Scribe: How Meta transports terabytes per second in real time

Manos Karpathiotakis
Meta Platforms Inc.
manos@meta.com

Vlassios Rizopoulos
Meta Platforms Inc.
vlassios@meta.com

Basri Kahveci
Meta Platforms Inc.
basrikahveci@meta.com

Tiziano Carotti
Meta Platforms Inc.
ivendor@meta.com

Artem Gelum
Meta Platforms Inc.
agelun@meta.com

Hazem Nada
Meta Platforms Inc.
hazemnada@meta.com

Yuri Dolgov
Meta Platforms Inc.
ydolgov@meta.com

ABSTRACT

Millions of web servers and a multitude of applications are producing ever-increasing amounts of data in real time at Meta. Regardless of how data is generated and how it is processed, there is a need for infrastructure that can accommodate the transport of arbitrarily large data streams from their generation location to their processing location with low latency.

This paper presents Scribe, a multi-tenant message queue service that natively supports the requirements of Meta's data-intensive applications, ingesting $> 15TB/s$ and serving $> 110TB/s$ to its consumers. Scribe relies on a multi-hop write path and opportunistic data placement to maximise write availability, whereas its read path adapts replica placement and representation based on the incoming workload as a means to minimise resource consumption for both Scribe and its downstreams. The wide range of Scribe use cases can pick from a range of offered guarantees, based on the trade-offs favourable for each one.

PVLDB Reference Format:
Manos Karpathiotakis, Vlassios Rizopoulos, Basri Kahveci, Tiziano Carotti, Artem Gelum, Hazem Nada, and Yuri Dolgov. Scribe: How Meta transports terabytes per second in real time. PVLDB, 18(12): 4817 - 4830, 2025.
doi:10.14778/3750601.3750607

1 INTRODUCTION

Data generation at Meta is on a continuous growth trajectory. Whether an explicit or implicit byproduct of organic user activity received by millions of web servers, or of a multitude of other sources, such as service health telemetry, the volume and variety of data sources and inputs that Meta systems encounter is ever increasing. Processing of the resulting datasets typically takes place via numerous types of data processing infrastructure, such as data warehouse queries [43], stream processing pipelines [11], or model training pipelines [51]. A common theme across all of them is the need to continuously transport datasets at a rate of multiple terabytes per second from their data generation services to their data processing services.

Regardless of whether data processing services adopt a lambda model combining the batch and streaming paradigms, or a pure streaming one, transporting inputs to them typically involves the use of a message queue [1–3, 17, 27, 39, 46]. For any message queue solution to be viable for the properties and scale of Meta, it would need to address the following requirements – some competing against each other:

(1) **Write availability:** Ensure that hundreds of millions of heterogeneous data producers are successful in persisting messages until consumers can process them; treat inputs as the company's golden source of truth.
(2) **Multi-tenancy:** Expose the abstraction of independent data streams with dedicated resources, on top of a shared infrastructure. Mask out heterogeneity in terms of factors such as traffic volume and shape, message sizes, fan-in / fan-out, speed of consumption, real-time vs. "historical" data consumption, and criticality levels.
(3) **High (global) read fanout:** Accommodate the presence of hundreds of consumers, spawning across multiple data center regions, for the same logical dataset.
(4) **"No-ETL" data consumption:** Allow real-time data consumers to read and process only their data subsets of interest, with a minimal deserialization tax.
(5) **One size does not fit all:** Serve heterogeneous use cases that have varying requirements in terms of ordering and delivery guarantees.

This paper presents Scribe, the message queue service responsible for making data available with high throughput and low latency to Meta's multitude of batch and realtime systems. The architecture of Scribe has evolved over the past 18 years to cater to all of the requirements above. Specifically:

(1) A multi-hop write path uses buffering and non-deterministic routing / placement of data to maximize **write availability**.
(2) Traffic shaping and quota management mechanisms protect the performance of individual requests in a **multi-tenant** environment, while the write and read paths minimize the impact of individual resource-demanding requests on any given host of the Scribe data plane.

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750607

(3) On-demand replication of the popular data tail end on ephemeral storage and integration with the job schedulers of prominent downstream compute platforms keep the impact of **high global read fanout** in check.
(4) Population of oft-used materialized (columnar) views minimizes the volume of transported data and the associated deserialization "tax" [24]. Combined with the ability to dynamically partition input datasets as well as preserve interesting orders in input batches, Scribe enables **lightweight, scalable ETL** for the downstream analytical systems.
(5) Exposing trade-offs around freshness, scalability, resource consumption, and delivery guarantees to customers enables them to opt for the **offering best "sized" for their needs**.

Contributions. The contributions of this work are the following:

- We present the architecture of Scribe, the message queue service that acts as Meta's data transport layer, and highlight the key properties that have empowered Scribe to scale to $> 15TB/s$ of ingress and $> 110TB/s$ of egress.
- We present learnings distilled from the 18-year evolution of Scribe, and retrospect on past decisions and their outcomes.

The rest of this work is structured as follows: Section 2 presents the high-level abstractions that Scribe exposes to its users. Section 3 outlines the end-to-end architecture, whereas Sections 4 and 5 dive deeper into the data and control planes, respectively. Section 6 discusses how we gradually introduced stronger guarantees on top of Scribe. Section 7 discusses learnings based on the evolution of Scribe over the past 18 years. Finally, Section 8 concludes the paper.

2 SCRIBE CONCEPTS

Scribe exposes the concept of a logical stream, called a *category*, to which *producers* can write messages and which *consumers* can read.

Writing messages: Scribe customers write messages to a category using a *Producer* library. While Scribe historically had allowed for messages to contain arbitrary payloads and had treated every message as an opaque blob regardless of its serialization (e.g., CSV, JSON, Thrift), nowadays the vast majority of Scribe traffic follows an explicit schema – published in a globally available registry – and a Thrift-based, schema-evolution-friendly serialization [48]. Customers can also annotate their messages with auxiliary metadata for purposes such as tracing, privacy enforcement, and filtering.

Callers of the *Producer* library can issue unary / N-ary calls: In the case of N-ary calls, Scribe allows for a message batch to use serialization formats that are more suitable for subsequent analytical processing, such as PrestoPage [8] – the wire format that the Velox [38] query engine is most compatible with. In addition, Scribe exposes the ability to treat the provided message batch as *atomic*, guaranteeing that the Scribe write path will store the messages of the batch contiguously and that the consumers will also receive the messages contiguously.

Durability: The messages that producers write to a category are durable and available to consumers for a configurable retention period: applications can thus decouple their data production from their data consumption rate, as well as recover from events such as outages and corruptions by reading the historical portion of the category's data stream.

Reading messages: Callers of the *Consumer* library read a category in streaming fashion starting from an arbitrary time point within the category's retention – although the majority of consumers are reading the tail end of a category's data – or resume from where they had previously left off by providing a *checkpoint*. Consumers can optionally specify a SQL expression to filter out a portion of the category, as well as specify a desirable *output* serialization format (e.g., request the output to be expressed as PrestoPage or JSON, even if the original messages had been Thrift-encoded).

Sharding: A voluminous category typically requires a group of consumers to consume it in its entirety – with said consumers relying on a *sharding scheme* to distribute the category among them. Customers can partition a category into multiple *logical shards* based on a key value provided at write time, or based on the values of a message's (sub)set of fields. A given consumer can then opt to read only the messages corresponding to a given logical shard.

For use cases where affinitizing messages of a given sharding key to a specific consumer is unnecessary, Scribe allows a consumer group to collectively read a category by distributing the whole traffic to them in arbitrary fashion: The group's only concern then becomes how to balance the category's traffic among its consumers. Finally, Scribe provides interfaces that allow users to rightsize a consumer group's size based on the current traffic volume.

3 THE ARCHITECTURE OF SCRIBE: A 10000 FOOT VIEW

Scribe addresses the need for reliable streaming data transport of voluminous datasets continuously generated across Meta's millions of geo-distributed hosts. As the scale of Meta kept increasing and its overall data ecosystem became more refined, Scribe underwent multiple architectural incarnations to accommodate novel requirements. Figure 1 provides a high-level view of Scribe's current architecture.

Write path: Scribe's utmost priority has always been the buffering and transportation of the datasets comprising Meta's "golden source of truth" with low-latency and high-throughput: An outage of the Scribe write path would result in loss of data for the pipelines powering ingestion into Meta's data warehouse, realtime monitoring [22], ads pacing, and numerous other use cases. As a result, write availability has been Scribe's main design criterion ever since its first architectural incarnation.

The *Producer* library is the main entry point for writing messages to a category. Meta applications and services written in various languages (e.g., C++, Python, PHP) use corresponding Producer bindings for writing to Scribe. The next hop in the Scribe write path is *ScribeD* – a local daemon running on every host in Meta. ScribeD is responsible for receiving messages from all the Producer instances on a host, buffering them, and eventually sending said messages to Scribe's first write backend service – the *Write Proxy*.

Once a Write Proxy instance receives an *inter-category* message batch, it performs various admission control checks as a first step, then demultiplexes the incoming batch into *per-category* batches, and proceeds to forward each to the *Batch Service*. The Batch Service accumulates batches which it flushes into durable storage. Scribe relies on a *decoupled metadata and payload storage*: Specifically, for a given message batch, the Batch Service compresses and flushes the corresponding message payloads to block storage; once

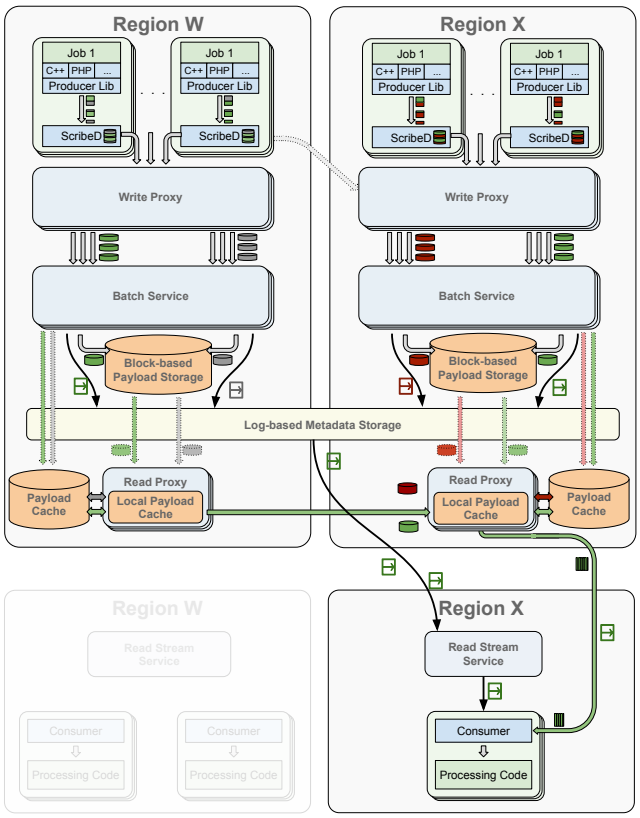

Figure 1: Scribe architecture.

successful, the Batch Service flushes metadata about the persisted batch, including a pointer to the block storage, to log-based metadata storage that will accommodate the sequential, stream-like read patterns of the Scribe consumers.

Read path: The *Consumer* library is the main entry point for reading messages of a category. Whenever an application starts a Consumer, the latter initiates a stateful connection to an instance of the *Read Stream Service* – the backend service responsible for exposing the abstraction of stream-like consumption to Consumers. The Read Stream Service proceeds to contact the log-based metadata storage and stream every metadata entry (and accompanying payload pointer) it retrieves to the Consumer. The Consumer, finally, "re-stitches" metadata and payloads by requesting the payloads from the *Read Proxy* – the service responsible for encapsulating all accesses to payload storage.

Beyond accessing durable payload storage, the Read Proxy populates what constitutes Scribe's *tiered, ephemeral payload storage*, in the form of a disaggregated cache tier as well as in-memory caches within Read Proxy hosts: Both cache tiers aim for minimizing accesses to the durable payload storage that is a scarcer resource and, most importantly, frequently located in a different geographical region than where the Consumer resides. Finally, the Read Proxy evaluates any filtering and serialization format transcoding requests that the Consumer has asked for.

Sections 4 and 5 dive deeper into all of the aforementioned components and logic, explain the motivation behind the architectural decisions, and detail the data and control planes, respectively.

4 THE DATA PLANE OF SCRIBE

The data plane of Scribe comprises relatively decoupled write and read paths: The write path is responsible for the persistence of records received via RPC-like requests, whereas the read path is responsible for exposing the abstraction of a data stream over the persisted records. The rest of this section goes into details on the write and read paths, and elaborates on how Scribe bridges their oft-times conflicting responsibilities.

4.1 Write Path

Applications that write messages into a category typically spawn a number of *Producers*, both within and across an arbitrary number of hosts. The write patterns can vary widely across different categories or even different hosts: A category can receive a trickle of messages via Producers residing across millions of web servers, while another category can be receiving hundreds of megabytes every second from a single host; the messages of a category can comprise a few bytes each, whereas another category can be receiving messages with sizes on the order of megabytes. The Scribe write path homogenises the diverse write patterns of the hundreds of millions of Producer instances, and aims to maximise write availability and efficiency of the overall service in the presence of a substantial fan-in factor.

The Producer library is the multi-tenant entry point to the Scribe write path. When a customer application issues a write call to the Producer library, the application receives a response object that it can use to inspect the result of the write operation. Each Producer instance accumulates incoming messages in a cross-category, multiplexed in-memory buffer in order to form large enough data batches: the use of a cross-category buffer rather than a per-category one aids in keeping a limited memory footprint, while the batching reduces the amount of RPCs to downstream write path components. Customers writing to Scribe in the hot path of their logic typically opt for a fire-and-forget write mode, relying on Scribe to flush their messages to durable storage in a best effort manner.

The Producer instances embedded in each binary running on a given host periodically flush their message buffers to ScribeD, a daemon process running on the same host. ScribeD uses similar criteria as the Producers for buffering messages and flushing them to the backend, and utilizes an *on disk* buffer to maintain write availability in case it is unable to communicate with the write backend. While Producer instances and ScribeD are heavily configurable, their default configuration values err on the conservative side in order to minimize the Scribe "resource tax" and any potential noisy neighbour effects: Being conservative is necessary given that multiple (e.g., hundreds) of Producer instances can run on the same host and that ScribeD is available on virtually every one of the millions of Meta hosts. Still, customers are allowed to tune Producers based on their needs and resource budget.

Conservative tuning aside, ScribeD acts as the first layer of fault tolerance to avoid message loss in the presence of unexpected events that prevent a message reaching durable Scribe storage, such as network outages or backend overloads.

Despite the provisions of ScribeD, a data loss event can still occur if ScribeD disk buffers fill up in the case of a prolonged outage, or if a host experiences a fault before ScribeD emitting its buffers to the Scribe write backend service. In addition, for use cases generating a high data throughput (e.g., 1 GB/s) per host, the on-disk buffer of ScribeD brings minimal value due to the limited local disk capacity: even short periods of ScribeD unavailability can lead to data loss. Thus, high throughput use cases often opt for skipping the ScribeD hop and make their Producers directly communicate with the Scribe Write Proxy to obtain higher availability guarantees.

Once a Producer / ScribeD instance is ready to offload a batch of messages, it issues a write request to the Write Proxy, a backend service with a tier deployed in every data center of Meta. The responsibilities of a Write Proxy instance are to perform basic admission control per received message, employ mechanisms to protect the write path from noisy neighbour problems, and route each category's messages to Batch Service . The eventual target of the Scribe write path is to accumulate a "large enough" amount of messages, typically in the order of 10MB, for storage efficiency purposes. Given, however, that the Write Proxy receives messages from hundreds of thousands of categories, it is non-trivial for the per-category buffers of a Write Proxy host to hold 10MB-worth of messages at a time, especially given that traffic between the write clients and the Write Proxy is opportunistic for load balancing and availability purposes.

Write Proxy instances thus forward per-category message batches to the Batch Service, for the latter to further buffer, compress, and flush them to Scribe storage. Compressibility of the per-category batches depends on accumulating as many *similar* messages as possible given a conservative memory budget. The hop between Write Proxy and Batch Service is thus the first place where the write path attempts to affinitize traffic dynamically via the following steps:

(1) continuously monitoring the incoming client traffic per (category, logical shard) pair
(2) logically splitting each (category, logical shard) pair's traffic into smaller "micro shards"
(3) using rendezvous hashing [45] to route a given micro-shard's messages towards a specific Batch Service instance to enable Batch Service to collect storage-efficient batches
(4) adapting the routing based on changes in the incoming traffic and availability of Batch Service instances to maintain the overall availability and efficiency of the write path

Having accumulated and compressed a batch of messages, the relevant Batch Service instance attempts flushing them to ephemeral payload storage (subject to the policies described in Section 5.4 and to durable payload storage. Upon success and reception of a pointer to the payload, Batch Service stores the payload pointer along with additional metadata to metadata storage. At this point, after having gone through multiple intermediate transient hops, Scribe considers a batch of messages as *persisted* to its storage.

4.2 Storage

For multiple years, Scribe relied on a log-based storage system as its sole storage solution, storing message payloads in append only files. The gradual transition to a decoupled model – with the log-based storage system storing sequences of pointers to external block storage – unlocked a number of benefits, some of the most prominent ones listed below:

- The ability to use a different storage type for the "hot" (realtime) and the "cold" (historical) portions of a category's data.
- The ability to retrofit stronger guarantees over a category's record sequence (e.g. read repeatability), stored across hundreds / thousands of physical shards, without having to access and manipulate voluminous data payloads; Section 6 offers more details.
- The ability to decouple stateful, stream-based data accesses to lightweight metadata information, from RPC-like accesses to block payload storage, subsequently enabling the straightforward introduction of numerous enhancements (e.g., *ephemeral replicas*, disaggregated compute) over said block storage.

The rest of this section discusses the components and services comprising Scribe's decoupled, tiered storage.

Log-based metadata storage. Scribe stores positional metadata for the message batches of a category in LogDevice [32], a distributed log storage system optimised for sequential data accesses. For a given category, LogDevice stores the positional metadata of each message batch as a record in append-only, trimmable files called *logs*: Each LogDevice log acts as a Scribe *physical shard*; each record within a log is identifiable by a monotonically increasing (log) sequence number, assigned by a *sequencer* entity. Multiple LogDevice logs (typically in the order of 1 million) are stored in a LogDevice *cluster*.

Akin to Scribe, a key tenet of LogDevice's architecture is write availability, which LogDevice accommodates through the use of non-deterministic data placement: Every LogDevice cluster comprises a set of storage nodes, and replicas (e.g., 5) of a record can end up in any of a large subset (e.g., 20) of said nodes – guaranteeing no single point of failure in terms of accepting write operations. Special care is built in the write protocol to ensure the storage nodes picked reside in different failure domains: Examples from a production environment include the nodes being i) powered by different main switch boards in a data center, ii) located in different data centers, or even iii) located in different geographical regions. Non-deterministic data placement at write time implies additional complexity on the read path; still a log's readers benefit from repeatable reads, and are guaranteed to receive records at the order of their sequence number. Finally, as stricter guarantees became relevant for a subset of Scribe traffic, LogDevice refined its write protocol to allow for i) linearizable writes and ii) exactly once writes, with each of the two enhanced modes being incrementally more rigid in terms of handling failure events of the sequencer entity.

Durable block (payload) storage. Scribe stores durably the payloads corresponding to a category's messages in Tectonic [37], the multitenant, distributed filesystem developed in Meta to replace HDFS. Besides Scribe, Tectonic backs multiple other services, such as Meta's data warehouse and media storage.

Scribe stores category payloads in Tectonic *files* residing in storage nodes that are backed by hard disk drives; a collection of storage nodes form a Tectonic *cluster*. Rather than maintaining N "full" replicas per payload, Tectonic allows for the use of Reed-Solomon [40] error correction to achieve similar redundancy properties, yet requiring a more involved reconstruction process in cases of a storage

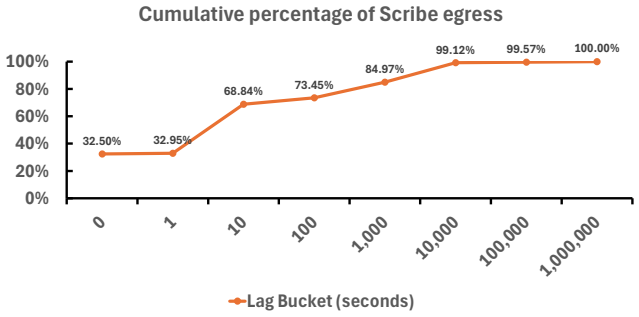

Figure 2: Buckets of time elapsed between the moment messages are written to Scribe and the moment a consumer reads them: ~85% of messages are consumed less than 20 minutes after being ingested; ~99% of messages are consumed less than 3 hours after being ingested.

node loss. Adapting to Tectonic properties and best practices, Scribe stores the payloads of multiple categories within a given file, in order to keep the number of said files under control. Likewise, Scribe flushes Tectonic *blocks* comprising payloads that can belong to multiple categories in order to avoid blocks of small size; by accumulating blocks in the order of 10s of megabytes, Scribe makes the flush operation disk-friendly and avoids bloating the per-block metadata that Tectonic maintains. Scribe also optimizes for the per-category payloads within each Tectonic block to maintain a size in the order of 2 megabytes because said batches are the typical Scribe unit of reading: Therefore, it is important for the read operation to be large enough to amortize the I/O seek operation involved with retrieving a payload from a hard disk drive.

Ephemeral block (payload) storage. Scribe uses a two-level ephemeral storage tier on top of its durable storage. While the use of replication is typically associated with fault tolerance, Scribe rather uses the ephemeral tier to optimize for the following reasons:

- Reduction of cross-region network traffic by populating payload replicas in a consumer's geographical region.
- Reduction of (more resource-intensive) IO operations over durable Tectonic storage.
- Transcoding payloads to a serialization format that is more favourable for downstream readers.
- Population of materialized views that are better suited for consumer queries.

Scribe relies on different caching technologies to form its two-level ephemeral storage tier. First, Memcache [35] – Meta's large-scale deployment of memcached [19] – acts as a disaggregated layer that holds payloads for a period in the order of 1-2 hours. Based on the Scribe read patterns depicted in Figure 2, said period is typically sufficient for interested readers to consume the "warm" tail end of a category. Memcache typically attempts to use the memory of the hosts comprising its tiers to store data entries, eventually offloading them to the hosts' solid-state drives.

Scribe workloads form an outlier for Memcache: While the majority of Memcache workloads are RPC-heavy and thus CPU-heavy, the size of Scribe payloads post-batching and the overall Scribe egress requirements make the workload bandwidth-heavy and thus NIC-heavy. In addition, given the streaming nature of Scribe workloads, most payloads end up promptly flushed to (and thus eventually retrieved from) the solid-state drives of Memcache tiers – thus also making the ability to efficiently serve Scribe workloads subject to the endurance thresholds of the Memcache hosts' solid state drives.

Scribe relaxes the strain on Memcache NIC and flash resources through the use of a second ephemeral storage tier, residing in Cachelib instances [9] that use the spare memory resources of Read Proxy hosts. The Cachelib-powered second tier benefits from the high-read fanout, hard-realtime nature of a sizable subset of Scribe workloads that consume a category's messages in a matter of (milli)seconds after message persistence. Section 5.4 will expand on the policies that Scribe uses to populate replicas across the two ephemeral storage tiers.

4.3 Read Path

Applications that read a category's data typically spawn a group of *Consumer* instances, collectively reading a category's contents in streaming fashion. Given the minimal resource footprint of a Consumer, it is typical for applications to spawn multiple instances on a single host. Each of the consumer instances will initiate a Thrift [50] connection to a Read Stream Service instance. The Scribe read path favours the establishment of such connections within the same geographical region in order to facilitate capacity provisioning / predictability as well as to minimize chances of flow control issues that could arise due to cross-region communication. After instantiating the connection with a Read Stream Service instance, the Consumer instance will start receiving streams of logical pointers corresponding to the payloads of messages, and will issue RPC calls to Read Proxy instances using the logical pointers in order to retrieve the actual payloads. Once retrieving a payload, the Consumer will proceed to decompress it and make it available to the application in a no-copy manner.

Forming a stream via the Read Stream Service. The *Read Stream Service* is the multi-tenant entity responsible for exposing the abstraction of a single stream per Consumer request, stitched together out of multiple LogDevice physical shards. Under the hood, every instance of the service maintains a pool of connections to LogDevice clusters, which it multiplexes across all incoming Consumer requests, thus amortising the memory tax of said connections. When a Read Stream Service instance receives a Consumer request, it will establish a connection to the Consumer and proceed to perform a discovery process to identify all LogDevice clusters and physical shards within them that are relevant for the request, as well as the offset within each shard that it should start reading from; Section 5.1 expands on the discovery process.

After identifying relevant physical shards, the Read Stream Service instance will spawn a LogDevice reader instance for each of them, start receiving a stream of records from each reader, and propagate them to the Consumer. This data flow relies on credit-based [28] techniques for flow control and application of backpressure when necessary: Each component indicates that it can process more messages by providing credits to its upstream. While being an elegant concept, in practice we have had to introduce significant complexity on top of our initial implementation in order to cater

for messages of different sizes, rely on different "channels" for messages and for stream metadata / telemetry, and to allow for control messages to flow downstream without consuming credits.

As messages flow downstream from LogDevice reader objects, the Read Stream Service needs to merge them together into the single metadata stream that the Consumer anticipates. The on-the-fly merging process defines the order in which a given Consumer will receive the category's messages, attempting to mask the impact of straggling, slow(er) shard readers; in the general case, the order can be different across Consumers making identical requests – with more details available in Section 6. Having established an order, the Read Stream Service forwards the records to the Consumer via the Thrift connection in streaming fashion.

At the time of receiving a Consumer request and establishing a connection, a Read Stream Service instance has little knowledge about the overall eventual resource consumption of the long-lived request: one Consumer request might involve reading a single physical shard, whereas another could involve hundreds / thousands of shards; the message rate per physical shard can also vary significantly. Given that each Read Stream Service instance serves hundreds of Consumer requests, it is possible to end up with hotspots if multiple "fat" streams land on a given Read Stream Service host. Therefore, the Read Stream Service performs load balancing by having every request undergo an eviction process a few minutes after its initialization, and has to re-establish a connection with a (less loaded) Read Stream service instance; Section 5.2 elaborates on host selection and traffic shaping.

Managing stream latency and liveness. Given the continuous nature of data streaming, the Read Stream Service attempts to mask the impact of any event that could disrupt data flow. When it comes to Consumer stream eviction events, given that the per-request shard discovery and initialization process can be time-consuming, the Read Stream Service accompanies each eviction message with information on the physical shards relevant for the request as well as the position in which reading them left off; the next connection event can thus bootstrap using this information and start the reading process instantaneously. In addition, despite allowing individual physical shard readers to progress independently at their own pace, the Read Stream Service still makes an effort to minimise chances of straggler effects and to detect non-transient issues as early as possible: For instance, each LogDevice physical shard punctuates [47] its record streams with heartbeat information [42], so that the Scribe read path can maintain accurate low-watermark [30] information as well as infer the liveness of each underlying physical shard. The Consumer also relies on heartbeats emitted from the Read Stream Service to infer whether it should retry its request against a different, healthier Read Stream Service instance. In terms of retry etiquette in general, all remote calls that the read path components issue follow best practices around the use of timeouts, jitter, and (exponential) backoff behaviour [10].

Serving and staging data payloads via the Read Proxy. The Read Proxy is a regional service, deployed in every data center of Meta, which encapsulates all accesses to Scribe payload storage. In contrast with the Read Stream Service and the long-lived, stream-like incoming requests that it receives, the Read Proxy receives short-lived, RPC-like requests from Consumers. Specifically, as a Consumer retrieves logical pointers to batches of data payloads from the Read Stream Service, it issues an RPC per logical pointer to a Read Proxy instance. The Read Proxy instance proceeds to retrieve the payload batch from Scribe's tiered storage and forwards it to the Consumer instance.

Shielding the network backbone. Scribe typically persists each record in the same data center region as its producer. Therefore, a given Scribe category typically receives input data records from producers deployed all over the globe rather than from a specific data center region. As a result, when a Consumer issues a read request, it receives the majority of relevant records – 80%-90% as a rule of thumb – via a cross-data-center-region data transport. Scribe operated in this fashion for almost its entire existence without issues. The rapid growth of recommender systems pipelines within Meta [51], however, drastically increased the egress of Scribe by more than 5× YoY. If not acted upon, such an increase trend could have surfaced risks in terms of satisfying cross-data-center network backbone [15] demand for critical data pipelines.

A major lever in accommodating the demand for ever-growing read fan-out came from the nature of the growth itself, which stemmed from data pipelines interested in reading the tail end of categories in hard real-time fashion. The growth also followed a power-law distribution – with a small set of popular categories being relevant for tens / hundreds of data pipelines performing feature engineering and model training. The Scribe read path thus evolved to minimize cross-region data fetching via the population of regional, ephemeral data replicas: When the Read Proxy receives a request, it attempts to retrieve the payload from a region-local replica – either an in-memory one or a Memcache-resident one – prior to resorting to a cross-region data fetch. Given the high read fanout of the workload, the aim is to perform the cross-region fetch once, and have the replica reused across all interested consumers; Section 5.4 elaborates on the replica population policies and on the scheduling extensions to maximize read fanout per replica.

Shielding the durable storage. Beyond the network backbone strain, the sheer amount of durable storage IO operations was an additional factor impacting the ability to accommodate read demand: Given that consumers for a category can spawn in multiple data center regions, Scribe would need to access its durable storage at least once per region with consumers. The Read Proxy thus supports *chaining* of requests: If a Read Proxy instance in region X receives a request for a payload stored in region Y, it will forward the request to the Read Proxy instance in region Y rather than contact storage directly, and indirectly populate an ephemeral replica in region Y if the payload is not already present in one. A subsequent request from a Read Proxy instance in region Z is thus likely to be served by the ephemeral replica of region Y. The end result is reduction in terms of i) IO operations for the scarcer durable storage, as well as in terms of ii) direct (cross-region) connections that can overwhelm durable storage, cause call latency unpredictability, and / or contribute to TCP incast effects [12].

Lightweight ETL. The multiple real-time stream processing applications consuming a popular category all conduct an identical data preparation process: Retrieve message batches from a Consumer instance, deserialize the batch into an in-memory columnar representation [38], and then proceed with the actual processing logic. The deserialization process involves a substantial compute "tax" [24, 36] for all the relevant applications. The tax is exacerbated

because it is typically non-trivial for the thousands of producers of a given category to independently accumulate enough data records at a time to convert them to an analytics-friendly, batched (columnar) representation prior to offloading them to Scribe: The majority of Scribe writers thus produce row-oriented, Thrift-encoded records rather than column-oriented batches, and the consuming applications have to perform the compute-intensive transcoding. In addition, while multiple stream processing applications can be consuming the same category, not all of them are interested in the entirety of the category's contents: such selective readers pay an even larger premium, given that they read and process data that they'd rather have pruned early on.

Scribe reduces the resource consumption of downstream applications by having the Read Proxy and the ephemeral storage act as a *data staging* area. Specifically, as part of populating an ephemeral replica, the Read Proxy transcodes the category's payloads to a columnar representation. At first glance, the transcoding computational cost is more pronounced for the Read Proxy compared to the case of a downstream application, because the Read Proxy has to i) decompress data batches, ii) deserialize them into Velox [38], iii) serialize them to a wire representation, and iv) recompress them prior to populating an ephemeral replica whereas a downstream consumer would only perform step (ii) on top of the decompression work they'd have to perform regardless. The Read Proxy computational cost, however, amortizes across the multiple readers of a high-fanout category.

Beyond transcoding, the Read Proxy is able to evaluate (a subset of) SQL queries, allowing for the pushdown [26] of the downstream application's filtering predicates to Scribe. In such cases, the Read Proxy populates materialized views [21, 23] rather than full data replicas. The Read Proxy refines the view definition based on a feedback loop that monitors the incoming, relatively static continuous query workloads; the current view definition policy optimizes for maximal reuse across all queries over a given category rather than exact matching with any specific query. The overall filtering process occurs in two phases, involving both the Read Proxy and the Consumer: Given that the Read Proxy compresses payloads for storage and network efficiency purposes prior to populating a view, it would incur a decompression - recompression tax per query: we instead opt for serving the entire view contents to a Consumer to avoid the (re)compression tax, even if they contain a superset of what the query requested; the Consumer then performs the final filtering step. We are also currently working on a workload-driven split of each view into multiple "mini-views" in order to increase the probability of answering queries without retrieving all of them.

Disaggregating the read path. A previous incarnation of the Scribe read path relied only on Read Stream Service, which would forward LogDevice records that contained the actual data payloads to the Consumer. The move to a decoupled storage model allowed for the corresponding disaggregation of the read path, with the Read Proxy becoming responsible for the manipulation of data payloads. The disaggregation brought multiple benefits:

- The (data) units of work shifted from long-lived, heterogeneous requests (i.e., streams) to short-lived, (more) homogeneous ones (i.e., single data payloads): The finer-grained work unit contributed to load balancing improvements.
- Disaggregation reduced the chance of a heavy request (in terms of compute or requested data volume) overwhelming a specific host, acting as a hotspot for the service, and being a noisy neighbour to other requests on the affected host; the computational cost of a stream now distributes across the entire Read Proxy tier. Coupled with the load balancing improvements, we were able to operate the Read Proxy tiers at 20-30% higher average utilization compared to the Read Stream Service ones.
- Having the Read Stream Service operate on metadata-only streams removed the heterogeneity factor stemming from varying message sizes, thus allowing for more predictable flow control.

5 THE CONTROL PLANE OF SCRIBE

Scribe relies on an extensive, multi-faceted control plane in order to accommodate the requirements for horizontal scalability and elasticity in the presence of multitenancy. The rest of this section highlights key aspects of Scribe's control plane.

5.1 Metadata management

Different parts of Scribe require access to metadata about categories to function correctly, such as who is the owner of a category, or whether a category exists at all. Category metadata thus needs to be accessible both on the write and the read path by the entire fleet of Scribe clients and backend services. While the canonical store for category metadata is a transactional, highly available database, having millions of hosts continuously querying the metadata service could overwhelm it. Scribe, thus, makes use of a caching and distribution layer on top of the canonical metadata store: A periodic job reads the entire database and publishes its contents to Meta's configuration distribution system [44], via which any interested entity can retrieve a category's metadata. Every instance of the Scribe fleet has to be resilient to issues related to i) the eventually consistent nature of metadata distribution that can lead to propagation delays for the latest metadata version, and ii) the small yet non-zero probability of corruption events. All relevant Scribe instances thus store in-memory and on-disk copies of the last "good" version of metadata they retrieved.

Besides coarse-grained category metadata, the Read Stream Service requires finer-grained knowledge on which LogDevice physical shards contain data for a given category. Populating this information involves a periodic job that polls LogDevice clusters to obtain the current tree-like mapping between cluster, categories and physical shards, and publishing said information in Meta's highly-replicated, global routing information database [41]. Each Read Stream Service instance subscribes to the relevant mapping for an incoming read request and dynamically initiates / terminates readers for each relevant physical shard added / removed from the mapping during the data stream's timeline.

5.2 Traffic management

Scribe has to accommodate virtually ad-hoc ingress and egress – with producers and consumers spawning in arbitrary fashion across the globe, and with arbitrary traffic spikes on a category / region / overall service level being the daily norm. The nature of the workload has thus motivated the introduction of traffic management mechanisms at multiple granularities.

Intra-cluster traffic shaping. The write path of Scribe reacts to a category's intra-region traffic increase by dynamically provisioning physical shards for it: A dedicated shard provisioning service monitors the traffic that existing shards of a LogDevice cluster receive. The service splits any shard that becomes too "hot" in terms of received throughput, in order to ensure horizontal scalability for each category. The same service is also responsible for reacting to traffic drops by deleting excess, empty logs once they are past their retention period. The periodic polling of LogDevice to identify empty logs can be resource-intensive for the LogDevice cluster; the provisioning service thus opts for pacing garbage collection of empty shards rather than eagerly deleting them at the very end of their retention period.

Inter-cluster traffic shaping. The write path of Scribe needs to be able to reason about how much traffic a given LogDevice cluster can receive and react accordingly once a cluster reaches its saturation point. To that end, the Scribe control plane monitors each cluster's write and read load, and continuously refines the threshold of incoming write volume that the cluster can accept. Based on this information, the regional tier of the Write Proxy can opt for redirecting traffic to a different LogDevice cluster. If all the clusters assigned to a regional tier of Write Proxy are saturated, or if the tier itself is overloaded, the write path will "spill" write traffic requests to nearby, less loaded regional tiers [41]. In case of a global overload event, Scribe opts for *graceful degradation* [34], shedding traffic of lower criticality to protect the overall service.

Host-level traffic shaping. Scribe relies on Meta's service mesh [41] to route traffic from its client libraries to its backend services. Specifically, each instance of a backend service advertises a load metric; the service mesh picks a random subset of hosts via rendezvous hashing [45] out of the overall candidate pool and relies on their load metric value to route the request to the least loaded one. For cases where the resource consumption of a (long-lived) request manifests seconds after its admission, such as in the case of the Read Stream Service, the host receiving the request advertises an artificially higher load metric value at the moment of a request's acceptance, and proceeds to exponentially decay the artificial value until the real resource consumption of the request becomes known.

Besides load-based routing, there are families of Scribe requests that benefit from affinitization to specific service hosts: The Write Proxy affinitizes traffic micro-shards to a specific Batch Service host to maximize the size of per-category message batches, and the Read Stream Service affinitizes payload requests to a specific given Read Proxy host when the request can potentially benefit from the latter's in-memory payload cache. Such cases affinitize requests via consistent hashing [25]. Still, given that consistent hashing can lead to hotspots in case of hot shards, Scribe's affinitized requests have a finite retry budget in case of reaching an overloaded or unavailable host. After the retry budget expires, the request salts its hash to deterministically route itself to 1 out of N fallback hosts – N being a configurable value.

Regardless of such fallback safeguards and of the reliance on the service mesh to achieve *proactive* traffic shaping, there is always a chance that a given host will receive increased load. For instance, a host can receive a sudden burst of requests; the load metric can lag behind; the overall tier of a service can be overwhelmed with incoming traffic. Thus, there is also a need for *reactive* traffic shaping to protect a service's goodput. Scribe service backends use a variety of techniques to react to a potential host-level overload event by transiently denying an incoming request and forcing its re-routing:

- Memory-bound Write Proxy instances rely on each incoming (Thrift) request annotating its header with the size of the payload to be processed. If the payload size would have lead to the host's memory budget to be exceeded, the Write Proxy instance rejects the request without paying for its full unmarshalling and processing.
- CPU-/NIC-bound Read Proxy instances rely on a PID controller to dynamically resize their request input queue based on a target host utilization.
- All backend services rely on token buckets to prevent thundering herd phenomena.

5.3 Quota management

Scribe relies on a quota management system to prevent global resource depletion, enforcing limitations on the acceptable growth of any given customer's write / read traffic on a global level. Different customer families can opt to distribute their available quota in different granularities. In addition, different enforcement mechanisms are popular for different use cases: On the write path, customers typically choose from i) fully blocking traffic to an offending category, ii) experiencing rate limiting in their write calls, or iii) sampling out a portion of their requests so that the overall traffic remains below the quota threshold. On the read path, outright blocking of readers is less favourable for Scribe, given that their eventual attempt to read the accumulated data backlog post-unblocking will consume more storage resources compared to real-time data reading. As a result, customers typically opt for sampling out a portion of their traffic as the quota enforcement mechanism of choice.

5.4 Ephemeral replicas management

Scribe populates inexpensive, ephemeral data replicas to minimize accesses to more expensive resources such as durable block storage and cross-region network traffic and to maintain data copies in a representation that is optimal for downstream analytical workloads. Deciding the criteria for maximum pay-off from replica population has involved dissecting the problem statement into two complementary ones: i) How can Scribe maximize pay-off given the current workload, and ii) how the overall workload shape can incrementally adapt to the available Scribe resources.

On the Scribe end, we model the problem statement as a constraint satisfaction problem: The control plane aims to come up with decisions on replica and materialized view creation at a data center region granularity, in the presence of constraints around the availability of resources such as durable storage IO, Memcache ingress / egress, and (cross-data-center) network backbone ingress / egress. Parameters that affect decisions include, among others, i) the read fanout of a category, ii) whether a category's consumers are reading the tail end of the category's messages rather than older ones, iii) the selectivity factor (in terms of bytes) of a candidate materialized view, and iv) whether the Read Proxy would have to (re-)pay a deserialization tax to transcode a category's contents to the output wire format of choice.

The Scribe control plane outputs population decisions per data center region. For instance, a region with limited Memcache egress availability could adopt a policy of hierarchical caching, with the in-memory replicas shielding accesses to Memcache. Another region with limited Memcache ingress availability could adopt a policy of treating Memcache and the Read Proxy's memory as disjoint caches – with the latter alleviating some of the ingress demand that would be destined for Memcache. Given that workloads can fluctuate, the control plane periodically recomputes the population policies to account for fluctuations; still, regardless of the quality and responsiveness of Scribe policies, a category's multiple readers could be spread across a large number of data center regions, thus making replica population less effective or even moot.

Scribe minimises the chance of such pathological cases by integrating with the schedulers [13, 29] of major downstream platforms: Specifically, Scribe advertises its available resources per data center to the schedulers; the schedulers have incorporated *reader co-location* as one of their optimization criteria in order to minimize the spread of readers for the same category across data center regions. In practice, the greedy approach of prioritizing the co-location of readers for the most voluminous categories has so far offered the best outcome in terms of network backbone utilization.

5.5 Service health management

Observability into the health of the Scribe service was initially heavily focused on write availability and on specific business-critical use cases. Beyond those limited use cases, write path monitoring was in place on a global, coarse-grained level. Introduction of read path observability also took place on a global level, with additional complications stemming from user behaviour: A reader application could be willingly reading historical rather than fresh messages, be underprovisioned for incoming traffic, or even be incurring a downstream dependency failure that applies backpressure to the overall data stream. The diversity of the thousands of applications writing into and reading from Scribe brought further complications, given the range of guarantees they expected from the service.

As the Scribe service and the overall data ecosystem in Meta evolved, Scribe introduced per-tenant SLO monitoring, focused on availability and data quality. A major part of the effort went into error attribution: We developed a monitoring system capturing a diverse set of signals and embedded it into Scribe clients in order to discern failures stemming from downstream issues or even infrastructure issues (e.g., faulty host, client networking overloads). In anticipation of the calibrating / de-noising of the monitoring taking multiple iterations, we ensured that the error attribution definitions were dynamically configurable. We also extended all components comprising the read path pipeline to be able to infer whether their downstream is not actively consuming messages, and thus be able to disambiguate Scribe-caused latency from customer-caused one.

6 GUARANTEES: ONE SIZE DOES NOT FIT ALL

Scribe serves messages to multiple downstream systems; inevitably, these systems exhibit different properties. For instance, the ingestion service that hauls TAO [49] social graph data into Meta's data warehouse performs in-order processing in hourly granularity; the general-purpose stream processing service [11] performs out-of-order-processing [7, 30]; select Ads revenue pipelines are extremely sensitive to both data duplication and loss, favouring exactly once processing. In short, the numerous Scribe customers have different requirements out of the service.

Looking at the storage solution that powers Scribe streams, LogDevice [32] offers strict properties such as repeatable reads and exactly once data delivery at a physical shard granularity. At the same time, numerous challenges arose for Scribe to expose stringent properties across multiple LogDevice clusters (and even more LogDevice shards): For a start, the majority of Scribe customers (in terms of volume) were uninterested in strict delivery properties if they would come at a Scribe resource consumption premium. Any proposed solution would thus need to become available as an optional feature, without bringing in a horizontal resource consumption regression, and without overcomplicating the existing infrastructure that had been optimised for supporting high-throughput traffic at scale.

The rest of this section discusses the variety of guarantees that Scribe exposes, goes through examples of how Scribe empowers different use cases to pick the right trade-off, and discusses ongoing work towards supporting strict guarantees, without an increase in resource consumption, for high-throughput traffic at scale.

6.1 Best effort guarantees

"Vanilla" Write Path: Best effort delivery. The majority of applications interacting with Scribe produce data in "firehose mode" – with some prominent categories receiving more than 500GB/s of traffic. When reasoning about the aggregate traffic of such applications, Scribe's primary care is successfully persisting the substantial data volumes at minimal resource consumption. Being write-availability-first means that Scribe opts for opportunistic data routing between client and Write Proxy hosts, and that traffic can occasionally "spill" to another data center region. Coupled with the use of retries in case of perceived write call failures / timeouts, it is unlikely yet plausible that multiple calls for the same message will successfully reach Scribe storage, leading to message duplication. At the same time, enabling customers to produce high-volume traffic per Producer instance implies that customers should be able to write their messages to Scribe in as non-blocking fashion as possible. Customers relying on such "fire and forget" write calls thus have to tolerate a potential degree of data loss in case one of their many write calls ends up failing.

"Vanilla" Read Path: Approximate ordering. When a consumer issues a read request for a given category, it would typically retrieve messages from numerous physical shards because of the opportunistic routing nature of Scribe's write path and because a given category's producers generate messages in geo-distributed fashion. Attempting to retrieve data from multiple LogDevice shards is inevitably subject to straggler effects [14]: At best, not all shard readers will be able to sustain the same pace. At worst, a given shard may be experiencing availability issues. Thus, attempting to enforce strict temporal order across the numerous log-based readers that form a category's output data stream imposes an inherent trade-off between throughput, latency, and data availability.

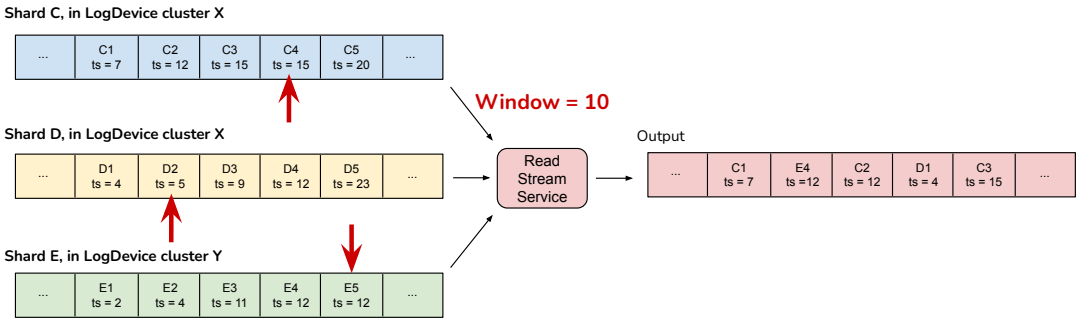

Figure 3: Keeping read traffic from multiple physical shards in approximate order: Assuming an ordering window of 10 time units, the (faster) reader of shard C will pause until the reader of shard D advances forward from message D2.

Scribe bridges the gap between throughput and latency by allowing its customers to embrace disorder [31]: A category's shard readers can advance individually as long as they don't drift *too much* from each other. Specifically, consumers have access to a configurable *ordering window* concept, an example of which is illustrated in Figure 3: The ordering window ensures that output messages are within a specified time range from each other, typically in the order of minutes, based on the time said messages reached persistent storage. Exposing the ordering trade-off allows different downstream applications to opt for their experience of choice: For instance, a general-purpose stream processing application can opt for a 30-minute ordering window and introduce a sorting buffer to reinstate order; another application that performs windowed joins can be more memory-conscious and opt for a 5-minute ordering window; a database used for realtime, timeseries-based data monitoring [5] that favours early data delivery and can tolerate out-of-order data can opt for an ordering window in the order of days.

6.2 At least once delivery

Moving beyond best effort guarantees, the most obvious scenario that comes to mind is the need for at least once data delivery, which is critical for use cases such as the ones ingesting critical datasets into Meta's data warehouse. Scribe offers at least once delivery by relying on storage acknowledgments that indicate whether the storage layer considers a message as successfully persisted, and relying on aggressive retries in each write path hop to ensure eventual success. On the other hand, aggressive retries increase the chance of data duplication if performed in a brute force fashion.

The Scribe write path is thus carefully tuned to minimize the chances of duplication: For instance, the write path maintains traffic statistics at different historical granularities to predict how many physical shards to initialize ahead of time in order to mask the potentially blocking nature of shard initialization. In addition, every downstream write path hop has more slack in terms of when to consider a request as timed-out compared to its upstream hop, in order to keep request duplication at a minimum. Finally, the write path applies more conservative batching thresholds for traffic subject to at least once delivery, effectively generating smaller batches, so that potential duplicate request spikes have a smaller chance to overwhelm the resources of its backend services. The additional measures to ensure at least once delivery lead to increased resource consumption for the relevant data traffic.

6.3 Repeatable reads & exactly once processing

Scribe can further enhance its at least once delivery offering by activating support for repeatable reads. Scribe currently offers two variants of repeatable reads, each surfacing different trade-offs.

Write path variant: Use cases interested in transporting (low-volume) change-data-capture-streams depend on strict data ordering and read repeatability. Scribe achieves this level of guarantees by *affinitizing* the traffic of a given logical shard's writer to a *single, dedicated (LogDevice) physical shard*. The side effect of serializing a logical shard's traffic is that its maximum supported throughput is now capped by what a single physical shard can support. In addition, the offering also "inherits" the less-aggressive batching involved with the at least once delivery offering, and is thus subject to scalability limits and increased resource consumption.

Read path variant: Stream processing pipelines with exactly-once processing requirements integrate vertically with Scribe to apply an established methodology [20]:

(1) *Repeatable reads* out of a given Scribe category allow the downstream to associate each message batch read with a *unique, monotonically increasing identifier*
(2) *Deterministic processing* without side-effects in the stream processing application ensures resilience to failures
(3) Scribe deduplicates the write operations of the stream processing application to a sink category via the use of a [data producer, message batch identifier] de-duplication map at the LogDevice layer, guaranteeing *idempotent writes*

On top of these properties, the relevant stream processing applications require the ability to fine-tune the size of the message batch associated with the aforementioned unique identifier – especially when performing memory-intensive operations such as streaming joins. The minimal batching of the Scribe write path was a poor fit for such a requirement. Scribe thus also introduced read repeatability via a *read path variant*, depicted in Figure 4, that decouples the degree of batching from the write path properties: The variant acts as a "bolt-on" over the vanilla read path, and uses a dedicated reader to consume all physical (metadata) shards of a given category, form message batches of a size dictated by the downstream application, and *sequence* them into a single metadata shard – establishing a

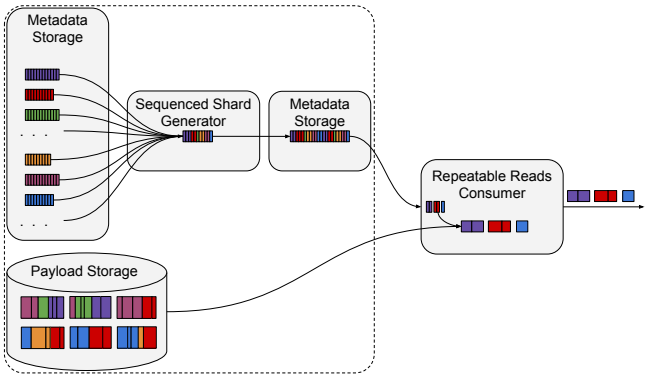

Figure 4: Generating a repeatable read order: An internal flow sequences a category's metadata entries into a single physical shard. The end consumers read the single, "sequenced" shard.

singular sequence for the category's messages. End application readers can then consume the single, sequenced shard and obtain a deterministic (rather than an approximate) data order for the category. The trade-off involved in the described Scribe enhancement is related to latency: An additional hop and (metadata) persistence step is required to generate the sequenced representation that all downstream consumers will read.

Towards exactly once processing at scale: Despite having two repeatable reads variants, each faces different scalability issues that make their adoption challenging for the most voluminous Scribe categories with a throughput greater than 100GB/s. The issues stem from legacy architectural decisions around the provisioning of physical shards: Specifically, thresholds for the creation of additional physical shards stem from the Scribe era when a message's metadata and payload were stored *together*, rather than *decoupled*. In addition, a physical shard had always been the unit of reading data out of Scribe. Therefore, the traffic directed to a single shard could not exceed the consumption rate of the "wimpiest" consumer out of the many consumers reading the shard.

The design choices above contributed to excessive shard fragmentation – an obstacle for the scalability of stronger guarantees solutions: For instance, the sequenced shard generator of Figure 4 could be responsible for the resource-intensive reading of tens of thousands of storage shards and be subject to multiple straggler effects. In addition, the proliferation of shards and the management of their extensive state hindered the adoption of alternative designs that would offload each message (batch) and its corresponding stream state to transactional storage [6].

We are currently working on making physical shard provisioning more judicious, based on per-message-count thresholds rather than on per-payload-size ones: Our aim is to reduce the amount of physical shards by at least an order of magnitude in order to make their sequencing more straightforward. We are also working to allow consumers to read *subsets* of a physical shard, decoupling a physical shard's write throughput threshold from the potentially limited processing capabilities of a consumer, and thus unblocking the creation of fewer and more voluminous physical shards.

7 LESSONS LEARNED

Our experience with Scribe over the past 18 years has helped us distill a number of observations around managing and evolving infrastructure at scale: The rest of this section presents some notable cases in which we revisited assumptions and evolved our designs.

On client binaries vs. (fat) libraries: A non-static decision driven by the ecosystem. While Scribe's client interfacing has largely converged towards the use of Producer / Consumer libraries, this had not always been the case. Using the read path as a case study, the de facto entry point used to be a binary that was encapsulating the entire read path functionality. The binary exposed messages via a pipe API: a user could pipe the "raw" byte stream into their business logic and rapidly spin a novel application over Scribe. We also accommodated customers stalling in terms of updating their applications: We introduced a bespoke "autoupdating" wrapper over the binary to avoid the presence of stale versions that would stand in the way of fixing bugs and introducing new features. Essentially, the binary had a very low entry bar and "assumed / allowed the worst" in terms of customer behaviour and capabilities.

While allowing our customers to "move fast", the low entry bar raised a set of challenges: Untyped access to a binary stream was error-prone and non-trivial to extend without careful orchestration with our long tail of customers. In addition, the bespoke autoupdating logic was brittle and occasionally conflicted with successful reader restart / shutdown. As Meta's data ecosystem matured, we resolved the pain points by making "close world" assumptions rather than accommodate every outlier behaviour: We partnered with our most voluminous downstream data platforms to migrate them to the lean, strictly typed Consumer library that was less prone to interface evolution pain points while providing first-class access to metadata related to privacy enforcement and schema management; we then encouraged our long tail of customers to migrate to said platforms. In addition, numerous Meta-wide safeguards and processes discouraged and prevented the presence of stale binaries, thus eschewing the need for auto-updating logic.

While we consider lean libraries as the ideal entry point to Scribe, we have frequently run into cases where our libraries accumulate logic relevant for optimisation purposes, and thus the challenges around distributing error-prone, resource-intensive, complicated logic to millions of instances resurface. For such cases, such as the one of our Producer library, we have adopted the model of having the ScribeD on-host daemon process engulf as much of the complicated logic as possible, while multiplexing its resource budget across all Producer instances on the host. We then minimized the blast radius of daemon bugs by adopting a very cautious, slow rollout process throughout the Meta fleet.

On user freedom: Encouraging / enforcing best practices. Historically, a Scribe consumer could spawn in an arbitrary data center region, at an arbitrary time point, and request to consume an arbitrary time slice of a category's N-day retention. While the isolated case of a single consumer would go unnoticeable, the same would not hold if repeated by large-scale data consumers (e.g., the data flows powering model training for video recommendations): Regardless of Scribe's multiple layers of overload protection, we consider large-scale traffic shift to an undersized backend tier as

an extraordinary event rather than a business-as-usual one. In similar spirit, Scribe's architecture is optimised under the assumption that real-time data accesses form the majority; a drastic change towards large-scale historical reading operations in steady state would likewise degrade the overall quality of service.

As the scale of realtime data flows kept increasing, we moved from encouraging best practices towards enforcing them:

- The expectation is for customers to now adhere to predefined, regionalized capacity contracts [18], offering predictability in terms of hardware provisioning across large-scale services.
- Borrowing ideas from "pure" pub-sub systems, we are moving towards consumers registering their intent with Scribe, allowing us to perform optimizations such as materialized view selection with higher confidence.
- We are looking into distinguishing between realtime vs. (lower) non-realtime SLOs / pricing models as a further incentive.

On (inevitable) vertical integration: Abstraction leakage avoidance. We have frequently revisited Scribe interfaces and capabilities to better fit those of upstream and downstream partners in an attempt to overcome scalability walls. To name a few examples:

- Co-partitioning of Scribe categories and the data warehouse tables that they feed into allows for warehouse ingestion of partitioned tables without involving a large-scale shuffle operation.
- APIs for the atomic transport of customer-provided message batches enable customers to offload to Scribe ready "sorted runs" for their downstream sorting workloads.
- Adopting columnar encoding schemes [4] that are specific to ML sparse features leads to succinct, storage-efficient representations for voluminous categories that contain training data.

In hindsight, making (too) strong connections between Scribe interfaces and at-the-time implementation details has been an anti-pattern that hindered system extensibility and innovation: For instance, implicitly connecting Scribe's physical sharding scheme with the ability of an individual consumer to keep up with a single shard's traffic tied our hands in terms of making shards more voluminous and combating fragmentation.

On the other hand, the way Scribe exposes data grouping and partitioning capabilities have been positive examples: Customers interested in writing pre-grouped data batches rely on a dedicated `write()` call that ensures a set of invariants hold; customers interested in partitioning their category based on a subset of its fields explicitly indicate so at category authoring time, and are subject to restrictions in terms of partition cardinality.

On architectural evolution: Incremental simplification and modernization. Up until recently, and ever since its creation, Scribe's focus was singular: Transporting binary, opaque payloads in streaming fashion from point A to point B. The need for resource-efficient realtime analytics across an ever growing scale and number of data centers forced a mindset transition: rather than offering "just" a *data transport* offering, we had to expose a *data staging* one.

Such architectural evolutions are multi-year endeavours that can be challenging to motivate and kickstart. We tackled this specific instance by identifying potential blockers as well as potential boons in the overall data ecosystem of Meta, and working incrementally towards the end goal, bringing added value in each step:

- We introduced support for data transcoding as a facilitator for the migration of Meta flows from CSV and JSON to a Thrift-based representation [48].
- We standardized (de)serialization logic across all major downstream platforms as part of a consolidation initiative to minimize discrepancies across query engines, while also benefiting from the widespread adoption of Velox [38]: customers issuing a transcoding / filtering request to Scribe would now be getting results consistent with performing the evaluation on their end.
- We introduced support for per-message metadata to facilitate the propagation of privacy- and lineage-related information [16]; the same support would later allow downstream platforms to perform data clustering without deserializing messages.
- We introduced a preliminary version of decoupling data payloads from ordering information as part of an earlier endeavour to separate our storage backend into two tiers, optimized for realtime and backlog accesses, respectively; said decoupling would end up powering both the stronger guarantees offerings as well as the first-class support for ephemeral replicas and filtering capabilities in the read path.
- The decoupling of data payloads from ordering information also meant that the scalability requirements for LogDevice, which used to store both, lowered by orders of magnitude. The relaxed requirements enabled the Scribe team to create a blueprint on how to deliver stronger delivery guarantees without the burden of scalability concerns, as well as "shed" a lot of the bespoke logic within LogDevice, and explore building Scribe's metadata layer on top of an existing, general-purpose key-value store [33].

With the above efforts in place, bridging the delta from the desired end state appeared less daunting for the team.

8 CONCLUSION

This paper presented Scribe, Meta's message queue service, and outlined the key properties that have empowered Scribe to ingest $> 15TB/s$ from its producers and serve $> 110TB/s$ to its consumers. Since its inception, we designed Scribe from the ground up to favour write availability over other properties. As Meta workloads became more read-intensive, we evolved the Scribe architecture: While write availability is still a first class citizen, the Scribe read path now adapts to workload properties in order to scale data delivery across an ever growing number of data centers. Our priorities also evolved from "just" reducing the resource consumption of Scribe to also reducing the one of its downstream consumers – with lightweight ETL capabilities meshed into Scribe data delivery. Likewise, the service's focus extended beyond serving high-throughput traffic with approximate guarantees, towards a wide range of flavours that enable customers to opt for trade-offs around delivery guarantees, freshness, scalability, and cost. Finally, the journey towards the current Scribe architecture has not been linear; we thus outlined some of the learnings we obtained along the way.

ACKNOWLEDGMENTS

Scribe has been an ever-evolving service for the past 18 years, with multiple architectural incarnations. Numerous past and present Scribe team members have contributed to the journey and the big picture described in this work; we are grateful to them all.

REFERENCES

[1] [n.d.]. *RabbitMQ.* Retrieved February 28, 2025 from www.rabbitmq.com

[2] 2022. *Redpanda Cloud brings the fastest Kafka API to the cloud.* Retrieved February 13, 2025 from https://www.redpanda.com/blog/introducing-redpanda-cloud-for-kafka

[3] 2025. *Redpanda.* Retrieved February 13, 2025 from https://www.redpanda.com/

[4] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis (Eds.). ACM, 671–682. https://doi.org/10.1145/1142473.1142548

[5] Lior Abraham, John Allen, Oleksandr Barykin, Vinayak R. Borkar, Bhuwan Chopra, Ciprian Gerea, Daniel Merl, Josh Metzler, David Reiss, Subbu Subramanian, Janet L. Wiener, and Okay Zed. 2013. Scuba: Diving into Data at Facebook. *Proc. VLDB Endow.* 6, 11 (2013), 1057–1067. https://doi.org/10.14778/2536222.2536231

[6] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. *Proc. VLDB Endow.* 6, 11 (2013), 1033–1044. https://doi.org/10.14778/2536222.2536229

[7] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *Proc. VLDB Endow.* 8, 12 (2015), 1792–1803. https://doi.org/10.14778/2824032.2824076

[8] Masha Basmanova. 2023. *SerializedPage Wire Format.* The Presto Foundation. Retrieved February 6, 2025 from https://prestodb.io/docs/current/develop/serialized-page.html

[9] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. 2020. The CacheLib Caching Engine: Design and Experiences at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 753–768. https://www.usenix.org/conference/osdi20/presentation/berg

[10] Marc Brooker. [n.d.]. *Timeouts, retries, and backoff with jitter.* Amazon. Retrieved February 22, 2025 from https://aws.amazon.com/builders-library/timeouts-retries-and-backoff-with-jitter/

[11] Guoqiang Jerry Chen, Janet L. Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. 2016. Realtime Data Processing at Facebook. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 1087–1098. https://doi.org/10.1145/2882903.2904441

[12] Yanpei Chen, Rean Griffith, David Zats, Anthony D. Joseph, and Randy H. Katz. 2012. Understanding TCP Incast and Its Implications for Big Data Workloads. *login Usenix Mag.* 37, 3 (2012). https://www.usenix.org/publications/login/june-2012/understanding-tcp-incast

[13] Arnab Choudhury, Yang Wang, Tuomas Pelkonen, Kutta Srinivasan, Abha Jain, Shenghao Lin, Delia David, Siavash Soleimanifard, Michael Chen, Abhishek Yadav, Ritesh Tijoriwala, Denis Samoylov, and Chunqiang Tang. 2024. MAST: Global Scheduling of ML Training across Geo-Distributed Datacenters at Hyperscale. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 563–580. https://www.usenix.org/conference/osdi24/presentation/choudhury

[14] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, Eric A. Brewer and Peter Chen (Eds.). USENIX Association, 137–150. http://www.usenix.org/events/osdi04/tech/dean.html

[15] Marek Denis, Yuanjun Yao, Ashley Hatch, Qin Zhang, Chiunlin Lim, Shuqiang Zhang, Kyle Sugrue, Henry Kwok, Mikel Jimenez Fernandez, Petr Lapukhov, Sandeep Hebbani, Gaya Nagarajan, Omar Baldonado, Lixin Gao, and Ying Zhang. 2023. EBB: Reliable and Evolvable Express Backbone Network in Meta. In *Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM 2023, New York, NY, USA, 10-14 September 2023*, Henning Schulzrinne, Vishal Misra, Eddie Kohler, and David A. Maltz (Eds.). ACM, 346–359. https://doi.org/10.1145/3603269.3604860

[16] Wenlong Dong, Rituraj Kirti, Diana Marsala, Kendall Hopkins, Lucas Waye, Hannes Roth, Jonathan Bergeron, Alex Ponomarenko, and Avtar Brar. 2024. *How Meta enforces purpose limitation via Privacy Aware Infrastructure at scale.* Meta Platforms Inc. Retrieved February 6, 2025 from https://engineering.fb.com/2024/08/27/security/privacy-aware-infrastructure-purpose-limitation-meta/

[17] Pavan Edara, Jonathan Forbesj, and Bigang Li. 2024. Vortex: A Stream-oriented Storage Engine For Big Data Analytics. In *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago AA, Chile, June 9-15, 2024*, Pablo Barceló, Nayat Sánchez-Pi, Alexandra Meliou, and S. Sudarshan (Eds.). ACM, 175–187. https://doi.org/10.1145/3626246.3653396

[18] Marius Eriksen, Kaushik Veeraraghavan, Yusuf Abdulghani, Andrew Birchall, Po-Yen Chou, Richard Cornew, Adela Kabiljo, Ranjith Kumar S, Maroo Lieuw, Justin Meza, Scott Michelson, Thomas Rohloff, Hayley Russell, Jeff Qin, and Chunqiang Tang. 2023. Global Capacity Management With Flux. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 589–606. https://www.usenix.org/conference/osdi23/presentation/eriksen

[19] Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux Journal* 2004, 124 (Aug. 2004), 5.

[20] Marios Fragkoulis, Paris Carbone, Vasiliki Kalavri, and Asterios Katsifodimos. 2024. A survey on the evolution of stream processing systems. *VLDB J.* 33, 2 (2024), 507–541. https://doi.org/10.1007/S00778-023-00819-8

[21] Alon Y. Halevy. 2001. Answering queries using views: A survey. *VLDB J.* 10, 4 (2001), 270–294. https://doi.org/10.1007/S007780100054

[22] Stavros Harizopoulos, Taylor Hopper, Morton Mo, Shyam Sundar Chandrasekaran, Tongguang Chen, Yan Cui, Nandini Ganesh, Gary Helmling, Hieu Pham, and Sebastian Wong. 2022. Meta's Next-generation Realtime Monitoring and Analytics Platform. *Proc. VLDB Endow.* 15, 12 (2022), 3522–3534. https://doi.org/10.14778/3554821.3554841

[23] Milena Ivanova, Martin L. Kersten, Niels J. Nes, and Romulo Goncalves. 2010. An architecture for recycling intermediates in a column-store. *ACM Trans. Database Syst.* 35, 4 (2010), 24:1–24:43. https://doi.org/10.1145/1862919.1862921

[24] Svilen Kanev, Juan Pablo Darago, Kim M. Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David M. Brooks. 2016. Profiling a Warehouse-Scale Computer. *IEEE Micro* 36, 3 (2016), 54–59. https://doi.org/10.1109/MM.2016.38

[25] David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin. 1997. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, El Paso, Texas, USA, May 4-6, 1997*, Frank Thomson Leighton and Peter W. Shor (Eds.). ACM, 654–663. https://doi.org/10.1145/258533.258660

[26] Donald Kossmann. 2000. The State of the art in distributed query processing. *ACM Comput. Surv.* 32, 4 (2000), 422–469. https://doi.org/10.1145/371578.371598

[27] Jay Kreps, Neha Narkhede, and Jun Rao. 2011. Kafka: A distributed messaging system for log processing. *Proceedings of NetDB* 11 (2011).

[28] H. T. Kung and Robert Morris. 1995. Credit-based flow control for ATM networks. *IEEE Netw.* 9, 2 (1995), 40–48. https://doi.org/10.1109/65.372658

[29] Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, Kaushik Veeraraghavan, Biren Damani, Pol Mauri Ruiz, Vikas Mehta, and Chunqiang Tang. 2021. Shard Manager: A Generic Shard Management Framework for Geo-distributed Applications. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 553–569. https://doi.org/10.1145/3477132.3483546

[30] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. 2008. Out-of-order processing: a new architecture for high-performance stream systems. *Proc. VLDB Endow.* 1, 1 (2008), 274–288. https://doi.org/10.14778/1453856.1453890

[31] David Maier, Jin Li, Peter A. Tucker, Kristin Tufte, and Vassilis Papadimos. 2005. Semantics of Data Streams and Operators. In *Database Theory - ICDT 2005, 10th International Conference, Edinburgh, UK, January 5-7, 2005, Proceedings (Lecture Notes in Computer Science)*, Thomas Eiter and Leonid Libkin (Eds.), Vol. 3363. Springer, 37–52. https://doi.org/10.1007/978-3-540-30570-5_3

[32] Mark Marchhukov. 2017. *LogDevice: a distributed data store for logs.* Meta Platforms Inc. Retrieved February 13, 2025 from https://engineering.fb.com/2017/08/31/core-infra/logdevice-a-distributed-data-store-for-logs/

[33] Sarang Masti. 2021. *How we built a general purpose key value store for Facebook with ZippyDB.* Meta Platforms Inc. Retrieved July 2, 2025 from https://engineering.fb.com/2021/08/06/core-infra/zippydb/

[34] Justin Meza, Thote Gowda, Ahmed Eid, Tomiwa Ijaware, Dmitry Chernyshev, Yi Yu, Md Nazim Uddin, Rohan Das, Chad Nachiappan, Sari Tran, Shuyang Shi, Tina Luo, David Ke Hong, Sankaralingam Panneerselvam, Hans Ragas, Svetlin Manavski, Weidong Wang, and Francois Richard. 2023. Defcon: Preventing Overload with Graceful Feature Degradation. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, Roxana Geambasu and Ed Nightingale (Eds.). USENIX Association, 607–622. https://www.usenix.org/conference/osdi23/presentation/meza

[35] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, Nick Feamster and Jeffrey C. Mogul (Eds.). USENIX Association, 385–398. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala

[36] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making Sense of Performance in Data Analytics Frameworks. In *12th*

USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015. USENIX Association, 293–307. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/ousterhout

[37] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar P, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, Christian Preseau, Pratap Singh, Kestutis Patiejunas, JR Tipton, Ethan Katz-Bassett, and Wyatt Lloyd. 2021. Facebook's Tectonic Filesystem: Efficiency from Exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 217–231. https://www.usenix.org/conference/fast21/presentation/pan

[38] Pedro Pedreira, Orri Erling, Maria Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: Meta's Unified Execution Engine. *Proc. VLDB Endow.* 15, 12 (2022), 3372–3384. https://doi.org/10.14778/3554821.3554829

[39] Anna Povzner, Prince Mahajan, Jason Gustafson, Jun Rao, Ismael Juma, Feng Min, Shriram Sridharan, Nikhil Bhatia, Gopi K. Attaluri, Adithya Chandra, Stanislav Kozlovski, Rajini Sivaram, Lucas Bradstreet, Bob Barrett, Dhruvil Shah, David Jacot, David Arthur, Manveer Chawla, Ron Dagostino, Colin Mccabe, Manikumar Reddy Obili, Kowshik Prakasam, Jose Garcia Sancio, Vikas Singh, Alok Nikhil, and Kamal Gupta. 2023. Kora: A Cloud-Native Event Streaming Platform for Kafka. *Proc. VLDB Endow.* 16, 12 (2023), 3822–3834. https://doi.org/10.14778/3611540.3611567

[40] I. S. Reed and G. Solomon. 1960. Polynomial Codes Over Certain Finite Fields. In *Journal of The Society for Industrial and Applied Mathematics*. 300–304.

[41] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. 2023. ServiceRouter: Hyperscale and Minimal Cost Service Mesh at Meta. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 969–985. https://www.usenix.org/conference/osdi23/presentation/saokar

[42] Utkarsh Srivastava and Jennifer Widom. 2004. Flexible Time Management in Data Stream Systems. In *Proceedings of the Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 14-16, 2004, Paris, France*, Catriel Beeri and Alin Deutsch (Eds.). ACM, 263–274. https://doi.org/10.1145/1055558.1055596

[43] Yutian Sun, Tim Meehan, Rebecca Schlussel, Wenlei Xie, Masha Basmanova, Orri Erling, Andrii Rosa, Shixuan Fan, Rongrong Zhong, Arun Thirupathi, Nikhil Collooru, Ke Wang, Sameer Agarwal, Arjun Gupta, Dionysios Logothetis, Kostas Xirogiannopoulos, Amit Dutta, Varun Gajjala, Rohit Jain, Ajay Palakuzhy, Prithvi Pandian, Sergey Pershin, Abhisek Saikia, Pranjal Shankhdhar, Neerad Somanchi, Swapnil Tailor, Jialiang Tan, Sreeni Viswanadha, Zac Wen, Biswapesh Chattopadhyay, Bin Fan, Deepak Majeti, and Aditi Pandit. 2023. Presto: A Decade of SQL Analytics at Meta. *Proc. ACM Manag. Data* 1, 2 (2023), 189:1–189:25. https://doi.org/10.1145/3589769

[44] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. 2015. Holistic configuration management at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, Ethan L. Miller and Steven Hand (Eds.). ACM, 328–343. https://doi.org/10.1145/2815400.2815401

[45] David Thaler and Chinya V Ravishankar. 1996. A name-based mapping scheme for rendezvous. *Electrical Engineering and Computer Science Department, The University of Michigan, Ann Arbor, Michigan* (1996), 1–31.

[46] Raúl Gracia Tinedo, Flavio Junqueira, Tom Kaitchuck, and Sachin Joshi. 2023. Pravega: A Tiered Storage System for Data Streams. In *Proceedings of the 24th International Middleware Conference, Middleware 2023, Bologna, Italy, December 11-15, 2023*. ACM, 165–177. https://doi.org/10.1145/3590140.3629113

[47] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. 2003. Exploiting Punctuation Semantics in Continuous Data Streams. *IEEE Trans. Knowl. Data Eng.* 15, 3 (2003), 555–568. https://doi.org/10.1109/TKDE.2003.1198390

[48] Bharat Vaidhyanathan, Dhruv Matani, Md Mustafijur Rahman Faysal, and Saurav Sen. 2023. *Tulip: Schematizing Meta's data platform*. Meta Platforms Inc. Retrieved February 6, 2025 from https://engineering.fb.com/2023/01/26/data-infrastructure/tulip-modernizing-metas-data-platform/

[49] Venkateshwaran Venkataramani, Zach Amsden, Nathan Bronson, George Cabrera III, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Jeremy Hoon, Sachin Kulkarni, Nathan Lawrence, Mark Marchukov, Dmitri Petrov, and Lovro Puzar. 2012. TAO: how facebook serves the social graph. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman (Eds.). ACM, 791–792. https://doi.org/10.1145/2213836.2213957

[50] Dave Watson. 2014. *Under the Hood: Building and open-sourcing fbthrift*. Meta Platforms Inc. Retrieved February 21, 2025 from https://engineering.fb.com/2014/02/20/open-source/under-the-hood-building-and-open-sourcing-fbthrift/

[51] Mark Zhao, Niket Agarwal, Aarti Basant, Bugra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. 2022. Understanding data storage and ingestion for large-scale deep recommendation model training: industrial product. In *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, Valentina Salapura, Mohamed Zahran, Fred Chong, and Lingjia Tang (Eds.). ACM, 1042–1057. https://doi.org/10.1145/3470496.3533044