# Golem Timeline

# TimeLine Execution with Golem Engine!

# Category of problems

- How long user spend re-buffering the video from CDN1, excluding ones within 5 seconds after a seek?
- Did the user swipe credit card twice within a span of 10 minutes from different suburbs?
- Duration of risky state before machine failed
- Lineage of bank card history correlated with transaction anomalies
- Did an iPhone user stop advancing in a game when the ad took ≥ 5 seconds?

## Solutions in various ways:

- Spark Streaming, Delta Lake
- Amazon timestream, Apache Flink and so on!
- Other Bespoke solutions (We built a Scala app for a major project)

# It is still hard!

- Complicated **State management** in a streaming context
- Extremely complicated **SQL queries**
- Vague boundaries of **stream vs batch**
- Suboptimal handling of **delayed events**
- Over reliance on **persistence** during computation
- **Distributed caching** being a second thought
- Underrated **in-memory** cache
- Zero application level **compute reuse**

# An example of a hard query

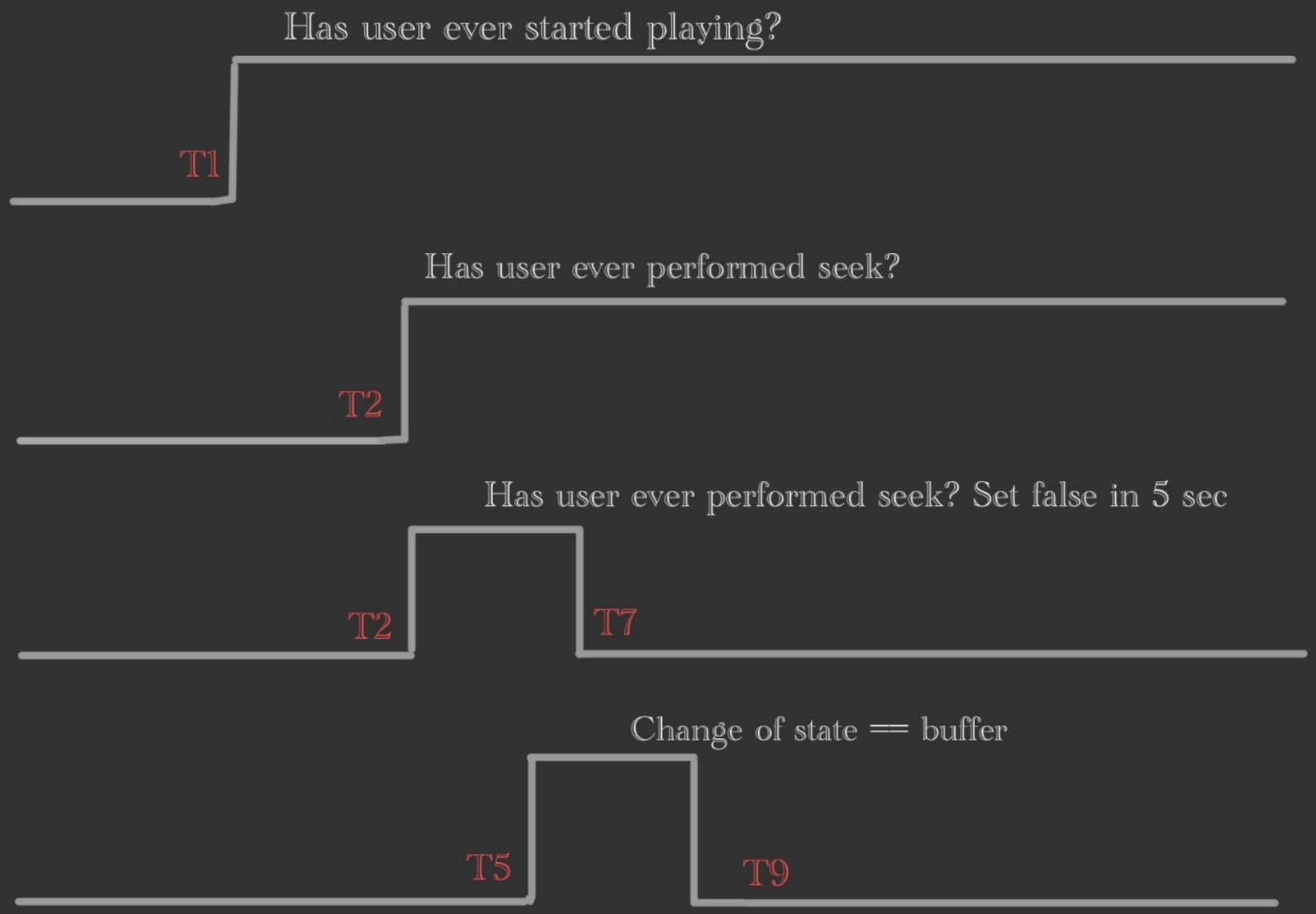
```
w = Window.partitionBy().orderBy("T")
seekAsPlayerState = heartbeats.filter("P is not null")
    .union(heartbeats.filter("A is not null")
        .withColumn("P", F.lit("Seek_st")))
    .union(heartbeats.filter("A is not null")
        .withColumn("T", col("T") + 5)
        .withColumn("P", F.lit("Seek_ed")))
    .select("T", "P")
ignoreBufBeforePlay = seekAsPlayerState.withColumn("H",
        F.max(when(col("P") == 'play', 1).otherwise(0)).over(w))
    .filter("H == True")
duringBuffer = ignoreBufBeforePlay.withColumn("DB",
        when(col("P") == "buffer", True)
        .when((F.col("newplayerState").contains("seek")), None)
        .otherwise(False))
    .withColumn("DB", last("DB", ignorenulls=True).over(w))
duringSeek = duringBuffer.withColumn("DS",
    (col("T") - F.max(when(col("P")=="Seek_st", col("T")).otherwise(0)).over
        ↪ (w)) < 5)
ignoreBufInSeek = duringSeek.withColumn("P",
        when((col("P") == "Seek_ed") & (col("DB") == True), "buffer")
        .otherwise(col("P")))
    .filter((col("P") != "buffer") | (~col("DS")))
    .select("T", "P")
CDN = heartbeats.select("T", "C").filter("C is not null")
queryPoints = spark.createDataFrame(["2022-07-22 10:05"], DateType()).toDF("
    ↪ T")
withCDNAndQuery = ignoreBufInSeek.unionByName(CDN, allowMissingColumns=
    ↪ True)
    .unionByName(queryPoints, allowMissingColumns=True)
intervals = withCDNAndQuery.withColumn("P", last("P", ignorenulls=True).
    ↪ over(w))
    .withColumn("C", last("C", ignorenulls=True).over(w))
    .withColumn("next_change_T", lead("T", 1).over(w))
    .withColumn("duration", col("next_change_T") - col("T"))
result = intervals.filter(col("C") == "CDN1")
    .filter("T < 2022-07-21 10:05")
    .filter(col("P") == "buffer")
    .agg(sum("duration").alias("cirDuration"))
```

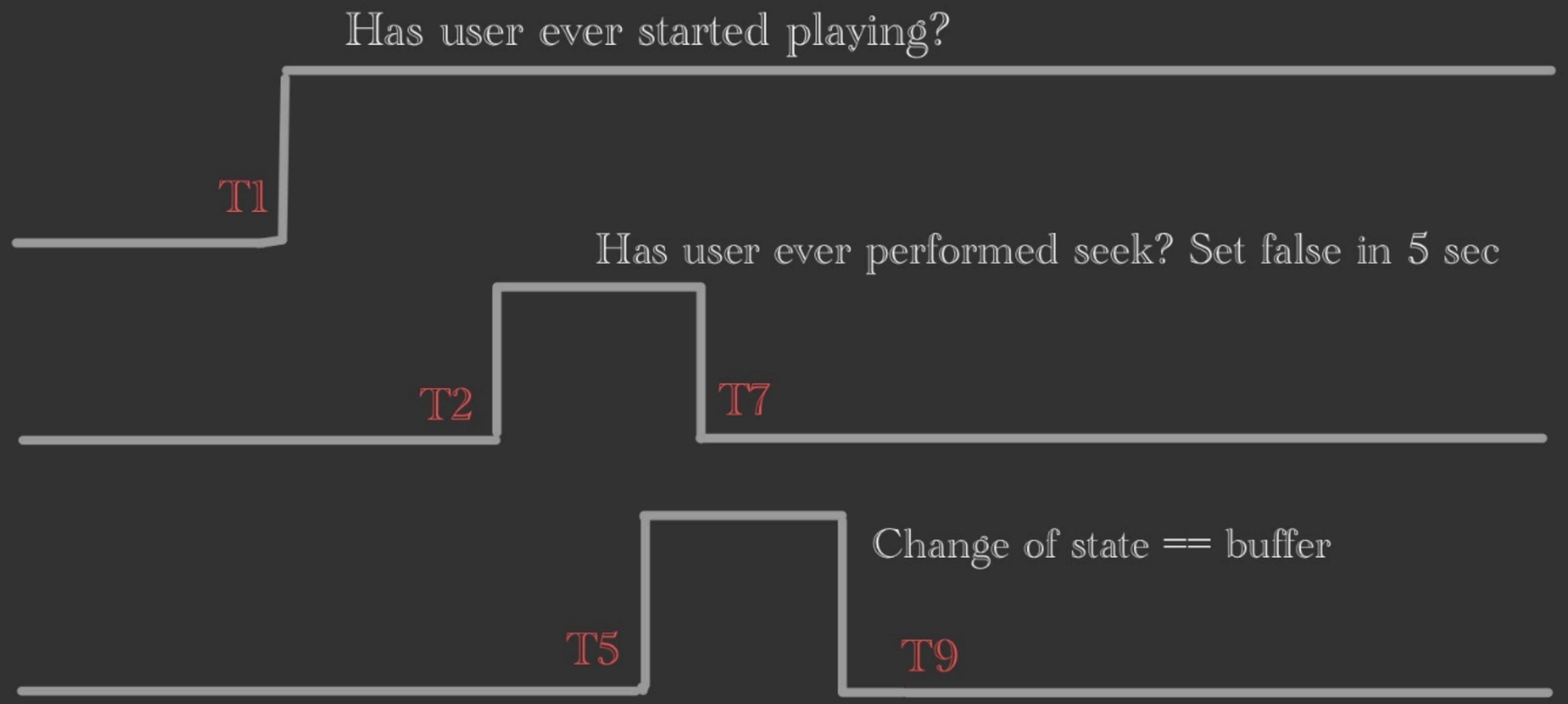
# On a high level

- We need a DSL backed by the right primitive, allowing composition, observability and optimisations
- We need an executor that's deterministic, and integrates well with the DSL
- An executor that's transparent to the developer, from a domain perspective
- Be able to come back and peek at these tasks/executors anytime

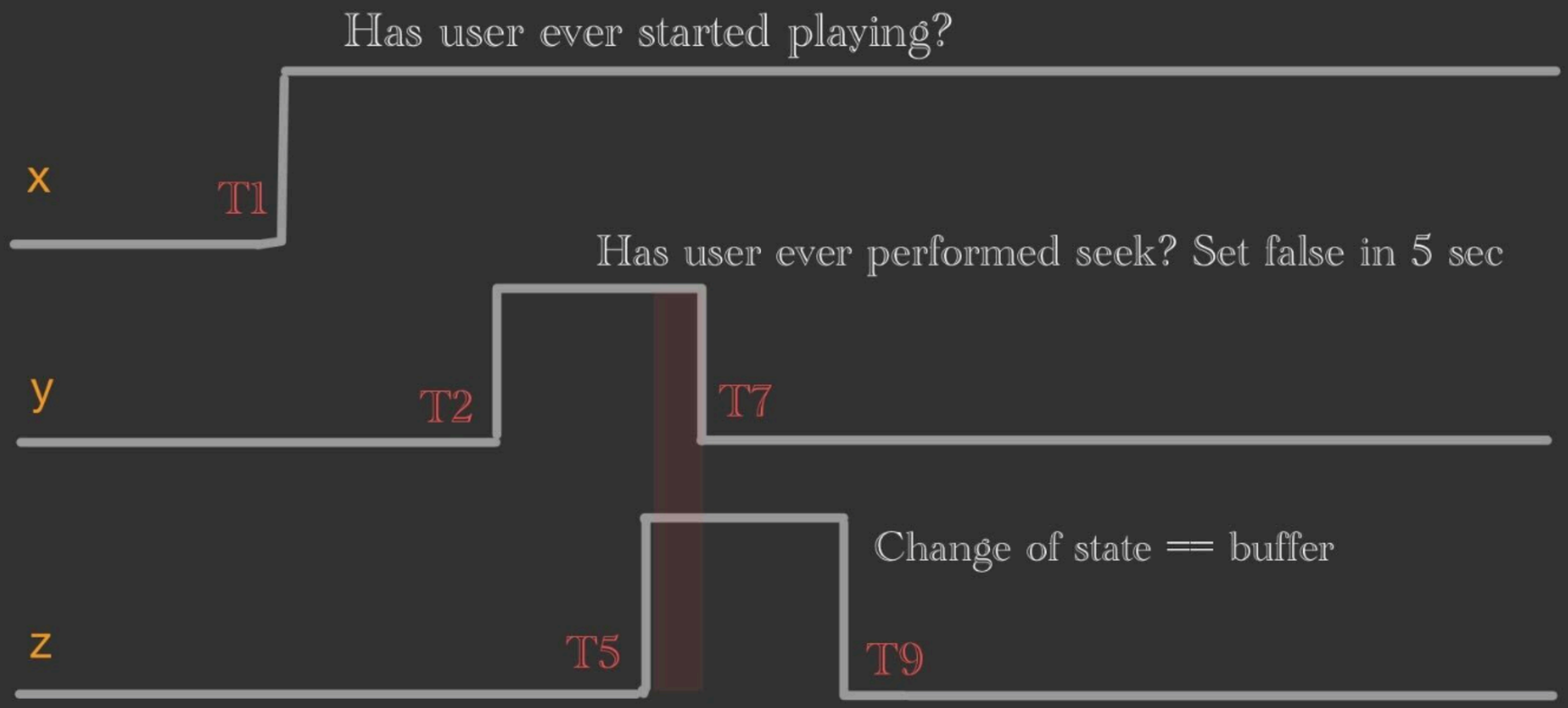
# What is a Timeline ?

https://www.cidrdb.org/cidr2023/papers/p22-milner.pdf

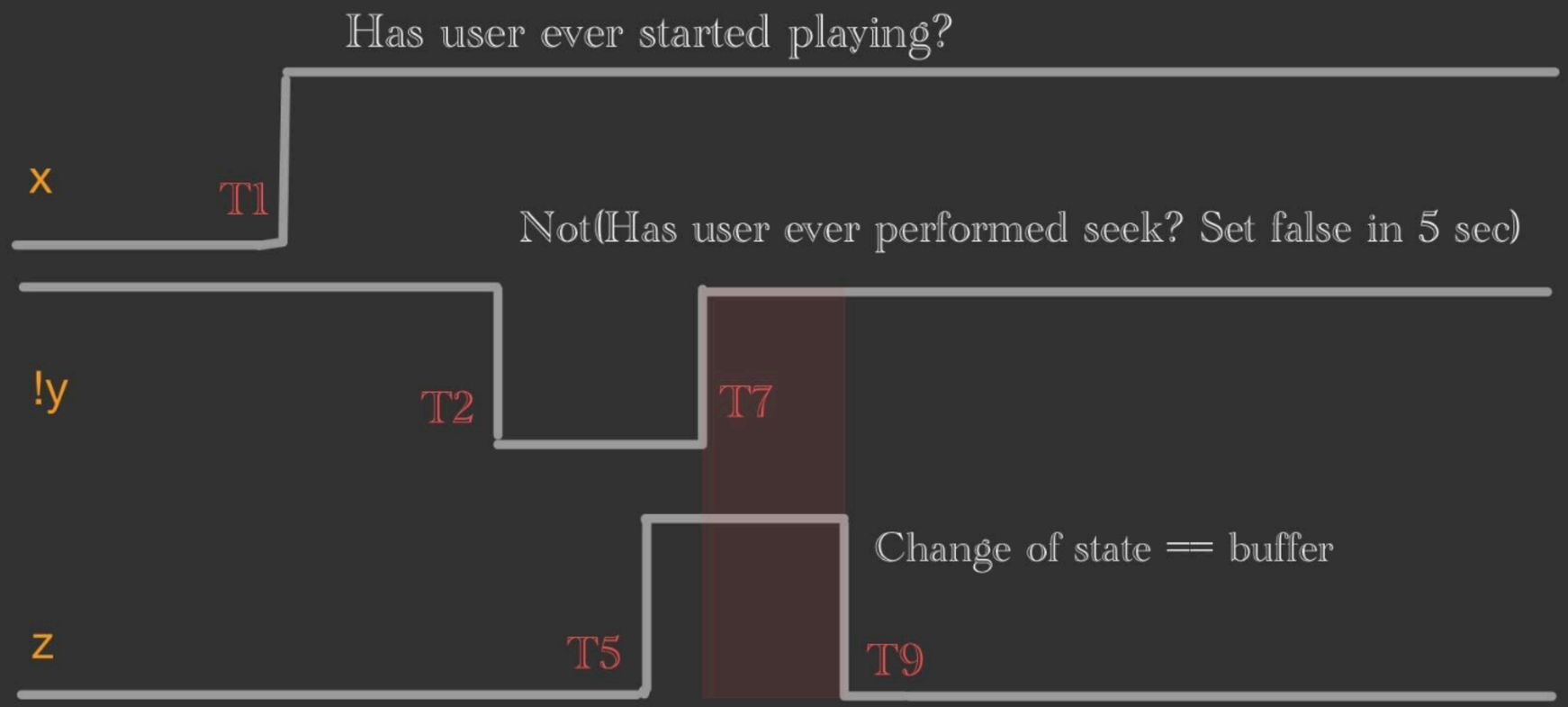

Has user ever started playing?
T1
Has user ever performed seek?
T2
Has user ever performed seek? Set false in 5 sec
T2
T7
Change of state == buffer
T5
T9

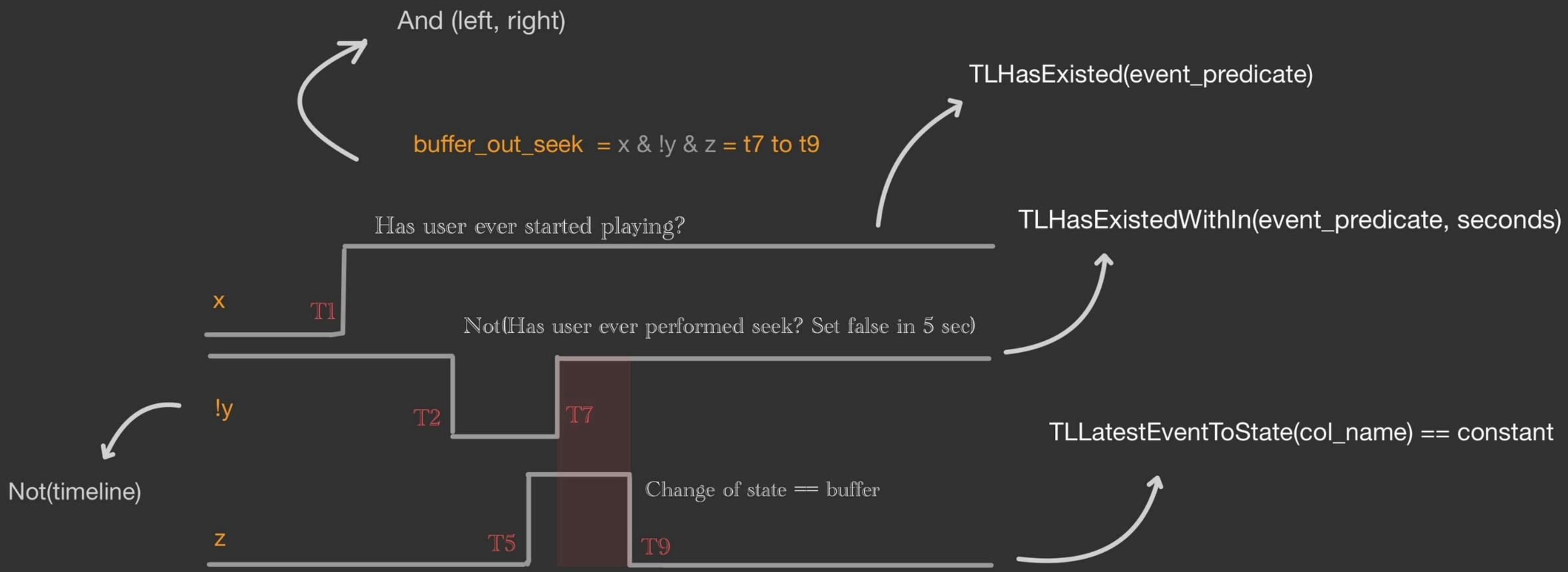

Has user ever started playing?
T1
Has user ever performed seek? Set false in 5 sec
T2
T7
Change of state == buffer
T5
T9

buffer_in_seek == x & y & z = t5 to t7



buffer_out_seek = x & !y & z = t7 to t9




And (left, right)
buffer_out_seek = x & !y & z = t7 to t9
TLHasExisted(event_predicate)
TLHasExistedWithIn(event_predicate, seconds)
TLLatestEventToState(col_name) == constant
Not(timeline)
Has user ever started playing?
x
T1
Not(Has user ever performed seek? Set false in 5 sec)
!y
T2
T7
Change of state == buffer
z
T5
T9

# Timeline DSL

```
t1: TLHasExisted(col("playerStateChange") == 'play')
t2: TLHasExistedWithin(col("userAction") == 'seek', 5)


t3: TLEventToState(col('playerStateChange')) == 'buffer'
t4: TLEventToState(col('cdnChange')) == 'CDN1'

result: And(And(And(t1, !t2), t3), t4)

result.at(2pm)
```

# Let's implement this with Golem

- Every Timeline DSL node is a Golem worker - That's long living, stateful and durable!
- A worker in a Golem is an instance of a web assembly component.

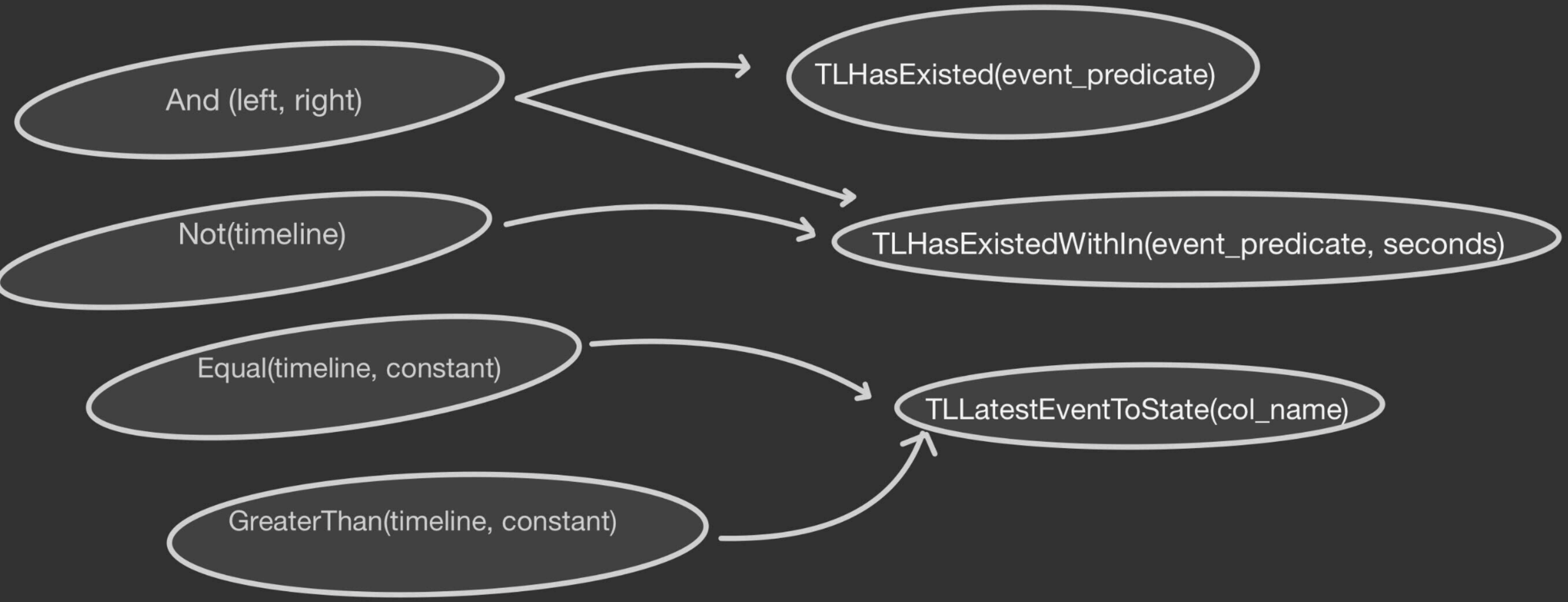
# Internal representation of Timeline DSL in Golem Timeline

```
pub enum TimeLineOp {
  EqualTo (WorkerDetail, Box<TimeLineOp>, GolemEventValue),
  ...
  Or(WorkerDetail, Box<TimeLineOp>, Box<TimeLineOp>),
  Not(WorkerDetail, Box<TimeLineOp>),
  TlHasExisted(WorkerDetail, GolemEventPredicate<GolemEventValue>),
  TlLatestEventToState(WorkerDetail, EventColumnName),
  ...
  TlDurationWhere(WorkerDetail, Box<TimeLineOp>)
}
```

We then segmented the Timeline nodes to be either `Leaf` or `Derived`

# THE DERIVED NODES (EXAMPLES)

# THE LEAF NODES



# Event Processor

- All leaf nodes are implemented into a web-assembly component, and we call it event-processor!
- They are called event processor because their input is an event timeline
- The output is timeline of states
- These three leaf node functions within 1 component module can run as one worker, or multiple workers, that's going to be configurable

# Timeline Processor

- We write another component called timeline-processor that implements rest of the derived nodes
- Every functions in this component takes another timeline as input
- They take the input from either another instance of timeline processor or event-processor
- These nodes may work as a single worker or multiple workers

Any state backing these functions are simple Rust datastructures.

No need to learn nuances of another framework

I was writing simple Rust program and build a WASM component to work with Golem

# Core Module - The orchestrator

We have a third component called `core` (the core engine) that parses the DSL that's coming from the driver

Core orchestrator assigns the work to the various workers and builds a real execution plan

It is another WASM component

# Driver

Driver itself is a web assembly component that is the starting point of the entire workflow.

# Instantiation and Computation

The entire workflow is divided into two parts.

- Instantiation
- Computation / Streaming / On-Demand Execution

# Instantiation

- Write the TimeLine DSL in Driver (as of now)
- Driver sends it to the core engine
- Core traverses through the timeline definition and instantiates and informs the workers of other workers
- Core returns to the driver of the execution plan that includes worker information

```
{
  "event_processors": [
    {
      "LeafTimeLine": {
        "TLHasExistedWithin": {
          "time_line_worker": {
            "component_id": "aa23e1a4-3384-43c1-8c33-7c74cb2ab2e5",
            "worker_id": "cirr-le2s-playerStateChange"
          }
        }
      }
    }
  ],
  "result_worker": {
    "DerivedTimeLine": {
      "Not": {
        "result_worker": {
          "component_id": "7fd082fa-0063-473e-8061-6fd5cca7a3ac",
          "worker_id": "cirr-tl-not-8b54ef0b-8814-4b3d-bb6f-3a91147a7a36"
        }
      }
    }
  }
}
```

# Computation Workflow

- Kick off event feeder - a simple Pulsar consumer sending the events to event processors
- The job keeps running. The event processors continuously stream.
- Invoke `get_timeline_result` in result_worker, returning the timeline value
- Every worker has the same function that can be called at anytime

# API Definition

```
path: /get-results?time={time},


binding: {
  component: timeline_processor_component_id,
  workerId: cirr-tl-not-95b2c1f2-670d-4d99-8c20-897a12dbd72a,
  functionName: "timeline:timeline-processor/api/get-timeline-result",
  functionParams: [${request.time}],
  response : ${ {body: match worker.response[0] { ok(value) => value, err(msg) => msg } }
}
```

# What does that mean?

You can instantly write a simple set of API definitions exposing a tree of subcomputation result (or current state) of every complex computation, that helps with debugging

# And we got the following!

# True Application Level Observability

At the core, the declarative DSL allows us to inspect what's going on, and with golem we also know which worker is taking care of which part of the computation too - forever!

# Example

Worker 1 is handling TLEventToState.

Worker2 is handling `Not(TLEventToState)`!

And worker3 is handling TLEventToState for some other event.

Now for `t1`, if `A & !B` is giving `None` probably it's because some events haven't reached yet for Worker3.

# Compute Reuse

- Every complex computation input simply invoke the `get_timeline_result` of workers handling their child DSL, resulting in automatic reuse of computations
- Timeline definition is associated with worker, so it can reuse the existing worker for the same definition

And that worker is never going to change!

Golem is scalable but doesn’t come up with the indeterminism of Scalability where in parts of computation keeps changing between tasks or executors.

# Handling Delayed Events For Free

- In all perspectives, delayed events are handled for free in Golem-Timeline
- If a worker needs to wait for another event for a month, or year, it can.
- Alternatively, you can poke at the `result_worker` another time to see if it now gives a different result

# Internal durable states

- Example: At the core every worker should have a complete knowledge of what every other worker
- All of this metadata, as well as data is just in-memory states. Why? Golem takes care of the rest.
- Golem-timeline currently use only in-memory for anything it needs to persist!

# Simply leading to In-Memory as the primary data source

- Persistence during computation to attack failures doesn't exist. Everything is in-memory as much as they can
- On demand computation mostly just rely on the information with in-memory by default
- It's durability and reliability are all handled by Golem

# Thank you!