Towards Observability Data Management at Scale

Suman Karumuri
Slack Technologies
skarumuri@slack-corp.com

Franco Solleza, Stan Zdonik
Brown University
{fsolleza,sbz}@cs.brown.edu

Nesime Tatbul
Intel Labs and MIT
tatbul@csail.mit.edu

ABSTRACT

Observability has been gaining importance as a key capability in today's large-scale software systems and services. Motivated by current experience in industry exemplified by Slack and as a call to arms for database research, this paper outlines the challenges and opportunities involved in designing and building Observability Data Management Systems (ODMSs) to handle this emerging workload at scale.

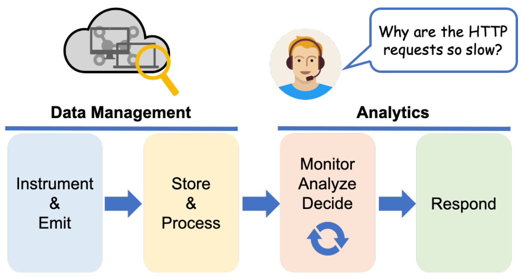


Figure 1: Observability

1 INTRODUCTION

On May 12, 2020 at 4:45pm PST, the cloud-based business communication platform Slack experienced a total service disruption [9]. To its millions of users, the outage lasted for 48 minutes; within Slack, the cascade of events that led to this outage began at 8:45am PST. It all started with a software performance bug that was caught and immediately rolled back during a routine code deployment. This triggered the auto-scaling of Slack's web tier, ramping it up to more instances than the hard limit allowed by the load-balancer. This in turn exposed a bug in how Slack updates the list of hosts in the load-balancer: some load-balancers had a mix of old, stale, and new unregistered host instances. Eight hours later, the only active host instances were the oldest ones still registered to the load-balancers. When the auto-scaling program started scaling hosts down for the night, it shut down these old instances. Since all remaining instances registered to the load-balancers were either stale or new and unregistered, the service experienced a total outage.

While this description of the Slack incident lays out the logical sequence of events that led to the outage, identifying the root cause of the problem required "all hands on deck" [9]. As soon as the alert was raised, engineers from multiple teams got together and explored several possible hypotheses based on operational data visible through their monitoring, dashboarding, and alerting infrastructure. This incident illustrates *Observability* in action, a critical capability not only at Slack, but at many other large software companies [2].

Today's web-scale, user-facing software systems and services (e.g., Slack, Google, Facebook, Twitter) are built and operated on micro-services managed by highly elastic and shared infrastructures. This *cloud-native* software ecosystem is increasingly more distributed, heterogeneous, and complex, making it challenging to predict their behavior in the face of failures and varying load [1]. Observability is emerging as a key capability for monitoring and maintaining cloud-native systems to ensure their quality of service to customers [25]. Borrowed from control theory, the notion of observability brings better visibility into understanding the complex behavior of software using telemetry collected from the system at run time [14]. Beyond simple black-box monitoring, observability provides deeper contextual insight about the correctness and performance of systems. Its goal is to minimize *time to insight* - a critical measure of understanding what is happening in the system, and why. As such, observability is inherently a data-intensive and time-sensitive process that involves humans in the loop (e.g., DevOps teams) [27].

We see observability as a data management problem. Systems are instrumented to generate large volumes of heterogeneous time series data that must indexed, stored, and queried in near-real time. Instrumentation can emit four types of data: (i) numeric data like gauges and counts, (ii) highly structured data on system events, (iii) logs of unstructured strings, and (iv) a graph of the execution path of a request. In industry, these are referred to as *Metrics, Events*[1]*, Logs, and Traces*.

Since observability is critical for meeting service-level objectives (SLOs) for web companies with millions of users, there is a lot of industrial activity in this domain. However, current solutions consist of a patchwork of various specialized tools to cater to the different needs of each time series data type. These ad-hoc solutions do not scale well and incur high performance overheads, operational complexities, and infrastructure costs [13]. There is a growing need to rethink the current design of data and software infrastructures to enable observability data management at scale.

In this vision paper, we analyze the data management requirements of observability workloads (§2) and challenges

[1]Many observability practitioners represent events as highly structured strings and a subset of logs. In §2, we describe how the structured nature of events warrants separate consideration from logs.

Data	Type	Queries	Storage	Volume	Retention
Metrics	numeric string metadata	aggregations filters on metadata	compressed time series hybrid column store	4B time series/day 12M samples/second 12 TB/day (compressed)	30 days
Events	highly structured strings or binary	filters on exact string matches	column store	250 TB/day (raw)	3-24 months
Logs	semi-structured strings	approximate string search	inverted index	90 TB/day (raw)	7 days
Traces	DAGs of durations of execution	disassociated graph search	columnar / inverted index	2 TB/day (raw)	14 days

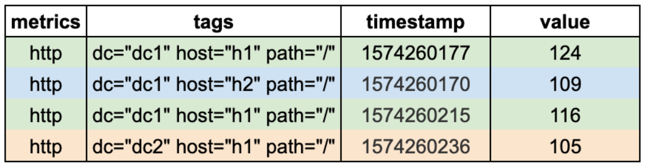
Table 1: The four categories of observability time series (MELT) widely differ in their characteristics and needs

experienced by today's systems using Slack's observability infrastructure as an exemplar (§3). Then we identify the general design principles for building web-scale ODMSs and provide the blueprint of a new architecture to realize them (§4).

2 METRICS, EVENTS, LOGS, TRACES

The fundamental challenge of ODMSs is timely insight into a system's state in the face of massive data volumes and heterogeneity. This section analyzes observability data in four categories: Metrics, Events, Logs, and Traces (MELT). We first discuss unique characteristics of these categories, followed by their common requirements for data management.

2.1 Metrics



metrics	tags	timestamp	value
http	dc="dc1" host="h1" path="/"	1574260177	124
http	dc="dc1" host="h2" path="/"	1574260170	109
http	dc="dc1" host="h1" path="/"	1574260215	116
http	dc="dc2" host="h1" path="/"	1574260236	105

Figure 2: A Metric for number of HTTP requests/second

Metrics provide quantitative measurements of system performance and availability at a specific point in time. They encompass three types of numeric data: (i) *counters* are values that can only increase or be reset to zero (e.g., total number of HTTP requests received); (ii) *gauges* are values that can go up or down to reflect system state (e.g., the number of HTTP requests waiting to be responded to); and (iii) *histograms* sample observations over a fixed time interval, counting them in configurable buckets (e.g., HTTP request durations or response sizes). In addition to a numeric value and a timestamp, metrics also include metadata as a set of key-value pairs, called *tags*. Tags identify a specific instance of a metric. A unique combination of a metric and tags is called a *time series*. A pair of a timestamp and a value is called a *sample*. Figure 2 provides an example metric, *http*, measuring the number of HTTP requests per second. Each measurement is tagged with the data center, host, and path metadata for the request's origin. There are three (color-coded) time series in this example, each with monotonically increasing timestamps.

Metrics are used in two ways: (i) to generate alerts on unexpected system state, or (ii) for analytical and dashboarding queries. Alerts are generated using small queries on the most current data (e.g., total number of HTTP requests per host per minute). Analytical queries may involve data from arbitrary times to observe system-wide trends (e.g., total number of HTTP requests per host per minute in dc1 over the last day). More complex analytics (e.g., similarity search or clustering [18, 21]) may also be performed.

Metrics require a hybrid storage engine. Prometheus [20], a widely used metric storage engine, employs a compressed storage strategy for the values of the metric and an inverted index for associated tags. This strategy results in high compression rates and fast filtering operations. Other storage strategies (e.g., data series storage [17]) have also been proposed. Slack generates about 4 billion time series per day, at the rate of 12 million samples per second, collecting 12TB (compressed) of metrics data every day. These are retained for 30 days.

2.2 Events

Events are highly structured data emitted during run time. Frequently, they come from a finite set of possible values. For example, there are 9 HTTP request methods (e.g., GET, POST) and a finite set of response status codes (e.g., "404: Not Found"). Events may also be used for high-cardinality data with higher dimensions (e.g., customer-ids and network addresses). Because events are emitted as structured strings or in a compact binary format [25], prior literature on observability typically considers events as a subcategory of logs (discussed in §2.3) [12, 25]. We categorize them separately, because the data model, queries, and access patterns for events are substantially different from those for logs.

Figure 3 shows an excerpt of raw events emitted by a system handling HTTP requests and responses.

```
Nov 20 17:35:23 2019: 192.168.100.11 POST PATH:/ 200 OK
Nov 20 17:35:24 2019: 192.168.100.10 GET PATH:/ 200 OK
Nov 20 17:35:27 2019: 192.168.100.10 DC=2 HOST=2 505 ERROR
Nov 20 17:35:28 2019: 192.168.101.11 POST PATH:/ 200 OK
Nov 20 17:35:28 2019: 192.168.101.10 GET PATH:/ui/ 404
```

Figure 3: Events emitted by an HTTP server

In addition to computing trends, events are typically used to identify specific instances of unexpected system state. For example, a SQL query like `SELECT * from Events WHERE ip=192.168.100.11 and method="POST"` would show all POST requests from the queried IP address `192.168.100.11`.

The structured nature of the data and the low selectivity filter queries make column stores ideal for events. The data in an event store is usually accessed via SQL queries over a column store. Slack generates over 250TB of raw events data per day and stores over 70 PB data at any one time. Event data is retained for relatively longer periods (3-24 months) for archival or legal audit purposes.

2.3 Logs

Logs are collections of semi-structured or unstructured strings. They expose highly granular information with rich local context. This flexibility makes logs crucial to understanding *why* unexpected behavior occurred in a service. For example, when responding to the outage incident, Slack relied on logs to identify the bug in updating the host list for their load-balancer. Similarly, for an HTTP request, an application developer might include a stack trace along with an exception showing the state of the application in an error log (see Figure 4).

```
Wed Nov 20 17:35:22 2019: GET / ERROR 500
InternalServerErrorException: HTTP 500 ...
    at ServiceUnavailableException ...
    at RedirectionException ...
```

Figure 4: Log entry with a stack trace

Unlike metrics, logs usually contain contextual information that can provide more detailed answers to questions like: “What server error caused the response to have a status of 500?". These *needle-in-a-haystack queries* are fundamentally different from the exact-match queries posed over events. They are best served using inverted index-based storage solutions due to the need for approximate string matches. Slack collects about 90TB of log data/day to be retained for 7 days.

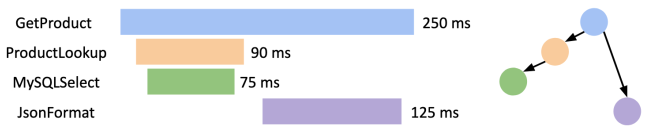
2.4 Traces

Traces encapsulate information about the execution path of a request similar to call graphs [8, 11, 23, 24]. In microservices and distributed settings, traces are call graphs across distributed services and include RPC invocations, asynchronous queues, and other inter-service communication. Traces are represented as directed acyclic graphs (DAGs) where a vertex represents a unit of execution (e.g., a function call) called a *span* and edges indicate a causal ordering (i.e., Lamport’s *happens-before* [10]) from one vertex to another. This definition is increasingly accepted with industry-wide efforts like OpenTelemetry [14].

Figure 5 shows an excerpt of a trace of a product lookup request through the *GetProduct* service. To respond to this request, *GetProduct* calls the *ProductLookup* service which in turn makes a database call named *MySQLSelect*. *GetProduct* then formats the result and finally responds to the request. The trace and its spans encapsulate the structure of the execution path of this request, and information about this path. The spans capture the duration of the execution and metadata stored as *tags* in the form of key-value pairs.

Trace ID	Span ID	Parent ID	Start Time (ms)	Duration (ms)	Name	Tags
1234	0		1574260177000	250	GetProduct	api=/getproduct, product=7
1234	1	0	1574260177020	90	ProductLookup	product=7, query=lookup
1234	2	1	1574260177030	75	MySQLSelect	query=select
1234	3	0	1574260177110	125	JsonFormat	encoding=json

(a) A table of spans in the execution of the “GetProduct” request.



(b) The trace visualized as a Gantt chart and as a directed graph

Figure 5: An example Trace for an HTTP request

There are usually two steps to accessing a specific trace: (i) *findTraces:* a user searches for traces that match a certain criteria (e.g., HTTP requests since yesterday over 200ms where a DB call returned at least 2 rows); (ii) *getTrace:* among the list of traces returned, a user selects a specific trace to view as a Gantt chart.

Although tracing has been used in distributed systems for decades [7], there is little research on the storage and management of trace data.

At Slack, trace data volumes are lower than others (e.g., 2 TB/day) because of the challenge in instrumenting and managing trace data. To manage this complexity, Slack represents spans in a format shown in Figure 5a, similar to industry efforts like OpenTelemetry’s tracing API [14].

2.5 Common Characteristics of MELT Data

As discussed in the previous subsections and summarized in Table 1, the data, query, and storage models for the four types of time series in an ODMS differ widely. However, MELT data also shares several important characteristics that influence how they should be managed overall.

Data characteristics. Fundamentally, all MELT data is *immutable* and *append-heavy*. Furthermore, because of the dynamic distributed environment, the volume of data generated over time is *highly variable (i.e., bursty)*. For example, a major auto-scaling event or a new log exception can increase input data volume rates by 10-fold for a short period of time.

Query characteristics. A *bias to freshness* accurately summarizes the general nature of queries on all observability data. To illustrate, Table 2 shows the percent distribution of Slack queries by data age, indicating that >97% of all queries are on data that is less than 24 hours old. During an incident, fresh data is needed not only to understand the current state of the system, but also to quickly see if the remediation is having the

Data Age	Logs (19.6M)	Metrics (17M)	Traces (46K)
<1 hour	92.5	94.7	85.2
<2 hours	92.6	95.2	94.8
<4 hours	94.5	97.5	95.0
<1 day	99.8	99.8	97.3

Table 2: % of queries by data age (with total query volumes in parentheses) for 1+ months of querying statistics over logs, metrics, and traces at Slack. More than 97% of the queries are for data produced in the last 24 hours.

intended effect. This implies that fresh data typically needs to be more accessible and available than older data. Furthermore, during major incidents, it is not uncommon to have twice the normal number of users interacting with the dashboards and writing custom queries against MELT data. Most queries are for *dashboards and alerts* that typically query only a small percentage of the collected data, but also require *sub-second latencies* to support real-time decision making [15].

Life-cycle management. Along with the bias to freshness, historical data is still required for a smaller percentage of queries when looking for longer term trends or for legal/business purposes. For example, during the aforementioned outage, Slack engineers looked at data over the last few weeks to check for any seasonal patterns. Hence, MELT data commonly requires data life-cycle management for fresh and historical data side by side, indicating that a *tiered storage strategy* based on data age should be designed into the ODMS.

3 TODAY'S CHALLENGES

Current ODMS solutions deployed in industry are custom-built approaches that only partially address the requirements of MELT data management. These systems are built as a result of reactive implementation [13], without guiding design principles and therefore face similar practical challenges. In this section, we present Slack's current observability system infrastructure as a case study to reveal the key practical challenges to be considered when architecting a scalable ODMS.

3.1 Observability at Slack

Figure 6 depicts Slack's current observability system infrastructure. Like many industry solutions, it consists of multiple tools, ingestion pipelines, and storage engines to collect, store, and serve MELT data generated by Slack applications.

Slack uses Prometheus [20], a single-node pull-based system, to store and serve metrics. It scrapes metrics data from HTTP endpoints exposed by Slack applications. For high availability, Slack uses a pair of Prometheus servers, each maintaining an independent copy of the data. Each application is allocated a pool of 2 to 64 Prometheus servers. In total, Slack runs about 100 such pools.

Events, logs, and traces (ELT) are pushed to Apache Kafka. For durability, Secor [19] consumes and writes the raw data

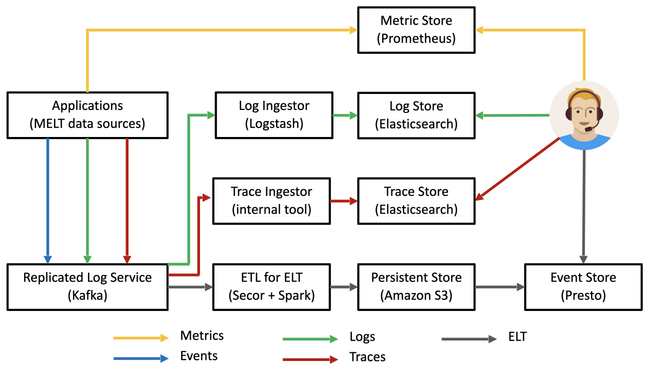

Metric Store (Prometheus)
Applications (MELT data sources)
Log Ingestor (Logstash)
Log Store (Elasticsearch)
Trace Ingestor (internal tool)
Trace Store (Elasticsearch)
Replicated Log Service (Kafka)
ETL for ELT (Secor + Spark)
Persistent Store (Amazon S3)
Event Store (Presto)
Metrics
Events
Logs
Traces
ELT

Figure 6: In Slack's current observability infrastructure, an engineer manages and queries four independent systems, using different query APIs and tools.

to Amazon S3. A custom Apache Spark job copies and transforms this data into a columnar Parquet file, also stored into Amazon S3 [26, 28]. These Parquet files can be queried via SQL using Presto [22]. In addition, Slack uses Logstash to ingest logs into Elasticsearch, and an internal tool to ingest traces into a vendored tracing solution and an in-house trace store backed by Elasticsearch [4]. Logs are retained for 7 days and traces for 14 days. Logs and traces are also written to a data warehouse (Presto) for historical querying.

3.2 Practical Challenges

Slack's current ODMS architecture faces a number of practical challenges unique to the requirements of growing observability workloads. Although we use Slack as a case study, we believe that almost all ODMSs currently deployed in industry face one or more of these practical challenges. We group these challenges into three overarching categories:

High Operational Complexity. The infrastructure to serve MELT data at Slack contains over 20 separate software components. Each component has a unique architecture and, as a result, needs custom operations for cluster management, security, and capacity planning. This heterogeneity leads to complex solutions. For example, Prometheus is a single-node system and maintains independent copies of the data to meet high-availability requirements. Meanwhile, although Elasticsearch has built-in replica management, at Slack's scale, it is challenging to manage and operate. During the May 12 incident, data volumes spiked to 4 times the normal peak volume. In such scenarios, the monitoring team performs complex scale-up operations to continue ingesting fresh telemetry data.

Maintaining Low Query Latency. Queries on observability data are highly skewed. Over 97% of queries access data $<$ 24 hours old for near-real-time alerting. Dashboards and major incidents or product changes result in significant spikes in the number of queries putting significant pressure on the ODMS. During the May 12 incident, Slack looked at the data for the prior 8 hours while handling 2 times the historical peak load. The high operational complexity in Slack's ODMS makes scaling in response to overall query workload and maintaining low query latency a significant challenge.

High Infrastructure Cost. At the petabyte scale, meeting ever-increasing data retention requirements and ensuring data availability in bursty workloads quickly becomes costly. At Slack, significant time is spent in balancing the cost of infrastructure, performance, durability, and availability of the data. For example, Slack duplicates processing ELT in Presto for availability and durability, at the risk of slower performance and higher infrastructure costs. Slack also provisions based on historical peaks to handle spiky workloads such as during the May 12 incident. The new peak load is now 2 times the peak load prior to the incident. This provisioning strategy adds to the infrastructure cost.

4 ODMS DESIGN PRINCIPLES

We now propose a set of design principles that address the real-world requirements and challenges described in prior sections. Considering the heterogeneous nature of MELT data and the need for reducing the complexity of managing this data while meeting the query performance requirements in a scalable manner, we believe that an ODMS should adhere to four core design principles:

Decouple real-time and historical data management. §2 and Table 2 showed that the ODMS's workload is significantly different from hybrid database workloads that combine OLAP and OLTP (e.g., HTAP [16]). It ingests petabytes of potentially bursty immutable writes and queries are biased to data < 24 hours old. ODMSs designed with this workload in mind decouple real-time and historical data management. This decoupling maintains fast ingestion rates, low query latency on recent data, and minimizes the cost of storing and accessing historical data.

Unify MELT data life-cycle management. Time ties all MELT data management to a common framework. The unique real-time and historical data management strategies should be governed in a unified data life-cycle based on data age. Doing so should abstract away the movement of data from real-time to historical storage, and decrease the cost and complexity of transparently accessing all data over arbitrary periods of time.

Provide a single query interface for MELT data. Observability queries commonly require data from different MELT types. A single query interface abstracts the individual storage and processing requirements of MELT data, decreasing the complexity of writing these queries, and providing opportunities for optimization across the storage strategies. This principle lends itself to a "polystore-like" pluggable storage engine architecture [3].

Support cloud-native, distributed deployment. Because of the highly bursty nature of both reads and writes, various tiers

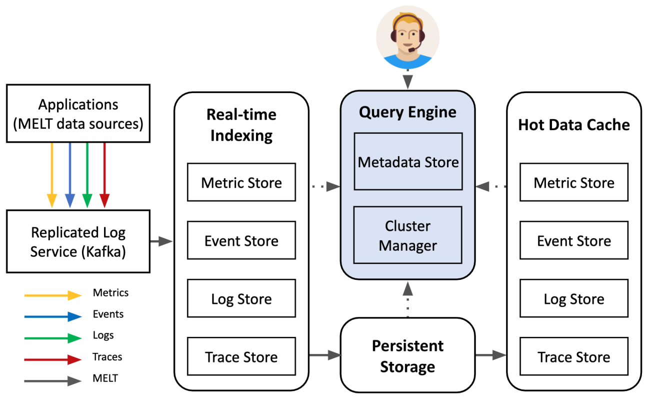


Figure 7: In the new architecture based on ODMS design principles, a user manages and queries a single system.

of the system should be distributed and elastic by design, so that the system can scale with the changing workload. A cloud-native ODMS allows the observability team to make a flexible trade-off between cost and performance for serving MELT data based on workload and the data life-cycle.

These four principles must be integrated into the design process of architecting an ODMS. Unlike current industry practice where ODMSs are built as a patchwork of features responding to ad-hoc requirements, following these principles results in a coherent architecture that provides control over an ODMS's complexity, performance, and cost.

Figure 7 shows an overview of such an architecture that realizes these design principles. It unifies the data life-cycle for MELT data into a single ODMS infrastructure. Unlike Slack's current system, which is influenced by modern Lambda architectures [5], this design is influenced by the design of modern polystore architectures [3]. This decreases the storage requirements and complexity of maintaining multiple data pipelines, and addresses the need for transparently coordinating multiple storage engines.

Data ingestion and storage. All MELT input data from instrumented applications are ingested into a *Replicated Log Service* (e.g., Apache Kafka [6]). The *Real-time Indexing* tier comprised of storage engines specific to each data type pulls data from the log service and indexes them for fast access. Periodically, indexed data files are migrated to the *Persistent Storage* tier similar to Amazon S3. This migration is coordinated by the *Cluster Manager*. At query time, the *Query Engine* coordinates the *Real-Time Indexing* and *Persistent Storage* tiers using the *Metadata Store*. When necessary, the *Query Engine* pulls hot historical data from the *Persistent Storage* tier into the *Hot Data Cache* to maintain query performance.

Query processing. The *Query Engine* services all queries. Using the *Metadata Store*, it coordinates between the three data tiers (*Real-Time Indexing*, *Persistent Storage*, and *Hot*

Data Cache) to filter data based on time-range and to optimize queries across multiple data types (akin to joins by time). It determines optimal data placement based on expected query distributions and costs. For example, ad-hoc queries that access data in *Persistent Storage* may not need to be cached. However, when accessing historical data repeatedly while resolving an issue, the query engine might decide to copy data to the *Hot Data Cache* to minimize query latency.

Distribution and availability. The ODMS's distribution strategy should maintain high availability during bursty periods of new data and during periods with particularly heavy query load. The *Real-Time Indexing* and *Hot Data Cache* tiers are natively elastic, where independent replicas are spun-up on-demand. The *Query Engine*'s *Cluster Manager* monitors the *Real-Time Indexing* tier's workload to determine whether to scale up any of the component stores. During periods of particularly heavy data writes (e.g., from instrumented applications) or queries on fresh data, it scales up the *Real-Time Indexing* tier. During particularly heavy reads of historical data, it scales up the *Hot Data Cache*. The two-tier design also helps support variable availability of data (e.g., higher availability guarantees for more recent/valuable data).

The architecture is generalizeable to settings where observability telemetry is unified in a centralized ODMS that makes the MELT data easily accessible to users. An ODMS architecture following these principles is intended to help during peak loads such as the one during Slack's May 12 incident. The **Query Engine** provides a single query interface and a unified view of the MELT data. It provides an abstraction over the *complexity* of querying and managing the heterogeneous types and their life-cycles across the **Real-Time Indexing** and the historical **Persistent Storage** tiers. The elastic **Hot Data Cache** responds to peak loads based on data access patterns to *maintain low query latency*. Finally, the **Cluster Manager** manages data life-cycles and operational complexity in this system. It also scales or shrinks based on the workload, thereby decreasing the system's overall *infrastructure cost*. These principles result in shorter time to insight during peak loads and decrease overall complexity and costs.

5 CONCLUSION

Observability data management is an emerging area of research that requires more attention from the database community. In this paper, we discussed real-world experience with observability data and its use cases at Slack – a cloud-based team collaboration service. The heterogeneous nature of time series data as well as varying workload characteristics call for a new observability data management architecture. In response, we proposed a new cloud-native polystore-like architecture that decouples real-time and historical data access tiers from the underlying persistent storage and the querying tier in a way that enables scaling them independently. We are currently working on building an initial prototype of this design to test with production data from Slack.

REFERENCES

[1] R. H. Arpaci-Dusseau et al. 2018. Cloud-Native File Systems. In *USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*.

[2] C. Chan et al. 2020. Debugging Incidents in Google's Distributed Systems. *ACM Queue* 18, 2 (2020).

[3] J. Duggan et al. 2015. The BigDAWG Polystore System. *ACM SIGMOD Record* 44, 2 (2015), 11–16.

[4] C. Gormley et al. 2015. *Elasticsearch: The Definitive Guide*. O'Reilly Media.

[5] M. Hausenblas et al. 2017. Lambda Architecture. http://lambda-architecture.net.

[6] N. Narkhede et al. 2017. *Kafka: The Definitive Guide Real-Time Data and Stream Processing at Scale*. O'Reilly Media.

[7] J. Jeffrey et al. 1987. Monitoring Distributed Systems. *ACM Transactions on Computer Systems (TOCS)* 5, 2 (1987), 121–150.

[8] J. Kaldor et al. 2017. Canopy: An End-to-End Performance Tracing And Analysis System. In *SOSP*. 34–50.

[9] R. Katkov. 2020. All Hands on Deck. https://slack.engineering/all-hands-on-deck-91d6986c3ee.

[10] L. Lamport. 1976. The Ordering of Events in a Distributed System. *Communications of the ACM* 21, 7 (1976), 558.

[11] J. Mace et al. 2015. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *SOSP*. 378–393.

[12] S. More. 2018. *A Practical Observability Primer*. mStakx.

[13] S. Niedermaier et al. 2019. On Observability and Monitoring of Distributed Systems – An Industry Interview Study. In *ICSOC*.

[14] OpenTelemetry. 2019. The OpenTelemetry Open-Source Observability Framework. https://opentelemetry.io/.

[15] J. O'Shea. 2020. Building Dashboards for Operational Visibility. https://aws.amazon.com/builders-library/building-dashboards-for-operational-visibility/.

[16] F. Özcan et al. 2017. Hybrid Transactional/Analytical Processing: A Survey. In *ACM SIGMOD Conference*. 1771–1775.

[17] T. Palpanas. 2015. Data Series Management: The Road to Big Sequence Analytics. *ACM SIGMOD Record* 44, 2 (2015), 47–52.

[18] T. Palpanas et al. 2019. Report on the First and Second Interdisciplinary Time Series Analysis Workshops. *ACM SIGMOD Record* 48, 3 (2019), 36–40.

[19] Pinterest. 2017. Pinterest Secor: A Service for Implementing Kafka Log Persistence. https://github.com/pinterest/secor.

[20] Prometheus. 2012. Prometheus Documentation. https://prometheus.io/docs/concepts/metric_types/.

[21] J. Rodrigues et al. 2017. Sieve: Actionable Insights from Monitored Metrics in Distributed Systems. In *ACM Middleware Conference*. 14–27.

[22] R. Sethi et al. 2019. Presto: SQL on Everything. In *IEEE ICDE*.

[23] Y. Shkuro. 2019. *Mastering Distributed Tracing: Analyzing Performance in Microservices and Complex Systems*. Packt Publishing.

[24] B. H. Sigelman et al. 2010. *Dapper: A Large-Scale Distributed Systems Tracing Infrastructure*. Technical Report. Google, Inc.

[25] C. Sridharan. 2018. *Distributed Systems Observability: A Guide to Building Robust Systems*. O'Reilly Media.

[26] D. Vohra. 2016. Apache Parquet. In *Practical Hadoop Ecosystem: A Definitive Guide to Hadoop-Related Frameworks and Tools*. 325–335.

[27] A. Wiedemann et al. 2019. The DevOps Phenomenon. *ACM Queue* 17, 2 (2019).

[28] M. Zaharia et al. 2010. Spark: Cluster Computing with Working Sets. In *USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*.