Gen-T: Reduce Distributed Tracing Operational Costs Using Generative Models

Saar Tochner
Carnegie-Mellon University
saar.tochner@gmail.com

Giulia Fanti
Carnegie-Mellon University
gfanti@andrew.cmu.edu

Vyas Sekar
Carnegie-Mellon University
vsekar@andrew.cmu.edu

Abstract

Distributed tracing (DT) is an important aspect of modern microservice operations. It allows operators to troubleshoot problems by modeling the sequence of services a specific request traverses in the system. However, transmitting traces incurs significant costs. This forces operators to use coarse-grained prefiltering or sampling techniques, creating undesirable tradeoffs between cost and fidelity. We propose to circumvent these issues using generative modeling to capture the semantic structure of collected traces in a lossy-yet-succinct way. Realizing this potential in practice, however, is challenging. Naively extending ideas from the literature on deep generative models in timeseries generation or graph generation can result in poor cost-fidelity tradeoffs.

In designing and implementing Gen-T, we tackle key algorithmic and systems challenges to make deep generative models practical for DT. We design a hybrid generative model that separately models different components of DT data, and conditionally stitches them together. Our system Gen-T, which has been integrated with the widely-used OpenTelemetry framework, achieves a level of fidelity comparable to that of 1:15 sampling, which is more fine-grained than the default 1:20 sampling setting in the Opentelemetry documentation, while maintaining a cost profile equivalent to that of 1:100 lossless-compressed sampling (i.e., a $7\times$ volume reduction).

1 Introduction

Microservice architectures are emerging as a popular design paradigm for building scalable and agile software systems being widely adopted by leading enterprises (e.g., Netflix [44], T-Mobile [65], and Coca-Cola [11]). As these environments grow in complexity, there's an increasing need for *observability* (often shortened to "o11y") tools to manage, diagnose, and optimize these systems. The o11y market has been on a meteoric rise [47], [48].

Within o11y, *distributed tracing* (DT) is a critical capability for troubleshooting. Tracing captures the fine-grained journey of a request as it travels through the services. Unlike local-view signals such as metrics (e.g., traffic, CPU timeseries) and logs (e.g, text output from apps) [50], traces stitch together the discrete steps into a coherent, end-to-end narrative [30]. For instance, the breakdown of time spent in different tasks can help operators pinpoint inefficiencies [14]. Creating trace graphs that show which components called each other allows developers to pinpoint common input pathways for root cause analysis [42, 46].

In practice, however, the cost of transmitting traces results in operators disabling or reducing tracing [45, 51]. Many rely on sampling to reduce cost [17, 53, 64]. Unfortunately, sampling forces

Temporal Graph Learning Workshop @ NeurIPS 2023, New Orleans.

operators into untenable tradeoffs between cost and fidelity. For instance, "head" sampling, selects a subset of requests when they first arrive. This may capture frequent requests and miss interesting details. On the other hand, "tail" sampling, requires operators to specify patterns of interesting requests to retain they complete. This may not detect new issues, defeating the very purpose of tracing.

We revisit DT through the lens of deep generative modeling to offer better fidelity-flexibility-cost tradeoffs. We observe that in many tracing use cases, operators are less concerned with individual traces and more focused on structural patterns spanning subpopulations of traces [54, 31, 15].

Modeling DT data with generative models poses two main challenges. First, there are algorithmic issues; existing generative algorithms are tailored either for time-series data [70, 56] or for graph structures [5, 16]. However, distributed tracing combines these data modalities, presenting a unique challenge. Second, making the system practical and integrating into existing tracing workflows [50], requires balancing scalability, cost-efficiency, and latency.

We present Gen-T, a practical system for generative compression of traces that can be plugged directly into existing distributed tracing workflows. We make the following contributions:

1. We exploit the unique structural properties of trace data to design a custom approach for trace compression, that carefully combines graph and tabular techniques to produce a time-dependent graph generative model.
2. We evaluate Gen-T using two widely-used demo applications: the serverless-focused Wildrydes [68], AWS's official guide, and HotROD [32], a staple example for distributed tracing. We compare Gen-T against a suite of existing solutions (several of which are industry standards today): two tail-sampling methods, and head-sampling combined with two distinct compression approaches: the generic gzip, and the domain-specific Compressed Log Processor (CLP) [58]. Our evaluation suggests that Gen-T achieves a level of fidelity comparable or better to that of 1:15 sampling, which is more fine-grained than the default 1:20 sampling setting in the Opentelemetry documentation, while maintaining a cost profile equivalent to that of 1:100 lossless-compressed sampling (i.e., a $7\times$ volume reduction).

Our open-source implementation integrate Gen-T seamlessly into industry-standard OpenTelemetry (OTEL) workflows. The generated data output by Gen-T follows OTEL schemas making it backward compatible with existing tooling.

2 Preliminaries and Motivation

Background on O11y and Tracing In the era of microservices, understanding system behavior has grown complex. This is the goal of Observability, or o11y: the ability to monitor various system components to help operators and engineers troubleshoot issues. We exemplify this using OpenTelemetry (OTEL)[50], an industry standard providing APIs, SDKs, and tools for o11y data management. The process begins with code instrumentation, followed by data collection by the Collector[55]. This data is sent to analysis platforms [30, 74, 49, 25] for further processing, alerting, and real-time detection, and stored in databases like Elasticsearch [18], Apache Cassandra [3], Quickwit [57], or OpenObserve [49].

There are three canonical types of data in o11y: logs, metrics, and traces. Logs record textual events from systems and applications, while metrics numerically measures the system (e.g. CPU utilization and rate). *Traces* chronicle a request's journey through microservices, making them essential for identifying issues and optimizing system performance. Our focus is specifically on traces since they are the most challenging and open question for o11y today.

In tracing, a **span** represents a system's discrete work unit, marked with unique identifiers (spanId, parentId, and traceId), timestamps, metadata, and a status indicating the operation's outcome. A **trace** combines these spans, representing a request's complete journey through the system. **Triggers**, within this context, clarify span dependencies, outlining the sequence of operations for a specific request.

Tracing systems typically provide two primary data views. The trace view, displayed in Figure 1, offers insight into trigger correlations, vividly illustrating the complex flow and real-time interaction among various services. Conversely, the timeline view, as shown in Figure 2, aids in identifying performance bottlenecks, highlighting where delays or failures transpire in a transaction. Any model of DT data must faithfully capture these graph structures and temporal correlations.

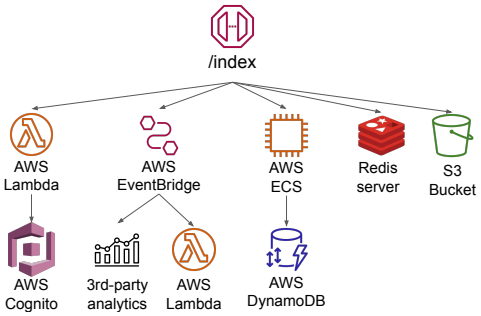

Figure 1: A DAG view of a trace. Illustrating the interaction between microservices. This flow consists of 21 Opentelemetry spans per request.

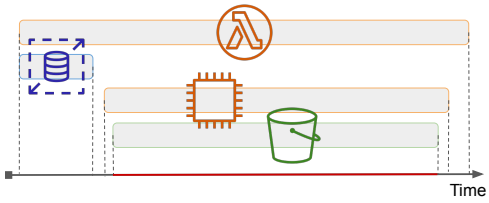

Figure 2: A timeline view of a trace. Illustrating duration breakdown. The grey boxes indicate the duration of each component.

Status Quo While tracing is valuable, it incurs significant costs, which has become a pressing industry-wide concern (e.g., Uber [67], Netflix [45], and Twitter [66]). While some efforts try to reduce the storage cost (e.g., OpenObserve [49] and Quickwit [57]), the primary strain on resources arises from transmitting traces.

To tackle this problem, several candidate proposals are being used. Firstly, lossless compression methods, targeting high-quality data, face limitations due to data's dynamic nature. The second class involves *sampling*. Today's industry best practices defines two types: head-based and tail-based sampling [33, 51]. A summary comparison can be found in Table 1.

Alternative	Cost Reduction	Ad-hoc Queries	Accuracy
Lossless Compress	Low	Y	Best
Metrics / Sketching	High	N	Best
Head-based Sample	Med	Y	Low
Tail-based Sample	Med	N	Varies

Table 1: Current alternatives for reducing DT costs

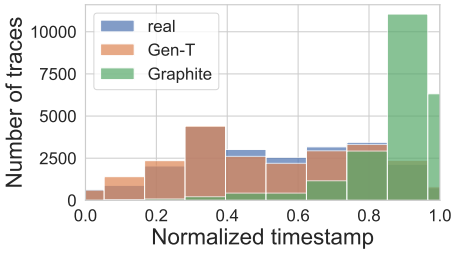

Figure 3: Generated timestamps using graph models vs. real data and Gen-T.

Head-based sampling uniformly drops traces *before collection*, offering a predictable cost reduction and a dataset suitable for general queries. However, this method can inadvertently omit vital data, potentially skewing query outcomes. Conversely, tail-based sampling filters traces post-collection based on predefined criteria (e.g., only traces with errors). This ensures a precise dataset based on that criteria but can fall short for ad-hoc queries. Its cost-effectiveness is also dependent on the selected criteria and the behavior of the observed system.

3 Our Approach

The ad-hoc nature of user queries in tracing means that compression techniques that make *a priori* assumptions about specific patterns in the data are likely to fail. Moreover, traces consists of a large number of small time-series data points with few repeated patterns, which limits the effectiveness of traditional lossless or lossy compression approaches. Generative models can act as a form of *lossy compression*, maintaining essential data features while reducing size [63, 7, 60]. However, existing generative models are not well-suited to DT data, which combines both graph and temporal information. We show why existing temporal- and graph-structured generative models do not solve the compression problem.

Time Series Models. Treating trace data as a time series is a conceivable approach where each element contains extensive information about spans. However, experiments with models like TabFormer [69] and NetShare [70] proved inadequate. We trained these models in default configurations, and evaluated the similarity of graph structures using the Tree Edit Distance [71], which counts the modifications

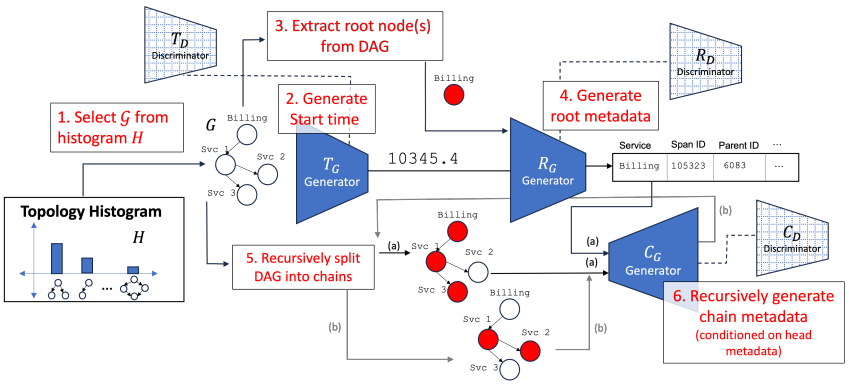

Figure 4: Overview of the Gen-T algorithmic architecture for generating a new trace. In total, we learn 4 structures, of which 3 are neural networks (NNs) (in reality, we train 6 NNs, because the models are GANs, which require training both a generator and a discriminator). The parameters of 3 NNs (shown in boldface) are transmitted to the target service. In steps 5 and 6, gray arrows labeled "(b)" denote a later iteration of the recursion, in which Gen-T populates the chain from `Svc 1` to `Svc 2` in the DAG.

needed to align the synthetic trace structure. Our results show that the synthetic data's graph topologies didn't align with the real data, showcasing an average TED of 45.5% for NetShare and 47.5% for TabFormer. This stark discrepancy emphasizes that these models fail to capture the hierarchical dependencies intrinsic in trace data, underscoring the necessity for an alternate modeling approach.

Graph generative models. We also evaluated graph generative models that can generate node metadata, like Graphite [27]. Even a simplified training task with Graphite, focusing only on the start timestamps, highlighted significant challenges. The model generated a very different timestamp distribution from the real one (Figure 3), exposing its inability to effectively model temporal correlations in trace data. Furthermore, the extensive disk space demands of existing graph generative models, like GRAN [40] and GraphGen [24] made them ineffective for DT compression.

3.1 Design

At a high level, Gen-T generates a trace in six steps, shown in Figure 4. Briefly, they are: (1) Select a DAG graph topology, $\mathcal{G}$. (2) Conditioned on $\mathcal{G}$, generate a random start time for the trace. (3) Identify the root nodes of $\mathcal{G}$. (4) Conditioned on the topology and the start time, populate the root nodes' metadata (recall that each node represents a span, and thus contains many metadata fields that must be populated). (5) Recursively extract chains whose head node's metadata has already been filled; the *head* node is the first node in the chain, with the smallest time stamp. (6) Given a chain whose head node's metadata is already filled, generate the metadata for the remaining unfilled nodes in the chain. Return to (5) unless all nodes in $\mathcal{G}$ are filled.

To support this generation pipeline, we learn four different structures during training: (1) **Topology modeling** consists of enumerating the set of all topologies in the source data, and the frequency with which they occur. This histogram H is compressed losslessly. (2) **Start time modeling** learns a neural network (NN) T_G. Conditioned on a trace graph $\mathcal{G}$ (which is selected from the topology histogram H), T_G randomly generates a start time for the trace. (3) **Root node metadata modeling** learns a model R_G that generates metadata for any root node. R_G is conditioned on the graph topology $\mathcal{G}$ and the generated start time. (4) **Recursive chain metadata modeling** learns a model C_G that takes as input a partially-filled graph $\mathcal{G}$, and a chain of fixed length, in which the head node's metadata is already filled. C_G then generates the metadata of the remaining nodes, conditioned on that of the head of the chain.

We next explain each step in detail, and illustrate how Gen-T design navigates the fidelity-cost tradeoff.

(1) Topology modeling. We use "topology" to refer to a graph whose nodes are labeled with only their service name (other metadata are not yet populated).

Insight 1. *In real-world microservices, the number of distinct graph topologies is typically small, usually numbering at most a few hundred. In this regime, there is no need or benefit in using generative graph models.*

In distributed traces, the number of unique graph topologies is usually restricted to a few hundred. Even though any microservice could theoretically trigger another, real-world requests often traverse similar topologies. Hence, there is no need to deploy a deep generative model to create the hierarchical edges within these topologies.

Compression Phase. We compute a histogram H that counts how many times each topology appears in the source data. We also store the form of each topology. This information is compressed losslessly.

Generation Phase. For each generated trace, the provider chooses a topology $\mathcal{G}$ from H such that aggregate synthetic topologies exactly match the counts in H.

(2) Start time generation. Our second step is to generate a start time for the trace. Instead of using the *startTime* and *endTime* format from OpenTelemetry (OTEL), we opt for a *gapFromParent* time interval and a *duration* (an illustration can be found in the appendix, Figure 7). While this representation is easier to model, it requires us to generate a global start time for each trace. Refer to Appendix B for an ablation test that measures the effect of this approach.

Compression Phase. We generate the start time using CTGAN, a popular tabular generative model based on generative adversarial networks (GANs) [69]; as such, Figure 4 pairs T_G with a discriminative model T_D because GANs require the training of both a generator and a discriminator. However, only the generator parameters are transmitted. We used a generative model here rather than a discriminative one for easier sampling from the (continuous) posterior distribution.

Generation Phase. The generated topology $\mathcal{G}$ from the histogram H is passed to T_G. We sample from distribution $T_G(\mathcal{G})$ to generate a start time for the trace.

(3) Root node metadata generation. At this point, the nodes in $\mathcal{G}$ contain only a service name; the rest of their metadata fields, including inter-arrival times from parents, are empty. A natural approach would be to generate metadata for all nodes conditioned on the selected graph $\mathcal{G}$ and the timestamp $t \sim T_G(\mathcal{G})$. However, this approach is inefficient because in typical trace graphs, different nodes fill very different services based on their *position* in the graph; hence, modeling the full range of services for every node increases the dimensionality (and hence, storage cost) of the model.

Insight 2. *Entry point services, responsible for initial steps like authentication and quota control, typically have no intersection with internal services that provide specific logical functionality. Hence, we can learn separate models for entry points vs. other nodes to reduce model dimensionality.*

Given this insight, we partition the graph $\mathcal{G}$ into root nodes, symbolizing system "entry points", and everything else (step 3 of Figure 4).[1] This separation of entry points allows us to employ a smaller model with a reduced conditioning vector, tailored exclusively to the subset of entry point services.

We use a graph embedding layer (GCNConv [35]) to find a representative embedding of the topology. Refer to Appendix B for an ablation test that measures the effect of this approach.

Compression Phase. We train a dedicated generative model only for the root nodes' metadata, labeled R_G in Figure 4. We again use CTGAN [69], and train both a generator and a discriminator (Figure 4). Only the parameters of the generator R_G are transmitted to the target.

Generation Phase. The decompressor uses R_G to generate metadata for each root node in the graph $\mathcal{G}$, conditioned on the trace start time and the graph embedding of the root node being populated.

(4) Recursive chain metadata generation. Finally, we learn a model to populate the empty internal nodes' metadata in the DAG. There are many ways to generate this metadata, such as by traversing the graph in some predefined order and autoregressively generating each node's metadata conditioned on earlier nodes. However, such approaches significantly degrade fidelity and/or training cost.

Insight 3. *Instead of using one model to generate the metadata of all internal nodes in a graph (e.g., using a time series model), we can learn a smaller* tabular *model that recursively generates metadata only for local motifs. This captures local correlations on the graph, with orders of magnitude less storage volume than comparable time series or graph models.*

[1] At least one root nodes is guaranteed to exist because $\mathcal{G}$ is a DAG.

Motivated by the concept of *motifs* in network science [20] and graph generative modeling [28], we decompose the graph into smaller *chains* of nodes for simpler modeling. Each chain has a size of at most $\mathcal{C}$, and each edge in the DAG aligns with one chain, and timestamps increase from the head of the chain (i.e., the first node) to the tail.

Compression Phase. We model the chains metadata recursively, starting with chains that emanate from root nodes (an illustration can be found in the appendix, Figure 9). Specifically, we learn a tabular generative model C_G (again, this is instantiated with CTGAN [69] paired with a discriminator C_D, which is not transmitted). C_G takes as input the graph $\mathcal{G}$, the trace start time $t \sim T_G(\mathcal{G})$, a chain C, and the metadata of the first node of C, which is represented using one-hot encoding for the categorical columns and a variational Gaussian mixture model (VGM) [6] for the continuous columns. It outputs the metadata of the remaining empty nodes in the chain.

Using the tabular CTGAN instead of a time series model to fill in the chains reduces the size of the generative model by *orders of magnitude*. For example, a comparable time series model (e.g., NetShare [70]) exhibits a storage cost of 60 MB, compared to the 695 KB model size of CTGAN.

Generation Phase. The decompressor receives the generator C_G, and fill in blank nodes' metadata in the interior of the DAG (steps 5 and 6 in Figure 4). Starting with chains that emanate from the DAG's root(s), it generates metadata for all empty nodes in the chain(s); recall that the root nodes' metadata is already set from part (3). The decompressor then identifies chains with filled head nodes but empty subsequent nodes' metadata, and uses C_G to complete them.

We ensure consistent chain construction based purely on topology. Any incomplete chains are padded with null values to maintain the structure. If a chain merges with a previously-filled node, we use the latest-generated metadata.

The end-to-end view of our setup in Opentelemetry, along with ablation tests and illustrations to our insights, can be found in Appendix B.

4 Evaluation

4.1 Experimental Setup

Datasets. In our experiments, we utilized two popular demo applications: Wildrydes [68], a serverless practices tutorial endorsed by AWS, and HotROD [32], Jaeger's primary application for distributed tracing demonstrations. Using AWS Distro for Opentelemetry [4],we collected spans into our dataset, stored in an SQLite database totaling about 1GB of trace data. We generated four distinct datasets for different workloads, each embodying distinct characteristics (such as variable error rates and high rates of trace counts). Each workload has its dedicated SQL table. [2]

Baselines. We tested Gen-T against several baselines: (1) Head Sampling without Compression: This approach aligns with default OTEL Protocol settings. (2) Gzip with Head Sampling: This combines Gzip, a general-propose compression method, with head sampling. Gzip is cited in OTEL official benchmark [52] for its superior compression ratio, compared to other supported methods like *snappy* and *zstd*. (3) Compressed Log Processor [58]: A recently proposed domain-specific lossless compression method. In practice, we used the python logging library's file handler [10], measuring the resulting file final size. We did not compare against other lossy compression baselines, as we are unaware of techniques that are compatible with o11y traces.

Evaluated Resources. We capture resource measurements using *NVIDIA V100* on a *n1-standard-8* Google Cloud instance with 8 CPUs and 30GB memory. As of September 2023, its *us-west1* region cost is $0.38/hour on-demand and $0.08/hour for a spot instance.

Fidelity Metrics. Measuring the fidelity of generative models is difficult [8]. O11y trace data presents additional structural challenges and general-purpose metrics like FID score [29] are not meaningful in an o11y context. We use a suite of realistic queries detailed in Appendix A. These queries are used only for fidelity measurement and do not influence model training. We consider two user stories for these evaluation. In the first user story, *Post-release Testing*, a user evaluates the impact of an entry point

[2] We are releasing the dataset open-source at `https://gen-t-code.s3.us-west-2.amazonaws.com/traces.zip`.

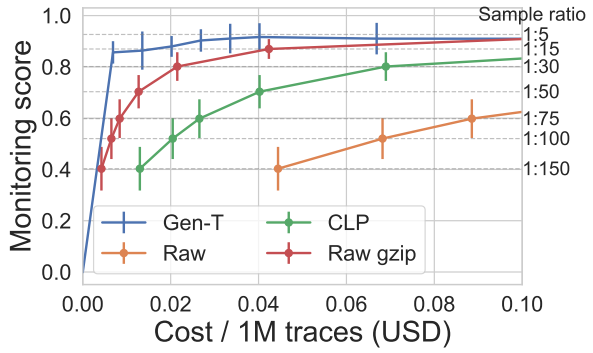

Figure 5: Gen-T E2E comparison to head sampling with gzip. Cribl [12] prices 1M traces ($\sim 2.9GB$) at 0.85$. Monitoring score measures the results from Query 3 as an F1 score indicating whether generated data contains the same events as the real data. Higher is better.

by studying all originating traces from a specific service. The goals include identifying all initiated subtasks (Query 1) and obtaining a cost breakdown (Query 2). In the second user story, Monitor and Troubleshoot, a user sets up a trace-based alert for a particular entry point (Query 3). An activated alert prompts the user to investigate the root cause by plotting histograms of every attribute within the entry point for traces with and without errors (Query 4).

4.2 End-to-End Cost-Fidelity Tradeoff

Gen-T offers high fidelity at a fraction of the operational cost of existing techniques. Our first experimental scenario depicts an observed system generating 15K traces per minute. These traces are aggregated every minute, and transmitted using Gen-T throughout the observability pipeline. We vary the number of training epochs for the generative models, resulting in a trade-off between fidelity and operational cost.

Our cost estimation is based on two two factors: GPU processing time and transmission charges. We price the GPU usage at $0.25 per hour, consistent with GCP rate. Transmission is priced at $0.32 per GB, following Cribl's rates [12].

Figure 5 shows fidelity for a monitoring use case, measured via Query 3. We run the experiment 3 times; the error bars show standard error. Gen-T achieves a level of fidelity comparable to that of 1:15 sampling while maintaining a cost profile equivalent to that of 1:100 lossless-compressed sampling.

Concretely, the left-most Gen-T data point introduces additional GPU and processing time of under a minute over the standard OTEL processing. This leads to an average total added latency of 1.5 minutes, considering both compression aggregation and processing and a single decompression in the decompressor. The total size transmitted in each setup averaged 695KB.

Interestingly, despite CLP being designed for observability logs, its compression efficiency falls short of broader methods like gzip. This could be due to the many unique identifiers present in each trace, potentially interrupting the anticipated log patterns of CLP. Moreover, CLP's objective to offer queryable compressed data introduces constraints that are not necessary for other applications.

Gen-T rolling model adapts to changes as well as 1:5 sampling, at a fraction of the cost. Microservice environments are dynamic, and commonly encounter changes in trace characteristics, such as spikes in the number of traces and spikes in errors. Figure 6 presents the model fidelity when these changes are introduced, for four different use cases (and their corresponding metrics).

To simplify the evaluation, we assume that each of these changes happens within its own distinct period of aggregated traces, and we allow Gen-T to re-train on the altered data. The horizontal axis in Figure 6 demonstrates the (sequential) time periods, each characterized by a different dataset.

Figure 6 demonstrates that Gen-T is able to adapt to substantial changes in the microservice environment. This level of adaptation is comparable to 1:5 sampling, all while maintaining a consistent transmission volume of 695KB.

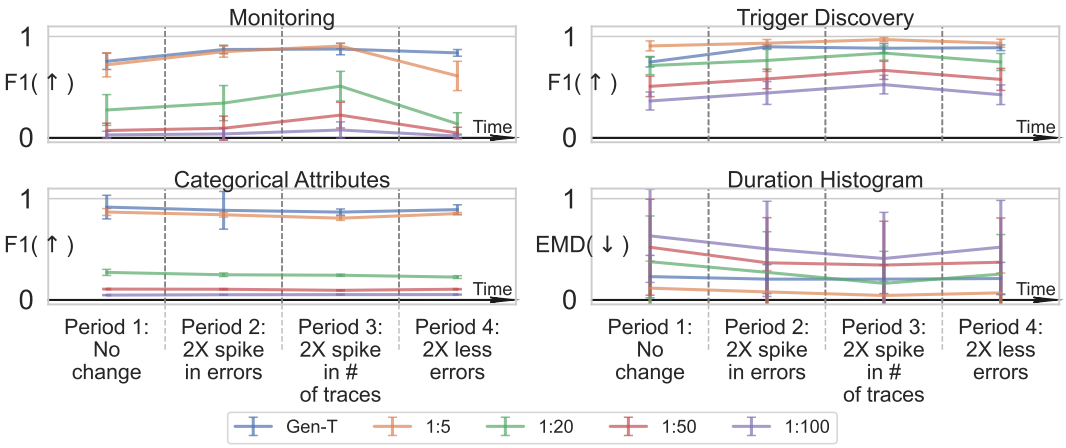

Figure 6: We introduced significant workload changes at different points in time. Gen-T's rolling model swiftly adapts to changes in the observed systems, with a fidelity similar to 1:5 sampling at a constant the data volume of less than 700KB.

5 Related Work

Recent studies emphasize generative models tailored for small graphs [28], often used in molecular design [5, 16, 59]. Unlike prior work focusing on novelty and validity, we aim for accurate system representation. In this work, we build on the iterative graph-building methodologies presented in [27, 39, 73], but focus on generating high-dimensional independent small graphs. Parallelly, generative models tailored for timeseries data have gained traction in areas such as networking [41, 70, 56] and sensors [7, 2], showcasing their ability to mimic various temporal patterns.

Monitoring and troubleshooting in systems is an expansive research area. Early efforts like [1] leveraged sampled events to retrace system pathways and extract performance insights. The distributed tracing concept evolved with X-Trace [19] and was augmented by solutions like PivotTracing [43], which employed scalable map-reduce storage techniques to amplify observability. Various machine learning investigations have focused on performance and metric collection [21, 22, 23, 26, 62, 61, 38, 34]. More recent research offers diverse perspectives on distributed tracing. For example, [9] proposes dynamic instrumentation approaches, while Sifter [37] and [36] introduce real-time anomaly trace identification methods. Finally, closely related to our work, Hindsight [72] suggests retrospective data collection upon error detection. However, this method may face challenges, especially in serverless or other low-resource microservice environments. Still, a noticable gap persists in research designed to transmit the full trace data for in-depth, flexible query observability.

6 Conclusions

The microservice and o11y industry is at a crossroads as the cost of monitoring a complex system is comparable or worse than that of building the system [51, 13]. This forces DevOps workflows to make undesirable tradeoffs losing visibility due to cost concerns, especially in fine-grained tracing scenarios. Our work offers operators a pragmatic alternative via generative modeling. Our key contribution is a structure-aware synthesis of generative modeling ideas and a backwards-compatible implementation to support fine-grained tracing use cases at a fraction of the cost of traditional approaches.

References

[1] Marcos K Aguilera, Jeffrey C Mogul, Janet L Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. *ACM SIGOPS Operating Systems Review*, 37(5):74–89, 2003.

[2] Moustafa Alzantot, Supriyo Chakraborty, and Mani Srivastava. Sensegen: A deep learning architecture for synthetic sensor data generation. In *2017 IEEE International Conference on Pervasive*

Computing and Communications Workshops (PerCom Workshops), pages 188–193. IEEE, 2017.

[3] Apache Cassandra. https://cassandra.apache.org/.

[4] AWS Distro for OpenTelemetry Lambda. https://aws-otel.github.io/docs/getting-started/lambda/lambda-python.

[5] Camille Bilodeau, Wengong Jin, Tommi Jaakkola, Regina Barzilay, and Klavs F Jensen. Generative models for molecular discovery: Recent advances and challenges. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 12(5):e1608, 2022.

[6] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

[7] Ashish Bora, Ajil Jalal, Eric Price, and Alexandros G Dimakis. Compressed sensing using generative models. In *International conference on machine learning*, pages 537–546. PMLR, 2017.

[8] Ali Borji. Pros and cons of gan evaluation measures: New developments. *Computer Vision and Image Understanding*, 215:103329, 2022.

[9] Lucas Castanheira, Theophilus A Benson, and Alberto Schaeffer-Filho. The case for more flexible distributed tracing. In *Proceedings of the Student Workshop*, pages 27–28, 2020.

[10] CLP Python Logging Library. https://github.com/y-scope/clp-loglib-py.

[11] Coca-Cola Freestyle Using AWS Lambda. https://aws.amazon.com/solutions/case-studies/coca-cola-freestyle/.

[12] Cribl. https://cribl.io/.

[13] Datadog's $65M Bill and Why Developers Should Care. https://thenewstack.io/datadogs-65m-bill-and-why-\developers-should-care/.

[14] Datadog Flame Graph Overview. https://www.datadoghq.com/knowledge-center/distributed-tracing/flame-graph/.

[15] Benefits and Challenges of Distributed Tracing. https://www.datadoghq.com/knowledge-center/distributed-tracing/#benefits-and-challenges-of-distributed-tracing.

[16] Nicola De Cao and Thomas Kipf. Molgan: An implicit generative model for small molecular graphs. *arXiv preprint arXiv:1805.11973*, 2018.

[17] Elastic: Transaction Sampling. https://www.elastic.co/guide/en/apm/guide/current/sampling.html.

[18] Elasticsearch. https://www.elastic.co/.

[19] Rodrigo Fonseca, George Porter, Randy H Katz, and Scott Shenker. X-trace: A pervasive network tracing framework. In *4th {USENIX} Symposium on Networked Systems Design & Implementation ({NSDI} 07)*, 2007.

[20] Hin Hark Gan, Daniela Fera, Julie Zorn, Nahum Shiffeldrim, Michael Tang, Uri Laserson, Namhee Kim, and Tamar Schlick. Rag: Rna-as-graphs database—concepts, analysis, and features. *Nutrition and Health*, 5(1-2):1285–1291, 1987.

[21] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. Sage: practical and scalable ml-driven performance debugging in microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 135–151, 2021.

[22] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*, pages 19–33, 2019.

[23] Anshul Gandhi, Parijat Dube, Alexei Karve, Andrzej Kochut, and Li Zhang. Adaptive, model-driven autoscaling for cloud applications. In *11th International Conference on Autonomic Computing ({ICAC} 14)*, pages 57–64, 2014.

[24] Nikhil Goyal, Harsh Vardhan Jain, and Sayan Ranu. Graphgen: A scalable approach to domain-agnostic labeled graph generation. In *Proceedings of The Web Conference 2020*, pages 1253–1263, 2020.

[25] Grafana Cloud Traces. `https://grafana.com/products/cloud/traces/`.

[26] Mark Grechanik, Chen Fu, and Qing Xie. Automatically finding performance problems with feedback-directed learning software testing. In *2012 34th international conference on software engineering (ICSE)*, pages 156–166. IEEE, 2012.

[27] Aditya Grover, Aaron Zweig, and Stefano Ermon. Graphite: Iterative generative modeling of graphs. In *International conference on machine learning*, pages 2434–2444. PMLR, 2019.

[28] Xiaojie Guo and Liang Zhao. A systematic survey on deep generative models for graph generation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2022.

[29] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium. *Advances in neural information processing systems*, 30, 2017.

[30] Jaeger. `https://www.jaegertracing.io/`.

[31] Jaeger Features. `https://www.jaegertracing.io/docs/1.46/getting-started/#features`.

[32] Jaeger Sample App: HotROD. `https://www.jaegertracing.io/docs/next-release/getting-started/#sample-app-hotrod`.

[33] Jaeger Sampling. `https://www.jaegertracing.io/docs/1.46/sampling/`.

[34] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, et al. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th symposium on operating systems principles*, pages 34–50, 2017.

[35] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[36] Pedro Las-Casas, Jonathan Mace, Dorgival Guedes, and Rodrigo Fonseca. Weighted sampling of execution traces: Capturing more needles and less hay. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 326–332, 2018.

[37] Pedro Las-Casas, Giorgi Papakerashvili, Vaastav Anand, and Jonathan Mace. Sifter: Scalable sampling for distributed traces, without feature engineering. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 312–324, 2019.

[38] Joshua Levin and Theophilus A Benson. Viperprobe: Rethinking microservice observability with ebpf. In *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*, pages 1–8. IEEE, 2020.

[39] Yujia Li, Oriol Vinyals, Chris Dyer, Razvan Pascanu, and Peter Battaglia. Learning deep generative models of graphs. *arXiv preprint arXiv:1803.03324*, 2018.

[40] Renjie Liao, Yujia Li, Yang Song, Shenlong Wang, Will Hamilton, David K Duvenaud, Raquel Urtasun, and Richard Zemel. Efficient graph generation with graph recurrent attention networks. *Advances in neural information processing systems*, 32, 2019.

[41] Zinan Lin, Alankar Jain, Chen Wang, Giulia Fanti, and Vyas Sekar. Using gans for sharing networked time series data: Challenges, initial promise, and open questions. In *Proceedings of the ACM Internet Measurement Conference*, IMC '20, page 464–483, New York, NY, USA, 2020. Association for Computing Machinery.

[42] Logz.io - What can I do in the Trace Graph? https://docs.logz.io/user-guide/distributed-tracing/trace-graph.

[43] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 378–393, 2015.

[44] The Netflix Cosmos Platform. https://netflixtechblog.com/the-netflix-cosmos-platform-35c14d9351ad.

[45] Building Netflix's Distributed Tracing Infrastructure. https://netflixtechblog.com/building-netflixs-distributed-tracing-\infrastructure-bb856c319304.

[46] New Relic - Understand the trace details UI page. https://docs.newrelic.com/docs/distributed-tracing/ui-data/trace-details/.

[47] Observability Platform Market. https://www.futuremarketinsights.com/reports/observability-platform-market.

[48] Observability Tools and Platforms Market worth $4.1 billion by 2028. https://www.prnewswire.com/news-releases/observability-tools-and-platforms-market\-worth-4-1-billion-by-2028---exclusive-report\-by-marketsandmarkets-301877412.html.

[49] OpenObserve. https://openobserve.ai/.

[50] Opentelemetry. https://opentelemetry.io/.

[51] O'Reilly: Overhead, Costs, and Sampling. https://www.oreilly.com/library/view/distributed-tracing-in/9781492056621/ch06.html.

[52] Opentelemetry Collector - Compression Comparison. https://github.com/open-telemetry/opentelemetry-collector/blob/main/config/configgrpc/README.md#compression-comparison.

[53] OpenTelemetry Sampling. https://opentelemetry.io/docs/concepts/sampling/.

[54] Opentelemetry Scenarios. https://opentelemetry.io/docs/demo/scenarios/.

[55] Opentelemetry Collector. https://opentelemetry.io/docs/collector/.

[56] Inkit Padhi, Yair Schiff, Igor Melnyk, Mattia Rigotti, Youssef Mroueh, Pierre Dognin, Jerret Ross, Ravi Nair, and Erik Altman. Tabular transformers for modeling multivariate time series, 2020.

[57] Quickwit. https://quickwit.io/.

[58] Kirk Rodrigues, Yu Luo, and Ding Yuan. {CLP}: Efficient and scalable search on compressed text logs. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pages 183–198, 2021.

[59] Bidisha Samanta, Abir De, Gourhari Jana, Vicenç Gómez, Pratim Kumar Chattaraj, Niloy Ganguly, and Manuel Gomez-Rodriguez. Nevae: A deep generative model for molecular graphs. *The Journal of Machine Learning Research*, 21(1):4556–4588, 2020.

[60] Shibani Santurkar, David Budden, and Nir Shavit. Generative compression. In *2018 Picture Coding Symposium (PCS)*, pages 258–262. IEEE, 2018.

[61] Junxian Shen, Han Zhang, Yang Xiang, Xingang Shi, Xinrui Li, Yunxi Shen, Zijian Zhang, Yongxiang Wu, Xia Yin, Jilong Wang, et al. Network-centric distributed tracing with deepflow: Troubleshooting your microservices in zero code. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 420–437, 2023.

[62] Yongmin Tan, Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Chitra Venkatramani, and Deepak Rajan. Prepare: Predictive performance anomaly prevention for virtualized cloud systems. In *2012 IEEE 32nd International Conference on Distributed Computing Systems*, pages 285–294. IEEE, 2012.

[63] Lucas Theis, Aäron van den Oord, and Matthias Bethge. A note on the evaluation of generative models. *arXiv preprint arXiv:1511.01844*, 2015.

[64] What You Need to Know About Distributed Tracing and Sampling. https://thenewstack.io/what-you-need-to-know-about-distributed-\tracing-and-sampling/.

[65] T-Mobile's Serverless Development Platform. https://opensource.t-mobile.com/blog/posts/introducing-jazz/.

[66] Twitter: Distributed Systems Tracing with Zipkin. https://blog.twitter.com/engineering/en_us/a/2012/distributed-systems-tracing-with-zipkin.

[67] Evolving Distributed Tracing at Uber Engineering. https://www.uber.com/en-IL/blog/distributed-tracing/.

[68] AWS decoupled services tutorial. https://catalog.us-east-1.prod.workshops.aws/workshops/e8738cf6-6eb0-4d1d-9e98-ae240d229535/en-US.

[69] Lei Xu, Maria Skoularidou, Alfredo Cuesta-Infante, and Kalyan Veeramachaneni. Modeling tabular data using conditional gan. *Advances in neural information processing systems*, 32, 2019.

[70] Yucheng Yin, Zinan Lin, Minhao Jin, Giulia Fanti, and Vyas Sekar. Practical gan-based synthetic ip header trace generation using netshare. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 458–472, 2022.

[71] Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing*, 18(6):1245–1262, 1989.

[72] Lei Zhang, Zhiqiang Xie, Vaastav Anand, Ymir Vigfusson, and Jonathan Mace. The benefit of hindsight: Tracing {Edge-Cases} in distributed systems. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 321–339, 2023.

[73] Dawei Zhou, Lecheng Zheng, Jiawei Han, and Jingrui He. A data-driven graph generative model for temporal interaction networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 401–411, 2020.

[74] Zipkin. https://zipkin.io/.

A Queries for Fidelity Evaluation

We use the following queries to evaluate our results in section 4.

```
SELECT DISTINCT
    (S1.serviceName, S2.serviceName)
FROM Spans as S1, Spans as S2
Where S1.spanId = S2.parentId
```

Query 1: Discover all the trigger correlations in the system

```
SELECT S2.serviceName, ROUND(
    S2.duration/S1.duration, 1
) AS ratio, count(*)
FROM Spans as S1, Spans as S2
WHERE S1.serviceName = 'billing'
  AND S1.traceId = S2.traceId
GROUP BY S2.serviceName, ratio;
```

Query 2: Find the relative duration histogram of every sub-task of the billing service

Context	Layer	Neurons	Activation	T_G	R_G	C_G
Start Time Embedding	Passthrough	9	N/A	-	+	+
Metadata Embedding	Fully connected	32	Relu	-	-	+
Graph Embedding	GCNConv	32	N/A	+	+	+
Noise Embedding	Fully connected	128	Relu	+	+	+
CTGAN Generator	Fully connected	128	BatchNorm1d, Relu	+	+	+
CTGAN Generator	Fully connected	128	tanh	+	+	+
CTGAN Discriminator	Fully connected	128	LeakyRelu(0.2), drop(0.5)	+	+	+
CTGAN Discriminator	Fully connected	1	N/A	+	+	+

Table 2: The layers configuration in our generators. T_G is the start-time generator, R_G is the root node metadata generator and C_G is the chain metadata generator.

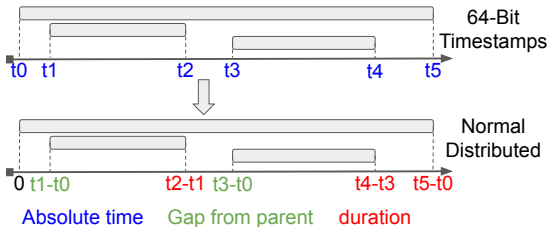

Figure 7: Decoupling timestamps from high-dimensional metadata enables parallel training, which increases GPU utilization and allows data-specific enhancements.

```
SELECT count(*) > 10
FROM Spans as S1, Spans as S2
Where
    S1.serviceName = 'billing'
    AND S1.traceId = S2.traceId
    AND S2.error IS NOT NULL
```

Query 3: Binary query on traces, to decide whether to send an alarm

```
SELECT S1.attr1, count(*)
FROM Spans as S1, Spans as S2
Where
    S1.serviceName = 'billing'
    AND S1.traceId = S2.traceId
    AND S2.error IS NOT NULL
GROUP BY S1.attr1
```

Query 4: Diversity query to find histogram of entry point's attributes for failed traces

B Detailed Design

B.1 Ablation Test

To assess the impact of the insights outlined in section 3, we evaluated our model across three different configurations:
Trace Topology Conditioning: In this scenario, we used One-Hot encoding instead of GCNConv to condition the trace topology. Such a configuration masks the individual components of the trace topology. This means the model lacks the ability to discern proximity between similar or intersecting topologies, forcing it to relearn similar triggers for every distinct topology. This evaluation showcase that this approach boost fidelity across some metrics by as much as 20%.
Hierarchical Correlation: We evaluated the fidelity when omitting the data of the triggering node. In this configuration, each row in the tabular data represented a single node from the graph. Although a consistent traversal on the graph provides implicit hierarchy for chain building, this configuration deprives the model of the explicit hierarchical relationships.

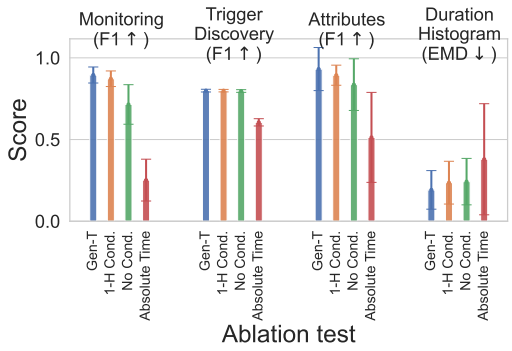

Figure 8: The effect of our insights on the fidelity, evaluated on metrics defined in section 4

.

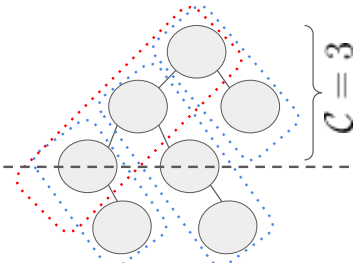

Figure 9: Our approach transforms the DAG's hierarchical dependencies into a union of bounded-size chains. We then generate each chain's metadata conditioned on the pre-existing metadata of the chain head. Smaller chain lengths reduce the dimensionality of the data that needs to be generated (i.e., they have lower training and volume cost), while longer chains improve fidelity by capturing longer-range correlations across the graph.

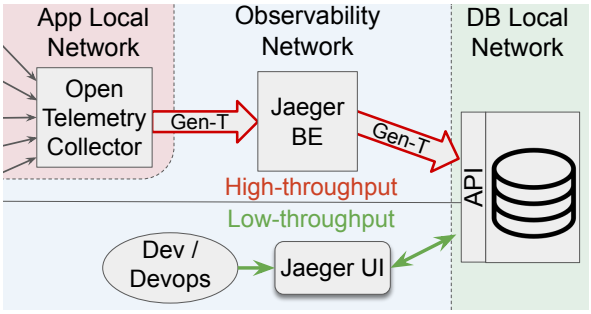

Figure 10: The observability flow. Gen-T replaces existing high-throughput links with a new transmission protocol

Timestamp Handling: We show a decline in fidelity when the absolute timestamps weren't separated and instead included them as a part of the high-dimensional metadata for the nodes.
The results clearly demonstrate that each of these design decisions plays a pivotal role in enhancing fidelity. Notably, the metric related to trigger discovery (Query 1), which relies solely on timeseries data and trace topology, remains unaffected by manipulations in metadata generation. Furthermore, it is evident that the remaining fidelity metrics are influenced by both timeseries and metadata properties. Notably, the utilization of One-Hot encoding for trace topology, while only slightly inferior to Gen-T, introduces a tradeoff by reducing transmission size by 3% and GPU time by 14%.

B.2 End-to-End View

Our end-to-end vision is illustrated in Figure 10.

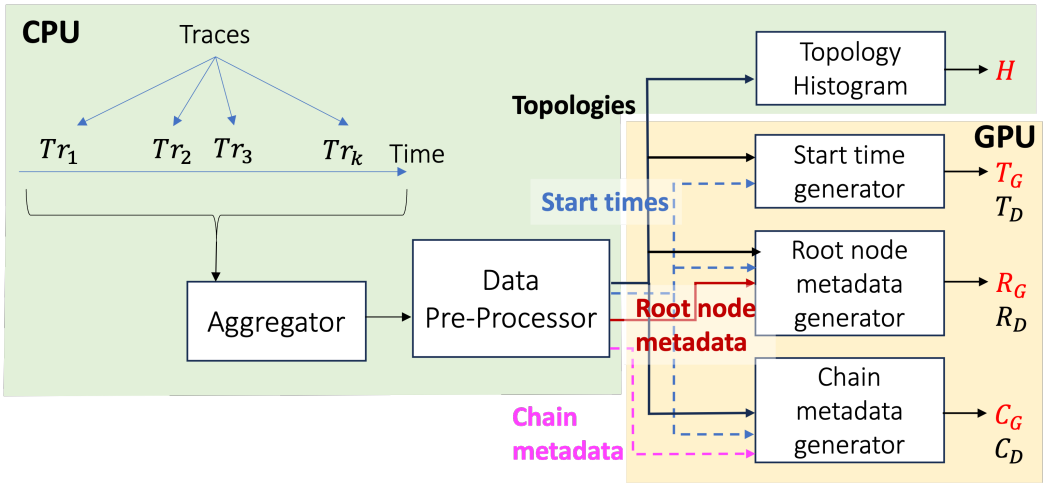

Figure 11: Gen-T learning pipeline. Traces are aggregated and split into sub-datasets, which are used to learn/train models in parallel.

Compression Phase. Traditional observability pipelines typically process individual spans in a continuous stream, whereas our approach operates on dataset of traces. We thus aggregate the trace stream prior to learning the relevant models and structures. This data is then preprocessed into the four sub-datasets: topology histogram, traces start time, root node metadata, and chain metadata. These sub-datasets are passed to their respective training modules (Figure 11). All generative models are learned in parallel on GPU, which is the bottleneck in wall-clock time and cost. Due to this parallelization, combined with the optimizations described in section 3, we empirically achieve a GPU utilization of 90% during training (30% per generative model).

To run Gen-T continuously, we aggregate traces at set intervals or sizes (e.g., 20K traces in section 4). We warm-start the metadata generators using the previous period's learned model weights, whereas the start time generator (and of course the topology histogram) are best re-learned from scratch.

After training, the structures shown in red text in Figure 11 are transmitted to the target. Prior to transmission, we use gzip to further compress, for an additional $\sim 60\%$ reduction in volume in our experiments. We additionally re-generate the data at the compressor side with different random seeds to find the seed with the best data quality. This seed is transmitted along with the compressed model parameters.

Generation Phase. The target decompresses the received models and uses them to re-generate a representative synthetic dataset. The target consistently uses the transmitted seed to generate identical data every time, ensuring alignment between the observability system and the database.

B.3 Implementation

We implemented Gen-T in 2,300 lines of *python3.10* code, incorporating *torch 1.11.0* and *CUDA 11.0*. Our underlying tabular generative model is *CTGAN 0.7.4* [56], which we extended to support conditional generation (320 LoC) as discussed in section 3. Gen-T can be seamlessly integrated Gen-T into existing o11y flows; e.g., our integration with Opentelemtry Collector [55] and Jaeger [30] is a simple configuration flag. Our compression phase is invoked when the OTEL Collector transmits the spans to the o11y system and when the system forwards the spans to the database. Similarly, generation is invoked when the o11y system or database receives the spans.

Our default configuration runs for 10 epochs, features a chain of length $\mathcal{C} = 2$, and utilizes CTGAN's generator with a single residual layer of dimension 128. Our full source code and dataset are made available at `https://gen-t-code.s3.us-west-2.amazonaws.com/gen-t-code.zip`.