Understanding the limitations of pubsub systems

Atul Adya
atul.adya@databricks.com

Phil Bogle
phil.bogle@databricks.com

Colin Meek
colin.meek@databricks.com

Abstract

This paper argues that publish-subscribe (pubsub) systems bundle both a messaging abstraction and a hard-state storage layer, and that this hurts their robustness, performance, and correctness for a variety of use cases. Pubsub fails to achieve its central goal of decoupling publishers and subscribers, violates the end-to-end principle, and exposes an ad hoc storage abstraction with a bespoke API and limited power.

We explain how to correct these issues by unbundling pubsub into: (1) a durable store, and (2) a *watch* system that notifies consumers about changes to the store. This approach respects the end-to-end argument by offering consumers guarantees relative to the store rather than the pubsub system. We show how this model addresses the challenges introduced by pubsub systems for various use cases and opens up new areas of research.

ACM Reference Format:
Atul Adya, Phil Bogle, and Colin Meek. 2025. Understanding the limitations of pubsub systems. In *Workshop in Hot Topics in Operating Systems (HOTOS 25), May 14–16, 2025, Banff, AB, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3713082.3730397

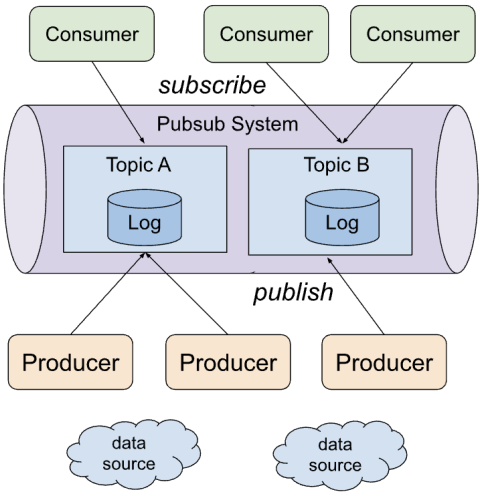

Figure 1: Pubsub model

1 Introduction

Pubsub systems [5, 12, 13, 24] have become increasingly popular in datacenter environments in recent years [19, 20, 25, 32, 38]. They are positioned as a solution to many fundamental problems in distributed systems, including availability, consistency, decoupling of microservices, and resilience.

Most systems attempt to guarantee delivery to consumers via a bundled, durable message log. To prevent unbounded log growth, they support garbage collection of old messages. This undermines the goal of decoupling and correctness, because messages can be lost without notifying the application or allowing it to recover. (Some older non-persistent pubsub systems provide only best-effort delivery [19]. The arguments in this paper apply even more strongly in that case, because message loss is more frequent and just as undetectable.)

This paper argues that pubsub systems violate the end-to-end argument: their delivery and ordering contracts add complexity and cost but do not provide end-to-end correctness with respect to authoritative data sources. Furthermore, their lack of support for dynamically sharded consumers limits application scalability.

We show how to resolve those issues by unbundling pubsub into an explicit storage abstraction exposed to producers and consumers (via a narrow, read-only view), and a notification abstraction, *watch*, used by consumers to learn about changes to the store. The watch abstraction [21] has been popularized by Kubernetes [17] and Spanner [16]. It does not require additional hard state and can be implemented as a layer on top of a traditional store. It includes resync signals that enable lagging consumers to be notified and automatically recover, and progress signals that allow subscribers to expose snapshot consistent views of the source.

We show that the proposed model is general enough to handle all pubsub use cases, often with improved efficiency and/or improved end-to-end correctness properties. This is because the explicitly exposed store is at least as powerful as the implicit message log store in traditional pubsub systems, and the watch mechanism is strictly more powerful than traditional subscribe mechanisms.

The organization of the remainder of this paper is as follows: Section 2 discusses pubsub use cases. Section 3 enumerates limitations of pubsub. Section 4 proposes our alternative abstraction and watch API and explains its advantages. Finally, Section 5 presents areas for future research.

2 Pubsub model and use cases

In the pubsub abstraction (Figure 1), producers publish messages to *topics*, and consumers subscribe to topics of interest. The pubsub system is responsible for distributing published messages to subscribed consumers without consumers needing to know about the source of the messages. Datacenter pubsub systems are capable of handling tens of thousands of producers and consumers, and millions of messages per second per topic.

Pubsub systems support two consumer models. *Consumer groups* distribute messages among members, ensuring that each message is routed to and acknowledged by a consumer within the group.

HOTOS 25, Banff, AB, Canada

ACM ISBN 979-8-4007-1475-7/25/05
https://doi.org/10.1145/3713082.3730397

The system can select a consumer at random, based on the message's partition, or according to a key specified in the message. *Free consumers* (to use terminology from [26]) handle all messages in a topic or topic partition. Neither approach allows a dynamically sharded consumer to subscribe to arbitrary subranges of the key space.

Pubsub systems are used in industry for a wide variety of use cases, surveyed in [33] and [34] and summarized below. (Marketers have also invented many others terms for logically equivalent uses, e.g. "enterprise event bus" and "IoT data streaming".)

Event ingestion and fanout: Pubsub systems can be used as an intermediary that receives ingested events (e.g. logs, sensor data, or website activity) and that fans events out to all interested systems and devices. We refer to the storage for the events as *ingestion storage*.

Replication across stores: Pubsub can be used to replicate data from a source store to a target store. In this scenario, a change data capture (CDC) system feeds change events from the source to the pubsub system, which in turn delivers these events to the target store. We refer to the source store for replication as *producer storage*.

Cache invalidation/freshness: Pubsub systems are also used to maintain the freshness of distributed caches. In this case, producer storage publishes updates, such as object IDs or updated payloads, to the pubsub system, which propagates these updates to distributed cache nodes. This ensures that caches remain consistent and up-to-date.

Work queueing and balancing: Pubsub can be used to queue and balance messages representing tasks across workers. Tasks are published as messages and processed and acknowledged by a worker in a consumer group.

3 Limitations of pubsub systems

Decoupling producers and consumers seems like a sound strategy, aligned with the principle of loose coupling. However, in practice, pubsub systems succeed at this goal only most of the time, creating emergencies/outages when they fail. Many practitioners view these problems as rare and consider it acceptable to address them with manual operational processes when they arise.

3.1 Failures of loose coupling

Pubsub's first limitation is a Kafka-esque interpretation of "decoupling". Pubsub systems allow consumers to accumulate large backlogs and may garbage-collect unprocessed messages if the backlog lasts too long. However, they neither inform consumers that this is happening, nor provide a mechanism for laggy consumers to "catch up" or recover lost state from a source of truth. Systems deliver messages approximately in publishing order, so excessive backlogs are indistinguishable from silent outages.

Pubsub systems assume that consumers will not accumulate excessive backlogs. In general, this is not true. For example, in a cache invalidation use case, an actual consumer was unavailable for multiple days because its data center was under maintenance. This resulted in a huge backlog even after adding emergency resources. Cache invalidation took hours rather than seconds, making it effectively useless for users.

Pubsub systems also assume that a retention period of (e.g.) several days will be sufficient for even the slowest of consumers to consume each message. However, systems will eventually garbage-collect messages even if some consumers have not processed them. This data loss sacrifices correctness for applications that depend on reliable delivery.

Although some pubsub systems offer the option to retain messages indefinitely, this is undesirable because pubsub systems offer a limited API that does not allow the relevant state to be efficiently queried by the application. Instead, as we discuss in Section 4, it is better to keep persistent state in a dedicated store. Some pubsub systems [24] support topic compaction, where each message is associated with a key. Compaction allows applications to configure a recent window for which every version is kept and before which only the last version is maintained. Unfortunately, without notification, subscribers do not discover that unseen events have been compacted. Compaction defers but does not eliminate message loss. Such intermittent data loss can affect the correctness of the application.

In general, system operators rely on a variety of ad hoc, manual procedures to recover when alerted about data loss or excessive backlogs in pubsub. These include deleting backlogs, relying on less effective fallback mechanisms such as TTLs for cache invalidation, and restoring from backups for replication. These mechanisms not only incur manual toil but also sacrifice correctness, availability, and/or latency. Manual heroics are risky [29], and should not be required in "loosely coupled" systems.

Decoupling also breaks down with respect to application scalability. Practitioners recognize that affinitized consumers are important for scale and efficiency – for example, to enable effective caching. However, existing pubsub consumer affinity mechanisms based on the message key or pubsub partition do not support independent, dynamic sharding of loosely-coupled application consumers.

3.2 Violations of the end-to-end principle

Pubsub systems provide guarantees at their layer – ordering, at-least-once delivery, transactions – that do not result in meaningful end-to-end guarantees. We will illustrate these limitations by considering popular use cases.

3.2.1 Replication across storage systems

In pubsub-based replication, a change data capture (CDC) system publishes change events from producer storage, and consumers apply them to a target store. Ideally, the target store should achieve point-in-time consistency; i.e. it only externalizes states that actually existed in the source.

The producer store is the authority on event ordering and transaction boundaries. When the pubsub system establishes a competing order of events or supports its own transactions, this adds complexity and cost. But at scale these guarantees at the pubsub layer are not useful, because they do not help provide a consistent view of the source from the target store or cache.

By serially publishing and applying transactions via a pubsub system that preserves ordering, an application could maintain snapshot

consistency in the target store. Unfortunately, the serial approach is not scalable; to avoid a scale bottleneck we need to *concurrently* publish and apply change events. But we can't simply apply change events in an arbitrary order. Reordering inserts, updates, and deletes could overwrite with stale state or resurrect recently deleted rows, violating eventual consistency with the producer store. By introducing version checks and tombstones, we can eliminate some replication errors, but still risk snapshot consistency violations. For example, suppose that in producer storage we remove a member from a group and then give that group access to a document. If we reverse the order of those operations on the target store, snapshot consistency is violated, because the target store transiently records a state where the member has access to the document, a state that never existed in producer storage.

Another strategy is to partition the pubsub topic such that any given row will be statically assigned to a single topic partition, and ensure that each partition is processed serially. This approach avoids version checks and tombstones but snapshot anomalies are still possible because transactions affecting multiple partitions are not atomically applied and the global transaction order of the source may be violated.

In practice, the challenges are daunting enough that some practitioners opt to give up on either scalability or consistency via the replication protocol. Some systems serialize all operations. Other systems periodically restore full snapshots of the source to the target. This ensures that any replication errors are eventually addressed, but over a much longer time frame.

3.2.2 Cache invalidation

When the target of the change feed is a cache rather than another storage system, the same concerns about out-of-order delivery remain, but there is an additional complication. There is no centralized target store, so if the new owner caches a stale value but the invalidation for that value is acknowledged by an old owner, that stale value can be cached in the new owner indefinitely. Modern caches employ dynamic key range assignment [3] for robustness, offering better availability/balancing than static approaches. However, pubsub consumer group limitations make missed invalidations possible during these dynamic handoffs.

Figure 2 shows a race between the invalidation of object x and the reassignment of x from pod p_{old} to pod p_{new} by an auto-sharder. p_{new} may learn about the reassignment before the pubsub system, and fetch the current value of x. When the pubsub system is subsequently informed of an update to x, the pubsub system causes p_{old} rather than p_{new} to update its cache. Therefore p_{new} *never* receives the updated value.

Some of the cases where change events are missed can be mitigated by using a leasing mechanism to ensure that at most one cache server at a time is allowed to acknowledge a change event from pubsub. But leases introduce an availability tradeoff because there will be times when there is no owner for a range of keys. As with replication, practitioners have fallback strategies to paper over inconsistencies permitted by pubsub. Using TTLs on cache entries ensures that stale entries eventually age out. Or in some systems, each cache server subscribes to the *entire* feed using free consumers (using the terminology from [26]), an approach that does not scale as update rates increase.

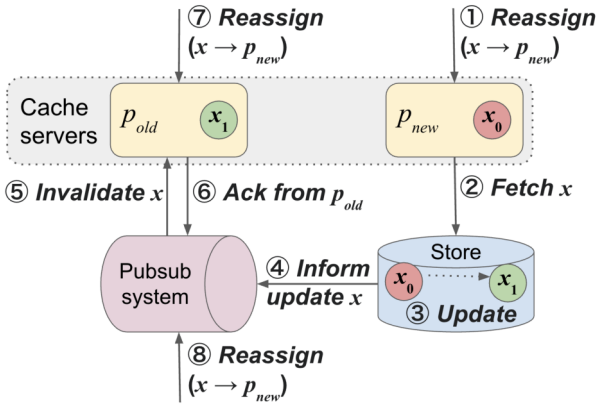

Figure 2: Invalidation eventual consistency failure in the presence of auto-sharding

3.2.3 Event ingestion and fanout

In this use case (see Section 2), receivers are expected to get all events from the publisher promptly to enable downstream analysis, such as fraud detection or sensor-based alerting. However, as discussed in Section 3.1, head-of-line blocking can occur and large backlogs can develop. Additionally, data may be lost due to garbage collection, resulting in a loss of semantic guarantees.

3.2.4 Work queueing and balancing

This scenario shares the same challenges as the ingestion case but introduces an additional inefficiency: lack of support for affinitized load balancing across dynamically sharded workers. Affinitization is important for efficient work processing because it enables consumers to cache state across for ranges of keys they are assigned. Dynamic sharding is important for availability because it ensures that workers are not overloaded and that affinitized work is reassigned when workers become unavailable. Additionally, the event-based approach makes it is more difficult to achieve correctness compared with the state-based approach using watch, as explained in Section 4.3.

3.3 Ad hoc storage APIs

As previously noted, pubsub systems include a storage layer, and have gradually introduced features such as schemas, versions, and transactions. Each pubsub system has organically developed its own set of ad hoc APIs to support these features, along with specialized extensions—such as the "replay and snapshot" functionality in GCP [15] and the "dead-letter queues" in Azure [6].

Rather than using pubsub as an ad hoc storage system, we believe applications should have the flexibility to use a traditional or special purpose storage system. Full-fledged storage systems provide various models that better suit applications' needs, offering greater power and standard APIs. For example, time-series databases or newer logging abstractions [7, 28] are ideal for log-like storage. Likewise, for structured storage, NoSQL databases like Bigtable [10] or SQL systems such as Spanner [11], CockroachDB [31], and TiDB [22] offer features that facilitate robust data access, e.g. reads, scans, writes, indices, and foreign key constraints.

4 Our solution: Explicit storage with Watch

We propose an alternative model that resolves the issues of pubsub, unbundling notification and storage, explicitly exposing the storage abstraction, and defining a watch API that enables end-to-end correctness without sacrificing scalability. In this model, producers write changes to a designated storage system and consumers receive mutation events from a view of the store using a *watch* API.

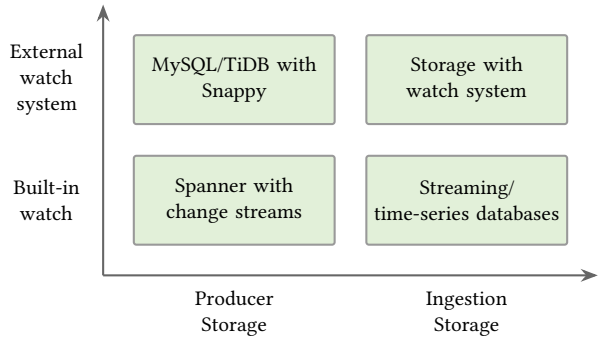

Figure 3: Separation of storage and notifications

Figure 3 illustrates design choices for unbundling storage and notification. The storage system, represented on the X-axis, can either be producer storage or ingestion storage (as defined in Section 2), to address scenarios where the source is a persistent store and where the source data are ephemeral respectively. The notification mechanism, represented on the Y-axis, may either be implemented directly by the storage system or provided as an external layer built on top of it. Below are examples for some of these cases.

- Spanner can serve as producer storage and has a built-in watch mechanism called Change Data Streams [16]. Other examples include Kubernetes API server [17], which is backed by the watchable etcd store [18].
- MySQL and TiDB are also popular producer storage systems, and we have implemented an external watch subsystem called *Snappy* on top of them, treating them as key value stores. Snappy is not yet published.
- Time-series databases [9, 23, 36], data stream management systems [1, 4, 8, 37] or other structured stores [10, 22, 31] can serve as ingestion stores, and offer efficient access to time-series data.

A refined version of a pubsub system such as Kafka would fit into the bottom right quadrant (ingestion storage with built-in watch), with some changes to its API to make the implicit storage layer more explicit. However, our model generalizes to other types of ingestion storage in cases where those are better suited to the application's requirements.

Guarantees: In our proposed approach, consumers receive guarantees with respect to the storage being watched whether permanent or temporary, i.e., the producer or ingestion storage. For instance, when watching a producer store in scenarios such as replication and cache invalidation, target stores and caches provide end-to-end guarantees relative to the producer store. This approach is fundamentally different from traditional pubsub systems, which interpose a problematic intermediate storage layer.

4.1 Hiding producer store internals

Our approach might seem to expose the internal storage format of producer storage. However, this is not the case. The producer can present a filtered view that exposes a limited subset of derived values to consumers. For example, consider a source managing contact information. It can create a new table or view that contains just those values and make that view accessible to consumers. This mechanism is similar to the control found in pubsub-based architectures. The only difference is where the consumed data are stored: in the producer's storage as opposed to the hidden storage of the pubsub system.

4.2 Watch API

We now present a watch API that reliably notifies the consumer of changes to the producer storage. The system supporting this API, illustrated in Figure 4, distributes change events, organized by key and by transaction version (e.g., "account *A* has balance $20 as of version 40"). A simplifying assumption is that the source of truth has monotonic transaction versions, e.g. TrueTime timestamps in Spanner [11], TSO timestamps in TiDB [35], gtid in MySQL [30], etc., that captures the agreed upon transaction order. See [37] for more sophisticated schemes.

4.2.1 Consumer API

The consumer, which we refer to as a *watcher*, requests state for a range of keys starting from a particular transaction version via a *watch* call:

```
class Watchable {
  Cancellable watch(
    Key low, Key high, Version version,
      WatchCallback callback);
}
class WatchCallback {
  void onEvent(ChangeEvent event);
  void onProgress(ProgressEvent event);
  void onResync();
}
struct ChangeEvent {
  Key key; Mutation mutation; Version version;
}
struct ProgressEvent {
  Key low; Key high; Version version;
}
```

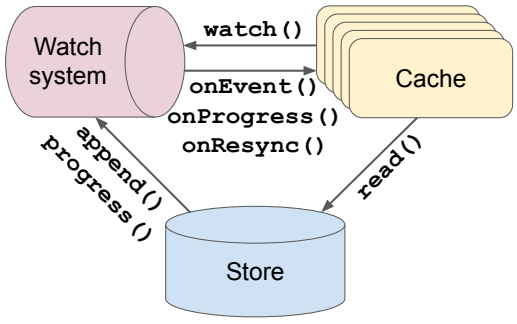

Figure 4: Unbundled architecture: Storage with watch

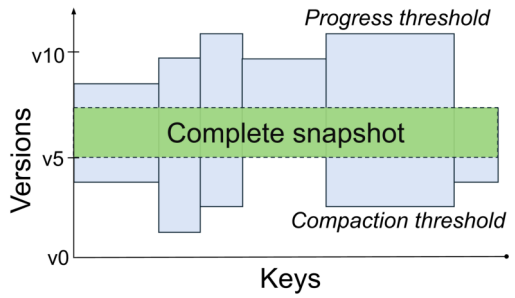

Figure 5: Knowledge by key and version for a watcher

In addition to events capturing what has changed in the store subsequent to the requested version (`onEvent`), the watch stream includes:

- Progress events (`onProgress`), indicating that all change events affecting some or all of the keys being watched have been supplied up to some version.
- Resync events (`onResync`), indicating that the version known to the watcher is no longer retained. This event prompts the watcher to read a recent snapshot of the state from the store then catch up by issuing a watch request starting at the snapshot version. Note that it is acceptable to read a stale snapshot, so we can optionally reduce load on the underlying storage by reading from a replica instead.

Applications may directly implement the watch callback interface, or may leverage linked caches similar to [2] that speak that protocol.

4.2.2 Ingester API

The store may directly implement the watch contract, but range-scoped progress events also allow the store to convey progress in a partitioned log to a separate watch system via an *Ingester* interface. Note that the watch system may use a storage system to maintain larger than memory data structures, but unlike in a pubsub system, we are not introducing any intermediate hard state. This is soft state that can be recovered if deleted (at the expense of some increased latency or staleness, but there is no data or consistency loss).

```
class Ingester {
  void append(ChangeEvent event);
  void progress(ProgressEvent event);
}
```

Once the store confirms that all updates below a specific version have been applied to a key range, it sends a `progress` event to the watch system. Progress events are scoped to key ranges rather than being global or tied to static partitions. This design enables scalability by allowing each system layer to define its own partition boundaries which can evolve independently, supporting loose coupling between layers.

4.3 Applying watch to pubsub use cases

In this section, we describe how to apply this model to the major pubsub use cases, and in the next section we detail why this approach is superior.

Caching and replication: Improving on the pubsub model, the watch model allows caches or storage replicas with modest capabilities to serve snapshot-consistent queries, even when dynamically sharded. They can use progress events to track key ranges and version windows for which they have complete knowledge and can serve consistent snapshot results. In a distributed cache or store, multiple affinitized servers may have overlapping and redundant knowledge regions for improved availability and performance.

Figure 5 illustrates the knowledge regions maintained by a watcher. Each blue rectangle represents a knowledge region — a key range and version window that define the versioned state the watcher knows for that range. This allows the watcher to serve snapshot-consistent queries within a single range, or stitch together a consistent snapshot across multiple ranges, as long as appropriate versions exist in each range (e.g., the green box in the figure). Although the figure depicts a single watcher, one can imagine combining knowledge regions across multiple watchers to serve snapshot-consistent queries at a broader scale. Each knowledge region is immutable: once a value is written at a given version, it does not change. This immutability enables dynamic replication and repartitioning of data without compromising consistency.

Event ingestion and fanout: To support ingestion in our model, the publisher exposes an ingestion store, e.g. a time-series database optimized for ingestion of events. As with a pubsub topic, the ingestion store isolates the main application database from load and security risks. Producers insert events into the ingestion store. Consumers watch all or a portion of the key range of the database to learn about new events. They may also query the ingestion store to obtain state if needed. As we will discuss in Section 4.4, this approach addresses the backlog and efficiency issues caused by the pubsub model.

Work queueing and balancing: Our approach enables affinitized, dynamically sharded workers, and reframes the problem as one of advancing entities to some desired state. Applications use an auto-sharding system [3, 27] to dynamically assign and replicate ranges of keys to workers based on load and health. Each worker initially queries the database for assigned entities requiring attention, and then uses watch to identify other such entities. The application can then prioritize entities, fully mitigating head-of-line blocking problems. By observing the current state rather than tracking a sequence of potentially unreliable, disordered events, applications become significantly more robust in distributed environments, especially for complex workflows.

Consider for example the problem of provisioning virtual machines for online data processing workloads in a cloud environment. This coordinator service's goal is to ensure that every workload is running on some set of virtual machines. The pubsub model encourages applications to enqueue tasks corresponding to each step of the workflow when a workload is added, e.g. to acquire VMs, bootstrap images, configure networks, start processing. However, in practice, the coordinator must constantly reconcile the current set of configured workloads with the set of available compute resources. The event-based approach introduces complexity because the state of the world (including available compute resources) changes constantly and in general does not match the state when the work event was enqueued. By watching both the desired configuration (which

workloads should be running) and the actual configuration (the states of the available VMs and allocations of work), the coordinator can correctly advance the actual state to the desired configuration.

4.4 Advantages over standard pubsub

We now recap the advantages of this unbundled architecture.

Better treatment of backlogs: Unlike pubsub, watch does not require unbounded backlogs for correctness. The watch system can send a resync signal to a consumer whenever its backlog is excessive. A lagging consumer can use the exposed store view to efficiently fetch a snapshot of state from the source database and resume watching from that later version.

Unbundling of storage and watch: Applications use whichever store offers the best combination of guarantees, features, and performance for their use case, rather than being locked into particular features of the storage system bundled into the pubsub systems. An external watch system can provide watch on top of any store that supports the ingestion interface. Applications can choose between different watch systems optimized for different scale points, e.g. degree of fan out.

End-to-end correctness: Key-range watches allow partitioned consumers to receive only the events they need. Key-range progress events allow all layers to serve consistent snapshots even in the presence of dynamic sharding.

Efficiency: The watch design avoids the need for an additional hard state message log and relies instead on the existing hard state provider store or an ingestion store. Unlike consumer groups, range-based watches allow related work to be affinitized to dynamically sharded servers. Affinitization enables efficient work on the same or nearby keys, e.g., for caching or when updates have to be applied to a replicated store.

Standard, powerful storage and notification APIs: In our model, applications benefit from the standard watch API andfull-featured storage APIs rather than relying on ad hoc pubsub APIs designed to partially compensate for the inherent shortcomings of the pubsub model.

5 Areas of future research

The storage-plus-watch model opens up several research opportunities to build scalable components and end-to-end correct applications:

Standalone watch system: A scalable, standalone watch system that implements the `Ingester` and `Watchable` contracts enables us to add watch capabilities to storage systems that lack native support. We are building a system called Snappy for a specific distributed storage system. However, we believe further research is needed to generalize this design to a wide variety of storage systems and scale requirements.

Auto-sharded caches supporting snapshot consistency: Snapshot consistent caches provide good semantics while lowering latency and improving scalability. The watch contract permits caches to expose snapshot consistent views, even in the face of dynamic repartitioning driven by an auto-sharder[3, 27]. Efficiently stitching together consistent views of source data from knowledge regions, potentially spread across multiple cache servers, will require careful protocol and data structure design.

Replication across different stores: While pubsub is commonly used to replicate state between heterogeneous stores, it often suffers from poor scalability, weak semantics, or both. Snapshot semantics have been achieved for replication in homogeneous database deployments for read-only replicas using internal protocols, as in Spanner [14]. The watch-based approach described in Section 4.3 offers a promising direction for replicating across diverse, real-world storage systems while continuing to provide strong semantics and low latency at scale. However, much work remains to achieve this in practice.

6 Summary

We have demonstrated that pubsub suffers from many issues because of its hidden storage layer, and that it is possible to cleanly extract and separate that layer using a watch abstraction. Unlike pubsub, explicit storage plus watch allows consumers to programmatically recover from excessive backlogs. It achieves better performance and correctness by respecting the end-to-end argument. Finally, it uses standard rather than ad hoc storage APIs. Realizing the full potential of the model for pubsub use cases opens up exciting new research directions.

References

[1] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. Zdonik. 2003. Aurora: a data stream management system. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (San Diego, California) *(SIGMOD '03)*. Association for Computing Machinery, New York, NY, USA, 666. doi:10.1145/872757.872855

[2] Atul Adya, Robert Grandl, Daniel Myers, and Henry Qin. 2019. Fast key-value stores: An idea whose time has come and gone. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. 113–119.

[3] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, Arif Merchant, and Kfir Lev-Ari. 2016. Slicer: Auto-Sharding for Datacenter Applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 739–753. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/adya

[4] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. 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. *Proceedings of the VLDB Endowment* 8 (2015), 1792–1803.

[5] Apache Software Foundation. 2023. Apache Pulsar. https://pulsar.apache.org/. Accessed: 2025-01-14.

[6] Microsoft Azure. 2025. Overview of Service Bus dead-letter queues. https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-dead-letter-queues. Accessed 2025-01-15.

[7] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, Jingming Liu, Filip Gruszczynski, Xianan Zhang, Huy Hoang, Ahmed Yossef, Francois Richard, and Yee Jiun Song. 2020. Virtual consensus in delos. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 35, 16 pages.

[8] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C. Platt, James F. Terwilliger, and John Wernsing. 2014. Trill: a high-performance incremental query processor for diverse analytics. *Proc. VLDB Endow.* 8, 4 (Dec. 2014), 401–412. doi:10.14778/2735496.2735503

[9] Tarak Chandrasekaran, Neeraj Kumar, and Sujay Sanghi. 2011. OpenTSDB: A Distributed, Scalable Time Series Database. In *Proceedings of the 8th International Workshop on Distributed Data Management*. ACM, 83–90. doi:10.1145/2093334.2093338

[10] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2006. Bigtable: A distributed storage system for structured data. In *Proceedings of the*

7th USENIX Symposium on Operating Systems Design and Implementation (OSDI). USENIX Association, 205–218.

[11] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.

[12] Azure Documentation. 2025. Service Bus Messaging. https://learn.microsoft.com/en-us/azure/service-bus-messaging/. Accessed: 2025-01-15.

[13] Google Cloud Documentation. 2025. Pub/Sub documentation. https://cloud.google.com/pubsub/docs. Accessed: 2025-01-15.

[14] Google Cloud Documentation. 2025. Rapidly expand the reach of Spanner databases with read-only replicas and zero-downtime moves. https://cloud.google.com/blog/products/databases/introducing-spanner-configurable-read-only-replicas. Accessed: 2025-01-15.

[15] Google Cloud Documentation. 2025. Replay a message in Pub/Sub by seeking to a snapshot or timestamp. https://cloud.google.com/pubsub/docs/replay-message. Accessed: 2025-01-15.

[16] Google Cloud Documentation. 2025. Spanner: Change streams overview. https://cloud.google.com/spanner/docs/change-streams Accessed: 2025-01-15.

[17] Kubernetes Documentation. 2025. kube-apiserver. https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/ Accessed: 2025-01-15.

[18] etcd. 2025. etcd.io. https://etcd.io/. Accessed: 2025-01-14.

[19] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. 2003. The Many Faces of Publish/Subscribe. *ACM Computing Surveys (CSUR)* 35, 2 (2003), 114–131. doi:10.1145/857076.857078

[20] Gauri M Gaikwad, Amit Sahai, and Shikha K Sinha. 2016. A Survey of Publish/-Subscribe Systems and Their Key Challenges. *IEEE Communications Surveys & Tutorials* 18, 1 (2016), 55–75. doi:10.1109/COMST.2015.2494095

[21] Google APIs. 2024. Google Watch API. https://github.com/googleapis/googleapis/blob/master/google/watcher/v1/watch.proto. Accessed: 2025-01-14.

[22] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.

[23] InfluxData. 2023. InfluxDB: An Open-Source Time Series Database. https://www.influxdata.com/products/influxdb/. Accessed: 2025-01-14.

[24] Kafka. 2025. https://kafka.apache.org/documentation/.

[25] Martin Kleppmann. 2017. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media. https://www.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/

[26] E. Koutanov. 2020. *Effective Kafka: A Hands-on Guide to Building Robust and Scalable Event-driven Applications with Code Examples in Java*. Emil Koutanov. https://books.google.com/books?id=Rv3i0AEACAAJ

[27] Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, et al. 2021. Shard manager: A generic shard management framework for geo-distributed applications. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 553–569.

[28] Xuhao Luo, Shreesha G. Bhat, Jiyu Hu, Ramnatthan Alagappan, and Aishwarya Ganesan. 2024. LazyLog: A New Shared Log Abstraction for Low-Latency Applications. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 296–312. doi:10.1145/3694715.3695983

[29] Alexander Malmberg. 2025. https://sre.google/resources/practices-and-processes/no-heroes/. Accessed: 2025-01-15.

[30] MySQL Reference Manual. 2025. Replication with Global Transaction Identifiers. https://dev.mysql.com/doc/refman/8.4/en/replication-gtids.html Accessed: 2025-01-15.

[31] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1493–1509. doi:10.1145/3318464.3386134

[32] Andrew S. Tanenbaum and Maarten van Steen. 2007. *Distributed Systems: Principles and Paradigms*. Prentice Hall. https://www.pearson.com/us/higher-education/program/Tanenbaum-Distributed-Systems-Principles-and-Paradigms/PGM332264.html

[33] Amazon Team. 2025. What is Pub/Sub Messaging? https://aws.amazon.com/what-is/pub-sub-messaging/. [Accessed 04-14-2025].

[34] Google Team. 2025. What is Pub/Sub? https://cloud.google.com/pubsub/docs/overview. [Accessed 04-14-2025].

[35] TiDB Team. 2025. Time Synchronization in Distributed Systems: TiDB's Timestamp Oracle. https://www.pingcap.com/blog/how-an-open-source-distributed-newsql-database-delivers-time-services/ Accessed: 2025-01-14.

[36] Timescale. 2023. TimescaleDB: An Open-Source Time-Series Database. https://www.timescale.com/. Accessed: 2025-01-14.

[37] P.A. Tucker, D. Maier, T. Sheard, and L. Fegaras. 2003. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering* 15, 3 (2003), 555–568. doi:10.1109/TKDE.2003.1198390

[38] Antonio Varzi. 2006. Messaging Systems: An Overview. *Comput. Surveys* 38, 4 (2006), 1–25. doi:10.1145/1180404.1180405