Stream processing with R in AWS
AWR, AWR.KMS, AWR.Kinesis (R packages) used in ECS

Gergely Daroczi
@daroczig
March 7, 2017
About me

Gergely Daroczi (@daroczig)

Stream processing using AWR

github.com/cardcorp/AWR
About me

3,633 contributions in the year before last

Lead R Developer

3,470 contributions in the last year

Director of Analytics
CARD.com’s View of the World

Gergely Daróczy @daroczig · Apr 11

Just received my "I ♥ R" prepaid debit card from @CARD. Will be fun to use this #rstats designed card at #user2015 :)

10 RETWEETS  16 FAVORITES
Modern Marketing at CARD.com

Artwork & Brands

Bank Partner

CARD.COM AdTech Platform

APIs

RTB Ad Xchgs

Visitors

deploy

CARD.COM Site / App

learn

CARD.COM Analytics Platform

1

2

3

Transactions

Members

github.com/cardcorp/AWR
Further Data Partners

- card transaction processors
- card manufacturers
- CIP/KYC service providers
- online ad platforms
- remarketing networks
- licensing partners
- communication engines
- others
Why not Hadoop instead of MySQL?

Infrastructure

The goal of Rapporter was to provide a front-end to R in all modern browsers running on various platforms [A] – let it be a desktop, a notebook, tablet or a mobile phone. Users can access their data, reports and statistical tools stored in the Rapporter cloud from any place over the Internet and even co-operate collaboratively with other fellows and contributors.

A minor but useful part of the infrastructure is hosted at Zendesk that provides an extensible knowledge base and support forum [B]. All the requests and data packets sent by the clients to Rapporter servers hit our Content Delivery Network provider [C] first that would return all static content of the webpage cached at several locations around the world for improved response times. The CDN also operates as a front-line firewall and filters out some unwanted and potentially dangerous packets and queries [D] beside minimizing the risk of (Distributed) Denial-of service attacks. Users can optionally use Rapporter over a secure channel by HTTPS protocol [E], as the data transmitted to and from CloudFlare is encrypted on demand for improved security.

The dynamic content is mainly served by our Ruby on Rails [F] workers in the means of a cluster of thin servers [G] running inside of our private internal network. This content management system is made of several shared threads replying to user request via a load balancing reverse proxy [H] that also serves static content, plus JavaScript, CSS and image assets.

Although we try to do our best with deploying working code on production servers, we also collect possible Rails error messages with Errbit[H].

Another major part of our setup is the HAProxy cluster [I] of Apache based Reworkers [J] running within an enforced AppArmor [K] profile and optional AppArmor hats based on user privileges. This latter Linux kernel security module ensures that the users could not directly touch the disks or make connections to our databases – even if some malicious code would somehow escape our in-house developed sandbox called sandboxR. The dynamic hat option allows fine-grained control over the hardware resources on a per-user basis in the means of e.g. CPU power and memory limit, or network access. Please see some further security considerations below.

As we are using R for creating complex one-time and temporary reports via a Graphical User Interface or the recently introduced Application Programming Interface called Raplications, our home-made internal R functions do not deal with any statistical problems, but rather provides an environment for the users to easily implement those. Rapporter is basically made of our open-source rapt and pander packages (please see below) beside the above described Rails front-end and hardened security tools, and the data, methods and results all bundled in various JSON driven databases [M].

All our servers are running Ubuntu LTS [N] on 64 bit with a decent amount of memory and CPU cores optionally dedicated to VIP customers, and continuously monitored 24 hours a day, 7 days a week via the public [S] Pingdom availability monitor [O] and a more detailed and technical, enterprise-class monitoring solution with thousands of metrics, called Zabbix [P] – beside Google Analytics of course.

Data storage [M]

Although administering and maintaining several similar database engines might not make much sense in most setup, we use two NoSQL databases for improved performance. CouchDB is awesome for its disk-based B-tree views, simple attachment concept and eventual consistency scheme, while MongoDB makes the Rails models a lot more convenient to work with. GlusterFS is a network filesystem that stores R generated images on a replicated and optionally distributed storage attached to the highly available Rails servers.
Why R?

Gergely Daroczi (@daroczig)
Why Amazon Kinesis?

Source: Kinesis Product Details
source: Kinesis Developer Guide
Source: AWS re:Invent 2013
Deep Learning

Capture & submit streaming data to Firehose

Firehose loads streaming data continuously into S3 and Redshift

Analyze streaming data using your favorite BI tools
Deep Learning

Capture & submit streaming data to Firehose

Firehose loads streaming data continuously into S3 and Redshift

Analyze streaming data using your favorite BI tools

Streaming Data
- PutRecord()
- PutRecordBatch()
- Kinesis Agent

Amazon Kinesis Firehose

Amazon S3

Amazon Redshift

Gergely Daroczi (@daroczig)
Stream processing using AWR
github.com/cardcorp/AWR
> x <- 3.14
> attr(x, 'class') <- 'standard'

> print.standard <- function(x, ...) {
+   ## SLA
+   if (runif(1) * 100 > 99.9) {
+         Sys.sleep(20)
+   }
+   futile.logger::flog.info(x)
+ }

> while (TRUE) print(x)
INFO [2017-03-03 22:27:57] 3.14
INFO [2017-03-03 22:27:57] 3.14
S4: Multiple Dispatch

- AWS Lambda
- Amazon EC2
- Streams
- Amazon EC2
- Firehose
- Amazon S3
- Amazon Redshift
How to Communicate with Kinesis

Writing data to the stream:

- Amazon Kinesis Streams API, SDK
- Amazon Kinesis Producer Library (KPL) from Java
- flume-kinesis
- Amazon Kinesis Agent

Reading data from the stream:

- Amazon Kinesis Streams API, SDK
- Amazon Kinesis Client Library (KCL) from Java, Node.js, .NET, Python, Ruby

Managing streams:

- Amazon Kinesis Streams API (!)
Now We Need an R Client!

```r
> library(rJava)
> .jinit(classpath = list.files('~/.Projects/AWR/inst/java/', full.names = TRUE))

> kc <- .jnew('com.amazonaws.services.kinesis.AmazonKinesisClient')
> kc$setEndpoint('kinesis.us-west-2.amazonaws.com', 'kinesis', 'us-west-2')

> sir <- .jnew('com.amazonaws.services.kinesis.model.GetShardIteratorRequest')
> sir$setStreamName('test_kinesis')
> sir$setShardId(.jnew('java/lang/String', '0'))
> sir$setShardIteratorType('TRIM_HORIZON')
> iterator <- kc$getShardIterator(sir)$getShardIterator()

> grr <- .jnew('com.amazonaws.services.kinesis.model.GetRecordsRequest')
> grr$setShardIterator(iterator)
> kc$getRecords(grr)$getRecords()
[1] "Java-Object{"[SequenceNumber: 495628941604494433215346371084313572324361665031176210,
ApproximateArrivalTimestamp: Tue Jun 14 09:40:19 CEST 2016,
Data: java.nio.HeapByteBuffer[pos=0 lim=6 cap=6],PartitionKey: 42}]"

> sapply(kc$getRecords(grr)$getRecords(),
+ function(x)
+ rawToChar(x$getData()$array()))
[1] "foobar"
```
Managing Shards via the Java SDK

Let’s merge two shards:

```java
> ms <- .jnew('com.amazonaws.services.kinesis.model.MergeShardsRequest')
> ms$setShardToMerge('shardId-000000000000')
> ms$setAdjacentShardToMerge('shardId-000000000001')
> ms$setStreamName('test_kinesis')
> kc$mergeShards(ms)
```

What do we have now?

```java
> kc$describeStream(StreamName = 'test_kinesis')$getStreamDescription()$getShards()
[1] "Java-Object{
{ShardId: shardId-000000000000,HashKeyRange: {StartingHashKey: 0,EndingHashKey: 1701411834604692317316873037158841057278,SequenceNumberRange: {
StartingSequenceNumber: 49562894160427143586954815717376297430913467927668719618,
EndingSequenceNumber: 4956289416043829395954081028945856364232263390243848194}},
{ShardId: shardId-000000000001,HashKeyRange: {StartingHashKey: 170141183460469231731687303715884105728,EndingHashKey: 34028236692093846346337460743176826,SequenceNumberRange: {
StartingSequenceNumber: 495628941604494433215334634051783314918611628917470050,
EndingSequenceNumber: 495628941604594704752611652087392082504911751749828626}},
{ShardId: shardId-000000000002,
ParentShardId: shardId-000000000000,AdjacentParentShardId: shardId-000000000001,
HashKeyRange: {StartingHashKey: 0,EndingHashKey: 34028236692093846346337460743176826,SequenceNumberRange: {
StartingSequenceNumber: 49562904991497673099704924344727019529,}}")
```

Gergely Daroczi (@daroczig)
Amazon Kinesis Client Library

- An easy-to-use programming model for processing data

```java
java -cp amazon-kinesis-client-1.7.3.jar \
com.amazonaws.services.kinesis.multilang.MultiLangDaemon \
app.properties
```

- **Scalable** and **fault-tolerant** processing (checkpointing via DynamoDB)
- Logging and metrics in CloudWatch
- The **MultiLangDaemon** spawns processes written in any language, communication happens via JSON messages sent over stdin/stdout
- Only a few events/methods to care about in the consumer application:
  1. initialize
  2. processRecords
  3. checkpoint
  4. shutdown
Messages from the KCL

1. **initialize:**
   - Perform initialization steps
   - Write “status” message to indicate you are done
   - Begin reading line from STDIN to receive next action

2. **processRecords:**
   - Perform processing tasks (you may write a checkpoint message at any time)
   - Write “status” message to STDOUT to indicate you are done.
   - Begin reading line from STDIN to receive next action

3. **shutdown:**
   - Perform shutdown tasks (you may write a checkpoint message at any time)
   - Write “status” message to STDOUT to indicate you are done.
   - Begin reading line from STDIN to receive next action

4. **checkpoint:**
   - Decide whether to checkpoint again based on whether there is an error or not.
Again: Why R?

Gergely Daroczi (@daroczig)

Stream processing using AWR

github.com/cardcorp/AWR
R Script Interacting with KCL

```r
#!/usr/bin/r -i

while (TRUE) {

    ## read and parse JSON messages
    line <- fromJSON(readLines(n = 1))

    ## nothing to do unless we receive records to process
    if (line$action == 'processRecords') {

        ## process each record
        lapply(line$records, function(r) {

            business_logic(fromJSON(rawToChar(base64_dec(r$data))))
            cat(toJSON(list(action = 'checkpoint', checkpoint = r$sequenceNumber)))
        })
    }

    ## return response in JSON
    cat(toJSON(list(action = 'status', responseFor = line$action)))
}
```
#!/usr/bin/r -i

while (TRUE) {

  ## read and parse JSON messages
  line <- fromJSON(readLines(n = 1))

  ## nothing to do unless we receive records to process
  if (line$action == 'processRecords') {

    ## process each record
    lapply(line$records, function(r) {

      business_logic(fromJSON(rawToChar(base64_dec(r$data))))
      cat(toJSON(list(action = 'checkpoint', checkpoint = r$sequenceNumber)))

    })
  }

  ## return response in JSON
  cat(toJSON(list(action = 'status', responseFor = line$action)))
}

Gergely Daroczi (@daroczig)  
Stream processing using AWR  
github.com/cardcorp/AWR
> `install.packages('AWR.Kinesis')`
also installing the dependency ‘AWR’

trying URL 'https://cloud.r-project.org/src/contrib/AWR_1.11.89.tar.gz'
Content type 'application/x-gzip' length 3125 bytes

trying URL 'https://cloud.r-project.org/src/contrib/AWR.Kinesis_1.7.3.tar.gz'
Content type 'application/x-gzip' length 3091459 bytes (2.9 MB)

* installing *source* package ‘AWR’ ...
** testing if installed package can be loaded
trying URL 'https://gitlab.com/cardcorp/AWR/repository/archive.zip?ref=1.11.89'
downloaded 58.9 MB
* DONE (AWR)

* installing *source* package ‘AWR.Kinesis’ ...
* DONE (AWR.Kinesis)
Add content to the boilerplate

Business logic coded in R (demo_app.R):

```r
library(AWR.Kinesis)
kinesis_consumer(processRecords = function(records) {
    flog.info(jsonlite::toJSON(records))
})
```

Note: This is not something you should run in RStudio.

Gergely Daroczi (@daroczig)
Add content to the boilerplate

Business logic coded in R (demo_app.R):

```r
library(AWR.Kinesis)
kinesis_consumer(processRecords = function(records) {
    flog.info(jsonlite::toJSON(records))
})
```

Note

This is not something you should run in RStudio.
Add content to the boilerplate

Business logic coded in R (demo_app.R):

```r
library(AWR.Kinesis)
kinesis_consumer(processRecords = function(records) {
    flog.info(jsonlite::toJSON(records))
})
```

Config file for the MultiLangDaemon (demo_app.properties):

```ini
executableName = ./demo_app.R
streamName = demo_stream
applicationName = demo_app
```

Start the MultiLangDaemon:

```
/usr/bin/java -cp AWR/java/*:AWR.Kinesis/java/*:./ \
    com.amazonaws.services.kinesis.multilang.MultiLangDaemon \
    ./demo_app.properties
```
library(futile.logger)
library(AWR.Kinesis)

kinesis_consumer()

initialize = function()
  flog.info('Hello'),

processRecords = function(records)
  flog.info(paste('Received', nrow(records), 'records from Kinesis')),

shutdown = function()
  flog.info('Bye'),

updater = list(
  list(1, function()
    flog.info('Updating some data every minute')),
  list(1/60*10, function()
    flog.info(paste('This is a high frequency updater call',
                     'running every 10 seconds'))),

checkpointing = 1,
logfile = '/logs/logger.log')
Let’s run it locally!

**Note**

In theory you could, but this is not something you should run in RStudio.

1. Create a Kinesis Stream
2. Create an IAM user with DynamoDB and Kinesis permissions
3. Write data to the Stream
4. Run the MultiLangDaemon referencing the properties file
Let's run it locally!

**Note**

In theory you could, but this is not something you should run in RStudio.

1. Create a Kinesis Stream
2. Create an IAM user with DynamoDB and Kinesis permissions
3. Write data to the Stream
4. Run the MultiLangDaemon referencing the properties file
Create a Kinesis Stream

Amazon Kinesis

Amazon Kinesis services make it easier to work with real-time streaming data in the AWS Cloud.

Amazon Kinesis Firehose
Continuously deliver streaming data to Amazon S3, Amazon Redshift, and Amazon Elasticsearch Service.

Amazon Kinesis Analytics
Analyze streaming data from Amazon Kinesis Firehose and Amazon Kinesis Streams in real-time using SQL.

Amazon Kinesis Streams
Collect and stream data for ordered, replayable, real-time processing.

Learn more about Firehose
Learn more about Analytics
Learn more about Streams
Create a Kinesis Stream

Stream processing using AWR

Gergely Daroczi (@daroczig)
Check the Kinesis Stream

Amazon Kinesis
- Streams
- Firehose
- Analytics

test-AWR

Monitoring
- Time range: Last hour
- Statistic: Average
- Period: 1 minute

- Get Records (bytes)
- Get Records (records/second)
- Get Records (success count)
- Get Records Iterator Age (ms)

Incoming Data (bytes)

Gergely Daroczi (@daroczig)
Stream processing using AWR
github.com/cardcorp/AWR
Create an IAM user
library(rJava)
.jcall("java/lang/System", "S", "setProperty", "aws.profile", "personal")

library(AWR.Kinesis)
library(jsonlite)
library(futile.logger)
library(nycflights13)

while (TRUE) {

## pick a ~car~flight
flight <- flights[sample(1:nrow(flights), 1), ]

## prr <- .jnew('com.amazonaws.services.kinesis.model.PutRecordRequest')
## prr$setStreamName('test1')
## prr$setData(J('java.nio.ByteBuffer')$wrap(.jbyte(charToRaw(toJSON(car)))))
## prr$setPartitionKey(rownames(car))
## kc$putRecord(prr)

res <- kinesis_put_record(stream = 'test-AWR', region = 'us-east-1',
                           data = toJSON(flight), partitionKey = flight$dest)

flog.info(paste('Pushed a new flight to Kinesis:', res$sequenceNumber))
}

Gergely Daroczi (@daroczig)
Stream processing using AWR
github.com/cardcorp/AWR
Write Data to the Stream from R

```
library(java)
.jcall(java/lang/System, "%s", new Property[1], "my.profile", "personal")
library(AWR_Kinesis); library(jsonlite); library(julielogger); library(nycflights13)
while (TRUE) {
  flight <- flights[sample(1:nrow(flights), 1)]
  res <- kinesis.putRecord(test-AWR, region = "us-east-1", data = toJSON(flight),
                          partitionKey = flight$dest)
  log.info(paste("Pushed a new flight to Kinesis", res$sequenceNumber))
```

---

## get an iterator

```r
sir <- .jnew('com.amazonaws.services.kinesis.model.GetShardIteratorRequest')
sir$setStreamName('test-AWR')
sir$setShardId(.jnew('java/lang/String', '0'))
sir$setShardIteratorType('TRIM_HORIZON')
kc <- .jnew('com.amazonaws.services.kinesis.AmazonKinesisClient')
kc$setEndpoint('kinesis.us-east-1.amazonaws.com')
iterator <- kc$getShardIterator(sir)$getShardIterator()
```

## get records

```r
grr <- .jnew('com.amazonaws.services.kinesis.model.GetRecordsRequest')
grr$setShardIterator(iterator)
records <- kc$getRecords(grr)$getRecords()
```

## transform to string

```r
json <- sapply(records, function(x)
    rawToChar(x$getData()$array()))
```

## decode JSON

```r
json[1]
fromJSON(json[1])
bindlist(lapply(json, fromJSON))
```
Running the MultiLangDaemon locally

Gergely Daroczi (@daroczig)

Stream processing using AWR

github.com/cardcorp/AWR
library(futile.logger)
library(AWR.Kinesis)

kinesis_consumer(

    initialize   = function()
        flog.info('Hello'),

    processRecords = function(records)
        flog.info(paste('Received', nrow(records), 'records from Kinesis')),

    shutdown     = function()
        flog.info('Bye'),

    updater      = list(
        list(1, function()
            flog.info('Updating some data every minute')),
        list(1/60*10, function()
            flog.info(paste(
                'This is a high frequency updater call',
                'running every 10 seconds'))),

    checkpointing = 1,
    logfile = '/logs/logger.log')
Running the MultiLangDaemon locally

```
[ec2-user@ip logs]$ head -n 44 logger.log
INFO [2017-03-05 03:35:23] Starting R Kinesis Consumer application
INFO [2017-03-05 03:35:23 UTC] shardid=00000000000000000000 Start of initialize
INFO [2017-03-05 03:35:23 UTC] shardid=00000000000000000000 End of initialize
INFO [2017-03-05 03:35:23 UTC] shardid=00000000000000000000 Received 2 records from Kinesis
INFO [2017-03-05 03:35:24 UTC] shardid=00000000000000000000 Received 3 records from Kinesis
INFO [2017-03-05 03:35:25 UTC] shardid=00000000000000000000 Received 3 records from Kinesis
INFO [2017-03-05 03:35:26 UTC] shardid=00000000000000000000 Received 3 records from Kinesis
INFO [2017-03-05 03:35:27 UTC] shardid=00000000000000000000 Received 3 records from Kinesis
INFO [2017-03-05 03:35:28 UTC] shardid=00000000000000000000 Received 3 records from Kinesis
INFO [2017-03-05 03:35:29 UTC] shardid=00000000000000000000 Received 3 records from Kinesis
INFO [2017-03-05 03:35:30 UTC] shardid=00000000000000000000 Received 3 records from Kinesis
INFO [2017-03-05 03:35:31 UTC] shardid=00000000000000000000 Received 3 records from Kinesis
INFO [2017-03-05 03:35:32 UTC] shardid=00000000000000000000 Received 2 records from Kinesis
INFO [2017-03-05 03:35:33 UTC] shardid=00000000000000000000 Received 2 records from Kinesis
INFO [2017-03-05 03:35:34 UTC] shardid=00000000000000000000 This is a high frequency updater call running every 10 seconds
INFO [2017-03-05 03:35:35 UTC] shardid=00000000000000000000 Received 3 records from Kinesis
INFO [2017-03-05 03:35:35 UTC] shardid=00000000000000000000 Received 2 records from Kinesis
INFO [2017-03-05 03:35:36 UTC] shardid=00000000000000000000 Received 3 records from Kinesis
INFO [2017-03-05 03:35:37 UTC] shardid=00000000000000000000 Received 3 records from Kinesis
INFO [2017-03-05 03:35:38 UTC] shardid=00000000000000000000 Received 3 records from Kinesis
INFO [2017-03-05 03:35:39 UTC] shardid=00000000000000000000 Received 2 records from Kinesis
INFO [2017-03-05 03:35:40 UTC] shardid=00000000000000000000 Received 2 records from Kinesis
INFO [2017-03-05 03:35:41 UTC] shardid=00000000000000000000 Received 2 records from Kinesis
INFO [2017-03-05 03:35:42 UTC] shardid=00000000000000000000 Received 2 records from Kinesis
INFO [2017-03-05 03:35:43 UTC] shardid=00000000000000000000 Received 2 records from Kinesis
INFO [2017-03-05 03:35:43 UTC] shardid=00000000000000000000 This is a high frequency updater call running every 10 seconds
INFO [2017-03-05 03:35:44 UTC] shardid=00000000000000000000 Received 3 records from Kinesis
INFO [2017-03-05 03:35:45 UTC] shardid=00000000000000000000 Received 2 records from Kinesis
INFO [2017-03-05 03:35:46 UTC] shardid=00000000000000000000 Received 3 records from Kinesis
INFO [2017-03-05 03:35:47 UTC] shardid=00000000000000000000 Received 3 records from Kinesis
INFO [2017-03-05 03:35:48 UTC] shardid=00000000000000000000 Received 3 records from Kinesis
INFO [2017-03-05 03:35:49 UTC] shardid=00000000000000000000 Received 3 records from Kinesis
INFO [2017-03-05 03:35:50 UTC] shardid=00000000000000000000 Received 3 records from Kinesis
INFO [2017-03-05 03:35:51 UTC] shardid=00000000000000000000 Received 3 records from Kinesis
INFO [2017-03-05 03:35:52 UTC] shardid=00000000000000000000 Received 3 records from Kinesis
INFO [2017-03-05 03:35:53 UTC] shardid=00000000000000000000 Received 3 records from Kinesis
INFO [2017-03-05 03:35:54 UTC] shardid=00000000000000000000 Received 3 records from Kinesis
INFO [2017-03-05 03:35:55 UTC] shardid=00000000000000000000 Received 3 records from Kinesis
INFO [2017-03-05 03:35:56 UTC] shardid=00000000000000000000 Received 3 records from Kinesis
INFO [2017-03-05 03:35:57 UTC] shardid=00000000000000000000 Received 3 records from Kinesis
INFO [2017-03-05 03:35:58 UTC] shardid=00000000000000000000 Received 3 records from Kinesis
INFO [2017-03-05 03:35:59 UTC] shardid=00000000000000000000 Received 3 records from Kinesis
```
Let’s run it in AWS!

1. Dockerize your Kinesis Consumer:
   - Java
   - R
   - AWR, AWR.Kinesis packages
   - app.R
   - app.properties
   - startup command

2. Put it on Docker Hub

3. Run as a EC2 Container Service Task:
   - Create an ECS cluster
   - Create ECS Task Role
   - Create a Task definition
   - Run it (as a service)
Dockerize your Kinesis Consumer

```
FROM cardcorp/r-aws-java-pandoc:latest
MAINTAINER Gergely Daroczi <gergely.daroczi@card.com>

## Install extra AWR packages
RUN install2r --error 
  AWR.KMS 
  AWR.Kinesis 
  && rm -rf /tmp/downloaded_packages/ /tmp/*.rds

## Run MultiLangDaemon on /app
ENTRYPOINT ["/usr/bin/java", 
  
  
  
  "java", 
  
  "-cp", 
  
  "/usr/local/lib/R/site-library/AWR/java/**:/usr/local/lib/R/site-library/AWR.Kinesis/java/**", 
  
  "com.amazonaws.services.kinesis.MultiLang.MultiLangDaemon"]

## Override this if the consumer app is mounted elsewhere or the config file has a different name
CMD ["/app/app.properties"]
```
Dockerize your Kinesis Consumer

```
FROM cardcorp/r-kinesis:latest
MAINTAINER Gergely Daroczi <gergely.daroczi@card.com>

# Add consumer
COPY files /app
```
Dockerize your Kinesis Consumer

Gergely Daroczi (@daroczig)

Stream processing using AWR

github.com/cardcorp/AWR
Dockerize your Kinesis Consumer

Gergely Daroczi (@daroczig)

Stream processing using AWR

github.com/cardcorp/AWR
Create an ECS cluster

Create Cluster

When you run tasks using Amazon ECS, you place them on a cluster, which is a logical grouping of EC2 instances. This wizard will guide you through the process to create a cluster. You will name your cluster, and then configure the container instances that your tasks can be placed on, the security group for your container instances to use, and the IAM role to associate with your container instances so that they can make calls to the AWS APIs on your behalf.

Cluster name: AWR-test

EC2 Instance type: t2.medium

Number of instances: 1

EC2 Ami Id: amzn-ami-2016.09.1-amazon-ecs-optimized

EBS storage (GB): 22

Networking

Configure the VPC for your container instances to use. A VPC is an isolated portion of the AWS cloud populated by AWS objects, such as Amazon EC2 instances. You can choose an existing VPC, or create a new one with this wizard.

VPC: Create a new vpc

CIDR Block: 10.0.0.0/16
Create ECS Task Role

Set Role Name

Enter a role name. You cannot edit the role name after the role is created.

Role Name: AWR-ECS

Maximum 64 characters. Use alphanumeric and `+/-/\_` characters.
Create ECS Task Role

Select Role Type

- **AWS Service Roles**
  - **Amazon EC2 Role for EC2 Container Service**
    - Role to allow EC2 instances in an Amazon ECS cluster to access Amazon ECS.
  - **Amazon EC2 Container Service Role**
    - Allows ECS to create and manage AWS resources on your behalf.
  - **Amazon EC2 Container Service Task Role**
    - Allows ECS tasks to call AWS services on your behalf.
  - **Amazon EC2 Spot Fleet Role**
    - Role to allow EC2 Spot Fleet to request and terminate Spot instances on your behalf.
  - **Amazon Elastic MapReduce**
    - Role to allow EMR to access other AWS services such as EC2 on your behalf.

- **Role for Cross-Account Access**
- **Role for Identity Provider Access**
Create ECS Task Role

Attach Policy

Select one or more policies to attach. Each role can have up to 10 policies attached.

<table>
<thead>
<tr>
<th>Policy Name</th>
<th>Attached Entities</th>
<th>Creation Time</th>
<th>Edited Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>AmazonKinesisFullAccess</td>
<td>1</td>
<td>2015-02-06 10:40 PST</td>
<td>2015-02-06 10:40 PST</td>
</tr>
<tr>
<td>AmazonKinesisReadOnlyA...</td>
<td>1</td>
<td>2015-02-06 10:40 PST</td>
<td>2015-02-06 10:40 PST</td>
</tr>
<tr>
<td>AmazonKinesisAnalyticsF...</td>
<td>0</td>
<td>2016-09-21 12:01 PST</td>
<td>2016-09-21 12:01 PST</td>
</tr>
<tr>
<td>AmazonKinesisAnalyticsR...</td>
<td>0</td>
<td>2016-09-21 11:16 PST</td>
<td>2016-09-21 11:16 PST</td>
</tr>
<tr>
<td>AmazonKinesisFirehoseF...</td>
<td>0</td>
<td>2015-10-07 11:45 PST</td>
<td>2015-10-07 11:45 PST</td>
</tr>
<tr>
<td>AmazonKinesisFirehoseR...</td>
<td>0</td>
<td>2015-10-07 11:43 PST</td>
<td>2015-10-07 11:43 PST</td>
</tr>
<tr>
<td>AWSLambdaKinesisExecution...</td>
<td>0</td>
<td>2015-04-09 08:14 PST</td>
<td>2015-04-09 08:14 PST</td>
</tr>
</tbody>
</table>

Step 1: Set Role Name
Step 2: Select Role Type
Step 3: Establish Trust
Step 4: Attach Policy
Step 5: Review
Create a Task Definition

A task definition specifies which containers are included in your task and how they interact with each other. You can also specify data volumes for your containers to use.

- **Task Definition Name**: AWR-logger
- **Task Role**: AWR-ECS (Optional IAM role that tasks can use to make API requests to authorized AWS services. Create an Amazon EC2 Container Service Task Role in the IAM Console.
- **Network Mode**: Bridge

**Constraint**
Constraints allow you to filter the instances used for your placement strategies using built-in or custom attributes. The scheduler first filters the instances that match the constraints and then applies the placement strategy to place the task.

<table>
<thead>
<tr>
<th>Type</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add constraint</td>
<td></td>
</tr>
</tbody>
</table>

**Container Definitions**

<table>
<thead>
<tr>
<th>Container Name</th>
<th>Image</th>
<th>Hard/Soft memory limits (MB)</th>
<th>Essential</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

No results

**Volumes**

<table>
<thead>
<tr>
<th>Name</th>
<th>Source Path</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

No results
Create a Task definition

Amazon ECS
Clusters
| Task Definitions
Repositories

Constraint
Constraints allow you to filter the instances used for your placement strategies using built-in or custom attributes. The scheduler first filters the instances that match the constraints and then applies the placement strategy to place the task.

<table>
<thead>
<tr>
<th>Type</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add constraint</td>
<td></td>
</tr>
</tbody>
</table>

Container Definitions

<table>
<thead>
<tr>
<th>Container Name</th>
<th>Image</th>
<th>Hard/Soft memory limits (MB)</th>
<th>Essential</th>
</tr>
</thead>
<tbody>
<tr>
<td>No results</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Volumes

<table>
<thead>
<tr>
<th>Name</th>
<th>Source Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>No results</td>
<td></td>
</tr>
</tbody>
</table>

Add volume

Name: logs
Source path: /logs

Gergely Daroczi (@daroczig)
Stream processing using AWR
github.com/cardcorp/AWR
Create a Task definition

Add container

- **Standard**
  - **Container name**: logger
  - **Image**: cardcorp/kinesis-example:latest
  - **Memory Limits (MB)**: Hard limit 512

- **Advanced container configuration**
  - **ENVIRONMENT**
    - **CPU units**: 
    - **Essential**: Yes

*Required
Create a Task definition

A task definition specifies how your tasks should run. When you deploy your application, Docker containers are automatically provisioned based on the task definitions you provide. A task definition includes constraints and configuration options that determine how instances that match the task definition should be assigned to your tasks.

### STORAGE AND LOGGING

- **Read only root file system**
- **Mount points**
  - Source volume: logs
  - Container path: /logs
  - Read only:
- **Add mount point**

### Volumes from

- **Source container**
- **Read only**

### Log configuration

- **Log driver**: [none]
- **Log options**
  - Key: Add key
  - Value: Add value

### SECURITY

*Required*
Run the ECS Task
Run the ECS Task
Run the ECS Task

Gergely Daroczi (@daroczig)
Stream processing using AWR
github.com/cardcorp/AWR
Scaling the Kinesis Consumer up

Gergely Daroczi (@daroczig)

Stream processing using AWR

github.com/cardcorp/AWR
Nice example project, but . . .

- I might want to avoid publishing my Consumer on Docker Hub
- I might want to avoid publishing my code on GitHub
- I might want to avoid committing credentials etc to the repo

Problems:

- How to store credentials in the Docker images?
- Where to store the Docker images?
Nice example project, but ...

- I might want to avoid publishing my Consumer on Docker Hub
- I might want to avoid publishing my code on GitHub
- I might want to avoid committing credentials etc to the repo

Problems:

- How to store credentials in the Docker images? **KMS**
- Where to store the Docker images? **ECR**
Source: AWS Encryption SDK
Current AWR.KMS Features

- encrypt up to 4 KB of arbitrary data:

```r
> library(AWR.KMS)
> kms_encrypt('alias/mykey', 'foobar')
[1] "Base-64 encoded ciphertext"
```

- decrypt such Base-64 encoded ciphertext back to plaintext:

```r
> kms_encrypt('Base-64 encoded ciphertext')
[1] "foobar"
```

- generate a data encryption key:

```r
> kms_generate_data_key('alias/mykey')
$cipher
[1] "Base-64 encoded, encrypted data encryption key"
$key
[1] "alias/mykey"
$text
[1] 00 01 10 11 00 01 10 11 ...
```
### Encrypting Data Larger Than 4 KB?

```r
## let's say we want to encrypt the mtcars dataset stored in JSON
library(jsonlite)
data <- toJSON(mtcars)

## generate a 256-bit data encryption key (that's supported by digest::AES)
library(AWR.KMS)
key <- kms_generate_data_key('alias/mykey', byte = 32L)

## convert the JSON to raw so that we can use that with digest::AES
raw <- charToRaw(data)
## the text length must be a multiple of 16 bytes
raw <- c(raw, as.raw(rep(0, 16 - length(raw) %% 16)))

## encrypt the raw object with the new key + digest::AES
## the resulting text and the encrypted key can be stored on disk
library(digest)
aes <- AES(key$text)
base64_enc(aes$encrypt(raw))

## decrypt the above returned ciphertext using the decrypted key
rawToChar(aes$decrypt(base64_dec(...), raw = TRUE))
```

Gergely Daroczi (@daroczig)
Stream processing using AWR
github.com/cardcorp/AWR
library(AWR.Kinesis); library(jsonlite); library(AWR.KMS); library(futile.logger); flog.threshold(DEBUG)

kinesis_consumer(
  initialize = function() {
    flog.info("Decrypting Redis hostname via KMS")
    host <- kms_decrypt("AQECAHiiz4GEPFQLL9AA0N5TY/1DR5euQQScpXQU9iYTn+u...")
    flog.info("Connecting to Redis")
    library(rredis); redisConnect(host = host)
    flog.info("Connected to Redis")
  },
  processRecords = function(records) {
    flog.info(paste('Received', nrow(records), 'records from Kinesis'))
    for (record in records$data) {
      flight <- fromJSON(record)$dest
      if (!is.null(flight)) {
        flog.debug(paste('Adding +1 to', flight))
        redisIncr(sprintf('flight:%s', flight))
      } else {
        flog.error('Flight destination not found')
      }
    }
  },
  updater = list(list(1/6, function() {
    flog.info('Checking overall counters')
    flights <- redisKeys('flight:*')
    for (flight in flights) {
      flog.debug(paste('Found', redisGet(flight), sub('^flight:', '', flight)))
    }
  })),
  logfile = '/logs/redis.log')

Private Docker Image

Dockerfile:

```
FROM cardcorp/r-kinesis:latest
MAINTAINER Gergely Daroczi <gergely.daroczi@card.com>

## Install R package to interact with Redis
RUN install2.r --error rredis && rm -rf /tmp/downloaded_packages/ /tmp/*.rds

## Add consumer
COPY files /app
```

Build and push to ECR:

```
docker build -t cardcorp/r-kinesis-secret .
`aws ecr get-login --region us-east-1`
docker tag -f cardcorp/r-kinesis-secret:latest \
    ***.dkr.ecr.us-east-1.amazonaws.com/cardcorp/r-kinesis-secret:latest
docker push ***.dkr.ecr.us-east-1.amazonaws.com/cardcorp/r-kinesis-secret:latest
```
library(treemap); library(highcharter); library(nycflights13)
library(rredis); redisConnect(host = '***', port = '***')

ui <- shinyUI(highchartOutput('treemap', height = '800px'))
server <- shinyServer(function(input, output, session) {

  destinations <- reactive({
    reactiveTimer(2000)()
    flights <- redisMGet(redisKeys('flight:*'))
    flights <- data.frame(faa = sub('^flight:', '', names(flights)),
      N = as.numeric(flights))
    merge(flights, airports, by = 'faa')
  })

  output$treemap <- renderHighchart({
    tm <- treemap(destinations(), index = c('faa'),
      vSize = 'N', vColor = 'tz',
      type = 'value', draw = FALSE)
    hc_title(hctreemap(tm, animation = FALSE), text = 'Flights from NYC')
  })

})
shinyApp(ui = ui, server = server)
Technical Details

- AWR repo:
  - 6.3 GB
  - 273 tags/versions
  - GitLab + CI + drat

```r
install.packages('AWR', repos = 'https://cardcorp.gitlab.io/AWR')
```
Technical Details

- AWR repo:
  - 6.3 GB
  - 273 tags/versions
  - GitLab + CI + drat

```
install.packages('AWR', repos = 'https://cardcorp.gitlab.io/AWR')
```

- Submitted to CRAN on
  - 2016-12-05
Technical Details

- AWR repo:
  - 6.3 GB
  - 273 tags/versions
  - GitLab + CI + drat

```
install.packages('AWR', repos = 'https://cardcorp.gitlab.io/AWR')
```

- Submitted to CRAN on
  - 2016-12-05
  - 2017-01-09
  - 2017-01-10
  - 2017-01-11
  - 2017-01-11
  - 2017-01-13
Technical Details

- AWR repo:
  - 6.3 GB
  - 273 tags/versions
  - GitLab + CI + drat

```
install.packages('AWR', repos = 'https://cardcorp.gitlab.io/AWR')
```

- Submitted to CRAN on
  - 2016-12-05
  - 2017-01-09
  - 2017-01-10
  - 2017-01-11
  - 2017-01-11
  - 2017-01-13

- Release cycle: 2 minor, ~125 patch versions in the past 12 months
- CI
> library(rJava)
> kc <- .jnew('com.amazonaws.services.s3.AmazonS3Client')
> kc$getVar3AccountOwner()$getDisplayName()

[1] "foobar"
Because "S" is so 1992.