplicação, latência, vazão e uso de recursos computacionais. Os resultados indicam respostas distintas sob estresse: a Replicação Ativa privilegia continuidade do serviço em instabilidade, o *Circuit Breaker* prioriza contenção e recuperação controlada, e o Pipeline por Estágios oferece durabilidade via desacoplamento temporal, condicionado ao desempenho do *backend*.

A Figura 4 sintetiza o consumo de recursos e evidencia o custo operacional de cada estratégia. A memória permanece praticamente constante entre os três *middlewares*, pois o volume de mensagens e as estruturas de dados do experimento são modestos para o ambiente local. A principal diferença ocorre no uso de CPU: a Replicação Ativa consome mais recursos devido ao envio redundante e coordenação; o Pipeline por Estágios apresenta custo intermediário por gerenciar filas e *retries*; e o *Circuit Breaker* registra o menor consumo ao reduzir chamadas quando entra no estado *OPEN*. O resultado destaca o *trade-off* entre disponibilidade imediata e custo computacional.

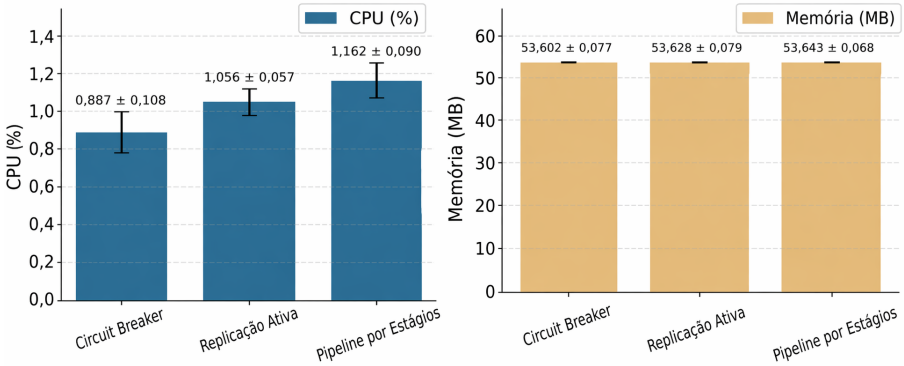

Figura 4. Consumo médio de CPU e pico de memória no cenário de *flapping* para cada estratégia de *middleware*.

Na Figura 5, observam-se respostas contrastantes no cenário de *flapping*. o *Circuit Breaker* apresenta perda moderada e duplicação nula, pois abre o circuito após falhas sucessivas e interrompe tentativas de entrega durante a instabilidade. A Replicação Ativa praticamente elimina perdas, porém gera alta duplicação devido ao envio simultâneo para múltiplas réplicas. O Pipeline por Estágios não apresenta perdas nem duplicações, sugerindo que o enfileiramento e o reprocessamento absorveram as oscilações do consumidor sem violar as métricas de integridade. O gráfico evidencia a diferença entre descarte controlado, redundância e resiliência temporal via filas.

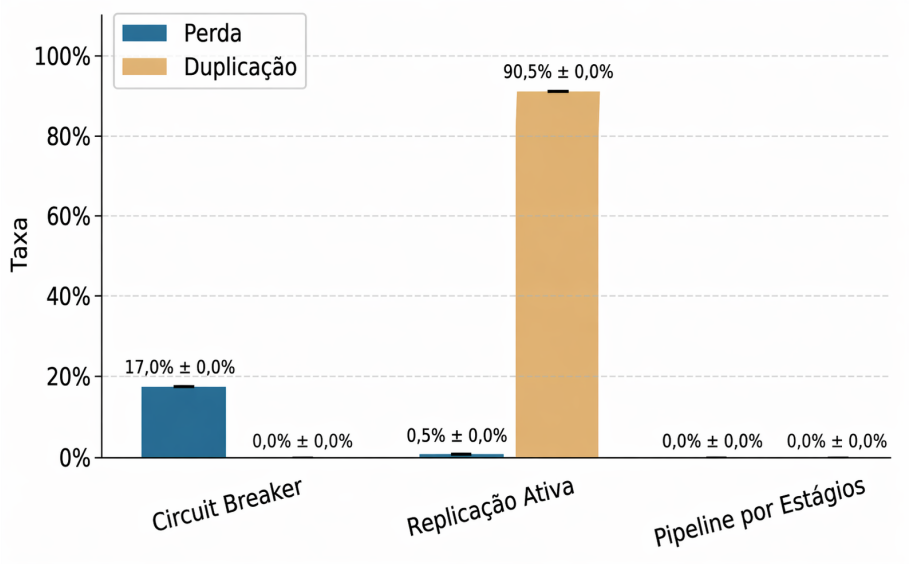

Figura 5. Taxas de perda e duplicação de mensagens no cenário de *flapping*.

Na Figura 6, o *Circuit Breaker* apresenta alta vazão e baixa latência, pois evita bloqueios prolongados ao falhar rapidamente quando o circuito abre. A Replicação Ativa registra vazão menor e latência mais elevada, consequência do overhead de múltiplos envios e coordenação. O Pipeline por Estágios mantém boa vazão, porém com latência p95 significativamente maior, efeito do acúmulo de mensagens nas filas e do reenvio após recuperação do consumidor. O gráfico ilustra o compromisso entre desempenho imediato (*Circuit Breaker*), disponibilidade via redundância (Replicação Ativa) e confiabilidade com atraso (Pipeline).

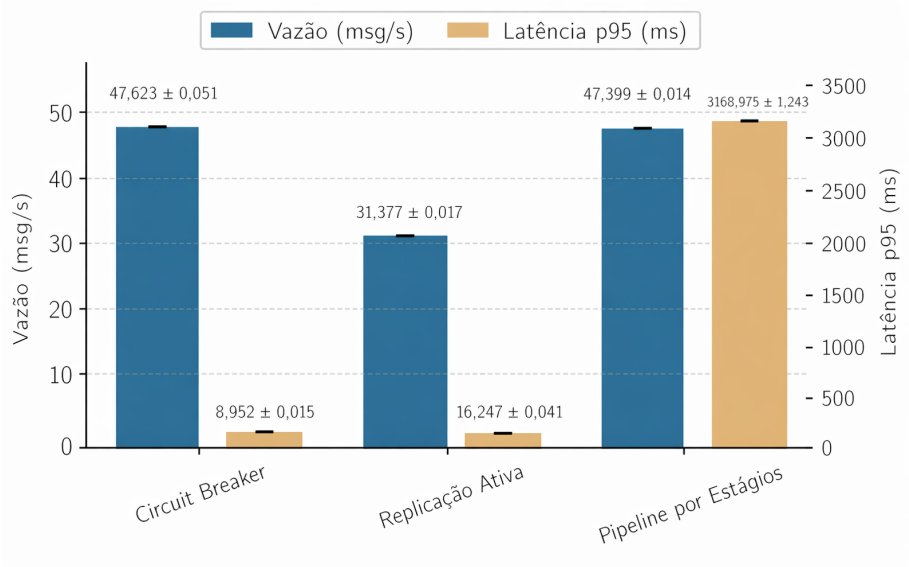

Figura 6. Vazão média e latência p95 no cenário de *flapping*.

Na Figura 7, referente ao cenário de *outage*, a memória novamente permanece estável entre as três arquiteturas, coerente com a carga controlada do experimento. O uso de CPU mostra diferenças discretas: *Circuit Breaker* e Replicação Ativa realizam

tentativas e verificações até a recuperação do serviço, enquanto o Pipeline por Estágios tende a consumir menos CPU ao priorizar o enfileiramento durante a indisponibilidade. O resultado sugere que, em quedas totais, a diferença principal não é o custo de CPU, mas a estratégia de preservação e recuperação das mensagens.

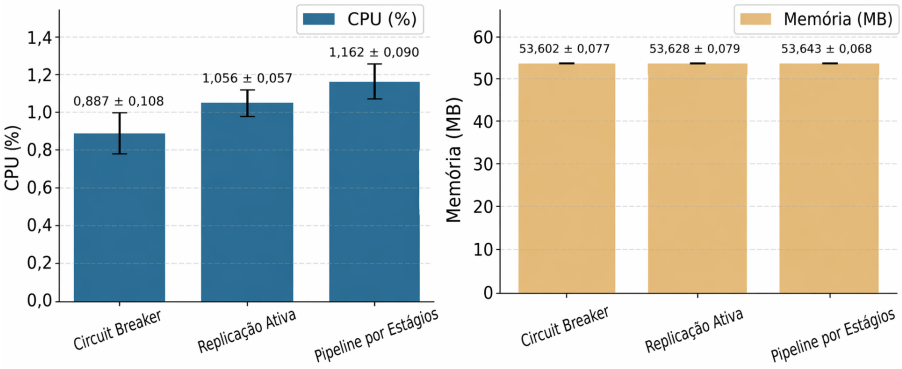

Figura 7. Consumo de CPU e memória no cenário de *outage* (indisponibilidade total do consumidor).

A Figura 8 mostra perda e duplicação no cenário de *outage*. O *Circuit Breaker* apresenta a maior perda, pois descarta mensagens enquanto permanece em estado *OPEN*, priorizando estabilidade. A Replicação Ativa reduz parcialmente a perda ao manter múltiplas tentativas, mas produz elevada duplicação como efeito colateral. O Pipeline por Estágios não apresenta perdas nem duplicações, evidenciando sua capacidade de acumular mensagens durante a interrupção e reenviá-las após o retorno do *backend*. Assim, o gráfico destaca o Pipeline como a abordagem mais eficaz para indisponibilidades prolongadas.

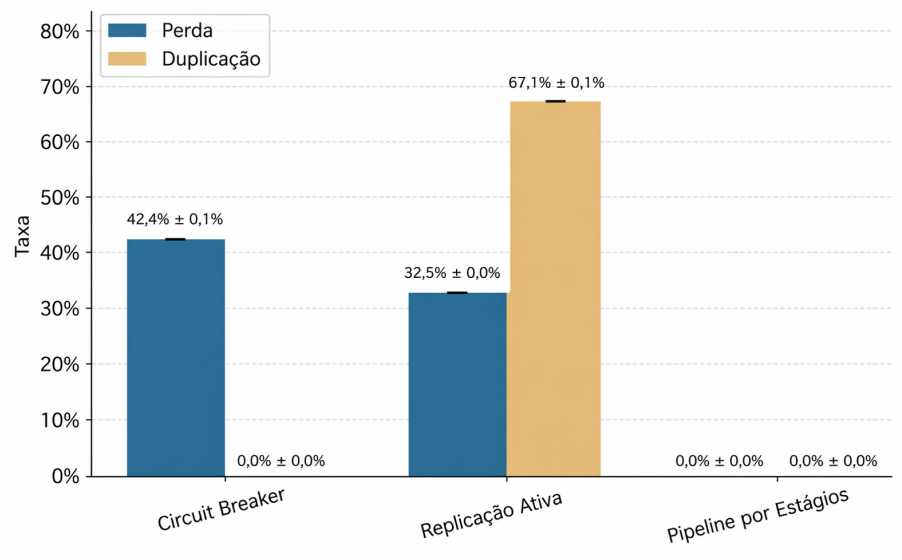

Figura 8. Taxas de perda e duplicação de mensagens no cenário de *outage*.

Na Figura 9, observa-se que as três arquiteturas apresentam latência p95 próxima, indicando que, após a recuperação, o processamento das mensagens efetivamente entregues tende a convergir. O *Circuit Breaker* apresenta maior vazão, pois interrompe

tentativas durante a queda e retoma a operação com a recuperação do consumidor. A Replicação Ativa registra a menor vazão, pois parte do esforço de envio durante a indisponibilidade não produz benefício e adiciona overhead quando o serviço retorna. O Pipeline por Estágios alcança a maior vazão, pois continua aceitando e acumulando mensagens durante a interrupção e as processa quando o *backend* volta, caracterizando resiliência temporal.

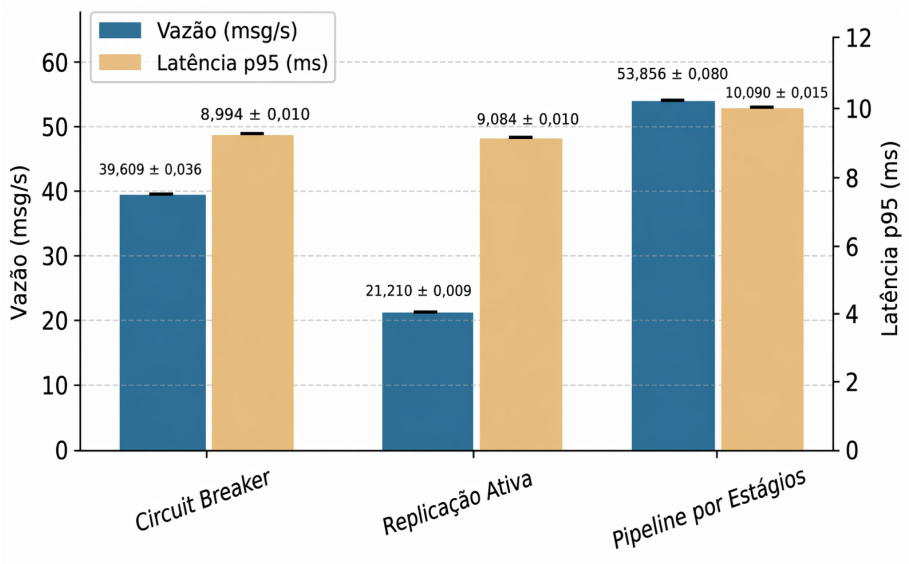

Figura 9. Vazão média e latência p95 no cenário de *outage*.

No cenário de consumidor extremamente lento, ilustrada na Figura 10, o comportamento indica contenção e bloqueio do fluxo. O *Circuit Breaker* ainda consome alguma CPU por monitorar o estado e executar verificações/timeout, enquanto Replicação Ativa e Pipeline por Estágios praticamente não utilizam CPU adicional por permanecerem bloqueados aguardando resposta do consumidor. O consumo de memória permanece semelhante entre todos, reforçando que a carga e o volume de dados do teste local não foram suficientes para provocar saturação, apesar de o Pipeline ser teoricamente mais sensível a acúmulos.

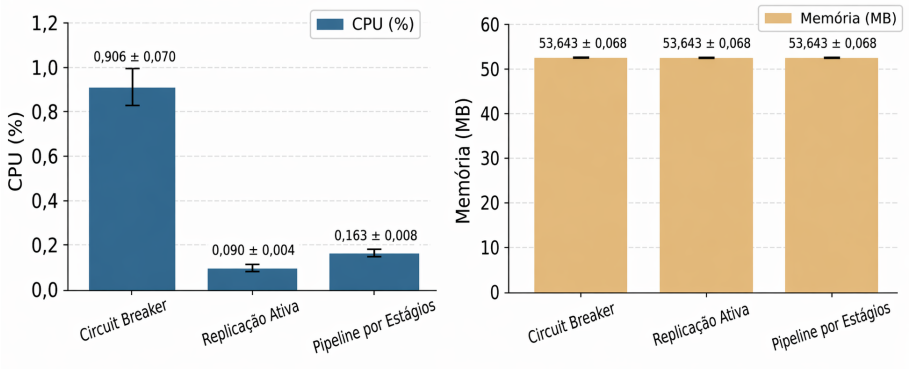

Figura 10. Consumo de CPU e memória no cenário de *slow consumer*.

De acordo com a Figura 11, a Replicação Ativa e o Pipeline por Estágios apresentam perda total no cenário de *slow consumer*, pois a lentidão impede a conclusão

de entregas, sem gerar duplicações relevantes. O *Circuit Breaker* apresenta perda parcial: ao detectar degradação severa, abre o circuito e passa a rejeitar rapidamente parte das requisições, evitando bloqueio completo. O resultado indica que, sob degradação de desempenho (e não queda), a capacidade de *fail-fast* se torna determinante para manter algum nível de serviço.

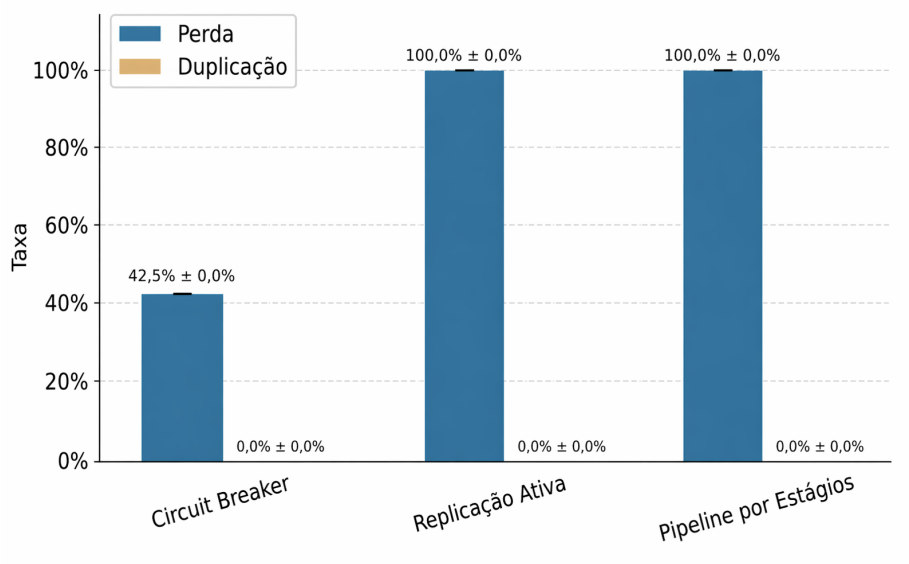

Figura 11. Taxas de perda e duplicação de mensagens no cenário de *slow consumer*.

Por fim, observa-se na Figura 12 que somente o *Circuit Breaker* apresenta valores efetivos de vazão e latência, pois continua respondendo rapidamente ao entrar no estado *OPEN*, evitando que o *middleware* fique preso ao consumidor lento. Replicação Ativa e Pipeline por Estágios mostram vazão nula e latência inexistente porque dependem da conclusão do processamento para progredir. O gráfico confirma que, em lentidão severa, o *Circuit Breaker* é a única abordagem operacionalmente viável no desenho avaliado, enquanto as demais colapsam por contenção de recursos no *downstream*.

6. Discussão

Os resultados evidenciam como cada arquitetura de *middleware* MQTT responde a padrões distintos de falha no consumidor, reforçando que a escolha do padrão impacta simultaneamente confiabilidade, desempenho e custo operacional. A comparação entre *Circuit Breaker*, Replicação Ativa e Pipeline por Estágios mostra vantagens e limitações dependentes do tipo de degradação. Esses achados são coerentes com a literatura, que destaca que mecanismos de contenção, redundância e desacoplamento alteram latência, estabilidade e uso de recursos .

No cenário de *flapping*, o *Circuit Breaker* atuou de forma conservadora, abrindo o circuito diante de erros sucessivos e evitando chamadas bloqueantes. Isso reduziu latência e consumo de CPU, ao custo de perda moderada de mensagens, compatível com a função do padrão de evitar cascatas e estabilizar o sistema . A Replicação Ativa, por sua vez, minimizou perdas ao explorar redundância paralela, mas gerou duplicação elevada e maior consumo de CPU, evidenciando o *trade-off* entre disponibilidade e eficiência. O Pipeline

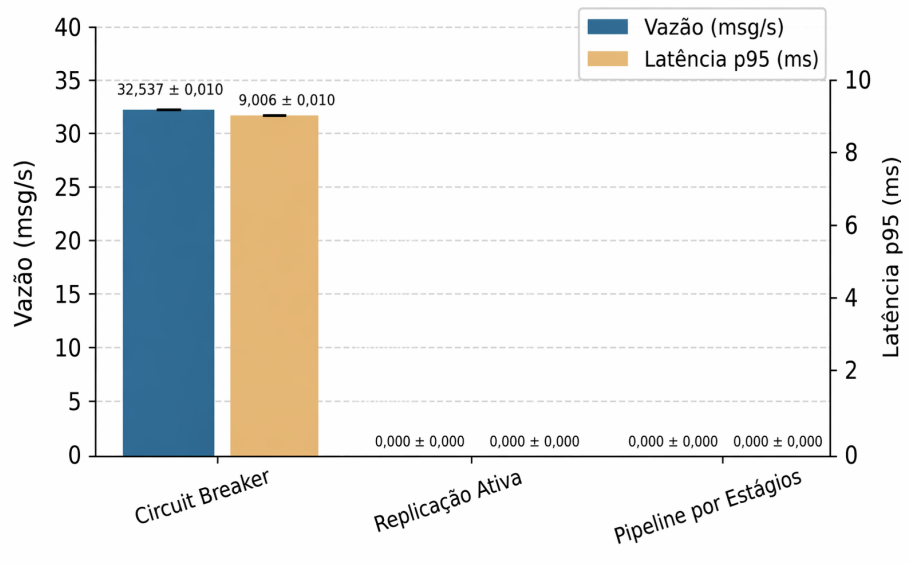

Figura 12. Vazão média e latência p95 no cenário de *slow consumer*.

por Estágios absorveu a instabilidade via enfileiramento e *retries*, preservando integridade (sem perdas e duplicações) porém elevando a latência p95 devido ao acúmulo temporário de mensagens, resultado esperado em estratégias de desacoplamento temporal.

No cenário de *outage*, as diferenças estruturais se tornaram mais pronunciadas. O *Circuit Breaker* descartou mensagens enquanto o circuito permaneceu *OPEN*, resultando na maior perda por não manter buffer para recuperação posterior. A Replicação Ativa reduziu parcialmente a perda, mas sem compensar a ausência do *backend*, além de introduzir duplicações e overhead. Já o Pipeline por Estágios destacou-se ao manter perda e duplicação nulas, enfileirando mensagens durante a indisponibilidade e processando-as após a recuperação. Esse comportamento se alinha ao argumento de que filas e desacoplamento são adequados para interrupções prolongadas..

No cenário de *slow consumer*, a falha é de temporização: o serviço existe, mas se torna incapaz de consumir em tempo hábil. Nessa condição, Replicação Ativa e Pipeline por Estágios colapsaram, pois dependem da conclusão do processamento no consumidor para liberar fluxo, ficando vulneráveis a contenção e *timeouts*. O *Circuit Breaker* manteve algum nível de operação ao falhar rapidamente quando detectou degradação severa, evitando bloqueio prolongado. O resultado reforça que, em degradação de desempenho, mecanismos de controle adaptativo tendem a ser mais efetivos do que redundância ou buffering sem controle de fluxo fino, aspecto também apontado em discussões sobre pipelines baseados em MQTT em cenários de carga e consumidores lentos .

A leitura integrada dos cenários confirma que não há estratégia universalmente superior. A Replicação Ativa é adequada quando o objetivo é maximizar entrega sob instabilidade intermitente, aceitando maior custo de CPU e duplicação, coerente com limitações de coordenação e overhead em soluções replicadas . O Pipeline por Estágios é o mais apropriado para quedas totais, por oferecer resiliência temporal e recuperação sem perda, exigindo atenção a controle de fluxo e armazenamento em escalas maiores . O *Circuit Breaker* é indicado quando a prioridade é estabilidade sob degradação severa,

reduzindo bloqueios e limitando propagação de falhas. Assim, em arquiteturas IoT *edge-cloud*, a escolha do *middleware* deve considerar o perfil de falha esperado, os requisitos de QoS, o custo computacional admissível e a sensibilidade a latência. A contribuição deste estudo está em oferecer uma base comparativa experimental sob estresse idêntico, auxiliando a seleção do padrão mais consistente com o cenário operacional.

7. Conclusão

Este trabalho avaliou três abordagens de resiliência para *middlewares* MQTT: *Circuit Breaker*, Replicação Ativa e Pipeline por Estágios. As estratégias foram comparadas sob os cenários de *flapping* (instabilidade intermitente), *outage* (indisponibilidade total) e *slow consumer* (degradação severa de desempenho), sendo analisadas em relação à confiabilidade da entrega (perda e duplicação), desempenho (vazão e latência) e custo operacional (consumo de CPU e memória). A metodologia, baseada em dez execuções independentes para cada cenário, permitiu a extração de médias e erros padrões que conferem validade estatística aos indicadores produzidos, garantindo que os resultados reflitam o comportamento inerente das arquiteturas e não oscilações ocasionais do ambiente de teste.

Os experimentos confirmam que não há solução universal, mas estratégias adequadas a perfis específicos de falha. A Replicação Ativa mostrou-se superior em instabilidades intermitentes ao maximizar a entrega e praticamente eliminar perdas, embora imponha o maior custo de CPU e uma elevada taxa de duplicação de mensagens. O Pipeline por Estágios revelou-se ideal para cenários de *outage*, garantindo perda e duplicação nulas via desacoplamento temporal e alcançando o maior vazão médio após a recuperação do serviço. No entanto, o Pipeline por Estágios apresentou o comportamento típico de sistemas assíncronos no cenário de *flapping*, onde a necessidade de preservar a integridade resultou em um acúmulo sistemático em filas, elevando significativamente a latência.

O *Circuit Breaker* destacou-se como a única abordagem operacionalmente viável diante da degradação severa do consumidor (*slow consumer*), preservando a estabilidade do sistema ao evitar bloqueios prolongados por meio de seu mecanismo de *fail-fast*, enquanto as demais arquiteturas colapsaram por contenção no *downstream*. A análise evidencia que a escolha arquitetural deve equilibrar o perfil de falha esperado, os recursos computacionais disponíveis e os requisitos de QoS da aplicação.

Como perspectivas futuras, propõe-se validar os padrões avaliados em ambientes distribuídos reais com múltiplos *brokers* e cargas de tráfego mais intensas, incluindo o teste de políticas de *backoff* exponencial. Investigar arquiteturas híbridas, como o *Circuit Breaker* operando em conjunto com pipelines por estágios e mecanismos de *bulkhead*, pode produzir soluções mais equilibradas entre desempenho e confiabilidade, consolidando o desenvolvimento de *middlewares* MQTT robustos e resilientes.

Agradecimentos

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001 e da Fundação Araucária, CP 23/2024 - Programa Institucional de Pesquisa Básica e Aplicada, projeto PBA2025201000012.

Referências

Banks, A. e Gupta, R. (2014). *MQTT Version 3.1.1*. OASIS Standard.

Bass, L., Clements, P., e Kazman, R. (2012). *Software Architecture in Practice*. Addison-Wesley, 3rd edition.

Chai, Y. et al. (2025). Dua-mqtt: Multi-agent mqtt architecture for industrial iot. *IEEE Transactions on Industrial Informatics*. Early Access.

Detti, A., Blefari-Melazzi, N., e Salsano, S. (2020). Analysis of MQTT broker clustering for iot systems. *IEEE Internet of Things Journal*, 7(10):9154–9167.

Dumitras, T. e Narasimhan, P. (2005). Fault-tolerant middleware and the magical 1%. In *International Middleware Conference*, pages 1–12.

Falahah, F. et al. (2021). Fault tolerance strategies in iot systems. *International Journal of Internet of Things*, 10(2):45–55.

Hmissi, H. e Ouni, A. (2025). Td-mqtt: Transparent and distributed mqtt architecture. *IEEE Internet of Things Journal*. Early Access.

ISO (2016). ISO/IEC 20922:2016 information technology — Message Queuing Telemetry Transport (MQTT).

Light, R. (2017). Mosquitto: Server and client implementation of the MQTT protocol. *Journal of Open Source Software*, 2(13):265.

Longo, A. et al. (2020). MQTT-ST: A fault-tolerant mqtt architecture. *Future Internet*, 12(6):1–15.

Mirampalli, S. et al. (2023). Fog computing architectures for iot systems. *IEEE Access*, 11:12345–12360.

Montesi, F. e Weber, J. (2018). Circuit breakers in microservice architectures. In *International Conference on Service-Oriented Computing Workshops*, pages 1–8.

Naim, M. et al. (2022). Edge–cloud collaboration for fault-tolerant iot systems. *Future Generation Computer Systems*, 128:10–20.

Nygard, M. T. (2007). *Release It!: Design and Deploy Production-Ready Software*. Pragmatic Bookshelf.

OASIS (2019). MQTT version 5.0 specification. https://mqtt.org. Accessed: 2026.

Ongaro, D. e Ousterhout, J. (2014). In search of an understandable consensus algorithm (raft). *USENIX Annual Technical Conference*, pages 305–319.

Pastorio, A. et al. (2020). Performance evaluation of mqtt brokers in iot systems. *IEEE Internet of Things Journal*, 7(5):4535–4546.

Poojara, S. et al. (2021). Performance evaluation of mqtt pipelines in fog computing environments. *Sensors*, 21(12):1–15.

Ren, J., Guo, H., Xu, C., e Zhang, Y. (2019). Serving at the edge: A scalable iot architecture based on edge computing. *IEEE Network*, 33(5):94–101.

Szentiványi, D. (2005). *Performance Studies of Fault-Tolerant Middleware*. PhD thesis, Budapest University of Technology and Economics.

Zhao, W., Melliar-Smith, P., e Moser, L. (2010). Fault tolerance middleware for cloud computing. In *IEEE International Conference on Cloud Computing*, pages 1–8.