A monadic DSL for Distributed Persistent Processes (workflows)

(Inspiration from Unix processes)

JUSPAY

HELLO!

I am Rahul Korthiwada

I work on systems engineering and building frameworks at Juspay.

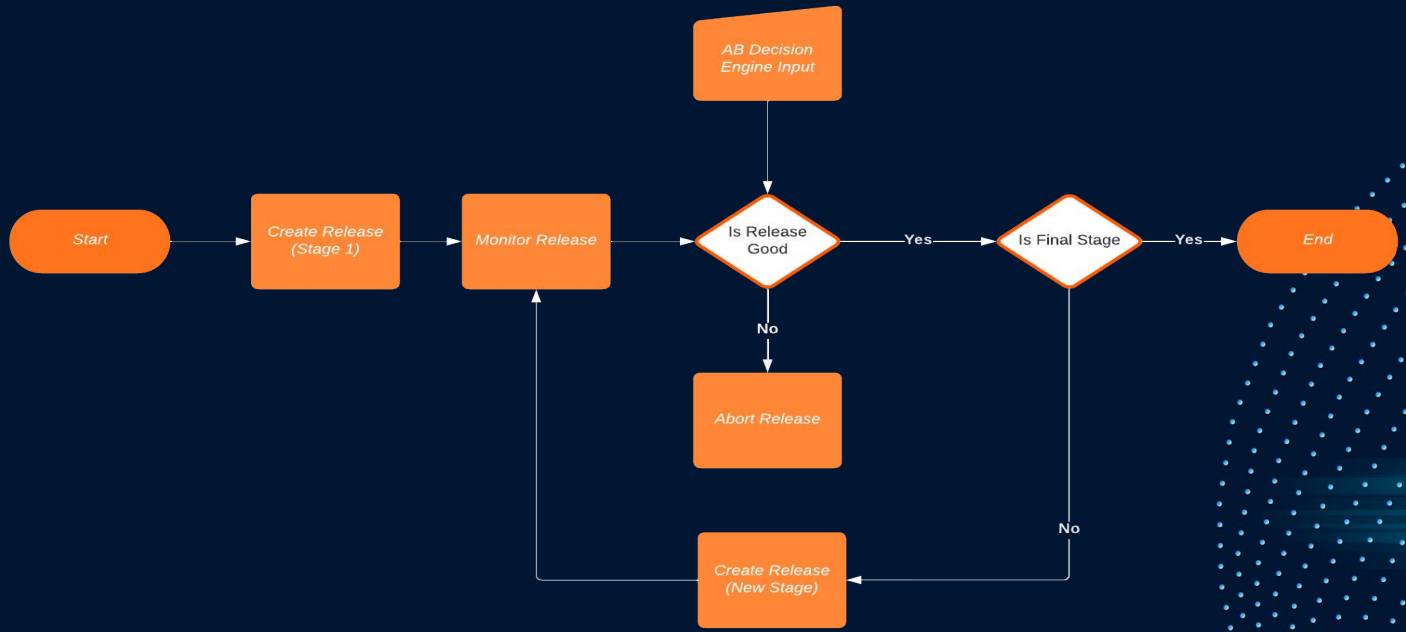
You can find me at
@rahulKorthiwada

- About Juspay: 1M LoC of Haskell code. 500+ fresher engineers learning and using Haskell everyday! We chose Haskell is to build DSLs like what Alan Kay talks about in his vision for programming & scaling.
- Problem: Workflows (Distributed Persistent processes) is a complex but pervasive need across varied use cases in Juspay. Can a Haskell DSL really simplify this?

JUSPAY

Let’s take an example problem

- Building a “custom CI/CD” release and monitoring pipeline for our SDK release that should be pushed to 300M devices
- Ensure Reliability, Visibility & Agility in evolution of this system.



JUSPAY

The old code (a typical state machine)

```
startBusinessFlow :: RT.ReleaseTracker -> L.Flow Common.ConsumerResp
startBusinessFlow releaseTracker =
   case releaseTracker ^. RT.event_status of
       ES.START                -> startFlow releaseTracker
       ES.WAITING_FOR_AB_PODS -> waitingForABFlow releaseTracker
       ES.STAGGERING           -> monitoringFlow releaseTracker
       ES.STAGGERING_FULL      -> staggeringFullFlow releaseTracker
       ES.POST_MONITORING      -> postMonitoringFlow releaseTracker
       ES.STABILIZE            -> stabilizationFlow releaseTracker
       ES.DONE                 -> doneFlow releaseTracker
       ES.ABORTING             -> abortReleaseFlow releaseTracker Nothing
       ES.ABORTED              -> Logger.logAPIDebug "Release Tracker already aborted"
                                    releaseId $> Common.ConsumerResp
                                                 { releaseId = releaseId,
                                                   execStatus = "RELEASE_ABORTED" }
       _                       -> Logger.logAPIDebug "Not a Proper Business Status"

   releaseId $> Common.ConsumerResp {releaseId = releaseId, execStatus =
                                     "RELEASE_NOT_IN_CONSUMER_FLOW"}
   where
      releaseId = releaseTracker ^. RT.release_id
```

Issues

- **Difficult to reason about as there is no top down structure.**
- **Too flexible and prone to bugs**
- **More effort to implement and maintain.**

JUSPAY

The Ideal system - Goals

Learn from the Linux process interface and design a DSL for the use cases of a **distributed, long running process**.

DSL Needs:

- **Wait for a timeout / external event (for hours/days)**
- **Loops & retries with checkpoints (running across machines)**
- **Reliable Exception handling (system and business failures)**
- **Fork & Join (schedule new child processes across hosts)**

Hide the complexity of distributed execution of the process but still provide good debuggability and traceability.

JUSPAY

How it should be

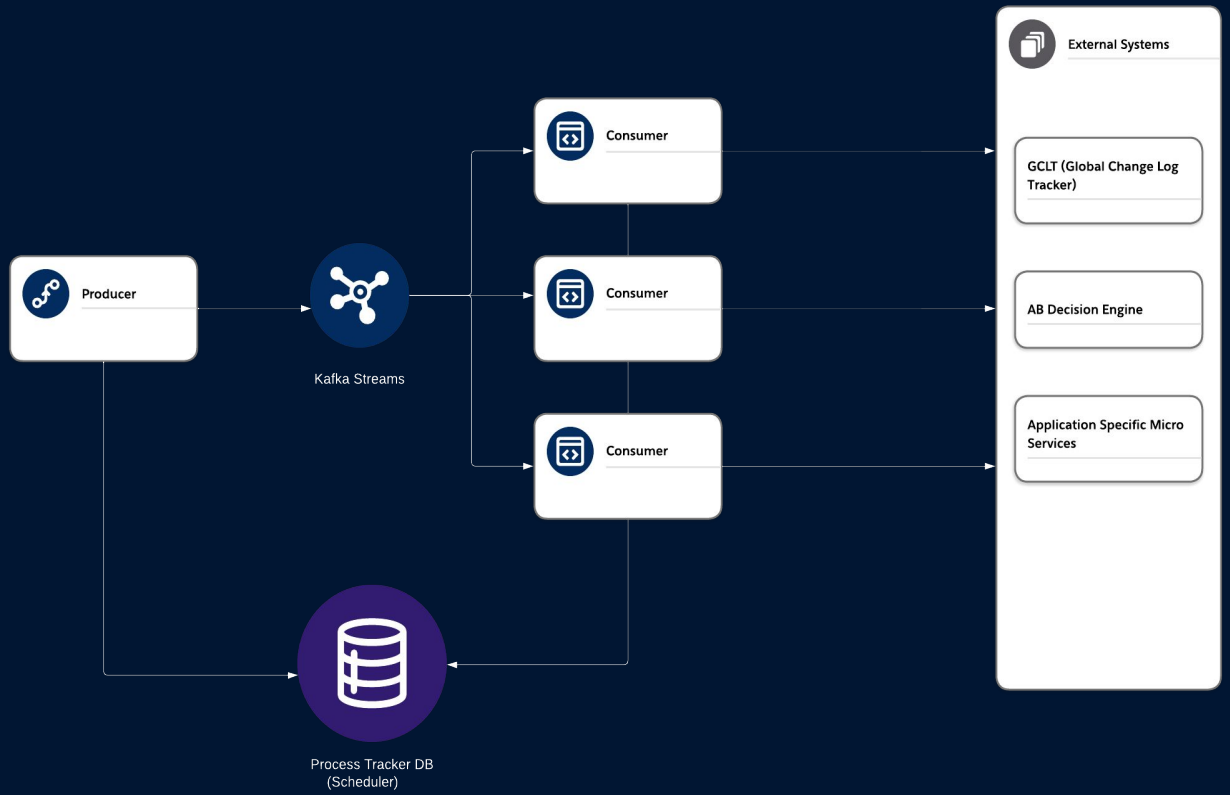
```
liveWorkflow ::  Deployment -> RT.ReleaseTracker -> L.Flow ConsumerResp
liveWorkflow dpmt rt = runExceptT do

        cpr ES.START startAB

        cpr ES.WAITING_FOR_AB_PODS checkABPodStatus

        cpr ES.START_RELEASE startRelease

        cpr ES.STAGGERING increaseStagger

        cpr ES.STAGGERING_FULL startPostReleaseAB

        cpr ES.POST_MONITORING moveToStabilize

        cpr ES.STABILIZE stabilize

        cpr ES.DONE done
```

JUSPAY

How did we implement this.. (monad)

- Use Monad's Computation Box to Store the features you want..
- For Checkpointing…
  - ❏ We need a State Monad for storing the computed value of each statement
  - ❏ And a Getter / Skip Function to verify if the instruction is executed or not and return the computed value.
- For Global Error Handling.. Use a Except Monad..

How did we implement this.. (systems)



JUSPAY

State management using Checkpoints

```
data Recorded s m a = Recorded
       { inner :: StateT s m a
       , getter :: s -> Maybe a }
```

- s -> State which stores the computed value of each step..
- inner -> instruction to be executed… (over state monad.. So that we can store the computed values info in state)
- getter -> A Getter Function.. Which takes the precomputed state and returns the computed value of the instruction.. If it is already executed.. (Maybe a)...

JUSPAY

Retries, Parallel flows, Join flows etc.

```
type WorkFlow m e = Except Me (Recorded s m)

    s :- State for Storing each flow
    e :- Error Type
    m :- Your Base Monad
```

- Instruction Level Checkpointing
- Skipping Executed Instructions in a Distributed Environment
- Multiple Iterations of a Single Instruction..
- Generic error handler

JUSPAY

Workflow Retries-Schedules

```
continueIf
 :: (Monad m, Monoid e)
 => (s -> Bool)
 -> WorkFlow s m e ()
continueIf predicate =
 lift get >>= guard . predicate
```

```
elsePerform
 :: (MonadIO m, ToJSON s)
 => (s -> Maybe a)
 -> StateT s m a
 -> Recorded s m a
elsePerform =
 flip (recordedWithPersisted
noStatePersist)
 where
 noStatePersist = const (pure ())
```

```
scheduleAfter
 :: (MonadIO m, ToJSON a)
 => NominalDiffTime
 -> WorkFlow (PTState a) m e ()
scheduleAfter after =
 lift $
   const Nothing `elsePerform`
changeScheduleTime
 where
   changeScheduleTime =
     nextScheduleTime <~ (Just . addUTCTime
after <$> liftIO getCurrentTime)
```

Workflow Looping

```
loopEveryUntil
 :: (MonadIO m, ToJSON a, Hashable a, FromJSON a)
 => NominalDiffTime
 -> (b -> Bool) --Should it be a (State -> Bool) or (b -> Bool) ?
 -> Maybe Int -- max iterations
 -> WorkFlow (PTState a) m e b
 -> WorkFlow (PTState a) m (WorkFlowError e) b
loopEveryUntil after predicate maybeMaxRetries wf =
 do
   b <- withExceptT DomainErr wf
   if predicate b -- done
     then do
       retryingStatus .= Nothing
       pure b
     else do
       maybe
         (scheduleAfter after) -- No limit on retries. Seriously ??
         (\maxRetries -> do
            retriesDone <- gets (fromMaybe 0 . preview (retryingStatus._Just))
            if (retriesDone <= maxRetries)
              then do
                retryingStatus .= Just (retriesDone + 1)
                scheduleAfter after
              else
                gets (view (domainState.re _JSON')) >>= except . Left . PTError . MaxRetriesReachedWith)
         maybeMaxRetries
       empty
```

Reliable Exception Handling

```
handleError :: ErrorType -> L.Flow Response
handleError = \case
   DB_ERROR -> scheduleAfter5mins *> pure err500Response
   INTERNAL_ERROR -> pauseRelease *> pure err500Response
   MJOS_ERROR -> pauseRelease *> pure err500Response
   ..
```

Future direction

- Distributed tracing for debugging
- Increase availability and scaling with Kafka integration
- Build helper libraries for DevOps use cases - for kubernetes, envoy releases etc.
- Migrate older payment workflows to this system & test at scale (our existing workflows process millions of txns / day)
- Refactor and open source it

JUSPAY

Q & A ?

JUSPAY

Thank you