UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Felipe Backes Kettl

Defigium: Implementação de Gerador de Carga Baseado em Logs para Benchmarking Personalizado

Florianópolis
2025

Felipe Backes Kettl

Defigium: Implementação de Gerador de Carga Baseado em Logs para Benchmarking Personalizado

Trabalho de Conclusão de Curso submetido ao Curso de Graduação em Ciência da Computação do Departamento de Informática e Estatística da Universidade Federal de Santa Catarina como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.
Orientador: Prof. Odorico Machado Mendizabal, Dr.
Coorientador: Prof. Marcos André Braz Vaz, Dr.

Florianópolis
2025

Ficha catalográfica gerada por meio de sistema automatizado gerenciado pela BU/UFSC.
Dados inseridos pelo próprio autor.

Backes Kettl, Felipe
Defigium: Implementação de Gerador de Carga Baseado em Logs para Benchmarking Personalizado / Felipe Backes Kettl ; orientador, Odorico Machado Mendizabal, coorientador, Marcos André Braz Vaz, 2025.
59 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Ciências da Computação, Florianópolis, 2025.

Inclui referências.

1. Ciências da Computação. 2. Avaliação de Desempenho. 3. Sistemas Distribuídos. 4. Geração de Workloads. I. Machado Mendizabal, Odorico. II. Braz Vaz, Marcos André. III. Universidade Federal de Santa Catarina. Graduação em Ciências da Computação. IV. Título.

Felipe Backes Kettl

Defigium: Implementação de Gerador de Carga Baseado em Logs para Benchmarking Personalizado

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do título de Bacharel em Ciência da Computação e aprovado em sua forma final pelo Curso de Graduação em Ciência da Computação.

Florianópolis, 01 de Dezembro de 2025.

Banca Examinadora:

Prof. Odorico Machado Mendizabal, Dr.
Orientador
Universidade Federal de Santa Catarina

Prof. Marcos André Braz Vaz, Dr.
Coorientador
Universidade Federal de Santa Catarina

Prof. Eduardo Camilo Inacio, Dr.
Universidade Federal de Santa Catarina

Profa. Simone Silmara Werner, Dra.
Universidade Federal de Santa Catarina

Florianópolis, 2025.

AGRADECIMENTOS

Agradeço primeiramente aos meus pais, Marcos Roberto Kettl e Maristela Elaine Backes Kettl, pelo apoio incondicional, pelo incentivo aos estudos desde o início e por todo o sacrifício que fizeram para que eu pudesse chegar até aqui. Sem a base de vocês, nada disso seria possível.

Ao meu orientador, Prof. Dr. Odorico Machado Mendizabal, e ao meu coorientador, Prof. Dr. Marcos André Braz Vaz, pela orientação segura, pela paciência nas revisões e, principalmente, pela liberdade que me deram para explorar a arquitetura do Defigium.

Aos professores do Departamento de Informática e Estatística (INE) da UFSC, que contribuíram para minha formação sólida em Ciência da Computação.

Aos meus amigos e colegas de curso, em especial o pessoal da trupe, pelas trocas de conhecimento, pelos momentos de descontração e pelo companheirismo durante a graduação.

À Universidade Federal de Santa Catarina, pela excelência no ensino e pela infraestrutura proporcionada.

RESUMO

A avaliação de desempenho de sistemas distribuídos requer cargas de trabalho realistas, uma vez que *benchmarks* padronizadas frequentemente falham em capturar a complexidade temporal e os padrões de acesso de ambientes de produção. O objetivo deste trabalho é desenvolver o Defigium, uma ferramenta modular para a geração de cargas de *benchmark* personalizadas fundamentadas na análise de rastros reais. A arquitetura proposta implementa um *pipeline* de três estágios (Análise, Geração e Execução), utilizando uma abordagem híbrida para aliar flexibilidade na modelagem probabilística à precisão temporal na injeção de carga. A validação experimental, realizada sobre um rastro do *benchmark* YCSB em um banco Redis, demonstrou que a estratégia de geração baseada em modelo probabilístico, *Heatmap*, replicou a composição de comandos e o ritmo temporal com alta fidelidade estatística, apresentando um V de Cramér inferior a 0.05, tamanho de efeito trivial. Além disso, o modelo preservou a distribuição de popularidade dos dados e a estrutura de rajadas da carga original. Os resultados indicam que o Defigium é eficaz para criar cenários de teste fidedignos, contribuindo com uma plataforma extensível para a engenharia de desempenho de sistemas distribuídos.

Palavras-chave: *Benchmark*; Geração de *Workloads*; Caracterização de Carga; Sistemas Distribuídos; Avaliação de Desempenho.

ABSTRACT

Performance evaluation of distributed systems requires realistic workloads, as standardized benchmarks often fail to capture the temporal complexity and access patterns of production environments. The objective of this work is to develop Defigium, a modular tool for generating customized benchmark workloads grounded in the analysis of real traces. The proposed architecture implements a three-stage pipeline (Analysis, Generation, and Execution), utilizing a hybrid approach to combine flexibility in probabilistic modeling with temporal precision in load injection. Experimental validation, performed on a YCSB benchmark trace against a Redis database, demonstrated that the generation strategy based on a probabilistic model, Heatmap, replicated the command composition and temporal rhythm with high statistical fidelity, presenting a Cramér's V lower than 0.05, trivial effect size. Furthermore, the model preserved the data popularity distribution and the burst structure of the original workload. Results indicate that Defigium is effective in creating faithful test scenarios, contributing an extensible platform for distributed systems performance engineering.

Keywords: Benchmark; Workload Generation; Workload Characterization; Distributed Systems; Performance Evaluation.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE LISTAGENS

LISTA DE ABREVIATURAS E SIGLAS

CAP	*Consistency, Availability, and Partition Tolerance* (Consistência, Disponibilidade e Tolerância a Partições)
CDF	*Cumulative Distribution Function* (Função de Distribuição Cumulativa)
CPU	*Central Processing Unit* (Unidade Central de Processamento)
E/S	Entrada e Saída
ECDF	*Empirical Cumulative Distribution Function* (Função de Distribuição Cumulativa Empírica)
EOS	*Exascale Open Storage* (Armazenamento Aberto em Exaescala)
FEI	Formato de Evento Intermediário
GARCH	*Generalized Autoregressive Conditional Heteroskedasticity* (Modelo Autorregressivo Condicional Heterocedástico Generalizado)
GAS	*Generalized Autoregressive Score* (Modelo de Score Autorregressivo Generalizado)
GPU	*Graphics Processing Unit* (Unidade de Processamento Gráfico)
HPC	*High Performance Computing* (Computação de Alto Desempenho)
HTTP	*Hypertext Transfer Protocol*
LBN	*Logical Block Number* (Número de Bloco Lógico)
LLM	*Large Language Model* (Modelo de Linguagem Grande)
LTTng	*Linux Trace Toolkit next generation*
OLAP	*Online Analytical Processing* (Processamento Analítico Online)
OLTP	*Online Transaction Processing* (Processamento de Transações Online)
ULA	*Arithmetic Logic Unit* (Unidade Lógica e Aritmética)
VM	*Virtual Machine* (Máquina Virtual)
YCSB	*Yahoo! Cloud Serving Benchmark*

SUMÁRIO

1 INTRODUÇÃO

A avaliação e otimização de sistemas distribuídos são aspectos fundamentais para compreender e aprimorar o comportamento de serviços e aplicações em ambientes operacionais reais. Ferramentas de *benchmark*, como o Yahoo! Cloud Serving Benchmark (YCSB) (COOPER *et al.*, 2010), consolidaram-se como uma base padronizada para a medição de serviços de computação em nuvem. Esses *benchmarks* permitem a comparação de diferentes configurações e cenários de uso, facilitando a identificação de gargalos e o aprimoramento do desempenho geral do sistema.

No entanto, as abordagens tradicionais de geração de carga de trabalho, que frequentemente utilizam perfis pré-definidos, não conseguem captar de forma eficaz a complexidade e a variabilidade dos padrões de uso observados em sistemas mais complexos. Atikoglu *et al.* (2012) demonstraram que, ao analisar detalhadamente as cargas de trabalho em ambientes como o Facebook, observa-se que *benchmarks* genéricos não refletem completamente os comportamentos reais dessas aplicações. Essa constatação ressalta a importância de basear as avaliações de desempenho em dados reais de operação.

Contribuições mais recentes, como a proposta de Ghandeharizadeh e Huang (2018), evidenciam o potencial de ferramentas modulares e expansíveis para a geração de carga de trabalho, utilizando especificações publicadas, como as distribuições estatísticas de acesso e os parâmetros de cargas de trabalho descritos em artigos científicos, para replicar cenários sem a necessidade de acesso a rastros (registros sequenciais de eventos ou operações de um sistema) privados. De forma análoga, a abordagem de Kim *et al.* (2014) enfatiza a importância dos perfis estatísticos detalhados na criação de cargas de trabalho sintéticas, que no caso do trabalho são perfis de execução microarquitetônicos coletados a partir dos contadores de hardware do processador (*e.g.*, número de *cache misses* e de instruções de memória), para gerar uma carga que reproduza com fidelidade o mesmo estresse sobre o sistema de hardware.

Além disso, estudos como o de Sladojević *et al.* (2022) destacam a relevância de incorporar características específicas do domínio – como latência e variação de carga em contextos financeiros de alto volume – para adaptar *benchmarks* aos padrões de uso reais. Tal perspectiva reforça que, embora *benchmarks* padronizados sejam essenciais para a comparação de sistemas em cenários controlados, a modelagem de cargas realistas é crucial para contextos mais específicos.

Neste contexto, o presente trabalho detalha o projeto e a implementação da ferramenta Defigium[1], uma solução para a geração de cargas de *benchmark*, funda-

[1] O nome Defigium é uma contração dos termos latinos *Definio*, que significa “eu defino”, e *Vestigium*, que significa “rastro” ou “vestígio”. O nome busca refletir a função da ferramenta em definir as características de uma carga de trabalho a partir de seu rastro original para, subsequentemente, gerar ou executar cargas sintéticas baseadas nessa definição.

mentada na análise de *logs* reais e estruturada em uma arquitetura modular. Essa abordagem foi projetada para ser altamente extensível, permitindo a integração de diferentes módulos para cada etapa do processo, desde o processamento de formatos de *log* distintos até a aplicação de variadas técnicas de caracterização e síntese. Dessa forma, a ferramenta poderá suportar, por exemplo, desde a análise de cargas de trabalho de requisições HTTP até de consultas a bancos de dados, e empregar métodos de geração que variam da amostragem de distribuições estatísticas à modelagem por Cadeias de Markov. O objetivo é, portanto, oferecer uma plataforma flexível que contribua para a criação de cenários de teste personalizados e de alta fidelidade, alinhados às características reais de operação de um sistema, viabilizando avaliações de desempenho precisas.

1.1 OBJETIVOS

1.1.1 Objetivo Geral

O objetivo deste trabalho é desenvolver uma ferramenta modular para a geração de cargas de *benchmark*, fundamentada na análise e caracterização de rastros reais, a fim de proporcionar análises de desempenho mais precisas e alinhadas às características de sistemas distribuídos em ambientes reais.

1.1.2 Objetivos Específicos

1. Realizar um levantamento de abordagens existentes para geração de cargas de *benchmark*, identificando suas limitações e pontos fortes.
2. Analisar e processar rastros reais, extraindo características relevantes para a modelagem de cargas de trabalho.
3. Implementar um modelo de geração baseado em distribuições de probabilidade derivadas das características do rastro.
4. Projetar e implementar uma arquitetura modular para a ferramenta, permitindo a incorporação de diferentes tipos de sistemas.
5. Validar a ferramenta por meio de experimentos que comparem a fidelidade da carga gerada em relação ao rastro original de alguma aplicação de armazenamento chave-valor.
6. Documentar os resultados e realizar uma análise crítica sobre a fidelidade e as limitações da ferramenta desenvolvida.

1.2 ORGANIZAÇÃO DO TRABALHO

Este documento está organizado da seguinte forma: O Capítulo 2 estabelece os conceitos teóricos que fundamentam esta pesquisa, abordando sistemas distribuídos, cargas de trabalho e o papel dos *benchmarks*. O Capítulo 3 revisa a literatura, apresentando uma análise das principais metodologias de caracterização e geração de cargas de trabalho propostas por outros pesquisadores. O Capítulo 4 detalha a concepção e a arquitetura modular da ferramenta Defigium. O Capítulo 5 descreve a implementação bilíngue (Python e C++) da ferramenta. O Capítulo 6 apresenta a metodologia experimental utilizada e analisa os resultados obtidos na validação da fidelidade da ferramenta. Finalmente, o Capítulo 7 apresenta as conclusões do trabalho, consolidando os resultados e discutindo os próximos passos e trabalhos futuros.

2 FUNDAMENTOS

A avaliação de desempenho de sistemas computacionais fundamenta-se em três conceitos interdependentes: a definição da arquitetura do sistema sob análise, a caracterização da demanda a ele imposta e a metodologia de medição. Este capítulo detalha esses pilares em uma sequência lógica, iniciando-se com os sistemas distribuídos para definir o ambiente de estudo. Em seguida, é apresentado o conceito de carga de Trabalho para formalizar a natureza das requisições e do uso desses sistemas. Por fim, são introduzidos os *benchmarks* como os instrumentos formais para quantificar e comparar o desempenho de forma controlada.

2.1 SISTEMAS DISTRIBUÍDOS

No panorama computacional contemporâneo, a demanda por maior escalabilidade, disponibilidade e resiliência levou ao surgimento e à proliferação dos sistemas distribuídos. De forma geral, um sistema distribuído é uma coleção de computadores autônomos, interconectados por uma rede, que cooperam para que suas capacidades de hardware e software sejam utilizadas em conjunto para realizar uma tarefa ou serviço comum, operando de forma coerente para o usuário final (TANENBAUM; STEEN, 2016). Essa arquitetura contrasta com os sistemas centralizados, onde todos os componentes residem em uma única máquina.

As principais características que definem um sistema distribuído incluem a concorrência (múltiplas operações ocorrendo simultaneamente), a ausência de um relógio global (cada nó tem seu próprio relógio, exigindo mecanismos de sincronização) e a independência de falhas (a falha de um componente não necessariamente derruba o sistema inteiro) (COULOURIS *et al.*, 2011). A escalabilidade, a capacidade de lidar com um volume crescente de trabalho adicionando recursos, é frequentemente um objetivo primário de sistemas distribuídos, permitindo que eles cresçam para atender às demandas de cargas de trabalho dinâmicas e de grande escala.

Apesar de suas vantagens, sistemas distribuídos introduzem desafios inerentes. A coordenação entre múltiplos nós pode ser complexa, especialmente para garantir a consistência de dados em um ambiente onde as cópias dos dados podem existir em diferentes locais. O Teorema CAP (do inglês *Consistency, Availability, and Partition Tolerance*) destaca que, na presença de uma partição de rede, um sistema distribuído deve escolher entre manter a consistência ou a disponibilidade. Essa realidade força uma compensação entre essas propriedades fundamentais, visto que falhas de rede são inevitáveis em ambientes distribuídos (BREWER, 2012). Além disso, a gerência de falhas e a comunicação em rede (latência, perda de pacotes) são considerações críticas, pois a ocorrência de falhas parciais é a norma em vez da exceção nesses ambientes.

A compreensão das características e dos desafios inerentes a esses sistemas é, portanto, fundamental. Para de fato avaliar e otimizar o desempenho dessas arquiteturas complexas, precisamos de ferramentas que consigam simular seu uso real e medir suas capacidades. Nesse ponto, a carga de trabalho emerge como o conceito que representa o comportamento e a demanda sobre o sistema, enquanto os *benchmarks* se tornam os instrumentos essenciais para quantificar e comparar o desempenho sob condições controladas.

2.2 CARGA DE TRABALHO

Sendo assim, a definição de carga de trabalho é fundamental para que possamos modelar, gerar e avaliar cenários de desempenho. De forma geral, o termo carga de trabalho refere-se a todas as entradas recebidas por uma dada infraestrutura tecnológica. A compreensão das propriedades e do comportamento das cargas de trabalho é fundamental para estudos de engenharia de desempenho, planejamento de capacidade e otimização de custos (CALZAROSSA; MASSARI; TESSERA, 2016).

Em diversos ambientes computacionais, a carga de trabalho se manifesta de maneiras distintas, refletindo as características do sistema e das aplicações. Em ambientes de *data center*, ela é descrita como um conjunto de atividades ou tarefas que consomem recursos como CPU, memória, disco e rede ao longo do tempo (GMACH *et al.*, 2007). Complementando, Delimitrou e Kozyrakis (2011) enfatizam que, nesse contexto, padrões de E/S, incluindo operações de leitura, escrita e metadados, definem a demanda sobre sistemas de arquivos e armazenamento. Para aplicações web, a carga de trabalho é frequentemente definida pelo número de requisições de usuários ou aplicações e caracterizada pela taxa de acertos, tamanho e distribuição de diferentes tipos de conteúdo, como imagens, texto, áudio e vídeo (PALLIS; KAZAKOS; DASKALOU, 2003; AGHILI *et al.*, 2024).

Já em computação de alto desempenho (HPC, do inglês *High Performance Computing*), a carga de trabalho é tipicamente compreendida como uma coleção de tarefas ou *jobs* computacionais que exigem grandes volumes de recursos, como CPU, memória, E/S e rede, frequentemente executadas em paralelo através de múltiplos nós (SIMAKOV *et al.*, 2018). Atributos cruciais para definir uma carga de trabalho HPC incluem seu tempo de execução e requisitos específicos de recursos, como intensividade de CPU ou GPU (CHU *et al.*, 2024).

No contexto de bancos de dados, a carga de trabalho refere-se ao conjunto de operações que o sistema de gerenciamento de banco de dados processa, incluindo consultas, inserções, atualizações e exclusões. Cargas de trabalho são comumente categorizadas como Processamento de Transações Online (OLTP) ou Processamento Analítico Online (OLAP). As cargas OLTP caracterizam-se por um alto volume de transações curtas, que acessam e modificam pequenas porções de dados, exigindo alta

vazão, consistência e baixa latência por operação, sendo tipicamente dominadas por muitos clientes concorrentes e operações parametrizadas (REHRMANN *et al.*, 2020). Em contraste, as cargas OLAP envolvem consultas complexas e de longa duração sobre grandes volumes de dados históricos, priorizando a leitura e demandando alto *throughput* de dados para análises. Elas são intensivas em operações aritméticas e largura de banda, processando dados com padrões de acesso variados, sequenciais e aleatórios (SIRIN; AILAMAKI, 2020).

Em suma, a carga de trabalho é um conceito multifacetado que se adapta ao contexto tecnológico em questão, descrevendo os padrões de acesso, operações e a demanda por recursos de um sistema. Seja em um *data center*, em sistemas de alta performance, ou em bancos de dados, a compreensão de suas características, como volume de operações, intensidade de recursos, padrões temporais e natureza das interações, é essencial para reconhecer qual é a carga de trabalho do sistema.

2.3 *BENCHMARK*

Benchmark é uma ferramenta ou processo padronizado empregado para medir e comparar o desempenho de hardware ou software sob condições controladas e reproduzíveis (JAIN, 1991). Seu principal objetivo é fornecer uma base objetiva para avaliar a eficiência, a capacidade e a escalabilidade de um sistema, permitindo que diferentes configurações ou implementações sejam comparadas de maneira justa e consistente (RAMAKRISHNAN; GEHRKE, 2000).

Um *benchmark* é um processo de avaliação mais abrangente que um teste de desempenho pontual, constituindo um conjunto estruturado de testes, regras e métricas cuidadosamente projetados para simular uma carga de trabalho específica ou representativa. Candidatos a *benchmark* devem passar por um processo que inclui a definição de metodologias de medição, seleção de carga de trabalho e testes de aceitação rigorosos (KISTOWSKI *et al.*, 2015). As métricas coletadas em um *benchmark* podem incluir vazão (número de operações por unidade de tempo), latência (tempo para completar uma operação), utilização de recursos (CPU, memória, E/S, rede) e escalabilidade (BAJABER *et al.*, 2020).

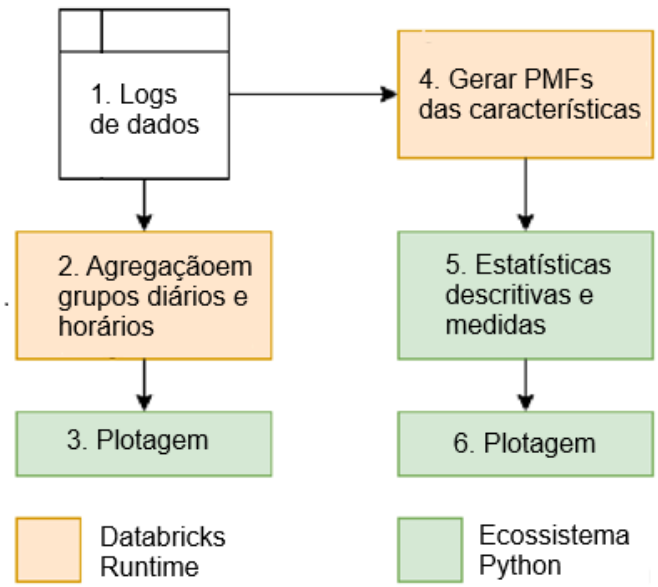
Existem diversos tipos de *benchmark*, cada um com foco em aspectos específicos. Alguns são concebidos para simular cargas de trabalho transacionais (tipicamente OLTP), caracterizadas por inúmeras operações pequenas e concorrentes, como as encontradas em sistemas de comércio eletrônico ou bancários (TPC, 2025a). Outros se concentram em cargas de trabalho analíticas (OLAP), as quais envolvem consultas complexas sobre grandes volumes de dados para processamento de informações e relatórios (TPC, 2025b). Além disso, há *benchmarks* que avaliam subsistemas específicos, como sistemas de arquivos, rede ou componentes de inteligência artificial, cada um com seus próprios padrões de acesso e demandas por recursos.

3 TRABALHOS RELACIONADOS

A geração de *benchmarks* personalizados e de alta fidelidade é um processo que depende da relação simbiótica entre duas atividades: a caracterização, que visa extrair as propriedades fundamentais de uma carga de trabalho real, e a síntese, que busca recriar essas mesmas propriedades em uma carga sintética. Uma não existe de forma útil sem a outra. Este capítulo, portanto, revisa a literatura científica que aborda as diversas metodologias para realizar esse processo de ponta a ponta. A revisão apresentada explora este panorama de metodologias, fluindo das técnicas de análise estatística até os modelos geradores mais complexos, de modo a contextualizar a necessidade de uma ferramenta flexível e a fundamentar as decisões de arquitetura desenvolvidas neste trabalho.

A metodologia de Talluri *et al.* (2019) para caracterizar carga de trabalho de armazenamento do Spark na nuvem envolve um processo de múltiplas etapas sobre um rastro de 6 meses, cujo fluxo de trabalho é ilustrado na Figura 3.1. A análise se inicia com o pré-processamento dos *logs* para anonimização e normalização de métricas. Em seguida, a abordagem aplica um conjunto diverso de ferramentas estatísticas: para analisar as propriedades das operações, são computadas Funções de Distribuição Cumulativa empíricas (ECDFs) e um conjunto de estatísticas descritivas, como média, mediana e coeficiente de variação; para estimar a dependência de longo prazo e o comportamento de rajada, é utilizado o parâmetro de Hurst.

Figura 3.1 – Fluxo de trabalho do processo de caracterização da carga de trabalho.

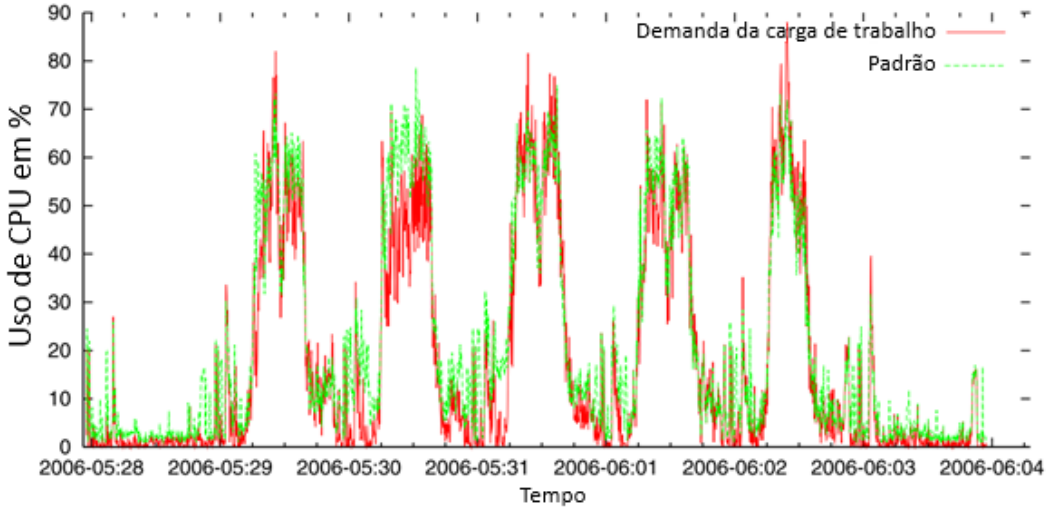


Fonte: Adaptado de Talluri *et al.* (2019).

O trabalho de Gmach *et al.* (2007), por sua vez, foca na caracterização de cargas de trabalho com comportamento periódico em *data centers*. A metodologia

deles utiliza técnicas de análise de séries temporais para extrair um padrão cíclico de um rastro histórico. Primeiramente, para identificar a duração do ciclo principal da carga de trabalho (*e.g.*, diário, semanal), eles empregam a análise do periodograma, obtido via Transformada de Fourier, e o cálculo da função de autocorrelação. Uma vez que o período mais provável é identificado, todas as ocorrências desse ciclo no *trace* histórico são usadas para calcular um "ciclo médio", que se torna o padrão da carga de trabalho e pode ser usado para gerar cargas sintéticas. A Figura 3.2 ilustra um exemplo deste padrão extraído sobreposto a uma semana do rastro original, evidenciando a alta similaridade.

Figura 3.2 – Padrão da carga de trabalho semanal extraída (curva mais suave) sobreposta a um trecho do *trace* original.

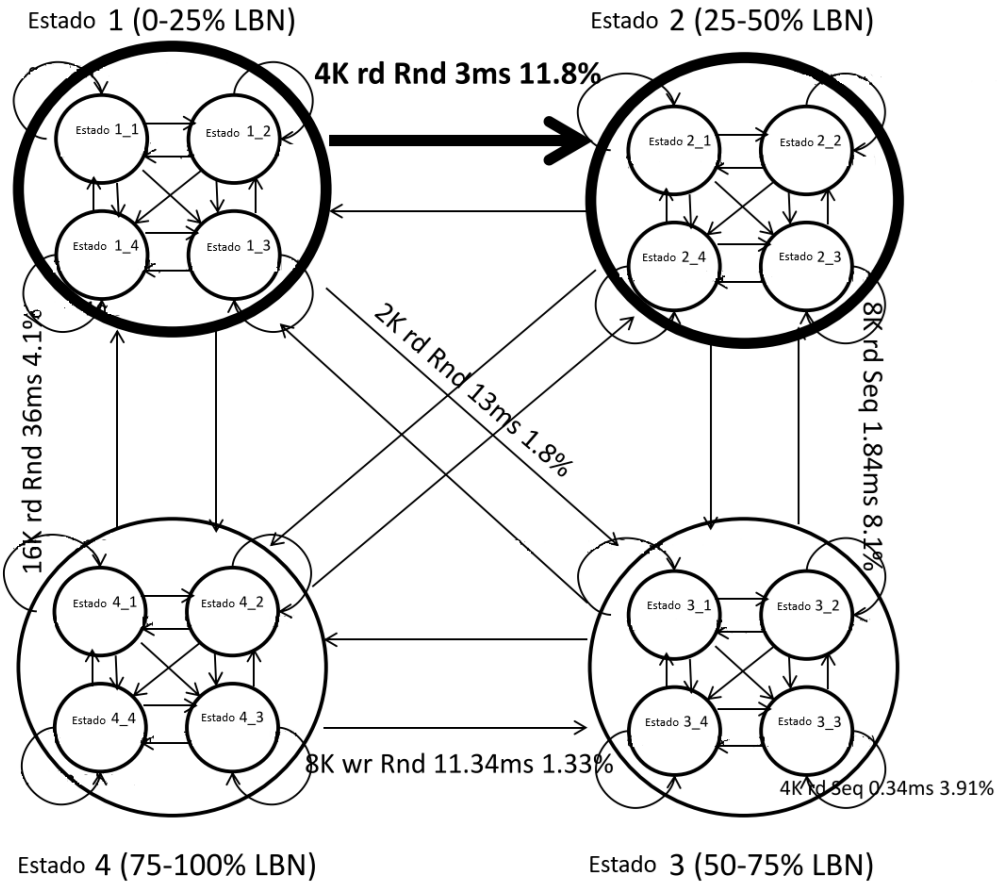


Fonte: Adaptado de Gmach *et al.* (2007).

A abordagem de Delimitrou e Kozyrakis (2011) também foca na caracterização e geração de cargas de trabalho de armazenamento, porém com o objetivo de capturar a localidade espacial e temporal. A técnica se baseia na construção de um modelo de estados hierárquico, fundamentado em Cadeias de Markov, a partir de rastros de E/S, conforme ilustrado na Figura 3.3. Neste modelo, os estados correspondem a faixas de endereços lógicos no disco (LBNs), e as transições entre estados possuem probabilidades e são anotadas com as características da operação de E/S (tamanho do bloco, tipo, aleatoriedade e tempo de chegada).

De forma distinta, Khan *et al.* (2012) propõe uma metodologia para caracterizar e prever cargas de trabalho de múltiplas Máquinas Virtuais (VMs) correlacionadas na nuvem. O processo deles tem dois estágios principais: primeiro, eles aplicam uma técnica de *co-clustering*, ilustrada de forma conceitual na Figura 3.4, sobre uma matriz de dados (VMs vs. tempo) para identificar grupos de VMs que exibem padrões de carga de CPU correlacionados em determinados períodos. Em um segundo momento, um Modelo Oculto de Markov é treinado sobre as sequências de padrões desses grupos

Figura 3.3 – Exemplo de um diagrama de estados hierárquico de dois níveis para modelagem de carga de trabalho de armazenamento.

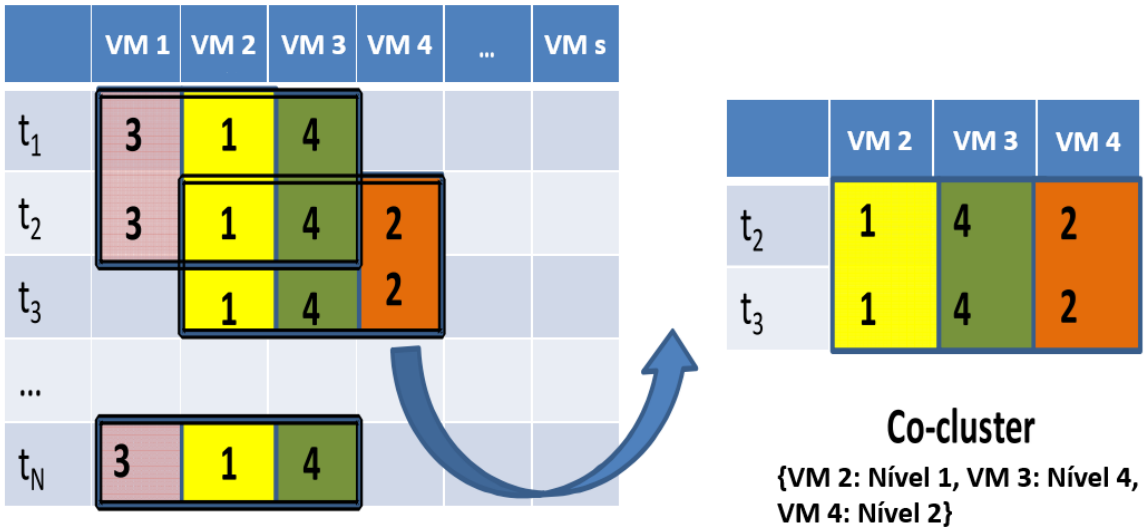


Fonte: Adaptado de Delimitrou e Kozyrakis (2011).

para capturar as transições temporais entre os diferentes estados de carga e permitir a predição do comportamento futuro.

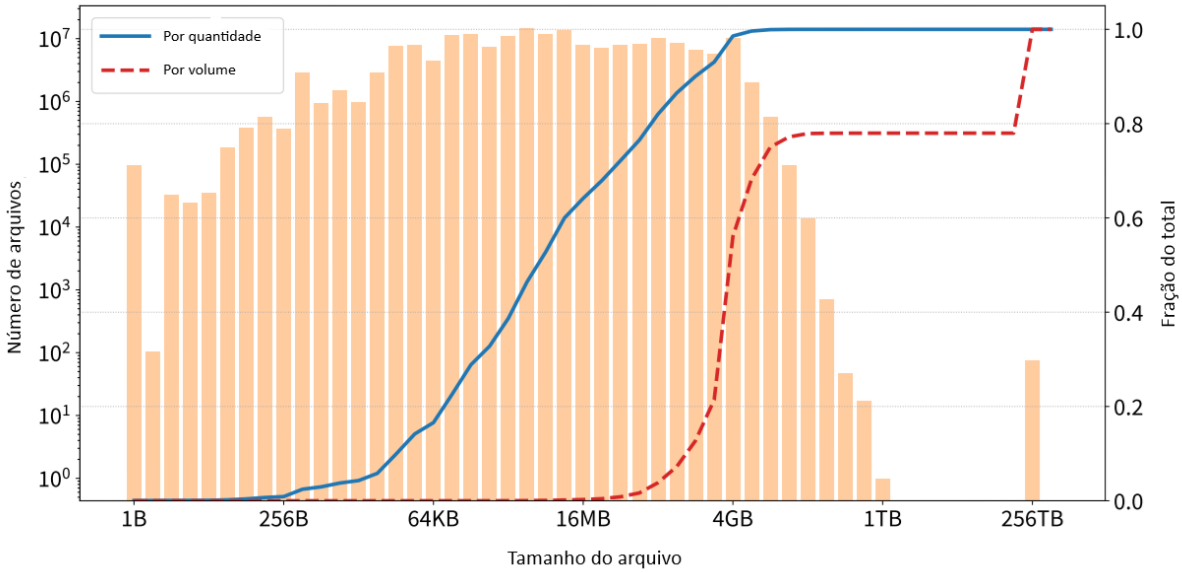
A caracterização da carga de trabalho do sistema de arquivos EOS do CERN, realizada por Purandare, Bittman e Miller (2022), baseia-se na análise de rastros de 11 meses que capturam o estado dos arquivos após cada operação. Uma vez que esses rastros não registram a operação explicitamente, a metodologia infere as ações (criação, leitura, atualização) a partir da análise de mudanças nos metadados dos arquivos, como o tamanho de um arquivo partindo de zero. A análise estatística da distribuição do tamanho dos arquivos é conduzida pela construção de Funções de Distribuição Cumulativa (CDFs), conforme ilustrado na Figura 3.5. O gráfico evidencia a natureza de cauda longa da carga de trabalho, mostrando que, embora a maioria dos arquivos seja pequena (curva por contagem), a maior parte do espaço é consumida por poucos arquivos grandes (curva por volume). Uma técnica central do trabalho é a categorização da atividade de E/S, que é segmentada com base no identificador de usuário e pela aplicação que originou a requisição.

Figura 3.4 – Exemplo conceitual da técnica de *co-clustering* em uma matriz de carga de trabalho (VMs vs. Tempo).



Fonte: Adaptado de Khan *et al.* (2012).

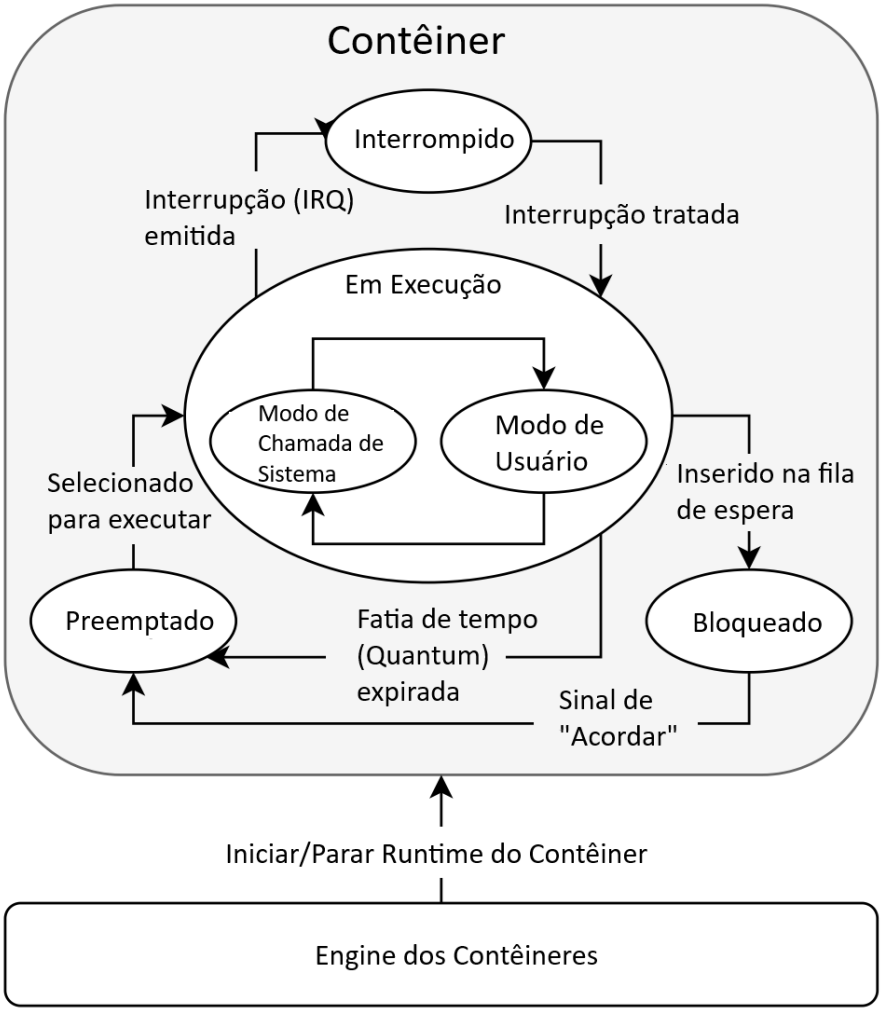
Figura 3.5 – Distribuição cumulativa do tamanho dos arquivos na carga de trabalho do CERN, por contagem e por volume.



Fonte: Adaptado de Purandare, Bittman e Miller (2022).

A metodologia de Janecek, Ezzati-Jivan e Azhari (2021) para caracterizar carga de trabalho de contêineres se baseia no rastreamento do sistema hospedeiro, utilizando `LTTng` para capturar eventos do *kernel* sem a necessidade de agentes internos. A caracterização é feita a partir da análise do tempo que as *threads* do contêiner passam em diferentes estados de execução, como modo de usuário, chamada de sistema e bloqueado, cujo diagrama de transições é apresentado na Figura 3.6. Para focar nos processos mais relevantes, a abordagem emprega um algoritmo baseado em *PageRank* para classificar a importância das *threads* com base em suas interações, identificadas por eventos *sched_wakeup*. Finalmente, uma técnica de clusterização *K-Means* em dois estágios é aplicada para agrupar os contêineres com perfis de execução semelhantes.

Figura 3.6 – Diagrama de transição entre os estados de execução de uma *thread* em nível de *kernel*.



Fonte: Adaptado de Janecek, Ezzati-Jivan e Azhari (2021).

Já em Adegboyega (2024), foca na caracterização da carga de trabalho de armazenamento em nuvem através da análise de séries temporais sobre o rastro do

Alibaba. A metodologia agrega os dados brutos de E/S em uma série temporal de taxa de bits. Para modelar o comportamento de rajada da carga de trabalho, a análise emprega modelos econométricos avançados, como o *GARCH* e os modelos *GAS (Generalized Autoregressive Score)*, que são projetados para capturar a variância que muda ao longo do tempo.

A partir da caracterização de uma carga de trabalho, a literatura avança para as metodologias de síntese. A metodologia de Kim *et al.* (2014) foca na síntese de cargas de trabalho que mimetizem o perfil de execução de baixo nível de uma aplicação. O processo se inicia com a coleta de um perfil estatístico a partir de contadores de desempenho de hardware, que registram eventos como o número de instruções de memória, desvios e operações de ULA, em vez de analisar o fluxo de instruções diretamente. Em seguida, um *solver* matemático determina a frequência de execução de um conjunto de "funções *kernel*" pré-definidas de forma a reproduzir o perfil estatístico da aplicação original.

De forma distinta, Sfakianakis *et al.* (2021) apresentam a ferramenta Tracie, que também parte de um modelo estatístico, mas extraído de rastros de alto nível de *datacenters*. A abordagem consiste em gerar uma sequência de *jobs* e tarefas amostrando as distribuições de probabilidade de variáveis como tempo de chegada e número de tarefas. Para tornar a carga de trabalho executável, a ferramenta, então, seleciona, de um repositório de aplicações de *benchmark* reais, aquelas cujas características microarquitetônicas mais se assemelham às da tarefa original do rastro.

Outras abordagens focam na geração a partir de modelos mais abstratos ou de domínio específico. O trabalho de Ghandeharizadeh e Huang (2018) apresenta a ferramenta Hoagie, um gerador de carga de trabalho que não utiliza rastros, mas especificações publicadas na literatura. A metodologia permite que um usuário defina as características da carga, como a distribuição de popularidade das chaves (e.g., Zipfiana) e a proporção de leituras e escritas, para então gerar tanto a base de dados quanto a sequência de operações correspondente.

Em uma linha de especialização ainda maior, Xiang *et al.* (2025) propõe a ferramenta ServeGen, focada em gerar cargas de trabalho de inferência para Modelos de Linguagem Grandes (LLMs). A metodologia se diferencia por modelar e gerar a carga de trabalho a partir da composição de componentes individuais: o comportamento de múltiplos clientes heterogêneos é caracterizado e modelado separadamente. A carga de trabalho final é então construída através da amostragem e agregação das requisições de um conjunto de clientes sintéticos, extraídos de um *Client Pool* com perfis realistas.

Para facilitar a visualização das diferentes abordagens discutidas, a Tabela 3.1 apresenta um resumo comparativo das metodologias. Os números na coluna Ref. referem-se à lista de referências detalhada abaixo da tabela.

Tabela 3.1 – Comparação Resumida de Metodologias (Agrupada por Domínio e Cronologia).

Ref./Ano	Domínio	Técnica Principal	Foco Distintivo
[1] 2011	Armazenamento (E/S)	Cadeias de Markov Hierárquicas	Modelagem de Localidade Espacial/-Temporal.
[2] 2019	Armazenamento Spark	ECDFs, Est. Descritiva, Hurst	Caracterização estatística detalhada, dependência de longo prazo.
[3] 2024	Armazenamento (Nuvem)	Séries Temporais (GARCH, GAS)	Modelagem de comportamento de rajada (*burst*).
[4] 2022	Sist. Arquivos (EOS)	Inferência de Operações, CDFs	Análise de longa duração, inferência a partir de metadados.
[5] 2007	*Data Center*	Análise de Séries Temporais	Extração de Padrões Periódicos (ciclos).
[6] 2021	*Data Center*	Amostragem + Seleção de *Benchmarks*	Geração executável combinando modelo estatístico e código real.
[7] 2021	Contêineres	Rastreamento de *Kernel* (`LTTng`)	Caracterização via host, análise de estados de *thread*.
[8] 2012	VMs (Nuvem)	*Co-clustering*, Modelo Oculto de Markov	Predição de Cargas Correlacionadas entre VMs.
[9] 2014	CPU/ Microarquitetura	Contadores de Hardware, Solver	Síntese de baixo nível para estresse de hardware.
[10] 2018	BD (Genérico)	Especificações Publicadas	Geração sem rastro original, baseada em literatura.
[11] 2025	Inferência LLM	Modelagem Composta de Clientes	Geração específica para *LLM Serving*, foco em clientes heterogêneos.

Referências da Tabela 3.1: [1] Delimitrou e Kozyrakis (2011); [2] Talluri *et al.* (2019); [3] Adegboyega (2024); [4] Purandare, Bittman e Miller (2022); [5] Gmach *et al.* (2007); [6] Sfakianakis *et al.* (2021); [7] Janecek, Ezzati-Jivan e Azhari (2021); [8] Khan *et al.* (2012); [9] Kim *et al.* (2014); [10] Ghandeharizadeh e Huang (2018); [11] Xiang *et al.* (2025).

A análise dos trabalhos apresentados evidencia que a caracterização e a geração de cargas de trabalho são áreas multifacetadas, com uma vasta gama de metodologias aplicadas a diferentes domínios e objetivos. Fica claro que não existe uma técnica universal; a escolha do método ideal depende intrinsecamente da natureza da carga de trabalho. Essa constatação reforça a principal motivação deste trabalho: a necessidade de uma ferramenta de arquitetura modular e extensível, capaz de implementar e combinar essas diversas técnicas, servindo como uma plataforma para a aplicação das diferentes abordagens já consagradas na literatura.

4 DEFIGIUM: ARQUITETURA PROPOSTA

Este capítulo detalha a arquitetura conceitual da ferramenta proposta, Defigium, que constitui a principal contribuição deste trabalho. A concepção do sistema foi guiada pelos princípios de modularidade e extensibilidade, visando permitir a combinação flexível de diversas técnicas de análise, geração e execução de cargas de trabalho, como as exploradas no Capítulo 3, em componentes autocontidos e intercambiáveis.

4.1 VISÃO GERAL DA ARQUITETURA EM PIPELINE

O Defigium é concebido como um sistema modular estruturado em um *pipeline* de processamento, conforme ilustrado na Figura 4.1. Essa arquitetura divide o processo de geração e execução de carga em estágios lógicos distintos, promovendo o desacoplamento e a especialização das responsabilidades. Para permitir a comunicação padronizada entre esses estágios, utiliza-se internamente um Formato de Evento Intermediário (FEI), que representa os eventos de carga de forma abstrata e genérica, independentemente do sistema de origem ou destino. O fluxo geral do *pipeline* compreende três módulos principais:

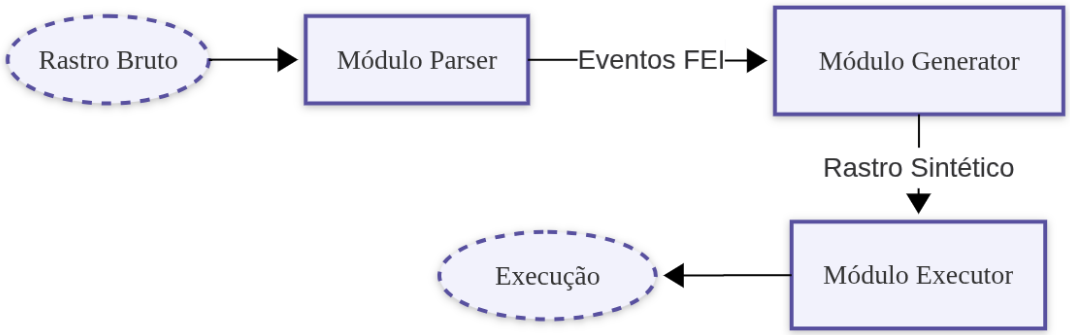
1. Módulo de Análise (*Parser*): Atua como um tradutor bidirecional. Sua primeira função é interpretar rastros brutos de sistemas reais (linhas de *log*), convertendo-os para a representação interna padronizada (FEI). Sua segunda função é formatar os eventos FEI de volta para linhas de *log*, utilizada na construção do rastro sintético final.
2. Módulo de Geração (*Generator*): Componente central que opera sobre a representação FEI dos eventos. Ele aplica uma estratégia selecionada para analisar as características do rastro original (fase de caracterização) e, com base nessas características, gera uma nova sequência de eventos sintéticos no formato FEI (fase de síntese) que mimetiza o comportamento desejado.
3. Módulo de Execução (*Executor*): Consome o rastro sintético, um arquivo de *log* gerado pela fase anterior. Ele interpreta as linhas desse arquivo e as traduz em operações reais contra um sistema-alvo. É responsável por disparar essas interações respeitando a cadência temporal definida no rastro.

A comunicação entre os módulos de Geração e Execução ocorre por meio do arquivo de rastro Sintético. Este arquivo contém a sequência de eventos gerados pelo *Generator* escolhido, formatados textualmente pelo *Parser* selecionado para espelhar o formato esperado pelo sistema-alvo.

A decisão de arquitetar o sistema em estágios desacoplados, comunicando-se através de um rastro sintético persistido em arquivo, oferece três vantagens. Primeiro, promove a independência tecnológica, permitindo que a fase de modelagem explore o

rico ecossistema de bibliotecas estatísticas do Python, enquanto a fase de execução usufrui do desempenho e controle de *hardware* do C++, sem a complexidade de interfaces de ligação direta. Segundo, o arquivo de rastro Sintético atua como um artefato de reprodutibilidade, permitindo que o mesmo cenário de carga seja auditado, compartilhado e re-executado múltiplas vezes sob condições idênticas. Por fim, o uso do FEI como representação interna abstrai a semântica da operação da sua sintaxe, permitindo que a lógica de geração, como distribuição de leituras vs. escritas, seja agnóstica ao sistema-alvo, gerando o *log* final no formato específico apenas no último passo.

Figura 4.1 – Visão geral conceitual do pipeline da ferramenta Defigium.



Fonte: O autor (2025).

4.2 MODULARIDADE E EXTENSIBILIDADE VIA PADRÕES DE PROJETO

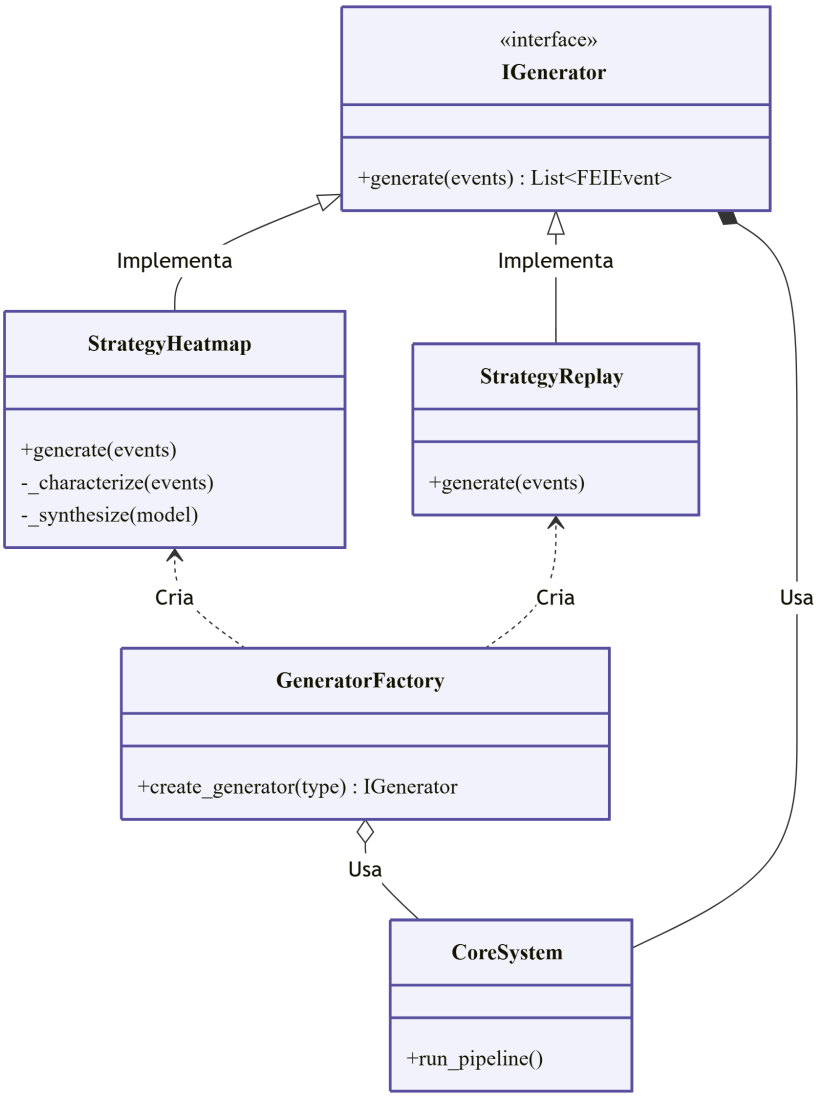
A flexibilidade arquitetural do Defigium é alcançada através da aplicação sistemática dos padrões de projeto *Strategy* e *Factory Method* (GAMMA *et al.*, 1994). Essa abordagem permite que a ferramenta seja facilmente estendida para suportar novos formatos de rastro, algoritmos de geração ou sistemas-alvo sem a necessidade de modificar o fluxo principal de controle.

O padrão *Strategy* é aplicado definindo interfaces abstratas para cada um dos módulos principais: `IParser`, `IGenerator` e `IExecutor`. Essas interfaces estabelecem um contrato comum que todas as implementações concretas devem seguir. Por exemplo, qualquer algoritmo de análise de *log* deve implementar a interface `IParser`, definindo como um rastro específico é convertido para o formato FEI. Da mesma forma, diferentes técnicas de geração de carga (como replicação direta, modelagem estatística, ou Cadeias de Markov) seriam implementadas como estratégias distintas que aderem à interface `IGenerator`.

Para gerenciar a criação e seleção dessas estratégias de forma desacoplada, o padrão *Factory Method* é empregado. Conforme exemplificado na Figura 4.2, fábricas dedicadas, como a `GeneratorFactory`, encapsulam a lógica de instanciação das implementações concretas. O sistema principal (`CoreSystem`) não interage diretamente com

as classes concretas (`StrategyHeatmap`, `StrategyReplay`), mas sim solicita à fábrica, através do método `create_generator(type)`, uma instância da interface `IGenerator`. O `CoreSystem` então utiliza esta instância, que por sua vez implementa o método principal definido pela interface. Isso isola o núcleo da ferramenta das especificidades de cada módulo, permitindo que novas estratégias sejam adicionadas simplesmente registrando-as na fábrica.

Figura 4.2 – Exemplo conceitual da aplicação dos padrões Factory e Strategy no Módulo Gerador.



Fonte: O autor (2025).

4.3 MÓDULO DE ANÁLISE (*PARSER*)

O primeiro estágio do *pipeline*, ilustrado conceitualmente na Figura 4.3, atua como um tradutor, abstraindo a heterogeneidade dos formatos de *log* de entrada e saída. A figura exemplifica esse papel, mostrando a forma a qual diferentes fontes

de dados brutos, como o *log* de um servidor Web ou de um banco de dados, são processadas por uma `ParserFactory`, que seleciona a estratégia de análise adequada para cada formato, resultando em uma saída padronizada. Sua função primária é consumir um arquivo de rastro bruto, interpretando suas linhas de acordo com um formato específico, como *logs* de acesso à web ou rastros de banco de dados, e produzir uma sequência de eventos na representação interna padronizada, o Formato de Evento Intermediário (FEI).

Conceitualmente, cada estratégia `IParser`, selecionada por uma fábrica de objetos (*Factory*), implementa a lógica necessária para essa interpretação. Crucialmente, a estratégia `IParser` também define a operação inversa: como formatar um evento FEI de volta para uma representação textual. Essa segunda função é utilizada após a fase de geração para criar o arquivo de Rastro Sintético em um formato consistente e legível pelo Módulo Executor.

O *Formato de Evento Intermediário (FEI)* é fundamental para o desacoplamento entre os módulos. Ele define um evento de forma genérica, incluindo atributos essenciais para a modelagem e execução da carga:

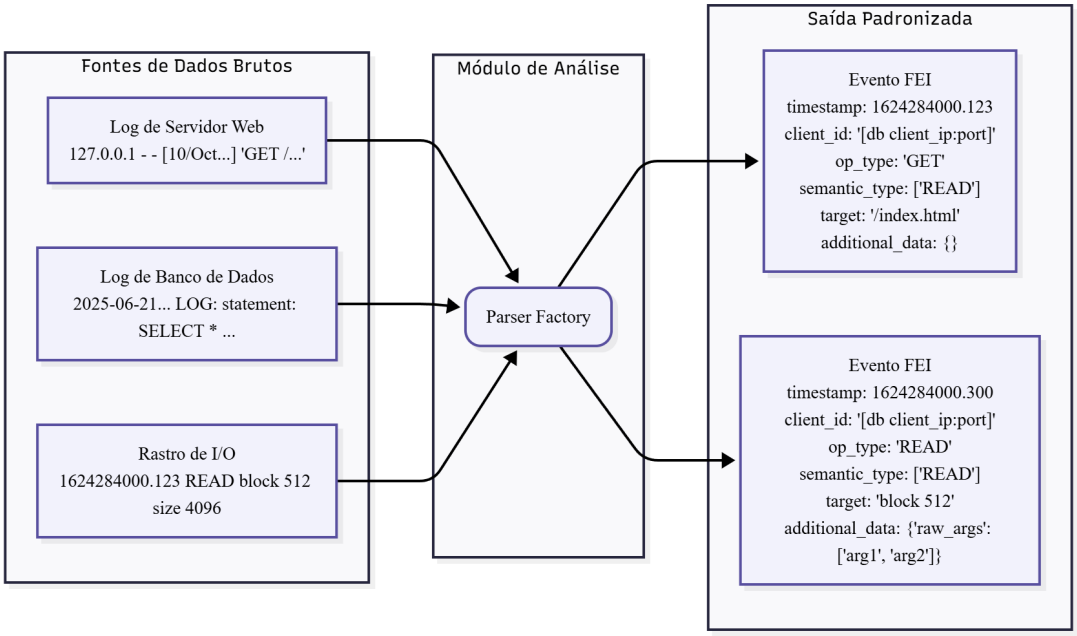
- `timestamp`: Momento de ocorrência do evento, registrado com precisão parametrizável.
- `op_type`: A operação específica realizada, como `GET`, `POST`, `SET`, `SELECT`.
- `target`: Identificador do recurso sobre o qual a operação atua, por exemplo, uma chave de banco de dados, uma URL ou um nome de arquivo.
- `client_id`: Identificador da entidade que originou a operação (opcional).
- `semantic_type`: Informação semântica adicional sobre a natureza da operação, tal qual `CREATE`, `UPDATE`, `DELETE`, útil para modelos de geração mais avançados que consideram o ciclo de vida dos dados.
- `additional_data`: Campo flexível para armazenar metadados específicos da operação original, como argumentos de comando completos ou informações de cabeçalho.

Ao normalizar os eventos para este formato interno, o Módulo Gerador pode operar de forma independente em relação aos detalhes sintáticos do rastro original e do rastro sintético final.

4.4 MÓDULO DE GERAÇÃO (*GENERATOR*)

Este é o núcleo lógico da ferramenta, responsável pela criação da sequência de eventos que comporá o rastro sintético. Ele recebe a sequência de eventos FEI, previamente processada pelo *Parser*, e aplica uma estratégia `IGenerator` para produzir uma nova sequência de eventos, também no formato FEI. Cada estratégia encapsula um

Figura 4.3 – Fluxo conceitual do Módulo de Análise (*Parser*).



Fonte: O autor (2025).

método específico de geração de carga, tipicamente envolvendo um processo interno de duas sub-etapas conceituais, conforme ilustra a Figura 4.4. A figura demonstra como a estratégia `IGenerator` consome os eventos FEI para a etapa de caracterização, resultando na construção do modelo de carga. Este modelo é então utilizado na etapa de síntese, juntamente com parâmetros externos, para a geração de novos eventos, que compõem o rastro sintético final.

A primeira sub-etapa é a caracterização, na qual a estratégia analisa a sequência de eventos FEI de entrada para extrair suas características relevantes. O resultado desta fase é um modelo de carga de trabalho interno, cuja estrutura depende da técnica utilizada, podendo incluir, por exemplo, distribuições de probabilidade por tipos de operação, popularidade de recursos por comando e tempos de chegada para um modelo *heatmap*, ou matrizes de transição para um modelo baseado em Cadeias de Markov.

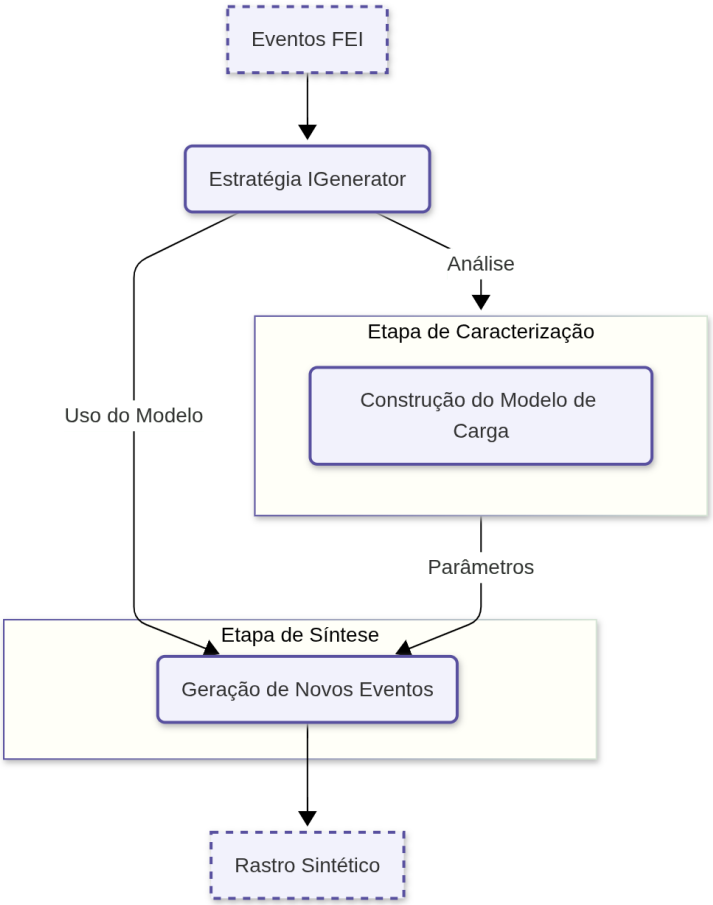
A segunda sub-etapa é a síntese, onde a estratégia utiliza o modelo de carga de trabalho construído para gerar a nova sequência de eventos FEI. O objetivo é que esta sequência sintética exiba características ou comportamentos semelhantes às do rastro original, conforme capturado pelo modelo. A duração e outras propriedades da carga sintética, como a estratégia de mapeamento temporal caso a duração desejada exceda a do rastro original, podem ser parametrizadas.

Após a fase de síntese produzir a sequência de eventos FEI desejada, uma etapa final é necessária para criar o artefato consumível pelo Módulo Executor. O sistema utiliza novamente a instância do *Parser* selecionado no início para converter

cada evento FEI sintético de volta para uma representação textual. Essas linhas de texto formatadas são escritas sequencialmente em um arquivo, constituindo o Rastro Sintético final. Este passo garante que o formato do rastro gerado seja consistente com o formato esperado pelo sistema-alvo e pela estratégia `IExecutor` correspondente.

A arquitetura permite a implementação de estratégias de geração de complexidade variada, desde uma simples replicação (*replay*) dos eventos originais até modelos estatísticos sofisticados que capturam dependências temporais e espaciais. Conforme será detalhado no Capítulo 5, a implementação deste trabalho fornece duas estratégias concretas: um *replay* simples e um gerador probabilístico baseado em *heatmap*.

Figura 4.4 – Fluxo conceitual interno de uma estratégia do Módulo Gerador, resultando em eventos FEI.



Fonte: O autor (2025).

4.5 MÓDULO DE EXECUÇÃO (*EXECUTOR*)

O estágio final do *pipeline* é responsável por transformar o rastro sintético, contido em um arquivo de *log*, em interações concretas com um sistema-alvo. Este módulo consome o arquivo de rastro sintético, linha por linha, e utiliza uma estratégia `IExecutor` para orquestrar a execução da carga, conforme ilustrado na Figura 4.5. Na figura, a

`Thread Main` atua como gerente, lendo o arquivo de rastro sintético, interpretando cada linha e enfileirando tarefas com o comando e o tempo alvo. Múltiplas `Threads Worker` consomem essas tarefas concorrentemente, aguardam o tempo exato agendado, e usam suas próprias instâncias da estratégia `IExecutor` para disparar a operação contra o sistema alvo.

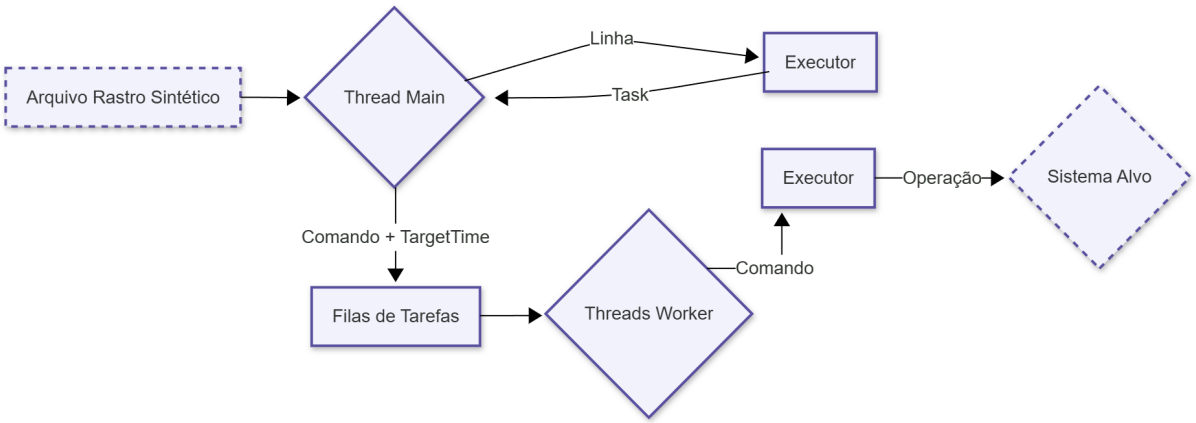
O processo inicia com um componente central de leitura e despacho, que lê as linhas do rastro sintético e determina o momento exato em que a operação representada por cada linha deve ser executada. Para cada linha lida, a interpretação e a tradução são realizadas pela estratégia `IExecutor` selecionada. Esta estratégia é responsável por analisar a sintaxe da linha de *log*, extraindo a operação e seus parâmetros, e por traduzir essa informação em uma ação concreta a ser realizada contra o sistema-alvo, como um comando de banco de dados ou uma requisição HTTP. Notavelmente, a mesma instância da estratégia pode ser utilizada pelo componente central para a interpretação inicial da linha antes do despacho.

O controle temporal é gerenciado por componentes executores concorrentes que recebem as tarefas agendadas. Nesta arquitetura, cada *thread* simula o comportamento de um cliente independente, mantendo sua própria conexão persistente com o sistema-alvo. Embora a implementação atual concentre a geração de carga em um único nó físico, limitado pelos recursos da CPU e rede da máquina hospedeira, a arquitetura baseada em leitura de arquivos de rastro foi projetada para ser intrinsecamente escalável. Para simular cenários distribuídos massivos, o arquivo de rastro sintético pode ser particionado e distribuído entre múltiplas instâncias do Defigium rodando em nós distintos, sincronizados via NTP ou orquestrados por tecnologias como MPI, permitindo a injeção de carga a partir de múltiplas origens simultaneamente.

A arquitetura prevê que diferentes estratégias `IExecutor` possam ser implementadas para interagir com distintos tipos de sistemas-alvo. Cada estratégia encapsula o conhecimento necessário para interpretar as linhas do rastro sintético, formatadas para aquele sistema específico, e para executar as operações correspondentes em seu protocolo nativo.

Em suma, a arquitetura proposta para o Defigium estabelece um *framework* modular e extensível, projetado especificamente para a geração de carga personalizada. Sua estrutura em *pipeline*, com estágios bem definidos para análise, geração e execução, promove um alto nível de desacoplamento. A aplicação dos padrões *Strategy* e *Factory Method*, juntamente com a definição de interfaces claras e o uso interno de um Formato de Evento Intermediário (FEI) abstrato, garante a flexibilidade necessária para integrar diferentes formatos de rastros, técnicas de modelagem estatística e sistemas-alvo. Essa concepção modular não apenas facilita a extensão futura da ferramenta, mas também permite a criação e execução de cenários de geração de carga de alta fidelidade, fundamentados nas características observadas em rastros reais.

Figura 4.5 – Fluxo conceitual do Módulo Executor.



Fonte: O autor (2025).

5 DEFIGIUM: IMPLEMENTAÇÃO

Este capítulo descreve a implementação[1] prática da ferramenta Defigium, detalhando como a arquitetura modular e extensível, apresentada conceitualmente no Capítulo 4, foi concretizada. A implementação adota uma abordagem bilíngue, utilizando Python para as fases de análise e geração de carga, aproveitando sua flexibilidade e ecossistema de bibliotecas, e C++ para a fase de execução, visando alto desempenho e controle preciso sobre o tempo.

5.1 ESTRUTURA GERAL E CONFIGURAÇÃO

A ferramenta é organizada em dois processos principais, correspondentes às fases de geração e execução, que se comunicam através de um arquivo de rastro sintético. O comportamento de ambos os processos é orquestrado por um único arquivo de configuração, `config.yaml`, localizado na raiz do projeto.

Este arquivo, interpretado pela função `load_config`, define qual estratégia de *Parser*, *Generator* e *Executor* utilizar, os caminhos dos arquivos de entrada e saída, e parâmetros específicos para cada módulo. A Listagem 5.1 apresenta um exemplo da estrutura deste arquivo.

Listagem 5.1 – Exemplo da estrutura do arquivo `config.yaml`.

```
# Caminhos de entrada e saida
input_trace: "traces/original.log"
output_trace: "traces/generated.log"

# Configuracao do Parser (arredondamento)
parser_config:
  type: "redis"
  precision: 6 # 6 casas decimais

# Configuracao do Gerador
generator_config:
  type: "heatmap" # Opcoes: "replay", "heatmap"
  duration: 111  # Segundos
  percentage_interval: 1.0 # Granularidade de 1%
  time_mapping: "cyclic" # Opcoes: "cyclic", "stretch"

# Configuracao do Executor C++
executor_config:
  target_address: "tcp://127.0.0.1:6379"
  worker_threads: 8
```

Para validar a arquitetura modular, este trabalho focou na implementação de instâncias concretas para um sistema-alvo específico. Foi desenvolvido um *Parser* e

[1] O código-fonte da ferramenta é aberto e está disponível em: https://github.com/FKettl/Defigium

um *Executor* para o sistema de armazenamento chave-valor Redis, cujo formato de *log* é obtido pelo comando `MONITOR`. Além disso, no Módulo Gerador, foram implementadas duas estratégias distintas: uma replicação direta (`ReplayGenerator`) e um modelo probabilístico baseado em *heatmap* (`HeatmapGenerator`).

5.2 PROCESSO DE GERAÇÃO (PYTHON)

A primeira parte do *pipeline*, responsável pela análise do rastro original e pela geração do rastro sintético, foi implementada em Python 3. O ponto de entrada é o script `main.py`, que coordena a execução sequencial dos módulos *Parser* e *Generator*.

5.2.1 Implementação do Módulo Parser

Seguindo o padrão *Strategy*, o módulo *Parser* é composto pela interface abstrata `IParser` e por implementações concretas para diferentes formatos de log. A instanciação da estratégia apropriada é gerenciada pela `ParserFactory`, que seleciona a classe correspondente com base no tipo especificado no arquivo `config.yaml`. Além disso, a fábrica também injeta parâmetros de configuração na instância do *parser*, como a granularidade desejada para o arredondamento dos *timestamps*.

A implementação concreta fornecida, `RedisParser`, especializa-se no formato de *log* gerado pelo comando `MONITOR` do Redis. Para ilustrar o processo de tradução, a Listagem 5.2 apresenta duas linhas de um *log* bruto.

Listagem 5.2 – Exemplo de linhas de log do comando `MONITOR` do Redis.

```
1760993.289413 [0 172.17.0.1:33600] "ZADD" "_indices" "-5.94" "user62"
1760104.359119 [0 172.17.0.1:46812] "HGETALL" "user43029"
```

O `RedisParser` processa essas linhas e as converte para a representação interna FEI, conforme exemplificado na Listagem 5.3. Note como o comando é mapeado para `op_type` e `target`, e os argumentos restantes são armazenados em `additional_data`.

Internamente, esta classe implementa a lógica necessária para as duas funções principais do *Parser*. Para a conversão de log bruto para FEI, ela utiliza expressões regulares para identificar os componentes básicos da linha (`timestamp`, `client_id`) e funções auxiliares dedicadas para processar o comando. Essas funções dividem corretamente os argumentos e identificam a operação (`op_type`) e o recurso-alvo (`target`) com base na sintaxe esperada. Durante este processo, o *timestamp* extraído é arredondado conforme a granularidade configurada. A classe também contém um mapeamento interno que associa os tipos de operação Redis à sua semântica correspondente

Listagem 5.3 – Representação dos eventos FEI gerados pelo `RedisParser`.

```
# Evento FEI 1 (convertido do ZADD)
{
  'timestamp': 1760993.289413,
  'client_id': '172.17.0.1:33600',
  'op_type': 'ZADD',
  'semantic_type': 'UPDATE',
  'target': '_indices',
  'additional_data': ['-5.94', 'user62']
}

# Evento FEI 2 (convertido do HGETALL)
{
  'timestamp': 1760104.359119,
  'client_id': '172.17.0.1:46812',
  'op_type': 'HGETALL',
  'semantic_type': 'READ',
  'target': 'user43029',
  'additional_data': []
}
```

(`CREATE`, `UPDATE`, `READ` e `DELETE`), com o objetivo de permitir considerar a dependência de dados na fase de síntese. Caso a linha seja inválida, um indicativo de falha é retornado e ela é ignorada pela rotina principal de *parsing*.

Para a função inversa, a formatação de FEI para log, o `RedisParser` implementa uma função que converte um objeto `FEIEvent` de volta para um string textual no formato `MONITOR`, incluindo a lógica para formatar e escapar corretamente os argumentos com aspas. Essa funcionalidade é essencial para gerar o arquivo de rastro sintético consumido pelo *Executor*.

Adicionalmente, o `RedisParser` expõe um método auxiliar para a geração de argumentos sintéticos. Este método é utilizado por estratégias de geração, como o `HeatmapGenerator`, para criar dados realistas, a exemplo de valores aleatórios para operações `SET` ou pares campo-valor para `HMSET`, durante a fase de síntese da carga, utilizando informações de contexto como o conjunto de chaves atualmente disponíveis na simulação.

Como uma escolha de projeto focada na validação da arquitetura e nos experimentos com o *benchmark* YCSB, a implementação atual do `RedisParser` concentra-se nos comandos tipicamente gerados pelo *Workload A* do YCSB. Embora não cubra a totalidade dos comandos Redis, a estrutura modular da ferramenta permite que o suporte a novos comandos ou mesmo a formatos de log completamente diferentes, como MongoDB ou HTTP, seja adicionado de forma simples: bastaria criar novas classes de *Parser* e registrá-las na fábrica correspondente.

5.2.2 Implementação do Módulo Generator

De forma análoga ao *Parser*, o módulo *Generator* segue o padrão *Strategy*, composto pela interface abstrata `IGenerator` e implementações concretas gerenciadas pela `GeneratorFactory`. Duas estratégias foram implementadas para demonstrar a flexibilidade da arquitetura.

5.2.2.1 ReplayGenerator

A primeira estratégia, `ReplayGenerator`, oferece uma implementação direta. Sua função é simplesmente retornar a sequência original de eventos `FEIEvent` recebida do *Parser*, sem qualquer alteração. Esta abordagem serve como uma linha de base para os experimentos e é útil para validar a precisão do Módulo *Executor*, garantindo que ele consiga reproduzir um rastro conhecido.

5.2.2.2 HeatmapGenerator

A segunda estratégia, `HeatmapGenerator`, representa uma abordagem mais sofisticada, baseada em captar características, implementando internamente as duas fases conceituais de caracterização e síntese.

Na fase de caracterização, a estratégia itera sobre os eventos `FEIEvent` de entrada, determinando a qual intervalo de tempo percentual cada evento pertence. Para cada intervalo, ela acumula distribuições de probabilidade para três características-chave: 1) a frequência dos tipos de operação, 2) a popularidade de cada recurso para cada tipo de operação e 3) a distribuição dos tempos de chegada entre eventos. O resultado é um modelo probabilístico, um dicionário Python que armazena o mapa de calor da carga. A Listagem 5.4 ilustra uma versão simplificada desta estrutura de dados.

Na fase de síntese, o `HeatmapGenerator` utiliza o modelo probabilístico construído para gerar uma nova sequência de eventos `FEIEvent` através de um processo de amostragem condicional hierárquica. Dessa forma, a ferramenta preserva as correlações observadas, onde a escolha do recurso-alvo é condicionada à distribuição de probabilidade específica do tipo de operação sorteado para aquele intervalo de tempo. Um *loop* principal avança o tempo simulado até atingir a duração alvo da simulação. A cada iteração, determina-se qual índice de intervalo do modelo original deve ser utilizado, aplicando a estratégia de mapeamento temporal configurada. A estratégia `cyclic` repete o padrão aprendido, enquanto a estratégia `stretch` estende a duração do padrão mantendo a taxa de operações de cada fase.

Uma vez determinado o índice do intervalo correto, realiza-se uma sequência de sorteios probabilísticos. Primeiro, seleciona-se um `op_type` com base na distribuição geral de operações daquele intervalo. Segundo, seleciona-se um `target` com base na

Listagem 5.4 – Estrutura de dados simplificada do modelo `Heatmap` gerado.

```
{
  "total_duration_ms": 111000.0,
  "op_semantics": {
    "ZADD": ["UPDATE"],
    "HGETALL": ["READ"]
  },

  # Distribuicao de operacoes por intervalo
  "heatmap": {
    0: { "ZADD": 0.4, "HGETALL": 0.6 },
    1: { "ZADD": 0.3, "HGETALL": 0.7 }
    # ... (demais intervalos)
  },

  # Distribuicao de alvos (hotspots) por operacao, por intervalo
  "target_probabilities_by_op": {
    0: {
      "ZADD": { "_indices": 1.0 },
      "HGETALL": { "user43029": 0.9, "user123": 0.1 }
    },
    1: {
      # ...
    }
  },

  # Distribuicao de tempos de chegada (ritmo) por intervalo
  "inter_arrival_probabilities": {
    0: { 0.001: 0.8, 0.002: 0.2 },
    1: { 0.002: 0.7, 0.003: 0.3 }
    # ...
  }
}
```

distribuição de popularidade de recursos específica para o `op_type` recém-sorteado e o intervalo atual, conforme aprendido na caracterização.

Com a operação e o alvo definidos, aplica-se uma lógica baseada em estado para gerenciar o ciclo de vida dos recursos na simulação. Para operações ambíguas, como `SET` ou `HMSET`, marcadas com semântica `CREATE` e `UPDATE`, verifica-se a existência do recurso sorteado em uma *pool* que rastreia as chaves ativas: se o alvo não existe, ele é adicionado ao conjunto, simulando uma criação; se já existe, nada é feito nessa *pool*, simulando uma atualização. Operações de leitura ou deleção sobre chaves que não constam na *pool* são simplesmente descartadas naquela iteração, mantendo a consistência lógica da simulação.

Após validar a operação e o alvo, invoca-se o método auxiliar do *Parser* para gerar argumentos sintéticos apropriados, se necessário. O objeto `FEIEvent` sintético é então criado. Por fim, sorteia-se o tempo de chegada para o próximo evento a partir da distribuição de tempos do intervalo atual, e o tempo simulado avança. O processo

retorna a lista completa de eventos sintéticos gerados.

A existência destas duas estratégias distintas, selecionáveis via configuração, visa demonstrar a capacidade da arquitetura de incorporar diferentes abordagens para a geração de carga, desde a replicação simples até a geração baseada em um modelo probabilístico de múltiplos intervalos.

5.3 PROCESSO DE EXECUÇÃO (C++)

A fase final do *pipeline*, responsável pela execução da carga contra o sistema-alvo, foi implementada em `C++` 17 para otimizar o desempenho e a precisão temporal, com o código contido no diretório `src/executors`. Esta fase adota a arquitetura Gerente-Trabalhador.

5.3.1 Estrutura e Orquestração

O ponto de entrada é a função `main` no arquivo `main.cpp`, que atua como a *thread* Gerente. Inicialmente, ela lê o arquivo de configuração `config.yaml`, utilizando a biblioteca `yaml-cpp`, para obter parâmetros essenciais como o caminho do arquivo de rastro sintético e as configurações específicas do executor, incluindo o número de *threads* trabalhadoras.

Em seguida, a *thread* Gerente instancia um objeto que implementa a interface `IExecutor` através da `ExecutorFactory`. Esta instância específica é utilizada pela *thread* Gerente com o propósito de interpretar as linhas do rastro sintético. Logo após, uma *pool* de *threads* trabalhadoras é criada e iniciada, onde cada *thread* executará a função designada.

A *thread* Gerente então entra em um *loop* para processar o arquivo de rastro sintético linha a linha. Para cada linha lida, ela invoca a rotina de interpretação da instância do executor para converter o string em uma estrutura de dados interna, que contém o *timestamp* original e a representação do comando. Com base nesse *timestamp* e no tempo de início da execução, calcula-se o tempo absoluto exato para o disparo daquela operação, utilizando a biblioteca `std::chrono`. Uma estrutura final, contendo o tempo alvo e o comando completo, é então enviada para uma das filas de tarefas associadas às *threads* trabalhadoras, utilizando uma estratégia de distribuição *round-robin*.

Após processar todas as linhas do arquivo, a *thread* Gerente envia uma mensagem especial de finalização para cada fila de trabalhador. Por fim, ela aguarda a conclusão de todas as *threads* trabalhadoras. Ao término de todas as *threads*, a Gerente lê os valores finais dos contadores atômicos que registraram o número de operações bem-sucedidas e malsucedidas, imprimindo um sumário simples no console. A classe `ThreadSafeQueue` utilizada para comunicação inter-thread é uma implemen-

tação padrão de fila bloqueante, garantindo a sincronização através de `std::mutex` e `std::condition_variable`.

5.3.2 Função da Thread Trabalhadora e Estratégia Redis

Cada *thread* trabalhadora, ao ser iniciada, executa a função `worker_function`. Sua primeira ação é criar sua própria instância, independente da *thread* Gerente, da estratégia `IExecutor`, selecionada pela `ExecutorFactory` com base na configuração. Em seguida, estabelece a conexão com o sistema-alvo invocando o método de conexão da estratégia.

No caso da estratégia implementada, `RedisExecutorStrategy`, o método de conexão utiliza a biblioteca cliente `sw::redis++`. Ele configura as opções de conexão, incluindo o endereço do servidor obtido da configuração geral, e estabelece *timeouts* para operações de *socket* e para a tentativa inicial de conexão. Uma instância do cliente `sw::redis::Redis` é então criada e mantida internamente para interagir com o servidor Redis.

Após a conexão ser estabelecida com sucesso, a *thread* entra em um *loop* de processamento. Ela aguarda por tarefas em sua fila designada (`ThreadSafeQueue`), bloqueando na operação de remoção até que uma tarefa esteja disponível. Ao receber uma tarefa, a *thread* primeiramente verifica se o tipo de operação corresponde à mensagem especial de finalização; caso positivo, o *loop* é encerrado e a *thread* prossegue para a finalização.

Caso contrário, tratando-se de uma tarefa regular, a *thread* utiliza a função `std::this_thread::sleep_until`, passando o tempo alvo contido na tarefa. Isso garante que a *thread* pause sua execução até o momento exato agendado para o disparo da operação, assegurando a precisão temporal da execução da carga. No momento agendado, a *thread* invoca o método de execução, `execute`, da sua instância `IExecutor`, passando o objeto recebido na tarefa.

A implementação específica em `RedisExecutorStrategy::execute` contém a lógica para interagir com o Redis. Ela analisa o tipo de operação do comando recebido e seus argumentos mapeando essa informação para a chamada de método correspondente na instância do cliente. Por exemplo, uma operação do tipo `SET` resultará em uma chamada da função `m_redis_client->set(...)` com o alvo e o primeiro argumento, enquanto um `HMSET` iterará sobre os argumentos para construir pares chave-valor e chamar `m_redis_client->hmset(...)`. Outros comandos implementados, como `GET`, `HGETALL`, `DEL` e `ZADD`, são tratados de forma análoga, traduzindo a representação interna do comando para a interação específica com o servidor Redis. O método retorna um `ExecutionResult` indicando o sucesso ou falha da operação.

Com base no resultado retornado, a *thread* trabalhadora incrementa um dos contadores atômicos globais de sucesso ou erro. Ao sair do *loop* principal, e antes

de finalizar completamente sua execução, a *thread* explicitamente destrói o objeto executor para assegurar que a conexão de rede seja fechada corretamente, mesmo em caso de erros anteriores, prevenindo assim que a *thread* principal fique bloqueada indefinidamente na operação `join` aguardando por uma *thread* que nunca termina.

5.3.3 Considerações sobre Compilação e Dependências

O processo de execução `C++`, devido à sua natureza compilada, requer uma etapa prévia de construção. Para automatizar e simplificar este processo, um arquivo `Makefile` é fornecido no diretório `src/executors`. Este *script* de compilação utiliza um compilador `C++` 17 padrão e gerencia a inclusão das dependências externas necessárias. As dependências base para o núcleo do Executor incluem a biblioteca `yaml-cpp` para a leitura e interpretação do arquivo de configuração `config.yaml`, além das bibliotecas padrão do `C++` para *threads* e tempo, `pthread` e `chrono`.

Uma característica importante da implementação do Executor `C++`, visando reforçar a modularidade e facilitar a extensibilidade, é o uso de compilação condicional. A `ExecutorFactory` utiliza diretivas de pré-processador (`#ifdef ... #endif`) para incluir o código de cabeçalho `.h` e registrar a lógica de instanciação de cada estratégia `IExecutor` somente se uma flag de compilação específica estiver definida. Por exemplo, o código referente à `RedisExecutorStrategy` só é compilado se a flag `BUILD_REDIS_STRATEGY` for passada ao compilador.

Essa abordagem é orquestrada pelo `Makefile`. Ele define alvos separados para compilar o Executor com diferentes conjuntos de estratégias habilitadas, como `make redis` ou `make http`. Cada alvo passa a flag correspondente para o compilador e especifica as bibliotecas cliente adicionais necessárias apenas para aquela estratégia, como `-lredis++ -lhiredis` para Redis. O `Makefile` também inclui um *template* comentado que serve como guia para adicionar facilmente novos alvos de compilação para futuras estratégias de execução. Este mecanismo garante que, ao compilar o Executor para suportar apenas um determinado sistema-alvo, não seja necessário instalar ou vincular as bibliotecas de dependência de outros sistemas-alvo não utilizados, mantendo o processo de compilação e as dependências do executável final mais enxutas. A compilação resulta em um único executável que pode ser invocado pela linha de comando. A Listagem 5.5 demonstra como o `Makefile` orquestra essa abordagem, definindo alvos que passam *flags* de pré-processador e bibliotecas específicas para o compilador. Já a Listagem 5.6 ilustra como essas *flags* são usadas dentro do código C++ da `ExecutorFactory` para incluir condicionalmente o código de cada estratégia.

Esta implementação bilíngue, combinando a flexibilidade do Python com o desempenho do C++ e utilizando mecanismos como a compilação condicional, materializa a arquitetura modular proposta, oferecendo uma ferramenta configurável e genuinamente extensível para análise, geração e execução de cargas de trabalho.

Listagem 5.5 – Trecho do `Makefile` ilustrando os alvos de compilação condicional.

```
# Alvo para compilar com a estrategia Redis
redis:
    g++ main.cpp ... \
    -DBUILD_REDIS_STRATEGY \  # Define a flag
    -lredis++ -lhiredis    # Linka bibliotecas especificas
    -o defigium_executor_redis

# Alvo (template) para compilar com a estrategia HTTP
http:
    g++ main.cpp ... \
    -DBUILD_HTTP_STRATEGY \
    -lcurl
    -o defigium_executor_http
```

Listagem 5.6 – Exemplo da diretiva `#ifdef` na `ExecutorFactory` C++.

```
// Dentro de ExecutorFactory::create_executor(...)

#ifdef BUILD_REDIS_STRATEGY
if (type == "redis") {
    return std::make_unique<RedisExecutorStrategy>(config);
}
#endif

#ifdef BUILD_HTTP_STRATEGY
if (type == "http") {
    // return std::make_unique<HttpExecutorStrategy>(config);
}
#endif

throw std::runtime_error("Estrategia␣desconhecida:␣" + type);
```

6 EXPERIMENTOS E ANÁLISE DE RESULTADOS

Este capítulo apresenta a metodologia experimental e os resultados obtidos para validar a arquitetura e a implementação da ferramenta Defigium, detalhadas nos Capítulos 4 e 5. O objetivo principal é conduzir uma avaliação de fidelidade, demonstrando, por meio de uma análise comparativa de métricas-chave, a fidelidade do Módulo Gerador e a precisão temporal do Módulo Executor. A análise estende-se à investigação de artefatos sistêmicos resultantes das escolhas de precisão temporal e da simulação de concorrência.

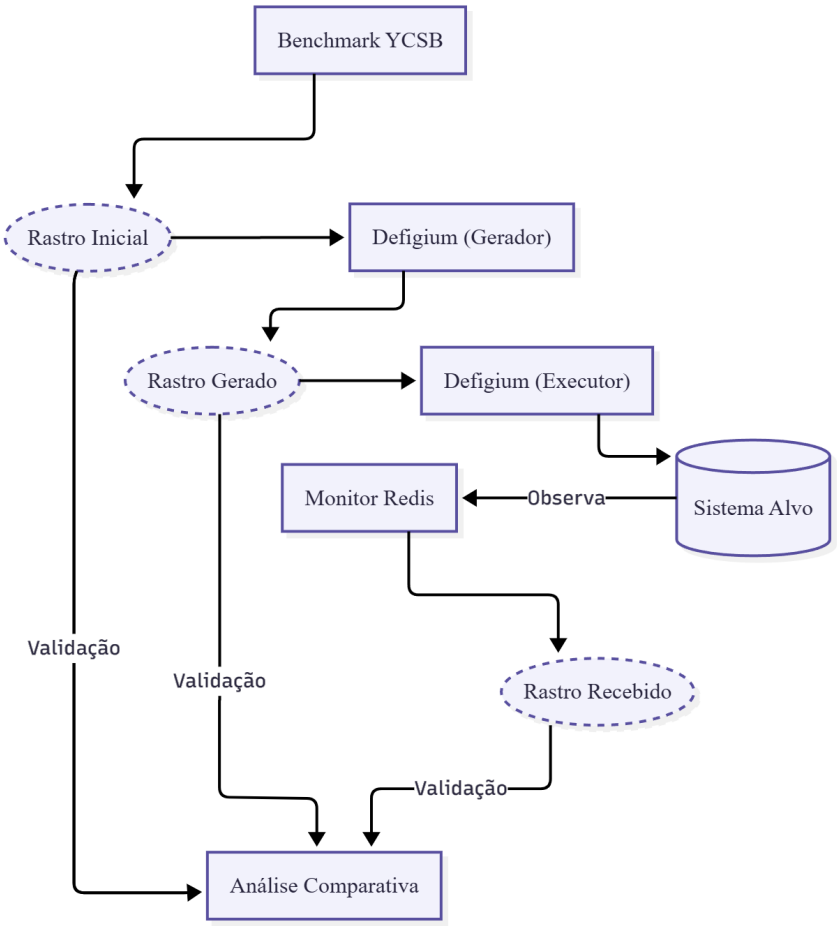
6.1 METODOLOGIA EXPERIMENTAL

A validação foi conduzida através de um processo de quatro etapas: 1) A captura de uma carga de trabalho “Inicial” real; 2) A geração de um rastro “Gerado” baseado na carga real, utilizando diferentes estratégias; 3) A execução da carga sintética contra o sistema-alvo para gerar um novo rastro “Recebido”; e 4) A análise comparativa dos três rastros resultantes. Esse fluxo metodológico, ilustrado na Figura 6.1, foi projetado para validar os diferentes componentes.

6.1.1 Carga de Trabalho Inicial

Como base para todos os experimentos, foi utilizado um rastro de operações do Redis gerado pelo *benchmark* YCSB com o *Workload A* (COOPER *et al.*, 2010), capturado com o comando `MONITOR` e precisão de 6 casas decimais. Este rastro foi construído pela concatenação de duas execuções distintas do YCSB. A escolha deste perfil de carga justifica-se por sua ampla utilização na indústria para representar aplicações de nuvem com alta concorrência e uso intensivo de atualizações, sendo 50% leitura e 50% escrita e exibindo uma distribuição de acesso a dados Zipfiana. A primeira foi a fase de carga do YCSB, contendo 10.000 inserções que geraram 20.000 operações no rastro, ocorrendo em uma rajada inicial de 0 a 9 segundos. A segunda foi a fase de execução, contendo 100.000 operações, leituras e atualizações, iniciada após um intervalo de inatividade. O resultado é o rastro “Inicial” com 120.000 operações e uma duração total de aproximadamente 111 segundos, caracterizado por três fases distintas: uma rajada de carga inicial, uma espera de aproximadamente 87 segundos, e uma rajada de execução final. Essa estrutura não estacionária foi escolhida para representar a natureza intermitente de tráfego real, onde períodos de ociosidade alternam com picos de demanda, permitindo avaliar se a ferramenta é capaz de replicar a fidelidade temporal sem distorcer os períodos de silêncio.

Figura 6.1 – Fluxograma da metodologia experimental de 4 etapas.



Fonte: O autor (2025).

6.1.2 Logs de Análise

Para cada experimento, três arquivos de rastro (*logs*) distintos são comparados. O primeiro é o rastro “Inicial”, que serve como a verdade-base. O segundo é o rastro “Gerado”, o arquivo sintético produzido pelo Módulo Gerador do Defigium, que utilizou uma precisão configurada de 6 casas decimais. A comparação entre “Inicial” e “Gerado” valida a fidelidade do Módulo Gerador. O terceiro é o rastro “Recebido”, capturado por um novo `MONITOR` do Redis, também com 6 casas decimais, enquanto o Módulo Executor executava a carga definida no rastro “Gerado”. A comparação entre “Gerado” e “Recebido” valida a fidelidade temporal do Módulo Executor e mede os artefatos da execução real. Finalmente, a comparação entre “Inicial” e “Recebido” serve como a validação ponta a ponta da ferramenta Defigium.

6.1.3 Métricas de Avaliação

A análise comparativa é realizada por meio de uma abordagem visual qualitativa e, para validar a fidelidade do rastro “Gerado” em relação ao “Inicial”, utiliza-se uma

abordagem quantitativa. A avaliação visual examina quatro dimensões fundamentais do comportamento da carga: a Composição da Carga, visualizada pela Proporção de Comandos; o Ritmo de Chegada, analisado pela Distribuição de Tempos entre Chegadas; o Comportamento Temporal e Vazão, observado no gráfico de Operações por Segundo ao longo do tempo; e a Localidade Espacial, representada pela CDF da Popularidade de Acesso a Recursos.

Para a análise quantitativa, foram aplicados testes estatísticos. Para as métricas categóricas, Proporção de Comandos, e discretizadas, Distribuição Tempos Chegada, foi utilizado o teste de aderência do Qui-Quadrado e o V de Cramér (V) para medir o tamanho do efeito, ou magnitude, da diferença (COHEN, 2013). Para a métrica de Acesso a Recursos, característica do *Workload A* do YCSB (COOPER *et al.*, 2010), a fidelidade é medida pela capacidade do gerador em replicar a distribuição Zipfiana subjacente. Para isso foi utilizada a biblioteca `powerlaw` do Python para ajustar um modelo a cada distribuição e extrair o expoente `alpha`, que define a forma da curva (CLAUSET; SHALIZI; NEWMAN, 2009).

6.2 DEFINIÇÃO DOS EXPERIMENTOS

Foram definidos cinco experimentos para validar sistematicamente os diferentes componentes e estratégias da ferramenta.

Para fins de clareza, as estratégias de geração mencionadas abaixo referem-se às implementações detalhadas no Capítulo 5: o *Replay*, que replica exatamente a sequência de eventos, e o *Heatmap*, que designa o modelo probabilístico baseado em intervalos temporais, descrito na Seção 5.2.2.2, capaz de aprender a distribuição de probabilidade variável ao longo do tempo.

1. Replay Simples: Utiliza o `Replay` para gerar um rastro sintético idêntico ao inicial. O objetivo é isolar e validar a fidelidade do Módulo Executor.
2. Heatmap 1% (Duração Original): Utiliza o `Heatmap` com intervalo igual a 1% e duração de 111s. O objetivo é validar a fidelidade do modelo de alta granularidade.
3. Heatmap 50% (Duração Original): Utiliza o `Heatmap` com intervalo igual a 50% e duração de 111s. O objetivo é demonstrar o impacto de um modelo de baixa granularidade na fidelidade da carga.
4. Heatmap 1% (Dobro, Cyclic): Utiliza o `Heatmap` com intervalo igual a 1%, duração de 222s e estratégia de mapeamento temporal `cyclic`. O objetivo é validar a estratégia de repetição de padrão.
5. Heatmap 1% (Dobro, Stretch): Utiliza o `Heatmap` com intervalo igual a 1%, duração de 222s e estratégia de mapeamento temporal `stretch`. O objetivo é validar a estratégia de alongamento de padrão.

6.3 ANÁLISE DOS RESULTADOS

Esta seção apresenta e analisa os resultados de cada um dos cinco experimentos definidos, correlacionando a análise visual das Figuras 6.2 a 6.6 com os resultados quantitativos apresentados nas tabelas de contagem (Tabelas 6.2 a 6.6) e na tabela de resumo estatístico (Tabela 6.1).

6.3.1 Resultados Estatísticos Quantitativos

Para fundamentar a análise visual, a Tabela 6.1 consolida os resultados quantitativos da comparação entre o rastro "Inicial" e o rastro "Gerado" para cada experimento. A tabela apresenta o V de Cramér (V) para as distribuições de Comandos e Tempos de Chegada, e o expoente `alpha` para a curva de distribuição de Acesso a Recursos.

Optou-se por não utilizar o P-valor nesta análise pois, em amostras de grande volume ($N > 100.000$), esta métrica torna-se excessivamente sensível, detectando diferenças estatísticas triviais que não possuem relevância prática (LIN; LUCAS; SHMUELI, 2013). Por esta razão, a métrica primária de fidelidade adotada é o V de Cramér, que mede o tamanho do efeito. Valores próximos de 0, tipicamente $V < 0.1$, indicam um tamanho de efeito pequeno ou trivial (COHEN, 2013, cap. 7), denotando alta similaridade entre as cargas.

Tabela 6.1 – Validação Estatística (Inicial vs. Gerado).

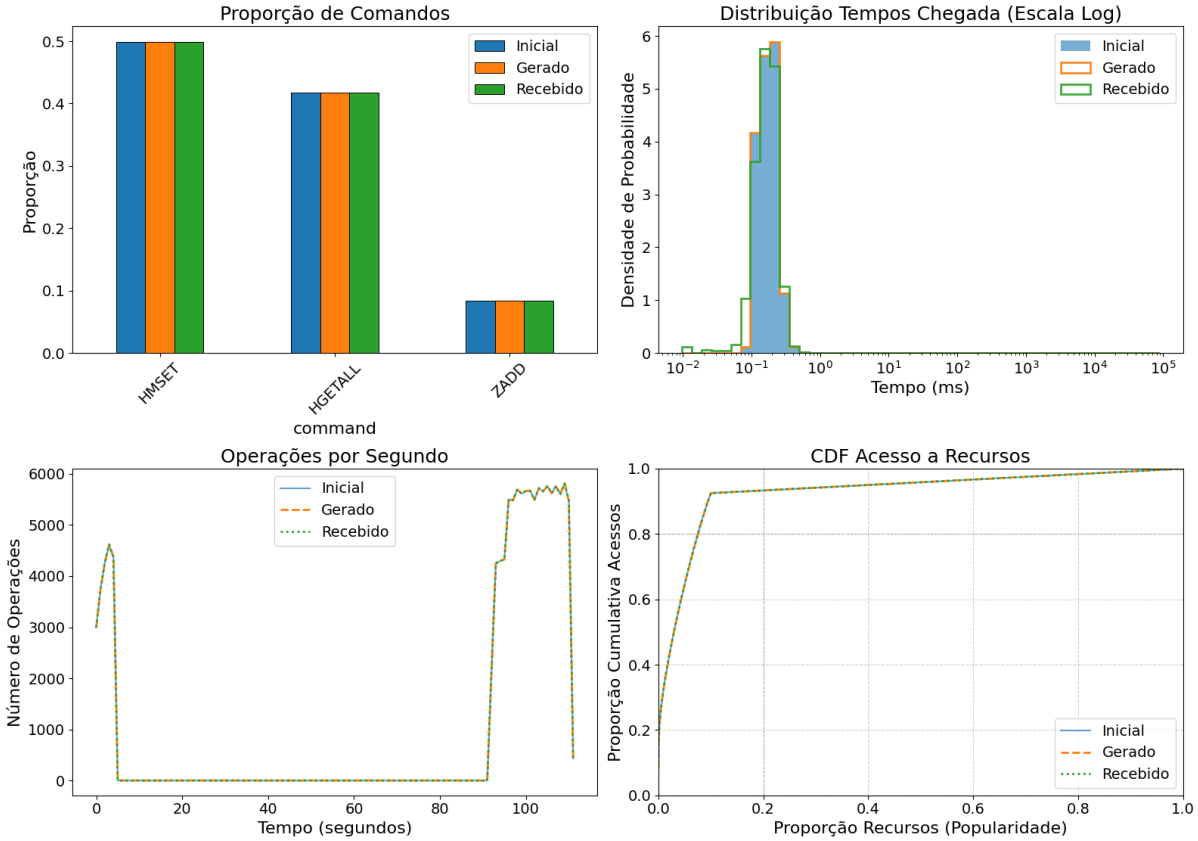
Experimento	V (Comandos)	V (Tempos)	Alpha (Inicial)	Alpha (Gerado)
Replay Simples	0.0000	0.0000	2.1536	2.1536
Heatmap 1% (Original)	0.0330	0.0170	2.1536	2.1792
Heatmap 50% (Original)	0.0933	0.0497	2.1536	2.2463
Heatmap 1% (Cyclic)	0.0018	0.0053	2.1536	2.1623
Heatmap 1% (Stretch)	0.0076	0.0045	2.1536	2.1543

Fonte: O autor (2025).

6.3.2 Validação de Fidelidade do Executor e do Gerador (Exp. 1 e 2)

Os experimentos 1 (Figura 6.2) e 2 (Figura 6.3) servem como a validação fundamental da ferramenta. A Figura 6.2 apresenta os resultados da estratégia `Replay`. Conforme detalhado na Tabela 6.2, a contagem de comandos gerada é idêntica à inicial (120.000 operações). A Tabela 6.1 confirma esta identidade, com V de Cramér em 0.0000 para Comandos e Tempos, e um expoente `alpha` idêntico (2.1536). Visualmente, a análise dos quadrantes demonstra a alta fidelidade temporal do Módulo Executor. Os gráficos de "Operações por Segundo" e "CDF Acesso a Recursos" mostram uma sobreposição quase exata entre as linhas "Gerado" (laranja) e "Recebido" (verde).

Figura 6.2 – Resultados do Experimento 1 (Replay Simples).



Fonte: O autor (2025).

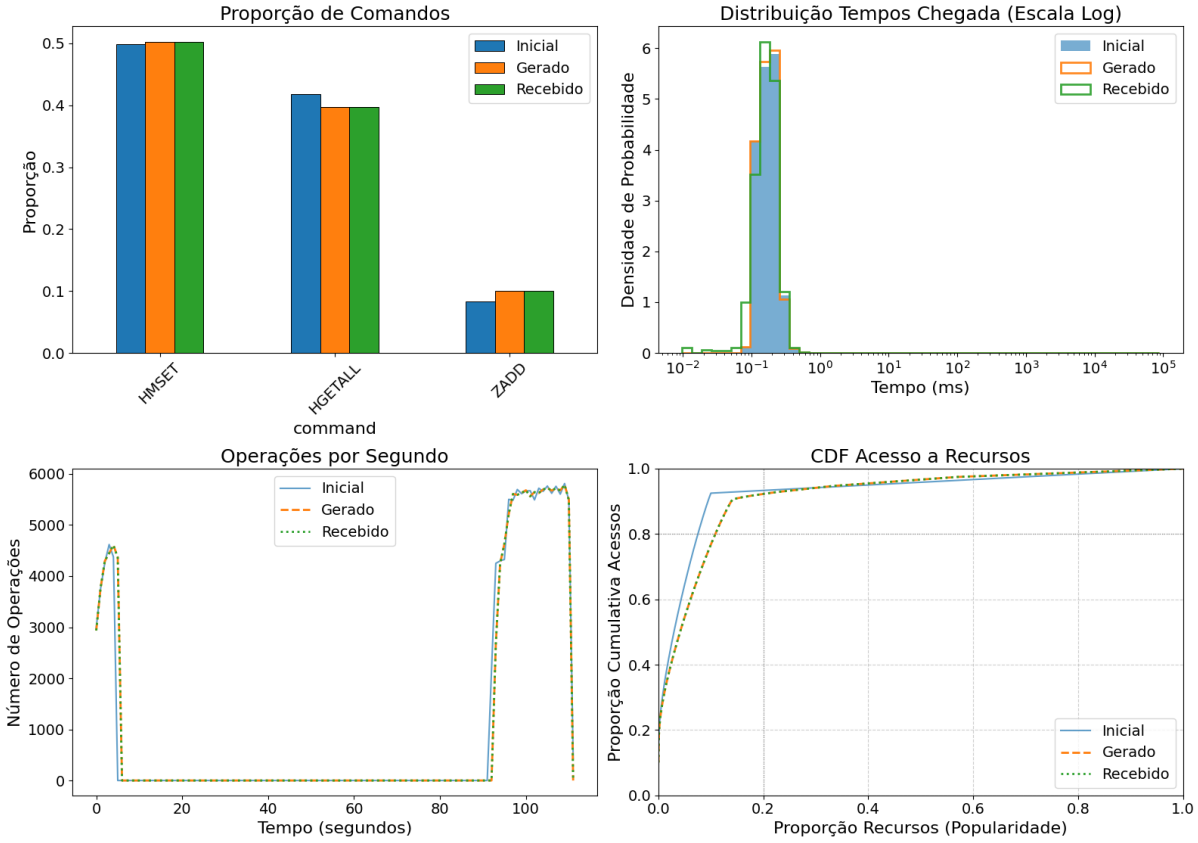
Tabela 6.2 – Contagem de Comandos: Experimento 1 (Replay Simples).

Comando	Inicial (%)	Gerado (%)
HMSET	49.90%	49.90%
HGETALL	41.77%	41.77%
ZADD	8.33%	8.33%
TOTAL	**120.000**	**120.000**

Fonte: O autor (2025).

A Figura 6.3 mostra os resultados do `Heatmap` com granularidade fina de 1%. Neste cenário, os resultados quantitativos da Tabela 6.1 indicam uma alta similaridade. A composição de comandos (Tabela 6.3) foi bem replicada, com o teste estatístico apontando um tamanho de efeito pequeno (V=0.0330). De forma similar, o ritmo da carga foi capturado com um V de Cramér ainda menor (V=0.0170). Adicionalmente, a forma da distribuição de popularidade foi bem modelada, com o expoente `alpha` (2.1792) apresentando-se próximo do original (2.1536). No entanto, foi observada uma limitação na replicação da cauda longa da distribuição: o rastro "Gerado" (com 7.084 chaves únicas) não reproduziu a totalidade dos acessos esparsos do "Inicial" (10.001 chaves únicas). Isso se deve à natureza probabilística da escolha de recursos, onde recursos com acessos mais esparsos têm baixa chance de serem selecionados.

Figura 6.3 – Resultados do Experimento 2 (Heatmap 1%, Duração Original).



Fonte: O autor (2025).

Visualmente, os gráficos corroboram essa análise. Os gráficos de "Proporção de Comandos" e "CDF Acesso a Recursos" mostram uma forte correspondência. Ademais, o gráfico "Operações por Segundo" demonstra que o modelo aprendeu e reproduziu o padrão temporal "burst-gap-burst".

Tabela 6.3 – Contagem de Comandos: Experimento 2 (Heatmap 1% Original).

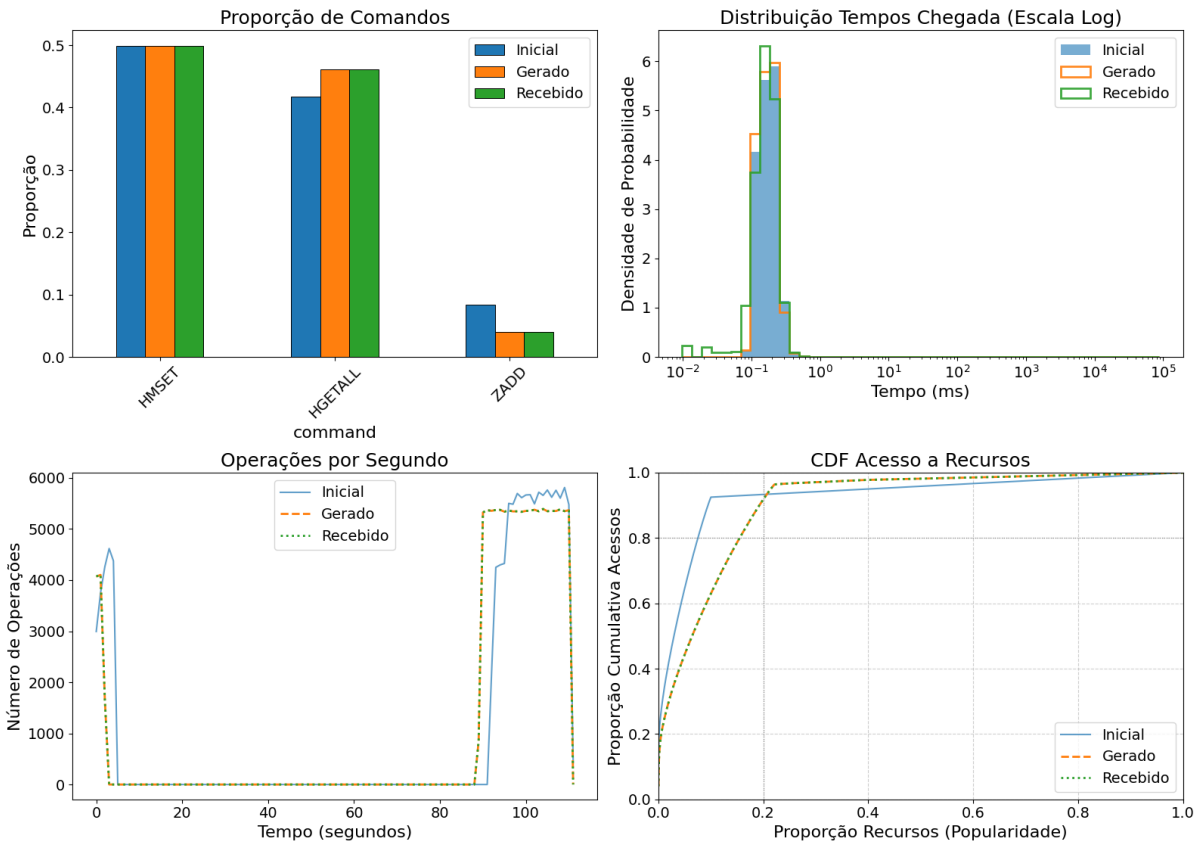
Comando	Inicial (%)	Gerado (%)
HMSET	49.90%	50.26%
HGETALL	41.77%	39.68%
ZADD	8.33%	10.06%
TOTAL	**120.000**	**120.282**

Fonte: O autor (2025).

6.3.3 Impacto da Granularidade do Modelo (Exp. 3)

O Experimento 3 (Figura 6.4) foi projetado para quantificar o impacto de um parâmetro de granularidade inadequado (50.0%). Este valor foi escolhido deliberadamente como um cenário de "pior caso" para contrastar com a granularidade fina de 1% (Exp. 2). Ao dividir a execução em apenas dois grandes intervalos, força-se o agrupamento de fases distintas (rajada e ociosidade) no mesmo cálculo probabilístico, permitindo visualizar a degradação da fidelidade quando a janela de amostragem excede a duração dos eventos temporais da carga. A Tabela 6.4 mostra a primeira distorção: o rastro "Inicial" tinha 8.33% de operações `ZADD`, enquanto o "Gerado" produziu apenas 4.00%.

Figura 6.4 – Resultados do Experimento 3 (Heatmap 50%, Duração Original).



Fonte: O autor (2025).

Tabela 6.4 – Contagem de Comandos: Experimento 3 (Heatmap 50% Original).

Comando	Inicial (%)	Gerado (%)
HMSET	49.90%	49.88%
HGETALL	41.77%	46.11%
ZADD	8.33%	4.00%
TOTAL	**120.000**	**123.182**

Fonte: O autor (2025).

Os testes estatísticos (Tabela 6.1) quantificam esta divergência. O V de Cramér para Comandos foi de 0.0933, quase três vezes maior que o do modelo de 1%. A divergência é observada também no ritmo da carga (V=0.0497) e, crucialmente, na captura da distribuição de recursos, com o expoente `alpha` (2.2463) que se mostrou mais distante do original (2.1536).

A causa raiz desta distorção é observada no gráfico “Operações por Segundo”. O modelo de 50% agrupou estatisticamente a fase de *load* (0-9s) e uma espera de 87 segundos. Na fase de síntese, a probabilidade de sortear o delta de 87s esteve presente desde o inicio, fazendo com que a rajada inicial fosse interrompido mais cedo pelo período de inatividade e, por consequência, sobrou mais tempo para a fase de execução final. Esta distorção temporal é a causa direta das diferenças, distorcendo a proporção de comandos e achatando a curva “CDF Acesso a Recursos”, que perdeu o padrão de *hotspot* da fase de execução.

6.3.4 Validação das Estratégias de Mapeamento Temporal (Exp. 4 e 5)

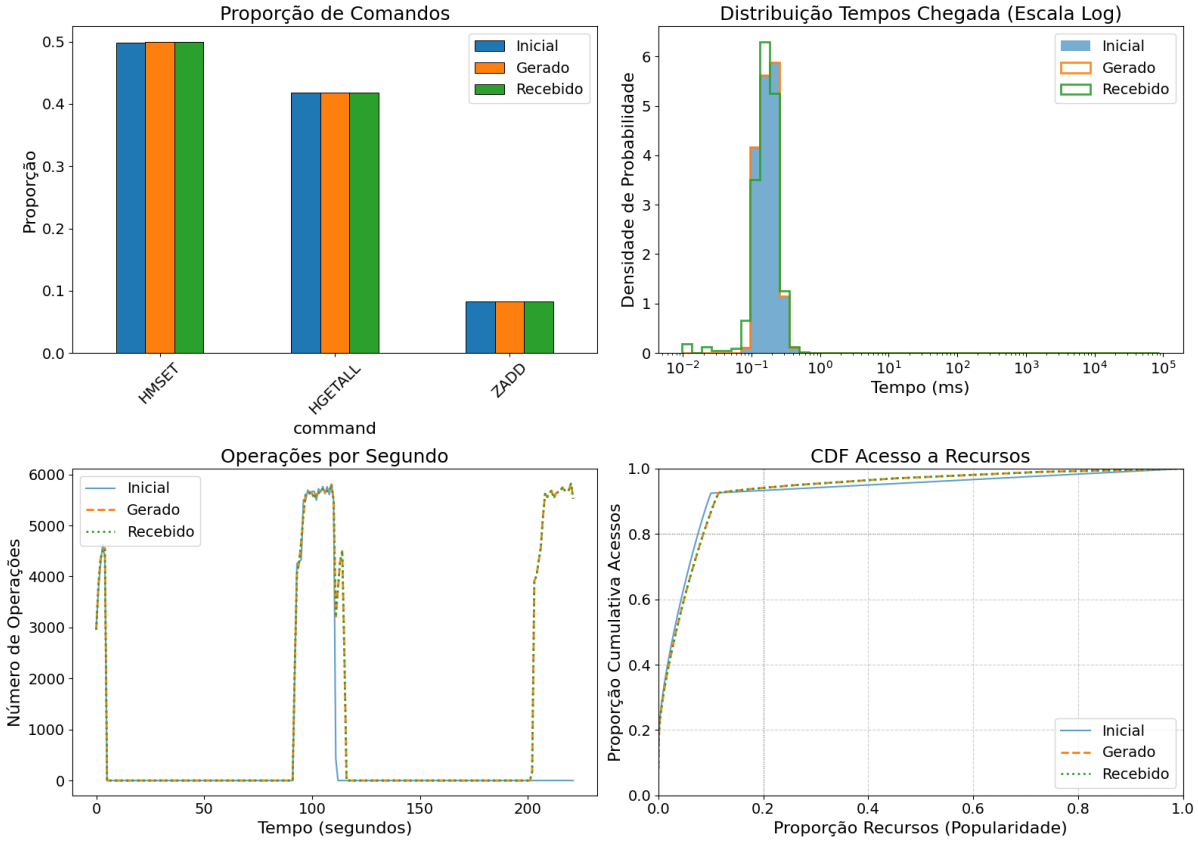
Os experimentos 4 (Figura 6.5) e 5 (Figura 6.6) validam as estratégias de mapeamento temporal, gerando cargas com o dobro da duração, 222 segundos. Em ambos os casos, os resultados estatísticos (Tabela 6.1) indicaram um alto grau de similaridade.

No Experimento 4 (`cyclic`), a contagem de comandos (Tabela 6.5) foi de 239.120 operações. A análise estatística indicou uma fidelidade extremamente alta, apresentando um V de Cramér de apenas 0.0018 para a distribuição de comandos. De forma similar, a análise para a distribuição de Tempos resultou em um V de 0.0053. Estes resultados confirmam que as proporções geradas, mesmo com o dobro do volume de dados, são praticamente idênticas às do rastro original em termos de tamanho de efeito.

Ao analisar a natureza estocástica do gerador, observa-se uma consistência robusta: tanto no Exp. 2 (V=0.0330) quanto no Exp. 4 (V=0.0018), o V de Cramér manteve-se substancialmente baixo. Isso demonstra que o perfil da carga foi preservado de forma eficaz, indicando que as variações introduzidas pela amostragem aleatória ou pela extensão da duração possuem uma magnitude desprezível.

No Experimento 5 (`stretch`), conforme a Tabela 6.6, o V de Cramér para Tempos (V=0.0045) também demonstrou alta similaridade, indicando que o alongamento das

Figura 6.5 – Resultados do Experimento 4 (Heatmap 1%, Dobro, *Cyclic*).



Fonte: O autor (2025).

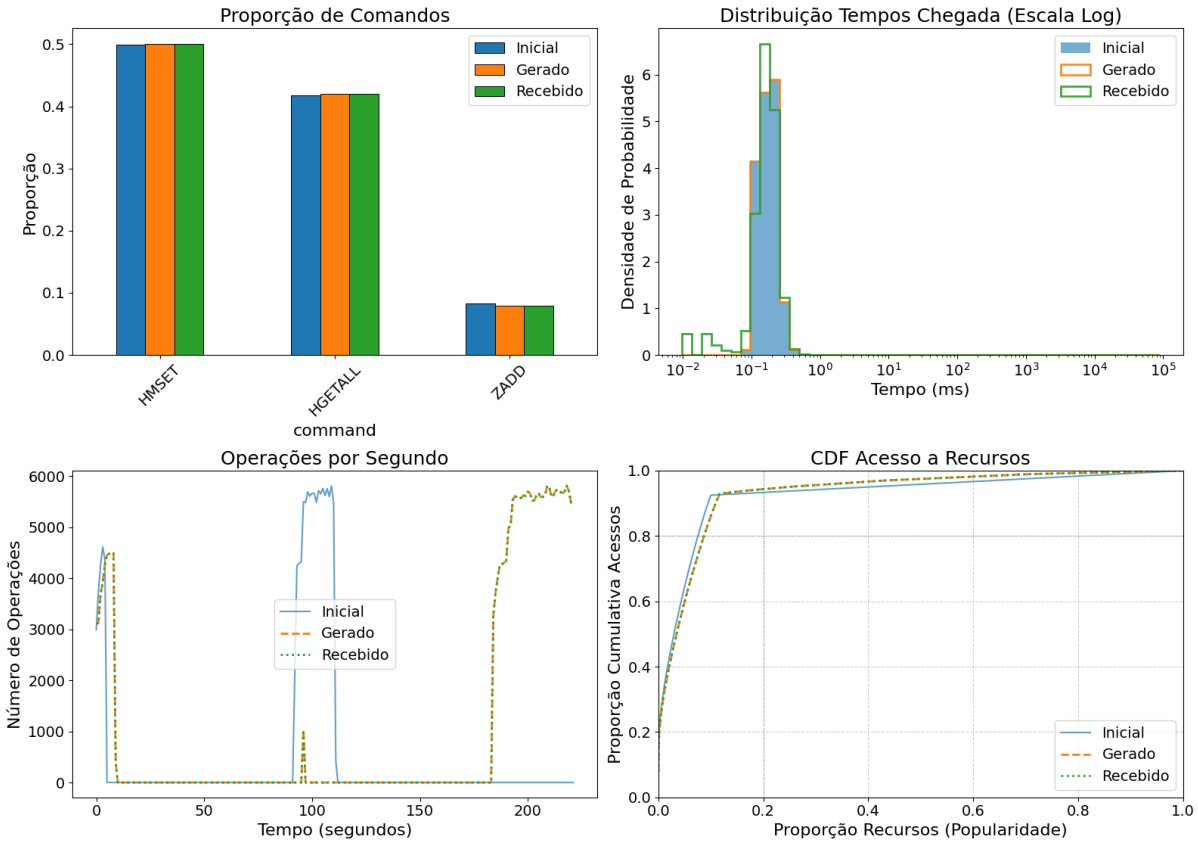
Tabela 6.5 – Contagem de Comandos: Experimento 4 (Heatmap 1% Cyclic).

Comando	Inicial (%)	Gerado (%)
HMSET	49.90%	49.97%
HGETALL	41.77%	41.80%
ZADD	8.33%	8.23%
TOTAL	**120.000**	**239.120**

Fonte: O autor (2025).

fases não distorceu o ritmo da carga. Para a métrica de recursos, ambas as estratégias replicaram o expoente `alpha` com baixa diferença (`cyclic`: 2.1623, `stretch`: 2.1543), com o `stretch` sendo quase idêntico ao original (2.1536).

Figura 6.6 – Resultados do Experimento 5 (Heatmap 1%, Dobro, *Stretch*).



Fonte: O autor (2025).

Tabela 6.6 – Contagem de Comandos: Experimento 5 (Heatmap 1% Stretch).

Comando	Inicial (%)	Gerado (%)
HMSET	49.90%	50.06%
HGETALL	41.77%	42.04%
ZADD	8.33%	7.90%
TOTAL	**120.000**	**239.197**

Fonte: O autor (2025).

Os gráficos “Operações por Segundo” (Figuras 6.5 e 6.6) demonstram visualmente como esses resultados foram alcançados, seja pela repetição do padrão (Exp. 4) ou pela sua expansão (Exp. 5).

6.3.5 Análise do Artefato de Precisão e Granularidade

Uma observação do gráfico “Distribuição Tempos Chegada” (por exemplo, na Figura 6.3) revela um comportamento consistente em todos os experimentos: a linha “Recebido” (verde) exibe uma pequena elevação na faixa entre 10^{-2} e 10^{-1} milissegundos, que não existe no rastro “Gerado” (laranja) e nem no “Inicial” (azul). Este fenômeno parece ser uma manifestação direta da latência de execução do Módulo Executor ao processar rajadas de operações muito próximas.

O processo pode ser explicado em duas etapas. Primeiro, os rastros “Gerado” e “Replay” (baseados no “Inicial” com 6 casas decimais) contêm eventos agendados com deltas de tempo extremamente pequenos (por exemplo, na faixa de 10^{-6} a 10^{-4} segundos), representando uma cadência teórica muito alta.

Na segunda etapa, o Módulo Executor tenta executar essas operações em seus tempos agendados. No entanto, a implementação possui uma latência inerente (devido ao enfileiramento de tarefas, despacho de *threads* e chamadas de sistema) que é superior a esses deltas teóricos. Essa discrepância, portanto, é a medida real, capturada pelo `MONITOR`, da cadência mínima que o Executor consegue de fato alcançar. Este comportamento reflete uma limitação da implementação em reproduzir rajadas de altíssima frequência, situando o tempo de despacho real por operação na faixa de 10^{-2} a 10^{-1} milissegundos.

6.4 CONCLUSÕES EXPERIMENTAIS

A análise dos cinco experimentos fornece um conjunto robusto de observações sobre o comportamento e a fidelidade da ferramenta Defigium. O experimento com `Heatmap 1%` (Exp. 2) indicou que o Módulo Gerador é capaz de aprender e reproduzir as características centrais da carga de trabalho. Isso é fundamentado quantitativamente: o gerador replicou a composição de comandos (V=0.0330) e o ritmo da carga (V=0.0170) com um tamanho de efeito muito pequeno. O modelo também se mostrou eficaz em aprender os *hotspots* da distribuição de recursos, como evidenciado pela alta proximidade do expoente `alpha` (2.1792 vs 2.1536).

No entanto, como detalhado na Seção 6.3.2, a análise identificou uma limitação do modelo probabilístico na replicação da cauda longa da distribuição, onde o número de chaves únicas geradas foi inferior ao do rastro original. A análise também revelou que a fidelidade do gerador é altamente sensível ao parâmetro de granularidade, como demonstrado pela divergência quantitativa em todas as métricas no Exp. 3 (V=0.0933, `alpha`=2.2463). Em contrapartida, o modelo mostrou-se robusto às estratégias de mapeamento temporal (Exp. 4 e 5), que mantiveram a fidelidade estatística mesmo com o dobro do volume de dados. Esta robustez foi comprovada especificamente pelos Experimentos 4 e 5, que validaram as estratégias de mapeamento temporal. Os

resultados demonstraram que tanto a repetição cíclica do padrão (estratégia *cyclic*) quanto o seu alongamento proporcional (estratégia *stretch*) foram capazes de duplicar a duração da carga de trabalho, mantendo a fidelidade estatística do rastro original. Conforme analisado na Seção 6.3.4, ambas as abordagens preservaram a proporção de comandos, o ritmo da carga e a distribuição de *hotspots*, confirmando a flexibilidade do gerador.

Em suma, os resultados experimentais sugerem que o Defigium se apresenta como uma ferramenta flexível, capaz de aprender e gerar cargas de trabalho que replicam com alta fidelidade os padrões de comandos, o ritmo temporal e a distribuição de *hotspots* de um rastro real, embora apresente alguma limitações.

7 CONCLUSÃO

Este trabalho apresentou o Defigium, uma ferramenta modular projetada para a geração de cargas de *benchmark* personalizadas, fundamentada na análise de rastros reais e na geração de cargas baseada em suas características observadas. A motivação principal residiu na constatação de que *benchmarks* padronizados, embora úteis para comparações gerais, frequentemente falham em capturar as características específicas e dinâmicas de cargas de trabalho em ambientes de produção reais, limitando a precisão das avaliações de desempenho.

A principal contribuição deste trabalho é a concepção e implementação de uma arquitetura flexível e extensível para o Defigium. Estruturada em um *pipeline* com módulos distintos para análise (*Parser*), geração (*Generator*) e execução (*Executor*), e utilizando padrões de projeto como *Strategy* e *Factory Method*, a ferramenta permite a fácil integração de novas técnicas de modelagem, formatos de *log* e sistemas-alvo. A implementação bilíngue, combinando Python para análise e geração com C++ para execução, buscou equilibrar flexibilidade de desenvolvimento com desempenho e precisão temporal.

A validação experimental, realizada com uma carga de trabalho complexa, permitiu quantificar a fidelidade da abordagem. O Módulo Executor exibiu alta fidelidade temporal na replicação de rastros. A estratégia de geração *Heatmap*, quando configurada com granularidade adequada, demonstrou ser capaz de aprender e replicar com alta similaridade as características-chave da carga original, incluindo os padrões temporais, a composição dos comandos e a distribuição de *hotspots*. Os experimentos também destacaram a importância da escolha correta da granularidade do modelo e comprovaram a robustez das estratégias de mapeamento temporal, que geraram cargas expandidas mantendo a fidelidade estatística do rastro original.

7.1 LIMITAÇÕES

Apesar dos resultados positivos, reconhece-se que o trabalho possui limitações. A validação experimental foi conduzida utilizando um único tipo de carga de trabalho (YCSB Workload A) e sistema-alvo (Redis). Embora a arquitetura do Defigium seja projetada para ser genérica, a fidelidade da ferramenta para outros domínios — como bancos de dados SQL, sistemas de arquivos ou serviços web — necessitaria de validação empírica específica com rastros e métricas relevantes para esses contextos.

Adicionalmente, a implementação atual apresenta limitações tanto no Módulo Gerador quanto no Executor. A estratégia `Heatmap`, embora eficaz na captura dos *hotspots* (o expoente `alpha`), representa apenas um dos possíveis modelos de geração. Como identificado na Seção 6.3.2, o modelo probabilístico atual apresentou uma limitação na replicação da cauda longa da distribuição de recursos, gerando um número

de chaves únicas inferior ao do rastro original. Da mesma forma, o Módulo Executor, conforme analisado na Seção 6.3.5, possui uma latência de despacho inerente que o impede de reproduzir rajadas de operações em frequências teóricas muito altas (abaixo de 10^{-2} ms). Modelos de geração mais sofisticados e otimizações no Executor poderiam, potencialmente, capturar dependências mais complexas e melhorar a precisão temporal. Por fim, a implementação atual focou na validação da arquitetura, não explorando otimizações de desempenho, como paralelização, relevantes para rastros de maior escala.

7.2 TRABALHOS FUTUROS

A natureza modular do Defigium abre diversas frentes para trabalhos futuros. Uma linha natural de expansão seria a implementação de novas estratégias de `Parser` e `Executor` para suportar uma gama mais ampla de sistemas-alvo e formatos de *log*, como MongoDB, PostgreSQL ou *logs* de acesso HTTP, ampliando assim a aplicabilidade da ferramenta.

Paralelamente, o desenvolvimento de estratégias de `Generator` alternativas representa um campo fértil para pesquisa. A incorporação de modelos baseados em Cadeias de Markov ou modelos de séries temporais, poderia permitir a captura de diferentes tipos de dependências temporais e estruturais presentes nos dados. Abordagens baseadas em aprendizado de máquina, como redes neurais recorrentes ou modelos generativos, também poderiam ser exploradas para a síntese de cargas.

Uma evolução arquitetural significativa seria a implementação nativa de execução distribuída. Embora a ferramenta atual utilize múltiplas *threads* em um único nó, a extensão do Módulo Executor para operar de forma coordenada em múltiplos nós físicos permitiria a simulação de cenários de carga massiva, superando as limitações de recursos de uma única máquina geradora.

Outra direção importante seria a condução de uma validação mais extensiva. Isso incluiria a utilização de uma variedade maior de cargas de trabalho reais, provenientes de diferentes domínios, e a comparação quantitativa da fidelidade do Defigium não apenas com o rastro original, mas também com outros geradores de carga existentes na literatura. Investigações sobre otimizações de desempenho para lidar com rastros de grande volume e a possível adição de uma interface gráfica para facilitar a configuração e a visualização dos modelos e resultados também constituem caminhos relevantes para a evolução da ferramenta.

Em suma, o Defigium demonstrou ser uma plataforma promissora para a geração de cargas de *benchmark* personalizadas. Sua arquitetura modular e os resultados quantitativos obtidos sugerem que a ferramenta é capaz de replicar com fidelidade os padrões de comandos, o ritmo temporal e a distribuição de *hotspots* de um rastro real, embora apresente limitações na replicação de acessos esparsos. Este potencial

para aprimorar a precisão das avaliações de desempenho de sistemas distribuídos, contribui para análises mais realistas e otimizações mais eficazes.

REFERÊNCIAS

ADEGBOYEGA, Abiola. Cloud Storage Workload Characterization: An Approach with Time-Series Analysis. *In*: 2024 IEEE 21st Consumer Communications & Networking Conference (CCNC). [*S.l.*: *s.n.*], 2024. P. 1090–1091.

AGHILI, Roozbeh; QIN, Qiaolin; LI, Heng; KHOMH, Foutse. **Understanding Web Application Workloads and Their Applications: Systematic Literature Review and Characterization**. [*S.l.*: *s.n.*], 2024. arXiv: 2409.12299 [cs.SE]. Disponível em: https://arxiv.org/abs/2409.12299.

ATIKOGLU, Berk; XU, Yuehai; FRACHTENBERG, Eitan; JIANG, Song; PALECZNY, Mike. Workload analysis of a large-scale key-value store. **SIGMETRICS Perform. Eval. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 40, n. 1, p. 53–64, jun. 2012. ISSN 0163-5999.

BAJABER, Fuad; SAKR, Sherif; BATARFI, Omar; ALTALHI, Abdulrahman; BARNAWI, Ahmed. Benchmarking big data systems: A survey. **Computer Communications**, v. 149, p. 241–251, 2020. ISSN 0140-3664.

BREWER, Eric. CAP Twelve years later: How the “rules” have Changed. **Computer**, v. 45, p. 23–29, fev. 2012.

CALZAROSSA, Maria Carla; MASSARI, Luisa; TESSERA, Daniele. Workload Characterization: A Survey Revisited. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 48, n. 3, fev. 2016. ISSN 0360-0300.

CHU, Xiaoyu; HOFSTÄTTER, Daniel; ILAGER, Shashikant; TALLURI, Sacheendra; KAMPERT, Duncan; PODAREANU, Damian; DUPLYAKIN, Dmitry; BRANDIC, Ivona; IOSUP, Alexandru. **Generic and ML Workloads in an HPC Datacenter: Node Energy, Job Failures, and Node-Job Analysis**. [*S.l.*: *s.n.*], 2024. arXiv: 2409.08949 [cs.DC]. Disponível em: https://arxiv.org/abs/2409.08949.

CLAUSET, Aaron; SHALIZI, Cosma Rohilla; NEWMAN, M. E. J. Power-Law Distributions in Empirical Data. **SIAM Review**, Society for Industrial Applied Mathematics (SIAM), v. 51, n. 4, p. 661–703, nov. 2009. ISSN 1095-7200.

COHEN, J. The Test of Goodness of Fit and Contingency Tables. *In*: STATISTICAL Power Analysis for the Behavioral Sciences. [*S.l.*]: Taylor & Francis, 2013. ISBN 9781134742707.

COOPER, Brian F.; SILBERSTEIN, Adam; TAM, Erwin; RAMAKRISHNAN, Raghu; SEARS, Russell. Benchmarking cloud serving systems with YCSB. *In*: PROCEEDINGS of the 1st ACM Symposium on Cloud Computing. Indianapolis, Indiana, USA: Association for Computing Machinery, 2010. (SoCC '10), p. 143–154.

COULOURIS, George; DOLLIMORE, Jean; KINDBERG, Tim; BLAIR, Gordon. **Distributed Systems: Concepts and Design**. 5th. USA: Addison-Wesley Publishing Company, 2011. ISBN 0132143011.

DELIMITROU, Christos; KOZYRAKIS, Christos. Accurate modeling and generation of storage I/O for datacenter workloads. **ACM Transactions on Storage**, v. 7, n. 3, 8:1–8:23, 2011.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design Patterns: Elements of Reusable Object-Oriented Software**. [*S.l.*]: Pearson Education, 1994. (Addison-Wesley Professional Computing Series). ISBN 9780321700698.

GHANDEHARIZADEH, Shahram; HUANG, Haoyu. Hoagie: A Database and Workload Generator using Published Specifications. *In*: 2018 IEEE International Conference on Big Data (Big Data). [*S.l.*: *s.n.*], 2018. P. 3847–3852.

GMACH, D.; ROLIA, Jerry; CHERKASOVA, Ludmila; KEMPER, Alfons. Workload Analysis and Demand Prediction of Enterprise Data Center Applications. *In*: PROCEEDINGS of the 2007 IEEE International Symposium on Workload Characterization, IISWC. [*S.l.*: *s.n.*], out. 2007. P. 171–180.

JAIN, Raj. **The Art of Computer Systems Performance Analysis: Techniques For Experimental Design, Measurement, Simulation, and Modeling, NY: Wiley**. [*S.l.*]: Wiley, abr. 1991. P. 685. ISBN 0471503361.

JANECEK, Madeline; EZZATI-JIVAN, Naser; AZHARI, Seyed Vahid. Container Workload Characterization Through Host System Tracing. *In*: 2021 IEEE International Conference on Cloud Engineering (IC2E). [*S.l.*: *s.n.*], 2021. P. 9–19.

KHAN, Arijit; YAN, Xifeng; TAO, Shu; ANEROUSIS, Nikos. Workload characterization and prediction in the cloud: A multiple time series approach. *In*: 2012 IEEE Network Operations and Management Symposium. [*S.l.*: *s.n.*], 2012. P. 1287–1294.

KIM, Keunsoo; LEE, Changmin; JUNG, Jung Ho; RO, Won Woo. Workload synthesis: Generating benchmark workloads from statistical execution profile. *In*: 2014 IEEE International Symposium on Workload Characterization (IISWC). [*S.l.*: *s.n.*], 2014. P. 120–129.

KISTOWSKI, Jóakim v.; ARNOLD, Jeremy A.; HUPPLER, Karl; LANGE, Klaus-Dieter; HENNING, John L.; CAO, Paul. How to Build a Benchmark. *In*: PROCEEDINGS of the 6th ACM/SPEC International Conference on Performance Engineering. Austin, Texas, USA: Association for Computing Machinery, 2015. (ICPE '15), p. 333–336.

LIN, Mingfeng; LUCAS, Henry; SHMUELI, Galit. Too Big to Fail: Large Samples and the p-Value Problem. **Information Systems Research**, v. 24, p. 906–917, dez. 2013.

PALLIS, George; KAZAKOS, Vasileios; DASKALOU, Christos. A Study on Workload Characterization for a Web Proxy Server. *In*: PROCEEDINGS OF THE 2003 INTERNATIONAL CONFERENCE ON COMPUTER NETWORKS AND MOBILE COMPUTING (ICCNMC). [*S.l.*]: IEEE, 2003. P. 256–261.

PURANDARE, Devashish R.; BITTMAN, Daniel; MILLER, Ethan L. Analysis and workload characterization of the CERN EOS storage system. *In*: PROCEEDINGS of the Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems. Rennes, France: Association for Computing Machinery, 2022. (CHEOPS '22), p. 1–7.

RAMAKRISHNAN, Raghu; GEHRKE, Johannes. **Database Management Systems**. 2nd. USA: McGraw-Hill, Inc., 2000. ISBN 0072440422.

REHRMANN, Robin; BINNIG, Carsten; BÖHM, Alexander; KIM, Kihong; LEHNER, Wolfgang. Sharing opportunities for OLTP workloads in different isolation levels. **Proc. VLDB Endow.**, VLDB Endowment, v. 13, n. 10, p. 1696–1708, jun. 2020. ISSN 2150-8097.

SFAKIANAKIS, Yannis; KANELLOU, Eleni; MARAZAKIS, Manolis; BILAS, Angelos. Trace-Based Workload Generation and Execution. *In*: EURO-PAR 2021: Parallel Processing: 27th International Conference on Parallel and Distributed Computing,

Lisbon, Portugal, September 1–3, 2021, Proceedings. Lisbon, Portugal: Springer-Verlag, 2021. P. 37–54.

SIMAKOV, Nikolay A.; WHITE, Joseph P.; DELEON, Robert L.; GALLO, Steven M.; JONES, Matthew D.; PALMER, Jeffrey T.; PLESSINGER, Benjamin; FURLANI, Thomas R. **A Workload Analysis of NSF's Innovative HPC Resources Using XDMoD**. [*S.l.*: *s.n.*], 2018. arXiv: 1801.04306 [cs.DC]. Disponível em: https://arxiv.org/abs/1801.04306.

SIRIN, Utku; AILAMAKI, Anastasia. Micro-architectural analysis of OLAP: limitations and opportunities. **Proc. VLDB Endow.**, VLDB Endowment, v. 13, n. 6, p. 840–853, fev. 2020. ISSN 2150-8097.

SLADOJEVIĆ, Vladimir; FRISCHBIER, Sebastian; ECHLER, Alexander; PAIC, Mario; MARGARA, Alessandro. Deriving a realistic workload model to simulate high-volume financial data feeds for performance benchmarking. *In*: PROCEEDINGS of the 16th ACM International Conference on Distributed and Event-Based Systems. Copenhagen, Denmark: Association for Computing Machinery, 2022. (DEBS '22), p. 126–131.

TALLURI, Sacheendra; ŁUSZCZAK, Alicja; ABAD, Cristina L.; IOSUP, Alexandru. Characterization of a Big Data Storage Workload in the Cloud. *In*: PROCEEDINGS of the 2019 ACM/SPEC International Conference on Performance Engineering. Mumbai, India: Association for Computing Machinery, 2019. (ICPE '19), p. 33–44.

TANENBAUM, Andrew S.; STEEN, Maarten van. **Distributed Systems: Principles and Paradigms**. 2nd. North Charleston, SC, USA: CreateSpace Independent Publishing Platform, 2016. ISBN 153028175X.

TPC. **TPC-C Standard**. Disponível em: http://www.tpc.org/tpcc/. Acesso em: 2 jun. 2025.

TPC. **TPC-H Standard**. Disponível em: http://www.tpc.org/tpch/. Acesso em: 2 jun. 2025.

XIANG, Yuxing; LI, Xue; QIAN, Kun; YU, Wenyuan; ZHAI, Ennan; JIN, Xin. **ServeGen: Workload Characterization and Generation of Large Language Model Serving in Production**. [*S.l.*: *s.n.*], 2025. arXiv: 2505.09999 [cs.DC]. Disponível em: https://arxiv.org/abs/2505.09999.

Apêndices

APÊNDICE A – ARTIGO EM FORMATO SBC

Defigium: Uma Ferramenta Modular para Geração de Cargas de Benchmark Baseadas em Rastros Reais

Felipe Backes Kettl[1], **Odorico Machado Mendizabal**[1], **Marcos André Braz Vaz**[1]

[1]Departamento de Informática e Estatística – Universidade Federal de Santa Catarina
Florianópolis – SC – Brasil

felipe.kettl@grad.ufsc.br, {odorico.mendizabal, braz.vaz}@ufsc.br

Abstract. *Performance evaluation of distributed systems requires realistic workloads, as standardized benchmarks often fail to capture the temporal complexity and access patterns of production environments. This paper presents Defigium, a modular tool for generating customized benchmark workloads grounded in the analysis of real traces. The proposed architecture implements a three-stage pipeline (Analysis, Generation, and Execution), utilizing a hybrid approach to combine flexibility in probabilistic modeling with temporal precision in load injection. Experimental validation on a Redis database using YCSB traces demonstrated that the Heatmap generation strategy replicated command composition and temporal rhythm with high statistical fidelity, Cramér's V < 0.05, preserving data popularity distributions.*

Resumo. *A avaliação de desempenho de sistemas distribuídos requer cargas de trabalho realistas, uma vez que benchmarks padronizadas frequentemente falham em capturar a complexidade temporal e os padrões de acesso de ambientes de produção. Este artigo apresenta o Defigium, uma ferramenta modular para geração de cargas de benchmark personalizadas fundamentadas na análise de rastros reais. A arquitetura proposta implementa um pipeline de três estágios (Análise, Geração e Execução), utilizando uma abordagem híbrida para aliar flexibilidade na modelagem probabilística à precisão temporal na injeção de carga. A validação experimental em um banco Redis, utilizando rastros do YCSB, demonstrou que a estratégia de geração Heatmap replicou a composição de comandos e o ritmo temporal com alta fidelidade estatística, V de Cramér < 0,05, preservando a distribuição de popularidade dos dados.*

1. Introdução

A avaliação e otimização de sistemas distribuídos dependem intrinsecamente da qualidade das cargas de trabalho utilizadas nos testes. Ferramentas de *benchmark* consolidadas, como o YCSB (Yahoo! Cloud Serving Benchmark) [Cooper et al. 2010], fornecem uma base essencial para comparações padronizadas entre diferentes sistemas de banco de dados. No entanto, abordagens baseadas em perfis sintéticos pré-definidos frequentemente falham em capturar a complexidade estocástica e a variabilidade dos padrões de uso observados em ambientes de produção reais. Características críticas, como comportamento de rajada, sazonalidade e pontos de acesso frequente em dados específicos, são muitas vezes simplificadas em modelos matemáticos uniformes [Atikoglu et al. 2012].

A literatura aponta que a modelagem de cargas realistas é crucial para contextos de domínio específico, como o mercado financeiro ou redes sociais [Sladojević et al. 2022].

Isso motiva o desenvolvimento de ferramentas capazes de sintetizar cargas a partir de rastros (*traces*) de operação reais. Contudo, muitas soluções existentes sofrem de limitações arquiteturais: ou são acopladas a sistemas-alvo específicos, dificultando sua reutilização, ou carecem de flexibilidade para experimentar diferentes modelos estatísticos de geração sem reescrever o núcleo da ferramenta.

Neste contexto, este trabalho apresenta o Defigium, uma ferramenta modular e extensível para a geração de cargas de *benchmark*, fundamentada na análise de logs reais. A principal contribuição é uma arquitetura em *pipeline* que utiliza uma abordagem bilíngue (Python e C++) para equilibrar a flexibilidade na modelagem estatística com a precisão temporal necessária para a injeção de carga em alta performance.

2. Trabalhos Relacionados

A caracterização e geração de cargas de trabalho são áreas amplamente estudadas, com abordagens que variam desde a análise estatística descritiva até o uso de aprendizado de máquina. A Tabela 1 resume as principais metodologias identificadas na literatura.

Tabela 1. Comparação Resumida de Metodologias de Caracterização e Geração.

Ref.	Domínio	Técnica Principal / Foco
[Delimitrou and Kozyrakis 2011]	Armazenamento (E/S)	Cadeias de Markov Hierárquicas para localidade espacial/temporal.
[Talluri et al. 2019]	Spark (Big Data)	Análise estatística profunda (ECDFs, Hurst) de cargas de longa duração.
[Purandare et al. 2022]	Sist. Arquivos (EOS)	Inferência de operações a partir de metadados em traces científicos.
[Adegboyega 2024]	Nuvem	Séries Temporais (GARCH) para modelagem de *bursts*.
[Gmach et al. 2007]	Data Center	Extração de padrões periódicos (ciclos) via análise espectral.
[Sfakianakis et al. 2021]	Data Center	Ferramenta *Tracie*: geração executável baseada em amostragem.
[Kim et al. 2014]	Microarquitetura	Síntese baseada em contadores de hardware e *solvers*.
[Ghandeharizadeh and Huang 2018]	BD Genérico	Geração baseada em especificações publicadas, sem rastro original.
[Xiang et al. 2025]	Inferência LLM	Modelagem composta de clientes para *LLM Serving*.

[Calzarossa et al. 2016] destacam a importância de compreender as propriedades das cargas para engenharia de desempenho. Em armazenamento, [Delimitrou and Kozyrakis 2011] propuseram o uso de Cadeias de Markov para modelar localidade espacial e temporal em *datacenters*. Já [Talluri et al. 2019] realizaram caracterizações estatísticas profundas em cargas do Spark, enquanto [Gmach et al. 2007] focaram na extração de padrões periódicos em *datacenters*.

No campo da síntese, [Kim et al. 2014] abordaram a geração baseada em perfis de execução de *hardware*. Ferramentas como o *Tracie* [Sfakianakis et al. 2021] focam na geração executável a partir de rastros, mas muitas vezes limitam-se a domínios específicos. Diferentemente de abordagens que dependem apenas de especificações publicadas [Ghandeharizadeh and Huang 2018], o Defigium propõe um fluxo completo que vai do *log* bruto à execução, com uma arquitetura agnóstica ao sistema-alvo.

3. Arquitetura Defigium

O Defigium foi concebido como um sistema modular estruturado em um *pipeline* de processamento de três estágios, conforme ilustrado na Figura 1. Essa arquitetura promove o desacoplamento e a especialização das responsabilidades, utilizando internamente um Formato de Evento Intermediário (FEI) para padronizar a comunicação entre os módulos.

A decisão de arquitetar o sistema em estágios desacoplados fundamenta-se em três pilares: (1) Independência Tecnológica, permitindo que a modelagem ocorra no rico

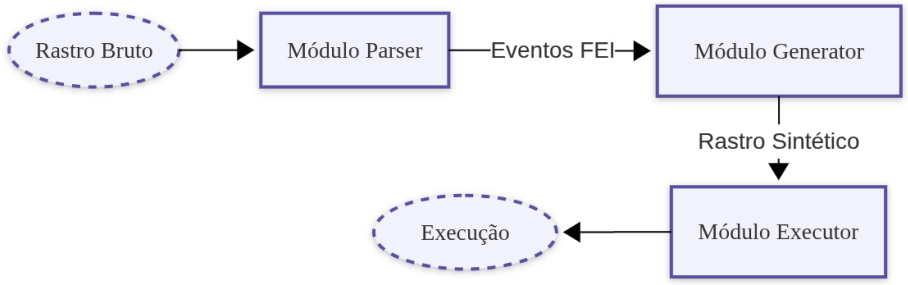


Figura 1. Visão geral conceitual do pipeline da ferramenta Defigium.

ecossistema Python enquanto a execução ocorre em C++; (2) Abstração Semântica, onde o FEI isola a lógica de geração dos detalhes do protocolo do sistema-alvo; e (3) Reprodutibilidade, garantida pela materialização do rastro sintético em arquivo.

3.1. Módulo de Análise (Parser)

Este módulo atua como um tradutor bidirecional. Sua função primária é interpretar rastros brutos (logs) de sistemas reais, como Redis, servidores Web e SQL, e convertê-los para o formato FEI. O objeto FEI normaliza atributos essenciais como `timestamp`, `op_type` (tipo de operação), `target` (chave ou recurso acessado) e metadados adicionais. A implementação utiliza o padrão de projeto *Factory Method* para instanciar analisadores específicos para cada formato de log suportado.

3.2. Módulo de Geração (Generator)

Implementado em Python, este é o núcleo lógico da ferramenta. Ele aplica uma estratégia selecionada, padrão *Strategy*, para analisar as características do rastro original e gerar uma nova sequência de eventos. Duas estratégias foram implementadas:

- Replay: Uma estratégia simples que replica a sequência exata de eventos do rastro original, servindo como linha de base para validação.
- Heatmap: Um modelo probabilístico que divide o tempo de execução em intervalos percentuais. Na fase de caracterização, o modelo aprende, para cada intervalo, a Distribuição de Probabilidade dos tipos de operação, a popularidade dos recursos associada a cada operação e a distribuição dos tempos entre chegadas. Na fase de síntese, o gerador utiliza amostragem condicional hierárquica para preservar as correlações observadas, como a probabilidade de acessar uma chave específica dado um comando de escrita naquele intervalo de tempo.

3.3. Módulo de Execução (Executor)

Implementado em C++17 para garantir alta performance e minimizar a latência de injeção, este módulo consome o rastro sintético e o traduz em operações contra o sistema-alvo. A arquitetura adota o modelo Gerente-Trabalhador.

A *thread* Gerente é responsável por ler o arquivo de rastro e agendar as operações em uma fila de prioridade baseada no tempo absoluto de disparo. Múltiplas *threads* Trabalhadoras consomem essa fila, utilizando `std::chrono` para aguardar o momento exato de execução. Cada trabalhador mantém uma conexão persistente com o sistema-alvo, permitindo a simulação de múltiplos clientes concorrentes.

4. Metodologia Experimental

A validação da ferramenta foi conduzida através de um ciclo de quatro etapas, conforme ilustrado na Figura 2. O fluxo inicia-se com a captura de uma carga de trabalho Inicial, seguida pela utilização do Defigium para a criação de um rastro Gerado. Posteriormente, este rastro é executado contra o sistema-alvo para produzir o rastro Recebido, finalizando com a análise comparativa estatística entre as três instâncias.

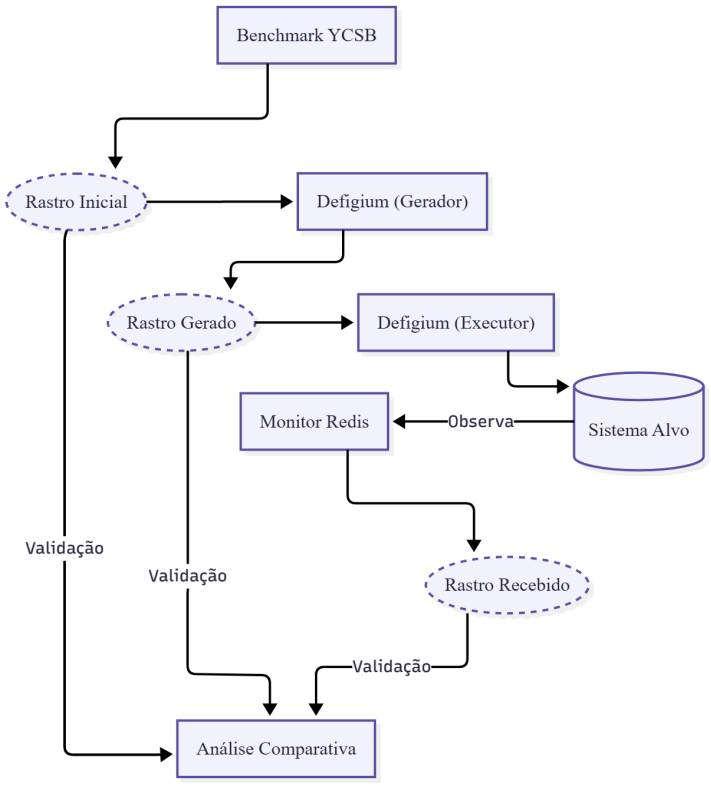


Figura 2. Fluxo de validação experimental comparando os três estágios do rastro.

4.1. Carga de Trabalho e Cenário

Utilizou-se um rastro do benchmark YCSB *Workload A* (50% leitura, 50% escrita) sobre um banco de dados Redis. Este perfil foi escolhido por sua ampla utilização na indústria. O rastro foi estruturado intencionalmente de forma não estacionária: uma rajada inicial de carga (inserções), seguida de um longo período de inatividade (87 segundos) e uma rajada final de execução (leituras/atualizações). Essa estrutura visa testar a capacidade da ferramenta em lidar com cargas intermitentes e preservar períodos de silêncio sem distorcer a densidade temporal.

4.2. Métricas de Avaliação

A fidelidade foi avaliada comparando as distribuições de Comandos, Tempos de Chegada e Acesso a Recursos. Devido ao grande volume de dados ($N > 100.000$), optou-se por não utilizar o P-valor, que se torna excessivamente sensível a diferenças triviais

[Lin et al. 2013]. As métricas adotadas foram: (1) O V de Cramér, que mede o tamanho do efeito para diferenças entre distribuições. Valores $V < 0.1$ indicam tamanho de efeito trivial [Cohen 2013]; (2) O Expoente Alpha, obtido através do ajuste de lei de potência (*Power Law*) sobre a curva de popularidade de chaves, indicando a inclinação da cauda da distribuição Zipfiana [Clauset et al. 2009].

5. Resultados e Análise

A avaliação experimental foi estruturada em cinco cenários distintos, delineados para isolar e validar componentes específicos da arquitetura, bem como testar a robustez do modelo probabilístico sob diferentes condições. Os experimentos definidos foram:

- Exp. 1 (Replay Simples): Utiliza a estratégia de replicação direta para validar a precisão temporal do Módulo Executor em C++.
- Exp. 2 (Heatmap 1%): Aplica o modelo probabilístico com intervalos finos. Serve como o cenário ideal para validar a capacidade de aprendizado da distribuição.
- Exp. 3 (Heatmap 50%): Utiliza granularidade grosseira. Concebido como teste de sensibilidade para demonstrar o impacto de agrupar fases temporais distintas.
- Exp. 4 (Cyclic) e Exp. 5 (Stretch): Avaliam estratégias de extensão temporal, gerando cargas com o dobro da duração (222s), para verificar a manutenção da fidelidade estatística.

A Tabela 2 apresenta o consolidado estatístico, comparando o rastro Gerado com o Inicial.

Tabela 2. Validação Estatística (Inicial vs. Gerado).

Experimento	V (Comandos)	V (Tempos)	Alpha (Inicial)	Alpha (Gerado)
Replay Simples	0.0000	0.0000	2.1536	2.1536
Heatmap 1% (Original)	0.0330	0.0170	2.1536	2.1792
Heatmap 50% (Original)	0.0933	0.0497	2.1536	2.2463
Heatmap 1% (Cyclic)	0.0018	0.0053	2.1536	2.1623
Heatmap 1% (Stretch)	0.0076	0.0045	2.1536	2.1543

5.1. Validação de Fidelidade (Replay e Heatmap)

O Experimento 1 (Replay) confirmou a precisão do Módulo Executor. A contagem de comandos foi idêntica à original e o V de Cramér foi 0,0000, validando a capacidade do executor C++ de respeitar o agendamento temporal com precisão na ordem de milissegundos.

O Experimento 2 avaliou a estratégia Heatmap com granularidade fina. A Figura 3 ilustra a comparação visual. O modelo replicou a composição de comandos com alta fidelidade ($V = 0,0330$) e capturou o ritmo da carga com precisão ($V = 0,0170$). O expoente Alpha gerado (2,1792) aproximou-se muito do original (2,1536), indicando que o padrão de acesso aos recursos foi preservado corretamente.

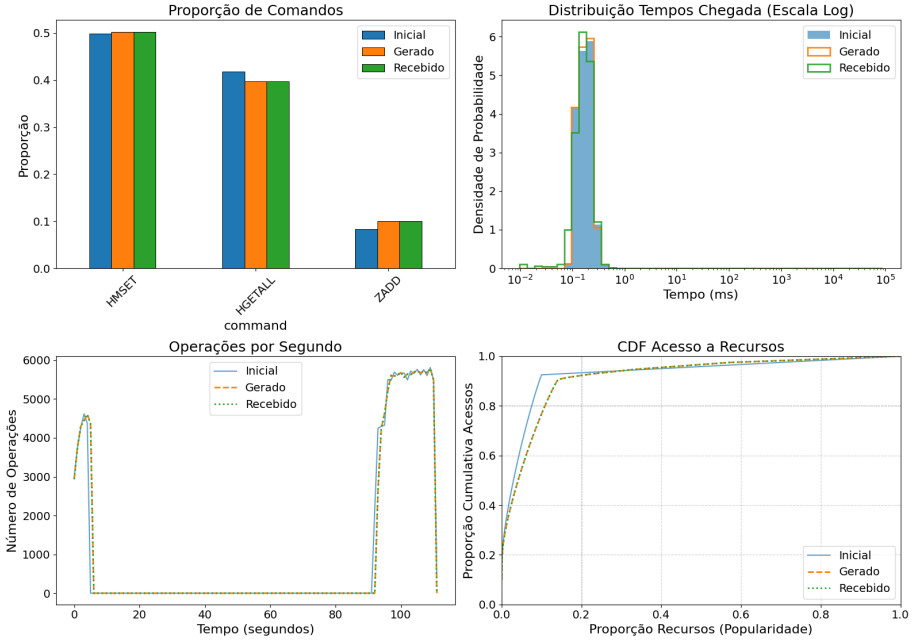


Figura 3. Resultados do Experimento 2 (Heatmap 1%): Alta fidelidade visual nas quatro métricas.

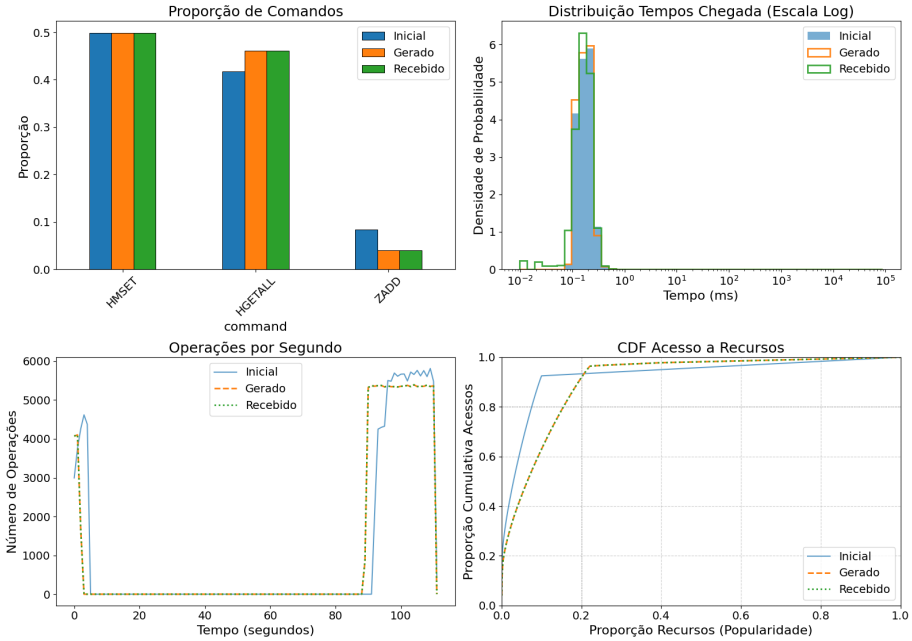


Figura 4. Resultados do Experimento 3 (Heatmap 50%): Degradação visual da fidelidade devido à granularidade grossa.

5.2. Impacto da Granularidade

O Experimento 3 (Figura 4) evidenciou a sensibilidade do modelo ao parâmetro de intervalo. Ao dividir a execução em apenas dois grandes blocos (50%), o modelo forçou o agrupamento estatístico de fases distintas, a rajada inicial e o período de ociosidade. Como resultado, houve degradação significativa na fidelidade: o V de Cramér para comandos subiu para 0,0933 e a curva de popularidade teve um Alpha mais distante. Isso confirma a necessidade de uma granularidade fina para capturar transições de fase em cargas não estacionárias.

5.3. Estratégias de Mapeamento Temporal

Os experimentos 4 e 5 validaram a capacidade do gerador de criar cargas de longa duração. No caso Cyclic (Figura 5), que repete o padrão aprendido, o V de Cramér manteve-se extremamente baixo $(0,0018)$, confirmando que a duplicação do volume de dados não degradou a reprodução.

De forma similar, o método Stretch (Figura 6) preservou o ritmo ($V = 0,0045$), demonstrando que o Defigium pode estender uma carga curta para testes de estresse prolongados sem perder as características de distribuição originais.

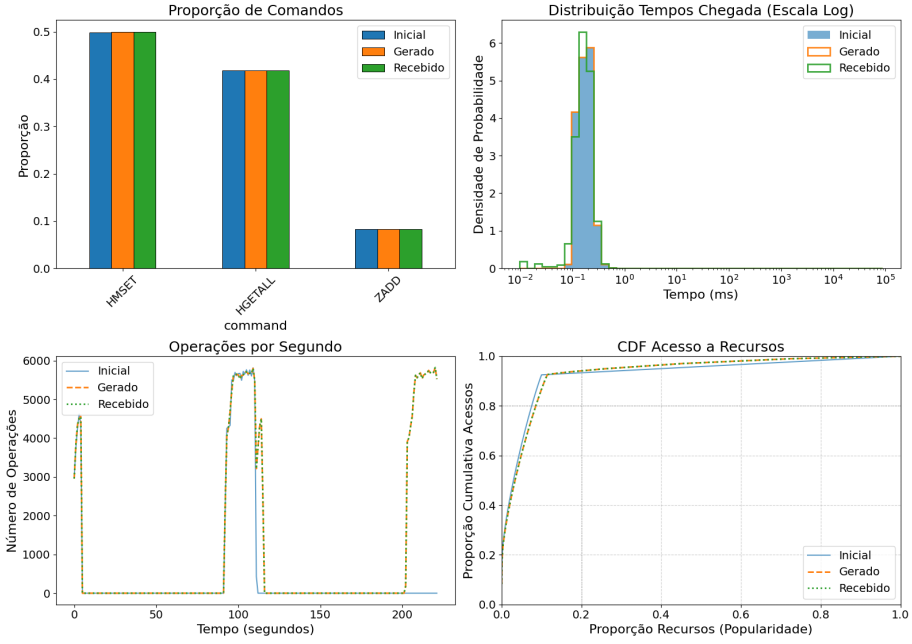


Figura 5. Resultados do Experimento 4 (Cyclic): Preservação do padrão em carga duplicada.

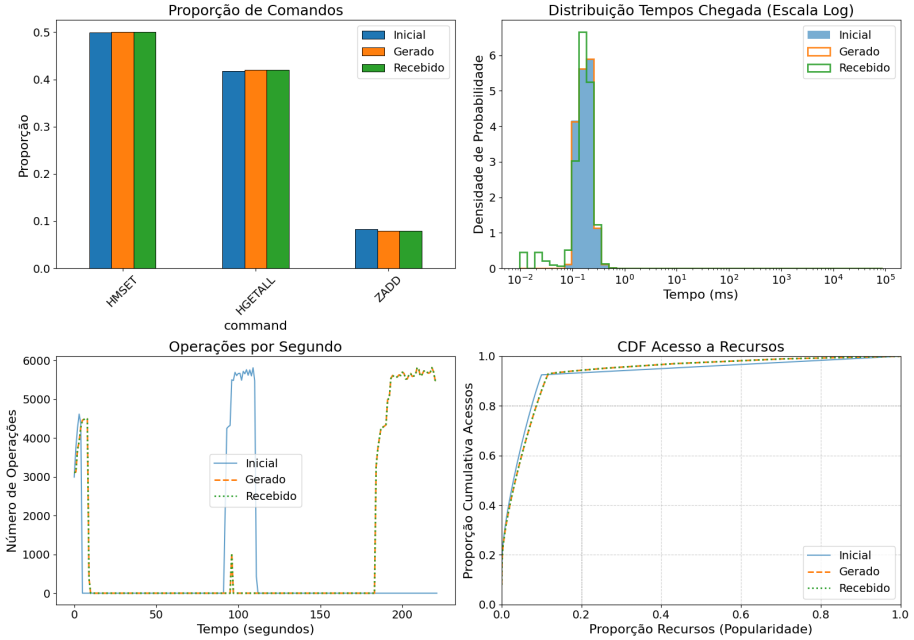


Figura 6. Resultados do Experimento 5 (Stretch): Preservação do ritmo com duração estendida.

6. Conclusão

O desenvolvimento do Defigium atendeu à necessidade de ferramentas capazes de capturar a complexidade e os padrões dinâmicos de uso de ambientes de produção, uma lacuna frequente em benchmarks padronizados. A arquitetura proposta, estruturada em um pipeline modular, demonstrou ser eficaz ao combinar a flexibilidade de análise do Python com a precisão temporal de execução do C++.

A validação experimental confirmou que a estratégia de geração Heatmap, quando configurada com a granularidade adequada, é capaz de aprender e replicar características fundamentais da carga de trabalho, incluindo a composição de comandos, o ritmo de chegada e a distribuição de hotspots. Além disso, a ferramenta mostrou-se robusta na aplicação de estratégias de mapeamento temporal, permitindo a geração de cargas de longa duração a partir de rastros curtos sem perda significativa de fidelidade estatística.

Contudo, reconhece-se que a abordagem atual possui limitações. O modelo probabilístico apresentou dificuldades na replicação da cauda longa da distribuição de recursos, especificamente nos acessos esparsos, e o Módulo Executor exibiu uma latência inerente que limita a reprodução de rajadas de altíssima frequência, situadas abaixo de um centésimo de milissegundo.

7. Trabalhos Futuros

A natureza modular do Defigium abre diversas frentes para a evolução da pesquisa. Uma evolução arquitetural significativa seria a implementação nativa de execução distribuída.

Embora a ferramenta atual utilize múltiplas threads em um único nó, a extensão do Módulo Executor para operar de forma coordenada em múltiplos nós físicos permitiria a simulação de cenários de carga massiva, superando as limitações de recursos de uma única máquina geradora. Tal orquestração poderia ser viabilizada através da integração com bibliotecas de passagem de mensagem, como MPI, para sincronizar o disparo de eventos particionados entre diferentes instâncias.

Paralelamente, o desenvolvimento de estratégias de geração alternativas representa um campo fértil. A incorporação de modelos baseados em Cadeias de Markov ou séries temporais permitiria a captura de dependências estruturais nos dados, enquanto abordagens baseadas em aprendizado de máquina, como redes neurais recorrentes, poderiam ser exploradas para a síntese de cargas mais complexas. Adicionalmente, a implementação de novas estratégias de análise e execução para suportar uma gama mais ampla de sistemas-alvo, como bancos de dados relacionais e logs de acesso HTTP, ampliaria a aplicabilidade da ferramenta para novos contextos.

Por fim, outra direção importante seria a condução de uma validação mais extensiva, incluindo a utilização de uma variedade maior de cargas de trabalho reais provenientes de diferentes domínios e a comparação quantitativa da fidelidade do Defigium com outros geradores de carga existentes na literatura.

Referências

Adegboyega, A. (2024). Cloud storage workload characterization: An approach with time-series analysis. In *2024 IEEE 21st Consumer Communications & Networking Conference (CCNC)*, pages 1090–1091.

Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. (2012). Workload analysis of a large-scale key-value store. *SIGMETRICS Perform. Eval. Rev.*, 40(1):53–64.

Calzarossa, M. C., Massari, L., and Tessera, D. (2016). Workload characterization: A survey revisited. *ACM Comput. Surv.*, 48(3).

Clauset, A., Shalizi, C. R., and Newman, M. E. J. (2009). Power-law distributions in empirical data. *SIAM Review*, 51(4):661–703.

Cohen, J. (2013). The test of goodness of fit and contingency tables. In *Statistical Power Analysis for the Behavioral Sciences*. Taylor & Francis.

Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. (2010). Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154.

Delimitrou, C. and Kozyrakis, C. (2011). Accurate modeling and generation of storage i/o for datacenter workloads. *ACM Transactions on Storage*, 7(3):8:1–8:23.

Ghandeharizadeh, S. and Huang, H. (2018). Hoagie: A database and workload generator using published specifications. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 3847–3852.

Gmach, D., Rolia, J., Cherkasova, L., and Kemper, A. (2007). Workload analysis and demand prediction of enterprise data center applications. In *Proceedings of the 2007 IEEE International Symposium on Workload Characterization, IISWC*, pages 171–180.

Kim, K., Lee, C., Jung, J. H., and Ro, W. W. (2014). Workload synthesis: Generating benchmark workloads from statistical execution profile. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 120–129.

Lin, M., Lucas, H., and Shmueli, G. (2013). Too big to fail: Large samples and the p-value problem. *Information Systems Research*, 24:906–917.

Purandare, D. R., Bittman, D., and Miller, E. L. (2022). Analysis and workload characterization of the cern eos storage system. In *Proceedings of the Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*, CHEOPS '22, page 1–7, New York, NY, USA. Association for Computing Machinery.

Sfakianakis, Y., Kanellou, E., Marazakis, M., and Bilas, A. (2021). Trace-based workload generation and execution. In *Euro-Par 2021: Parallel Processing: 27th International Conference on Parallel and Distributed Computing, Lisbon, Portugal, September 1–3, 2021, Proceedings*, page 37–54, Berlin, Heidelberg. Springer-Verlag.

Sladojević, V., Frischbier, S., Echler, A., Paic, M., and Margara, A. (2022). Deriving a realistic workload model to simulate high-volume financial data feeds for performance benchmarking. In *Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems*, DEBS '22, page 126–131, New York, NY, USA. Association for Computing Machinery.

Talluri, S., Łuszczak, A., Abad, C. L., and Iosup, A. (2019). Characterization of a big data storage workload in the cloud. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ICPE '19, page 33–44, New York, NY, USA. Association for Computing Machinery.

Xiang, Y., Li, X., Qian, K., Yu, W., Zhai, E., and Jin, X. (2025). Servegen: Workload characterization and generation of large language model serving in production.

APÊNDICE B – CÓDIGO FONTE

O código fonte e os conjuntos de dados desenvolvidos durante esta pesquisa estão disponíveis no repositório institucional no seguinte endereço:

https://codigos.ufsc.br/f.kettl/Defigium