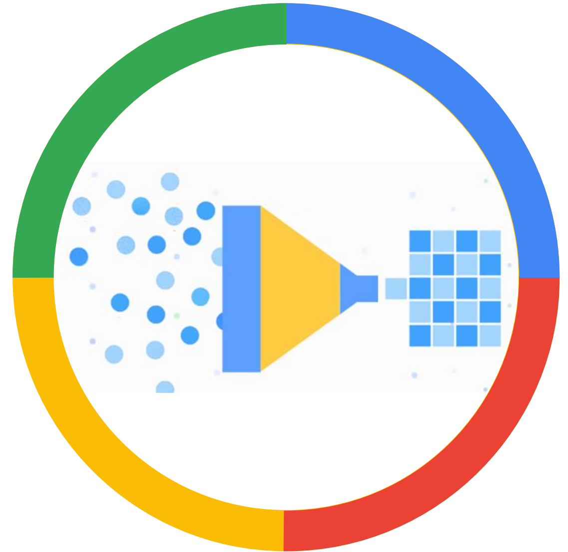
Breaking stormy ~~monolith~~ antipattern

Sept 2022



Google Cloud

Radek Stankiewicz
Strategic Cloud
Engineer
Google Cloud

Complex Event Processing

Drools

Topic → Read from Kafka — Events — Enrich — Rich Events → Alarm (per MSISDN) —Write→ Notifications

Refresh dictionaries

MDM

Read

Google Cloud

Complex Event Processing

Drools

Read

Topic

Read from Kafka

Events

Enrich

Rich Events

Alarm (per MSISDN)

Write

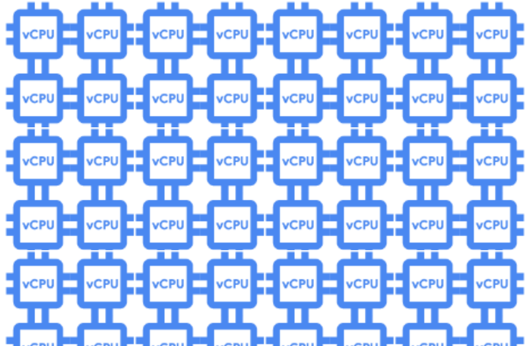
Notifications

LRU cache

Refresh dictionaries

MDM

Complex Event Processing



Google Cloud

Complex Event Processing

Drools

Read

Topic

Read from Kafka

Events

Enrich

Rich Events

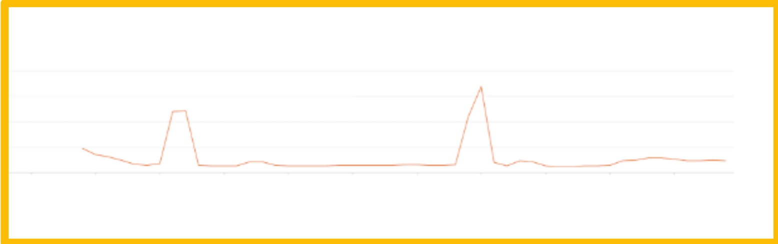
Alarm (per MSISDN)

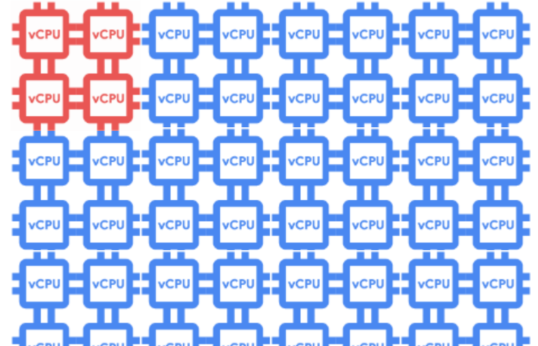
Write

Notifications

LRU cache

Refresh dictionaries





Lift & Shift requirements

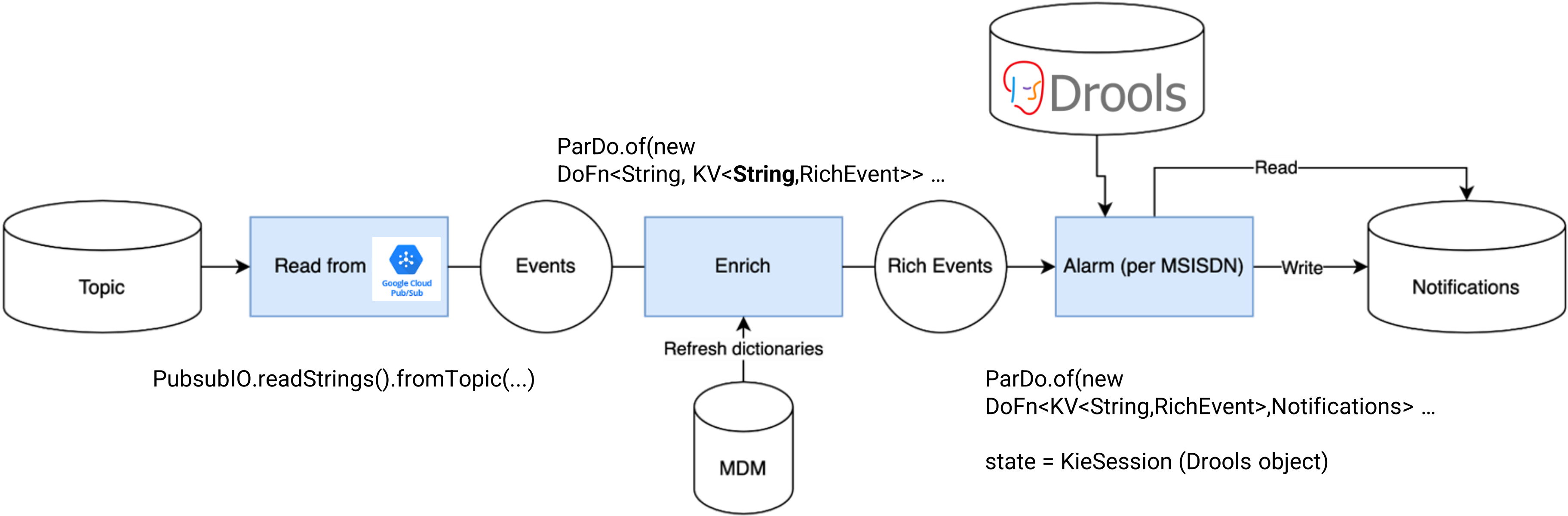
1. Maintain ~60GB of LRU cache in memory
2. Run parDo with Alarm leveraging cache
3. Refresh dictionaries periodically

It’s not cost effective to maintain worker with 48 cpu and 90GB RAM that is idle most of the time.

Lift & Shift requirements

1. Implement sticky session (per MSISDN) so cache is next to Alarm (drools engine) so the pipeline can scale

Dataflow - first iteration with PSO



Issues:

- Slow
- Won't drain

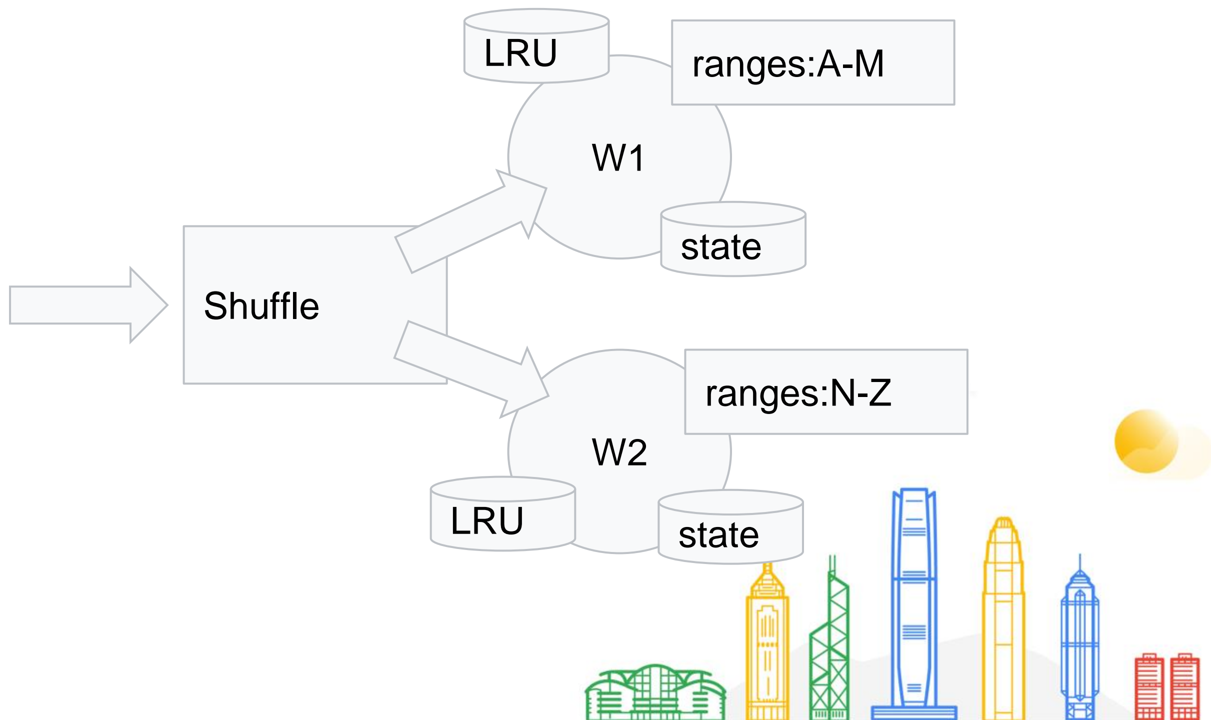
Dataflow - alarm

```
@StateId(KNOWLEDGE_STATE)
                    private final StateSpec<ValueState<KieSession>> knowledgeSpec=
StateSpecs.value(new DroolsStateCoder(getBase()));


public void processElement(
       ProcessContext context,
       @StateId(KNOWLEDGE_STATE) ValueState<KieSession> state) {
         KV<String,Event> element = context.element();
         KieSession ksession = state.read();
         if (ksession == null)  ksession = base.newKieSession();

         ksession.setGlobal("LOGGER", LOG);
         ksession.setGlobal("bigtableClient", cbtClient);
         ksession.insert( element.getValue());
         ksession.fireAllRules();

         state.write(ksession);
...
```

Google Cloud

Why sticky session works?

Bundle Work items:
KV(A, event),
KV(B,..),
KV(C,..),
KV(A),
KV(D),
KV(Z,),
KV(A,)

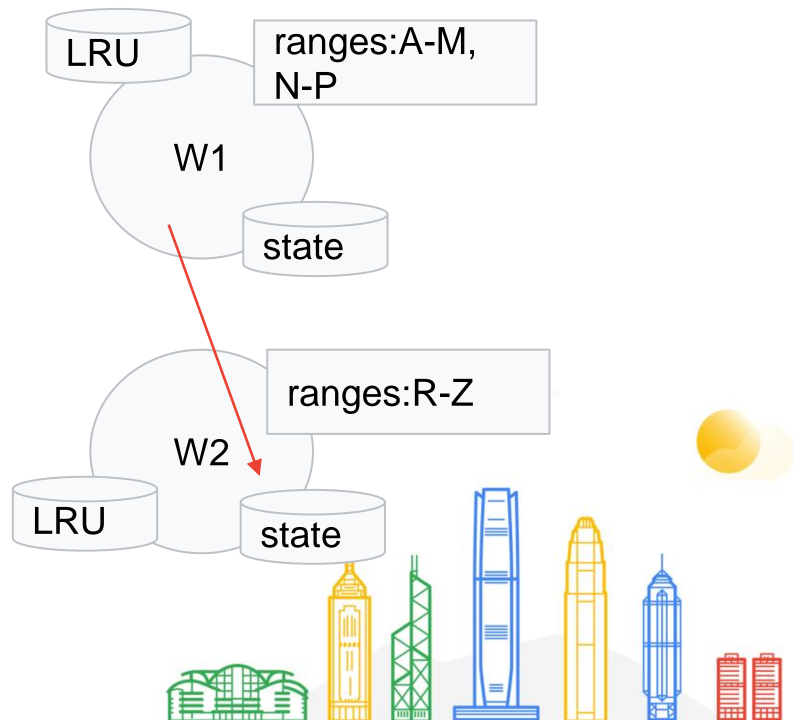


Why sticky session works?

State is handled by shuffle appliance or Streaming Engine

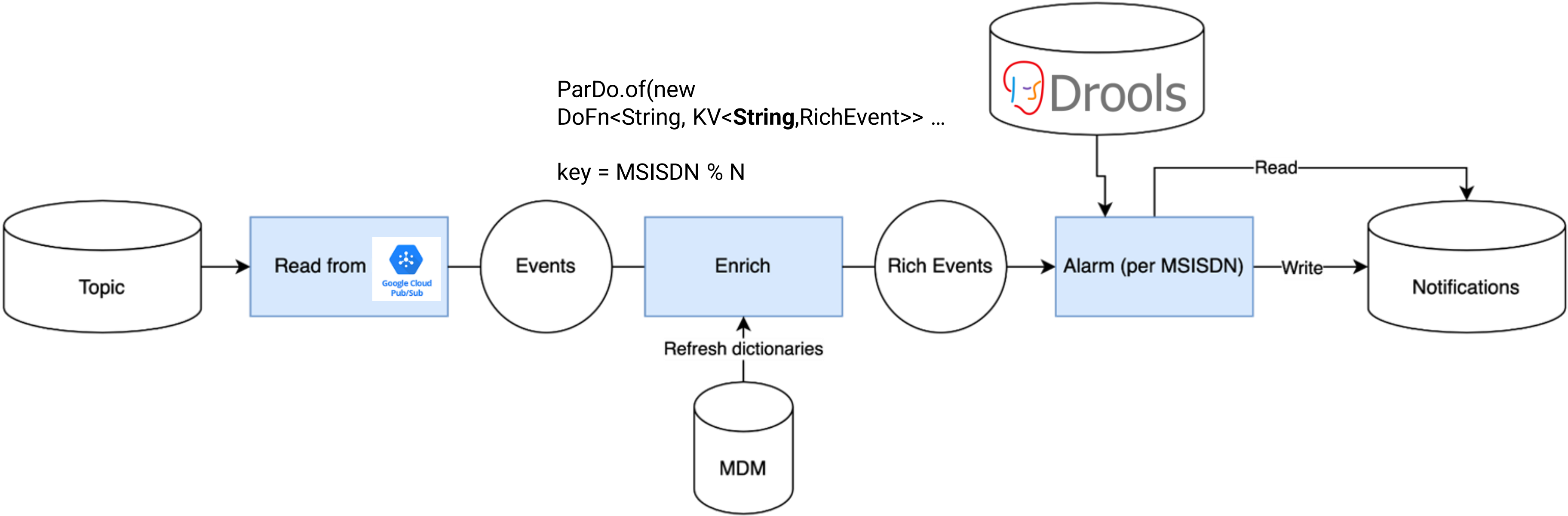
Shuffle Appliance implementation is based on levelDB (https://github.com/google/leveldb)

By design it was meant to by “Sticky”



Google Cloud

Dataflow - what customer did to improve speed



Outcome:

- Fast
- Still won't drain

Google Cloud

What they should do - AlwaysFetched

```
@ProcessElement
public void processElement(
      ProcessContext context,
      @AlwaysFetched @StateId(KNOWLEDGE_STATE) ValueState<KieSession>
state){
…
state.read()
…
```

Dataflow - second iteration

Drools

Read

Topic → Read from Google Cloud Pub/Sub — Events — Enrich — Rich Events → Alarm (per MSISDN) —Write→ Notifications

Refresh dictionaries

MDM

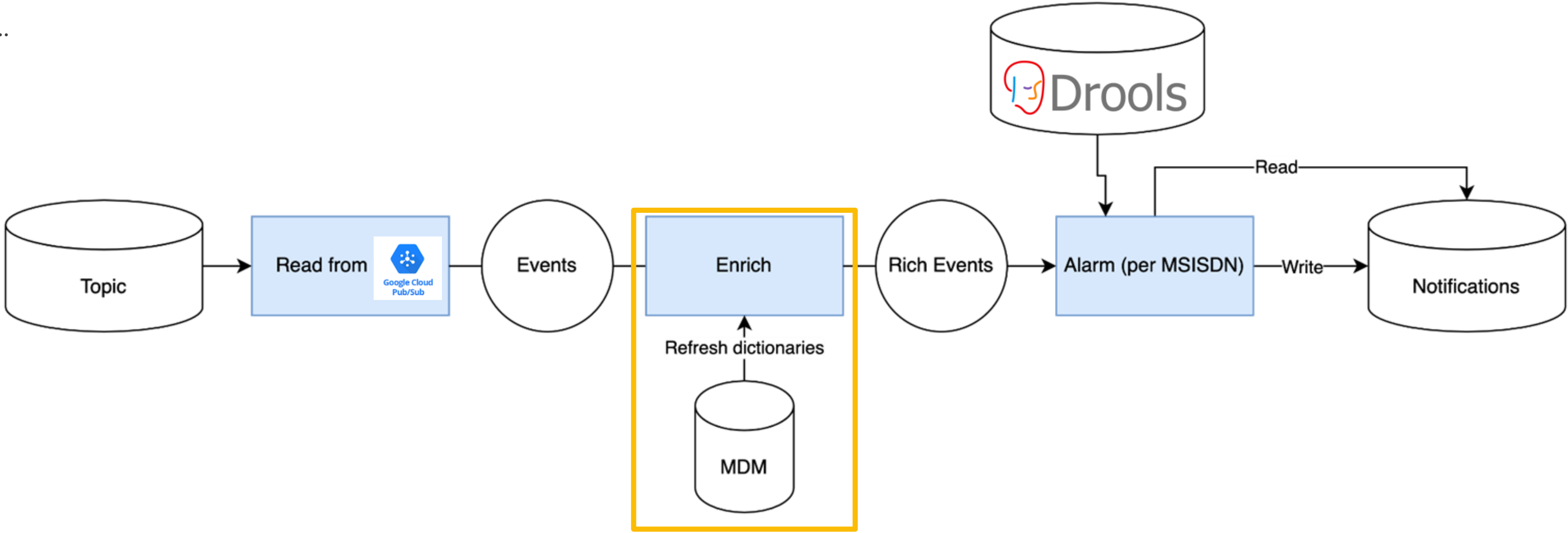
```
@ProcessElement
public void processElement(
      ProcessContext context,
      @AlwaysFetched @StateId(KNOWLEDGE_STATE)
ValueState<KieSession> state) …

@OnTimer("timer") public void onTimer(
@StateId(KNOWLEDGE_STATE) ValueState<KieSession>
state …
      state.clear()
```

Outcome:

- Fast!
- Drains!

Dataflow - dictionaries

...



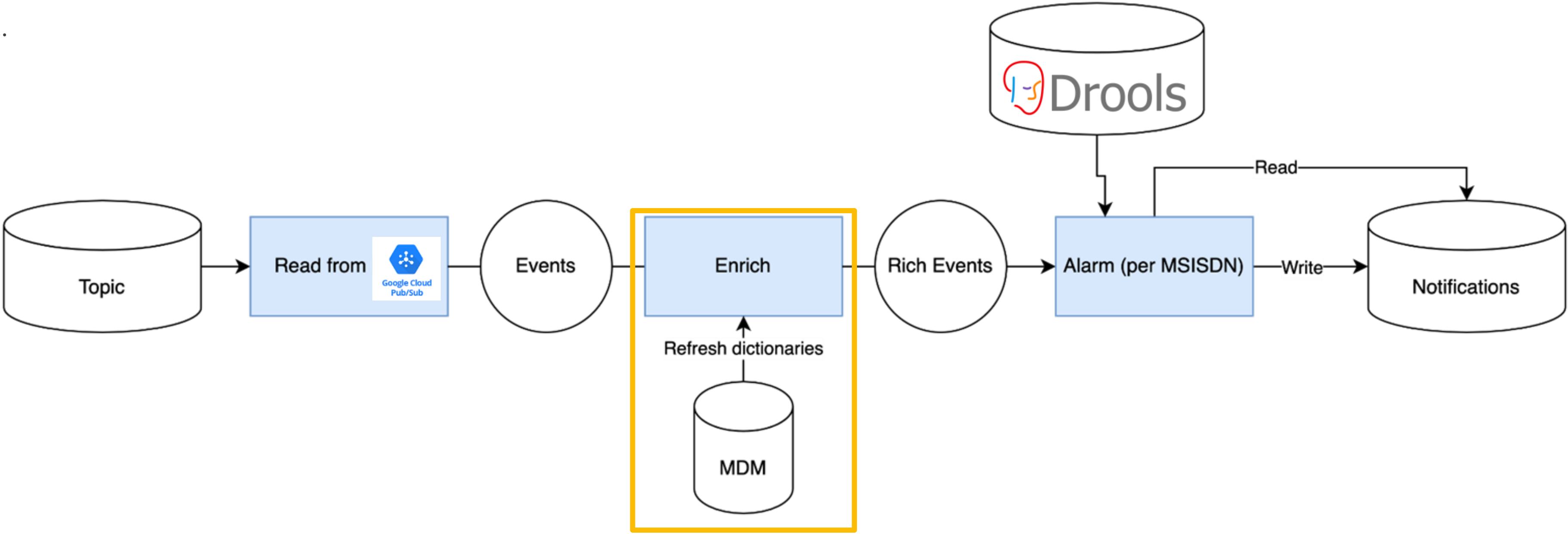
Dataflow - dictionaries

```
input.apply("Impulse", GenerateSequence.from(0).withRate(1, refreshDuration))
.apply(Window.<Long>into(new GlobalWindows())
.triggering(Repeatedly.forever(AfterProcessingTime.pastFirstElementInPane()))
                     .discardingFiredPanes())
 .apply(
     "Load dicts",
        ParDo.of(
            new DoFn<Long, Map<String, String>>() {
               @ProcessElement
               public void process(
               @Element Long element, OutputReceiver<Map<String, String>> o) {
                        // load here
                     }}))
            .setCoder(coder).apply(View.asSingleton());
```

Google Cloud

Dataflow - dictionaries

...



Outcome:

- It works!
- “Eventually” refreshes

New requirement - ordered processing

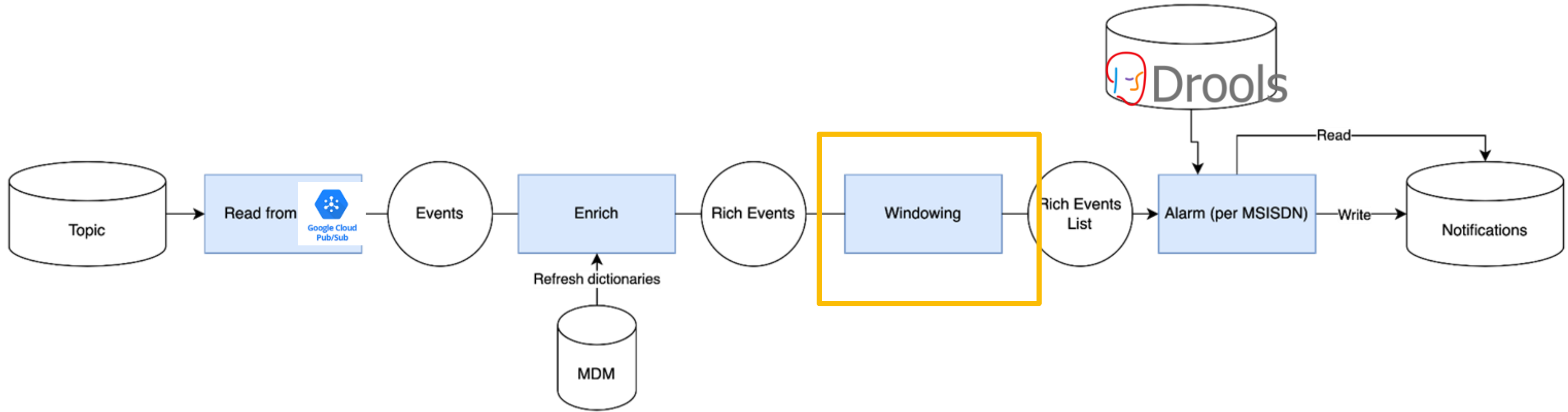
On-prem - Kafka topic partitioned on MSISDN (modulo). Storm kept order

Dataflow + PubSub

- Lack of support for ordering in PubSub
- Stateful processing can’t guarantee ordering within bundle due to shuffle
- Maybe windowing to order? But it will break state.

Google Cloud

Dataflow - faking windowing to implement ordering



Google Cloud

Dataflow - faking windowing - what does it mean?

1. Support for Global window
2. Build mix of session and fixed window:
    a. “Close window” and emit items after X minutes
    b. Start timer after first occurrence of event
    c. Buffer items for same key, sort them

Google Cloud

Dataflow - faking windowing

```
static class EmulateWindowing
        extends DoFn<KV<String,Event>,KV<String, List<Event>>> {

    @StateId("state") private final StateSpec<BagState<KV<String,Event>>> numElements = StateSpecs.bag();
    @TimerId("timer") private final TimerSpec timer = TimerSpecs.timer(TimeDomain.EVENT_TIME);
    @StateId("isTimerSet") private final StateSpec<ValueState<Boolean>> isTimerSet = StateSpecs.value();
```

Dataflow - faking windowing

```
@ProcessElement
        public void processElement(
            ProcessContext context,
            @StateId("state") BagState<KV<String,Event>> state,
            @StateId("isTimerSet") ValueState<Boolean> isTimerSetState,
            @TimerId("timer") Timer timer) {
            KV<String, Event> message = context.element();
            state.add(message);
            if (!MoreObjects.firstNonNull(isTimerSetState.read(), false)) {
                timer.offset(Duration.standardMinutes(5)).setRelative();
                isTimerSetState.write(true);
             }
        }
```

Dataflow - faking windowing

```
@OnTimer("timer") public void onTimer(
            @TimerId("timer") Timer timer,
            @StateId("state") BagState<KV<String, String>> elementsState,
            @StateId("isTimerSet") ValueState<Boolean> isTimerSetState,
            OnTimerContext context) {

            Iterable<KV<String, String>> read = elementsState.read();
            Iterator<KV<String, String>> iterator = read.iterator();
            List<String> output = new ArrayList<>();
            String key = null;
            while(iterator.hasNext()){
                KV<String, String> next = iterator.next();
                if(key==null) key = next.getKey();
                output.add(next.getValue());
            }
            elementsState.clear();
            isTimerSetState.clear();
            context.output(KV.of(key, output.sort()));// sort it properly:)
         }
```

Dataflow - alarm

```
public void processElement(
        ProcessContext context,
        @AlwaysFetched @StateId(KNOWLEDGE_STATE) ValueState<KieSession> state) {
          KieSession ksession = state.read();
          KV<String,List<String>> elements = context.element();
          if (ksession == null)  ksession = base.newKieSession();
          for(Event event : elements.getValue()){

               ksession.setGlobal("LOGGER", LOG);
               ksession.setGlobal("bigtableClient", cbtClient);
               ksession.insert( event );
               ksession.fireAllRules();
           }
          state.write(ksession);

…
```

Final requirements vs implementation

Feature	Solution/recommendation
Maintain state for X days, sticky cache	Stateful processing with global window and Timer to garbage collect
Performant, per key, state	@AlwaysFetched annotation
Dictionary for enrichment	Side input refreshed with timer
Ordered processing	Stateful buffer function with timer
Persist drools session	Customer coder for state

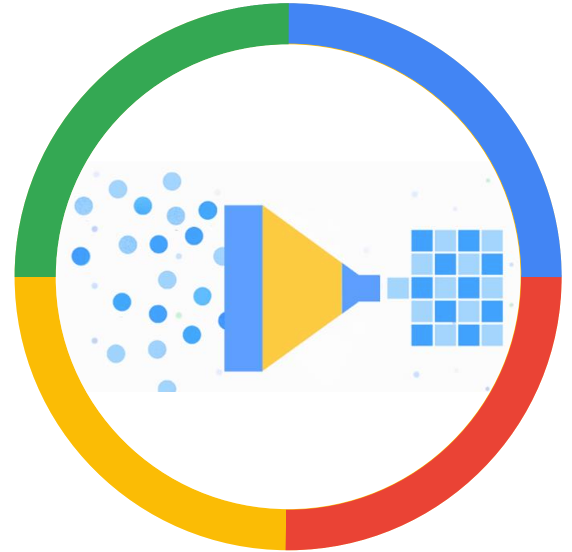
Google Cloud

Summary

1. Original implementation was antipattern
2. Final requirements were clear
3. Stateful processing with timers is super flexible
4. It works really well with all features:
   a. Autoscaling
   b. Streaming engine
   c. Apache Beam, Dataflow
   d. Buzz buzz cloud native buzzwords, customer like it

Thank you!

Kudos: lhr@ - thank you!



Google Cloud

Hiring?

Contact me: radoslaws@google.com or find me on LinkedIn