ChatGPT

Product Requirements Document: Timeline Framework Service (Go 1.24.3)

Introduction and Background

Time-state analytics refers to analyzing context-sensitive metrics over continuously evolving state, such as tracking system or user states over time [1]. Traditional data pipelines (SQL, timeseries DBs, streaming frameworks) rely on tabular models (rows of events or time ticks) and are ill-suited for stateful, continuous-time calculations [1]. This often leads to *"hackish and unreadable code"* that is hard to develop and maintain, and yields poor performance at scale [1] [2]. The Timeline Framework (from CIDR'23) raises the abstraction level for time-state analytics by providing a geometric, timeline-based mental model [3] [4]. Instead of forcing events into tables, analysts can work with intuitive **Timeline data types** and operations (the **Timeline Algebra**) to model stateful behaviors directly [5] [6]. For example, one can *"fill forward"* discrete events into a continuous state timeline (using a `LatestEventToState` operator) and combine multiple state timelines to compute metrics like *connection-induced rebuffering time* in video streaming [7] [6]. In production, this approach has shown an **order-of-magnitude reduction** in development complexity and significant cost-performance gains [8].

This PRD defines a high-level design to implement the full Timeline Framework in Go v1.24.3, delivering a RESTful service that exposes timeline-based analytics capabilities. The system will use Go's standard HTTP router for API endpoints and employ Temporal (a workflow orchestration engine) to reliably coordinate ingestion of events, transformation of those events into timeline data, and execution of timeline queries. All persistent metadata and state will be kept within Temporal workflows (leveraging Temporal's built-in persistence), while large data blobs (e.g. event logs, lengthy timeline representations) are stored in AWS S3 with only references kept in workflows. We will utilize structured logging via Go's `log/slog` package for observability, and follow idiomatic Go patterns (using generics sparingly and only when needed) as well as Temporal best practices for workflow design. Comprehensive testing (unit, integration, end-to-end) will be implemented, including translating all examples and use cases from the CIDR paper into automated tests using Temporal's testing framework. The goal is a clear, maintainable architecture that enables another engineer to implement the Timeline Framework without needing to read the original research paper.

Objectives and Key Requirements

Functional Requirements:
- **Timeline Query API:** Provide a RESTful HTTP API (using Go's built-in `net/http` router) that allows clients to ingest events and perform timeline queries. The API should expose the Timeline Framework's query interface – i.e. clients can specify analytics computations in terms of timeline operations (such as converting events to state timelines, combining timelines, and extracting metrics). The service must interpret these requests, execute the appropriate timeline transformations, and return results.
- **Temporal Orchestration:** Use Temporal workflows to orchestrate all background processing: - **Event Ingestion Workflows:** Ingest incoming events reliably and update timeline state. Each timeline (e.g.

representing a distinct entity or session being analyzed) will have an associated long-running Temporal workflow responsible for accumulating its events. - **Timeline Transformation & Query Workflows:** Execute complex timeline algebra operations as needed. When a query request comes in, initiate Temporal workflows (or activities) that fetch stored event data, transform it into timeline form, apply the requested timeline operators, and compute the final result. - **Workflow Coordination:** Define clear **workflow ID schemes**, parent/child workflow relationships, and usage of Temporal **signals**: - Assign stable workflow IDs per timeline (e.g. `"timeline-{entityID}"`) so that all events for a given timeline route to the same workflow instance. Leverage Temporal's guarantee of single workflow execution per ID to ensure in-order, exactly-once processing of events. - Use **signals** to asynchronously deliver new events to running timeline workflows (avoiding constant polling). Signals should carry event payloads, and workflows will handle them in a defined signal handler, buffering or immediately processing the event. - Employ **parent-child workflows** where appropriate. For example, a high-level *QueryWorkflow* might spawn multiple child workflows to compute parts of a query in parallel (e.g. querying multiple timelines concurrently in a fan-out) and then aggregate the results. Similarly, an ingestion workflow might spawn a child workflow for heavy computations or periodic batch processing (to avoid blocking the main event loop). - Provide recommendations for **workflow ID strategies** to avoid collisions and allow easy management (e.g. using well-defined prefixes or namespaces for different workflow types like `timeline-<id>` for ingestion and `query-<guid>` for one-off query executions). - **Data Model – Timeline Abstraction:** Implement the core Timeline data types and operations as described in the paper: - Support the **three basic Timeline types** – (1) *Discrete Event* timeline, (2) *State timeline* (state transitions over time), and (3) *Numeric timeline* (time-varying numeric values) [9]. These data types capture fundamental temporal dynamics (events at points, piecewise-constant states, and continuously changing or sampled values) [4]. - Provide timeline algebra **operators** to transform and combine timelines. Key operations include:

- **LatestEventToState:** a fill-forward operation to convert a stream of discrete events (e.g. state change events) into a piecewise-constant state timeline [7]. This yields the "current state" of an entity at each moment by holding the last event's value until a new event arrives.

- **HasExisted:** produces a boolean timeline that is true after a certain event has occurred at least once (and remains true thereafter) [10]. This is useful for context like "has playback started at this time" or any condition that triggers a permanent state after first occurrence.

- **HasExistedWithin:** a boolean timeline that is true if a certain event occurred within a rolling time window up to the present moment [11]. For example, `HasExistedWithin(userAction="seek", 5s)` is true at times when a *seek* event happened in the last 5 seconds, and false otherwise [12]. (This can be used to **exclude** periods immediately following a user seek, by negating the timeline).

- **Logical Combination:** ability to combine timelines with logical AND, OR, NOT to form composite conditions. E.g., intersect a buffering-state timeline with the negation of a "recent seek" timeline to isolate buffering events not caused by seeking [6].

- **Duration and Aggregation:** operations to reduce a boolean or state timeline to numeric metrics. Chief among these, **DurationWhere** computes the total duration that a given condition holds true [13]. This is essentially an integration or sum of time for which a boolean timeline is true. (Other possible aggregations could include counting occurrences of an event, computing time-weighted averages of a numeric timeline during certain states, etc., though *DurationWhere* covers the primary use case of the paper's examples.)

- **Value sampling/evaluation:** in general, the framework might allow querying the value of a timeline at a specific time or summarizing numeric timelines (e.g. `EvaluateAt(t)` or computing max/min over an interval), but the core scope will focus on the operations explicitly mentioned in the paper's use cases.

- Express timeline queries as compositions of the above operators. The system should allow arbitrary DAGs of these operations (the paper describes a *"DAG of Timeline operators"* as the way to formulate complex queries [6] [14]). The API will need to capture such a composition (see **API Design** below). Internally, ensure

the timeline operations are implemented as reusable library functions that can be unit-tested in isolation. Use Go generics *only if needed* to avoid code duplication (for instance, if state and numeric timelines share algorithmic logic, a generic function or interface could be used; otherwise, prioritize clarity with separate implementations for each type). All code should be idiomatic Go, e.g. using slices for sequences, `time.Time` or `time.Duration` for time values (instead of custom epochs), and avoiding overly complex type hierarchies.

- **API & Service Implementation:** Develop the service in Go 1.24.3, using the standard `net/http` router (`http.ServeMux` or `http.HandlerFunc`) for request routing (no external frameworks). Define clear endpoints and request/response formats:
- `POST /timelines/{id}/events` – Ingest new events for a given timeline. The request body will contain one or multiple events (optionally allow batching for efficiency). Each event includes at least a timestamp and an event type or name, plus optional metadata or value. Example JSON for a single event:

```
{
  "timestamp": "2025-06-02T08:30:00Z",
  "type": "PlayerStateChange",
  "value": "Play"
}
```

  On receiving this, the service will log the request, then signal the corresponding Temporal workflow (with ID e.g. `"timeline-{id}"`) to handle the event. The response can be 202 Accepted if ingestion is asynchronous, or 200 OK if we choose to confirm synchronous completion of the signal handling. (In practice, we will treat ingestion as fire-and-forget: once the event is handed off to the workflow, we return success. The workflow's durability ensures it will be processed.)
- `POST /timelines/{id}/query` – Submit a timeline query on a specified timeline (or possibly across multiple timelines; see **Future Extensions**). The request should describe the query in terms of Timeline operations. We will use a structured JSON format to represent the query DAG. For example, the connection-induced rebuffering metric from the paper can be expressed as:

```
{
  "operations": [
    {
      "id": "bufferPeriods",
      "op": "LatestEventToState",
      "source": "playerStateChange",
      "equals": "buffer"
    },
    {
      "id": "afterPlay",
      "op": "HasExisted",
      "source": "playerStateChange",
      "equals": "play"
    },
    {
```

```
      "id": "noRecentSeek",
      "op": "Not",
      "of": {
        "op": "HasExistedWithin",
        "source": "userAction",
        "equals": "seek",
        "window": "5s"
      }
    },
    {
      "id": "cdnFilter",
      "op": "LatestEventToState",
      "source": "cdnChange",
      "equals": "CDN1"
    },
    {
      "id": "result",
      "op": "DurationWhere",
      "conditionAll": ["bufferPeriods", "afterPlay", "noRecentSeek",
"cdnFilter"]
    }
  ]
}
```

This JSON defines intermediate named results and a final operation. The service will parse this specification and execute it via the Timeline engine. The response could be synchronous (returning the result value, e.g. total duration in seconds) if computation is quick, but since complex queries may involve significant data, the safer design is asynchronous. In asynchronous mode, the endpoint would immediately start a Temporal *QueryWorkflow* to perform the computation and return a job identifier (e.g. a query workflow ID or a token). The client can then poll a `GET /queries/{queryId}` endpoint for completion or have results pushed via webhook (not in scope for initial version). However, to keep the interface simple, we may choose to block the HTTP response until the query workflow completes, with a reasonable timeout. This blocking call will utilize the Temporal client's capability to wait for workflow result (with appropriate timeout to avoid hanging indefinitely). For the initial implementation, we will assume moderate data sizes and use synchronous query handling (with timeouts); we will document that extremely heavy queries should use an async pattern.

- **API Output:** The service should return results and statuses in JSON. For example, a successful query response might be: `{"result": 12.5, "unit": "seconds"}` for a duration metric, or a structured timeline if the query returns a timeline (not common for final outputs, but the API could allow retrieving a timeline as an array of intervals or events). Errors (bad input, internal failures) should result in appropriate HTTP status codes (400 for validation errors, 500 for internal exceptions, etc.) and error messages in JSON.

- **Structured Logging:** All components of the system will produce structured, context-rich logs using Go's `slog` package (part of the standard library for structured logging). We will configure `slog`

with a JSON output handler by default (for easy ingestion into log aggregators), or a text handler in development for readability. Each log entry will include a severity level (Info, Debug, Warn, Error) and key fields such as: timestamp, service component (HTTP handler, workflow, activity, etc.), timeline ID (if applicable), workflow ID, and correlation IDs where relevant (e.g. an HTTP request ID that flows into workflows). For example, when an ingest request is received, we log an Info like:

```
logger.Info("Ingest event request",
  "timelineID", id,
  "eventType", evt.Type,
  "ts", evt.Timestamp)
```

Inside Temporal activities (which run in normal Go context), we can use the same `slog` logger to record operations (including S3 read/write actions). Inside workflows (which are deterministic functions), we will use Temporal's provided logger (`workflow.Logger`) – we can either adapt it to `slog` or use it directly but ensure it also includes the workflow and timeline identifiers for traceability. Logs should be structured to facilitate debugging and auditing of the system's behavior, without leaking sensitive data. We will follow idiomatic usage of `slog`, e.g. using `logger.With()` to attach static context (like adding a `workflowID` field for all logs within a given workflow execution).

- **Performance & Scalability Requirements:** The system must be designed to scale to high event volumes and large timelines:
- The architecture should **horizontally scale**: We can run multiple instances of the service and multiple Temporal worker processes. Because each timeline has its own workflow (sharded by ID), ingest load is naturally partitioned. Temporal's task queues will distribute workflows and tasks across workers. The design will ensure no single workflow handles more than one timeline's events, preventing lock contention across timelines.
- As a timeline's event volume grows, the system should avoid performance degradation. We will use **Temporal's "Continue-As-New"** feature for long-running workflows: the timeline ingestion workflow will periodically checkpoint and start a fresh workflow run (carrying over essential state) when it reaches a certain size (e.g. after processing N events or when its history size approaches a threshold). This prevents any single workflow execution from accumulating an extremely large event history in Temporal's database. The workflow's internal state (like last known state or unprocessed buffer) will be passed to the new run. All new signals will automatically route to the latest run (Temporal ensures continuity when using the same workflow ID). This mechanism allows infinite event streams to be handled over time without exhausting Temporal resources.
- **S3 storage for large data**: Event payloads and constructed timelines that are too large to keep in memory or in workflow state will be stored in Amazon S3. For example, the ingestion workflow might append incoming events to an S3 object (or a series of objects) rather than storing every event in memory. Only a reference (S3 key/URL) and maybe a small buffer of recent events would be kept in the workflow's state. Similarly, if a query requires the full timeline, the workflow can retrieve it from S3. Offloading to S3 ensures that even if a timeline has millions of events, the Temporal workflow state remains small (just pointers), and workers only load data on demand. We will design the S3 data layout to allow efficient access: e.g., one object per timeline containing all events, or objects partitioned by time windows (such as one file per day/week of events for that timeline). The workflow stores these object keys. When a query arrives for a specific time range, it can fetch only the relevant objects.

- **Workflow concurrency & throughput:** Temporal will ensure each workflow is single-threaded (to preserve determinism), so a single timeline's events are processed sequentially. This is acceptable for moderate event rates. If a particular timeline has an extremely high event throughput that one workflow cannot keep up with, we acknowledge this could become a bottleneck. In such cases, the design could be extended with *timeline partitioning* – for instance, splitting a single logical timeline into sub-timelines by time range or event type, each handled by its own workflow, and merging results as needed. However, this adds complexity and is not in the initial scope. We will note this as a potential future scaling improvement. For now, we assume each single timeline's event rate is within what one Temporal workflow can handle (Temporal can handle hundreds of signals per second per workflow in our expected deployment, which is sufficient for many use cases).

- **Parallel query execution:** The system should handle multiple query requests concurrently. Different queries will simply start distinct Temporal workflows (or activities) that run in parallel across the worker pool. Within a single complex query, if sub-tasks can be parallelized (e.g. processing multiple timelines), the QueryWorkflow can launch them as child workflows or parallel activities. For example, if we introduced a query that spans multiple timeline IDs (say, "compute a metric across all user sessions in region X"), the QueryWorkflow could spawn a child workflow per relevant timeline (using the timeline ID as input) and then aggregate their results. Temporal makes it straightforward to wait for multiple child workflows to finish (using `workflow.ExecuteChildWorkflow` and waiting on futures) [15] [16]. This parent-child pattern will be recommended for any multi-entity queries in future extensions. (In the initial version, queries will target one timeline at a time, so parallelism is mainly at the multi-request level, not within a single request.)

- **Testability and Quality:** All components will be built with testing in mind, targeting **80% or higher code coverage**. We will write exhaustive tests to ensure the correctness of timeline operations and the reliability of workflow orchestration. The examples and use cases from the paper will directly inform test cases:

- Every example described in the CIDR paper will have a corresponding automated test. For instance, the video streaming *"connection-induced rebuffering"* metric scenario will be used to craft a test timeline of events and verify that the system computes the correct rebuffer duration. We will encode the event sequence from Figure 1 of the paper (e.g. events like `Init`, `Play`, `Pause`, `Seek`, CDN switches, etc., at specific timestamps) and assert that the query result matches the expected outcome (the paper's example indicates certain time intervals that should count towards the metric [17] [18]). This ensures our Timeline Algebra implementation aligns with the intended semantics.
- **Unit Tests:** We will create unit tests for individual timeline functions (pure logic without Temporal). For example, test the `LatestEventToState` function with a simple sequence of events: ensure it produces correct state intervals (e.g. if events [Play at t1, Buffer at t2, Play at t3], the resulting state timeline should be "Play" from t1 to t2, "Buffer" from t2 to t3, then "Play" from t3 onward). Similarly, test `HasExisted` by feeding an event stream with one matching event and confirming the output timeline flips to true after that event time, and test `HasExistedWithin` with various event timings relative to the window size. Each timeline operator will have targeted tests for edge cases (no events, events exactly at window boundaries, overlapping windows, etc.). These unit tests use Go's `testing` package and will be run with race detection to ensure thread safety where applicable (though most timeline logic is single-threaded).
- **Integration Tests (with Mocks):** We will test interactions between components by substituting dependencies with mocks. For instance, test the HTTP ingestion handler by mocking the Temporal

client: we simulate that when the handler calls `client.SignalWorkflow` or `client.StartWorkflow` it succeeds, and verify that the handler returns the correct HTTP status. We will use dependency injection for external services like S3 as well: define an interface (e.g. `StorageClient` with methods `AppendEvent(timelineID, event)` and `LoadTimeline(timelineID)`), and implement a fake in tests to verify that the ingestion workflow calls the storage correctly. We can also simulate error conditions (e.g. S3 failure) by having the fake return an error, and assert that our workflow handles it (retries or logs an error). Temporal's Go SDK allows registering activity mocks in its test environment, so we can intercept calls like `WriteToS3Activity` and simulate results without actually connecting to AWS.

- **End-to-End Tests:** Using **Temporal's testing suite**, we will spin up a **Test Workflow Environment** (provided by the Temporal SDK for Go) that runs workflows and activities in-memory [19] [20]. In this environment, we will register our actual workflow and activity implementations, but for full end-to-end tests we may **mock out external side effects**. For example, we will register a fake implementation of the S3 write/read activities to avoid needing real S3 (the fake can simply record that data "would be written" and allow retrieval from an in-memory map). We will then simulate a full scenario:
    1. **Ingestion E2E Test:** Start a timeline ingestion workflow (via `env.ExecuteWorkflow` in the test) for a sample timeline ID. Then simulate incoming events by calling `env.SignalWorkflow(workflowID, signalName, eventPayload)`. After sending a series of events, we can use Temporal's query feature or track the workflow state to ensure events were processed (e.g. perhaps the workflow updates a counter we can query). We will also assert that the fake S3 storage contains the expected event data (meaning our activity was called with correct inputs).
    2. **Query E2E Test:** Using the same test environment, we then execute a QueryWorkflow for that timeline ID (or simply call the query workflow function directly in test with appropriate parameters). The query workflow will call our fake S3 read (getting the events we "stored" earlier) and then run the timeline computations. We verify the final result matches expectations. For example, using the video rebuffering scenario, we expect the query result to equal the sum of certain time intervals. We will calculate the expected result in the test based on the known input event times and compare it to the workflow's return value.
    3. **Full HTTP-to-Workflow Test (optional):** For true end-to-end coverage, we can run a lightweight HTTP server in a goroutine (backed by our real handlers but configured to use the test Temporal environment and the fake S3 client). Then the test can perform actual HTTP requests (using Go's http test client) to the `/events` and `/query` endpoints with sample data, and check the HTTP responses. This would validate the wiring from HTTP layer through Temporal and back. However, this approach is somewhat complex (since tying the test environment with the HTTP layer requires custom Temporal client wiring). We may cover the same logic by calling handler functions directly and using the Temporal test environment underneath (which is simpler and still represents end-to-end logic sans actual network calls).

- We will ensure **idiomatic testing practices**: clear, small test functions with descriptive names, table-driven tests where appropriate for multiple cases, and avoidance of flaky timing-based checks (Temporal's test environment can advance "virtual time" if needed to simulate timers without waiting in real-time). Code coverage will be measured with `go test -cover`, and we will aim for at least 80% of lines covered, focusing on critical logic (timeline computations, workflow decision paths, etc.).

- **Non-Functional Requirements:**

- **Idiomatic Go Design:** The codebase will follow standard Go project structure. We will organize packages for clarity:
  - e.g. `pkg/api` for HTTP handlers, `pkg/timeline` for timeline data types and operations (pure logic), `pkg/workflows` for Temporal workflow and activity definitions, and `pkg/storage` for S3 integration, etc.
  - Use Go **context** per request and propagate cancellation: the HTTP handlers will respect `req.Context()` to cancel operations if the client disconnects. The Temporal client calls and S3 calls will accept context. Within workflows, Temporal provides its own context that handles cancellation if a workflow is terminated.
  - Use **Go concurrency safely**: Most of our concurrency is managed by Temporal (which sequences workflow execution and can schedule activities in parallel). We will avoid spawning uncontrolled goroutines in workflows or activities. In the HTTP server, we rely on the server's concurrency (one goroutine per request). We will protect any shared structures (likely minimal, maybe an in-memory cache or the logger instance) with mutexes if needed or prefer lock-free designs.
  - **Generics usage:** Go generics will be used in a limited fashion. For example, if we implement a generic function to merge overlapping intervals that could work for both state intervals (with a state value) and boolean intervals (with a true/false value), we might use a type parameter for the interval value. However, we will be cautious – readability and clarity are paramount. Many timeline operations can be implemented in a straightforward way without generics (or by using interfaces like a `Timeline` interface with methods, which might actually be more idiomatic for varying behavior). We will not introduce generics unless it clearly reduces duplication and remains easy to understand for other engineers.
  - **Temporal idioms:** Workflows and activities will be implemented following Temporal's Go SDK guidelines. Workflows will be defined as functions that take a `workflow.Context` and input parameters, and they will use `workflow.ExecuteActivity` to call activities. No blocking I/O or random timestamps in workflow code – all external calls go through activities. We will configure appropriate **Activity Options** (like timeouts and retry policies). For example, S3 write/read activities might have a retry policy to handle transient network errors, and a timeout to avoid hanging a workflow too long. Parent workflows will properly await child workflows or activities using futures, and use `workflow.Select` or channels if waiting for multiple concurrent tasks. We will also implement **signal handlers** in workflows: the ingestion workflow will use `workflow.GetSignalChannel` to listen for incoming events, and loop to process them. If signals arrive faster than processing, the channel will queue them (Temporal's channels are buffered). We will document the signal names and payload schemas clearly (e.g. a signal named `"NewEvent"` carrying an `Event` struct).
  - **Workflow ID and Task Queue management:** We will decide on a naming convention for workflows and ensure the Temporal client calls use the correct IDs. For instance, when an ingest request comes, the service will call `StartWorkflowOptions{ ID: "timeline-{id}", TaskQueue: "TIMELINE_INGEST_QUEUE", WorkflowIDReusePolicy: AllowDuplicate/RejectDuplicate }` and then `ExecuteWorkflow(TimelineIngestionWorkflow, id)` if the workflow doesn't exist. If it already exists, the Start call will fail with "workflow already running", in which case we'll instead send a signal via `client.SignalWorkflow`. (Temporal also offers an upsert signal via `SignalWithStart`, which we can use to simplify: this single API call will start the workflow if not running or signal it if it is [21]. We will likely use `SignalWithStart` for ingestion for atomicity and simplicity.) All query workflows might use a separate task queue

(e.g. "TIMELINE_QUERY_QUEUE") which could be processed by the same worker pool or a dedicated set of workers if we want to isolate load.

- **S3 integration:** We will use AWS SDK for Go (v2) in the `pkg/storage` for writing to and reading from S3. This will be encapsulated in a small set of functions or activities (`WriteEventsToS3`, `ReadEventsFromS3`) to keep the interface between our system and S3 narrow (easy to mock and replace). We'll ensure S3 keys are structured (perhaps prefix by timeline ID and date, etc.) and consider enabling server-side compression for large logs. Any credentials for S3 will be passed via secure config (not hard-coded) and the code will follow best practices (e.g. retry on throttling, use multi-part upload if needed for very large payloads, although initial implementation can assume events are small and just PUT an object).
- **Security & Privacy:** Although not explicitly requested, we should mention that this design will enforce proper data isolation per timeline (each timeline has its own data file and workflow). The API will validate inputs to avoid injection (since we parse JSON, we should ensure the query spec only contains allowed operations and fields). If multi-tenant, authentication/authorization would be layered on top (out of scope here, but e.g. an API key mapping to allowed timeline IDs). All logs and stored data should avoid containing sensitive personal information unless necessary (and if so, should be protected by access controls).

System Architecture Overview

Overall Architecture: The system is composed of a **RESTful API service** and the **Temporal workflow engine** with worker processes executing the workflows (the workers can be part of the same service process or separate). Additionally, it integrates with **Amazon S3** for durable storage of event data and intermediate large results. The following diagram illustrates the major components and data flow:

Timeline Framework architecture and query flow. Ingestion events flow from clients via HTTP into Temporal workflows which update Timeline state (persisting raw events to S3). A query request triggers a Temporal workflow (or activity) that fetches needed event data from S3, performs timeline transformations (LatestEventToState, etc. forming a DAG of operations), and returns the result. The result is then sent back to the client through the REST API.

Request Handling Flow:
1. **Event Ingestion:** A client (or upstream system) sends events to the `POST /timelines/{id}/events` endpoint. The API handler authenticates (if applicable) and quickly parses the JSON body into our internal `Event` struct(s). The handler then calls the Temporal client's `SignalWithStart` for workflow ID `"timeline-{id}"` on the *Ingestion Task Queue*. If no workflow exists for that timeline, Temporal will start one (running our `TimelineIngestionWorkflow` function). If it's already running, it delivers the new event signal to that workflow. The handler immediately responds 202 Accepted (the event is safely handed off to the workflow for processing). The ingestion workflow receives the signal: in its loop, it unmarshals the event data and appends it to an internal list or buffer. It then calls an **activity** like `AppendEventActivity` with the event details. That activity executes outside the workflow context (on a worker), performing the S3 write (e.g. appending to a file or writing a new object). On success, the workflow updates any in-memory state (for example, it might update a "current player state" field if this event was a state change, so it knows the latest state without rereading from S3). The workflow then waits for the next signal. This design decouples the ingestion rate from S3 I/O by allowing the workflow to batch events if needed (for example, the workflow could accumulate events for a short interval and flush them in one go to S3 to reduce calls –

this can be a configurable optimization).
2. **Timeline Query:** A client calls `POST /timelines/{id}/query` with a query specification. The API handler verifies the timeline ID and parses the query JSON into an internal representation (perhaps a struct with a list of operations, or directly into a closure that can execute the query – there are multiple implementation approaches, but internal representation likely a tree or list of operations as in the JSON). The handler then starts a **QueryWorkflow** on the *Query Task Queue*, passing the timeline ID and the query spec. This `QueryWorkflow` will perform the following in steps:
a. **Data Retrieval:** Depending on the query, determine what data is needed. By default, it will need the entire event history (or a substantial subset) of the timeline. It invokes a `ReadTimelineActivity` (or possibly multiple, if data is sharded) to load the raw events from S3. The activity will fetch from S3 and return the events (or a compressed timeline representation). The workflow now has, say, a list of events in memory.
b. **Timeline Construction:** The workflow (or a sub-activity) applies timeline transformations to these events. We have two possible approaches: (i) Do the transformations within the workflow code (in-memory) – since our operations are pure computations and the data set for one timeline might be reasonably sized (e.g. maybe thousands of events). This keeps it simple and deterministic. Or (ii) call a `ComputeQueryActivity` where we pass in the events and the query spec, and the activity executes the timeline algebra and returns the final result. Approach (ii) offloads computation from the workflow thread, which is better if data is very large or operations are heavy. We prefer offloading heavy computation to an activity for scalability. Thus, `QueryWorkflow` will likely call something like `ExecuteActivity(ComputeTimelineQuery, events, querySpec)` which returns either the numeric result or a smaller dataset.
c. **Result:** The workflow then completes, returning the result (e.g. a float or an integer or JSON) as the workflow outcome. The API handler, if waiting synchronously, will have been blocking on the workflow future – it now unblocks and gets the result. It formats it into the HTTP response (e.g. JSON with the number or timeline). If we opted for an asynchronous API, the handler would instead return a query ID immediately and not wait; in that case, we would need another mechanism (like client polling `GET /queries/{id}` which internally calls `client.QueryWorkflow` or checks if workflow finished). For simplicity, we will implement synchronous wait for now, with the understanding that it may timeout for extremely long queries.
d. **Parallelism Consideration:** If the query spec indicated multiple timelines or some parallelizable sub-queries (not in initial scope), the QueryWorkflow could spawn children. For example, if querying 10 timeline IDs, the workflow would start 10 child workflows in parallel (each child does steps (a)–(b) for one timeline) and then aggregate their numeric results in the parent (perhaps summing them, etc.). Temporal's parent-child execution ensures the parent is notified when each child completes [15] [16]. This pattern will be recommended if needed in the future.

1. **Workflow and Activity Workers:** The Temporal **Worker** is a Go process (which can be the same process as the API or separate) that registers all our workflow and activity functions and polls the Temporal service for tasks. In our design, we may have a single binary that on startup does both: starts an HTTP server *and* starts a Temporal worker (with two task queue subscriptions, one for ingest workflows and one for query workflows). This combined approach is convenient for development and ensures the API can immediately trigger workflows on the local worker. In a production deployment, however, it might be more scalable to separate concerns: run multiple instances of just the worker (no HTTP) for background processing, and separate instances of API servers for handling client requests. We will design the system to support either mode. All configurations (like Temporal service address, S3 bucket names, credentials, etc.) will be provided via environment variables or config files to not hard-code values.

2. **Temporal Service:** We assume a Temporal server/cluster is available (it could be self-hosted or Temporal Cloud). The workflows and activities rely on Temporal's guarantees for reliability: if an activity fails, Temporal will automatically retry it (per policy); if a worker crashes while a workflow is mid-execution, Temporal will reschedule that workflow on another worker without losing state (since the state is persisted in Temporal's database). This greatly simplifies error handling in our code – we focus on defining retries and compensations as needed, and Temporal handles the rest. We will still implement proper **error logging and handling** in workflows: e.g., if a non-retryable error occurs (like a bad query spec), the workflow can fail and we propagate that error message to the API response.

Timeline Framework Details and Examples

To solidify the understanding for implementers, this section walks through a concrete example from the paper using our proposed system, and outlines the data structures and pseudo-code for key operations.

Example Use Case: Connection-Induced Rebuffering (Video Analytics)
In a video streaming context, suppose we want to calculate *"how much time a video session spent buffering due to connection issues while on CDN1 (as opposed to initial load or user actions)"*. This was the motivating example in the paper [22] [23]. Traditionally, this would require complex SQL or streaming jobs, but with the Timeline Framework it's a sequence of simple timeline operations [6]. We will illustrate how our system would handle this query.

<small>*Inputs:* A timeline of events for one video session. The events include: `Init` (session start), `Play` (user started playback), `Pause`, `Seek` (user seeked), and `RebufferStart/RebufferEnd` or a `PlayerStateChange` event that indicates entering "Buffering" state or leaving it, as well as `CDNSwitch` events indicating switching the CDN (content delivery network) between CDN1 and CDN2. Each event has a timestamp. The raw event stream might look like:*</small>

```
t1: Init
t2: RebufferStart       (player starts buffering after init)
t3: CDNSwitch CDN1      (switch to CDN1)
t4: Play                (user presses play, buffering ends at this moment
implicitly)
t5: ...
t6: RebufferStart       (buffering start, possibly connection issue)
t7: RebufferEnd         (buffering ends)
t8: RebufferStart       (another rebuffer, maybe connection issue)
t9: RebufferEnd         (ends)
t10: Seek               (user seeks, causing likely a buffering)
t11: RebufferStart      (buffering due to seek)
t12: RebufferEnd        (ends)
t13: CDNSwitch CDN2     (switch to CDN2)
t14: ...
```

(This is an illustrative sequence; actual events could differ. In our tests, we will use a concrete set matching Figure 1 of the paper [24] [25].)

Using timeline operations, we would calculate the metric as follows:

- **Player State Timeline:** Apply `LatestEventToState` on the player's state-change events (e.g. events that signal **buffering vs playing**). This produces a timeline where at any given time the state is either "Buffering" or "Playing/Paused" etc. For the example, this timeline would show segments of "Buffering" from t2 to t4, then "Playing" from t4 until t6, "Buffering" t6–t7, "Playing" t7–t8, "Buffering" t8–t9, "Playing" t9–t10, "Buffering" t10–t12, etc. We specifically care about when it's "Buffering". So we could get a boolean timeline `isBuffering = (LatestEventToState(playerStateChange) == "Buffering")`. This timeline is true exactly during those buffering intervals.

- **Exclude Initial Buffering and Seek-induced Buffering:** We refine `isBuffering` by excluding periods that are due to *initial startup* or *user seeks*. How to detect these? The paper's approach is:

- Define a timeline `hasPlayed = HasExisted(playerStateChange == "Play")`, which becomes true after the first *Play* event (i.e., after the user has started playback) [10]. Before playback starts, any buffering is "initial buffering" which we want to ignore. So by requiring `hasPlayed` to be true, we only count buffering that happens after playback began. Effectively we ignore buffering before the first play.
- Define `recentSeek = HasExistedWithin(userAction == "Seek", 5s)`, which is true if a seek occurred in the last 5 seconds. If the player is buffering and this timeline is true, it means the buffering is likely caused by a seek that happened just now, so we should ignore it (the problem definition only wants connection-induced, not user-induced). We actually need the **negation** of this: `noRecentSeek = NOT(recentSeek)`. This is a boolean timeline that is true whenever a seek hasn't happened in the last 5 seconds.

- Now refine the buffering timeline: `effectiveBuffering = isBuffering AND hasPlayed AND noRecentSeek`. This timeline is true only during buffering events that occur *after* playback started and not within 5s of a seek – i.e., buffering due to connection issues (not startup or seeks) [11].

- **CDN Filter:** We are specifically interested in times when the CDN is "CDN1". We apply `LatestEventToState` to the CDN switch events to get a timeline of which CDN is in use over time (e.g., timeline might say CDN is "C1" from t3 to t13, then "C2" onward). Create `usingCDN1 = (LatestEventToState(cdnChange) == "CDN1")` [18]. This is a boolean timeline true when CDN1 is active.

- **Combine Conditions:** Now take the intersection of the conditions: `problemBuffering = effectiveBuffering AND usingCDN1`. This timeline indicates periods where the player was buffering (with context filters applied) *and* was on CDN1.

- **Measure Duration:** Finally, apply `DurationWhere(problemBuffering)` which sums up the total time this timeline is true [13]. The output is a numeric value (e.g. "X seconds"). This is our desired metric (often called *CIR – connection-induced rebuffering* duration).

Our system will automate this pipeline. The query JSON shown earlier encodes these steps. The `ComputeTimelineQuery` activity would implement this logic, or the workflow would call our timeline library functions in this sequence. To double-check, the result should equal the intervals t6–t7 and t8–t9 in

the example (since those were buffering on CDN1 after play started and not within 5s of a seek; the buffering at t10–t12 would be excluded because it's right after a seek at t10, and t2–t4 excluded as it's before first play). So if t6=10s, t7=12s, t8=20s, t9=25s for example, the total would be (12-10) + (25-20) = 7 seconds. Our test case for this scenario will assert that the result matches the expected sum of those intervals.

The above example demonstrates how Timeline Algebra allows queries to be specified as compositions of simple operations, closely mirroring an analyst's thought process [6]. The PRD ensures that all such operations described in the paper are supported and that they are orchestrated correctly in a distributed setting via Temporal.

Internal Data Structures: To implement the timeline operations, we will define some core Go types:

```
// Event represents a discrete event with an optional value or metadata.
type Event struct {
    Timestamp time.Time
    Type      string            // e.g. "playerStateChange", "userAction",
"cdnChange"
    Value     string            // e.g. "Play", "Seek", "CDN1" (or empty if not
needed)
    // ... possibly other fields like duration or numeric value for numeric
events
}

// StateInterval represents an interval during which an entity was in a certain
state.
type StateInterval struct {
    State    string    // state name or category
    Start    time.Time // inclusive start of interval
    End      time.Time // exclusive end (or zero if ongoing till "now")
}

// Timeline data containers
type EventTimeline []Event
type StateTimeline []StateInterval
type BoolTimeline []StateInterval // intervals where condition holds (State
could be implicitly "true")

// NumericTimeline could be represented similarly as piecewise segments or a
series of (time, value) points.
type NumericTimeline []struct {
    Value    float64
    Start    time.Time
    End      time.Time
}
```

We will implement functions corresponding to timeline operations. For example, `LatestEventToState(events EventTimeline, initialState string) StateTimeline` will sort the events by time and then produce a StateTimeline: it will start at the first event's time (or timeline start) with the state from that event's value, and change state at each event. We might need to define how the timeline ends (e.g. an interval can go until "now" or until the end of the query window – in queries we might consider the timeline up to the latest event or a provided end time). For simplicity, we can assume the timeline extends to the last event timestamp, and for metrics like duration we only consider up through that.

Other operations: - `HasExisted(events EventTimeline, condition func(Event) bool) BoolTimeline` – returns a boolean timeline that flips to true when an event satisfying the condition occurs. Implementation: scan through events in time; before the first matching event, the timeline is false; after the first match (at time T), produce an interval [T, end) as true. We will likely cap "end" at the last event or query end time.
- `HasExistedWithin(events EventTimeline, condition, window) BoolTimeline` – returns a timeline that is true whenever a matching event occurred in the last `window` duration. Implementation: we can iterate through events and maintain a sliding window. One approach: for each event that matches, mark an interval [eventTime, eventTime+window) as true (meaning for that duration after the event, the condition "has existed within window" is true). Then, we take the union of all such intervals, and also ensure it resets to false window-duration after the last event. (The pseudocode in the paper indicates it "resets to False after D seconds" [26], which aligns with this implementation.) We must also handle overlapping intervals (e.g. if events occur in quick succession less than window apart, the timeline stays true continuously until window after the last one). So effectively, if events are at times tA, tB, … the timeline is true from tA to tA+window, and if tB occurs before tA+window ends, extend the timeline to tB+window, etc. We will implement that carefully (perhaps merging intervals).
- `DurationWhere(condition BoolTimeline) time.Duration` – sum up all interval lengths in the BoolTimeline. This is straightforward: iterate each interval and add `End - Start`. (We might choose the output type as float64 seconds or as a time.Duration. Internally using time.Duration is safer for precision. The API can format it as seconds or milliseconds for output as needed.)
- `AND/OR/NOT combinations`: We can implement these either as separate functions or by treating BoolTimelines as sets of intervals. For example, `AND` of two BoolTimelines can be computed by intersecting their true intervals (like an interval intersection problem). We might implement a helper that intersects two sorted interval lists. `OR` would be union of intervals, and `NOT` relative to a full span would be the complement within a time range. In our example, we only needed AND and NOT. We will implement AND for any number of conditions by pairwise intersection (e.g. intersect first two to get partial result, then intersect with third, etc.). If performance is a concern and number of conditions is large, we might optimize, but since typical queries have a handful of conditions, this is fine.

The above logic will be encapsulated in the `ComputeTimelineQuery` activity or in library functions that the QueryWorkflow can call. The implementation will be idiomatic (using slices and append for intervals, avoiding overly clever pointer arithmetic, etc.). Additionally, we will handle time equality carefully (if one interval ends exactly when another begins, in an AND we treat that boundary as exclusive so intervals that just touch might not overlap – these details will be documented and tested).

Workflow Coordination and Temporal Strategies

We outline specific Temporal usage strategies critical for a correct and maintainable implementation:

- **Workflow ID Strategy:** Use deterministic, informative workflow IDs:
- For timeline ingestion workflows, the ID will be `"timeline-{timelineID}"`, where `{timelineID}` is a client-provided identifier for the entity or session (e.g. a session GUID or user ID). This ensures idempotency (if the service receives the same event twice, it will signal the same workflow, not create duplicates) and exclusivity (Temporal will not start two workflows with the same ID concurrently).
- For query workflows, the service will generate a unique ID per query request, such as `"query-{timelineID}-{uuid}"` or a ULID. We include the timelineID in the ID for debugging clarity, though it's not strictly necessary. If queries are always started fresh, we could also use Temporal's **RunID** to track them and not worry about name collisions. But using a composite ID with a unique suffix and possibly a client-supplied part (client could include a request ID) allows easier traceability in logs (you can see which timeline a query was for by its ID). There is no need to reuse query workflow IDs; each is one-time.

- Child workflows (if used) can be given IDs that scope them under the parent's context (Temporal by default gives child workflows a new random ID if not specified, and tracks parent-child relationship). We may not need custom IDs for children unless we want to signal them directly or ensure idempotent behavior on retries. For now, we can allow Temporal to auto-generate child workflow IDs or use a prefix like `"query-{timelineID}-part-{N}"` if splitting queries.

- **Parent-Child Workflows:** While initial version might not heavily use child workflows, we have a plan for when to use them:

- *Ingestion Offload:* The ingestion workflow could start a child workflow for periodic heavy tasks. For example, if merging or compacting the timeline in S3 is needed (say, compress events into summary), the parent ingestion workflow could spawn a `CompactionWorkflow` to do it asynchronously, while the parent continues ingesting new events. The parent can proceed without waiting, or if it needs to ensure compaction finished, it can await or periodically check. Alternatively, this can be done via activities as well.
- *Cross-Timeline Queries:* If in future a query needs to span many timelines, a parent query workflow would orchestrate children per timeline. This keeps each child's execution time reasonable and leverages Temporal to manage parallelism. The parent would gather results.

- Benefits of child workflows in Temporal include isolation (each child can run on a potentially different task queue or set of workers specialized for that data type), and resilience (if one child fails, it can be retried independently, and the parent can proceed or handle partial failures). We will document scenarios where using a child workflow is preferred over a simple activity. For example, if an activity's work is long-running and could exceed Temporal activity execution limits (like hours), a child workflow might be more appropriate because workflows can continue indefinitely and do work in small increments. However, our timeline queries likely complete in seconds to minutes, so activities suffice.

- **Signals Usage:**

- We use **signals for event ingestion** as described. We will define a signal name, e.g. `"NewEvent"`, and the workflow will listen for it. We'll include fields for event payload. Since Go SDK allows direct binding of signal to a channel, our workflow code will look like:

```
signalCh := workflow.GetSignalChannel(ctx, "NewEvent")
for {
    var evt Event
    if ok := signalCh.Receive(ctx, &evt); !ok {
        // channel closed, maybe workflow terminating
        break
    }
    // Process event (append to S3, update state, etc.)
}
```

  The workflow remains running indefinitely (or until explicitly closed) and reacts to each event. We will also implement a mechanism to **gracefully close** the workflow if needed – perhaps via another special signal like `"CloseTimeline"` to indicate no more events will come (e.g. session ended). Upon receiving that, the workflow could perform any finalization (write a final state or mark as closed) and then complete. Otherwise, workflows might just time out or be terminated by an external call when obsolete.
- **Signal ordering:** Temporal does not strictly guarantee ordering of different signal channels, but signals on the same channel are FIFO. In our case we have one channel for new events, so the order of events is preserved as they were signaled. We assume clients send events in correct chronological order. If out-of-order events are possible, our workflow might need to buffer and sort before writing, or adjust logic (this adds complexity, so we'll document that we expect events per timeline to be timestamped in order or only slightly out of order such that the effect is negligible. Strict ordering can be enforced by the sending system if needed).
- We can also use signals for other things: e.g., a "stop query" signal if a client wants to cancel a long-running query (Temporal supports external cancellation). But for now, not required.

- If we implement asynchronous query completion, we might use a signal to notify an external listener when done. More likely, the client will just poll or the workflow result will be available via client request. We will not complicate with callback signals to client in this PRD.

- **Error Recovery:** Temporal's model provides automatic retries for activities. We will configure activities with appropriate retry options (exponential backoff, max attempts). For example, an S3 write might be retried a few times if network fails. If it continues to fail, the ingestion workflow can be programmed to respond by logging and perhaps signalling an alert (maybe via another activity to notify ops). But it will not drop events. Because of the nature of signals, if an ingestion workflow crashes or is restarted, any signals that were not processed remain in the queue and will be delivered when it resumes. This ensures at-least-once processing of events. We do have to be mindful of **idempotency**: if an activity succeeded (wrote to S3) but then the workflow crashed before marking it done, Temporal might replay the workflow (since workflows are re-executed from history after a crash). Our workflow code should be written to handle duplicate events or at-least not double-append them. One strategy: include a unique Event ID and when writing to S3, ensure we don't write an event that's already in storage. We could keep track of the last written event timestamp or an incremental sequence in the workflow state and ignore any duplicate on replay.

Temporal ensures the workflow replay sees the same signals in order, so by checking an event ID against the last processed ID, we can skip if it's a duplicate delivery due to retry. We will incorporate such checks to maintain exactly-once semantics on top of Temporal's at-least-once.

- **Temporal Testing:** We will leverage Temporal's test environment for most of our workflow tests. This environment allows us to simulate signals, time, and activity execution in code without a real Temporal server. We will use it to validate our signal handling logic and ensure no nondeterministic behavior (the Temporal test will actually error if our workflow uses non-deterministic patterns). For example, calling `time.Now()` in a workflow is illegal; we will avoid that or use `workflow.Now()` which Temporal can virtualize. Our tests will cover things like: sending two events in quick succession and ensuring both are processed, continuing-as-new triggers after N events (we can simulate N signals and see that the workflow indeed called ContinueAsNew and the new run resumes properly).

In summary, the workflow coordination is carefully designed to align with Temporal's strengths: long-running, stateful workflows for ingestion (akin to an **actor model** per timeline [27]), and short-lived on-demand workflows for queries. Signals provide event-driven inputs, and parent/child workflows (though optional at first) provide a path for scaling out queries.

Development & Testing Plan

Implementation Phases: We suggest implementing this project in phases to manage complexity: 1. *Timeline Library & Basic Tests:* Start by implementing the Timeline data types and operations as a pure library (`pkg/timeline`). Write unit tests for all operations (before integrating Temporal). This ensures the core logic is sound. This phase addresses the correctness of timeline algebra (LatestEventToState, HasExisted, etc.).
2. *Temporal Workflows Integration:* Implement the ingestion workflow and query workflow using the timeline library. Initially, use a simplified storage approach (even in-memory) to get the plumbing working. Write tests using Temporal's test suite to simulate sending signals and executing queries, verifying that the timeline library is invoked correctly and results match expectations. In this phase, also integrate slog logging in the workflows/activities.
3. *S3 Integration:* Implement the S3 storage layer and replace the dummy storage. Test the S3 activities in isolation (possibly using AWS SDK against a mock S3 service or using localstack for integration tests). Ensure that the S3 writes and reads behave as expected (e.g. appending events yields a correct file, reading returns all events). Since S3 is an external dependency, we won't use real S3 in automated tests (to avoid flakes and cost), but we will abstract it enough to simulate. For end-to-end, either use a local in-memory stub or one could use local filesystem as a stand-in for S3 (since the concept is similar: writing to a file per timeline).
4. *End-to-End Testing and Hardening:* Bring it all together by running the full system in a staging environment or using the test environment with all components. Simulate real usage patterns, e.g. a burst of events followed by a query. Measure performance for a large number of events to see if any part is a bottleneck and needs adjustment (e.g. maybe switch to batch writes if too slow). Add more tests for edge cases (no events then a query comes in, or query on a timeline with no matching events, etc., ensuring graceful handling).
5. *Documentation & Examples:* Prepare documentation for users of the API (how to format requests, what operations are available in queries) and for developers (how the system is structured). Also, ensure to document how to run tests and what the coverage results are, and how to deploy (particularly, setting up Temporal, S3 config).

Conclusion: By implementing this PRD, we will deliver a robust Go service that fully realizes the Timeline Framework abstraction for time-state analytics. The service will enable analysts or systems to define complex time-based metrics declaratively and obtain results without dealing with low-level event processing code. The combination of Go's performance and Temporal's reliability will ensure that the system scales with growing data volumes while remaining easy to reason about (each timeline is an isolated workflow actor [27]). With thorough testing in place (covering the academic examples and beyond), we will have confidence in the correctness of the timeline computations and the resilience of the orchestration. This PRD has outlined the architecture, data model, workflow design, and testing strategy in detail, so an engineer can proceed with implementation directly, without needing to refer to the research paper. All critical timeline concepts from the paper have been translated into concrete requirements and design elements in our system, bridging the gap from academic idea to production-ready service.

References: The design draws on the timeline concepts from Milner *et al.* CIDR'23 [9] [4] and leverages Temporal's workflow paradigm for implementation. The approach addresses the limitations of classical approaches [1] by using timeline-specific abstractions that simplify query logic and reduce bugs [6] [28]. The expected outcome is a highly maintainable and scalable system for time-state analytics that can be extended with new timeline operations or integrated into larger data platforms in the future.

---

[1] [2] [3] [4] [5] [6] [7] [14] [28] Time-State Analytics—Why Everyone Should Care - Conviva
https://www.conviva.com/blog/time-state-analytics-why-everyone-should-care/

[8] [9] [19] [20] [24] [25] Raising the Level of Abstraction for Time-State Analytics With the Timeline Framework
https://www.cidrdb.org/cidr2023/papers/p22-milner.pdf

[10] [11] [12] [13] [17] [18] [22] [23] [26] Raising The Level of Abstraction For Time State Analytics | PDF | Databases | System
https://www.scribd.com/document/727437469/Raising-the-Level-of-Abstraction-for-Time-State-Analytics

[15] Child workflows send a signal to the parent - Temporal Community
https://community.temporal.io/t/child-workflows-send-a-signal-to-the-parent/5580

[16] What is a good use case for a child workflow in Temporal or Uber ...
https://stackoverflow.com/questions/55840458/what-is-a-good-use-case-for-a-child-workflow-in-temporal-or-uber-cadence

[21] java - Starting Child workflow from Signal in Temporal - Stack Overflow
https://stackoverflow.com/questions/76578061/starting-child-workflow-from-signal-in-temporal

[27] Actors, Entities, and Temporal Workflows - Berlin Tech News from ...
https://news.siliconallee.com/2023/08/10/actors-entities-and-temporal-workflows/