ChatGPT

Timeline Framework PRD Update: Replay & Backfill Support for Analytics Queries

Introduction & Background

The Timeline Framework was originally designed for real-time processing of streaming event data. It ingests **structured logs** (JSON events with varying schemas and high-cardinality fields) and applies **Timeline operators** to produce analytic insights in near-real-time. However, the **majority use-case** has shifted towards retrospective analysis on historical data, where events are **not processed live** but need to be **replayed from durable storage** for analytics (e.g. business analysis, fraud detection). This PRD augments the existing Timeline design to support **backfill and replay workflows** orchestrated by Temporal, using Apache Iceberg on S3 for durable event storage and **VictoriaLogs** for efficient indexing and filtering.

Assumptions & Context:
- All incoming event logs are written to **Apache Iceberg** tables on Amazon S3 immediately upon ingestion (as the system's source-of-truth durable store). Iceberg provides reliable, high-performance storage for huge analytic datasets [1].
- **Temporal** (a workflow orchestration engine) coordinates *all* data movement and transformation workflows – including real-time ingestion, as well as on-demand backfills and replays. Temporal ensures reliability (with built-in retries, state checkpoints, and failure recovery) and idiomatic Go integration for our services.
- **VictoriaLogs** (from VictoriaMetrics) is optionally deployed as a log indexing service. It indexes all log fields (even those with **high cardinality** like user IDs or trace_ids) and supports full-text and field-based search on the logs [2]. We will use VictoriaLogs as an auxiliary system to quickly identify which subsets of data (which events/fields) are relevant for a given query, helping avoid expensive full-table scans on S3.
- **Real-time ingestion** of events continues to use the existing Temporal workflow + signals design (for low-latency processing of incoming events). The new addition is that **analytical queries** (often ad-hoc or scheduled) will trigger **replay/backfill workflows** that extract historical data from Iceberg for processing. These analytic queries might be driven by business needs (e.g. user behavior analysis, quality metrics) or fraud detection use cases, and they may cover large time ranges or complex filters that were impractical to maintain in real-time memory.

In summary, this update enables the Timeline system to run **batch analytics queries over historical event data** in an on-demand fashion, while leveraging our existing infrastructure: Iceberg as a scalable "data lake" table, Temporal for workflow orchestration (both streaming and batch), and VictoriaLogs for log indexing and querying.

Architecture for Replay & Backfill Workflows

To support replay of historical events, we introduce a **Replay Workflow architecture** that pulls data from the S3/Iceberg store and processes it through the Timeline Framework. The high-level architecture and data flow are as follows:

- **Durable Event Storage (Apache Iceberg on S3):** All events are written to Iceberg tables in near-real-time during ingestion. Iceberg acts as the central repository for *time-series event data*. Its table format (with Parquet files and rich metadata) allows efficient querying of large volumes of data with schema evolution and time-travel capabilities [3]. We partition the Iceberg table by time (e.g. date, hour) to prune scans for time-bound queries, and we leverage Iceberg's columnar storage to read only needed columns. This means even though logs are JSON with many fields, we can **project** only the relevant fields for a given query, reducing I/O.

- **Indexing & Query Filtering (VictoriaLogs):** VictoriaLogs indexes every log entry by its fields/tags, enabling fast search over attributes (it's optimized for wide events with many fields and handles high-cardinality fields out of the box [2]). For any given analytics query, the Temporal workflow can first query VictoriaLogs (via its LogsQL API) to **identify a subset of data of interest**. This can serve two purposes: (1) Determine which **fields and event types** are involved in the query, so we know what columns to extract from Iceberg; and (2) Potentially get a list of candidate event IDs or timestamps that satisfy the query filter, so we can target our Iceberg reads to relevant partitions or even specific files/rows. In practice, VictoriaLogs might return the matching log records (or their keys) for the query filter, which we then use to guide the Iceberg scan – effectively **avoiding a full table scan** if the filter is selective. *For example, if an analyst asks for "all rebuffer events on CDN1 in March 2025, device=Android," a VictoriaLogs query for those field values and time range can quickly find the matching events or at least the count and distribution. The replay workflow can then fetch exactly those events from S3.*

- **Temporal Orchestrator:** Temporal workflows coordinate the end-to-end process. We introduce a new **Replay Workflow** type (in Go) that handles a query request. The workflow will:

- **Receive Query Parameters:** (either via a direct start or a Temporal signal) specifying the event criteria (time range, event types/attributes filters, etc.).
- **Query VictoriaLogs (Activity):** If VictoriaLogs is available, the workflow invokes an activity task to run a LogsQL query for the given filters and time window. The result might be a set of matching event metadata (timestamps, IDs, etc.) or at least a refined filter (VictoriaLogs could tell which attributes are actually present). This guides the next step.
- **Fetch from Iceberg (Activity):** The workflow then launches one or more activities to **read from the Iceberg table** on S3. We leverage Iceberg's ability to do predicate pushdown (using query filters like event_type = "rebuffer", device="Android", timestamp in March 2025) and column projection (reading only the columns of interest). This activity might use an Iceberg-compatible engine or library to scan the table (for instance, using a Trino/Presto query, a Spark job, or a Go Iceberg reader library). The key is that **only the needed data is pulled**: thanks to partition pruning by date and fine-grained filtering by attributes, this read is much more efficient than scanning every log. *(If VictoriaLogs returned explicit pointers to log entries, the activity could even do direct lookups, but in general it will perform a filtered scan on the Iceberg table.)* The output of this step is the **set of events** (in JSON or structured form) that satisfy the query.

- **Process Events via Timeline Operators:** The retrieved events are then passed through the Timeline Framework's processing logic. This may involve classifying the events into known types, then applying any Timeline operators (such as aggregations, sequence analysis, etc.) that the query needs. In many cases, the "query" might simply be asking for the list of matching events or a count, which could be done directly in the Iceberg query or via VictoriaLogs. But for more complex analytics (e.g. computing the average buffering duration or correlating sequences of events), we utilize the same code that real-time ingestion uses – just operating on a batch of historical events. The events can be fed into the Timeline pipeline in order (sorted by time, possibly partitioned by some key like user session if needed).

- **Return or Store Results:** Finally, the workflow returns the result to the requester or stores it in a durable location (perhaps writing a result file to S3 or updating a dashboard). Since Temporal workflows in Go can return a result value upon completion, an synchronous API call could receive the result directly. For larger results or asynchronous usage, the workflow could signal back or the analyst could retrieve the output from a known location.

- **Workflow Checkpoints & Scaling:** Temporal ensures the above steps are fault-tolerant and scalable. The workflow can be designed to handle large data volumes by processing in **chunks** with durable checkpoints. For example, if the query spans a long time range (say a year of data), the workflow might split it into month-by-month or day-by-day sub-tasks: it could iterate through each partition (month) in sequence, or spawn parallel child workflows for each partition, then merge results. After each chunk, the workflow can record progress (e.g., "processed up to Feb 2025") in its state or via a Temporal **Continue-As-New** pattern. Using `ContinueAsNew` effectively checkpoints the workflow and starts a new run with the remaining work, preventing the workflow history from growing too large [4]. Temporal's event history allows it to replay from the last checkpoint in case of failure, ensuring we don't lose progress. Each activity (VictoriaLogs query or Iceberg scan) is also retried automatically by Temporal on failure, providing reliability for these external IO operations.

- **Concurrent Real-time and Replay Processing:** The architecture cleanly separates real-time ingestion from batch replay. **Real-time workflows** (driven by Temporal signals) continue ingesting new events as they arrive – writing them to Iceberg and updating any live metrics – while **Replay workflows** run on-demand for historical analysis. Because Iceberg is an ACID table format with snapshot isolation, we can have concurrent reads and writes without issue [3]. A replay workflow can either query a specific **snapshot** of the Iceberg table (for example, "data up to end of March 2025") to get a consistent view, or simply read the latest data if eventually consistent results are acceptable. There is minimal interference: ingestion writes new snapshots, and queries can be directed at a stable snapshot ID. If needed, we can coordinate by signaling the ingestion workflow to **flush** any in-memory batches to Iceberg before running a large analysis query, to ensure the latest events are included. Otherwise, analysts can specify a cutoff time for their query (e.g. "data up to last hour"). The Temporal orchestrator thus balances real-time vs. batch needs by isolating them in different workflows and leveraging Iceberg's design for simultaneous ingest & query [3].

Diagram (Architecture Overview): *The system integrates three layers – (1) S3/Iceberg as the data lake storage of events, (2) VictoriaLogs as an index for filtering query conditions, and (3) Temporal workflows as the orchestrator. An analyst's query triggers a Temporal workflow, which interacts with VictoriaLogs to narrow down relevant data, reads the needed events from Iceberg, and feeds them into the Timeline processing logic to produce results.* (See the end-to-end example below for a concrete illustration.)

Event Type Extraction & Classification Strategy

One challenge with arbitrary JSON logs is that they may contain many fields and various event types in a single stream. To **enable Timeline operators to work with these events cleanly**, we need a systematic way to **extract key fields and classify each event** into a known type (e.g. “Play”, “Seek”, “Rebuffer”, etc.), even if the underlying log schema is dynamic.

Key Strategies for Extraction and Classification:
- **Identify Event Type:** We assume each log record contains some indicator of its type. Common patterns include a field like `"eventType"` or `"action"` in the JSON payload (for example, `"eventType": "VIDEO_PLAY"` or `"action": "Seek"`), or a distinguishable field presence (e.g. if a log has a field `"bufferDuration"`, it might imply a Rebuffer event). The ingestion pipeline will use a **classification function** that inspects each JSON record and assigns it a **canonical event type label**. This could be as simple as reading a specific field (if the log format has one), or applying a set of rules (for instance, if `event.name == "playback.start"` then Type = Play; if `event.buffering == true` then Type = Rebuffer, etc.). New event types can be accommodated by updating this mapping or ruleset.

- **Extract Key Fields (Attributes):** For each recognized event type, we define which fields are **key attributes** relevant to timeline analysis. For example:
- *Play Start event:* extract fields like `video_id`, `start_time`, `cdn` (CDN provider), `device_type`, etc.
- *Seek event:* extract `seek_from_time`, `seek_to_time`, maybe `seek_duration` if available.
- *Rebuffer (Buffering) event:* extract `buffer_duration`, `buffer_reason`, `cdn`, `device_type`, etc.
- *Generic/Other events:* we might not extract many details, or we handle them as unclassified if they’re not important to current analysis.

The extraction can be implemented using JSON parsing libraries in Go (e.g. decoding into a map or struct). The **VictoriaLogs index** is helpful here as well: it automatically flattens nested JSON and indexes all fields [5] [6]. We can leverage this by querying for specific fields. For instance, if we want to classify events by `cdn`, VictoriaLogs can quickly tell us all values of `cdn` present and help filter by `cdn=CDN1`. Moreover, for logs with nested structure, VictoriaLogs flattens them, making it easier to refer to fields uniformly. *(We will ensure our Iceberg table schema or processing logic similarly can handle nested JSON – likely by storing logs as a struct or map type in Iceberg so that fields can be accessed by name.)*

- **Schema and Iceberg Table Design:** The Iceberg table will be designed to accommodate these dynamic logs. We will use a **flexible schema** with either:
- A few universal columns (timestamp, event_type, possibly a top-level `attributes` map for all other fields), or

- A wide schema that includes the most important fields as columns and a generic JSON blob for the rest.
  Using a **map of string→value** for attributes is a convenient way to store arbitrary key-value pairs in Iceberg. This way, new fields do not require an immediate schema change – they can be added into the map. Over time, if certain fields become important to many queries, we can evolve the Iceberg schema to add them as explicit columns (Iceberg’s schema evolution allows adding new columns

without breaking old data [3]). This strategy means our pipeline can parse out known keys but still retain unknown ones for future use.

- **Classification Implementation:** We propose implementing an **EventClassifier** component (as a Go function or set of functions) that takes a raw JSON log (or a generic map of fields) and returns a `(eventType, EventDataStruct)` pair. Under the hood, this might use a simple if/else or map lookup on a `"type"` field if present, or examine certain fields to determine the type. The `EventDataStruct` could be a struct tailored to that type or a generic container of key fields. For example:

```
type Event struct {
    Timestamp time.Time
    Type      string
    Attributes map[string]interface{}
}
```

We might then define specialized structs or simply use the Attributes map for specific keys. If using specialized structs, the classifier could populate a `PlayEvent{VideoID, CDN, Device, ...}` struct for Type "Play", a `RebufferEvent{Duration, CDN, Device, ...}` for "Rebuffer", etc. These could implement a common interface so that Timeline operators can accept an `EventInterface` and then type-assert if needed for specific handling.

Design Patterns for Dynamic Event Types: We consider two approaches to modeling events: 1. **Strongly-Typed Events with Inheritance/Interfaces:** Define Go structs for each known event type (PlayEvent, SeekEvent, BufferEvent, etc.), all implementing a common interface (say `TimelineEvent` with methods like `GetTimestamp()` and maybe type-specific getters). The classifier will instantiate the appropriate struct based on the log content. This gives us compile-time safety and clear definition of fields. Timeline operators can directly work with these types – for example, an operator that calculates rebuffer percentages can specifically expect `RebufferEvent` inputs to get the `Duration` field. The downside is that introducing a brand-new event type requires code changes (a new struct and logic in the classifier). However, unknown types could be mapped to a generic `UnknownEvent` struct with just the Attributes map, to avoid losing data.
2. **Generic Event Model with Flexible Attributes:** Use a single `Event` struct as shown above with a map of attributes. The `Type` field distinguishes the kind of event, and the Attributes contain all other data. Timeline operators would then be written in a more generic way: e.g. an operator can filter events by `event.Type == "Rebuffer"` and then look up `"buffer_duration"` in the Attributes map. This is very flexible – new event types automatically flow through the system – but it's more error-prone at usage time (typos in attribute keys, lack of auto-completion, etc.). Performance might also be slightly lower due to dynamic type handling.

We propose a **hybrid approach**: define a core set of important event types as first-class structs (for clarity and performance in known analyses), and treat less common/unexpected events with a generic representation. This way, Timeline operators that focus on key events get the convenience of typed fields, and the system can still accommodate dynamic/unmodeled events without dropping them. We will also maintain a **registry of event type definitions** – essentially metadata that lists known event types and how

to parse their JSON (which keys to extract, what they mean). This registry can be updated as new event types emerge, without affecting the rest of the pipeline.

- **Example Classification:** Suppose a raw JSON log entry looks like:

```
{
  "timestamp": "2025-03-15T10:00:00Z",
  "event": "VIDEO_REBUFFER",
  "session_id": "ABC123",
  "cdn": "CDN1",
  "device": "Android",
  "buffer_duration_ms": 5000,
  "reason": "network"
}
```

  The classifier would map `"event":"VIDEO_REBUFFER"` to event type = **Rebuffer**. It would extract `buffer_duration_ms` (5000 ms) and perhaps convert to a standardized field (e.g. `Duration=5s`), along with common fields `cdn="CDN1"`, `device="Android"`, `session_id`, etc., into a `RebufferEvent` struct. If another log entry had `"event": "PLAY"` or lacked an explicit type but had, say, a `"position":0` and `"action":"start"`, we'd classify that as a **Play** event, capturing `video_id`, `device`, etc., accordingly. All events get a `Timestamp` field normalized to a standard time object. These clean event objects are then ready for Timeline analysis.

By implementing this structured classification layer, we turn arbitrary JSON blobs into a stream of **typed events** with well-defined fields. This greatly simplifies writing Timeline operators, since they can assume a consistent model (rather than dealing with raw JSON each time). It also means that replayed historical data and real-time data will follow the **same event model**, allowing reuse of the analysis code for both live and batch scenarios.

Modeling Dynamic Event Types for Timeline Operators

To ensure Timeline operators can handle dynamic event types (which may evolve over time), we need design patterns that keep the system flexible and maintainable. Here are the proposed patterns in more detail, focusing on how events are modeled in Go and used in Temporal workflows:

- **Polymorphic Event Objects:** Define an interface `TimelineEvent` with methods like `Type() string` and perhaps type-specific ones (`GetAttribute(key)` or defined getters for certain common fields). Each event type struct implements this. For example:

```
type TimelineEvent interface {
    Timestamp() time.Time
    Type() string
}
type PlayEvent struct { /* fields */ }
func (p PlayEvent) Type() string { return "Play" }
```

```
type RebufferEvent struct { /* fields */ }
func (r RebufferEvent) Type() string { return "Rebuffer" }
// etc.
```

The Timeline processing pipeline can then operate on `[]TimelineEvent` generically. Where needed, it can switch on `event.Type()` or use type assertions to access specific struct fields. This pattern keeps event handling code organized by type and leverages Go's type system. Adding a new event type requires adding a struct and implementing the interface, but does not require changing the interface itself or the core pipeline (it will just treat the new type as `TimelineEvent`).

- **Attribute Map + Helper Accessors:** Alternatively (or additionally), each event could carry a `Attributes map[string]interface{}` for all its data (even if we also have struct fields). Timeline operators that are written generically can use this map to retrieve fields without needing to know the concrete type. We can provide utility functions or methods to fetch common fields. For instance, a function `GetDevice(e TimelineEvent) string` could try `if e has Device field, return it; else if in Attributes map, get "device"`. This way, operators can be written in a generic style, which is useful for ad-hoc queries that might involve fields not originally anticipated.

- **Dynamic Event Registration:** We maintain a mapping from event type name to a handler or parser function. For example, a map `map[string]func(map[string]interface{}) TimelineEvent` that knows how to construct the appropriate struct from a generic JSON map. This makes the classifier extensible – new event types can be added by registering a new parse function (which could even happen at startup via configuration, rather than hard-coding every type). If an unknown event type is encountered at runtime and not in the registry, the system can default to a generic `GenericEvent` type (with all data in Attributes) so that it's not lost.

- **Schema Evolution Handling:** Because event types and schemas can evolve (new fields added, old fields deprecated), our design isolates those changes. The parsing logic for each event type can ignore fields it doesn't recognize or store them in the Attributes map. When new fields are added to an event (and consequently to the Iceberg schema or just appear in JSON), we update the corresponding struct and parser to extract it. Thanks to Apache Iceberg's schema evolution features, we can add new columns for these fields without rewriting old data [3]. The Timeline operators should ideally be resilient to missing fields (e.g. if an operator expects a field that older events didn't have, it should handle default values). We will ensure to **test with partial schema evolution** (see Testing Strategy) to guarantee that adding a new field doesn't break processing of old events and vice versa.

- **Idiomatic Go Considerations:** The event modeling will follow Go best practices: simple data structures, avoid reflection where possible, and keep code easy to understand. Defining explicit structs for known events aligns with an idiomatic approach (clear types and fields), while using maps for extensibility provides flexibility. We will document the expected keys and types in the Attributes map for each event type, serving as a loose schema. This approach balances type safety with the dynamic nature of log data.

By adopting these modeling patterns, the Timeline Framework can **cleanly support dynamic event types**. Timeline operators (which are essentially functions or components that compute metrics or detect patterns over event sequences) can be written against a stable interface. For example, an operator that calculates "total rebuffer time per session" can assume events implement `TimelineEvent` and simply do: filter for events where `Type()=="Rebuffer"`, then sum their `Duration` field (retrieved via a type assertion or attribute lookup). Even if new kinds of events appear (say a "Stall" event), if they are classified as a type and have a similar duration attribute, the operator might even handle them automatically if configured to include that type. If not, adding support is straightforward. This extensibility is crucial as the log schemas are "unstructured structured" data that will grow and change.

Temporal Workflow Design for Backfill & Replay

Temporal is central to coordinating both ingestion and replay workflows. We outline how Temporal workflows are designed to manage **long-running replay queries with checkpoints**, partial backfills, and interaction with external systems (Iceberg and VictoriaLogs), all while adhering to idiomatic usage of Temporal in Go.

1. Real-Time Ingestion Workflow (Recap):
For context, the existing real-time ingestion likely uses a long-running Temporal Workflow that receives events via **signals**. Each time new log events arrive (from a streaming source or message queue), a Temporal signal wakes the workflow, which then processes those events (parses, classifies, updates some in-memory state or writes to a database, etc.). This workflow can loop indefinitely, using `Workflow.await()` or a Selector to wait for signals, and periodically calls activities to flush data to S3/Iceberg. To avoid unbounded growth of the workflow history (since it's never-ending), it uses `ContinueAsNew` after a certain number of events or time interval [4] – essentially creating a new workflow execution that picks up where the last left off, carrying forward any necessary state (like an offset or pending events). This design is **idiomatic Temporal** for streaming ingestion: it provides fault tolerance (Temporal will replay signals in case of failure) and controlled resource usage (the workflow is passive when waiting, consuming no CPU [7]).

Real-time signals example: A data collection process (could be an external scheduler or another workflow) detects new data and sends a signal to the ingestion workflow to start processing [8]. The ingestion workflow fetches and ingests the new events, then perhaps goes back to waiting. This pattern keeps the ingestion responsive to live data while still being resilient (Temporal will persist the signals and state in its journal).

2. Replay/Backfill Workflow:
For historical queries, we introduce a new Temporal Workflow, say `ReplayWorkflow`, which is *started on-demand* with parameters describing the query (time range, event filters, etc.). This could be triggered by an API call, CLI command, or even a Temporal Schedule (Temporal's scheduling feature can also allow manual triggering or cron-based triggers, which support backfill runs on past intervals [9]). The ReplayWorkflow is a *short-lived workflow* (relative to the endless ingestion workflow) – it runs to completion for one query, then stops (or could continue-as-new for extremely large queries).

Key aspects of the ReplayWorkflow design:
- **Execution Plan:** Upon start, the workflow may plan how to divide the work. For instance, if asked to

process a large date range, it could split it by month or by some logical key to enable parallel or sequential chunk processing. We can implement this as a loop inside the workflow, or by spawning **child workflows** for each chunk. An advantage of child workflows is that each can run in parallel and has its own retry logic, and the parent can aggregate results. However, if the analysis requires considering the entire dataset as one (e.g. for chronological ordering), we might keep it sequential. We will likely choose a time-based chunking for simplicity, because queries commonly are constrained by time. Partial backfill by attribute (if the query only wants a subset of attributes) doesn't necessarily need chunking by attribute – the filtering is handled by the query itself.

- **Temporal Activities for External IO:** The ReplayWorkflow will make use of **activities** to perform heavy-lifting steps: contacting VictoriaLogs, reading from Iceberg, and even processing data if it's CPU-intensive. In Temporal, activities are the units that actually execute external calls and can be retried; the workflow code itself should remain lightweight (no heavy loops over millions of events, for example). So we will implement:
- `QueryVictoriaLogsActivity(queryParams) -> filterResult`
- `ReadIcebergActivity(queryParams/ filterResult) -> eventsChunk`
- Possibly a `ProcessEventsActivity(eventsChunk) -> partialResult` if processing is intensive (though if it's just filtering and formatting, the workflow might handle it in memory).
  Each activity can have customized retry policies (exponential backoff, etc.) to handle transient failures like network issues to S3 or the VictoriaMetrics service. Temporal guarantees *at-least-once* execution of each activity (with idempotence considerations if we code them carefully).

- **Durable Checkpoints & Continue-As-New:** If the workflow processes multiple chunks sequentially, after each chunk it can record progress. This could be done simply by keeping an index in workflow state (which is preserved across retries), or by writing a checkpoint to an external store (not usually necessary unless we want recovery even if the whole workflow fails and we restart manually). Temporal's own state (event history) acts as a durable log of which steps completed. If the amount of history becomes large (e.g., thousands of events, or very large data passed through), using `ContinueAsNew` is prudent. For example, after processing 3 months of data, we could have the workflow re-start itself with parameters updated to the next 3 months. The new execution will have a fresh history, but we can carry over any cumulative results or pointers via workflow inputs or via writing interim results to a temporary location. This technique is recommended by Temporal for long-running workflows [4] – it avoids hitting the history size limits and keeps performance optimal.

- **Partial Backfill by Time or Attribute:** Our design supports partial reprocessing in multiple ways:

- **By Time Range:** The query parameters explicitly include time bounds. Analysts can request, say, Jan–Mar 2025 specifically. The workflow will only scan Iceberg for those partitions and only retrieve events in that range. This avoids any need to process data outside the interest window. If later an analysis is needed for a different time, a new workflow is run for that period.
- **By Attribute (Selective Fields):** If an analysis only concerns certain attributes or event types, we effectively do a "partial backfill" by filtering. For instance, a fraud detection query might only care about login events with a certain property – we don't need to ingest all other event types. This is achieved by **filter conditions** in the query, which propagate down to the data read. Using VictoriaLogs here is key: it can instantly narrow down which events match the attribute criteria, so the Iceberg read is inherently partial. Even without VictoriaLogs, we could push attribute predicates

to Iceberg (e.g., using Iceberg's support for filtering on data columns). However, since our logs are high-cardinality and not all fields are partition keys, Iceberg alone might still have to scan many files. With an index (VictoriaLogs), we turn an attribute filter into an efficient lookup.

- **Replay vs. Full Recompute:** Sometimes a "partial backfill" might mean reprocessing only data that was missed or that needs re-computation (for example, if we introduced a new metric and want to compute it for past events). Temporal can coordinate this by either allowing the query to specify "only process events where this metric is missing" or by maintaining markers. But that is more a data completeness backfill scenario. In our context, partial backfill is mostly about focusing on relevant subsets to save time.

- The system can handle multiple backfill workflows in parallel if, say, two different analytics queries are triggered by different teams. Each will operate on the data lake independently; the only contention might be on reading from S3, which Iceberg's table services handle with concurrent readers.

- **Balancing Real-Time and Batch Needs:** As noted, the real-time ingestion and batch replays run concurrently but separately. One area to consider is **deduplication or double-processing**: If we run a replay for "March 2025" in April 2025, those March events were already ingested in real-time when they happened. Are we reprocessing them twice? In this design, yes – but for different purposes. The real-time pipeline likely produced some immediate metrics or triggered alerts at that time, whereas the replay is for an offline query (perhaps computing something not maintained in real-time). We accept that duplication because the contexts are separate. However, we ensure that **replay workflows do not modify the source data** or interfere with real-time state. They read from the immutable log store and output results elsewhere. If an analyst wants to combine real-time and historical results, they can either run a query that spans into recent data or we can merge the results of a replay with whatever current metrics exist. The architecture could even allow a "hybrid" approach: e.g. an analyst query for "last 7 days including today" might run a backfill for the first 6 days from Iceberg and also query the live system for today's in-memory data. Designing that coordination is complex, so initially, we treat all queries as pure historical or up-to-now (assuming data is flushed quickly to Iceberg, so even "today" can be read from Iceberg after minor delay).

- **Ensuring Idiomatic Temporal Usage:** We keep workflow code simple (no heavy computation, just orchestration and decision logic). All long-running or CPU-heavy tasks are in activities. We utilize **Temporal's Go SDK features** like the testing framework, context propagation, and careful structuring of workflow vs activity code. We avoid using global state or non-deterministic logic in workflows (Temporal requires workflows to be deterministic in replay). For example, our workflow will not iterate over events in a non-deterministic order; it will use the query results (which are fixed) and process them in a defined sequence. If random or time-based decisions are needed, we use Temporal APIs for those. The design leverages Go's strength in concurrency by possibly launching multiple activities in parallel (e.g., parallel reads for different partitions) but using `workflow.Await` or futures to gather results in order, which is a typical pattern. By following these practices, the system remains **idiomatic Go** (clear struct definitions, interfaces, type safety) and **idiomatic Temporal** (clear separation of workflow vs activity, using signals, continues, and child workflows appropriately).

End-to-End Query Flow (Analyst Trigger to Result)

Let's walk through a concrete example of a replay-based Timeline query to illustrate how Iceberg, VictoriaLogs, and Temporal interact end-to-end. We'll use the example query mentioned: *"Find all Rebuffering events that occurred on CDN1 during March 2025 where device_type = Android."* We want to retrieve these events and perhaps calculate some aggregate like total rebuffer time or simply list them on a timeline.

Scenario: In April 2025, a video streaming quality analyst wants to analyze buffering events in March for Android users on CDN1, to assess if that CDN had performance issues.

Steps in the End-to-End Workflow:

1. **Analyst Triggers the Query:** The analyst uses an internal tool (could be a CLI or a web UI) to specify the query filters (EventType = "Rebuffer" (i.e., video buffering events), CDN = "CDN1", Device = "Android", Date range = 2025-03-01 to 2025-03-31). When they submit this query, the tool behind the scenes uses the Temporal client to **start the ReplayWorkflow** with these parameters. For example, it might call `workflowOptions := StartWorkflowOptions{ID: "replay-March2025-analyst1", ...}; temporalClient.ExecuteWorkflow(ctx, workflowOptions, ReplayWorkflow, queryParams)`. Temporal will create a new workflow execution for this query. The analyst is informed that the query is running (maybe with an ID to check status, etc.).

2. **VictoriaLogs Filtering (Activity):** The ReplayWorkflow's first step is to query the VictoriaLogs index to narrow down relevant data. It runs an activity, say `QueryVictoriaActivity`, which uses the LogsQL query interface of VictoriaLogs. The query could look like:

   ```
   {type="Rebuffer", cdn="CDN1", device="Android"}  and timestamp in
   [2025-03-01, 2025-03-31]
   ```

   (The exact syntax depends on VictoriaLogs' query language; it might be something like `eventType="VIDEO_REBUFFER" and cdn="CDN1" and device="Android" and time >= "2025-03-01" and time < "2025-04-01"`.)
   VictoriaLogs will use its full-text index to quickly find log entries matching those conditions [2]. Suppose there were 50,000 such events in March 2025 out of millions of total events. VictoriaLogs returns a result which might include the list of matching log records or perhaps just some metadata. Likely, we will retrieve at least the timestamps (and maybe an identifier or the log content). If the result set is huge, we might page through it or use VictoriaLogs just to get counts per day, etc. For simplicity, assume we get either all matching log lines (50k JSON objects) or references that we can use to fetch them.

Alternate path: If VictoriaLogs is not available or not used (it's optional), the workflow would skip this and directly query Iceberg. In that case, it would rely on Iceberg's own filtering: we'd construct a filter expression for Iceberg (via SQL or API) like `WHERE event_type = 'Rebuffer' AND cdn = 'CDN1' AND device = 'Android' AND date BETWEEN '2025-03-01' AND '2025-03-31'`. Iceberg's query engine would then scan the March 2025

partition(s) and apply this filter. This is less efficient if those fields aren't partition columns, but Iceberg will at least only read March data. The process below from step 3 onward would be similar, just without the pre-filter hint from Victoria.

1. **Reading from Iceberg (Activity):** Next, the workflow calls `ReadIcebergActivity` with the query parameters and potentially hints from VictoriaLogs (like list of file paths or list of exact timestamps to retrieve). This activity is executed by a worker that has access to the data lake. Here's how it might operate:
2. It initializes an Iceberg table reader (e.g., using the Iceberg Go API or by spawning a small Spark job if needed – but ideally we keep it within Go for simplicity).
3. It applies the filter: only read partitions within March 2025. (Our Iceberg table is partitioned by date, e.g., `partition by year, month, day`, so it will skip other months entirely.) It also applies column projection: we know we need at least `timestamp`, `cdn`, `device`, `buffer_duration`, maybe `session_id` and any other fields relevant to Rebuffer events. The reader will only fetch those columns from the Parquet files.
4. If VictoriaLogs provided an exact list of matching event primary keys or something similar, the reader could even do a more directed fetch. If not, it will scan the March data and internally filter each record by the attribute conditions. Because the data is columnar, reading only `cdn`, `device`, and `eventType` columns first can quickly eliminate non-matching rows without reading the entire JSON. We might also leverage Iceberg's **metadata tables or indexes** if available – for example, Iceberg can maintain column stats and value ranges per file; if it knows a file has no "Android" device events (by reading min/max stats of the device column), it will skip that file entirely (this is called data skipping).

5. The output of this activity is the collection of events that match – likely as a slice of JSON records or already parsed event objects. If the result set is very large (50k events might be fine to handle in memory; if it were millions, we'd consider streaming or chunking at this stage), we might split this into multiple batches. For example, the activity could stream results and the workflow handles them incrementally, but Temporal activities usually return after completion, so better would be to chunk at source: e.g. process one day at a time in a loop of activities. In our example, perhaps 31 daily activities each returning up to whatever events. However, to keep it simpler, assume it's manageable in one go or a few goes.

6. **Processing & Timeline Operators:** Now with the raw historical events in hand, the workflow (or a subsequent activity) processes them. This involves:

7. Parsing each JSON log record to our internal event structs (if not already done in the Iceberg read step – we might directly deserialize Parquet to struct if using an Iceberg library that maps to Go structs). We run the **EventClassifier** on each to classify type and extract fields. In this case, nearly all events will classify as "Rebuffer" type (since that's what we asked for). But it's possible that within "Rebuffer" events there are subtypes or different schemas (maybe some logs have `buffer_duration_ms` and others use a different field). The classifier handles those differences and outputs a consistent `RebufferEvent(duration, cdn, device, timestamp, etc.)`.
8. We then feed the events into the relevant Timeline operator or analysis logic. If the query is simply "give me all these events," the operator might just sort them by time and format them for output. If the query is more analytical – say the analyst actually wants "the total buffering time in March for CDN1 on Android" – then an operator would aggregate the `duration` field of all events. Our

design allows either: the query could specify an aggregation to perform, or the analyst could just retrieve raw events and do analysis externally.

9. Let's say we want to produce a **timeline of rebuffer events**. We could group events by day or by video session. For example, an operator might create a timeline visualization data: each event becomes a point on a time-series (with timestamp and duration). Or perhaps we detect the number of rebuffer occurrences per day. For illustration, imagine the result we want is: "For each day of March 2025, how many rebuffer events on CDN1 from Android devices, and total buffering time." The operator would iterate over the events, bucket them by day, and sum counts and durations. It would produce an output like:

    - 2025-03-01: 1,200 rebuffer events, 3.5 hours total buffer time
    - 2025-03-02: 1,050 events, 3.0 hours
    - … (and so on for each day)
      This is just one example of a metric; the framework could support many analyses. The key is that by reusing the same event parsing and handling code, we ensure consistency with any real-time calculations (if, say, real-time was also tracking daily rebuffer counts, we'd get the same numbers).

10. **Result Delivery:** The final step is delivering results back to the analyst. Temporal's workflow completes and can return a result object (for instance, an array of daily stats or a file reference). Given the volume of data might be large, another approach is writing the results to a durable store: e.g., the workflow could call an activity to save the result to an Iceberg *results* table or to a CSV in S3, and then return a pointer (like "your results are in s3://…/rebuffer_report_Mar2025.csv"). For interactive use, if the result is small enough, the client can just get it directly when the workflow finishes. In our example, suppose the output is the daily summary table – that's small and can be returned directly. If the query was returning all 50k raw events, we'd likely not send that over Temporal's complete result (since that goes through the Temporal server). Instead, we'd have an activity write those 50k events as a JSON file in S3 or index them into a results database, and provide the location.

11. **Analyst Consumes Results:** The analyst can then see the outcome. Perhaps our system integrates with a Jupyter notebook or BI tool; or if it's CLI-based, it might automatically download the result file. The analyst sees that CDN1 on Android had a spike of buffering on a particular date, etc., and can make business decisions or further queries. If further drilling is needed (say, "show me examples of sessions that had many rebuffer events on March 15"), they could trigger another query (e.g., filter by that date and maybe session ID).

Throughout this process, **Temporal orchestrates each step reliably**. If any step fails (say the network to S3 glitches during reading), Temporal will retry that activity (according to a retry policy). If a failure is non-recoverable (bug in code, etc.), the workflow can fail gracefully, and the analyst is notified of the error (with logs accessible). We also have the ability to track the workflow's progress via Temporal query APIs or just by logging and Temporal's UI – for example, the workflow could periodically report "Reading data for day X done, Y events processed" via `workflow.GetLogger().Info` and custom workflow search attributes for progress (Temporal allows attaching application metadata to workflows for visibility). This way, long queries aren't a black box; we can monitor them.

Performance considerations: In this example, 50k events is easily handled. If a query resulted in tens of millions of events, we would further refine our approach – perhaps by using the index to pre-aggregate or by having the Iceberg query itself perform some aggregation (like a SQL count) rather than fetching raw data. We could have a mode where simple count or sum queries are answered by pushdown to the database rather than full replay. The architecture is flexible to incorporate that (for instance, the VictoriaLogs query itself could compute count of matches, since log databases often can count events matching a filter quickly). But the focus here is enabling *full fidelity* replays when needed, through a scalable workflow.

This end-to-end example demonstrates how **Temporal, Iceberg, and VictoriaLogs interact**: VictoriaLogs narrows the search space and identifies relevant fields/values; Iceberg provides the persisted source data with efficient scanning and projection; and Temporal ties it together, ensuring the process is reliable, repeatable, and integrated with our Timeline processing code. The result is that analysts can query historical timelines with the same level of detail and confidence as the real-time system, without needing a separate big data platform – it's all within the Timeline Framework orchestrated by Temporal.

Testing Strategy for Replay & Query Workflows

Supporting these new replay and query capabilities adds complexity to the system. We will implement a comprehensive **testing strategy** covering unit tests, integration tests, and scenario simulations to ensure the system behaves correctly – including edge cases like schema changes and high-cardinality filters. Below are the key testing approaches:

- **Unit Testing with Temporal's Framework:** We leverage Temporal's Go SDK testing framework to write unit tests for workflow and activity logic. Temporal provides an in-memory test environment (`WorkflowTestSuite`) that can simulate workflow execution, including timing, signals, and activity results [10]. We will write unit tests for:
- The ReplayWorkflow's control logic (e.g., given certain query params, it correctly splits into N chunks, calls the expected activities in order, and aggregates results). In tests, we will **mock** the activities: for example, have `QueryVictoriaActivity` return a pre-canned set of "matching events" for a known small input, and ensure the workflow then calls `ReadIcebergActivity` with appropriate filters. We can assert that the workflow output matches expected aggregation.
- The EventClassifier and parsing logic: feed sample JSON logs to the classifier and verify it produces the correct `TimelineEvent` struct with the right type and fields. We will test various shapes of JSON, including ones with missing fields, extra fields, or unknown event types (to ensure unknowns are handled gracefully).

- Timeline operator functions: test that given a list of event objects (possibly manually constructed), the operators compute the correct results (e.g., the daily rebuffer calculation returns correct sums). These can be pure unit tests without involving Temporal at all, just normal Go tests of the functions, since the operators should be written as library code used by the workflow.

- **Integration Testing with Mini-Services:** For integration tests, we want to exercise the real interactions between components:

- **Iceberg Integration:** We can set up a local or in-memory Iceberg environment (for example, using a local file system as the "S3" and an Iceberg table metadata). Apache Iceberg has a variety of

bindings; we might use a small Parquet file to simulate the dataset. In tests, we could preload an Iceberg table with a small sample of event data (maybe 100 events with known distribution of attributes). Then run the `ReadIcebergActivity` against it (perhaps using an Iceberg Go client or by querying it via a small Spark embedded setup). The goal is to verify that the filtering works – e.g., if we query for device=Android, we only get those events. This tests our query predicate construction and ensures the data access layer is correct. If a direct Iceberg read in Go is too complex for testing, we could mock the Iceberg call in unit tests and rely on a separate integration test with a real engine to validate queries. For instance, use an **Athena** or **Trino** in a test environment to run an equivalent SQL on a Parquet file.

- **VictoriaLogs Integration:** Similar approach – possibly run a lightweight instance of VictoriaLogs (it's open source, can run locally) with some log data ingested, and then test that our `QueryVictoriaActivity` can connect and retrieve expected results. Alternatively, since VictoriaLogs supports a simple API, we might mock the HTTP responses for known queries to simulate its behavior. The focus is to ensure our parsing of its responses and error handling are solid.

- **End-to-End Workflow Integration Test:** Using Temporal's test suite, we can simulate an entire workflow run without external dependencies by **mocking the external interactions**. For example, in a test we set `QueryVictoriaActivity` to return a known set of event keys, and set `ReadIcebergActivity` to return a known list of fake events (we fabricate some `RebufferEvent{...}` items). The workflow then processes them, and we check the final result. This verifies the orchestration logic including loops, continues, etc. The Temporal test environment also allows simulating time progression, so we could test a scenario with Continue-As-New by, say, setting a small threshold for iteration and asserting that the workflow indeed calls ContinueAsNew after processing the first chunk.

- **Simulating Schema Evolution:** We will write tests to simulate the addition of new fields and event types:

- For example, initially process an event with a certain schema (say, PlayEvent with fields A, B). Then simulate that a new version of the log adds a field C. We update the classifier to extract C if present. In a test, we feed an old log (without C) and a new log (with C) through the pipeline and ensure that:
    - The old log is still processed (C might be missing but that should not cause errors; the new classifier code might just not find it and default it).
    - The new log yields the additional data in the output.

- Similarly, add a completely new event type JSON that our system didn't originally know. Initially, the classifier will classify it as UnknownEvent; we then simulate updating the system (as if we deployed a new version with the new type support) and ensure it then classifies properly. We can even simulate running a backfill before and after adding support to see that previously unrecognized events can be reclassified by replaying with the new logic. This is a key benefit of having replay: if today we don't measure something but tomorrow we realize it's important, we can backfill the historical data to compute it retroactively, once we add the parsing logic. Tests will confirm this works as expected. Iceberg's schema evolution (adding new column for that field) can be tested by updating the table schema between runs and verifying old data reads still succeed [3]. (In unit tests we might not alter a real Iceberg schema, but we can mimic the scenario by reading JSON with and without the field using our code paths.)

- **High-Cardinality Filtering Tests:** We want to ensure that queries on attributes with very high cardinality (e.g. an attribute like `user_id` that's nearly unique per event) are handled. Two aspects to test:

- **Performance/Index Behavior:** If we have the VictoriaLogs available, such a query (e.g. "find events for user_id = X") should still run quickly because VictoriaLogs indexes all fields. We can test that our `QueryVictoriaActivity` correctly forms the query and returns only events for user X, even if in the test dataset each user appears once (worst-case cardinality). This ensures that we do rely on the index and not accidentally scan everything. We might simulate a scenario with 100 unique user_ids in 100 events and query for one user; the expected result is 1 event.

- **Fallback without Index:** If running without the index, a high-cardinality filter essentially degenerates to scanning everything (since no selective index). We should test that path too: e.g., 100 events each with unique user, query for one user; our Iceberg scan (or our filtering logic) should still return the correct single event. Performance in a unit test is fine (100 events); in production, this would be slow for millions, but that's exactly why we have VictoriaLogs. We might include a safeguard in the code: if a filter is likely high-cardinality (like user_id unless combined with something else), maybe warn or require the index to be present. However, that's more of a product decision. At minimum, we ensure correctness: the system must return the right results even in worst-case filters, though it might be slow if the index isn't there.

- **Load and Stress Testing:** Before deployment, we will conduct stress tests where we run the replay workflows on large volumes (perhaps using a subset of real data or synthesized data). We can simulate a backfill of a full month and measure memory and time. Temporal's visibility into workflow execution will help identify any bottlenecks (e.g., if our single workflow approach is too slow, we might adjust to use more parallelism). We will test multiple concurrent workflows to ensure the Temporal worker pool can handle it (Temporal scales horizontally by workers; we can allocate enough worker processes to handle heavy read tasks in parallel). We will also test failure recovery: intentionally kill a worker in the middle of a read to see that Temporal retries the activity on another worker, etc., confirming our idempotence assumptions (for example, if an Iceberg read was partially done, we need to ensure either it can resume or the partial results are discarded and retried safely – likely the latter, meaning the activity should be idempotent or guarded).

- **Temporal Workflow Testing Tools:** We use features like **time skipping** in Temporal's test suite for any time-based logic. For example, if we had a TTL for queries or a timeout mechanism in the workflow, we could simulate advancing the Temporal test clock to test those branches. We also use the `AssertExpectations` in the test suite to ensure our mocked activities were called exactly as expected [11], verifying the control flow. The Temporal test framework will help catch any non-deterministic code in the workflow (since it'll panic if we do something invalid like random numbers in workflow), so by writing tests we can flush out determinism issues early.

- **Go Idiomatic Testing:** We will incorporate standard Go testing patterns too (using `testing` package and possibly `testify` for assertions). Our code will be organized so that much of the logic (parsing, classification, operators) is pure functions that are easy to test without needing a Temporal environment. Temporal-specific code (the workflow struct and methods) will be minimal glue that we can test with their framework. This separation of concerns improves testability and maintainability.

In summary, our testing strategy ensures that the new replay/backfill capabilities work correctly and robustly. We cover from unit level (ensuring one function does what it should) up to end-to-end scenarios (ensuring an analyst query yields the correct outcome through the whole system). We will particularly focus on tricky areas like evolving schemas (to not break on new log fields) and performance of filtering (to confirm the VictoriaLogs integration truly restricts data scanning). By using Temporal's testing tools and thorough integration tests, we will achieve high confidence in the system before rolling it out.

Conclusion

This PRD update outlines how the Timeline Framework will support *analytics over historical event data* by introducing replay and backfill workflows. The architecture leverages our existing stack in a cohesive way: **Apache Iceberg** provides a strong foundation for storing and querying large-scale event logs (with features like concurrent ingestion+query, time travel, and schema evolution [3] critical for our use case), **VictoriaLogs** augments it by indexing the semi-structured logs for rapid filtering (crucial for handling high-cardinality and arbitrary attributes [2]), and **Temporal** orchestrates the complex processes reliably (ensuring exactly-once processing semantics, recovery from failure, and an idiomatic Go implementation of workflows and activities).

We described how key event types and attributes will be extracted from JSON logs and modeled as structured events so that Timeline operators can treat them uniformly. We proposed design patterns for handling dynamic event types that keep the system extensible as new log schemas emerge. The Temporal workflow design balances real-time streaming ingestion (via signals) with on-demand batch queries (via triggered replay workflows), with careful use of Temporal features like signals, continues, and activities for durability and performance [4] [8]. We also provided a detailed example of a real query to illustrate the end-to-end data flow and the interactions between components.

Finally, the testing strategy ensures that these new workflows will be reliable from day one – covering unit tests for classification logic, integration tests for Iceberg and VictoriaLogs interactions, and simulations of schema changes and edge cases to prove out the design in practice.

With this design, the Timeline Framework will be capable of **seamlessly querying historical timelines** without sacrificing the real-time capabilities. Analysts and systems can trigger complex backfills or replays (for investigative analysis or retroactive metrics) and get results with the same confidence as if those events were processed in real-time. This bridges the gap between live monitoring and historical analytics, all within an idiomatic, maintainable Go codebase orchestrated by Temporal. The system remains scalable, robust, and flexible to future needs, supporting our growing data and evolving event schemas in a principled way.

Sources:

- Temporal Go SDK Documentation – Testing Workflows [10]
- VictoriaMetrics Documentation – VictoriaLogs features (high-cardinality indexing, backfilling support) [2]
- Temporal usage in data pipelines – best practices for long-running workflows and Continue-As-New [4] [8]
- AWS Big Data Blog – Using Apache Iceberg for log analytics (highlights Iceberg's concurrent ingest/query, time travel, schema evolution) [3]

1 3 Streamline AWS WAF log analysis with Apache Iceberg and Amazon Data Firehose | AWS Big Data Blog
https://aws.amazon.com/blogs/big-data/streamline-aws-waf-log-analysis-with-apache-iceberg-and-amazon-data-firehose/

2 5 6 VictoriaLogs
https://docs.victoriametrics.com/victorialogs/

4 7 8 9 Event-driven and reactive Data Ingestion pipelines with Temporal
https://www.linkedin.com/pulse/event-driven-reactive-data-ingestion-pipelines-tarun-annapareddy

10 11 Testing - Go SDK | Temporal Platform Documentation
https://docs.temporal.io/develop/go/testing-suite