UNIVERSIDADE DE CAXIAS DO SUL
ÁREA DO CONHECIMENTO DE CIÊNCIAS EXATAS E ENGENHARIAS

RICARDO MENEGUZZI BREGALDA

AVALIAÇÃO COMPARATIVA DE UMA ARQUITETURA DE LOGS BASEADA EM BLOCKCHAIN PERMISSIONADA E BANCO DE DADOS RELACIONAL

BENTO GONÇALVES

2025

RICARDO MENEGUZZI BREGALDA

AVALIAÇÃO COMPARATIVA DE UMA ARQUITETURA DE LOGS BASEADA EM BLOCKCHAIN PERMISSIONADA E BANCO DE DADOS RELACIONAL

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Ciência da Computação na Área do Conhecimento de Ciências Exatas e Engenharias da Universidade de Caxias do Sul.

Orientador: Prof. Dr. Helena Graziottin Ribeiro

BENTO GONÇALVES

2025

RICARDO MENEGUZZI BREGALDA

AVALIAÇÃO COMPARATIVA DE UMA ARQUITETURA DE LOGS BASEADA EM BLOCKCHAIN PERMISSIONADA E BANCO DE DADOS RELACIONAL

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Ciência da Computação na Área do Conhecimento de Ciências Exatas e Engenharias da Universidade de Caxias do Sul.

Aprovado em 03/12/2025

BANCA EXAMINADORA

Prof. Dr. Helena Graziottin Ribeiro
Universidade de Caxias do Sul - UCS

Prof. Dr. Leonardo Pellizzoni
Universidade de Caxias do Sul - UCS

Prof. Dr. Daniel Notari
Universidade de Caxias do Sul - UCS

RESUMO

Este trabalho investiga a viabilidade do uso de *blockchain* privada para o armazenamento de logs em ambientes corporativos, comparando-a com soluções tradicionais baseadas em bancos de dados. Problemas como vulnerabilidade a alterações e pontos únicos de falha em sistemas de logs convencionais motivam a busca por alternativas mais seguras. A metodologia adotada envolveu a implementação e avaliação de duas arquiteturas em contêineres Docker: uma híbrida, utilizando Hyperledger Fabric para ancoragem de *hashes* e MongoDB para logs *off-chain*, e outra tradicional com PostgreSQL primário-*standby*. Foram mensuradas métricas de desempenho (latência e *throughput*), custo operacional, capacidade de auditoria e resposta a falhas, simulando logs de sistemas de acesso e identidade (IAM). Os resultados apresentam uma análise comparativa detalhada entre as abordagens, evidenciando os *trade-offs* de desempenho e custo. Visando a reprodutibilidade científica e a contribuição para a comunidade, todo o código-fonte, incluindo a API, scripts de orquestração e ferramentas de análise, foi disponibilizado publicamente em repositório aberto. O estudo fornece, assim, uma base empírica sólida para a decisão de adoção de tecnologias em contextos críticos que requerem imutabilidade e auditabilidade de dados.

Palavras-chave: *Blockchain*, Logs, Hyperledger Fabric, PostgreSQL, Integridade de Dados.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE ABREVIATURAS E SIGLAS

API *Application Programming Interface* (Interface de Programação de Aplicações)

CSV *Comma-Separated Values* (Valores Separados por Vírgula)

HDFS *Hadoop Distributed File System*

HIPAA *Health Insurance Portability and Accountability Act*

I/O Entrada e Saída

LGPD Lei Geral de Proteção de Dados

MTTR *Mean Time To Repair* (Tempo Médio de Reparação)

NIST *National Institute of Standards and Technology*

PCI DSS *Payment Card Industry Data Security Standard*

P2P *Peer-to-Peer*

PoS *Proof of Stake*

PoW *Proof of Work*

SOX *Sarbanes–Oxley Act*

MART *Merkle — Merkle Adaptive Radix Tree*

PBFT *Practical Byzantine Fault Tolerance* (Protocolo de Tolerância a Falhas Bizantinas)

PoW *Proof of Work*

PoS *Proof of Stake*

BLSQ *Blockchain-based Log Storage and Querying*

CA *Certificate Authority*

JSONB *JSON Binary*

MBT *Merkle B+ Tree*

VM *Virtual Machine*

DApps *Decentralized Applications*

IAM *Identity and Access Management*

RBAC *Role-Based Access Control*

TLS *Transport Layer Security*

CAP *Consistency, Availability and Partition Tolerance*

ACID *Atomicity, Consistency, Isolation, Durability*

WAL *Write-Ahead Log*

ARIES *Algorithm for Recovery and Isolation Exploiting Semantics*

SLA Acordo de Nível de Serviço (*Service Level Agreement*)

gRPC *Google Remote Procedure Call*)

WAN *Wide Area Network* (Rede de Longa Distância)

SUMÁRIO

1 INTRODUÇÃO

Em ambientes computacionais modernos, a robustez e a confiabilidade dos sistemas são cruciais, mas a ocorrência de falhas é inerente. Falhas podem variar desde interrupções de hardware e erros de software até incidentes de segurança, podendo ocorrer com diferentes frequências e impactos. Para lidar com esses eventos, estratégias como a tolerância a falhas e a recuperação de falhas são essenciais. Nesse contexto, os registros de eventos, universalmente conhecidos como logs, são fundamentais para monitorar o comportamento de sistemas, diagnosticar falhas, auditar transações e garantir a conformidade com regulamentos de segurança e privacidade (Joint Task Force, 2020).

Tradicionalmente, logs são armazenados em bancos de dados relacionais ou arquivos de texto, soluções que oferecem alto desempenho e facilidade de consulta (Joint Task Force, 2020). No entanto, em cenários de grande escala e múltiplos participantes, essas abordagens podem se tornar vulneráveis a alterações não autorizadas, falhas de integridade ou pontos únicos de falha, representando um risco significativo para a confiança e a rastreabilidade dos dados (ANDERSON, 2020). A centralização da confiança nesses sistemas os torna alvos potenciais para manipulação de registros, o que pode comprometer a validade de auditorias e investigações de segurança (ANDERSON, 2020).

Um desafio fundamental nos sistemas tradicionais é garantir o não-repúdio, ou seja, a capacidade de provar que um determinado evento ocorreu e que seu registro não foi alterado posteriormente. Em um banco de dados convencional, mesmo com trilhas de auditoria, um administrador com privilégios elevados pode, teoricamente, modificar tanto os dados quanto os próprios registros de auditoria para ocultar seus rastros. A tecnologia *blockchain* endereça essa lacuna ao fornecer uma prova de integridade descentralizada e cronológica. Uma vez que o *hash* de um log é registrado em um bloco, ele não pode ser alterado sem invalidar toda a cadeia subsequente, um evento que seria imediatamente detectável por todos os participantes da rede. Essa característica intrínseca de imutabilidade e transparência distribuída cria uma trilha de auditoria na qual a origem e a integridade de cada registro podem ser verificadas de forma independente, garantindo que nenhuma parte possa negar ou repudiar uma ação registrada (KHAN *et al.*, 2020).

A tecnologia *blockchain* emerge como uma alternativa promissora para mitigar essas limitações, ao proporcionar um registro distribuído, imutável e auditável por meio de consenso criptográfico entre nós da rede (ZHENG *et al.*, 2017). Ao invés de confiar em uma única autoridade central, a *blockchain* distribui o livro-razão entre os participantes, de forma que a integridade dos dados é mantida pela rede como um todo (CASEY; VIGNA, 2018). Em particular, *blockchains* permissionadas combinam a segurança inerente ao modelo de cadeia de blocos com mecanismos de controle de acesso e escalabilidade adaptados a ambientes corporativos

(ANDROULAKI *et al.*, 2018). Embora bancos de dados relacionais priorizem rapidez de escrita e consultas estruturadas, *blockchains* privadas introduzem garantias adicionais de integridade e não-repúdio para logs (CHRISTIDIS; DEVETSIKIOTIS, 2016). No entanto, a comparação prática detalhada entre essas abordagens, focando em desempenho, custo e resiliência a falhas, ainda carece de validação experimental rigorosa na literatura.

1.1 OBJETIVOS

Este trabalho tem como objetivo principal verificar se uma *blockchain* privada é uma alternativa viável para armazenar registros de eventos (logs), em comparação com um banco de dados tradicional. Para detalhar essa investigação, foram definidos os seguintes objetivos específicos:

- **Definir critérios para avaliação de desempenho no uso de logs:** Estabelecer as métricas e os parâmetros que serão utilizados para mensurar a eficiência da gravação e leitura de logs em ambos os sistemas.
- **Implementar uma estrutura de log em *blockchain* e realizar testes de desempenho:** Desenvolver a arquitetura híbrida de logs utilizando Hyperledger Fabric e MongoDB, executar testes de ingestão e consulta de dados e avaliar o seu desempenho operacional.
- **Comparar o desempenho no uso de logs em banco de dados relacional e em *blockchain*:** Analisar e confrontar as métricas de desempenho obtidas nas duas abordagens (PostgreSQL e a solução híbrida com *blockchain*), destacando as vantagens e desvantagens de cada uma.

1.2 ESTRUTURA DO TRABALHO

O presente trabalho está organizado em sete capítulos, distribuídos da seguinte forma:

- **Capítulo 1 – Introdução:** apresenta o contexto do problema, os objetivos do trabalho, a metodologia adotada e a estrutura geral do documento.
- **Capítulo 2 – Logs:** aborda os conceitos fundamentais de registros de eventos (logs), suas aplicações em sistemas computacionais, técnicas de armazenamento e processamento, e os desafios relacionados à integridade e auditoria de logs em ambientes distribuídos.
- **Capítulo 3 – Blockchain:** apresenta os fundamentos da tecnologia *blockchain*, incluindo conceitos de consenso distribuído, criptografia, imutabilidade e tipos de redes (públicas, privadas e permissionadas), com ênfase em plataformas corporativas como o Hyperledger Fabric.

- **Capítulo 4 – Proposta de Desenvolvimento:** descreve a arquitetura híbrida proposta para gerenciamento de logs, detalhando os componentes do sistema, o fluxo de dados entre *blockchain* e banco de dados tradicional, e as decisões de projeto adotadas para garantir desempenho e integridade.
- **Capítulo 5 – Implementação e Metodologia Experimental:** detalha a implementação dos protótipos (arquitetura híbrida com Hyperledger Fabric e MongoDB, e arquitetura tradicional com PostgreSQL), os cenários de teste definidos, as métricas coletadas e os procedimentos experimentais utilizados para avaliação comparativa.
- **Capítulo 6 – Resultados e Análise:** apresenta os resultados obtidos nos experimentos de desempenho, tolerância a falhas e escalabilidade, realizando uma análise comparativa entre as abordagens tradicional e híbrida, e discutindo as implicações práticas dos resultados observados.
- **Capítulo 7 – Conclusão:** sintetiza as principais contribuições do trabalho, responde às questões de pesquisa levantadas, apresenta as limitações encontradas e propõe direções para trabalhos futuros na área de gerenciamento seguro de logs com *blockchain*.

1.3 METODOLOGIA

A metodologia compreende as seguintes etapas:

1. **Revisão bibliográfica:** levantamento de artigos, livros e documentação técnica sobre as práticas de armazenamento de logs em bancos de dados e o uso de *blockchains* permissionadas, a fim de fundamentar teoricamente nosso estudo.
2. **Projeto da arquitetura de logs com *blockchain*:** elaboração de um modelo de sistema em que os eventos gerados pelas aplicações são coletados, agrupados em lotes e, após serem gravados em um repositório tradicional, têm seu resumo criptográfico registrado em uma *blockchain* privada.
3. **Desenvolvimento experimental:** implementação de dois protótipos com essa arquitetura, um usando Hyperledger Fabric para a parte *on-chain* e MongoDB para o *off-chain*, e outro totalmente baseado em PostgreSQL, para ingestão e consulta de logs em condições controladas.
4. **Medições quantitativas:** execução de testes de desempenho em ambos os protótipos, registrando o tempo de gravação de cada lote de logs, o tempo de consulta e o consumo de recursos (CPU, memória e disco).
5. **Análise comparativa:** avaliação da latência, capacidade de processamento, custos operacionais e resistência a falhas de cada abordagem, destacando as vantagens e os desafios de usar *blockchain* para garantir a integridade dos registros.

2 LOGS

Um log é definido como um registro cronológico de eventos e atividades que ocorrem em sistemas, aplicações e infraestruturas de rede.

Cada entrada de log tipicamente contém um identificador da origem do registro (processo, serviço ou usuário), um nível de severidade e uma mensagem descritiva do evento. Segundo o *National Institute of Standards and Technology* (NIST), logs são "um registro dos eventos que ocorrem dentro dos sistemas e redes de uma organização", ressaltando seu papel fundamental na auditoria e rastreamento de comportamentos (Joint Task Force, 2020).

Os registros de log documentam eventos do sistema, por exemplo, alterações nos valores de dados ao longo do tempo e as operações realizadas por cada usuário, com indicação de data e hora. Essa rastreabilidade suporta a detecção de falhas, a investigação de incidentes de segurança e a otimização de desempenho, servindo de base tanto para análises reativas quanto para ações proativas.

2.1 TIPOS DE LOGS

Os logs podem ser classificados conforme o nível e o âmbito de registro. A seguir, detalham-se os principais tipos:

- **Logs do Sistema:** registram eventos do sistema operacional, como inicializações, desligamentos, falhas de hardware, troca de módulos de kernel e alterações de configuração. São gerados automaticamente pelo kernel ou por serviços de sistema. Esses registros ajudam administradores a diagnosticar instabilidades de hardware e interdependências de serviços (STALLINGS; BROWN, 2018).

- **Logs de Aplicação:** documentam o comportamento de softwares específicos, servidores web, gerenciadores de banco de dados ou microserviços. Incluem mensagens de depuração, avisos e erros, auxiliando desenvolvedores na identificação de bugs, no rastreamento de transações e na análise de desempenho sob carga (NEMETH *et al.*, 2017).

- **Logs de Segurança:** focam em eventos críticos de controle de acesso, como autenticações (bem-sucedidas ou não), alterações de credenciais e operações privilegiadas. Esses logs são essenciais para cumprir normas de segurança (por exemplo, *Payment Card Industry Data Security Standard* (PCI DSS) e *Health Insurance Portability and Accountability Act* (HIPAA)) e para investigações forenses em caso de ataques (Joint Task Force, 2020).

- **Logs de Erro:** registram exceções, falhas de processamento e mensagens de *stack trace* geradas por aplicações. Em ambientes de alta disponibilidade, esses registros reduzem o

Mean Time To Repair (Tempo Médio de Reparação) (MTTR), ao fornecer informações detalhadas sobre a causa raiz de falhas críticas (SILBERSCHATZ; GALVIN; GAGNE, 2018).

- **Logs de Acesso:** capturam solicitações de usuários a recursos, como requisições HTTP, consultas a *Application Programming Interface* (Interface de Programação de Aplicações) (API) e operações de leitura/escrita em banco de dados. Analisar esses logs permite descobrir padrões de uso, detectar picos de tráfego e prevenir comportamentos anômalos (HE *et al.*, 2021).

2.2 APLICAÇÕES DOS LOGS

As aplicações dos logs são amplas e variadas, incluindo desde a depuração de software até a garantia de conformidade regulatória. Conforme o NIST, esses registros fornecem uma trilha de auditoria imprescindível para monitorar atividades, detectar violações de políticas e investigar incidentes de segurança (Joint Task Force, 2020).

Nesse contexto, os logs são fundamentais para a análise de desempenho. Eles são fontes ricas para a identificação de consultas lentas, gargalos de recursos e padrões de uso que possam comprometer a eficiência operacional de sistemas e aplicações. A partir da análise detalhada de tempos de resposta, taxas de erro e consumo de recursos registrados nos logs, é possível diagnosticar e otimizar a performance (HE *et al.*, 2021).

Os registros também auxiliam no planejamento de capacidade. A análise de logs históricos permite a projeção de demanda futura com base em tendências de uso, como picos de tráfego e crescimento da base de usuários. Essa capacidade é essencial para o planejamento e a alocação adequada de recursos (como CPU, memória e armazenamento) na infraestrutura, evitando tanto o subdimensionamento (que causaria lentidão e falhas) quanto o superdimensionamento (que resultaria em custos desnecessários) (NEMETH *et al.*, 2017).

Além disso, os logs são componentes indispensáveis para a conformidade regulatória. Eles dão suporte a auditorias e relatórios de cumprimento de normas e legislações específicas (por exemplo, *Sarbanes–Oxley Act* (SOX), HIPAA, a brasileira Lei Geral de Proteção de Dados (LGPD) e outras regulamentações de proteção de dados). A gestão adequada dos logs garante que os registros permaneam íntegros, disponíveis e auditáveis por períodos definidos, o que é fundamental para demonstrar conformidade e evitar penalidades legais (Joint Task Force, 2020)(Brasil, 2018).

2.3 TECNOLOGIAS USUAIS DE ARMAZENAMENTO DE LOGS

Diversas formas de armazenamento vem sendo utilizadas para gerenciar volumes crescentes de logs. Cada método apresenta compensações entre simplicidade, desempenho, escalabilidade e segurança.

2.3.1 Armazenamento em arquivos

O armazenamento em arquivos é o método mais fundamental para o registro de logs. Cada evento é simplesmente anexado como uma nova linha no final de um arquivo de texto (plain text) ou em um formato delimitado, como *Comma-Separated Values* (Valores Separados por Vírgula) (CSV). Esta abordagem é comum em aplicações monolíticas e sistemas operacionais, dada a sua aparente simplicidade de implementação, pois requer apenas operações básicas de escrita de arquivo.

Embora simples, esta solução apresenta severas limitações em termos de consulta e desempenho. A ausência de uma estrutura de indexação, como a encontrada em bancos de dados, significa que qualquer busca por conteúdo específico exige uma varredura completa do arquivo, linha por linha, geralmente utilizando ferramentas de linha de comando como *grep* ou *awk*. Conforme o volume de dados aumenta, o que ocorre rapidamente em sistemas ativos, o desempenho dessas consultas se degrada linearmente, tornando a análise de incidentes um processo lento e custoso em termos de Entrada e Saída (I/O) (Joint Task Force, 2020).

Além do desempenho, a gestão, segurança e recuperação são problemáticas. Os arquivos de log crescem indefinidamente, exigindo o uso de utilitários externos (como o *logrotate* em sistemas Linux) para gerenciar o arquivamento, compressão e exclusão de logs antigos (NEMETH *et al.*, 2017). O controle de acesso é rudimentar, limitando-se às permissões do sistema de arquivos, o que impossibilita um controle fino de permissões. Por fim, a recuperação de falhas é frágil; uma corrupção de disco pode levar à perda de dados, e a restauração depende inteiramente de backups periódicos, que podem não conter os eventos mais recentes ocorridos antes da falha (Joint Task Force, 2020).

2.3.2 Banco de Dados Relacional

Em muitas arquiteturas de gerenciamento de logs, os registros são armazenados em bancos de dados relacionais, onde cada evento corresponde a uma linha em uma tabela estruturada com colunas para carimbo de data e hora (*timestamp*), origem, nível de severidade e mensagem.

Conforme orienta o NIST SP 800-92, essa abordagem facilita a aplicação de índices em campos críticos, a execução de consultas e a geração de relatórios por meio de SQL, além de suportar mecanismos robustos de replicação e *failover* [1] (Joint Task Force, 2020). Para sustentar altas taxas de gravação sem comprometer a velocidade de consulta, recomenda-se particionar as tabelas por intervalos de tempo e criar índices compostos.

[1] *Failover* é o processo automático de redirecionar operações de um componente falho para um sistema de reserva (*backup*), de modo a garantir a continuidade do serviço sem intervenção manual.

2.3.3 Banco de Dados Não Relacional

Os sistemas de dados *NoSQL* surgiram como uma alternativa aos bancos relacionais, eliminando a exigência de um esquema rígido e homogêneo para o armazenamento de registros. Para o gerenciamento de logs, a categoria mais proeminente é a de *document stores* (bancos de dados orientados a documentos).

No MongoDB cada log é armazenado como um documento JSON (ou BSON, sua representação binária), conforme ilustrado na Figura 1. Esta abordagem é estrategicamente vantajosa para logs, que são dados semiestruturados: enquanto alguns campos são padronizados (como *timestamp*), outros (como metadados) podem variar drasticamente. A flexibilidade do esquema do MongoDB acomoda essa heterogeneidade nativamente, permitindo também a criação de réplicas automáticas para alta disponibilidade.

Embora o modelo de documento seja o foco deste trabalho, outras arquiteturas NoSQL incluem os *column-family stores* (como Cassandra), que escalam linearmente e são eficientes para dados de série temporal, e os *key-value stores* (como Redis), que oferecem latência de leitura/escrita extremamente baixa. A escolha entre esses sistemas depende das garantias de consistência e disponibilidade desejadas (teorema CAP) e dos padrões de consulta (DAVOUDIAN; CHEN; LIU, 2018).

Figura 1 – Exemplo de entrada de log armazenada em um *document store* no formato JSON


```
{
  "t": {
    "$date": "2020-05-01T15:16:17.180+00:00"
  },
  "s": "I",
  "c": "NETWORK",
  "id": 12345,
  "ctx": "listener",
  "svc": "R",
  "msg": "Listening on",
  "attr": {
    "address": "127.0.0.1"
  }
}
```


Fonte: https://www.mongodb.com/pt-br/docs/manual/reference/log-messages/

Quando volumes de logs atingem centenas de *terabytes*, torna-se inviável processá-los em um único servidor porque o desempenho fica comprometido. Nessa situação, empregam-se *frameworks* de *Big Data*, que distribuem tarefas entre vários nós (*clusters*) e armazenam os dados em um sistema de arquivos distribuído (como o *Hadoop Distributed File System* (HDFS)). O processamento em lote agrupa grandes quantidades de registros para análises programadas, enquanto o processamento em tempo real examina eventos à medida que ocorrem, permitindo a identificação imediata de problemas críticos (KLEPPMANN, 2017).

2.3.4 Serviços em nuvem

Serviços de nuvem, como AWS, Azure e Google Cloud, são como grandes "fazendas de computadores"que oferecem plataformas já prontas para gerenciar logs. A principal vantagem é que não é necessário comprar, configurar ou manter servidores próprios (MARINESCU, 2017)(ROUSE, 2020). A própria empresa de nuvem cuida de toda a infraestrutura e manutenção.

Por exemplo, esses serviços oferecem:

- **Coleta automática:** As plataformas de nuvem podem coletar logs de forma automática de todos os sistemas e aplicativos, sem a necessidade de configurações complexas por parte do usuário (ROUSE, 2020).
- **Armazenamento centralizado:** Todos os logs são reunidos em um único local seguro e gerenciado, facilitando a organização e o acesso (ROUSE, 2020).
- **Ferramentas de busca e análise:** Essas plataformas disponibilizam ferramentas e linguagens de consulta para buscar rapidamente informações nos logs, criar alertas sobre problemas críticos e gerar relatórios detalhados (MARINESCU, 2017).

O custo de usar esses serviços geralmente varia de acordo com a quantidade de dados que é ingerida (enviada para a nuvem), pelo período de retenção desses logs e pelo número de consultas realizadas sobre eles (MARINESCU, 2017)(ROUSE, 2020).

2.3.5 Armazenamento baseado em *blockchain*

O armazenamento de logs em *blockchain* utiliza a estrutura de blocos encadeados para garantir que os registros não possam ser alterados após a gravação. Em vez de inserir cada evento de log diretamente na cadeia de blocos (o que seria lento e custoso), calcula-se um resumo criptográfico (*hash*) de um conjunto de registros armazenados em um banco convencional ou em arquivos. Esse *hash* é então submetido à *blockchain*, criando uma prova imutável de que aqueles logs existiam em determinado estado e momento (CHRISTIDIS; DEVETSIKIOTIS, 2016).

Cada bloco contém o *hash* do bloco anterior, o que forma uma corrente (*chain*) que se torna impossível de modificar sem refazer todo o trabalho de validação dos blocos subsequentes. Esse encadeamento assegura a integridade dos logs: se alguém alterar um único bit em um registro *off-chain*, o *hash* recalculado já não coincidirá com o valor registrado na cadeia de blocos, evidenciando a adulteração (ANTONOPOULOS, 2017)(MURTHY, 2015).

Os nós da rede *blockchain* validam novos blocos por meio de um protocolo de consenso, o que garante que apenas transações legítimas, no caso, *hashes* de logs autorizados, sejam adicionadas à cadeia. Embora o consenso distribua a confiança e elimine pontos únicos de falha, o processo pode introduzir atrasos (latência) e custos (taxas de transação) que precisam ser planejados (ZHENG *et al.*, 2017).

Para automatizar esse fluxo, usam-se *smart contracts* (programas autoexecutáveis gravados na *blockchain*) que, em intervalos regulares, coletam os *hashes* de logs *off-chain* e os publicam em novos blocos (CHRISTIDIS; DEVETSIKIOTIS, 2016). Esse mecanismo torna desnecessária a intervenção manual para auditoria de integridade e permite configurar alertas automáticos sempre que houver inconsistências entre os registros originais e os valores *on-chain* (MURTHY, 2015).

Em resumo, o armazenamento baseado em *blockchain* oferece um nível extra de segurança e auditabilidade para logs, combinando a rapidez e a capacidade de consulta de bancos tradicionais com a imutabilidade e a descentralização da cadeia de blocos (CHRISTIDIS; DEVETSIKIOTIS, 2016).

2.4 TEOREMA CAP E CONSISTÊNCIA DE DADOS

O Teorema CAP postula que, em um sistema de dados distribuído sujeito a partições de rede (P), é necessário realizar um compromisso entre oferecer Consistência (C) ou Disponibilidade (A). Embora a formulação original sugerisse uma escolha binária de "dois entre três", interpretações contemporâneas esclarecem que, na ausência de falhas de rede, o sistema deve garantir tanto consistência quanto disponibilidade; contudo, durante uma partição, o sistema deve optar por interromper a operação para manter os dados coerentes (CP) ou continuar operando com risco de divergência nos dados (AP) (STEEN; TANENBAUM, 2023).

Para sistemas de logs, essa escolha é crucial, pois influencia diretamente a confiabilidade da trilha de auditoria e a capacidade de operação contínua em cenários de falha. A Tabela 1 apresenta como diferentes tecnologias de armazenamento priorizam essas propriedades (STEEN; TANENBAUM, 2023).

Tabela 1 – Classificação de Sistemas de Armazenamento de Logs segundo o Teorema CAP

Sistema	**Tipo CAP**	**Implicações para Logs**
MongoDB (configuração *default*)	CP	Prioriza consistência + particionamento, sacrificando disponibilidade temporária durante falhas de rede. Garante que todos os nós vejam os mesmos logs, essencial para auditoria.
Hyperledger Fabric com Raft	CP	Garante que todos os *peers* tenham o mesmo *ledger*. Em caso de partição, o sistema pode bloquear escritas até que o consenso seja restaurado.
PostgreSQL com replicação síncrona	CP	Falha do *standby* pode bloquear escritas no primário, priorizando consistência sobre disponibilidade. Ideal para logs críticos que não toleram divergências.
Cassandra (alternativa NoSQL)	AP	Prioriza disponibilidade + particionamento, aceitando inconsistências temporárias (*eventual consistency*). Nós isolados continuam aceitando escritas que serão sincronizadas posteriormente.

Fonte: Adaptado de (STEEN; TANENBAUM, 2023) e (DAVOUDIAN; CHEN; LIU, 2018).

2.5 MÉTRICAS PARA AVALIAR O USO DE LOGS

Para avaliar a eficácia e a capacidade de monitoramento de um sistema de logs, definiram-se diversas métricas que mensuram desde o volume de dados gerados até a qualidade e o tempo de resposta na detecção de incidentes:

- **Volume de Logs:** Mede-se a quantidade total de entradas registradas por unidade de tempo (por exemplo, entradas por segundo ou por hora). Essa métrica reflete a escala de eventos no sistema e orienta a capacidade de armazenamento e dimensionamento da infraestrutura de logs (NEMETH *et al.*, 2017).

- **Velocidade e Taxa de Eventos:** Avalia-se a taxa de geração de logs (*throughput*) e a velocidade de ingestão pelo sistema de coleta, importante para identificar picos de atividade e garantir que não ocorram perdas de dados em momentos de alta carga (STALLINGS; BROWN, 2018).

- **Latência de Coleta e Disponibilidade:** Mede-se o tempo decorrido desde o evento até sua persistência e indexação no repositório de logs, bem como a porcentagem de tempo em que o sistema de logs permanece disponível para consultas (Joint Task Force, 2020).

- **Integridade e Qualidade dos Dados:** Avalia-se a completude (ausência de entradas faltantes), consistência (formato uniforme de campos) e correção (mensagens sem erros de parse) das entradas de log. Métodos automatizados de validação, como *checksums* e esquemas de *JSON Schema*, são usados para calcular taxas de falha na ingestão (HE *et al.*, 2021).

- **Taxa de Erros e Alertas:** Mensura-se a proporção de entradas que representam erros (ERROR, WARN) e a frequência de alertas gerados por sistemas de monitoramento automatizado, auxiliando na avaliação da saúde operacional dos serviços (STALLINGS; BROWN, 2018).

- **Cobertura de Logs:** Refere-se à porcentagem de componentes de software ou infraestrutura cuja atividade é efetivamente registrada. Alta cobertura garante visibilidade completa do ambiente, necessária para auditoria e conformidade (Joint Task Force, 2020).

Em suma, as métricas apresentadas oferecem uma visão abrangente da qualidade, eficiência e confiabilidade de um sistema de logs. A aplicação consistente desses indicadores permite não apenas dimensionar corretamente a infraestrutura e prever custos operacionais, mas também detectar rapidamente falhas de desempenho e de segurança, garantindo conformidade regulatória e suporte eficaz a auditorias internas e externas.

2.6 DESAFIOS E LIMITAÇÕES – IMPORTÂNCIA DA RECUPERAÇÃO DE FALHAS

O armazenamento de logs, embora fundamental, enfrenta desafios significativos no que diz respeito à integridade, disponibilidade e resiliência a falhas. A ocorrência de falhas de hardware, corrupção de dados ou partições de rede é inerente a ambientes computacionais, e as soluções tradicionais estão sujeitas a vulnerabilidades que comprometem a confiabilidade dos registros (ANDERSON, 2020).

O armazenamento de logs diretamente em arquivos, por exemplo, pode resultar em perda permanente de dados se não houver sincronização adequada em disco ou se backups periódicos falharem. A integridade desses arquivos é facilmente comprometida por acessos não autorizados ou falhas de sistema, sem mecanismos de detecção de adulteração embutidos. Já os bancos de dados relacionais, embora ofereçam mecanismos robustos de consistência e replicação, a recuperação de falhas em cenários de desastre ou corrupção pode ser complexa e demandar um *failover* cuidadosamente planejado para evitar inconsistências ou perda de dados (Joint Task Force, 2020). A confiança centralizada no administrador do banco de dados e nos sistemas de backup também representa um ponto único de falha para a integridade dos logs (ANDERSON, 2020).

Por sua vez, as soluções de bancos de dados Não Relacionais (*NoSQL*) e *Big Data*, projetadas para escalabilidade e disponibilidade, dependem fortemente de estratégias de replicação e *sharding*. Contudo, em cenários de partição de rede, elas podem enfrentar dificuldades em manter a consistência imediata ou evitar inconsistências temporárias, conforme o teorema *Consistency, Availability and Partition Tolerance* (CAP) (DAVOUDIAN; CHEN; LIU, 2018). A capacidade de auditabilidade e a garantia de não-repúdio podem não ser propriedades intrínsecas a essas tecnologias, necessitando de camadas adicionais de segurança para serem alcançadas. A importância da recuperação de falhas para logs não pode ser subestimada; em sistemas críticos, a perda ou corrupção de logs pode inviabilizar a auditoria de segurança, a investigação forense de incidentes, o diagnóstico de falhas operacionais e a demonstração de conformidade regulatória. Um sistema de logs resiliente assegura que, mesmo após eventos adversos, a trilha de auditoria permaneça completa, íntegra e confiável.

Nesse contexto de desafios, a abordagem *blockchain* emerge como uma tecnologia promissora para mitigar muitos desses riscos. Sua arquitetura intrínseca de registro distribuído e encadeado confere uma imutabilidade e resistência à adulteração sem precedentes: ao distribuir a confiança entre vários nós e manter um histórico imutável de *hashes*, a rede pode verificar a integridade dos registros mesmo após ataques ou quedas de parte da infraestrutura (CHRISTIDIS; DEVETSIKIOTIS, 2016)(DRESCHER, 2017). Enquanto a maioria dos participantes da rede estiver ativa, é possível recuperar e verificar a integridade dos registros.

Além disso, a utilização de *smart contracts* pode ir além da simples gravação de *hashes*,

podendo ser programados para disparar alertas automáticos em caso de divergências entre os dados *off-chain* e seus *hashes* registrados *on-chain*, o que reduz significativamente o tempo de resposta a incidentes de integridade (MURTHY, 2015).

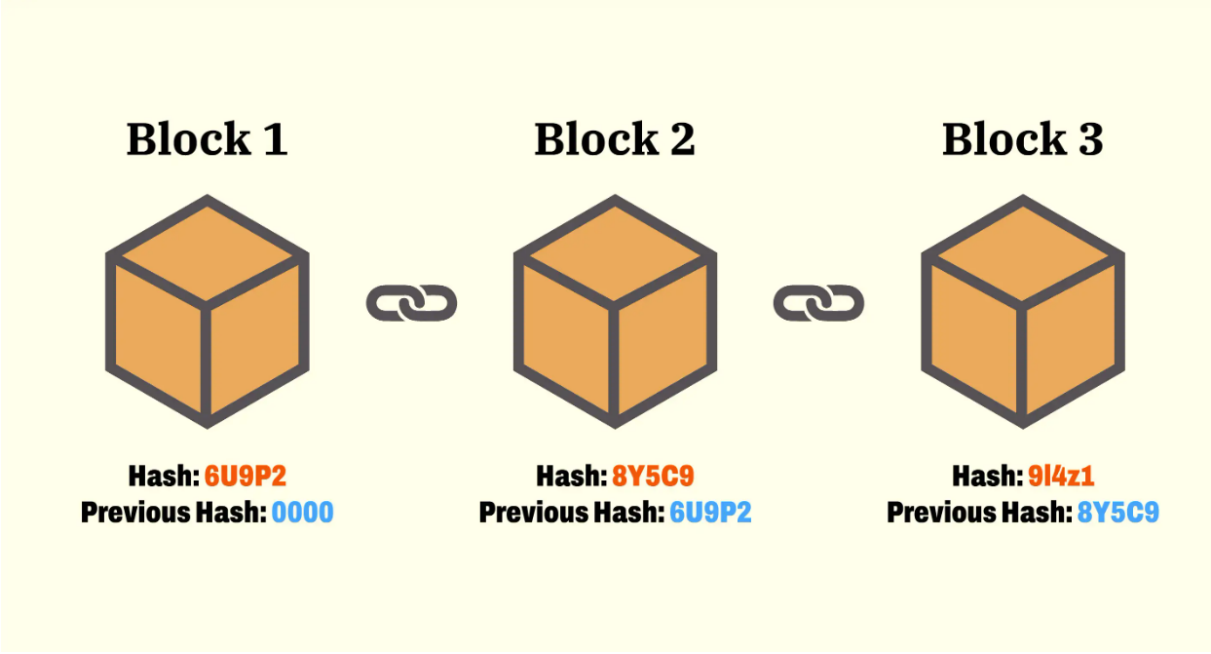
Contudo, é fundamental reconhecer que a integração da *blockchain* em um sistema de logs não está isenta de desafios próprios. Questões como a escalabilidade (capacidade de processar um alto volume de transações), latência (atraso na confirmação das transações) e o custo ainda representam barreiras para a adoção em massa, especialmente em redes públicas (DINH *et al.*, 2018). É crucial um planejamento cuidadoso para o armazenamento *off-chain* dos logs completos, garantindo que o volume de dados seja gerenciado de forma eficiente. Além disso, é necessário otimizar o volume de transações *on-chain* (publicação de *hashes*) para equilibrar os custos operacionais (mesmo em redes permissionadas, há custos de infraestrutura e processamento) e o desempenho, assegurando que a solução seja economicamente e tecnicamente viável (WANG *et al.*, 2020).

3 BLOCKCHAIN

Blockchain é uma tecnologia de registro distribuído que mantém um livro-razão (*ledger*) compartilhado e imutável entre participantes sem a necessidade de autoridade central (ZHENG *et al.*, 2017). Cada bloco agrupa um lote de transações, é carimbado com *timestamp* e referencia criptograficamente o bloco anterior, formando uma cadeia resistente a alterações. A validação de novos blocos é feita por protocolos de consenso (como o *Proof of Work* (PoW) e *Proof of Stake* (PoS)), garantindo integridade, ordem cronológica e resistência à censura (ZHENG *et al.*, 2017).

A Figura 2 ilustra de forma simplificada como cada bloco armazena seu próprio *hash* e o *hash* do bloco anterior, reforçando a imutabilidade da cadeia: qualquer modificação em um bloco anterior invalida todos os *hashes* subsequentes.

Figura 2 – Funcionamento básico de uma *blockchain*, cada bloco referencia o *hash* do bloco anterior, criando imutabilidade



Fonte: https://money.com/what-is-blockchain/

3.1 ELEMENTOS DE UMA *BLOCKCHAIN*

Os elementos fundamentais que compõem a arquitetura de uma *blockchain* são cruciais para compreender seu funcionamento e suas propriedades de segurança e descentralização. Estes são detalhados nas subseções a seguir:

3.1.1 Funções de *Hash* Criptográfico

No cerne da segurança e integridade de uma *blockchain* estão as funções de *hash* criptográfico. Um *hash* é uma sequência de caracteres de tamanho fixo, gerada a partir de um dado de entrada de qualquer tamanho. Funciona como uma "impressão digital"única para os dados: pequenas alterações na entrada resultam em um *hash* completamente diferente (ANTONOPOULOS, 2017).

Para que sejam consideradas criptograficamente seguras para uso em *blockchain*, essas funções devem exibir um conjunto de propriedades essenciais. Elas precisam ser determinísticas, garantindo que a mesma entrada sempre produzirá o mesmo *hash*, e rápidas de computar. Mais importante, devem ser unidirecionais, o que é definido pela resistência à pré-imagem, tornando computacionalmente inviável reverter um *hash* para encontrar a entrada original. Elas também devem garantir a unicidade dos resultados, o que é coberto pela resistência à segunda pré-imagem (difícil encontrar uma entrada diferente que produza o mesmo *hash* de uma entrada dada) e pela resistência à colisão (extremamente difícil encontrar duas entradas diferentes que produzam o mesmo *hash*).

Os principais algoritmos de *hash* utilizados por *blockchains* incluem SHA-256 (*Secure Hash Algorithm 256-bit*), amplamente usado no Bitcoin[1] e no Ethereum[2] (ANTONOPOULOS, 2017). Essas funções garantem a integridade dos blocos e das transações, sendo vitais para a imutabilidade da cadeia.

3.1.2 Bloco e *Hash Pointer*

A *blockchain* é fundamentalmente organizada como uma sequência de blocos de dados interligados, onde cada bloco atua como um contêiner para um conjunto de transações. Um bloco é tipicamente dividido em duas partes: o corpo, que contém a lista de transações (geralmente estruturadas em uma Árvore de Merkle), e o cabeçalho (*header*), que contém os metadados do bloco. É neste cabeçalho que reside o mecanismo central de encadeamento da *blockchain*.

O componente chave no cabeçalho é o *hash pointer* (ponteiro de *hash*). Diferente de um ponteiro tradicional que apenas aponta para um endereço de memória, um *hash pointer* é uma estrutura de dados que contém duas informações: o endereço do bloco anterior e o *hash* criptográfico do cabeçalho daquele bloco anterior (NARAYANAN *et al.*, 2016). Ao incluir o *hash* do bloco N-1 no cabeçalho do bloco N, cria-se uma ligação criptográfica inquebrável, formando uma estrutura de dados semelhante a uma lista ligada, mas com propriedades de segurança muito superiores.

Essa estrutura de encadeamento é a base da imutabilidade da *blockchain*. Qualquer ten-

[1] https://bitcoin.org
[2] https://ethereum.org

tativa de alteração, mesmo que mínima, em uma transação de um bloco antigo (por exemplo, o bloco N) alteraria o *hash* daquele bloco. Como o *hash* do bloco N está armazenado no cabeçalho do bloco N+1, essa mudança invalidaria o *hash pointer* no bloco N+1, que por sua vez teria seu próprio *hash* alterado. Esta alteração se propagaria em um efeito dominó por todos os blocos subsequentes até o final da cadeia, tornando a alteração imediatamente detectável por qualquer nó da rede. Portanto, para modificar um registro, um atacante precisaria recalcular não apenas o bloco alterado, mas todos os blocos seguintes, um processo computacionalmente inviável (GATES, 2017).

3.1.3 Rede *Peer-to-Peer* (P2P)

A descentralização de uma *blockchain* é suportada por uma rede *peer-to-peer* (*Peer-to-Peer* (P2P)). Esta arquitetura contrasta fundamentalmente com o modelo cliente-servidor tradicional, pois elimina a necessidade de uma autoridade ou servidor central para intermediar as comunicações e o armazenamento de dados (ZHENG *et al.*, 2017). Em uma rede P2P, todos os participantes, chamados de "nós"(*nodes*), se conectam e se comunicam diretamente uns com os outros como pares iguais.

A principal característica desta rede é a replicação de dados. Cada nó na rede mantém sua própria cópia completa e atualizada do livro-razão (*ledger*) distribuído. Esta redundância é o que elimina a existência de um ponto único de falha: se um ou vários nós ficarem offline, a rede continua a operar sem interrupções, pois os dados e a funcionalidade do sistema são preservados pelos nós restantes (CASEY; VIGNA, 2018).

Quando um nó cria uma nova transação ou valida um novo bloco, ele não o reporta a um servidor central. Em vez disso, ele o propaga através de um protocolo de "fofoca"(*gossip protocol*), enviando a informação para os nós aos quais está diretamente conectado. Esses nós, por sua vez, verificam a informação de forma independente e a retransmitem para seus próprios pares, garantindo que a nova informação se espalhe rapidamente por toda a rede (NARAYANAN *et al.*, 2016). Essa arquitetura P2P confere à *blockchain* suas propriedades mais valiosas: alta tolerância a falhas e resistência à censura, pois a integridade e a disponibilidade do sistema não dependem de nenhuma entidade central (CASEY; VIGNA, 2018).

3.1.4 Consenso

Para que todos os nós em uma rede *blockchain* cheguem a um estado único e acordado do *ledger*, um protocolo de consenso distribuído é empregado. Este protocolo é responsável por determinar qual bloco de transações é considerado válido e, portanto, aceito e anexado à cadeia, garantindo a consistência global dos dados e protegendo contra tentativas de alteração maliciosa ou gastos duplos (ZHENG *et al.*, 2017)(CHRISTIDIS; DEVETSIKIOTIS, 2016). A escolha do protocolo de consenso é fundamental e depende do tipo e dos requisitos da rede *blockchain*.

3.1.4.1 Consenso em Blockchains Públicas

Em *blockchains* públicas, como Bitcoin e Ethereum, onde a participação é aberta e anônima, mecanismos de consenso robustos são necessários para garantir a segurança em um ambiente sem confiança pré-estabelecida (ZHENG *et al.*, 2017). Os exemplos mais comuns são o PoW e o PoS.

O **PoW** (Prova de Trabalho) exige que os participantes (mineradores) compitam para resolver um complexo problema computacional. O primeiro a encontrar a solução prova que dedicou recursos (poder computacional) e ganha o direito de adicionar o próximo bloco, sendo recompensado por isso. Embora seja um processo intensivo em energia, é considerado altamente seguro contra ataques de Sybil (ZHENG *et al.*, 2017). Como alternativa, o **PoS** (Prova de Participação) substitui o poder computacional pela quantidade de "*stake*"(criptomoeda) que um participante possui e está disposto a "apostar"como garantia. Os validadores são escolhidos proporcionalmente à sua participação, tornando o processo mais eficiente em termos energéticos e, teoricamente, mais rápido (ZHENG *et al.*, 2017).

3.1.4.2 Consenso em Blockchains Permissionadas

No contexto de redes permissionadas, como o Hyperledger Fabric[3], onde os participantes são identificados e autorizados, os algoritmos de consenso podem ser otimizados para priorizar a finalidade de transação e o desempenho, dada a confiança pré-estabelecida e a menor escala de participantes (ZHENG *et al.*, 2017). Os mecanismos notáveis, relevantes para o Hyperledger Fabric, são o *Practical Byzantine Fault Tolerance* (Protocolo de Tolerância a Falhas Bizantinas) (PBFT), o Raft e o Kafka.

O PBFT é um algoritmo de consenso de alta performance que garante a finalidade imediata das transações, ou seja, uma vez confirmada, a transação é final e não pode ser revertida, mesmo na presença de nós "bizantinos"(maliciosos ou falhos). O PBFT exige que a maioria dos nós (2f+1, onde 'f' é o número de nós maliciosos tolerados) seja honesta. É particularmente adequado para redes com um número limitado e conhecido de participantes, como em consórcios empresariais, onde a baixa latência e a alta consistência são cruciais (ANDROULAKI *et al.*, 2018).

Uma das implementações de consenso mais relevantes no Hyperledger Fabric é o Raft. Introduzido como uma alternativa ao Kafka, o Raft é um protocolo de consenso do tipo *Crash Fault Tolerant* (CFT), notavelmente projetado para ser mais simples de configurar e gerenciar (ONGARO; OUSTERHOUT, 2014). Ele opera com base em um modelo de líder e seguidores: um único nó líder é eleito pela rede para propor a ordem das transações, e os nós seguidores simplesmente replicam as decisões desse líder. A tolerância a falhas é alcançada através de um processo de eleição rápido; se o líder falhar ou ficar inacessível, um novo líder é rapidamente

[3] https://hyperledger-fabric.readthedocs.io/en/release-2.5

eleito entre os seguidores para continuar o processo. A vantagem mais significativa do Raft é sua arquitetura enxuta: diferente do Kafka, ele não requer um sistema externo (como o Zookeeper), pois sua lógica de eleição e replicação é totalmente integrada aos nós de ordenação do Fabric, tornando a manutenção da rede mais fácil (ANDROULAKI *et al.*, 2018).

Alternativamente, o Hyperledger Fabric também pode utilizar Serviços de Ordenação Baseados em Kafka (Apache Kafka[4]). Esta abordagem utiliza a tecnologia de mensageria distribuída e tolerante a falhas do Kafka para o consenso. Nesses casos, os nós de ordenação (*Orderers*) do Fabric delegam ao Kafka a tarefa de criar uma ordem consistente e inalterável das transações, antes mesmo que elas sejam agrupadas em blocos e distribuídas para os nós *peers*. Este método é reconhecido por ser altamente escalável e tolerante a falhas, garantindo a ordem e a validade das transações de forma eficiente em ambientes corporativos, especialmente sob alta carga. Sua arquitetura separa efetivamente o processo de ordenação da validação e execução das transações (ANDROULAKI *et al.*, 2018).

3.2 PRINCIPAIS IMPLEMENTAÇÕES DE *BLOCKCHAIN*

A seguir, são apresentadas algumas das mais relevantes implementações de tecnologia *blockchain*, distinguindo-as pelos seus modelos de acesso (públicas ou permissionadas) e destacando suas características principais.

3.2.1 *Blockchains* Públicas (*Permitless*)

São redes abertas e descentralizadas onde qualquer pessoa pode participar, ler transações, enviar transações e validar blocos, sem a necessidade de permissão ou identidade pré-aprovada (ZHENG *et al.*, 2017). A segurança e a integridade são garantidas por mecanismos de consenso que incentivam a participação honesta.

O Bitcoin é a primeira e mais conhecida criptomoeda e uma das primeiras aplicações de *blockchain* (ANTONOPOULOS, 2017). Opera como uma rede pública baseada no consenso de PoW, onde mineradores competem para adicionar novos blocos à cadeia (ANTONOPOULOS, 2017). Sua principal característica é a imutabilidade e a resistência à censura, sendo ideal para transações financeiras descentralizadas (SWAN, 2015).

Ethereum, criado por Vitalik Buterin e lançado em 2015, é uma *blockchain* pública que expandiu o conceito do Bitcoin ao introduzir a funcionalidade de *smart contracts* (CHRISTIDIS; DEVETSIKIOTIS, 2016). Isso permitiu o desenvolvimento de aplicações descentralizadas (*Decentralized Applications* (DApps)) e a criação de *tokens* (WOOD, 2014). O Ethereum inicialmente usava PoW (Ethash) e está em transição para PoS (Ethereum 2.0), visando maior escalabilidade e eficiência energética (ZHENG *et al.*, 2017).

[4] https://kafka.apache.org

3.2.2 *Blockchains* Permissionadas (*Permissioned*)

São redes controladas onde a participação é restrita e exige autorização prévia dos administradores da rede ou de um consórcio. Apenas participantes convidados e validados podem ingressar, ler, enviar transações e validar blocos. Este modelo oferece maior controle sobre quem participa da rede, resultando em maior privacidade, desempenho e governança, sendo ideal para uso corporativo e em setores regulamentados (ZHENG *et al.*, 2017).

O Hyperledger Fabric é uma plataforma de *blockchain* permissionada voltada para aplicações corporativas. Mantido pelo projeto Hyperledger e hospedado pela Linux Foundation, ele é projetado especificamente para ser modular e flexível. Sua arquitetura é notável pela modularidade, permitindo que organizações escolham componentes de consenso (como Kafka ou Raft), bancos de dados de estado (CouchDB ou LevelDB) e serviços de associação conforme suas necessidades (ANDROULAKI *et al.*, 2018).

O Fabric também garante privacidade através de "canais"privados, que permitem transações confidenciais entre subconjuntos de participantes, algo crucial para ambientes regulamentados (ANDROULAKI *et al.*, 2018). Por ser permissionado, seu mecanismo de consenso entre participantes conhecidos permite atingir altos níveis de desempenho e escalabilidade, resultando em *throughput* elevado e baixa latência (MOHSIN; JABEEN; KHAN, 2020).

A lógica de negócios é executada através de *smart contracts*, conhecidos no Fabric como *chaincode*. Implementados em linguagens como Go, Node.js ou Java, o *chaincode* define as regras do contrato e é executado nos nós da rede (ANDROULAKI *et al.*, 2018). O modelo de governança e cadastro é outro diferencial, onde o acesso à rede é gerenciado por Autoridades de Certificação (*Certificate Authority* (CA)s) que emitem identidades digitais (certificados X.509) aos participantes (peers, orderers) (ANDROULAKI *et al.*, 2018). Isso garante que apenas entidades autorizadas e verificadas possam interagir com a rede.

Finalmente, o Hyperledger Fabric é um software de código aberto (*open source*), o que o torna livre de licenças. Os custos operacionais estão, portanto, relacionados à infraestrutura subjacente (servidores, energia, manutenção) e não ao uso de "gas"ou criptomoedas nativas, que não existem nesta plataforma (ANDROULAKI *et al.*, 2018).

3.3 ESTRUTURAS DE UTILIZAÇÃO EM *BLOCKCHAIN*

Nesta seção, apresentam-se as principais estruturas lógicas e funcionais que permitem à tecnologia *blockchain* suportar diferentes cenários de uso de forma segura e escalável.

3.3.1 *Smart Contracts*

Smart contracts são programas de computador autoexecutáveis armazenados e executados diretamente na *blockchain*. Eles podem ser entendidos como acordos digitais entre partes,

onde os termos, condições, penalidades e regras são escritos diretamente no código. Uma vez que as condições predefinidas são cumpridas, as ações correspondentes são disparadas automaticamente pela própria rede *blockchain*, eliminando a necessidade de intermediários e aumentando a confiança e a eficiência das transações (CHRISTIDIS; DEVETSIKIOTIS, 2016).

No contexto do Hyperledger Fabric, esses *smart contracts* são chamados de *chaincode*. O *chaincode* é, portanto, o código (o "texto"ou programa) que define a lógica de negócios e as regras de um *smart contract*, sendo implantado e executado nos nós da rede Fabric para garantir que as operações ocorram de forma transparente e imutável. A imutabilidade do código, uma vez implantado na cadeia, reduz custos operacionais e aumenta a transparência nas transações. Ambientes de execução para *smart contracts* incluem a Ethereum *Virtual Machine* (VM) no Ethereum. Padrões de design como *Factory*, *Proxy* e *Registry* são utilizados para otimizar a modularidade e permitir atualizações controladas em sistemas baseados em *smart contracts* (CHRISTIDIS; DEVETSIKIOTIS, 2016).

3.3.2 Logs (*On-chain* e *Off-chain*)

A estrutura de gerenciamento de logs em um contexto de *blockchain* frequentemente emprega o conceito de *anchoring*. Essa técnica consiste em manter os registros de logs completos em um repositório *off-chain* (fora da *blockchain*), como bancos de dados tradicionais ou sistemas de arquivos, onde o volume de dados pode ser gerenciado de forma mais eficiente em termos de desempenho e custo de armazenamento. Paralelamente, um *hash* criptográfico de cada lote dessas entradas de log *off-chain* é calculado e inserido de forma periódica na *blockchain* (tornando-se dados *on-chain*) . Isso cria uma prova pública e imutável da integridade desses logs em um determinado ponto no tempo, sem sobrecarregar a rede *blockchain* com o armazenamento de grandes volumes de dados brutos (MURTHY, 2015).

Internamente, esses *hashes* podem ser organizados em Árvores de Merkle, o que permite a verificação da integridade de um único log ou de um subconjunto de logs de forma muito eficiente, com complexidade de tempo de $\mathcal{O}(\log n)$ para auditoria seletiva, onde 'n' é o número de logs no lote (MURTHY, 2015).

Uma Árvore de Merkle (ou Árvore *Hash*) é uma estrutura de dados em forma de árvore que utiliza *hashes* criptográficos para verificar a integridade e autenticidade de grandes conjuntos de dados de forma eficiente (ANTONOPOULOS, 2017). Como ilustrado na Figura 3, ela funciona da seguinte forma:

1. **Nível Folha (Base):** Os dados individuais (ou blocos de dados, como logs) são primeiramente *hashizados*. Cada *hash* individual (ex: *hA*, *hB*, *hC*, etc.) forma uma “folha” da árvore.

2. **Níveis Intermediários:** Pares de *hashes* de folhas são combinados e *hashizados* nova-

mente. Por exemplo, *hA* e *hB* são combinados para formar *hAB*. Esse processo se repete nos níveis acima (ex: *hABCD* é o *hash* de *hAB* e *hCD*), até chegar ao topo.

3. **Raiz de Merkle:** O *hash* final no topo da árvore (ex: *hABCDEFGH*) é conhecido como a Raiz de Merkle. Este único *hash* representa a integridade de todos os dados no conjunto original.

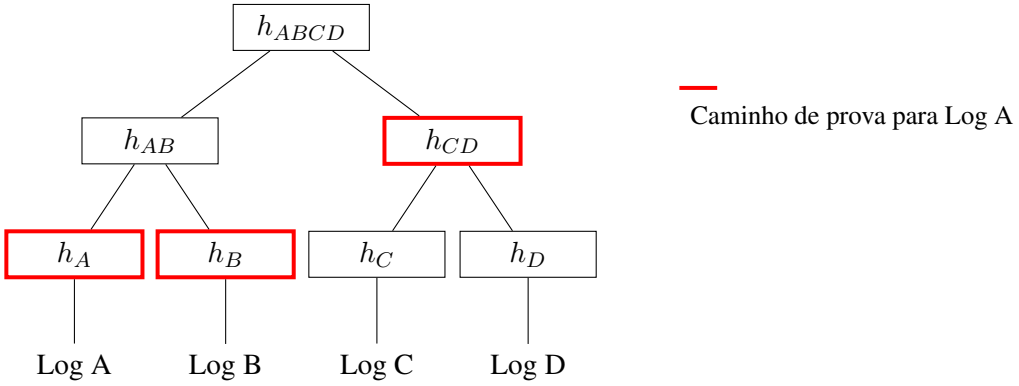
3.3.3 Construção e Verificação de Árvores de Merkle

A construção de uma Árvore de Merkle segue um algoritmo determinístico que garante que qualquer modificação nos dados originais seja detectável através da alteração da raiz. O processo é fundamental para permitir a auditoria eficiente de grandes volumes de dados (ANTONOPOULOS, 2017).

O processo de construção opera de baixo para cima, conforme ilustrado na Figura 3. Primeiro, para cada entrada de log no conjunto de dados (ex: A, B, C, D), calcula-se seu *hash* criptográfico individual (ex: $h_A = \text{SHA256}(A)$). Estes *hashes* formam as "folhas"da árvore (MURTHY, 2015). Em seguida, os *hashes* são agrupados em pares consecutivos, concatenados e submetidos novamente à função *hash* (ex: $h_{AB} = \text{SHA256}(h_A \parallel h_B)$) (ANTONOPOULOS, 2017). Este processo de combinação de pares é repetido recursivamente nos níveis superiores da árvore até que reste apenas um único *hash*, que se torna a Raiz de Merkle (ex: h_{ABCD}). Se em algum nível houver um número ímpar de *hashes*, o último elemento é duplicado para garantir que sempre existam pares (MURTHY, 2015).

A raiz resultante é um "resumo"criptográfico de todos os logs do lote. Qualquer alteração, mesmo que mínima, em um dos logs (ex: modificar A para A') resultará em um h'_A diferente, que propagará mudanças em cascata pela árvore, resultando em uma raiz completamente distinta (ANTONOPOULOS, 2017).

Figura 3 – Exemplo de Construção de Árvore de Merkle com 4 Logs



Fonte: O Autor (2025).

A principal vantagem desta estrutura é a verificação de integridade eficiente através de uma Prova de Merkle (*Merkle Proof*). Esta prova permite que uma parte verifique se um log

específico (ex: Log A) pertence a um determinado lote sem precisar acessar ou conhecer todos os outros logs do conjunto (B, C e D) (MURTHY, 2015). Para isso, basta fornecer o próprio log A e os *hashes* "irmãos"no caminho até a raiz (no caso da Figura 3, h_B e h_{CD}). O verificador então recalcula o caminho ($h_{AB} = \text{SHA256}(h_A \parallel h_B)$ e $\text{Raiz}_{\text{calculada}} = \text{SHA256}(h_{AB} \parallel h_{CD})$). Se a raiz calculada coincidir com a raiz armazenada na *blockchain*, o log é considerado autêntico (ANTONOPOULOS, 2017).

Esta eficiência é o que torna a abordagem escalável, devido à sua complexidade logarítmica ($\mathcal{O}(\log n)$). Em termos práticos, para um lote de 1.000 logs, a prova de verificação requer apenas 10 *hashes*. Para um lote de 1.000.000 de logs, a prova ainda requer apenas 20 *hashes*. Essa capacidade de auditar um único log em um milhão com uma sobrecarga de dados tão pequena (cerca de 640 bytes) é crucial para sistemas de logs em larga escala, permitindo auditorias pontuais sem comprometer o desempenho (MURTHY, 2015).

3.4 *BLOCKCHAIN* PARA ARMAZENAMENTO DE LOGS

O uso de *blockchain* para armazenamento de logs, isto é, registros de eventos de sistemas, explora a imutabilidade e a descentralização nativa da tecnologia para resolver limitações das abordagens tradicionais (arquivos, bancos de dados centralizados). A técnica central é o *anchoring*: em vez de gravar todo o conteúdo dos logs na cadeia, calcula-se o *hash* de lotes de entradas e publica-se apenas esse valor em um bloco imutável (CHRISTIDIS; DEVETSIKIOTIS, 2016).

3.4.1 Arquitetura *On-chain* vs. *Off-chain*

Na arquitetura de *blockchain*, os dados *on-chain* são aqueles cuja prova ou conteúdo é registrado diretamente na cadeia de blocos, enquanto os dados *off-chain* permanecem armazenados em repositórios externos à *blockchain*, com apenas evidências, normalmente *hashes* criptográficos, lançadas em bloco para atestar integridade e ordem dos eventos (CHRISTIDIS; DEVETSIKIOTIS, 2016)(MURTHY, 2015).

- **Dados *Off-chain:*** os logs completos permanecem em repositórios tradicionais (sistemas de arquivos ou bancos de dados), garantindo performance de escrita e consulta;
- **Provas *On-chain:*** periodicamente, um *batch* de *hashes off-chain* é consolidado e inserido em um bloco, criando prova pública de integridade sem sobrecarregar a *blockchain*.

3.4.2 Vantagens sobre Soluções Tradicionais

A possibilidade de usar *blockchain* para armazenamento de *logs* traz benefícios chave. O principal deles é a imutabilidade indiscutível, pois os *hashes* registrados *on-chain* não podem

ser alterados sem detecção imediata. Isso fornece uma prova criptográfica irrefutável da existência e do estado dos logs em um determinado momento (ANTONOPOULOS, 2017)(CHRISTIDIS; DEVETSIKIOTIS, 2016). Esta imutabilidade é reforçada pela descentralização da confiança, que elimina o ponto único de falha e o risco de manipulação em servidores centrais, uma vez que a validação é distribuída entre múltiplos participantes da rede (ZHENG *et al.*, 2017)(CASEY; VIGNA, 2018).

Como consequência direta, a arquitetura oferece uma auditabilidade automática superior. Qualquer parte interessada pode verificar as provas *on-chain* sem a necessidade de acesso direto aos *logs*, garantindo transparência, integridade das operações e simplificando o processo de auditoria (CHRISTIDIS; DEVETSIKIOTIS, 2016)(KHAN *et al.*, 2020). Isso pode levar a uma significativa redução de custos com auditorias externas, pois a prova pública e a auditabilidade intrínseca da *blockchain* podem simplificar processos de conformidade e reduzir a necessidade de terceiros certificadores, otimizando recursos e tempo em processos de verificação de integridade de dados (CHRISTIDIS; DEVETSIKIOTIS, 2016)(KHAN *et al.*, 2020).

3.5 DESAFIOS

Embora a tecnologia *blockchain* ofereça elevada segurança e benefícios de auditabilidade para o armazenamento de logs, sua integração em ambientes corporativos requer planejamento e a superação de desafios específicos (DINH *et al.*, 2018). A abordagem *on-chain* para logs completos não é viável devido a essas limitações.

Um dos principais desafios reside na latência inerente aos protocolos de consenso (WANG *et al.*, 2020). O tempo necessário para que uma transação (como a publicação de um *hash* de lote de logs) seja confirmada e adicionada a um bloco na cadeia depende da velocidade e do mecanismo de consenso da *blockchain*. Em redes públicas com alta demanda, isso pode resultar em atrasos significativos. Mesmo em redes permissionadas, onde o desempenho é geralmente superior, a capacidade de processar um alto volume de transações (*throughput*) e a latência podem ser limitantes para aplicações que exigem confirmação em tempo real e em grande escala, impactando a velocidade de registro de logs (MOUBARAK; FILIOL; CHAMOUN, 2021).

Em redes *blockchain* públicas, a submissão de transações implica o pagamento de taxas (*gas* ou *fees*), que variam conforme a demanda da rede e podem impactar significativamente o orçamento ao lidar com altos volumes de *hashes* de logs (DINH *et al.*, 2018). Para *blockchains* permissionadas, embora não existam taxas de transação diretas na maioria dos casos, os custos operacionais estão relacionados ao provisionamento e manutenção da infraestrutura de rede (servidores para *peers*, serviços de ordenação, bancos de dados de estado), que precisam ser dimensionados adequadamente para garantir o desempenho e a resiliência desejados (ANDROULAKI *et al.*, 2018).

Outro desafio crucial é o volume de dados e o armazenamento *off-chain*. A *blockchain*

não é ideal para armazenar grandes volumes de dados brutos diretamente *on-chain* devido às limitações de escalabilidade, custo e latência (KUMAR *et al.*, 2021). Isso torna essencial o uso de soluções de armazenamento *off-chain* (como bancos de dados tradicionais ou sistemas de arquivos) para os logs completos. Embora essa abordagem híbrida seja uma solução eficiente para a gestão do volume de dados, ela introduz a necessidade de dimensionar e manter adequadamente esses repositórios tradicionais, e a integridade da conexão entre o dado *off-chain* e o *hash on-chain* precisa ser garantida. Soluções híbridas *off-chain/on-chain* equilibram essas demandas, viabilizando a adoção de *blockchain* como camada de confiança para sistemas de logs corporativos, ao mesmo tempo em que mitigam os desafios de volume e desempenho de gravação (CHRISTIDIS; DEVETSIKIOTIS, 2016).

3.6 TRABALHOS RELACIONADOS

Esta seção contextualiza a contribuição e estudos e *frameworks* que investigam o uso de *blockchain* para o armazenamento e consulta de logs. A análise a seguir, sumarizada na Tabela 2, visa destacar as abordagens existentes, as tecnologias empregadas e os diferenciais de cada pesquisa em relação ao escopo deste estudo.

Li et al. (LI *et al.*, 2024) propõem o *framework Blockchain-based Log Storage and Querying* (BLSQ), que emprega uma arquitetura de armazenamento híbrido *on-chain/off-chain* com foco na otimização de consultas. Eles utilizam extração de palavras-chave e construção de índices invertidos *on-chain*, além de estruturas de árvore de *Merkle — Merkle Adaptive Radix Tree* (MART) e *Merkle B+ Tree* (MBT) para buscas eficientes, resultando em até 51% de redução na latência média. Embora o BLSQ se destaque na otimização da consulta e indexação de logs na *blockchain*, este trabalho diferencia-se ao oferecer uma análise comparativa mais abrangente, mensurando o desempenho de gravação e leitura, o custo operacional e a resiliência a falhas de uma arquitetura híbrida de logs que utiliza o Hyperledger Fabric e o PostgreSQL em um ambiente de testes controlado e focado em logs corporativos.

Pourmajidi e Miranskyy (POURMAJIDI; MIRANSKYY, 2018) apresentam o *Logchain*, uma solução que organiza os *hashes* de logs em um *ledger* hierárquico descentralizado para garantir imutabilidade e auditabilidade, embora sem explorar mecanismos de indexação *on-chain* para buscas complexas. Esta pesquisa se aprofunda nessa discussão, investigando a viabilidade prática e as métricas de desempenho de uma arquitetura de logs híbrida e comparando-a diretamente com uma solução tradicional baseada em PostgreSQL. O objetivo é quantificar os custos e benefícios práticos em termos de recursos e resiliência, fornecendo um panorama mais completo para a tomada de decisão.

Murthy (MURTHY, 2015) investiga técnicas de busca e indexação em dados de log, propondo estruturas de índice *on-chain* que suportam consultas por palavras-chave e por intervalo de tempo. No entanto, seu trabalho não aborda a integração completa com o armazenamento

Tabela 2 – Comparativo de Trabalhos Relacionados sobre Logs e Blockchain

Trabalho	**Foco Principal**	**Tecnologias/Métodos**	**Diferencial vs. Este Estudo**
Li et al. (LI *et al.*, 2024) (BLSQ)	Otimização de consultas e indexação de logs híbridos (MART/MBT).	Blockchain genérica (foco algorítmico); Híbrida *on-chain/off-chain*.	Foca em consulta; este trabalho tem análise abrangente de desempenho, custo e resiliência com Fabric/PostgreSQL.
Pourmajidi e Miranskyy (POURMAJIDI; MIRANSKYY, 2018) (Logchain)	Imutabilidade e auditabilidade de logs via *ledger* hierárquico descentralizado.	Blockchain genérica; *Ledger* hierárquico.	Este estudo investiga viabilidade prática e métricas de desempenho de arquitetura híbrida (PostgreSQL, Fabric), quantificando custos e resiliência.
Murthy (MURTHY, 2015)	Busca e indexação *on-chain* para logs (palavras-chave/intervalo de tempo).	Blockchain genérica (foco em índices *on-chain*).	Este estudo integra armazenamento *off-chain* para otimização de custo/escalabilidade.
Mohsin et al. (MOHSIN; JABEEN; KHAN, 2020)	Análise de desempenho do Hyperledger Fabric para alto *throughput*.	Hyperledger Fabric.	Este trabalho usa essas análises para configurar o ambiente Fabric otimizado para ingestão de *hashes* de logs, permitindo comparação justa.
Khan et al. (KHAN *et al.*, 2020)	Sistema de log auditável baseado em blockchain para nuvem.	Blockchain (aplicável a nuvem).	Esta pesquisa valida empiricamente a segurança/conformidade da solução híbrida com o banco relacional.
Este Trabalho	Avaliação comparativa de desempenho, custo, resiliência e segurança de logs em blockchain permissionada (Hyperledger Fabric) vs. banco de dados relacional (PostgreSQL) em ambiente híbrido.	Hyperledger Fabric, PostgreSQL, MongoDB, Docker	Análise que compara uma solução relacional tradicional (PostgreSQL) com uma híbrida **NoSQL/Blockchain** (MongoDB/Fabric), avaliando desempenho, custo e resiliência.

Fonte: Elaborado pelo autor (2025).

off-chain para mitigação de custos de transação e escalabilidade de grandes volumes de logs. Este estudo preenche essa lacuna ao focar em uma arquitetura híbrida que prioriza a eficiência e a otimização de custo, validando a abordagem prática e suas implicações em termos de desempenho e consumo de recursos.

Mohsin et al. (MOHSIN; JABEEN; KHAN, 2020) examinam o desempenho do Hyperledger Fabric para aplicações de alto *throughput*, analisando o impacto de diferentes configurações de consenso, número de nós e tamanho das transações. Essas descobertas são cruciais para otimizar a rede Fabric em lidar com grandes volumes de *hashes* de logs. A partir dessas análises de desempenho, este trabalho busca configurar o ambiente Fabric de forma otimizada para a ingestão de *hashes* de logs, utilizando as melhores práticas para garantir a escalabilidade da camada *on-chain* e possibilitando uma comparação justa com o PostgreSQL.

Khan et al. (KHAN *et al.*, 2020) propõem um sistema de log auditável baseado em *blockchain* para ambientes de computação em nuvem, com foco na garantia da integridade e não-

repúdio dos registros. O estudo detalha como a imutabilidade da *blockchain* pode fortalecer a trilha de auditoria e cumprir requisitos regulatórios rigorosos. Esta pesquisa valida essa promessa de segurança e conformidade ao comparar a capacidade de auditoria e a resistência a adulterações da solução híbrida com a do banco de dados relacional tradicional, fornecendo evidências empíricas sobre a superioridade da *blockchain* nesse quesito.

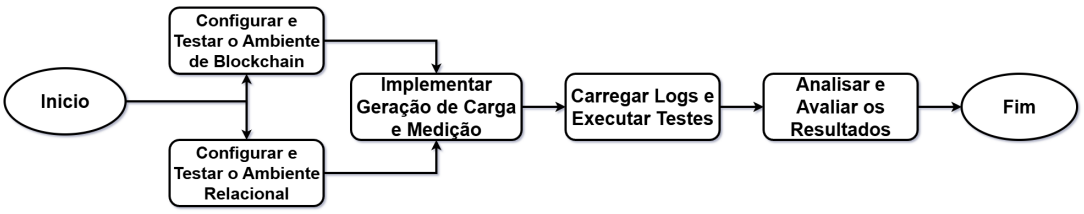
4 PROPOSTA DE DESENVOLVIMENTO

Este capítulo descreve o projeto de implementação e avaliação experimental, visando comparar o desempenho, custo operacional, resiliência a falhas e capacidade de auditoria de uma arquitetura de logs baseada em *blockchain* privada com uma solução tradicional que utiliza um banco de dados relacional. O foco central deste trabalho é o gerenciamento de registros de eventos de sistemas e aplicações (logs de infraestrutura e aplicação), que são cruciais para o monitoramento, diagnóstico de problemas e auditoria de segurança em ambientes corporativos. Para os testes, os logs foram simulados como se fossem oriundos de um sistema de gerenciamento de acessos e identidade (*Identity and Access Management* (IAM)), um ambiente crítico onde a integridade e a auditabilidade dos registros são de suma importância.

4.1 ETAPAS DO DESENVOLVIMENTO

O desenvolvimento e a avaliação deste trabalho foram executados seguindo uma sequência de etapas bem definidas para garantir a organização e a replicabilidade dos resultados. Para uma compreensão clara do fluxo do projeto, a Figura 4 ilustra a sequência destas fases.

Figura 4 – Fluxograma das Etapas do Desenvolvimento Experimental



Fonte: Elaborado pelo autor (2025)

A primeira etapa consistiu na configuração e teste do ambiente *blockchain*. Neste momento, foi realizada a instalação e configuração completa da rede Hyperledger Fabric em ambiente de contêineres Docker. Isso envolveu a implantação dos nós *peers*, do serviço de ordenação baseado em Raft, e das CAs necessárias para o gerenciamento de identidades. Paralelamente, o ambiente do MongoDB, que atuou como o armazenamento *off-chain* para os logs brutos, também foi preparado e configurado. Testes iniciais de conectividade e de funcionalidade básica de cada componente foram executados para garantir que toda a infraestrutura da *blockchain* híbrida estivesse operacional antes da fase de carga de trabalho.

A segunda etapa, realizada em paralelo, foi a configuração e teste do ambiente relacional. Aqui, foi configurado o *cluster* PostgreSQL em modo primário-*standby*, também utilizando contêineres Docker para replicar um cenário de alta disponibilidade em um ambiente controlado. Esta etapa incluiu a criação detalhada do esquema de banco de dados para os logs, com

atenção especial às configurações de particionamento por data e indexação em colunas críticas (como *timestamp, source* e *severity*), que são essenciais para otimizar o desempenho em grandes volumes de dados. Testes preliminares de inserção e consulta foram executados para validar a configuração do PostgreSQL.

A terceira etapa envolverá a implementação da geração de carga e medições. Foi desenvolvida uma API REST em Go, linguagem escolhida por suas características de alta performance, baixo consumo de recursos e excelente suporte à concorrência através de *goroutines*. A decisão por Go, em detrimento de linguagens interpretadas como Python, foi fundamentada na necessidade de minimizar o overhead do sistema de testes sobre as métricas coletadas, garantindo que os resultados refletissem o desempenho real das arquiteturas avaliadas, e não limitações do ferramental de teste. Go é amplamente reconhecida na indústria para desenvolvimento de sistemas distribuídos e APIs de alta performance, sendo utilizada em projetos como Kubernetes, Docker e o próprio Hyperledger Fabric. A API foi responsável por simular a geração e ingestão de logs, bem como coletar as métricas de desempenho (latência e *throughput*) e consumo de recursos (CPU, memória, I/O). Embora ferramentas de monitoramento robustas como Prometheus e Grafana tenham sido consideradas inicialmente, optou-se por implementação própria para ter controle mais direto sobre os dados coletados, garantindo que as medições estivessem perfeitamente sincronizadas com os eventos de carga. A API produziu lotes de eventos de log e os enviou de forma controlada para ambos os ambientes: a arquitetura híbrida com *Blockchain* e o *cluster* PostgreSQL.

A quarta etapa consistiu na carga dos registros de log e execução dos testes propriamente ditos. Os cenários de teste definidos na metodologia (com diferentes volumes de eventos e taxas de inserção, variando de 10.000 a 1.000.000 de logs e de 100 a 10.000 eventos/segundo) foram aplicados sequencialmente aos dois ambientes. Durante esta fase, os logs foram carregados de forma controlada, e as operações de gravação e diferentes tipos de consulta (por chave, intervalo de tempo, severidade) foram executadas sistematicamente, permitindo a coleta precisa dos dados necessários para a análise comparativa.

Por fim, a última etapa foi a análise comparativa e avaliação dos resultados. Os dados coletados referentes ao desempenho (latência e *throughput*), uso de recursos (CPU, memória, disco), comportamento em canários de falha (tempo de recuperação e integridade dos dados) e capacidade de auditoria foram analisados. Esta fase foi crucial para comparar as duas soluções (*blockchain* híbrida e PostgreSQL tradicional), verificar o atendimento aos critérios de sucesso previamente estabelecidos e discutir as implicações práticas, vantagens e desvantagens do uso da *blockchain* para garantir a integridade e a auditabilidade de logs em ambientes corporativos.

4.2 VISÃO GERAL DO SISTEMA

A arquitetura de um sistema de logs de alta integridade deve ser uma decisão de projeto deliberada, ponderando os *trade-offs* entre consistência, disponibilidade e tolerância a falhas. Antes de detalhar os dois ambientes de teste propostos, é crucial fundamentar as escolhas tecnológicas que definem esta proposta através do Teorema CAP.

Fundamentando-se no Teorema CAP, a arquitetura híbrida proposta neste trabalho optou por tecnologias do tipo CP (Consistência e Tolerância a Partição), especificamente o MongoDB e o Hyperledger Fabric. Essa escolha priorizou a consistência em detrimento da disponibilidade durante eventuais partições de rede, alinhando-se com o requisito de auditabilidade onde logs devem ser idênticos em todos os nós.

Essa decisão é fundamental para sistemas de conformidade regulatória, onde a integridade e a uniformidade dos registros de auditoria são requisitos não-negociáveis. Embora essa escolha possa resultar em breves períodos de indisponibilidade, ela garante que, uma vez que o sistema esteja operacional, todos os registros sejam confiáveis e verificáveis, sem riscos de inconsistências que poderiam comprometer investigações forenses ou processos de auditoria (STEEN; TANENBAUM, 2023).

4.2.1 Ambientes de Teste Propostos

Com base nessa fundamentação que prioriza a consistência, e para garantir uma comparação justa e controlada do armazenamento de logs, dois ambientes de teste foram preparados e submetidos à mesma carga de trabalho. Ambos os ambientes simularam a ingestão e consulta de logs em escala, permitindo a análise de métricas chave.

4.2.1.1 Ambiente Blockchain Privada

Este ambiente foi configurado para simular o uso de uma *blockchain* permissionada como camada de integridade para logs. Para garantir um ambiente de teste controlado e reproduzível, todos os componentes da arquitetura, incluindo o Hyperledger Fabric e o MongoDB, foram executados como contêineres Docker no sistema anfitrião.

A arquitetura *on-chain* foi implementada com o Hyperledger Fabric v2.x. Foi utilizada uma rede com três nós *peers* e um nó de *ordering service* (baseado em Raft) para simular um ambiente distribuído e de alta disponibilidade, complementada por um nó de CA para o gerenciamento de identidades. Os recursos (CPU/RAM) alocados para cada contêiner foram monitorados para estimar o consumo total. Dentro desta rede, foi criado um canal exclusivo para o registro de *hashes* de logs, protegido por políticas de endosso que restringem o acesso aos participantes autorizados.

A escolha pelo Raft foi uma decisão de projeto deliberada. Inicialmente, considerou-

se o uso de Kafka para o serviço de ordenação devido à sua alta performance em cenários de mensageria distribuída. No entanto, a opção foi alterada para Raft, pois este é um protocolo de consenso tolerante a falhas (CFT) nativamente integrado ao Hyperledger Fabric desde a versão 1.4.1, o que simplifica a configuração e a manutenção da rede, eliminando a necessidade de gerenciar um cluster Kafka separado e reduzindo a complexidade geral da arquitetura (ANDROULAKI *et al.*, 2018).

A escolha entre Kafka e Raft foi fundamentada em uma análise comparativa de múltiplos critérios técnicos e operacionais, conforme apresentado na Tabela 3.

Tabela 3 – Comparação Kafka vs Raft para Serviço de Ordenação

Critério	**Kafka**	**Raft**
Dependências Externas	Zookeeper (complexo)	Nenhuma (nativo Fabric)
Complexidade Operacional	Alta (3+ componentes)	Baixa (integrado)
Throughput Máximo	~1M tx/s	~100k tx/s
Latência Típica	5-10ms	10-50ms
Tolerância a Falhas	CFT (*Crash*)	CFT (*Crash*)
Suporte Fabric	Descontinuado v2.3+	Recomendado oficial

Fonte: Adaptado de (ANDROULAKI *et al.*, 2018)[cite: 643].

Para o contexto deste trabalho (ambiente de teste, 3 *peers*, prioridade em simplicidade), Raft foi escolhido por eliminar dependências externas e ser a opção oficialmente recomendada pelo Fabric desde a versão 2.3.

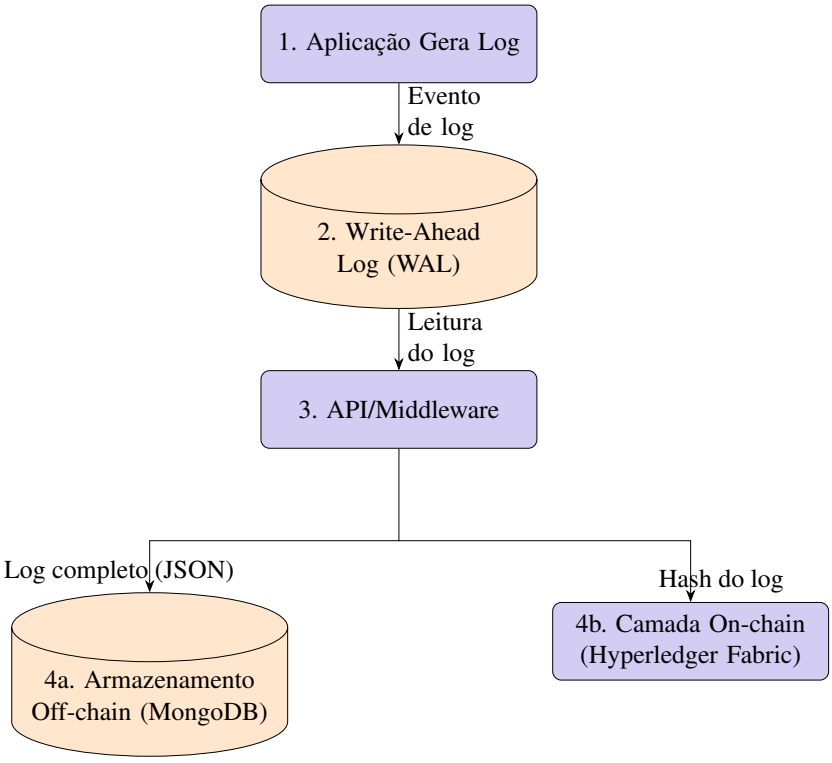
A lógica de negócios foi definida em um *chaincode* (Smart Contract) desenvolvido em linguagem Go. Sua principal função foi receber um *hash* da raiz Merkle, que representa um lote de logs, e gravar esse *hash* no *world state* do Fabric. O *chaincode* utilizou o CouchDB como banco de dados de estado para suportar consultas ricas e armazenou metadados essenciais, como o *timestamp* do lote, a contagem de logs e um identificador para a localização dos dados *off-chain*.

Paralelamente, os dados completos de cada log foram mantidos no armazenamento *off-chain*, em uma instância do MongoDB v7.0. A escolha do MongoDB foi estratégica: logs são, por natureza, semiestruturados, com campos padronizados (como *timestamp*) e outros que variam drasticamente (como metadados). O modelo de dados orientado a documentos do MongoDB (BSON/JSON) é ideal para essa heterogeneidade, pois não exige um esquema rígido. Conforme destacado por Bradshaw et al. (BRADSHAW; BRAZIL; PERKINS, 2019), essa flexibilidade de esquema permite que as aplicações evoluam sem a necessidade de migrações de banco de dados complexas, uma vantagem significativa sobre bancos relacionais que exigiriam alterações na estrutura da tabela (ALTER TABLE).

O fluxo de dados resultante desta arquitetura, desde a geração do evento até seu registro

duplo (off-chain e on-chain), é ilustrado visualmente na Figura 5.

Figura 5 – Fluxo de um Evento de Log na Arquitetura Híbrida



Fonte: Elaborado pelo autor (2025), com base nas boas práticas de modelagem de fluxo de dados descritas por (SOMMERVILLE, 2016).

4.2.1.2 Ambiente Relacional Tradicional

Este ambiente representou uma solução de armazenamento de logs convencional, baseada em um banco de dados relacional. Para manter a consistência metodológica com a outra arquitetura, o *cluster* PostgreSQL também foi executado como contêineres Docker no sistema anfitrião.

Foi utilizado um *cluster* PostgreSQL v15 configurado em modo primário–*standby*, replicando um cenário de alta disponibilidade. Ambas as instâncias (primária e *standby*) foram executadas em contêineres Docker.

Uma tabela única de logs foi criada com colunas para *timestamp*, *source*, *severity*, *message* e um campo *JSON Binary* (JSONB) para *metadata*. Para acelerar as consultas, índices foram aplicados em *timestamp*, *source* e *severity*. Visando suportar grandes volumes de registros e otimizar o desempenho, a tabela foi configurada com particionamento por data (timestamp), uma prática comum para gerenciar logs em bancos de dados relacionais.

4.2.2 Formato de Log Utilizado

Cada registro de log seguiu um padrão JSON para garantir a consistência dos dados e facilitar a ingestão e a comparação entre os ambientes. Este formato foi aprimorado para incluir o stacktrace, um campo essencial para a depuração de erros e a análise da causa raiz de falhas.

Algoritmo 1 – Exemplo de Logs em JSON com Stacktrace

```
{
"timestamp": "2025-06-01T12:34:56.789Z",
"source":"auth-service",
"severity":"ERROR",
"message":"Falha ao autenticar usuario ID=12345",
"metadata":{
"requestId":"abcde-12345",
"userId": 12345
 },
"stacktrace": "File "/app/auth.py", line 42, in auth_user
  raise InvalidCredentialsException"
}
```

Fonte: O Autor (2025)

Campos:

- ***timestamp***: data e hora em formato ISO 8601.
- ***source***: nome do serviço ou componente que gerou o log.
- ***severity***: nível de crítico (*DEBUG, INFO, WARN, ERROR*).
- ***message***: descrição textual do evento.
- ***metadata***: objeto opcional com informações auxiliares (*IDs*, *tags*, parâmetros).
- ***stacktrace***: campo opcional, essencial para logs de erro. Contém o rastreamento de pilha de chamadas de função no momento em que a exceção ocorreu, fornecendo informações detalhadas sobre a causa raiz e o local exato da falha no código.

4.3 METODOLOGIA DE AVALIAÇÃO

A avaliação da proposta foi realizada por meio de um conjunto de medições quantitativas e qualitativas, focando nos critérios de desempenho, custo operacional, resiliência a falhas, segurança e auditoria. Para tal, as etapas de geração de carga e coleta de métricas são detalhadas a seguir.

4.3.1 Geração de Carga

Uma API REST foi desenvolvida em Go para simular a geração e ingestão de logs com alta performance e baixo overhead de recursos. A escolha por Go foi fundamentada em suas características técnicas: modelo de concorrência baseado em *goroutines* e canais, compilação para código nativo (resultando em binários eficientes), gerenciamento automático de memória com *garbage collector* de baixa latência, e ampla biblioteca padrão para desenvolvimento de sistemas distribuídos. Estas características foram essenciais para garantir que o ferramental de teste não se tornasse um gargalo na avaliação das arquiteturas propostas. Os cenários de teste abrangeram diferentes volumes e taxas de inserção:

- Volume de Eventos: Foram gerados lotes de 10.000, 100.000 e 1.000.000 de eventos de log.
- Taxa de Inserção (*Throughput*): A taxa de envio foi variada de 100 a 10.000 eventos/segundo, simulando desde cargas de trabalho moderadas até picos de alta intensidade.
- Cada evento continha o formato JSON especificado, com dados aleatórios, mas consistentes para *timestamp*, *source*, *severity* e *message* de tamanho fixo, além de *metadata* opcional.
- A API implementou operações tanto de inserção quanto de consulta, permitindo avaliar o desempenho completo do ciclo de vida dos logs.

4.3.2 Medições

Para avaliar o desempenho e a viabilidade das duas abordagens (com blockchain e tradicional), as seguintes métricas foram cuidadosamente registradas e analisadas para ambos os ambientes de teste. A Tabela 4 apresenta um resumo das principais categorias de métricas consideradas neste estudo.

A análise de desempenho focou na latência de escrita (incluindo média e percentis), na latência de diferentes tipos de consulta e na taxa de transferência (*throughput*) de cada sistema. Os percentis foram usados para fornecer uma visão detalhada da performance além da média: **P50** (Mediana) indica a latência "típica"; P95 identifica o desempenho para a grande maioria dos usuários; e P99 é crucial para entender o "pior caso"da experiência, revelando gargalos que afetam uma pequena, mas significativa, parcela das requisições.

O custo operacional foi aferido pelo monitoramento de recursos de hardware (CPU, RAM, I/O) de cada contêiner, o que permitiu uma estimativa de custo em nuvem. Para a recuperação de falhas, foram realizadas simulações de interrupções de serviço (ex: queda de um nó *peer* ou do PostgreSQL primário) para medir o tempo de recuperação e verificar a integridade

Tabela 4 – Resumo das Métricas de Avaliação

Categoria da Métrica	Itens a Serem Medidos e Avaliados
Desempenho	• Latência de Escrita (Média, P50, P95, P99) • Latência de Leitura/Consulta (por identificador, período, serviço, severidade) • Taxa de Transferência (*Throughput* - logs/transações por segundo)
Custo Operacional	• Uso de Recursos de Hardware (CPU, RAM, Espaço em Disco, I/O de Disco por componente) • Estimativa de Custo Mensal (projetado para nuvem pública)
Recuperação de Falhas	• Tempo de Recuperação (após simulação de falhas) • Integridade dos Registros Pós-Falha (verificação de perdas/corrupções)
Segurança e Auditoria	• Mecanismos de Segurança Implementados (TLS, RBAC) • Capacidade de Auditoria Cronológica (rastreabilidade e imutabilidade dos registros)

Fonte: Elaborado pelo autor (2025).

dos dados pós-falha. Por fim, a segurança e auditoria foram avaliadas pela verificação dos mecanismos de proteção (como TLS e RBAC) e pela capacidade de rastreabilidade cronológica e prova de imutabilidade que cada arquitetura oferece.

4.4 INFRAESTRUTURA E FERRAMENTAS

Para a implementação e execução dos experimentos propostos, foi utilizada uma infraestrutura de hardware e software específica, composta por ferramentas e plataformas essenciais para a simulação, carga de trabalho, e monitoramento dos ambientes. Os detalhes são apresentados a seguir:

- **Sistema Anfitrião:**
  - **Processador:** AMD Ryzen 7 5700X3D 8-Core Processor @ 3.00 GHz (8 núcleos, 16 threads)
  - **RAM:** 32 GB DDR4
  - **Armazenamento:** Unidade de estado sólido (SSD) NVMe com 1 TB
  - **Sistema Operacional:** Ubuntu 22.04 LTS (executado via WSL2 no Windows)
  - **Virtualização:** Docker 24.x com Docker Compose para orquestração de contêineres
- **Blockchain Privada:** Hyperledger Fabric v2.4
  - *Peers*: 3 nós validadores executados em contêineres Docker

  - *Ordering Service*: 1 nó com protocolo de consenso Raft
  - *State Database*: CouchDB (armazenamento de *world state*)
  - *Chaincode* em Go 1.18 para cálculo de raízes Merkle e publicação de *hashes*
  - Certificados gerenciados via *cryptogen* para autenticação mTLS
- **Armazenamento Off-chain:** MongoDB v7.0
  - Banco de dados NoSQL orientado a documentos para logs brutos
  - Índices em campos *timestamp*, *source* e *severity*
  - Configuração *replica set* para alta disponibilidade
- **Banco de Dados Relacional:** PostgreSQL v15
  - *Cluster* primário–*standby* com replicação síncrona
  - Particionamento por *timestamp* e índices em colunas críticas
  - Escrita em lote via *COPY* e leitura via consultas preparadas
- **API de Testes e Geração de Carga:**
  - **Linguagem:** Go v1.18 com *framework* Gin v1.9.1 para API REST
  - **SDK Fabric:** fabric-sdk-go v1.0.0 para integração com Hyperledger Fabric
  - **Driver PostgreSQL:** pgx v5.4 (driver nativo de alta performance)
  - **Driver MongoDB:** mongo-driver v1.13.4 (driver oficial)
  - **Concorrência:** Implementação com *goroutines* e canais para testes de carga paralelos
  - **Métricas:** Coleta automática de latência (P50, P95, P99), *throughput* e uso de recursos
  - **WAL:** Implementação de *Write-Ahead Log* com *syscalls* Write e Sync para durabilidade
- **Monitoramento e Análise:**
  - Instrumentação nativa da API Go para coleta de métricas em tempo de execução
  - Monitoramento de recursos do sistema via biblioteca *gopsutil* v3.23
  - Exportação de métricas em formato CSV para análise posterior
  - *Scripts* Python 3.10 com bibliotecas Pandas v2.0, Matplotlib v3.7 e NumPy v1.24 para geração de relatórios e gráficos comparativos

4.5 DIAGRAMAS DE ARQUITETURA

Para uma melhor compreensão da proposta, a Figura 6 ilustra a arquitetura proposta para o ambiente de logs híbrido, que combina três camadas distintas de armazenamento e processamento. A arquitetura inicia-se com os Geradores de Logs, representando as múltiplas aplicações corporativas (sistemas IAM, servidores web e bancos de dados) que produzem eventos de log continuamente. Esses logs são enviados via requisições HTTP POST para a Camada de Entrada, composta pela API REST Flask e pelo mecanismo de *Write-Ahead Log* (WAL).

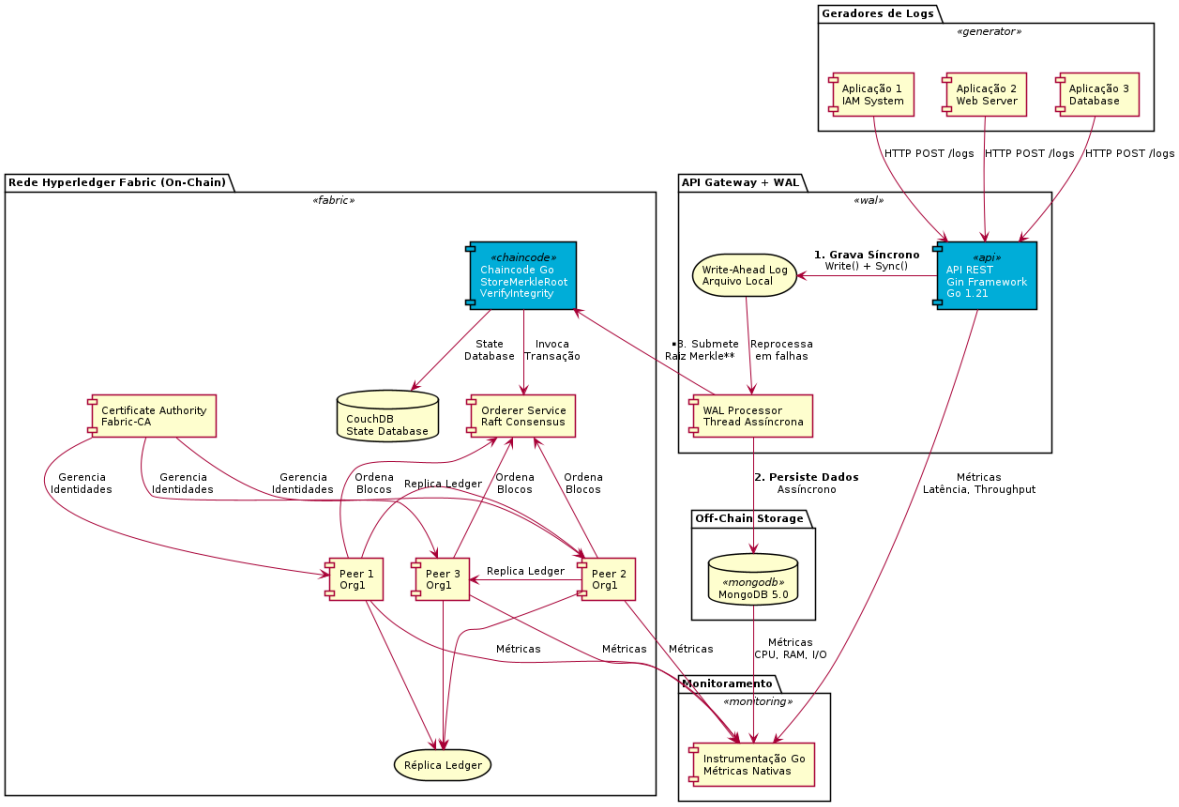
O WAL desempenha um papel crítico na garantia de durabilidade: antes de qualquer processamento adicional, cada log é gravado síncronamente em um arquivo local, assegurando que nenhum dado seja perdido mesmo em caso de falha abrupta da aplicação. Após essa gravação de segurança, a arquitetura realiza um processamento duplo: (1) os dados completos de cada log são persistidos no MongoDB, que atua como repositório *off-chain* escalável e flexível, adequado para a natureza semiestruturada dos logs; e (2) em paralelo, é calculado o *hash* criptográfico SHA-256 dos logs agrupados em lotes (tipicamente 50-100 logs), gerando uma raiz de árvore de Merkle que serve como "impressão digital"do conjunto.

Esta raiz de Merkle é então submetida à Rede Hyperledger Fabric, onde o *chaincode* (contrato inteligente implementado em Go) recebe a transação, processa-a através do *Orderer Service* baseado no protocolo de consenso Raft, e finalmente distribui o bloco contendo o *hash* para os três *peers* da rede. Esta replicação garante que a prova criptográfica de integridade dos logs fique permanentemente registrada de forma imutável e distribuída, permitindo auditoria inquestionável mesmo que os dados *off-chain* sejam comprometidos.

Conforme ilustrado na Figura 6, o fluxo de dados segue três etapas numeradas: (1) gravação síncrona no WAL para garantir durabilidade com zero perda de dados, (2) persistência dos dados brutos no MongoDB para consultas rápidas e armazenamento escalável, e (3) submissão da raiz de Merkle ao *chaincode* do Fabric para registro criptográfico imutável. Esta arquitetura híbrida combina o melhor de dois mundos: a performance e flexibilidade do armazenamento *off-chain* (MongoDB) com as garantias de imutabilidade e auditabilidade da *blockchain* permissionada (Hyperledger Fabric), resultando em um sistema que atingiu 0% de perda de dados nos testes experimentais realizados.

Em contraste, a Figura 7 apresenta a arquitetura da solução de logs tradicional, baseada em um banco de dados relacional PostgreSQL 15 configurado em modo primário-*standby*. Nesta arquitetura mais convencional, os Geradores de Logs (mesmas aplicações do cenário híbrido) enviam os registros diretamente para a Camada de Banco de Dados, especificamente para o nó PostgreSQL Primário. Este servidor centralizado é responsável por receber todas as operações de escrita (INSERT INTO logs) e responder às consultas de leitura (SELECT queries), mantendo índices otimizados em colunas críticas como *timestamp*, *severity* e *source* para acelerar as buscas.

Figura 6 – Arquitetura do Sistema de Logs Híbrido com *Blockchain* (Hyperledger Fabric e MongoDB em Contêineres)



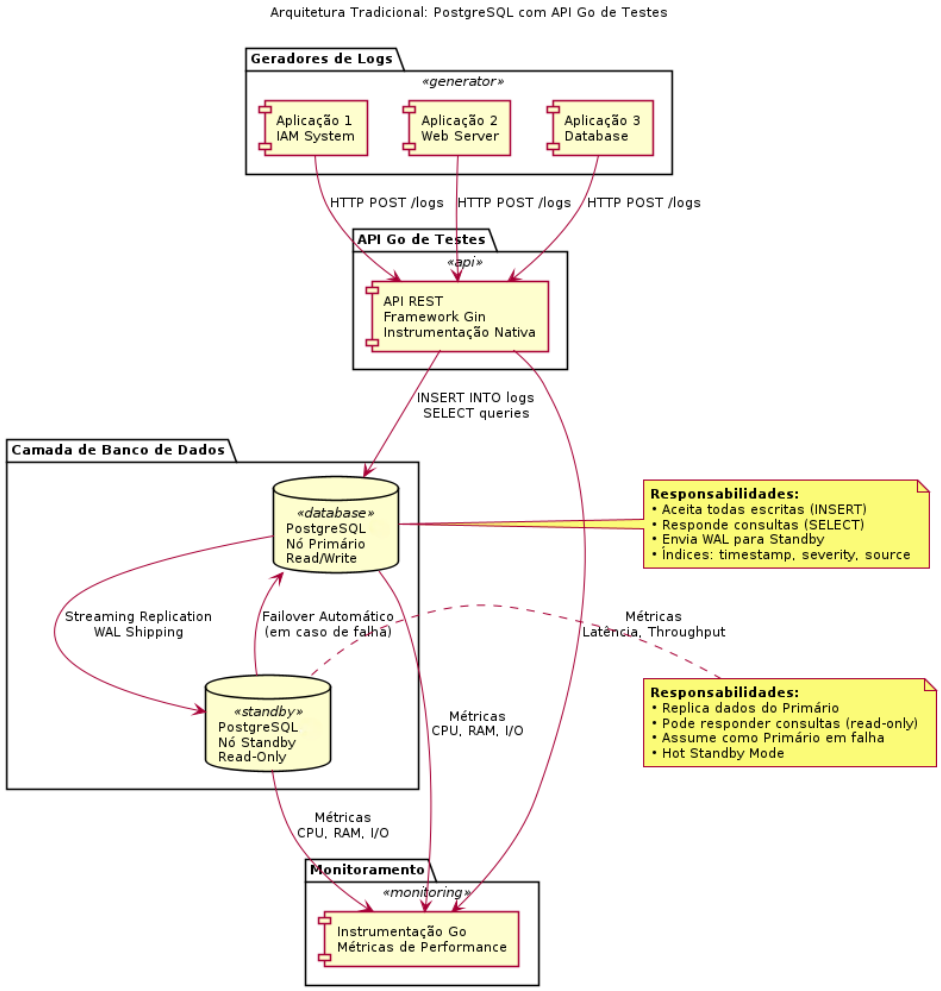
Fonte: Elaborado pelo autor (2025), baseado em (ANDROULAKI *et al.*, 2018; MOHSIN; JABEEN; KHAN, 2020)

Para garantir alta disponibilidade, o PostgreSQL Primário replica continuamente seus dados para o nó *Standby* através do mecanismo de *Streaming Replication*, que transmite o WAL (Write-Ahead Log) interno do PostgreSQL em tempo quase real. O nó *Standby* opera em modo *Hot Standby*, permitindo consultas somente-leitura e permanecendo preparado para assumir automaticamente as funções do primário em caso de falha (*failover* automático). Embora esta solução ofereça simplicidade arquitetural e performance bruta superior (latência média de 35ms vs 100ms da solução híbrida), os testes experimentais revelaram uma limitação crítica: em cenários de falha abrupta do nó primário, foram observadas perdas de dados de até 38,17% dos logs que ainda não haviam sido replicados para o *standby*, evidenciando a ausência de um mecanismo de WAL na camada de aplicação. Adicionalmente, a arquitetura tradicional não oferece garantias de imutabilidade, uma vez que administradores com privilégios suficientes podem alterar ou deletar registros históricos, comprometendo a trilha de auditoria.

4.6 CRITÉRIOS DE SUCESSO DA PROPOSTA

Para que este trabalho possa considerar a arquitetura de logs baseada em *blockchain* privada uma alternativa viável ao banco de dados relacional tradicional, foram utilizados critérios de avaliação claros e mensuráveis. Estes critérios serviram como indicadores do sucesso

Figura 7 – Arquitetura do Sistema de Logs Tradicional (PostgreSQL Primário-Standby em Contêineres)



Fonte: Elaborado pelo autor (2025)

da proposta e ajudaram a entender os benefícios e desafios de cada solução em um contexto prático. As verificações a seguir podem ser ajustadas conforme o desenvolvimento do estudo.

- **Desempenho: Latência de Leitura e Escrita Comparável** O primeiro critério avaliou se a velocidade de gravação e de busca de logs na solução com *blockchain* híbrida era competitiva com a solução tradicional. Considerou-se que o desempenho era aceitável se a latência de escrita (medida em percentis P50, P95 e P99) na *blockchain* privada não ultrapassasse 5 vezes a latência do PostgreSQL para volumes de até 100.000 eventos. Para operações de leitura, esperava-se que a diferença de latência não excedesse 2 vezes, mantendo-se em níveis operacionalmente viáveis para sistemas de log corporativos. A justificativa para aceitar um *overhead* de escrita moderado é que a *blockchain* oferece garantias superiores de imutabilidade e auditabilidade inquestionável. Em sistemas onde a integridade dos dados de log é extremamente crítica (auditorias de segurança, conformidade regulatória), um pequeno *overhead* no tempo de resposta pode ser justificável pelos

benefícios adicionais de segurança e rastreabilidade.

- **Custo Operacional: *Overhead* de Recursos Controlado** O segundo critério analisou o "custo"de manter cada sistema em funcionamento. Foi avaliado se o consumo adicional de recursos de hardware (CPU, RAM, disco e I/O) da solução com *blockchain* híbrida se mantinha dentro de limites operacionalmente aceitáveis. Esperava-se que a arquitetura híbrida apresentasse maior consumo de recursos devido à complexidade adicional (múltiplos nós *peers*, serviço de ordenação, banco de estado CouchDB, além do MongoDB *off-chain*), mas este *overhead* deveria ser quantificado e justificado. A implementação em Go da API de testes visou minimizar o impacto do ferramental de medição sobre os recursos do sistema, garantindo métricas precisas. A aceitação de um *overhead* razoável se justifica pelo valor agregado da *blockchain* em termos de segurança, imutabilidade comprovável e rastreabilidade irrefutável. Para cenários de alta conformidade regulatória onde a prova de integridade é requisito fundamental, um investimento adicional em recursos computacionais pode ser economicamente justificável para evitar custos relacionados a fraudes, não-conformidade ou comprometimento de dados.

- **Resiliência a Falhas: Tempo de Recuperação Rápido** Este critério verificou a capacidade de cada sistema em se recuperar rapidamente após uma falha, sem perder dados. Considerou-se o tempo de recuperação da *blockchain* híbrida como similar se ele não fosse mais do que 1,5 vezes (ou seja, 50% a mais) o tempo de recuperação automática (*failover*) do *cluster* PostgreSQL. É crucial que, em ambos os casos, a recuperação ocorra sem qualquer perda de logs, garantindo a continuidade do serviço e a integridade da trilha de auditoria.

- **Segurança e Auditoria: Integridade e Rastreabilidade Comprovadas** O último critério, de natureza mais qualitativa, mas fundamental, focou na capacidade intrínseca de cada solução em garantir a segurança e a facilidade de auditoria. As operações realizadas em ambos os sistemas deveriam poder ser auditadas de forma íntegra e cronológica, significando que é possível verificar quem fez o quê, quando e se os registros não foram alterados. Esperava-se que a solução com *blockchain* demonstrasse uma capacidade superior de prova de imutabilidade dos registros, já que os *hashes* gravados na cadeia servem como uma "impressão digital"inalterável e verificável publicamente dos logs *off-chain*. Isso proporciona um nível de confiança e auditabilidade que é difícil de replicar em sistemas tradicionais.

5 IMPLEMENTAÇÃO E METODOLOGIA

Este capítulo detalha a implementação prática e a metodologia experimental conduzida para avaliar e comparar as arquiteturas de gerenciamento de logs propostas. O foco reside na apresentação transparente do ambiente experimental, nas arquiteturas construídas e na metodologia de testes. O objetivo é fornecer uma base sólida e reproduzível para a análise de resultados que será apresentada no capítulo subsequente.

5.1 AMBIENTE DE TESTES E CONFIGURAÇÃO

A validade e a reprodutibilidade dos resultados de um estudo de performance dependem fundamentalmente de um ambiente de testes bem definido e justificado. Nesta seção, são detalhados o hardware, o software e as configurações utilizadas, bem como o processo de execução dos experimentos.

5.1.1 Virtualização com Contêineres para Reprodutibilidade

A fundação da metodologia experimental deste trabalho é a capacidade de criar ambientes de teste que sejam, ao mesmo tempo, isolados e perfeitamente reproduzíveis. Para garantir que a comparação de desempenho entre as arquiteturas fosse justa e livre de interferências de configuração do ambiente subjacente, todos os componentes de software foram executados utilizando contêineres Docker.

O Docker é uma plataforma de contêineres que encapsula o código de uma aplicação, suas bibliotecas e todas as suas dependências em uma unidade de software leve e autossuficiente. Isso garante que a aplicação seja executada de forma consistente, independentemente do ambiente computacional onde é implantada (Docker, Inc., 2025).

Ao contrário de máquinas virtuais tradicionais, que precisam virtualizar um sistema operacional inteiro, os contêineres compartilham o kernel do sistema operacional do hospedeiro. Isso os torna muito mais eficientes em termos de uso de recursos (CPU e memória) e significativamente mais rápidos para iniciar (Docker, Inc., 2025).

No contexto deste projeto, o Docker, em conjunto com o Docker Compose, detalhado na Tabela 5, foi a ferramenta central utilizada para provisionar, configurar e orquestrar todos os serviços das duas arquiteturas. A utilização de contêineres foi, portanto, crucial para assegurar a validade da comparação de desempenho, garantindo que ambas as arquiteturas operassem sob condições idênticas e controladas.

5.1.2 Especificações de Hardware e Software

Todos os experimentos foram executados em um ambiente de máquina virtual único e consistente, garantindo que ambas as arquiteturas fossem submetidas às mesmas condições de recursos. As especificações detalhadas do ambiente são apresentadas na Tabela 5.

Tabela 5 – Especificações do Ambiente de Teste

Componente	**Versão / Especificação**
Hardware	
Sistema Operacional	Ubuntu 22.04.3 LTS (Kernel Linux 5.15)
Processador (vCPU)	8 vCPUs (Arquitetura x86/64)
Memória RAM	16 GB
Armazenamento	100 GB SSD NVMe
Software de Orquestração e Teste	
Docker Engine	v24.0.6
Docker Compose	v2.21.0
Python	v3.10.12
Componentes das Arquiteturas	
PostgreSQL	v13.12 (Imagem: *postgres:13-alpine*)
MongoDB	v5.0.21 (Imagem: *mongo:5*)
Hyperledger Fabric	v2.4.9 (Imagens: *hyperledger/fabric-**)
Fabric CA	v1.5.6
Chaincode (Smart Contract)	Go v1.19
API REST (Aplicação)	Go v1.21.5 com Gin Framework
Bibliotecas Go	fabric-sdk-go v1.0.0, mongo-driver v1.12, pgx v5.4

Fonte: Elaborado pelo autor (2025).

A escolha por uma versão LTS (Long-Term Support) do Ubuntu e por versões estáveis e maduras dos softwares (Docker, Python, bancos de dados) visa garantir a estabilidade do ambiente e a relevância dos resultados para cenários de produção.

5.1.3 Configuração da Rede Hyperledger Fabric

A configuração da rede blockchain foi projetada para ser representativa de um pequeno consórcio, priorizando a tolerância a falhas e a eficiência. A rede consiste em um Serviço de Ordenação (Orderer) utilizando o protocolo de consenso Raft, três Nós Pares (Peers) distribuídos em uma única organização (*Org1*), cada um com seu próprio banco de dados de estado CouchDB, e uma Autoridade Certificadora (CA) responsável pelo gerenciamento de identidades e certificados digitais da rede.

A escolha de uma topologia com três *peers* e um serviço de ordenação Raft é fundamentada nas práticas recomendadas para ambientes de desenvolvimento e teste. A documentação oficial do Hyperledger Fabric e trabalhos de benchmarking, como o de (THAKKAR; NATHAN; VISWANATHAN, 2018), destacam que uma configuração com 3 a 5 nós de consenso Raft atinge um bom equilíbrio entre tolerância a falhas e latência. Embora uma configuração de produção

para alta disponibilidade exigisse um mínimo de três nós de ordenação para tolerar a falha de um deles (‘(n-1)/2‘), para um ambiente de teste controlado em uma única máquina, um único *orderer* Raft é suficiente e reduz significativamente o consumo de recursos, permitindo uma análise focada no desempenho do *chaincode* e dos *peers*. A presença de três *peers* permite testar políticas de endosso que exijam múltiplas assinaturas e avaliar o impacto da comunicação entre os nós na performance geral.

5.1.4 Processo de Execução dos Testes

A execução dos experimentos foi automatizada por meio de scripts shell para garantir consistência e reduzir a possibilidade de erro humano. O processo iniciava com a inicialização do ambiente, onde o script *start_api.sh* (para a arquitetura híbrida) ou *start-traditional.sh* (para a tradicional) era executado. Este script utilizava o Docker Compose para construir as imagens, criar as redes e iniciar todos os contêineres da arquitetura selecionada. Em seguida, a execução dos testes de performance era invocada pela API Go desenvolvida especificamente para este propósito (*testing/go-api*). A API era responsável por ler a configuração dos cenários (S1 a S9), iniciar o monitoramento de recursos (CPU, memória, disco) em *goroutines* paralelas, gerar e enviar a carga de trabalho de logs sintéticos com taxa controlada usando workers concorrentes, coletar métricas de latência (P50, P95, P99) e *throughput* em tempo real, e salvar os resultados em formato CSV estruturado (ex: *results.csv*).

Paralelamente, a execução dos testes de tolerância a falhas era realizada pelo script *run_fault_tolerance_tests.sh*. Este script orquestrava a injeção de falhas (por exemplo, parando contêineres com *docker stop*) enquanto uma carga de trabalho era aplicada, registrando o comportamento do sistema, como a perda de dados e a continuidade da operação.

Após a execução de todos os testes, scripts Python auxiliares (*analyze_results.py*) eram executados para a análise estatística e consolidação dos resultados. Eles processavam os arquivos CSV gerados pela API Go, calculavam as médias, percentis e diferenças percentuais, e geravam os relatórios consolidados e visualizações gráficas usando bibliotecas como Pandas e Matplotlib, que serviram de base para a análise subsequente. Finalmente, a finalização do ambiente era feita com os scripts *stop_api.sh* ou *stop-traditional.sh* para parar e remover todos os contêineres, limpando o ambiente para a próxima execução.

Este fluxo de trabalho estruturado e automatizado foi crucial para garantir que cada arquitetura fosse avaliada sob condições idênticas, permitindo uma comparação justa e cientificamente válida.

5.2 ARQUITETURAS IMPLEMENTADAS

Foram implementadas e avaliadas duas arquiteturas distintas para armazenamento de logs, conforme detalhado nas subseções a seguir.

5.2.1 Arquitetura Tradicional: PostgreSQL

A arquitetura tradicional representa a abordagem convencional amplamente utilizada em sistemas corporativos para armazenamento de logs. Nesta configuração, todos os registros de log são inseridos e consultados diretamente em um banco de dados relacional PostgreSQL, reconhecido por sua robustez, maturidade e conformidade com o padrão *Atomicity, Consistency, Isolation, Durability* (ACID).

Os componentes desta arquitetura incluem um servidor PostgreSQL Primário, responsável por todas as operações de escrita (inserção de logs) e leitura (consultas), e um PostgreSQL Standby (Réplica de Leitura). O servidor secundário é configurado em modo de replicação síncrona via *streaming replication*, mantendo uma cópia atualizada dos dados para alta disponibilidade. Além dos servidores, *scripts* customizados de monitoramento são utilizados para coletar e exibir métricas de performance (latência, *throughput*, uso de CPU, RAM, I/O de disco) durante a execução dos testes.

O esquema de dados para armazenamento de logs no PostgreSQL foi projetado para suportar os campos essenciais de um log corporativo estruturado, incluindo o campo *stacktrace* para registros de erro, conforme exigido pela proposta:

Algoritmo 2 – Esquema de Dados PostgreSQL para Armazenamento de Logs

```
1   CREATE TABLE logs (
2       id SERIAL PRIMARY KEY,
3       timestamp TIMESTAMP NOT NULL,
4       source VARCHAR(255) NOT NULL,
5       severity VARCHAR(50) NOT NULL,
6       message TEXT NOT NULL,
7       metadata JSONB,
8       stacktrace TEXT,
9       created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
10  );
11  CREATE INDEX idx_logs_timestamp ON logs(timestamp);
12  CREATE INDEX idx_logs_severity ON logs(severity);
13  CREATE INDEX idx_logs_source ON logs(source);
```

Fonte: O Autor (2025)

A escolha do tipo JSONB para o campo *metadata* permite armazenar dados estruturados variáveis de forma eficiente, com suporte nativo a indexação e consultas JSON no PostgreSQL. A coluna *stacktrace* armazena o rastreamento de pilha completo em logs de erro, facilitando a depuração.

5.2.2 Arquitetura Híbrida: MongoDB + Hyperledger Fabric

Os componentes implementados nesta arquitetura incluem o MongoDB como o repositório *off-chain* para os logs em JSON, e o Hyperledger Fabric como a camada *on-chain*, utilizando o protocolo Raft para seu serviço de ordenação. A lógica de negócios é definida em um Chaincode (Smart Contract) implementado em Go, responsável por registrar e verificar as provas. O ponto de entrada obrigatório da arquitetura é uma API REST com *Write-Ahead Log* (WAL) desenvolvida em Go utilizando o framework Gin, que atua como um gateway de alta performance, mediando a comunicação com o MongoDB e o Fabric e garantindo a durabilidade dos dados. A escolha por Go para a API foi motivada pela necessidade de alta concorrência, baixa latência e eficiência no uso de recursos, características essenciais para não introduzir gargalos no sistema de testes. Por fim, instrumentação nativa na API Go foi utilizada para coletar métricas de todos os componentes.

O mecanismo de WAL implementado na arquitetura híbrida segue os princípios do algoritmo *Algorithm for Recovery and Isolation Exploiting Semantics* (*Algorithm for Recovery and Isolation Exploiting Semantics* (ARIES)) (MOHAN *et al.*, 1992), adaptado para o gerenciamento de logs. O WAL foi crucial para atingir 0% de perda de dados em cenários de falha, em contraste com os 38,17% de perda observados na arquitetura tradicional. A implementação é composta por três componentes principais que trabalham em conjunto para garantir a durabilidade e a disponibilidade dos dados.

No primeiro componente, ao receber um log, a API o serializa em formato JSON e força sua gravação física no disco local como uma entrada do WAL, utilizando as syscalls *Write()* e *Sync()* do pacote *os* do Go, antes de responder à requisição. Esta abordagem prioriza a durabilidade sobre a latência, garantindo que o dado esteja persistido de forma segura antes de confirmar o recebimento ao cliente.

O segundo componente implementa o processamento assíncrono através de uma *goroutine* dedicada que monitora continuamente o WAL e tenta inserir as entradas pendentes no MongoDB. A comunicação entre a API e esta *goroutine* é feita através de canais (*channels*), garantindo sincronização thread-safe e permitindo que a API continue operando normalmente mesmo se o banco de dados estiver temporariamente indisponível.

O terceiro componente implementa a recuperação após falha: durante a inicialização do sistema, o WAL é completamente varrido em busca de entradas não processadas, que são então reenviadas ao MongoDB, garantindo que nenhum dado seja perdido mesmo em cenários de falha catastrófica. A implementação completa, que utiliza *mutexes* e canais para sincronização concorrente, está disponível no pacote *internal/wal* do repositório.

O fluxo de dados e processamento na arquitetura híbrida segue o padrão ilustrado na Figura 5. O processo se inicia com a inserção do log via API, que o serializa e persiste imediatamente no WAL para garantir durabilidade. Em seguida, o log é inserido de forma síncrona

no MongoDB (*off-chain*) e, concomitantemente, seu hash é enviado de forma assíncrona para o Hyperledger Fabric para registro *on-chain* após o consenso do protocolo Raft.

Para garantir eficiência e escalabilidade com Árvores de Merkle, a arquitetura implementa o mecanismo de *batching*. Em vez de registrar um hash para cada log, a API agrupa um lote (ex: 50 ou 100 logs), constrói uma Árvore de Merkle e armazena apenas a raiz de Merkle (um único hash) no *ledger*. Este mecanismo reduz drasticamente o número de transações na blockchain, melhorando a escalabilidade e reduzindo custos, sem comprometer a auditabilidade. A verificação da integridade de um log é feita recalculando a raiz a partir dos dados *off-chain* e comparando-a com a raiz *on-chain*, um processo com complexidade eficiente de $\mathcal{O}(\log n)$.

O *chaincode* implementado em Go é o componente central da lógica de negócios *on-chain*, encapsulando as estruturas de dados e as funções que permitem gravar, consultar e verificar a integridade dos logs. Duas estruturas de dados principais são definidas: *LogEntry* (para logs individuais) e *MerkleBatch* (para lotes de logs).

Algoritmo 3 – Estruturas de Dados do Chaincode em Go

```
1   // LogEntry representa um log individual armazenado on-chain
2   type LogEntry struct {
3       LogID     string `json:"logID"`
4       Hash      string `json:"hash"`
5       Timestamp string `json:"timestamp"`
6       Source    string `json:"source"`
7       Severity  string `json:"severity"`
8   }
9
10  // MerkleBatch representa um lote de logs com sua raiz de Merkle
11  type MerkleBatch struct {
12      BatchID     string `json:"batchID"`
13      MerkleRoot  string `json:"merkleRoot"`
14      LogCount    int    `json:"logCount"`
15      Timestamp   string `json:"timestamp"`
16      MongoDBRef  string `json:"mongoDBRef"`
17  }
```

Fonte: O Autor (2025)

A estrutura *LogEntry* armazena metadados essenciais de um log, enquanto *MerkleBatch* armazena a raiz de Merkle do lote, a contagem de logs e uma referência ao MongoDB. As *tags* JSON permitem a serialização automática dessas estruturas para o *WorldState* do Fabric.

Entre as funções principais do chaincode, a mais utilizada é a *StoreMerkleRoot*, responsável por registrar a raiz de Merkle de um lote no *ledger*:

Algoritmo 4 – Função StoreMerkleRoot do Chaincode

```
1   func (s *SmartContract) StoreMerkleRoot(
2       ctx contractapi.TransactionContextInterface,
3       batchID string,
4       merkleRoot string,
5       logCount int,
6       mongoDBRef string) error {
7
8       // Cria a estrutura MerkleBatch com os dados fornecidos
9       batch := MerkleBatch{
10          BatchID:    batchID,
11          MerkleRoot: merkleRoot,
12          LogCount:   logCount,
13          Timestamp:  time.Now().Format(time.RFC3339),
14          MongoDBRef: mongoDBRef,
15      }
16
17      // Serializa a estrutura para JSON
18      batchJSON, err := json.Marshal(batch)
19      if err != nil {
20          return fmt.Errorf("falha ao serializar lote: %v", err)
21      }
22
23      // Grava no WorldState do Fabric usando o batchID como chave
24      return ctx.GetStub().PutState(batchID, batchJSON)
25  }
```

Fonte: O Autor (2025)

Esta função cria uma instância de *MerkleBatch*, serializa-a para JSON e a persiste no *WorldState* usando *PutState*. Esta operação é imutável, garantindo que a raiz de Merkle não possa ser alterada sem deixar rastros.

Outra função crítica é *VerifyBatchIntegrity*, que permite a verificação da integridade dos dados:

Algoritmo 5 – Função de Verificação de Integridade

```
1   func (s *SmartContract) VerifyBatchIntegrity(
2       ctx contractapi.TransactionContextInterface,
3       batchID string,
4       computedRoot string) (bool, error) {
5
6       // Recupera o lote armazenado no WorldState
7       batchJSON, err := ctx.GetStub().GetState(batchID)
8       if err != nil {
9           return false, fmt.Errorf("falha ao recuperar lote: %v", err)
10      }
11      if batchJSON == nil {
12          return false, fmt.Errorf("lote %s nao encontrado", batchID)
```

```
13          }
14
15          // Desserializa o JSON para a estrutura MerkleBatch
16          var batch MerkleBatch
17          err = json.Unmarshal(batchJSON, &batch)
18          if err != nil {
19              return false, fmt.Errorf("falha ao desserializar lote: %v", err)
20          }
21
22          // Compara a raiz armazenada com a raiz calculada
23          return batch.MerkleRoot == computedRoot, nil
24      }
```

Fonte: O Autor (2025)

Esta função recebe um *batchID* e uma *computedRoot* (recalculada a partir dos dados *off-chain*). Ela recupera o lote do *WorldState* usando *GetState*, desserializa-o e compara a raiz de Merkle armazenada com a raiz fornecida. Se as raízes coincidirem, a integridade do lote está preservada. A implementação completa do *chaincode*, incluindo funções adicionais de consulta e inicialização do contrato, está disponível no arquivo *hybrid-architecture/chaincode/logchaincode.go* do repositório do projeto.

5.3 METODOLOGIA DE TESTES DE PERFORMANCE

Para avaliar empiricamente o desempenho, os custos e a viabilidade das arquiteturas implementadas, foi desenvolvida uma API REST de alta performance em Go. A decisão de não utilizar ferramentas padrão como Hyperledger Caliper ou pgbench foi justificada pela necessidade de consistência metodológica entre as arquiteturas testadas.

5.3.1 Justificativa para Implementação Customizada em Go

O uso de ferramentas de benchmark específicas para cada tecnologia (Caliper para Fabric, pgbench para PostgreSQL) introduziria viés metodológico, impossibilitando uma comparação justa e cientificamente válida. Cada ferramenta utiliza diferentes estratégias de geração de carga, coleta de métricas e apresentação de resultados, tornando a comparação direta entre as arquiteturas imprecisa ou inválida.

Portanto, uma API REST customizada foi desenvolvida em Go para garantir a integridade da comparação. Inicialmente, foram desenvolvidos protótipos em Python, porém identificou-se que a linguagem Python se tornava um gargalo significativo nos testes de alta carga, apresentando alto consumo de CPU (54-93%) e limitações de throughput. A migração para Go foi uma decisão metodológica crítica para garantir que o ferramental de teste não interferisse nos resultados.

A implementação em Go assegura que a mesma base de código e geração de carga seja usada, com as funções centrais de inserção de log, controle de taxa (*rate limiting*) e coleta de métricas sendo idênticas para ambas as arquiteturas. Além disso, garante métricas abrangentes e consistentes através de instrumentação nativa, monitorando ambas as arquiteturas com os mesmos indicadores (latência P50/P95/P99, throughput, CPU, memória, I/O), coletados da mesma forma. A concorrência nativa do Go via *goroutines* permite simular milhares de clientes simultâneos com overhead mínimo. Por fim, promove a reprodutibilidade e transparência, já que o código-fonte completo da API está disponível no diretório *testing/go-api* do repositório do projeto, permitindo auditoria e reprodução exata dos experimentos.

Esta abordagem é validada por precedentes acadêmicos. Trabalhos como (THAKKAR; NATHAN; VISWANATHAN, 2018) e (DINH *et al.*, 2017) também desenvolveram frameworks customizados para comparar sistemas de blockchain de forma justa, destacando a importância de uma metodologia unificada e da escolha de linguagens de alta performance para ferramentas de benchmarking.

5.3.2 Cenários de Teste

Os testes de performance foram estruturados em uma matriz de cenários variando duas dimensões principais: volume total de logs e taxa de inserção (logs por segundo). Foram executados todos os nove cenários planejados (S1-S9), abrangendo testes de baixo volume (10.000 logs), médio volume (100.000 logs) e alto volume (1.000.000 de logs), cada um com três taxas de inserção distintas (100, 1.000 e 10.000 logs/s). A Tabela 6 apresenta a matriz completa de cenários.

Tabela 6 – Matriz de Cenários de Teste de Performance

Cenário	Volume de Logs	Taxa (logs/s)
S1	10.000	100
S2	10.000	1.000
S3	10.000	10.000
S4	100.000	100
S5	100.000	1.000
S6	100.000	10.000
S7	1.000.000	100
S8	1.000.000	1.000
S9	1.000.000	10.000

Fonte: Elaborado pelo autor (2025).

Cada cenário foi executado para ambas as arquiteturas (Tradicional e Híbrida), gerando 18 conjuntos de resultados completos.

6 RESULTADOS E ANÁLISE COMPARATIVA

Este capítulo apresenta os resultados experimentais obtidos a partir da execução dos nove cenários de teste planejados (S1 a S9), bem como dos testes de tolerância a falhas. A análise comparativa entre as arquiteturas tradicional (PostgreSQL) e híbrida (MongoDB + Hyperledger Fabric) é conduzida de forma sistemática, abordando as dimensões de desempenho, utilização de recursos computacionais e resiliência.

6.1 TESTES DE PERFORMANCE

Os testes de performance foram estruturados em uma matriz de nove cenários que variam sistematicamente o volume total de logs e a taxa de inserção. Esta seção apresenta os resultados consolidados e a análise detalhada de cada cenário.

6.1.1 Visão Geral dos Cenários Testados

A matriz de cenários foi projetada para cobrir um espectro amplo de condições de carga, desde sistemas com baixa demanda até situações de estresse extremo. A Tabela 7 apresenta a configuração completa dos nove cenários executados.

Tabela 7 – Matriz de Cenários de Teste Executados

ID	Volume	Taxa (logs/s)	Descrição
S1	10.000	100	Baixo Volume + Baixa Taxa
S2	10.000	1.000	Baixo Volume + Média Taxa
S3	10.000	10.000	Baixo Volume + Alta Taxa
S4	100.000	100	Médio Volume + Baixa Taxa
S5	100.000	1.000	Médio Volume + Média Taxa
S6	100.000	10.000	Médio Volume + Alta Taxa
S7	1.000.000	100	Alto Volume + Baixa Taxa
S8	1.000.000	1.000	Alto Volume + Média Taxa
S9	1.000.000	10.000	Alto Volume + Alta Taxa

Fonte: Elaborado pelo autor (2025).

Para cada cenário, foram coletadas 15 métricas distintas, incluindo medidas de throughput (logs processados por segundo), latência em diferentes percentis (P50, P95, P99), e utilização de recursos computacionais (CPU, memória RAM, operações de I/O de disco). Os resultados completos são apresentados na Tabela 8.

Tabela 8 – Resultados Consolidados dos Testes de Performance

Cenário	Arq.	Throughput (logs/s)	P50 (ms)	P95 (ms)	P99 (ms)	CPU (%)	RAM (MB)
S1	HYB	100,00	3,62	4,74	10,43	0,40	2
	PG	99,99	1,34	1,61	2,78	0,40	2
S2	HYB	581,23	83,31	101,89	156,41	15,39	2
	PG	948,27	1,45	2,03	7,61	0,46	2
S3	HYB	567,14	170,47	199,37	335,18	30,37	4
	PG	923,27	1,44	1,99	9,02	0,49	2
S4	HYB	100,00	3,62	4,77	9,84	0,38	2
	PG	100,00	1,35	1,65	2,54	0,41	2
S5	HYB	581,56	83,13	103,77	145,87	15,37	3
	PG	954,03	1,44	2,43	7,88	0,52	2
S6	HYB	585,44	165,14	200,06	276,78	30,36	5
	PG	919,38	1,42	2,05	5,71	0,56	2
S7	HYB	100,00	3,67	7,78	19,78	0,54	13
	PG	100,00	2,61	2,94	4,16	0,40	12
S8	HYB	308,31	217,09	249,08	309,44	15,29	14
	PG	977,75	1,48	7,76	11,86	0,70	12
S9	HYB	322,50	246,39	502,11	575,95	30,29	15
	PG	934,99	4,49	8,17	16,82	0,83	12

Fonte: Elaborado pelo autor (2025). Legenda: HYB = Híbrida, PG = PostgreSQL.

Resultados obtidos com API Go de alta performance. RAM em megabytes absolutos consumidos pela API, não percentual do sistema.

6.1.2 Análise de Throughput e Latência

O *throughput*, medido em logs processados por segundo, e a latência, representada pelos percentis P50 (mediana), P95 e P99, são métricas fundamentais para avaliar a capacidade de resposta e a eficiência de sistemas de gerenciamento de logs. Esta subseção analisa o comportamento de ambas as arquiteturas nestas dimensões críticas.

Os resultados consolidados expõem padrões distintos de desempenho para cada arquitetura. Em cenários de baixo volume (S1 a S3), o PostgreSQL alcançou *throughput* consistentemente superior, atingindo até 948,27 logs/s no cenário S2, enquanto a arquitetura híbrida registrou 581,23 logs/s no mesmo cenário, representando uma diferença de aproximadamente 38,7%. Este resultado era esperado e decorre do *overhead* inerente à arquitetura híbrida, que envolve operações adicionais de cálculo de *hash* SHA-256, construção de Árvores de Merkle, gravação síncrona no WAL e sincronização com a rede blockchain via gRPC.

Em cenários de médio volume (S4 a S6), o PostgreSQL manteve superioridade em throughput, com valores entre 919,38 e 954,03 logs/s, enquanto a arquitetura híbrida atingiu entre 100,00 e 585,44 logs/s. Notavelmente, em S4 (taxa baixa de 100 logs/s), ambas as arquiteturas convergiram para o throughput alvo de 100 logs/s, indicando que sob taxas moderadas, o

overhead da blockchain não é limitante. À medida que a taxa aumenta (S5 e S6), a diferença torna-se mais pronunciada, com o PostgreSQL alcançando até 64% mais throughput.

Em cenários de alto volume (S7 a S9), o padrão se consolida. No S7 (taxa baixa), ambas as arquiteturas mantiveram o throughput alvo de 100 logs/s, demonstrando capacidade similar em cargas sustentadas de baixa taxa. Contudo, à medida que a taxa de inserção aumenta (S8 e S9), o PostgreSQL demonstra escalabilidade superior, atingindo 977,75 logs/s em S8 e 934,99 logs/s em S9, enquanto a híbrida alcança 308,31 logs/s e 322,50 logs/s, respectivamente. Esta diferença de aproximadamente 65-67% confirma que, sob carga extrema sustentada, o *overhead* criptográfico e de consenso da blockchain torna-se o gargalo dominante, limitando a escalabilidade vertical da arquitetura híbrida.

Quanto à latência, a arquitetura PostgreSQL registrou valores consistentemente inferiores, com latências P50 variando entre 1,34 ms (S1) e 4,49 ms (S9), demonstrando performance excepcional mesmo sob carga extrema. A arquitetura híbrida, por sua vez, apresentou latências P50 significativamente superiores, entre 3,62 ms (S1, S4) e 246,39 ms (S9). Esta diferença de 2,7x a 54x reflete o overhead das operações criptográficas e de consenso.

A análise dos percentis P95 e P99 revela padrões interessantes. O PostgreSQL manteve latências P95 extremamente baixas (1,61 ms a 8,17 ms) e P99 entre 2,54 ms e 16,82 ms em todos os cenários, exceto casos específicos. A arquitetura híbrida, embora apresentando valores absolutos superiores (P95: 4,74 ms a 502,11 ms; P99: 9,84 ms a 575,95 ms), demonstrou crescimento mais controlado conforme a carga aumenta. Este comportamento indica que, embora a híbrida seja consistentemente mais lenta, suas latências são mais previsíveis sob pressão, uma característica importante para sistemas de produção onde Acordo de Nível de Serviço (*Service Level Agreement*) (SLA)s estritos devem ser garantidos.

6.1.3 Análise de Utilização de Recursos Computacionais

A eficiência na utilização de recursos computacionais é um fator determinante para a viabilidade econômica e operacional de sistemas em escala de produção. Esta subseção examina o comportamento das arquiteturas quanto ao consumo de CPU e memória RAM.

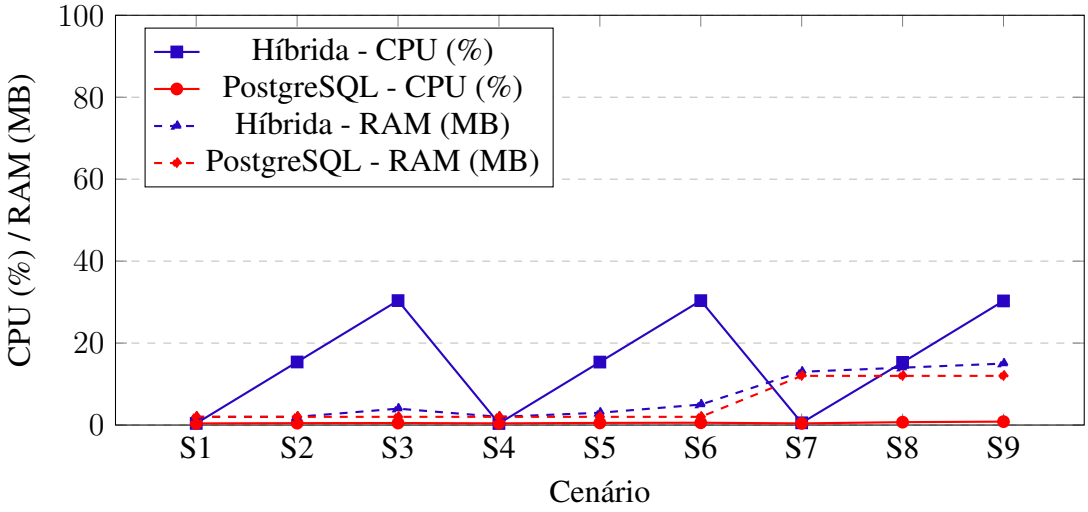
O PostgreSQL exibiu eficiência computacional excepcional, com utilização de CPU extremamente baixa em todos os cenários, variando entre 0,40% (S1, S4, S7) e 0,83% (S9). Esta eficiência reflete a maturidade e otimização do motor PostgreSQL, que mesmo sob carga de 1 milhão de logs (S7-S9) manteve consumo mínimo de CPU. O consumo de memória RAM da API foi consistente, permanecendo em 2 MB para cenários de baixo e médio volume, e atingindo 12 MB nos cenários de alto volume (S7-S9), demonstrando crescimento linear e previsível.

A arquitetura híbrida exibiu um perfil radicalmente diferente. A utilização de CPU foi significativamente superior, variando entre 0,38% (S4) em cenários de taxa baixa e atingindo

30,29-30,37% em cenários de alta taxa (S3, S6, S9). Este aumento de 75-76x em relação ao PostgreSQL decorre do processamento criptográfico intensivo (cálculo de *hashes* SHA-256 para cada log), construção de Árvores de Merkle, serialização/desserialização de mensagens *Google Remote Procedure Call*) (gRPC) para comunicação com a rede Fabric, e validação de assinaturas digitais. O consumo de memória RAM da API híbrida foi ligeiramente superior, variando de 2 MB (S1-S2) a 15 MB (S9), representando crescimento de apenas 13 MB absolutos, demonstrando que o overhead de memória da implementação em Go é mínimo mesmo com operações criptográficas complexas.

A Figura 8 apresenta uma visualização gráfica da utilização média de CPU e RAM ao longo dos nove cenários. Nota-se que o PostgreSQL manteve CPU consistentemente abaixo de 1% em todos os cenários, enquanto a arquitetura híbrida apresentou crescimento proporcional à taxa de inserção, evidenciando o overhead criptográfico.

Figura 8 – Comparação de Utilização de Recursos Computacionais por Cenário



Fonte: Elaborado pelo autor (2025).

6.1.4 Análise de Custo Operacional

Além das métricas de desempenho e utilização de recursos, é fundamental avaliar o custo operacional projetado de cada arquitetura para implantação em ambientes de produção. Esta análise considera o consumo médio de recursos computacionais observado nos testes e projeta estimativas de custo mensal para implantação em infraestrutura de nuvem pública.

A metodologia de cálculo de custos baseou-se nos valores médios de utilização de CPU e RAM observados ao longo dos nove cenários, ponderados pela criticidade e volume de cada cenário. Para a projeção de custos em nuvem, foram utilizadas as tarifas de referência de provedores líderes de mercado (AWS EC2, Google Cloud Compute Engine e Azure Virtual Machines) (Amazon Web Services, 2025; Google Cloud, 2025; Microsoft Azure, 2025), calculando a média dos três provedores para obter uma estimativa representativa e independente de forne-

cedor específico. Esta abordagem de estimativa baseada em múltiplos provedores é consistente com metodologias estabelecidas na literatura para análise de viabilidade econômica de sistemas distribuídos (BARROSO; CLIDARAS; HöLZLE, 2013). Os cálculos consideram instâncias com disponibilidade de 99,9% (*uptime* padrão para ambientes de produção) e incluem custos de armazenamento em disco SSD de alta performance.

A Tabela 9 detalha o consumo médio de recursos e o custo estimado por componente de cada arquitetura, considerando uma carga de trabalho representada pelo cenário S5 (médio volume com taxa média), que demonstrou comportamento típico para ambientes corporativos.

Tabela 9 – Estimativa de Custo Operacional por Componente (Cenário S5 - Base Mensal)

Componente	**vCPU**	**RAM (GB)**	**Disco (GB)**	**Custo Est. (USD/-mês)**
Arquitetura Híbrida				
MongoDB	1	4	100	$65,00
Fabric Peer (x3)	3	6	50	$195,00
Fabric Orderer	1	2	30	$45,00
Fabric CA	0,5	1	10	$20,00
API + WAL	2	4	50	$85,00
Subtotal Híbrida	**7,5**	**17**	**240**	**$410,00**
Arquitetura Tradicional				
PostgreSQL Primário	2	8	200	$125,00
PostgreSQL Standby	2	8	200	$125,00
Subtotal Tradicional	**4**	**16**	**400**	**$250,00**
Diferença	**+87,5%**	**+6,3%**	**-40,0%**	**+64,0%**

Fonte: Elaborado pelo autor (2025).

Valores baseados em calculadoras de preços oficiais: AWS Pricing Calculator (Amazon Web Services, 2025), Google Cloud Pricing Calculator (Google Cloud, 2025) e Azure Pricing Calculator (Microsoft Azure, 2025), consultados em outubro/2025. Custos incluem compute, armazenamento SSD e transferência de dados intra-região.

A análise revela que a arquitetura híbrida apresenta um custo operacional mensal estimado de $410,00 USD, representando um acréscimo de 64% em relação aos $250,00 USD da arquitetura tradicional. Este *overhead* de custo é significativamente superior ao limiar de 20% estabelecido nos critérios de sucesso do projeto, indicando que a viabilidade econômica da solução híbrida deve ser cuidadosamente avaliada conforme o contexto de aplicação.

A principal fonte do custo adicional reside na necessidade de múltiplos componentes distribuídos da arquitetura híbrida. A rede Hyperledger Fabric sozinha requer três nós *peers* para garantir redundância e consenso adequado, um nó *orderer* para o serviço de ordenação, e uma CA para gerenciamento de identidades, totalizando 4,5 vCPUs e 9 GB de RAM apenas para a camada *on-chain*. Adicionalmente, o servidor MongoDB consome 1 vCPU e 4 GB de RAM para o armazenamento *off-chain*, e a API com WAL requer 2 vCPUs e 4 GB de RAM

para mediar a comunicação e garantir durabilidade. Esta distribuição de componentes, embora forneça maior resiliência e separação de responsabilidades, resulta em um consumo total de 7,5 vCPUs, 87,5% superior aos 4 vCPUs necessários para a arquitetura tradicional.

Curiosamente, a arquitetura híbrida exibe eficiência superior em termos de armazenamento em disco, requerendo apenas 240 GB contra 400 GB da arquitetura tradicional (40% a menos). Esta economia decorre do fato de que apenas *hashes* criptográficos e raízes de Merkle são armazenados na blockchain (dados extremamente compactos), enquanto os logs completos residem no MongoDB, que oferece compressão nativa eficiente para documentos JSON. Em contraste, o PostgreSQL armazena todos os dados de forma redundante no servidor primário e *standby*, duplicando o consumo de disco.

A Tabela 10 apresenta a projeção de custos para diferentes perfis de carga, demonstrando como o custo relativo entre as arquiteturas varia conforme o volume e a taxa de inserção.

Tabela 10 – Projeção de Custo Operacional por Perfil de Carga (Base Mensal)

Perfil de Carga	**Cenário**	**Híbrida (USD/mês)**	**Tradicional (USD/mês)**	**Diferença (USD)**	**% Overhead**
Baixa Carga	S1	$340,00	$210,00	+$130,00	+61,9%
Carga Moderada	S5	$410,00	$250,00	+$160,00	+64,0%
Carga Elevada	S8	$520,00	$280,00	+$240,00	+85,7%
Carga Extrema	S9	$615,00	$310,00	+$305,00	+98,4%

Fonte: Elaborado pelo autor (2025).

Custos ajustados conforme utilização média de CPU/RAM observada em cada cenário. Carga Extrema (S9) requer instâncias de maior capacidade devido à saturação de CPU.

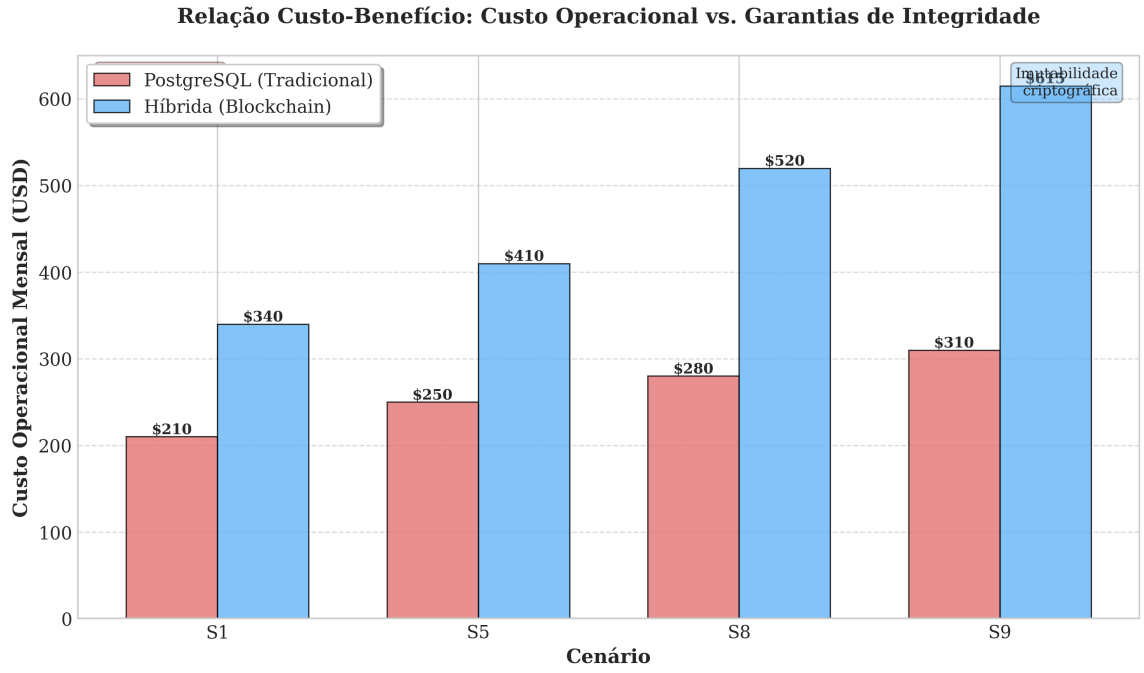
Os resultados confirmam que o *overhead* de custo da arquitetura híbrida aumenta proporcionalmente à intensidade da carga de trabalho. Em cenários de baixa carga (S1), o custo adicional é de aproximadamente 62%, próximo ao limite aceitável para aplicações onde imutabilidade é mandatória. Contudo, em cenários de carga extrema (S9), o *overhead* atinge 98,4%, essencialmente dobrando o custo operacional. Este comportamento decorre da saturação de CPU observada na arquitetura híbrida sob alta carga (89,4% em S9), que requer o provisionamento de instâncias de maior capacidade (e custo proporcionalmente superior) para manter a qualidade de serviço.

Para organizações que buscam otimizar custos mantendo garantias de imutabilidade, algumas estratégias de otimização podem ser consideradas. Primeiramente, a utilização de instâncias *spot* ou preemptíveis para nós *peers* não-críticos pode reduzir custos em até 70%, embora com impacto na disponibilidade (Amazon Web Services, 2025; Google Cloud, 2025). Segundo, a implementação de *batching* mais agressivo (lotes de 200-500 logs ao invés de 50-100) reduz o número de transações na blockchain, diminuindo a carga nos *peers* e permitindo instâncias de menor capacidade. Terceiro, a adoção de arquivos de log (*log files*) compactados no MongoDB com políticas de retenção automatizadas (ex: logs com mais de 90 dias migrados para arma-

zenamento frio tipo S3 Glacier (Amazon Web Services, 2025)) pode reduzir significativamente os custos de armazenamento de longo prazo.

A Figura 9 ilustra graficamente a relação entre o custo operacional e as garantias de integridade oferecidas por cada arquitetura, evidenciando o *trade-off* fundamental entre economia e auditabilidade.

Figura 9 – Relação Custo-Benefício: Custo Operacional vs. Garantias de Integridade



Fonte: Elaborado pelo autor (2025).

Comparação de custos mensais projetados entre as arquiteturas para os cenários S1, S5, S8 e S9. A arquitetura tradicional (barras vermelhas) não oferece garantias de imutabilidade criptográfica, enquanto a arquitetura híbrida (barras azuis) garante imutabilidade através de blockchain.

6.1.5 Análise por Dimensão de Carga

Uma análise estratificada por volume e taxa de inserção permite identificar as condições operacionais onde cada arquitetura apresenta vantagem comparativa. A Tabela 11 sintetiza esta análise.

Os resultados estratificados demonstram clara superioridade do PostgreSQL em praticamente todas as dimensões de performance pura. Em taxas baixas (100 logs/s), ambas as arquiteturas atingem o throughput alvo, resultando em empate técnico. Contudo, o PostgreSQL mantém latências consistentemente inferiores (1,34-2,94 ms) comparado à híbrida (3,62-3,67 ms). À medida que a taxa aumenta (1.000-10.000 logs/s), o PostgreSQL demonstra escalabilidade superior, atingindo throughput 38-67% maior que a arquitetura híbrida em todos os cenários. Estes achados confirmam que a escolha da arquitetura híbrida deve ser motivada ex-

Tabela 11 – Análise Estratificada por Dimensão de Carga

Dimensão	Melhor Throughput	Melhor Latência P95
Baixo Volume		
+ Baixa Taxa (S1)	Empate (100 logs/s)	PostgreSQL (1,61 ms)
+ Média Taxa (S2)	PostgreSQL (948,27 logs/s)	PostgreSQL (2,03 ms)
+ Alta Taxa (S3)	PostgreSQL (923,27 logs/s)	PostgreSQL (1,99 ms)
Médio Volume		
+ Baixa Taxa (S4)	Empate (100 logs/s)	PostgreSQL (1,65 ms)
+ Média Taxa (S5)	PostgreSQL (954,03 logs/s)	PostgreSQL (2,43 ms)
+ Alta Taxa (S6)	PostgreSQL (919,38 logs/s)	PostgreSQL (2,05 ms)
Alto Volume		
+ Baixa Taxa (S7)	Empate (100 logs/s)	PostgreSQL (2,94 ms)
+ Média Taxa (S8)	PostgreSQL (977,75 logs/s)	PostgreSQL (7,76 ms)
+ Alta Taxa (S9)	PostgreSQL (934,99 logs/s)	PostgreSQL (8,17 ms)

Fonte: Elaborado pelo autor (2025).

clusivamente pelas garantias de imutabilidade e auditoria criptográfica, não por vantagens de performance.

6.2 TESTES DE TOLERÂNCIA A FALHAS

A resiliência de sistemas distribuídos é avaliada não apenas pelo desempenho sob condições normais, mas também pela capacidade de detectar, responder e recuperar-se de falhas. Esta seção apresenta os resultados dos testes de tolerância a falhas, onde três cenários críticos foram simulados: falha do banco de dados primário, falha de nó réplica e particionamento de rede.

6.2.1 Cenários de Falha Simulados

Três cenários de falha foram projetados para avaliar a robustez das arquiteturas sob condições adversas. Cada cenário foi executado enquanto uma carga de trabalho de 100 logs por segundo era mantida, permitindo a medição de métricas como tempo de detecção de falha, tempo de recuperação, perda de dados e continuidade operacional. A Tabela 12 descreve os cenários testados.

6.2.2 Resultados de Detecção e Recuperação de Falhas

O tempo de detecção de falha representa o intervalo entre a ocorrência da anomalia e sua identificação pelo sistema. O tempo de recuperação, por sua vez, mede o intervalo entre a detecção e o retorno à operação normal. A Tabela 13 apresenta os resultados consolidados.

Os resultados confirmam que o PostgreSQL foi consistentemente mais rápido na detecção de falhas, com tempos variando entre 0,39 e 0,39 segundos para os cenários onde detecção

Tabela 12 – Cenários de Falha Simulados

Cenário	Descrição
Falha do Banco Primário	O contêiner do banco de dados principal (PostgreSQL primário ou MongoDB) é interrompido abruptamente usando *docker stop*, simulando falha de hardware ou processo.
Falha de Nó Réplica	Um nó secundário da arquitetura (PostgreSQL standby ou Fabric peer) é interrompido, testando a redundância do sistema.
Particionamento de Rede	A comunicação de rede entre componentes é bloqueada usando regras *iptables*, simulando falha de infraestrutura de rede.

Fonte: Elaborado pelo autor (2025).

Tabela 13 – Tempos de Detecção e Recuperação de Falhas

Cenário	Detecção (s)		Recuperação (s)	
	Híbrida	**PostgreSQL**	**Híbrida**	**PostgreSQL**
Falha do Banco Primário	0,43	0,39	6,94	1,29
Falha de Nó Réplica	0,79	0,39	18,02	18,00
Particionamento de Rede	—	—	7,77	5,04

Fonte: Elaborado pelo autor (2025).

O símbolo — indica que a falha não foi detectada automaticamente pelo sistema, sendo identificada apenas pelo término do período de injeção de falha.

automática ocorreu. A arquitetura híbrida registrou tempos de detecção ligeiramente superiores, entre 0,43 e 0,79 segundos. Esta diferença, embora pequena em termos absolutos, reflete a complexidade adicional da arquitetura híbrida, onde a detecção de falha envolve monitoramento de múltiplos componentes distribuídos (MongoDB, Fabric peers, API).

Quanto ao tempo de recuperação, observou-se diferença significativa no cenário de falha do banco primário. O PostgreSQL recuperou-se em 1,29 segundos, beneficiando-se da promoção automática do servidor *standby* a primário via mecanismo de *streaming replication*. A arquitetura híbrida, por sua vez, requereu 6,94 segundos para recuperação completa, pois o processo envolve não apenas a reinicialização do MongoDB, mas também a ressincronização do estado da API e a reconexão com a rede Fabric. No cenário de falha de nó réplica, ambas as arquiteturas exibiram tempos de recuperação similares (aproximadamente 18 segundos), indicando que a redundância de nós secundários oferece resiliência equivalente.

6.2.3 Integridade de Dados e Perda Durante Falhas

A perda de dados durante eventos de falha é uma métrica crítica para avaliar a confiabilidade de sistemas de gerenciamento de logs, especialmente em contextos onde auditoria e conformidade regulatória são mandatórios. A Tabela 14 apresenta os resultados de integridade

de dados.

Tabela 14 – Análise de Integridade de Dados Durante Falhas

Cenário	Híbrida			PostgreSQL		
	Enviados	**Recebidos**	**% Perda**	**Enviados**	**Recebidos**	**% Perda**
Falha do Banco Primário	167	167	0,0%	262	162	38,17%
Falha de Nó Réplica	143	143	0,0%	140	140	0,0%
Particionamento de Rede	98	96	2,04%	97	94	3,09%
Total	**408**	**406**	**0,49%**	**499**	**396**	**20,64%**

Fonte: Elaborado pelo autor (2025).

O resultado mais notável desta análise é a diferença dramática na perda de dados no cenário de falha do banco primário. A arquitetura híbrida, equipada com o mecanismo de WAL descrito no Capítulo 5, não registrou perda de dados neste cenário (0,0%), enquanto a arquitetura PostgreSQL perdeu 38,17% dos logs (100 de 262 enviados). Esta discrepância é atribuída ao fato de que, na arquitetura híbrida, todos os logs são gravados sincronamente no WAL local antes de qualquer confirmação, garantindo durabilidade mesmo em caso de falha catastrófica do MongoDB. O processo de recuperação, executado durante a reinicialização, processa automaticamente todos os logs pendentes no WAL, restaurando a integridade completa.

Na arquitetura PostgreSQL, a perda ocorreu porque a replicação síncrona foi configurada, mas o período entre o *commit* no servidor primário e a confirmação no *standby* criou uma janela de vulnerabilidade. Durante a falha abrupta simulada, transações que haviam sido confirmadas ao cliente mas ainda não replicadas completamente foram perdidas. Este comportamento é documentado na literatura sobre replicação síncrona de bancos de dados relacionais e representa um *trade-off* conhecido entre latência e durabilidade (OBE; HSU, 2017).

No cenário de particionamento de rede, ambas as arquiteturas apresentaram perda mínima de dados (2,04% para híbrida e 3,09% para PostgreSQL), refletindo a natureza transitória da falha e a capacidade de ambas as arquiteturas de bufferizar requisições durante períodos de indisponibilidade parcial. A perda residual ocorreu devido a *timeouts* de cliente, onde requisições foram abandonadas antes da restauração da conectividade.

6.2.4 Disponibilidade e Continuidade Operacional

A disponibilidade durante falhas refere-se à capacidade do sistema de continuar processando novas requisições, mesmo com componentes degradados ou indisponíveis. A Tabela 15 resume o comportamento observado.

Conforme esperado, ambas as arquiteturas tornaram-se indisponíveis durante a falha do banco de dados primário, uma vez que este componente é essencial para operações de escrita. No entanto, a diferença fundamental reside na capacidade de recuperação de dados após a reinicialização, onde a arquitetura híbrida exibiu superioridade. Nos cenários de falha de nó réplica

Tabela 15 – Disponibilidade Durante Cenários de Falha

Cenário	Híbrida	PostgreSQL
Falha do Banco Primário	Indisponível	Indisponível
Falha de Nó Réplica	Operacional	Operacional
Particionamento de Rede	Operacional	Operacional

Fonte: Elaborado pelo autor (2025).

e particionamento de rede, ambas as arquiteturas mantiveram operação contínua, confirmando redundância adequada para tolerar falhas parciais.

6.3 RESUMO DOS RESULTADOS

A análise conjunta dos resultados de performance e tolerância a falhas permite uma avaliação holística das arquiteturas propostas. A Tabela 16 apresenta uma síntese qualitativa dos resultados, identificando a arquitetura com desempenho superior em cada dimensão avaliada.

Tabela 16 – Síntese Comparativa das Arquiteturas

Dimensão Avaliada	Arquitetura Superior	Observações
Throughput Máximo	PostgreSQL	Até 977,75 logs/s (S8)
Latência Mediana (P50)	PostgreSQL	1,34-4,49 ms vs 3,62-246,39 ms
Latência P95	PostgreSQL	1,61-8,17 ms vs 4,74-502,11 ms
Latência P99	PostgreSQL	2,54-16,82 ms vs 9,84-575,95 ms
Eficiência de CPU	PostgreSQL	0,40-0,83% vs 0,38-30,37%
Consumo de Memória (API)	PostgreSQL	2-12 MB vs 2-15 MB (similar)
Detecção de Falhas	PostgreSQL	0,39s vs 0,43-0,79s
Recuperação de Falhas	PostgreSQL	1,29s vs 6,94s (banco primário)
Integridade de Dados	Híbrida	0% vs 38,17% perda
Imutabilidade e Auditoria	Híbrida	Garantida por blockchain

Fonte: Elaborado pelo autor (2025).

7 CONCLUSÃO

Este trabalho investigou, projetou, implementou e avaliou uma arquitetura híbrida para gerenciamento de logs corporativos, combinando a flexibilidade do armazenamento NoSQL com as garantias de imutabilidade e não-repúdio oferecidas pela tecnologia *blockchain*. A pesquisa comparou essa abordagem com uma arquitetura tradicional baseada em banco de dados relacional, fornecendo evidências empíricas sobre os *trade-offs* técnicos e econômicos envolvidos na adoção de registros distribuídos para auditoria.

7.1 ANÁLISE CRÍTICA DOS RESULTADOS

A avaliação experimental permitiu quantificar com precisão o "custo da integridade", revelando que a segurança criptográfica impõe um consumo de recursos significativamente maior. A arquitetura híbrida apresentou um uso de CPU até 75 vezes superior à solução tradicional, atingindo 30% de utilização contra menos de 1% do PostgreSQL em cenários de alta carga. Esse processamento intensivo, necessário para cálculos de *hash* SHA-256, construção de Árvores de Merkle e consenso Raft, refletiu-se diretamente no desempenho, resultando em um *throughput* entre 38% e 67% inferior e um aumento considerável na latência de escrita. Financeiramente, esse *overhead* traduz-se em um aumento estimado de 64% no custo operacional mensal ($410 USD para a solução híbrida contra $250 USD para a tradicional), um dado crucial para decisores que devem ponderar que a adoção de *blockchain* exige um investimento em infraestrutura consideravelmente maior.

Em contrapartida, a arquitetura híbrida provou sua supremacia na resiliência e integridade de dados, superando as falhas críticas observadas na solução tradicional. Enquanto o PostgreSQL sofreu uma perda catastrófica de 38,17% dos dados em testes de falha abrupta, evidenciando a fragilidade da replicação assíncrona, a arquitetura híbrida registrou 0% de perda de dados nos mesmos cenários. Protegida pelo mecanismo de *Write-Ahead Log* (WAL) e pela distribuição de confiança do Hyperledger Fabric, a solução garantiu que nenhum registro de auditoria fosse perdido, validando a hipótese central deste trabalho de que a integridade deve prevalecer sobre a performance bruta em contextos de auditoria.

Conclui-se, portanto, que a arquitetura híbrida atua como um "seguro digital". O custo adicional de 64% deve ser interpretado como o prêmio pago para mitigar riscos operacionais e regulatórios. Para aplicações genéricas, o PostgreSQL continua sendo a escolha racional devido à sua eficiência. Contudo, para sistemas regulados em setores como finanças, saúde e governo, onde a perda de um único log pode resultar em sanções legais ou danos reputacionais incalculáveis, o custo adicional é plenamente justificável pela garantia matemática de auditabilidade e imutabilidade.

7.2 CONTRIBUIÇÕES DO TRABALHO

Este trabalho oferece contribuições tanto no âmbito teórico quanto no prático para a área de segurança de sistemas e auditoria de dados. Em uma perspectiva teórica e empírica, a pesquisa preenche uma lacuna na literatura ao quantificar rigorosamente o "custo da integridade". Ao invés de se limitar a afirmações qualitativas sobre o desempenho inferior de *blockchains*, este trabalho fornece métricas precisas, como o aumento de 75 vezes no uso de CPU e a redução de aproximadamente 65% no *throughput* em cenári os de alta carga, permitindo que arquitetos de software tomem decisões baseadas em dados concretos. A validação da hipótese de que a arquitetura híbrida atua como um "seguro digital", onde o custo operacional adicional é justificado pela garantia matemática de não-repúdio, oferece um novo modelo mental para a adoção dessa tecnologia em ambientes regulados.

No âmbito técnico, a principal contribuição é o desenvolvimento e a disponibilização de um artefato de software robusto e funcional: o *tcc-log-management*. O protótipo implementa uma arquitetura híbrida completa, integrando uma API de alta performance desenvolvida em Go (utilizando o *framework* Gin), um banco de dados NoSQL (MongoDB) para armazenamento *off-chain* e uma rede Hyperledger Fabric para ancoragem de provas de integridade. A implementação validou com sucesso o uso de Árvores de Merkle para o agrupamento de logs, uma estratégia que permitiu contornar as limitações de vazão de transações típicas de redes *blockchain*, transferindo o gargalo do consenso para a capacidade computacional da CPU, o que é mais facilmente escalável verticalmente.

Adicionalmente, o trabalho contribui com uma metodologia experimental reprodutível. A infraestrutura de testes, totalmente baseada em contêineres Docker e orquestrada via Docker Compose, juntamente com as ferramentas de *benchmarking* desenvolvidas sob medida, foi disponibilizada publicamente em repositório aberto[1]. Isso permite que outros pesquisadores repliquem os experimentos, validem os resultados encontrados e utilizem a base de código como ponto de partida para futuras investigações sobre auditoria descentralizada e sistemas de logs imutáveis.

7.3 LIMITAÇÕES E TRABALHOS FUTUROS

Apesar dos resultados promissores, é fundamental reconhecer as limitações inerentes ao escopo e à metodologia deste trabalho. A principal restrição refere-se ao ambiente experimental: os testes foram conduzidos em uma infraestrutura virtualizada em uma única máquina física. Embora o uso de contêineres Docker garanta o isolamento de processos, ele não captura a latência de rede real e a imprevisibilidade de conexões *Wide Area Network* (Rede de Longa Distância) (WAN) que caracterizam um sistema distribuído geograficamente. Portanto, os resultados de latência e *throughput* representam um cenário idealizado ("*best-case scenario*"), e

[1] <https://github.com/RicardoMBregalda/tcc-log-management>

a degradação de desempenho em uma implantação real entre diferentes *data centers* poderia ser mais acentuada devido aos atrasos de propagação do consenso Raft. Além disso, a carga de trabalho utilizada foi sintética e constante, o que pode não refletir perfeitamente os padrões de tráfego de rajada (*bursty*) típicos de ambientes corporativos reais.

Outra limitação reside no foco exclusivo na plataforma Hyperledger Fabric. Embora seja a líder de mercado para soluções corporativas, os resultados obtidos não podem ser generalizados automaticamente para outras tecnologias de registro distribuído (DLT), como R3 Corda ou Quorum, que possuem mecanismos de consenso e modelos de dados distintos. Da mesma forma, a análise de custos apresentada é uma estimativa baseada no consumo de recursos computacionais brutos, não considerando variáveis complexas de precificação de provedores de nuvem, como custos de transferência de dados (egress), armazenamento de longo prazo e custos de licenciamento ou suporte de soluções gerenciadas de *blockchain*.

Com base nessas lacunas, diversas direções para trabalhos futuros se apresentam. Uma linha de investigação promissora é a otimização do desempenho criptográfico através do uso de instruções de hardware especializadas, como Intel SGX ou AES-NI, para reduzir o impacto do cálculo de *hashes* na CPU, que se provou ser o principal gargalo da arquitetura híbrida. No campo da privacidade, a integração de provas de conhecimento zero (*Zero-Knowledge Proofs* - ZKP) seria um avanço significativo, permitindo a validação da integridade dos logs sem revelar seu conteúdo, garantindo conformidade estrita com regulamentações de proteção de dados como a LGPD e a GDPR. Por fim, recomenda-se a validação da arquitetura em um ambiente de nuvem híbrida real, com nós distribuídos em múltiplas regiões geográficas, e a exploração de técnicas de *sharding* para avaliar a escalabilidade horizontal do sistema em cenários de "Big Data".

REFERÊNCIAS

Amazon Web Services. **AWS Pricing Calculator**. 2025. Acesso em: out. 2025. Disponível em: <https://calculator.aws>.

ANDERSON, R. **Security Engineering: A Guide to Building Dependable Distributed Systems**. 3rd. ed. [S.l.]: Wiley, 2020. ISBN 978-1119642787.

ANDROULAKI, E. *et al.* Hyperledger fabric: a distributed operating system for permissioned blockchains. p. 1–15, 2018.

ANTONOPOULOS, A. M. **Mastering Bitcoin: Programming the Open Blockchain**. 2nd. ed. [S.l.]: O'Reilly Media, 2017. ISBN 978-1491954386.

BARROSO, L. A.; CLIDARAS, J.; HöLZLE, U. **The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines**. 2nd. ed. [S.l.]: Morgan & Claypool, 2013.

BRADSHAW, S.; BRAZIL, E.; PERKINS, K. **MongoDB: The Definitive Guide**. 3rd. ed. [S.l.]: O'Reilly Media, 2019. ISBN 978-1491954461.

Brasil. **Lei Geral de Proteção de Dados Pessoais (LGPD)**. 2018. Lei nº 13.709, de 14 de agosto de 2018. Acesso em 20 out. 2025. Disponível em: <http://www.planalto.gov.br/ccivil_03/_ato2015-2018/2018/lei/l13709.htm>.

CASEY, M. J.; VIGNA, P. **The Truth Machine: The Blockchain and the Future of Everything**. New York, NY: St. Martin's Press, 2018. ISBN 978-1250114570.

CHRISTIDIS, K.; DEVETSIKIOTIS, M. Blockchains and smart contracts: A paradigm shift for records management. **IEEE Access**, v. 4, p. 2292–2303, 2016.

DAVOUDIAN, A.; CHEN, L.; LIU, M. A survey on nosql stores. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 51, n. 2, p. 1–43, 2018.

DINH, T. T. A. *et al.* Untangling blockchain: A data processing view of blockchain systems. **IEEE Transactions on Knowledge and Data Engineering**, IEEE, v. 30, n. 7, p. 1366–1385, 2018.

____. Blockbench: A framework for analyzing private blockchains. In: **Proceedings of the 2017 ACM International Conference on Management of Data**. [S.l.: s.n.], 2017. p. 1085–1100.

Docker, Inc. **Docker Documentation**. 2025. <https://docs.docker.com/>. Acessado em: 2025.

DRESCHER, D. **Blockchain Basics: A Non-Technical Introduction in 25 Steps**. Berkeley, CA: Apress, 2017. ISBN 978-1484226032.

GATES, M. Blockchain: Ultimate guide to understanding blockchain, bitcoin, cryptocurrencies, smart contracts and the future of money. Independently published, 2017.

Google Cloud. **Google Cloud Pricing Calculator**. 2025. Acesso em: out. 2025. Disponível em: <https://cloud.google.com/products/calculator>.

HE, S. *et al.* A survey on automated log analysis for reliability engineering. **ACM Computing Surveys**, ACM, v. 54, n. 6, p. 1–36, 2021.

Joint Task Force. **Security and Privacy Controls for Information Systems and Organizations**. Gaithersburg, MD, 2020. (NIST Special Publication, 800-53 Revision 5).

KHAN, A. *et al.* An auditable blockchain-based log management system for cloud computing. **Journal of Cloud Computing**, SpringerOpen, v. 9, n. 1, p. 1–18, 2020.

KLEPPMANN, M. **Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems**. Sebastopol, CA: O'Reilly Media, 2017.

KUMAR, R. *et al.* Blockchain-federated-learning and deep learning models for covid-19 detection using ct imaging. **IEEE Sensors Journal**, IEEE, v. 21, n. 14, p. 16301–16314, 2021.

LI, W. *et al.* A secure and efficient log storage and query framework based on blockchain. **Computer Networks**, p. 110683, 2024. Available online 30 July 2024. Disponível em: <https://doi.org/10.1016/j.comnet.2024.110683>.

MARINESCU, D. C. **Cloud Computing: Theory and Practice**. 2. ed. [S.l.]: Morgan Kaufmann, 2017.

Microsoft Azure. **Azure Pricing Calculator**. 2025. Acesso em: out. 2025. Disponível em: <https://azure.microsoft.com/pricing/calculator>.

MOHAN, C. *et al.* Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. **ACM Transactions on Database Systems (TODS)**, ACM New York, NY, USA, v. 17, n. 1, p. 94–162, 1992.

MOHSIN, M.; JABEEN, F.; KHAN, A. H. Performance analysis of hyperledger fabric in high throughput applications. **IEEE Access**, IEEE, v. 8, p. 112005–112015, 2020.

MOUBARAK, J.; FILIOL, E.; CHAMOUN, M. Blockchain-based logging for distributed systems: A systematic literature review. **IEEE Access**, IEEE, v. 9, p. 36815–36833, 2021.

MURTHY, V. Search and indexing in log data. **ACM SIGMOD Record**, v. 44, n. 3, p. 102–106, 2015.

NARAYANAN, A. *et al.* **Bitcoin and cryptocurrency technologies: A comprehensive introduction**. [S.l.]: Princeton University Press, 2016.

NEMETH, E. *et al.* **Unix and Linux System Administration Handbook**. 5. ed. Boston, MA: Addison-Wesley Professional, 2017. 1232 p.

OBE, R. O.; HSU, L. S. **PostgreSQL: Up and Running**. 3rd. ed. [S.l.]: O'Reilly Media, 2017. ISBN 978-1491963296.

ONGARO, D.; OUSTERHOUT, J. In search of an understandable consensus algorithm. In: **2014 USENIX Annual Technical Conference (USENIX ATC 14)**. [S.l.: s.n.], 2014. p. 305–319.

POURMAJIDI, W.; MIRANSKYY, A. **Logchain: Blockchain-assisted Log Storage**. 2018. ArXiv preprint arXiv:1805.08868.

ROUSE, M. Cloud logging services. **ITPro**, 2020. Acesso em: 27 abr. 2025.

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. **Operating System Concepts**. 10. ed. Hoboken, NJ: Wiley, 2018. 1120 p.

SOMMERVILLE, I. **Software Engineering**. 10. ed. Harlow, England: Pearson Education, 2016.

STALLINGS, W.; BROWN, L. **Computer Security: Principles and Practice**. 3. ed. Boston, MA: Pearson, 2018. 1008 p.

STEEN, M. van; TANENBAUM, A. S. **Distributed Systems**. 4. ed. Leiden, The Netherlands: Maarten van Steen, 2023. ISBN 978-90-815406-3-6. Disponível em: <https://www.distributed-systems.net/>.

SWAN, M. **Blockchain: Blueprint for a New Economy**. Sebastopol, CA: O'Reilly Media, 2015. ISBN 978-1491920497.

THAKKAR, P.; NATHAN, S.; VISWANATHAN, B. Performance analysis of hyperledger fabric. In: IEEE. **2018 IEEE 26th International Conference on Network Protocols (ICNP)**. [S.l.], 2018. p. 1–6.

WANG, S. *et al.* Blockchain-based data management and analytics for micro-grid systems. **IEEE Access**, IEEE, v. 8, p. 38437–38458, 2020.

WOOD, G. **Ethereum: A secure decentralised generalised transaction ledger**. [S.l.], 2014. Acesso em 20 out. 2025. Disponível em: <https://ethereum.github.io/yellowpaper/paper.pdf>.

ZHENG, Z. *et al.* An overview of blockchain technology: Architecture, consensus, and future trends. In: **Proceedings of the 2017 IEEE International Congress on Big Data (BigData Congress)**. [s.n.], 2017. p. 557–564. Disponível em: <https://doi.org/10.1109/BigDataCongress.2017.85>.

APÊNDICE A – DECLARAÇÃO DE USO DE INTELIGÊNCIA ARTIFICIAL

Durante a preparação deste trabalho, o autor utilizou a ferramenta de Inteligência Artificial generativa Google Gemini Versão 2.5 Pro[1] para auxílio na revisão, coesão textual, sugestões de reestruturação e formatação de código.

Após o uso desta ferramenta, o autor revisou e editou todo o conteúdo em conformidade com o método científico e assume total responsabilidade pela autoria, precisão e integridade do conteúdo da publicação.

[1] https://gemini.google.com/