Defigium: A Modular Framework for Trace-Based Workload Generation

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

[1]Federal University of Santa Catarina (UFSC)
Florianópolis – SC – Brazil

felipe.kettl@ufsc.br,bernardo.c.fraga@grad.ufsc.br,

braz.vaz@ufsc.br,odorico.mendizabal@ufsc.br

Abstract. *Performance testing for distributed systems depends heavily on the accurate reproduction of realistic workloads. Across domains such as scientific computing, cloud computing, and enterprise services, request patterns vary widely in terms of arrival rates, resource access popularity, and other application-specific characteristics. Furthermore, user behavior may change over time, as some applications exhibit pronounced seasonality, such as workload spikes during Black Friday or major sports events. Existing benchmarking standards frequently fail to capture this temporal complexity and the nuanced access patterns found in production environments. In this paper, we present Defigium, a modular framework designed to generate customized benchmark workloads by analyzing real execution traces. The framework employs a three-stage pipeline consisting of trace-based parsing, workload model generation, and load execution. To demonstrate Defigium's modularity and utility, we implemented parsing modules for Redis and HTTP to cover key-value stores and web application examples. We further developed two distinct workload model generation modules: a standard replay and a compacted version utilizing heatmaps to capture the frequency of operations and inter-arrival times. Our evaluation shows that Defigium provides extensibility and reuse while effectively characterizing dynamic production environments.*

1. Introduction

Evaluating and optimizing distributed systems relies heavily on reproducing realistic operational environments. Standardized benchmarking tools, from the Yahoo! Cloud Serving Benchmark (YCSB) [Cooper et al. 2010] to industry standards such as TPC-C and TPC-H [TPC 1988], offer reliable baselines for comparing system configurations, but depend on static mathematical profiles or structurally rigid transactional scenarios. These pre-defined profiles struggle to capture the complexity, non-stationarity, and variability of actual production usage, and lack the flexibility required to simulate the volatile reality of modern distributed systems. Prior research has demonstrated that generic benchmarks oversimplify critical system behaviors, including traffic burstiness and complex data dependencies, which can lead to inaccurate performance assessments where systems appear optimized in controlled settings but fail under real-world pressure [Atikoglu et al. 2012, Cao et al. 2020]. Thus, researchers must choose between standardized but potentially unrepresentative tools or complex, specialized standards that are difficult to adapt.

Recognizing this gap, recent efforts have shifted toward trace-based workload generators that extract properties directly from operational logs. Although these tools provide higher fidelity than synthetic baselines, they are tightly coupled to specific domains, such as storage systems or financial data feeds [Sladojević et al. 2022]. As a result, applying them to different layers of the computing stack requires rewriting the generation engine. A generalized approach is required to accommodate the specificities of diverse contexts without sacrificing domain-specific trace realism.

In this paper, we introduce Defigium, a modular and extensible framework designed to synthesize benchmark workloads through the analysis of real-world traces. The core contribution of this work is the decoupling of the generation pipeline into three independent stages: Analysis, Generation, and Execution. This separation is achieved via an Intermediate Event Format (IEF) that isolates the statistical modeling logic from the syntax of the target system's protocol. By abstracting these concerns, Defigium can integrate distinct generation methods, from probability-based temporal heatmaps to Markov Chains or machine learning-based approaches, and apply them across diverse application domains, such as databases, HTTP-based web applications or Message Queuing traffic.

The remainder of this paper is organized as follows. Section 2 reviews related work. Sections 3 and 4 detail the architecture and implementation of Defigium, highlighting its modular design, target parsing modules (Redis and HTTP), and generative strategies (replay and probabilistic heatmaps). Section 5 evaluates the framework's analytical fidelity and structural modularity, and Section 6 concludes the paper.

2. Related Work

Operational logs provide empirical ground truth in distributed systems by capturing nondeterministic concurrency and real-world behavioral nuances absent from synthetic models. Trace-driven characterization underpins diverse areas. In requirements engineering, production logs ground feature prioritization in empirical usage to align stakeholders [Mendizabal et al. 2012, Maalej et al. 2015]. In security, audit logs enable anomaly and intrusion detection via deviations from operational baselines [Du et al. 2017], and in cloud infrastructure, traces drive predictive auto-scaling against QoS guarantees [Lorido-Botran et al. 2014] while providing the telemetry required for root-cause diagnosis in microservices [Zhou et al. 2018].

Concerning software performance testing, the generation of high-fidelity, customized benchmarks is a process that inherently depends on two fundamental activities: characterization, which extracts the core properties of a real workload, and synthesis, which recreates these properties to produce a representative synthetic load. The literature in this field explores a wide spectrum of methodologies across these two fronts, ranging from empirical observation [Cao et al. 2020, Yang et al. 2021] and pure statistical analysis techniques [Talluri et al. 2019, Gmach et al. 2007] to complex generative models [Delimitrou and Kozyrakis 2011, Khan et al. 2012].

In terms of characterization, research frequently focuses on extracting statistical properties from production traces, such as Empirical Cumulative Distribution Functions and the Hurst parameter to analyze long-term dependencies [Talluri et al. 2019], or time series analysis to identify cyclic patterns in enterprise data centers [Gmach et al. 2007]. While these studies provide robust mathematical models of system behavior, their

methodologies remain highly domain-specific, rigidly coupled to a particular infrastructure and lacking the flexibility to be repurposed across different computing environments.

Synthesis approaches vary depending on the target abstraction layer: at the hardware level, Kim et al. [Kim et al. 2014] synthesize workloads directly from microarchitectural execution profiles, such as cache misses and memory instructions collected via hardware counters; at the application level, tools like Hoagie [Ghandeharizadeh and Huang 2018] generate workloads without relying on operational traces, instead utilizing published specifications and standard distributions to build database schemas and issue operations. Other efforts aim for high fidelity by modeling sequential dependencies; for instance, Delimitrou and Kozyrakis [Delimitrou and Kozyrakis 2011] used Hierarchical Markov Chains to capture spatial and temporal locality in storage I/O.

Closest to our approach is Tracie [Sfakianakis et al. 2021], which generates workloads by sampling probability distributions and mapping them to existing benchmark tools. However, most trace-based generators remain tightly coupled to specific application domains. Defigium addresses this limitation through a modular architecture. Employing an intermediate event representation, the tool decouples the statistical modeling load generator from the underlying execution protocol. This design allows the same load generator to be applied across heterogeneous environments, such as key-value databases and HTTP web services, as we demonstrate in our use cases.

3. Defigium Architecture

Defigium operates as a modular framework structured in a three-stage processing pipeline. The design decouples the workload synthesis lifecycle into distinct phases: Parsing, Generation, and Execution. To standardize communication across these isolated stages, the framework introduces the Intermediate Event Format (IEF). The architecture employs structural and behavioral design patterns to ensure the core engine remains agnostic to specific log formats, statistical models, and target network protocols.

Figure 1 illustrates the end-to-end data flow within this architecture. The process begins with the Parser ingesting a Raw Trace and translating its domain-specific entries into the standardized IEF. This intermediate stream is then processed by the Generator module, which applies a designated statistical strategy to synthesize a new workload, persistently stored as a Synth File. Finally, the Executor module consumes this generated artifact to systematically dispatch the modeled operations against the intended Target System.

3.1. The Intermediate Event Format (IEF)

Trace-based generation must handle the structural heterogeneity of production logs. Defigium addresses this by standardizing data exchange through the IEF. Instead of passing protocol-specific text strings through the pipeline, the system serializes workload events into a canonical representation.

The IEF is structured as a generalized schema comprising the following core attributes: *timestamp* (the temporal execution mark), *client_id* (the source identifier), *op_type* (the explicit original command), *category* (a semantic classification, such as read, update, or delete), and *target* (the accessed resource identifier). Furthermore, it includes

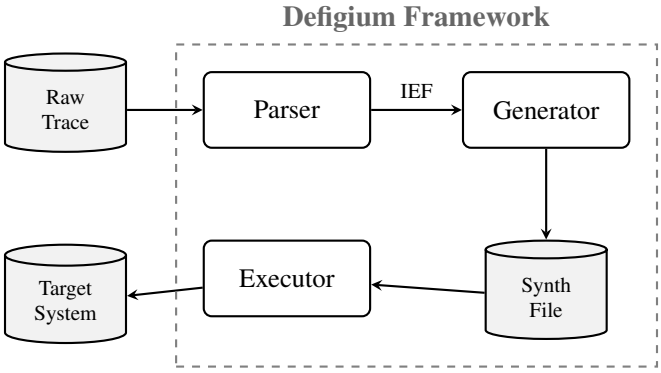


Figure 1. Data flow across Defigium Parser, Generator, and Executor modules.

a dynamic *payload* map to accommodate supplementary, protocol-specific arguments required for physical replay.

This schema design is fundamental to the framework's domain agnosticism. By mapping protocol-specific instructions into unified semantic classifications via the *category* attribute, the analysis module shields the generation engine from the underlying technology stack. As illustrated in Figure 2, fundamentally different operations are abstracted into generic semantic states. Thus, the generative algorithms can model state changes, track resource lifecycles, and calculate probability mass functions purely based on these standardized abstractions, remaining entirely decoupled from the specificities of the target systems.

3.2. Parser Module

The first stage of the pipeline functions as a bidirectional translator. During data ingestion, it converts raw production traces into the standardized IEF representation. The module employs the Factory Method pattern to instantiate domain-specific parsers based on the target system. The selected parser reads the input file, applies lexical analysis to extract operational features, and maps them to the IEF structure. At the end of the generation pipeline, this module performs the inverse operation. It receives the sequence of IEF events and formats them into a textual representation that complies with the communication protocol expected by the execution engine. This dual role isolates the complexity of parsing and formatting within interchangeable components.

Figure 2 exemplifies this abstraction process. Within the Parser Module, the system instantiates specialized parsers (e.g., a Redis Parser or an HTTP Parser) tailored to the specific syntax of the incoming raw logs. As illustrated, fundamentally different operations are processed by their respective modules and then converged and standardized into agnostic IEF objects.

3.3. Generator Module

The Generation Module applies a designated workload generation strategy to the chronologically ordered sequence of IEF events to produce a synthetic trace. The implementation relies on the Strategy pattern, allowing the interchange of modeling techniques without altering the control flow of the application.

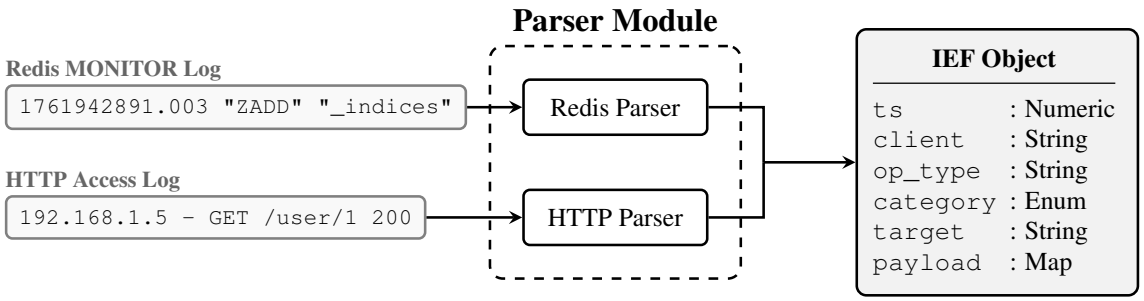


Figure 2. Parsers convert raw logs into technology-agnostic IEF objects.

The internal flow of a generation strategy is strictly divided into two distinct execution phases. During the initial Characterization Phase, the module processes the injected stream of IEF events to extract relevant properties, mathematically abstracting the input data into an internal Statistical Model. Depending on the chosen strategy, this model may encapsulate probability mass functions, state-transition matrices, or time-series profiles. Once the characterization is complete, the Synthesis Phase utilizes this constructed Statistical Model, alongside user-defined parameters such as simulation duration or temporal scaling factors, to recursively generate a novel, probabilistic sequence of synthetic IEF events. The resulting output sequence is subsequently routed back to the Parser Module to be formatted into the final, reproducible textual synthetic trace.

3.4. Execution Module

The final stage of the pipeline transforms the synthetic trace into physical interactions over the network. The Execution Module consumes the generated artifact and orchestrates the dispatch of operations against a target system. To achieve high throughput and deterministic timing, this module relies on a Thread Pool architecture, in which operations are pre-ordered into a shared dispatch queue and consumed independently by a pool of worker threads, as detailed in Section 4.3.

4. Implementation Strategy

To instantiate and evaluate the proposed framework, we developed a full-fledged implementation of Defigium.[1] We adopted a dual-language design to balance analytical capabilities with operational control. The Parsing and Generation phases are implemented in Python 3, utilizing its data processing ecosystem for statistical modeling. The Execution phase is implemented in C++17 to ensure deterministic timing and fine-grained control over system threads. The entire pipeline execution is orchestrated via a standard YAML configuration file.

4.1. Data Ingestion and Parsing

The Parser module utilizes the Factory Method pattern to instantiate domain-specific parsing strategies. To validate the framework's domain agnosticism, we developed two concrete parsers: one tailored to the Redis database, processing execution logs generated by the Redis `MONITOR` command, and a parallelized HTTP parser designed to ingest large-scale web server access logs.

[1] Source code available at `https://github.com/FKettl/Defigium`

During ingestion, the parser applies regular expressions to extract fundamental attributes such as timestamps, client identifiers, and command strings. The timestamps are rounded based on a configurable granularity to normalize the data. The parser then maps the specific database commands to general semantic categories, assigning tags such as read, update, or create. This mapping populates the Intermediate Event Format (IEF) objects. Entries that deviate from the expected protocol syntax are bypassed, and the validated IEF sequence is forwarded to the generation module. To complete the pipeline, the Parser module also implements an inverse routine, transforming synthetic IEF objects back into a valid textual format to output the reproducible synthetic trace file.

4.2. Workload Characterization and Synthesis

The Generation module adopts the Strategy design pattern to facilitate the integration of interchangeable modeling techniques. To establish a baseline for validation, we developed a *Replay Strategy*, which translates each trace entry into a corresponding IEF representation. This ensures the capture and reproduction of a workload that preserves the characteristics of the original operational log.

The primary implementation of our generation module is a probabilistic *Heatmap Strategy*. To capture non-stationary and fluctuations on workload common in production, the characterization phase partitions the chronological input trace into N equal-sized temporal windows, denoted as $\mathcal{W} = \{W_1, W_2, \ldots, W_N\}$. Instead of computing a single global average, which would incorrectly smooth out localized traffic spikes, the module extracts a distinct statistical profile for each specific window W_i. It computes independent probability mass functions for the frequency of command categories, denoted as $P(C|W_i)$. Simultaneously, it extracts the continuous probability distribution of inter-arrival times, $P(\Delta t|W_i)$. As illustrated in Figure 3, this process abstracts the entire trace into a multi-dimensional probability matrix, mapping exactly how the likelihood of specific operations evolves across time frames. Darker cells indicate a higher probability of a specific operation occurring within a given temporal window.

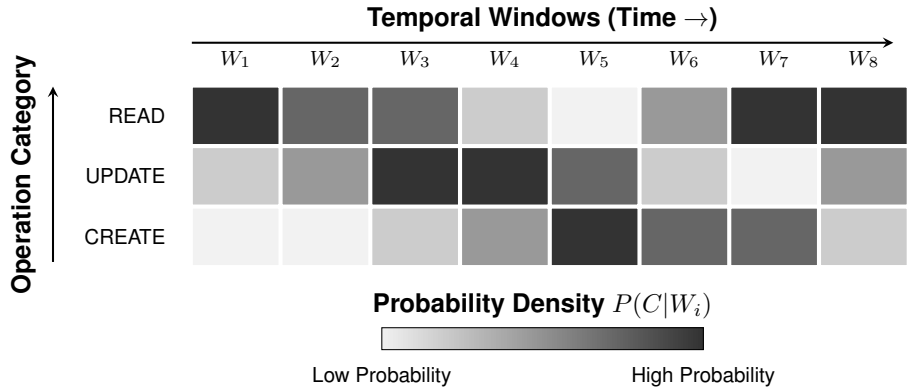


Figure 3. Conceptual representation of the Statistical Heatmap Strategy.

During the synthesis phase, the generative engine produces a synthetic trace by navigating the learned probability matrix through a sequence of conditional sampling steps. The process is governed by a virtual simulation clock t, which dictates the currently active time window W_i. To construct a new event, the engine first determines its

chronological placement by sampling an inter-arrival time $\Delta t \sim P(\Delta t|W_i)$ and advancing the clock. Immediately after, it samples the categorical distribution $P(C|W_i)$ to define the semantic nature of the operation.

Once the timing and the action are established, the engine assigns a target resource to construct the complete operational event. To ensure successful execution during the subsequent replay phase, the generator uniformly selects targets from a predefined pool of valid identifiers extracted during the characterization.

Furthermore, to accommodate requests for synthetic traces that extend beyond the duration of the original baseline, the module applies temporal mapping functions. By employing techniques such as cyclic repetition or proportional stretching over the $\mathcal{W}$ windows, the framework algorithmically scales the workload duration while strictly preserving the underlying statistical rhythms.

The intentional modulation of specific parameters (e.g., inter-arrival time and probability mass function) could enable system designers to simulate projected shifts in user behavior or extreme operational scenarios, such as seasonal fluctuations or flash crowd events like Black Friday. By inducing these modifications, the generation engine can produce high-fidelity, *what-if* scenarios that remain grounded in the structural characteristics of real-world traces.

4.3. Concurrent Execution Engine

The Execution module is implemented in C++17 using a Thread Pool pattern. The main thread reads the synthetic trace file, calculates the absolute dispatch time for each operation relative to the start of the execution, and enqueues the tasks, pre-ordered chronologically, into a shared thread-safe queue consumed by a configurable number of worker threads. Each worker establishes an independent, persistent network connection to the target system. Upon receiving a task, the worker thread invokes the standard *sleep* function, passing the scheduled absolute timestamp. This routine halts the thread execution until the exact scheduled moment, minimizing scheduling drift without introducing busy-waiting overhead. Once awake, the worker translates the IEF event into the appropriate network protocol call and increments atomic counters to register the operation result.

To maintain modularity at the compilation level, the executor employs conditional compilation directives. The executor factory evaluates preprocessor flags to include header files and instantiate specific strategies only when explicitly required. This build system ensures that domain-specific dependencies, such as third-party database drivers or HTTP libraries, are not linked to the final executable unless they are explicitly configured for that target. This design keeps the framework lightweight and simplifies the integration of new execution modules.

5. Experimental Evaluation

The experimental evaluation assesses the analytical fidelity and structural modularity of the Defigium framework. The methodology consists of a quantitative comparative analysis across three distinct trace states: the original baseline trace, the synthetic trace produced by the Generation module, and the received trace captured during the physical injection by the Execution module.

The primary baseline workload consists of the execution phase of the YCSB Workload A, directed at a Redis in-memory database. The trace was captured using the Redis MONITOR command with microsecond precision. The dataset isolates an execution window containing approximately 100,000 operations, characterized by an approximate 50% read and 50% update ratio.

The quantitative analysis employs both the p-value from Pearson's Chi-squared tests and Cramér's V to measure, respectively, the statistical significance and the effect size of the differences between the original and synthetic traces, applied to the categorical command composition and the discretized inter-arrival time distributions. Massive sample sizes cause the p-value to become overly sensitive to trivial variations in high-resolution continuous variables, such as microsecond-level inter-arrival times, systematically approaching zero even when distributions are practically identical [Lin et al. 2013]; for strictly categorical or heavily quantized temporal data (e.g., second-level web logs), however, it remains a robust indicator of equivalence. To ensure a comprehensive evaluation across all resolutions, we use Cramér's V complementarily to quantify the actual magnitude of the divergence, where values below 0.1 denote a trivial effect size and high practical similarity [Cohen 2013].

5.1. Test Environment

To evaluate the system under controlled conditions, we established a dedicated bare-metal test environment utilizing two identical *d430* physical compute nodes provisioned via the Emulab[2] testbed [White et al. 2002]. To ensure performance isolation and eliminate virtualized multi-tenant overhead, the nodes were allocated with exclusive hardware access. The first node was configured exclusively for load generation, running the Defigium Execution Engine. The second node hosted the target server application, *i.e.*, the Redis kv-store. Each *d430* node is equipped with two 8-core Intel Xeon E5-2630v3 processors running at 2.4 GHz (totaling 16 physical cores per node), 64 GB of DDR4 ECC RAM, and a 1 Gbps onboard network interface card connected to a local area network switch fabric. Both machines were provisioned with a standardized Ubuntu 22.04 LTS (64-bit) operating system environment.

5.2. Scenarios and Validation

The evaluation comprises six experiments designed to demonstrate the capabilities of workload characterization. To ensure statistical reliability and mitigate sampling variance, each experiment was subjected to 30 independent runs. The quantitative results reported in this section represent the arithmetic mean across these iterations. Table 1 summarizes the statistical similarity metrics.

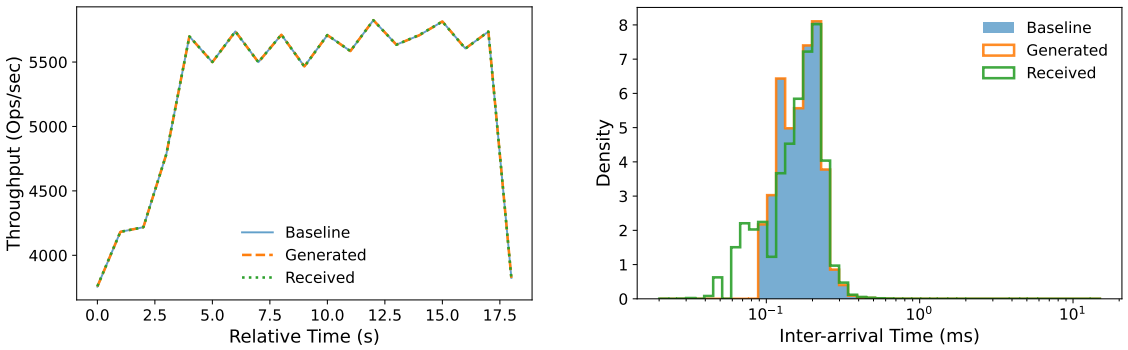
The first experiment evaluates the baseline parsing and execution flow utilizing the replay strategy. This configuration aims to validate the lossless nature of the complete pipeline, ensuring that the IEF translation introduces no structural artifacts. As indicated in Table 1, the Cramér's V value for the generated command composition and inter-arrival times is 0.0, indicating deterministic parsing. Visual analysis of Figures 4a and 4b corroborates an overlap between the baseline and generated throughput and inter-arrival density.

[2]https://www.emulab.net/

Table 1. Statistical similarity ($N = 30$ runs): P (p-value), V (Cramér's V).

Configuration	P (Cmd)	V (Cmd)	P (Time)	V (Time)
Baseline	-	-	-	-
Exp. 1 (Replay)	1.0000	0.0000	1.0000	0.0000
Exp. 2 (1%)	0.6146	0.0013	< 0.0001	0.0493
Exp. 3 (50%)	0.5415	0.0016	< 0.0001	0.0498
Exp. 4 (Cyclic)	0.5886	0.0011	< 0.0001	0.0536
Exp. 5 (HTTP)	0.7493	0.0003	0.8104	0.0004
Exp. 6 (Cluster)	0.6459	0.0079	0.9456	0.0449

To evaluate the execution stage, the Replay synthetic trace was dispatched to a local, containerized Redis instance using the custom C++ executor. The executed operations were logged via the `redis-cli monitor` command to log the "Received" trace. The original baseline trace was generated in the same hardware and network environment using YCSB. Figure 4a illustrates the throughput over time for the baseline, generated, and received traces. The generated and received loads align with the baseline, accurately reproducing the intended throughput.



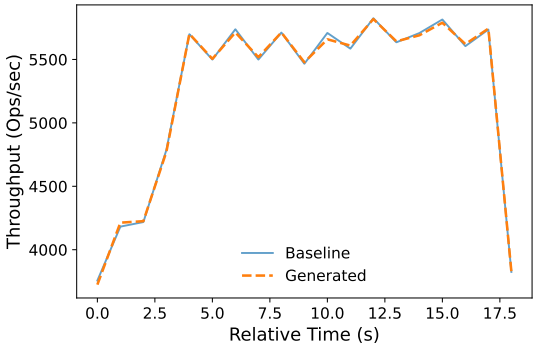
(a) Exp 1: Replay (Throughput)

(b) Exp 1: Replay (Inter-arrival)

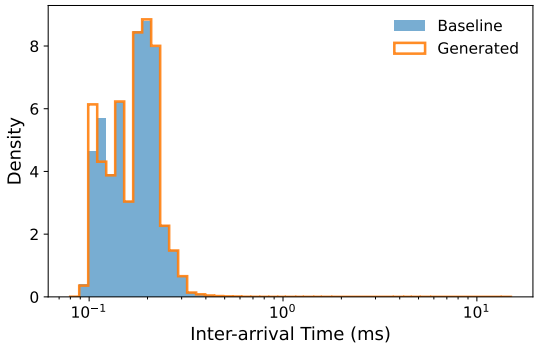
Figure 4. Throughput and inter-arrival time density for the Replay Strategy.

Experiments 2 and 3 introduce the probabilistic heatmap strategy to evaluate the framework's temporal modeling capabilities, exposing the operational trade-off between capturing localized request burstiness and applying statistical smoothing. Exp. 2 applies a fine-grained characterization window of 1% over the input trace, capturing the workload's non-stationary nature and replicating micro-bursts and dynamic fluctuations in the request rate (Figure 5a). Conversely, Exp. 3 applies a coarse-grained setting with window of 50%; while it reproduces the global command ratio and overall inter-arrival density (Figure 6b), the large temporal window acts as a smoothing filter, homogenizing the request rate and failing to reproduce the localized traffic spikes observed in the baseline (Figure 6a). Across both configurations, the quantitative results (Table 1) yield high p-values for command composition ($p > 0.54$), failing to reject the null hypothesis, while Cramér's V for inter-arrival times remains below 0.06, implying a trivial effect size and practical equivalence despite the expected p-value collapse.

Exp. 4 evaluates the temporal expansion capability of the framework. Using the cyclic mapping strategy, the motivation is to demonstrate how a constrained 19-second



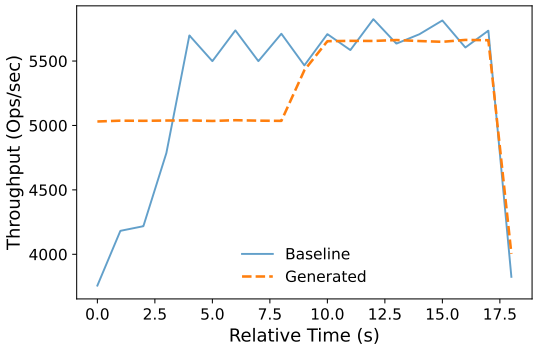
(a) Exp 2: 1% Window (Throughput)



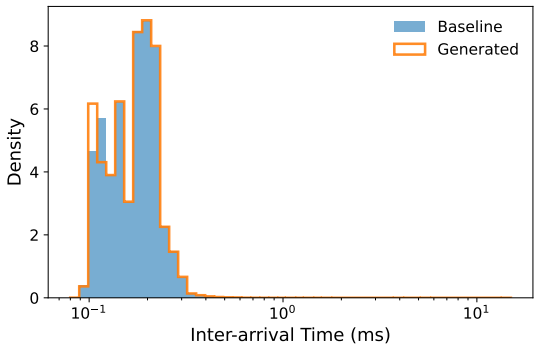
(b) Exp 2: 1% Window (Inter-arrival)

Figure 5. Throughput and inter-arrival time varying Heatmap window granularity: fine-grained (1%) window.

baseline trace can be extended into a 40-second synthetic workload while preserving the learned statistical distributions. Figure 7 shows the probability mass functions and inter-arrival rhythms for the extended duration, demonstrating temporal scalability without statistical degradation.

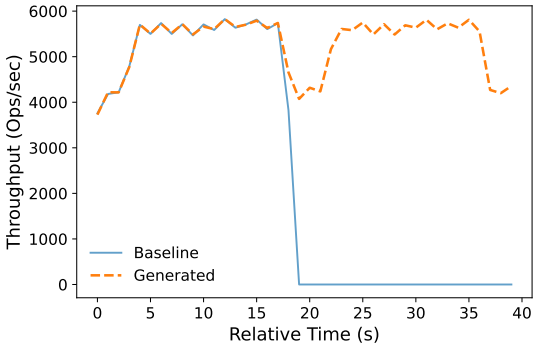


(a) Exp 3: 50% Window (Throughput)

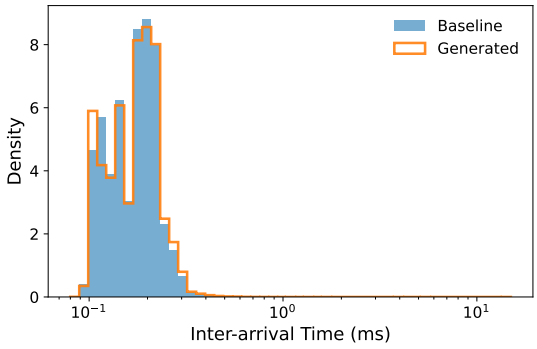


(b) Exp 3: 50% Window (Inter-arrival)

Figure 6. Throughput and inter-arrival time varying Heatmap window granularity: coarse-grained (50%) window.



(a) Exp 4: Cyclic (Throughput)



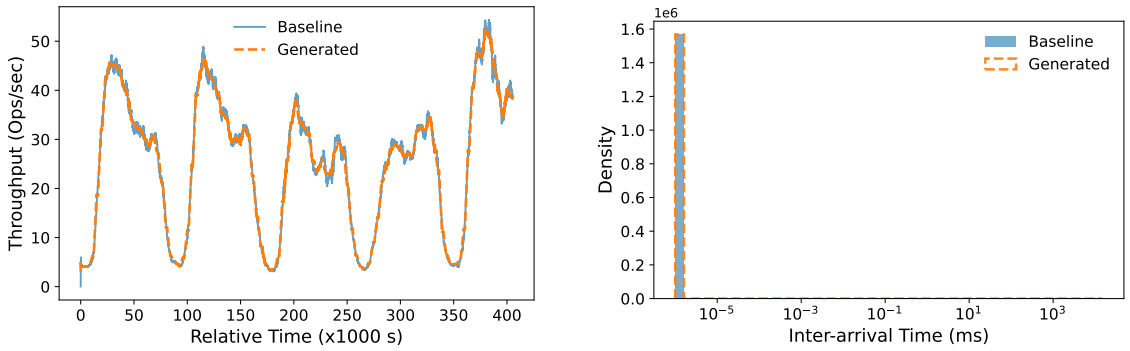
(b) Exp 4: Cyclic (Inter-arrival)

Figure 7. Throughput and inter-arrival time density for expanded synthetic workload.

5.3. Domain Agnosticism and Modularity

To demonstrate the framework's domain agnosticism, the evaluation was extended to a real-world HTTP web server dataset [Zaker 2019] containing over 10 million access logs from an online shopping platform. A parallelized HTTP parser was instantiated to interface with the core architecture, extracting semantic operations and temporal metrics. This modular substitution allowed the exact same probabilistic Heatmap strategy to be applied to synthesize the workload without requiring any modifications, except for the parser module itself.

Figure 8 illustrates the workload fidelity across both macro and micro temporal scales. Figure 8a shows the throughput, where the generator captures the cycles of web traffic over hundreds of thousands of seconds (more than 4 days). As can be noticed, the periodic fluctuations characteristic of diurnal cycles and nighttime troughs are accurately reproduced by the heatmap strategy. Figure 8b details the inter-arrival time density. Because the timestamps in this specific dataset are recorded with a one-second granularity, the vast majority of sequential operations are logged at the exact same second. Consequently, the calculated inter-arrival time between these consecutive requests evaluates to zero. This discretization creates the massive density spike at zero observed in the chart, a structural behavior replicated by the synthetic trace.



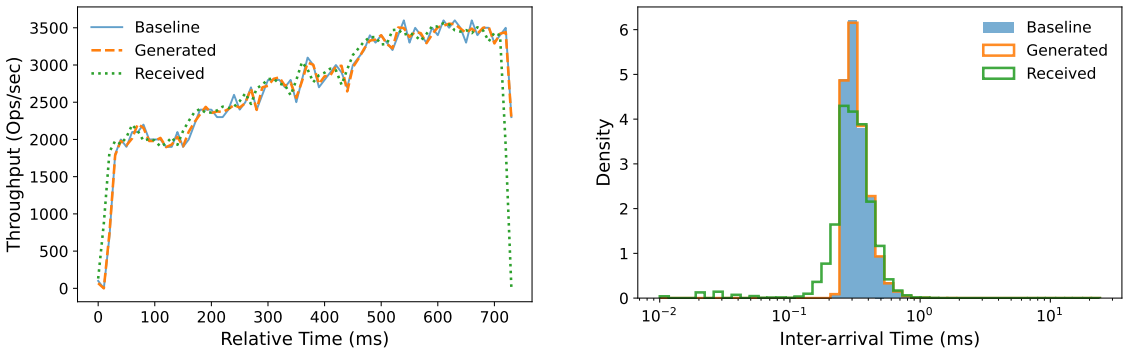
(a) Exp. 5: HTTP (Throughput)

(b) Exp. 5: HTTP (Inter-arrival)

Figure 8. Throughput and inter-arrival time density for the HTTP dataset.

As shown in Table 1 (Exp. 5), the Pearson's Chi-squared tests yielded high p-values, averaging 0.7493 for command proportions and 0.8104 for inter-arrival times across the synthetic runs. Because $p \gg 0.05$, the tests fail to reject the null hypothesis, suggesting high statistical similarity. These results, supported by exceptionally low Cramér's V values ($V \leq 0.0004$), indicate that Defigium can learn and reproduce workload behaviors across different application domains, logging formats, and time scales.

To further assess execution fidelity beyond a single-host container, Exp. 6 dispatched a heatmap-generated workload against a real multi-node Redis Cluster over the network. Figure 9 shows the resulting throughput and inter-arrival time density, closely tracking the baseline and received traces. As shown in Table 1 (Exp. 6), the high p-values ($p = 0.6459$ for commands, $p = 0.9456$ for inter-arrival times) and low Cramér's V values ($V \leq 0.0449$) confirm that timing and command fidelity are preserved even when operations traverse a real network topology, consistent with the results observed in the previous experiments.



(a) Exp 6: Cluster (Throughput)

(b) Exp 6: Cluster (Inter-arrival)

Figure 9. Throughput and inter-arrival time density for the distributed Redis Cluster execution.

6. Conclusion

This paper introduced Defigium, a modular framework designed to address the inherent rigidity of synthetic benchmarks and the domain-coupling of traditional trace replayers. By utilizing an Intermediate Event Format (IEF), the architecture decouples data analysis, statistical synthesis, and physical execution into interchangeable components. This separation allows for high-fidelity workload reproduction while maintaining protocol agnosticism across diverse distributed layers.

Experimental results validated the framework's analytical precision and flexibility. Utilizing probabilistic Heatmap and Cyclic mapping strategies, Defigium synthesized complex operation ratios and inter-arrival time distributions from production Redis traces, with statistical analysis confirming a high degree of practical equivalence between original and generated workloads and robust scalability across extended trace durations. The evaluation further confirmed the framework's modularity and network robustness through successful application to HTTP web server datasets and a real multi-node Redis Cluster deployment, without core modifications, facilitating the integration of specialized execution modules tailored to a variety of target applications.

References

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.

Cao, Z., Dong, S., Vemuri, S., and Du, D. H. (2020). Characterizing, modeling, and benchmarking RocksDB Key-Value workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223.

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*, SoCC '10, page 143–154, New York, NY, USA. Association for Computing Machinery.

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.

Du, M., Li, F., Zheng, G., and Srikumar, V. (2017). Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 1285–1298.

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.

Khan, A., Yan, X., Tao, S., and Anerousis, N. (2012). Workload characterization and prediction in the cloud: A multiple time series approach. In *2012 IEEE Network Operations and Management Symposium*, pages 1287–1294.

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.

Lorido-Botran, T., Miguel-Alonso, J., and Lozano, J. A. (2014). A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing*, 12(4):559–592.

Maalej, W., Nayebi, M., Johann, T., and Ruhe, G. (2015). Toward data-driven requirements engineering. *IEEE software*, 33(1):48–54.

Mendizabal, O. M., Spier, M., and Saad, R. (2012). Log-based approach for performance requirements elicitation and prioritization. In *2012 20th IEEE International Requirements Engineering Conference (RE)*, pages 297–302. IEEE.

Sfakianakis, Y., Kanellou, E., Marazakis, M., and Bilas, A. (2021). Trace-based workload generation and execution. In *International Conference on Parallel and Distributed Computing (Euro-Par)*.

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.

TPC (1988). TPC-Homepage.

White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. (2002). An integrated experimental environment for dis-

tributed systems and networks. *ACM SIGOPS Operating Systems Review*, 36(SI):255–270.

Yang, J., Yue, Y., and Rashmi, K. (2021). A large-scale analysis of hundreds of in-memory key-value cache clusters at twitter. *ACM Transactions on Storage (TOS)*, 17(3).

Zaker, F. (2019). Online Shopping Store - Web Server Logs.

Zhou, X., Peng, X., Xie, T., Sun, J., Ji, C., Li, W., and Ding, D. (2018). Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering*, 47(2):243–260.