Stream processing with R in AWS

AWR, AWR.KMS, AWR.Kinesis (R packages) used in ECS

Gergely Daroczi

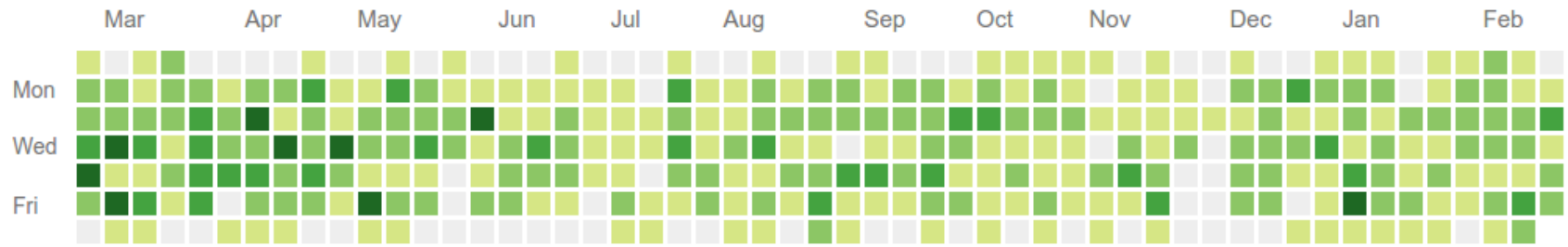
@daroczig

June 06, 2017

About me

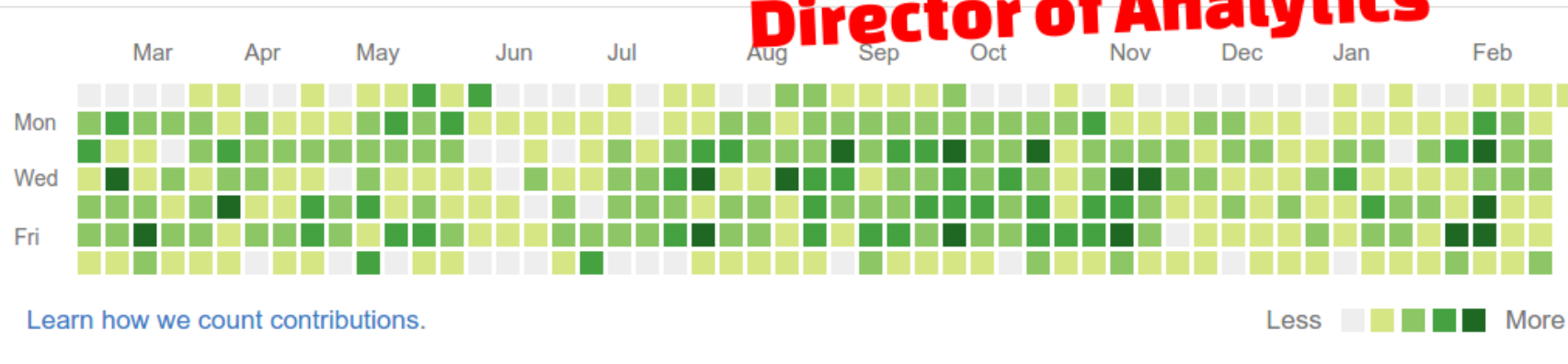
3,633 contributions in the year before last

Lead R Developer



3,470 contributions in the last year

Director of Analytics



About me

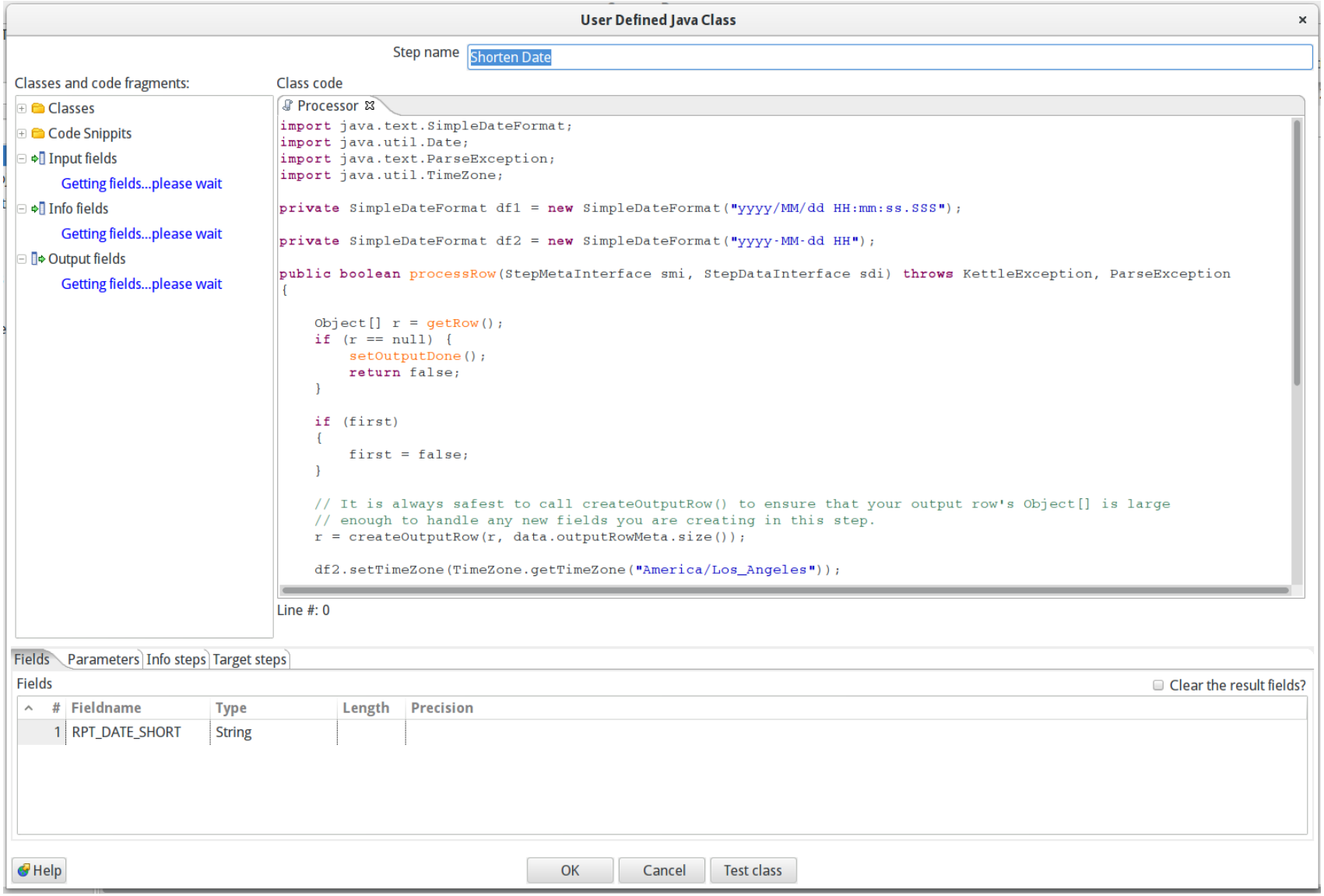
About me

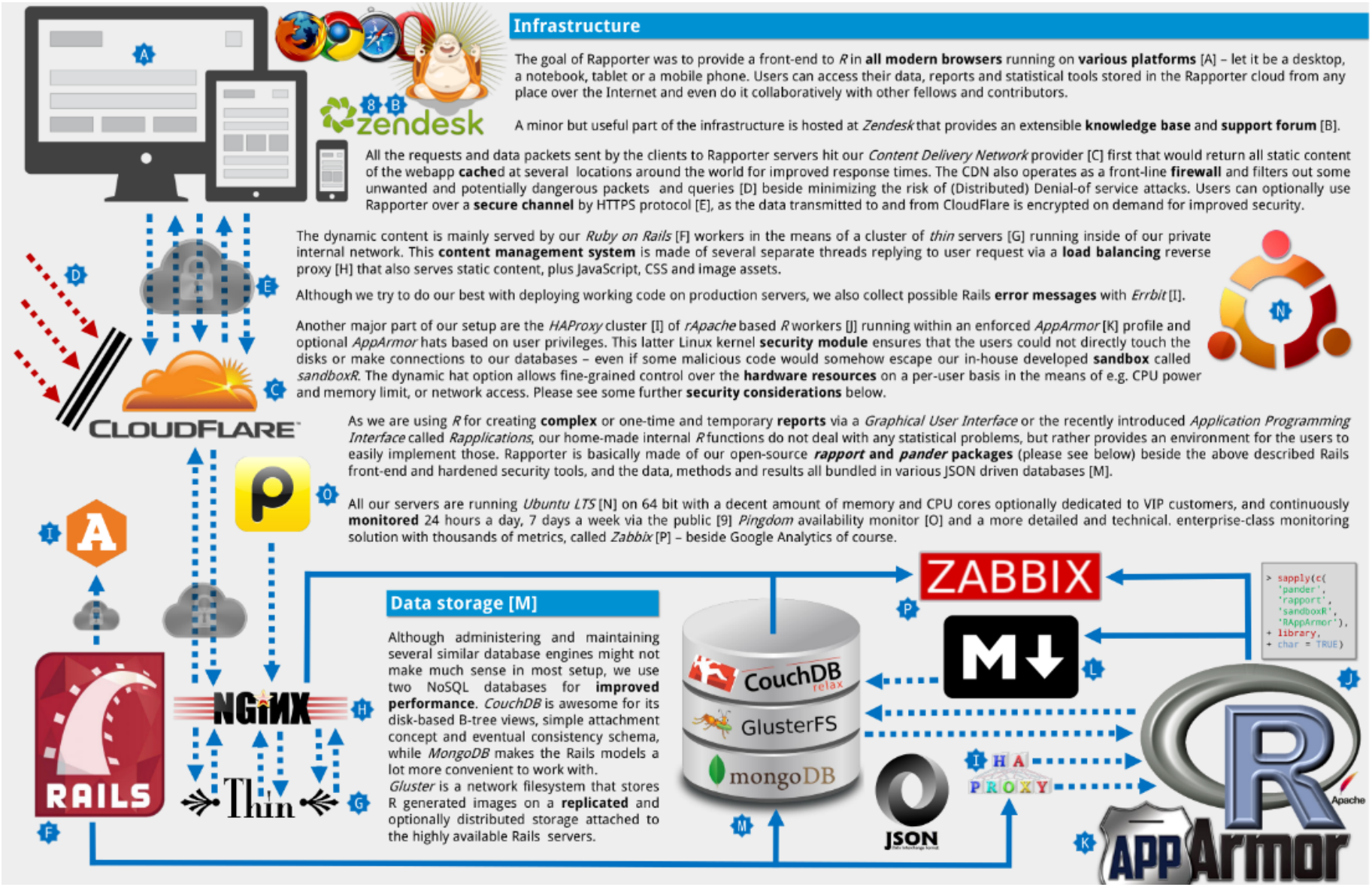
SYSTEM1

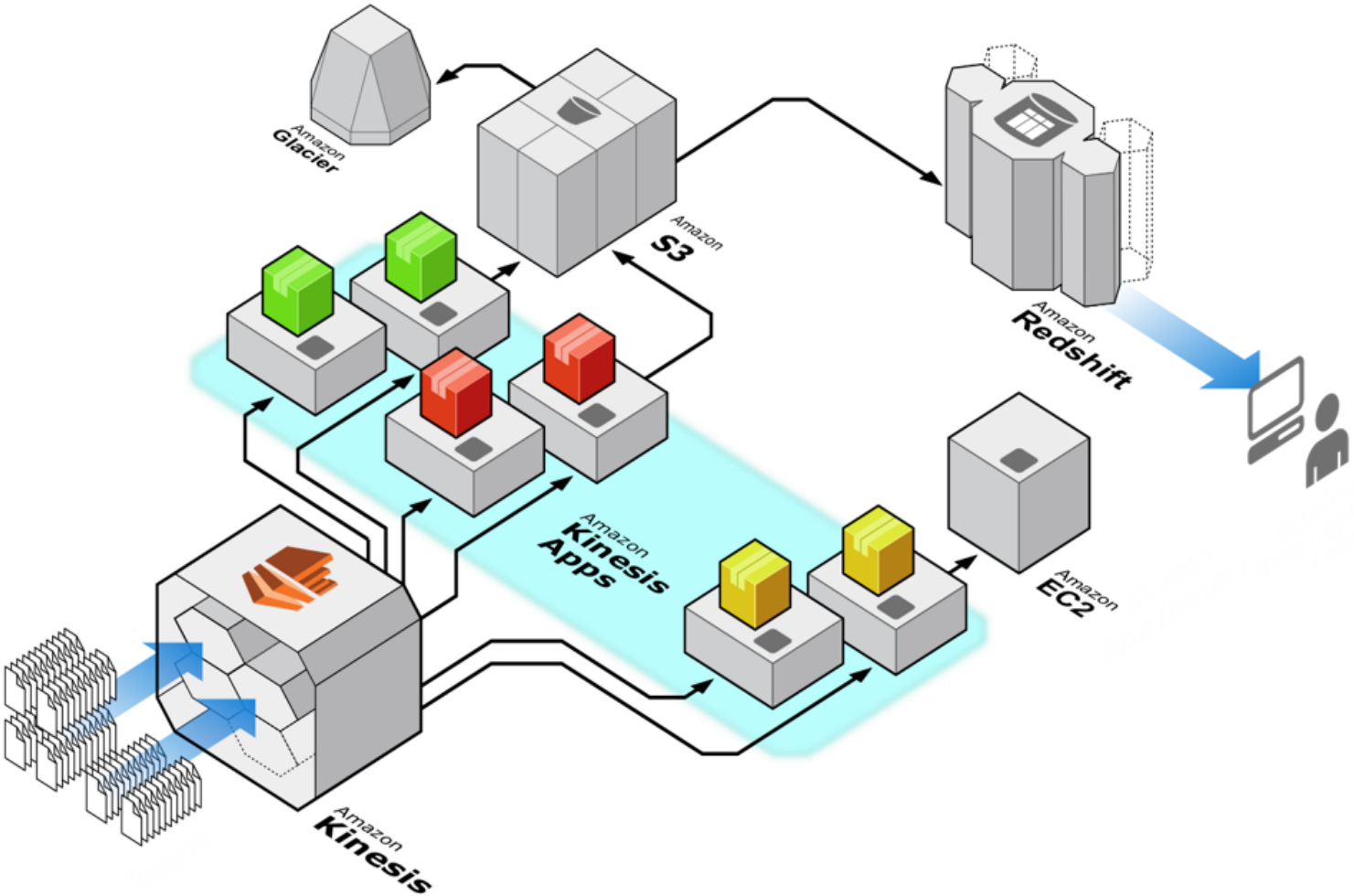
About me

Java™

Stream Processing . . . Why R?

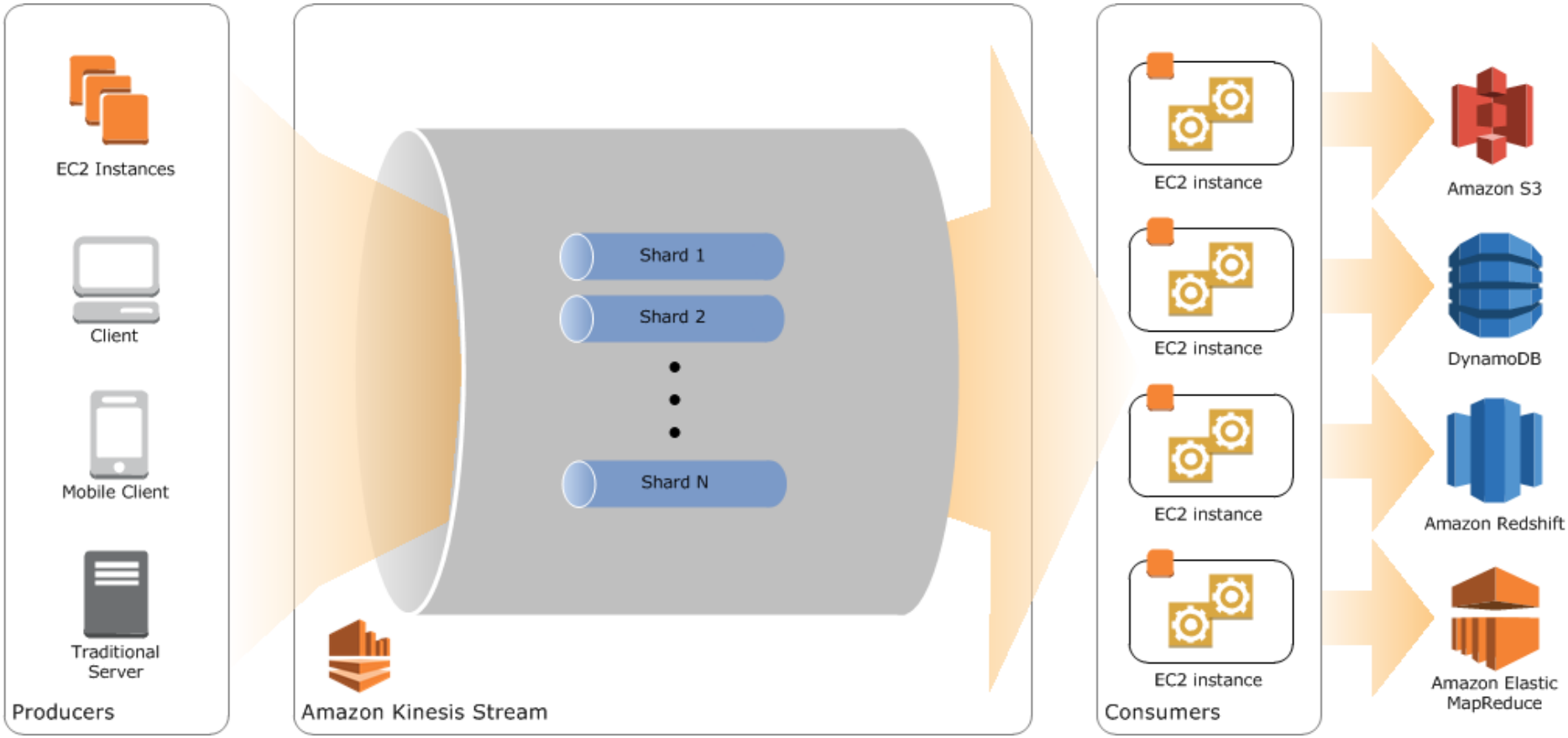



Infrastructure
The goal of Rapporter was to provide a front-end to R in all modern browsers running on various platforms [A] – let it be a desktop, a notebook, tablet or a mobile phone. Users can access their data, reports and statistical tools stored in the Rapporter cloud from any place over the Internet and even do it 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 webapp 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 separate 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 [I].
Another major part of our setup are the HAProxy cluster [I] of rApache based R workers [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 or one-time and temporary reports via a Graphical User Interface or the recently introduced Application Programming Interface called Rapplications, 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 rapport 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 [9] 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 schema, while MongoDB makes the Rails models a lot more convenient to work with.
Gluster is a network filesystem that stores R generated images on a replicated and optionally distributed storage attached to the highly available Rails servers.
CLOUDFLARE
zendesk
ZABBIX
CouchDB
GlusterFS
mongoDB
NGINX
Thin
RAILS
JSON
HAPROXY
AppArmor
Apache
> sapply(c('pander', 'rapport', 'sandboxR', 'RAppArmor'), + library, + char = TRUE)



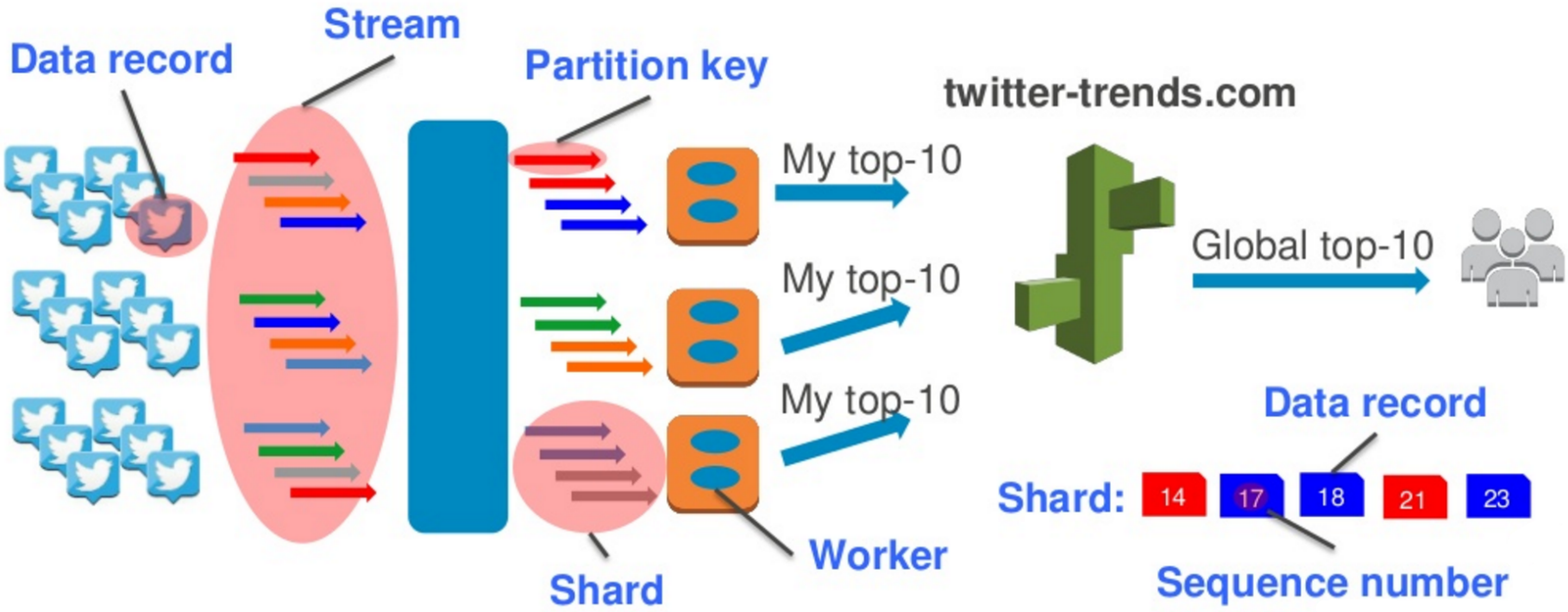
Source: Kinesis Product Details

Intro to Amazon Kinesis Streams



Source: Kinesis Developer Guide

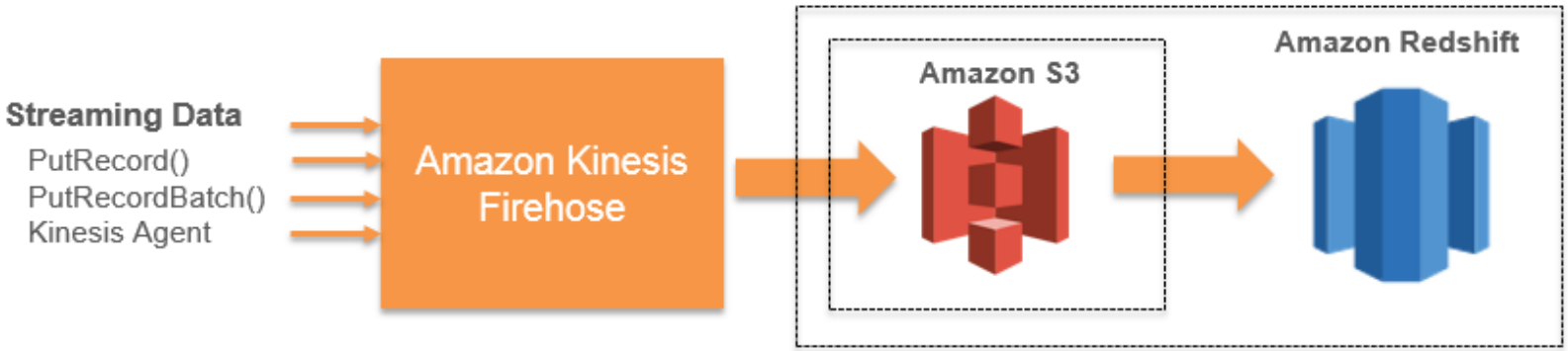
Intro to Amazon Kinesis Shards



Source: AWS re:Invent 2013

A Very Deep Learning

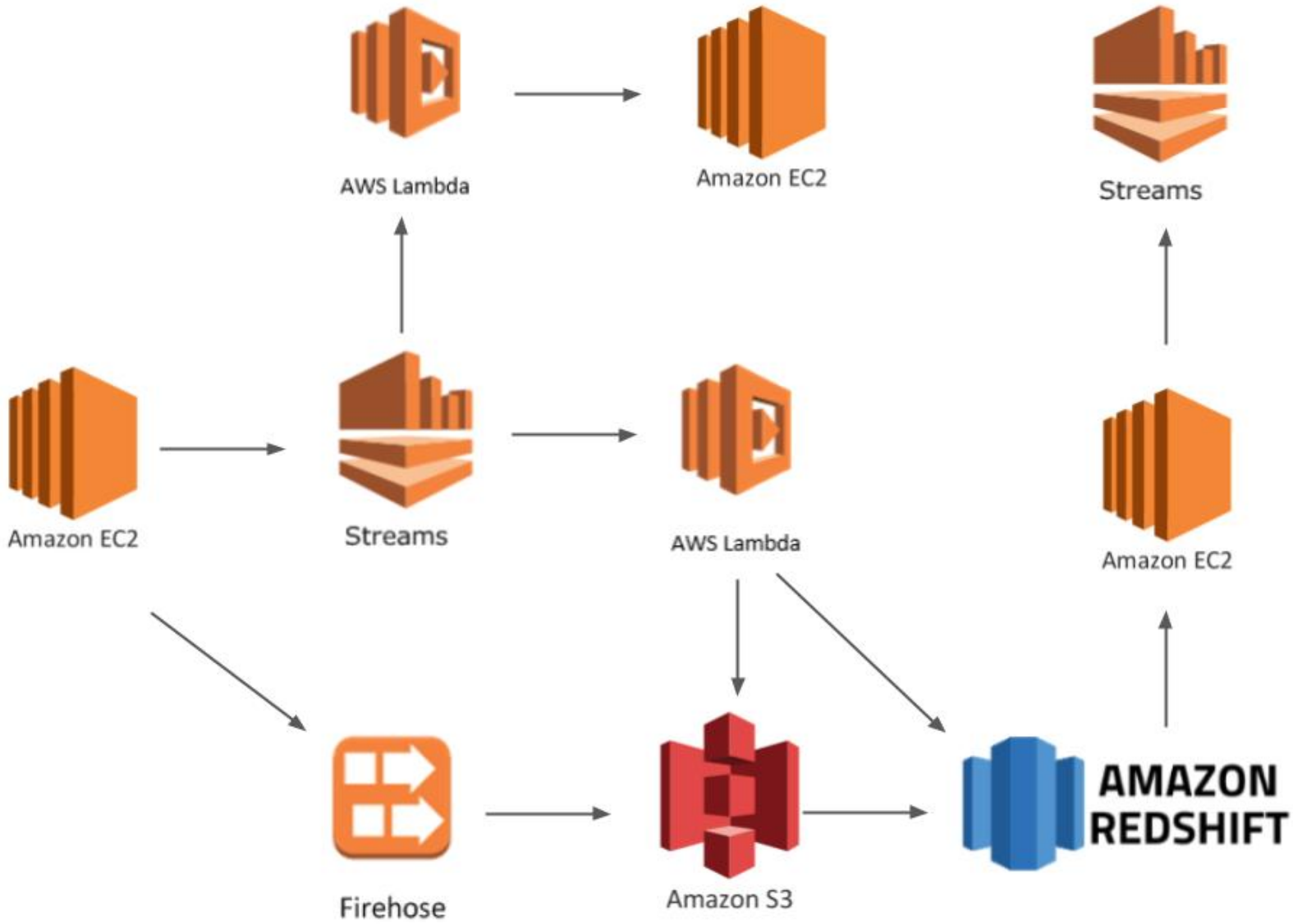

S3
FIREHOSE
REDSHIFT
Capture & submit streaming data to Firehose
Firehose loads streaming data continuously into S3 and Redshift
Analyze streaming data using your favorite BI tools


Streaming Data
PutRecord()
PutRecordBatch()
Kinesis Agent
Amazon Kinesis Firehose
Amazon S3
Amazon Redshift

The S3 Object System

```
> 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
INFO [2017-03-03 22:27:57] 3.14
INFO [2017-03-03 22:28:17] 3.14
INFO [2017-03-03 22:28:17] 3.14
```

S4: Multiple Dispatch



How to Communicate with Kinesis

EARL
ENTERPRISE APPLICATIONS OF THE R LANGUAGE

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!

EARL
ENTERPRISE APPLICATIONS OF THE R LANGUAGE

```
> 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: 49562894160449444332153346371084313572324361665
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

EARL
ENTERPRISE APPLICATIONS OF THE R LANGUAGE

Let's merge two shards:

```
> 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?

```
> kc$describeStream(StreamName = 'test_kinesis')$getStreamDescription()$getShards()
[1] "Java-Object{[
{ShardId: shardId-000000000000,HashKeyRange: {StartingHashKey: 0,EndingHashKey: 170
SequenceNumberRange: {
StartingSequenceNumber: 49562894160427143586954815717376297430913467927668719618,
EndingSequenceNumber: 49562894160438293959554081028945856364232263390243848194}},
{ShardId: shardId-000000000001,HashKeyRange: {StartingHashKey: 170141183460469231731
SequenceNumberRange: {
StartingSequenceNumber: 49562894160449444332153346340517833149186116289174700050,
EndingSequenceNumber: 49562894160460594704752611652087392082504911751749828626}},
{ShardId: shardId-000000000002,
ParentShardId: shardId-000000000000,
AdjacentParentShardId: shardId-000000000001,
HashKeyRange: {StartingHashKey: 0,EndingHashKey: 340282366920938463463374607431768
SequenceNumberRange: {StartingSequenceNumber: 49562904991497673099704924344727019527
```

Amazon Kinesis Client Library

EARL
ENTERPRISE APPLICATIONS OF THE R LANGUAGE

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

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

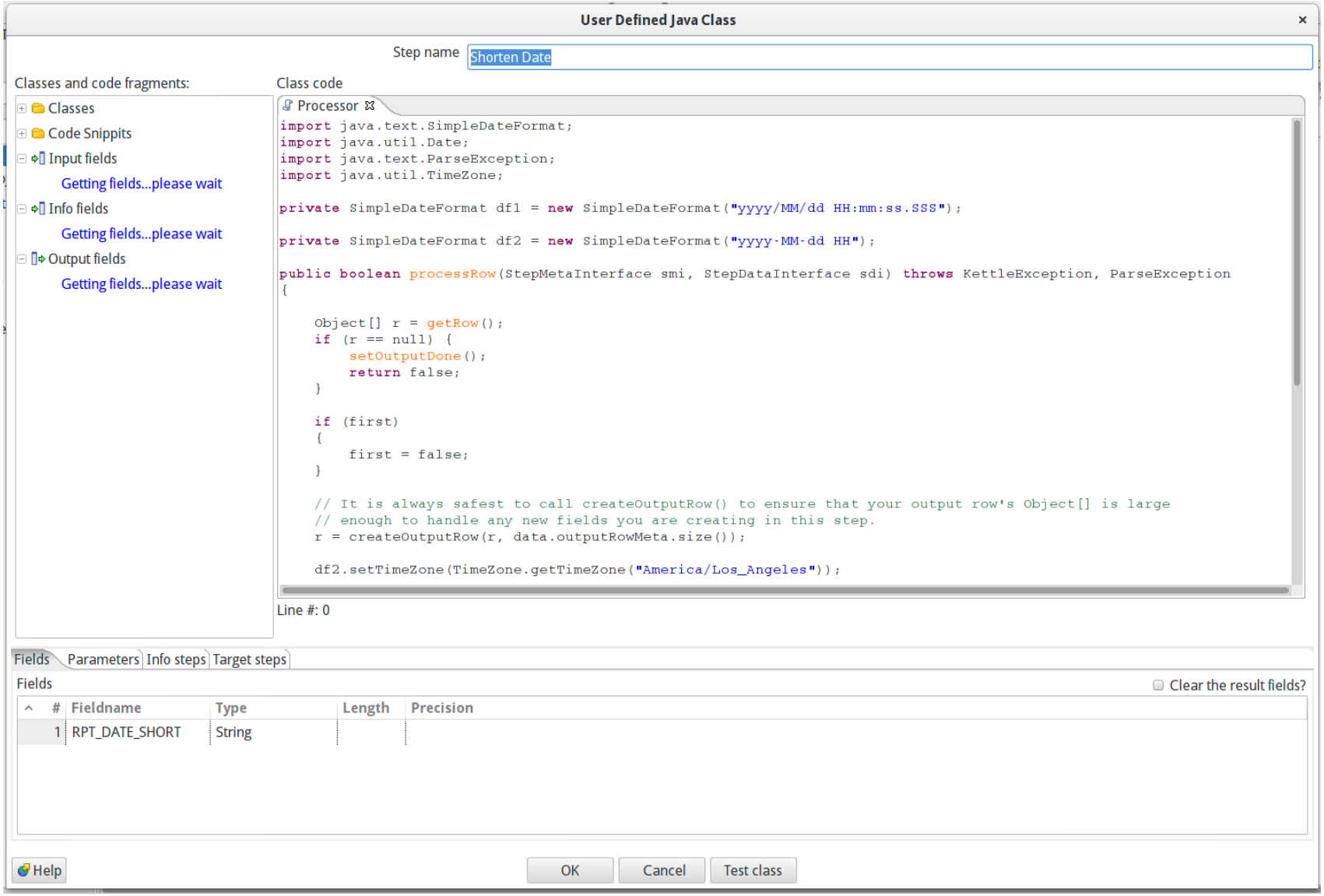
- *Scalable* and *fault-tolerant* processing (checkpointing via DynamoDB)
- Logging and metrics in CloudWatch
- The **MultiLangDaemon** spawns processes written in any language, communication happens via JSON messages sent over stdin/stdout
- Only a few events/methods to care about in the consumer application: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

Messages from the KCL

EARL
ENTERPRISE APPLICATIONS OF THE R LANGUAGE

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?


User Defined Java Class
Step name
Shorten Date
Classes and code fragments:
Class code
Classes
Code Snippits
Input fields
Getting fields...please wait
Info fields
Getting fields...please wait
Output fields
Getting fields...please wait
Processor
import java.text.SimpleDateFormat;
import java.util.Date;
import java.text.ParseException;
import java.util.TimeZone;
private SimpleDateFormat df1 = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss.SSS");
private SimpleDateFormat df2 = new SimpleDateFormat("yyyy-MM-dd HH");
public boolean processRow(StepMetaInterface smi, StepDataInterface sdi) throws KettleException, ParseException
{
Object[] r = getRow();
if (r == null) {
setOutputDone();
return false;
}
if (first)
{
first = false;
}
// It is always safest to call createOutputRow() to ensure that your output row's Object[] is large
// enough to handle any new fields you are creating in this step.
r = createOutputRow(r, data.outputRowMeta.size());
df2.setTimeZone(TimeZone.getTimeZone("America/Los_Angeles"));
Line #: 0
Fields
Parameters
Info steps
Target steps
Fields
Clear the result fields?
Fieldname Type Length Precision
1 RPT_DATE_SHORT String
Help
OK
Cancel
Test class

R Script Interacting with KCL

```
#!/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)))

}
```

R Script Interacting with KCL

EARL
ENTERPRISE APPLICATIONS OF THE R LANGUAGE

```
#!/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)))

}
```

Get rid of the bugs and the boilerplate

EARL
ENTERPRISE APPLICATIONS OF THE R LANGUAGE

```
> 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

EARL
ENTERPRISE APPLICATIONS OF THE R LANGUAGE

Business logic coded in R (demo_app.R):

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

Add content to the boilerplate

Business logic coded in R (demo_app.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

EARL
ENTERPRISE APPLICATIONS OF THE R LANGUAGE

Business logic coded in R (demo_app.R):

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

Config file for the MultiLangDaemon (demo_app.properties):

```
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
```

"Advanced" AWR.Kinesis features

EARL
ENTERPRISE APPLICATIONS OF THE R LANGUAGE

```
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!

EARL
ENTERPRISE APPLICATIONS OF THE R LANGUAGE

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!

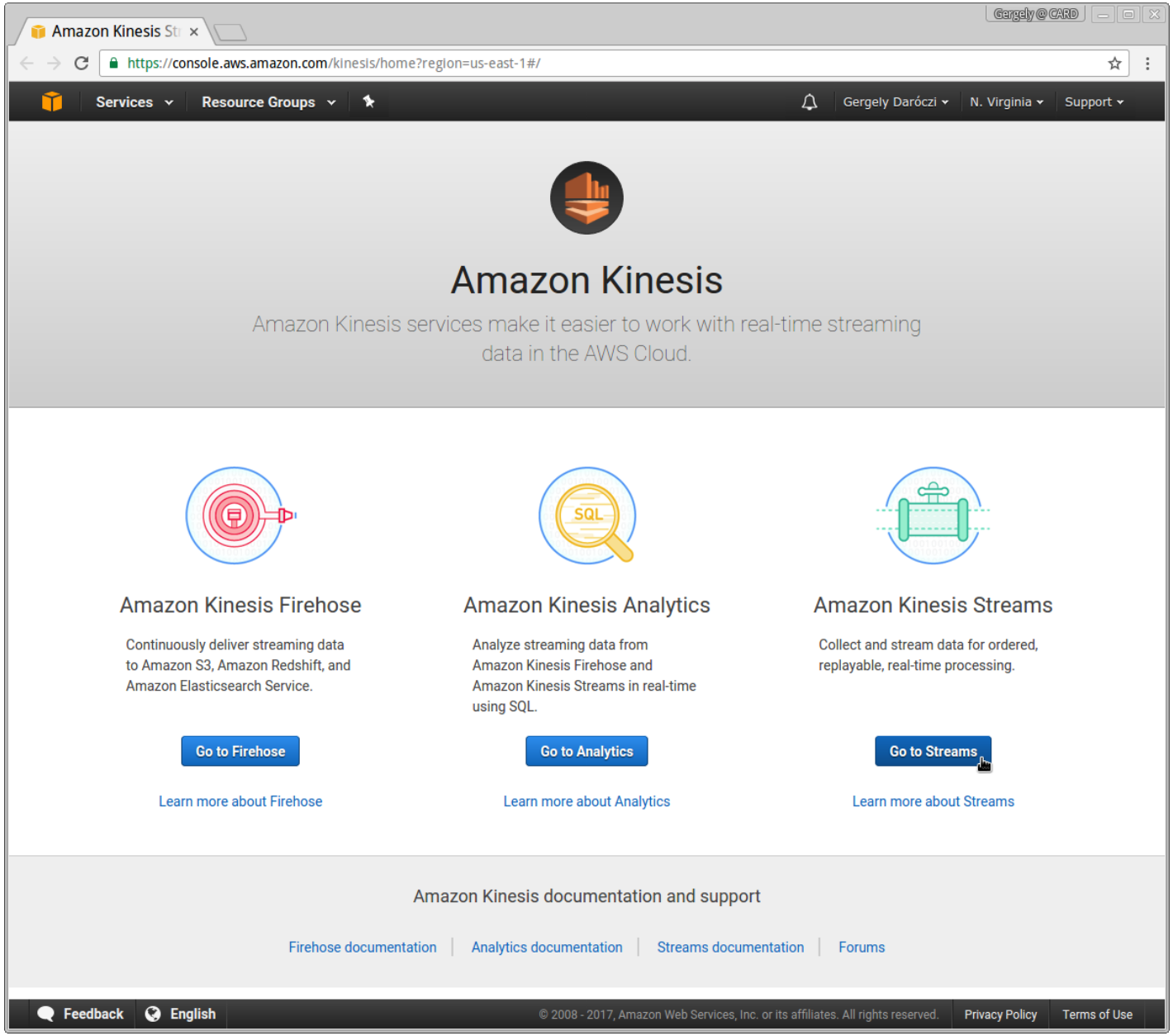
EARL
ENTERPRISE APPLICATIONS OF THE R LANGUAGE

Note

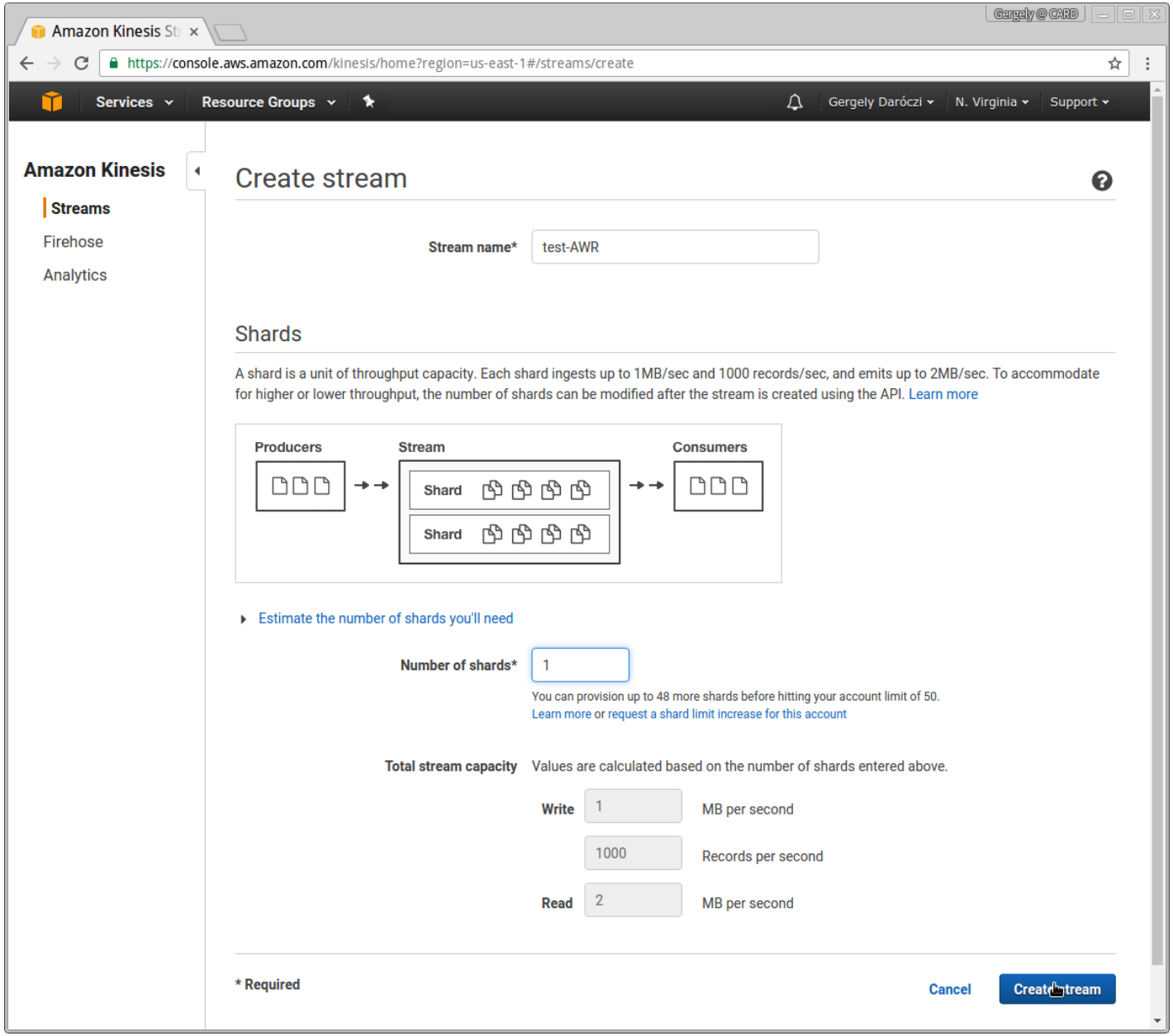
In theory you could, but this is not something you should run in RStudio.

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

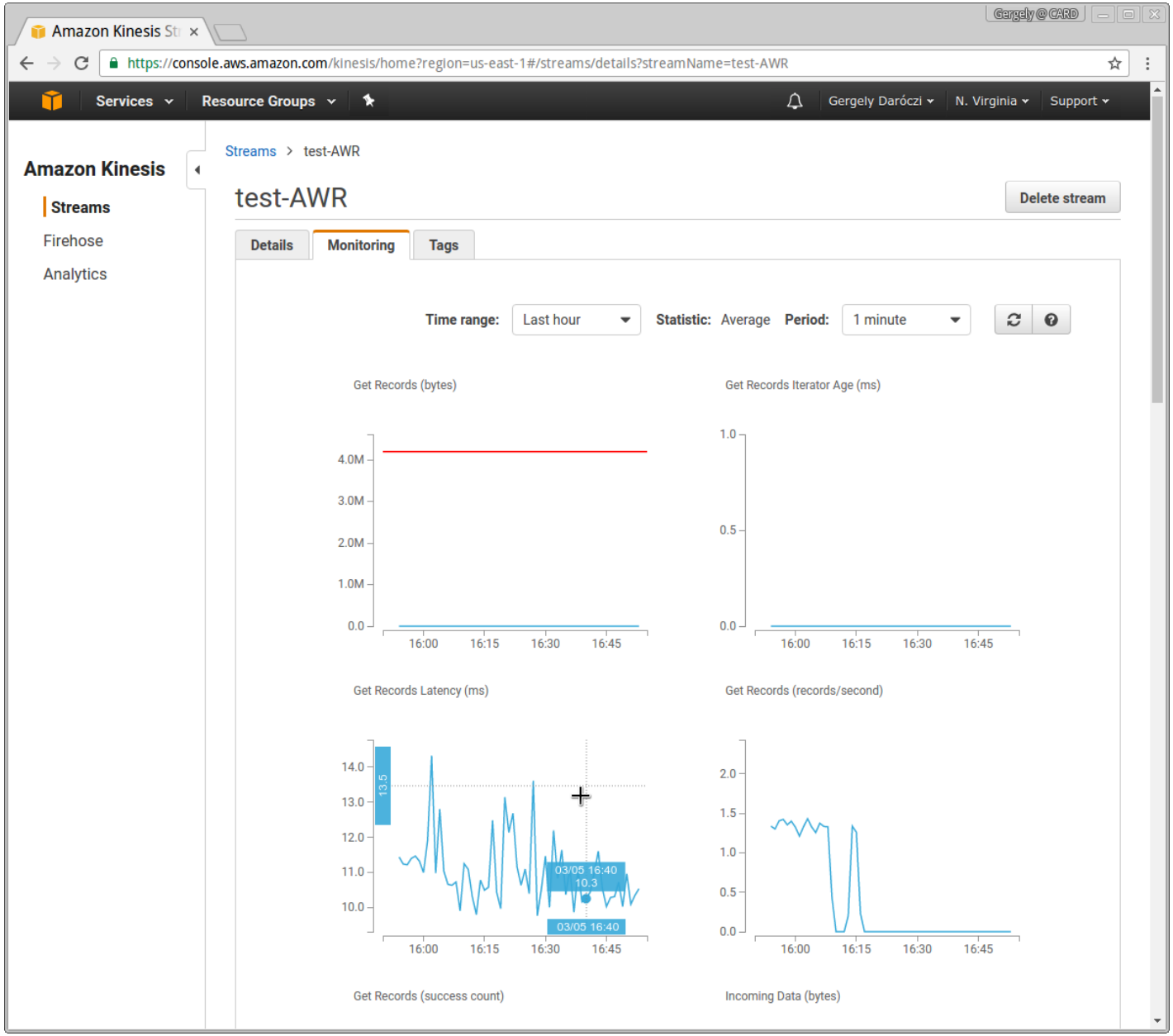
Create a Kinesis Stream



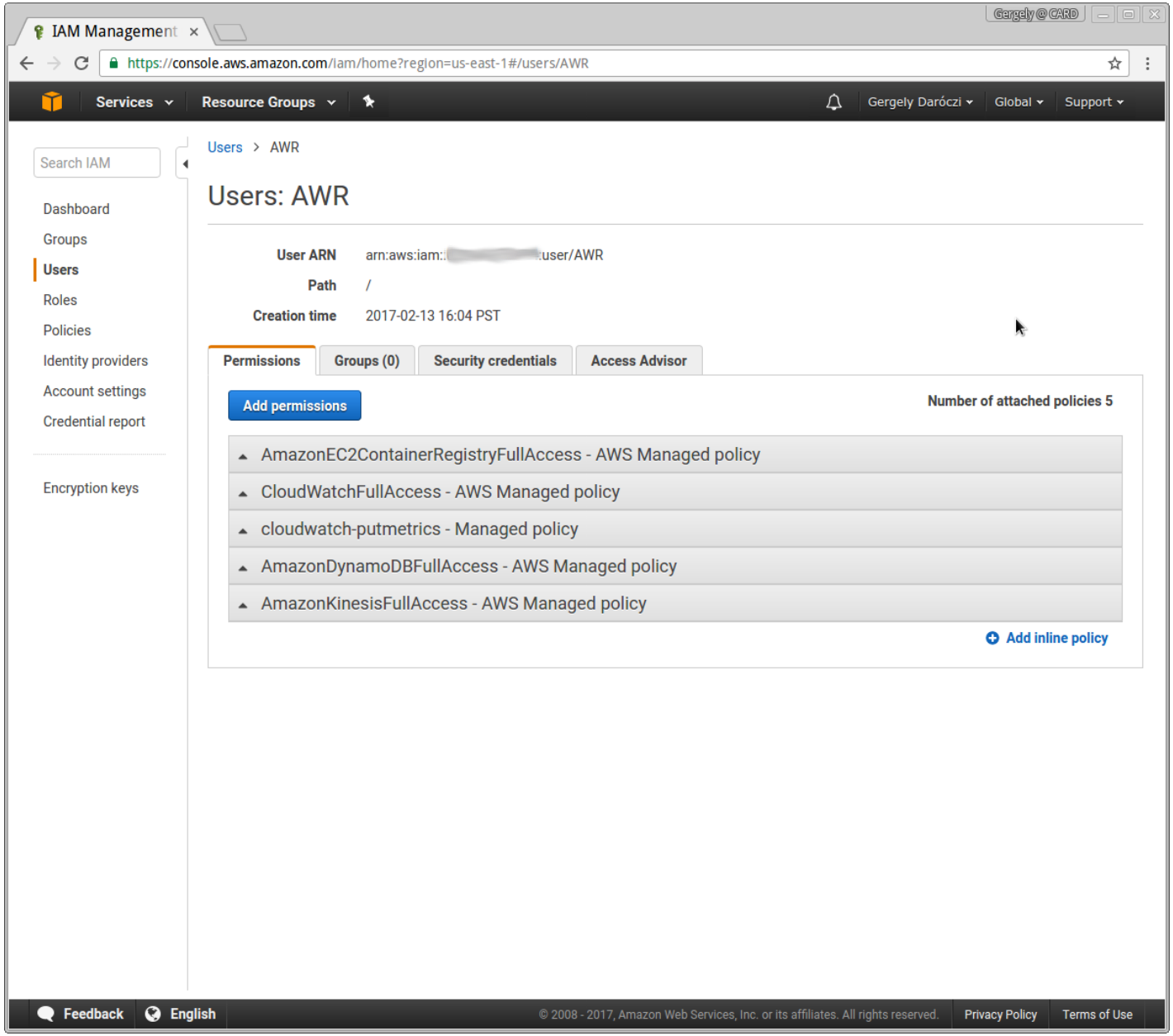
Create a Kinesis Stream



Check the Kinesis Stream



Create an IAM user



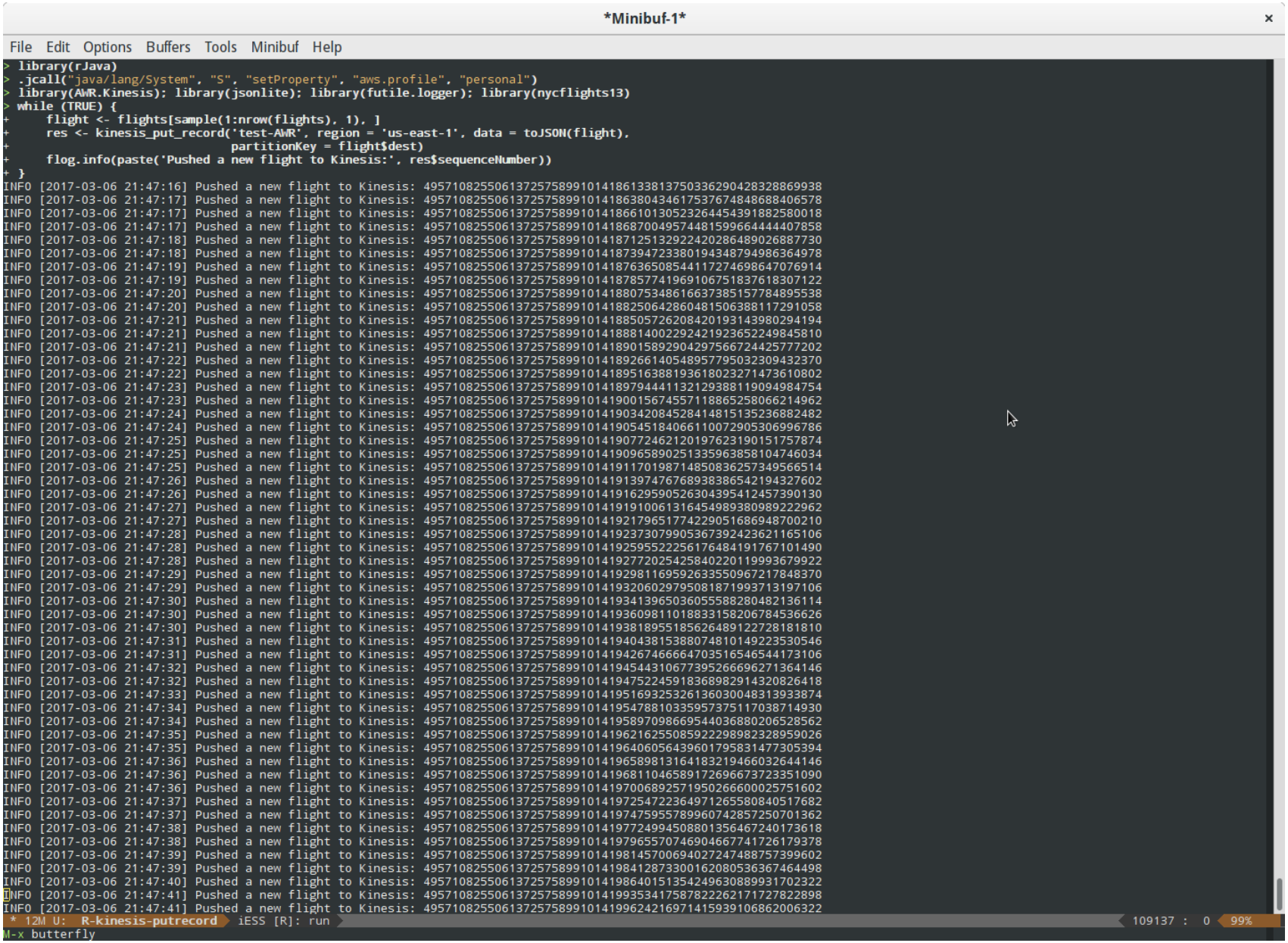
Write Data to the Stream from R

EARL
ENTERPRISE APPLICATIONS OF THE R LANGUAGE

```
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))

}
```

Write Data to the Stream from R

EARL
ENTERPRISE APPLICATIONS OF THE R LANGUAGE

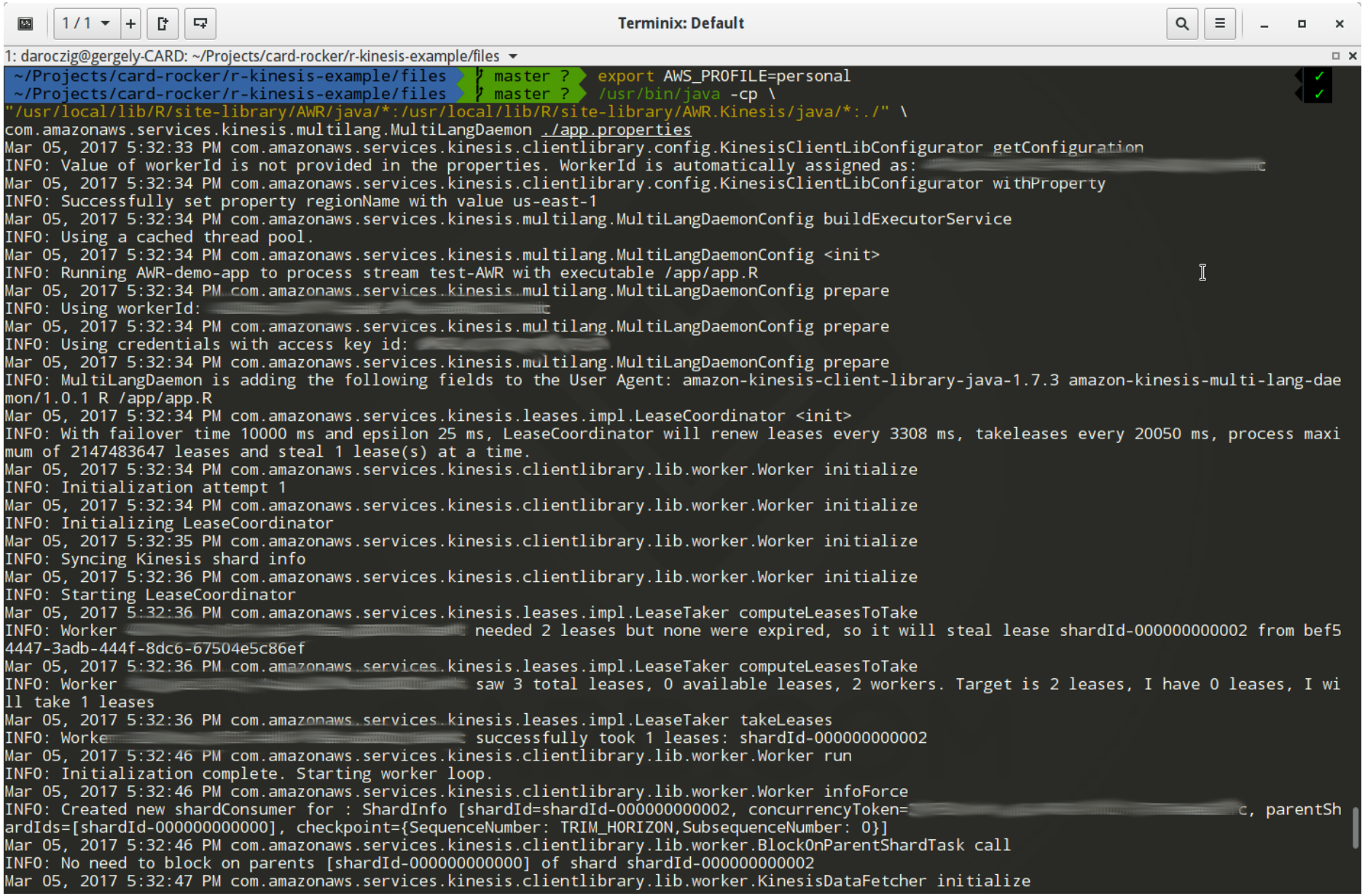


Reading Data from the Stream

EARL
ENTERPRISE APPLICATIONS OF THE R LANGUAGE

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

## transform to string
json <- sapply(records, function(x)
    rawToChar(x$getData()$array()))

## decode JSON
json[1]
fromJSON(json[1])
rbindlist(lapply(json, fromJSON))
```

Running the MultiLangDaemon locally

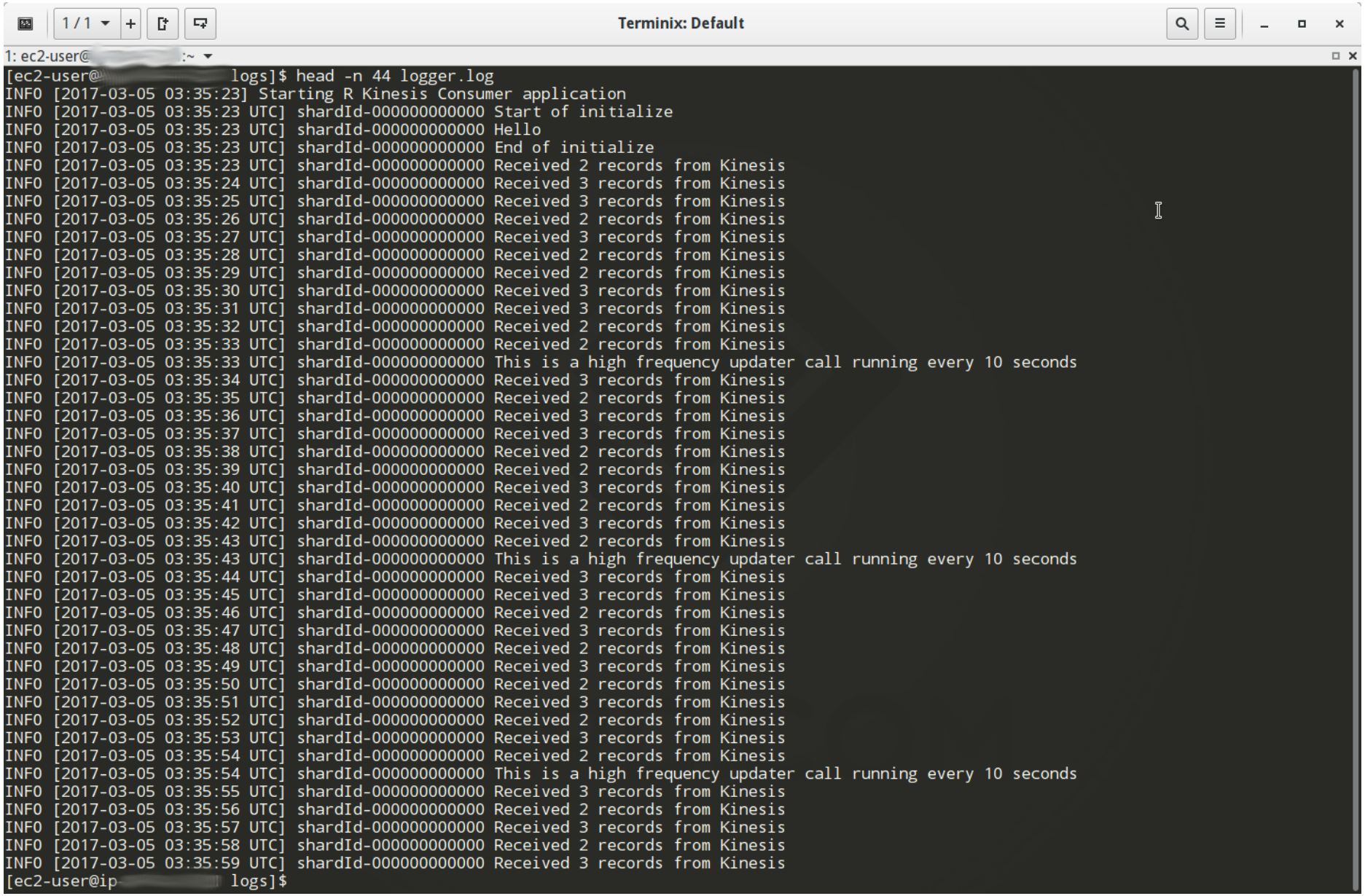


This Kinesis app is being run

EARL
ENTERPRISE APPLICATIONS OF THE R LANGUAGE

```
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

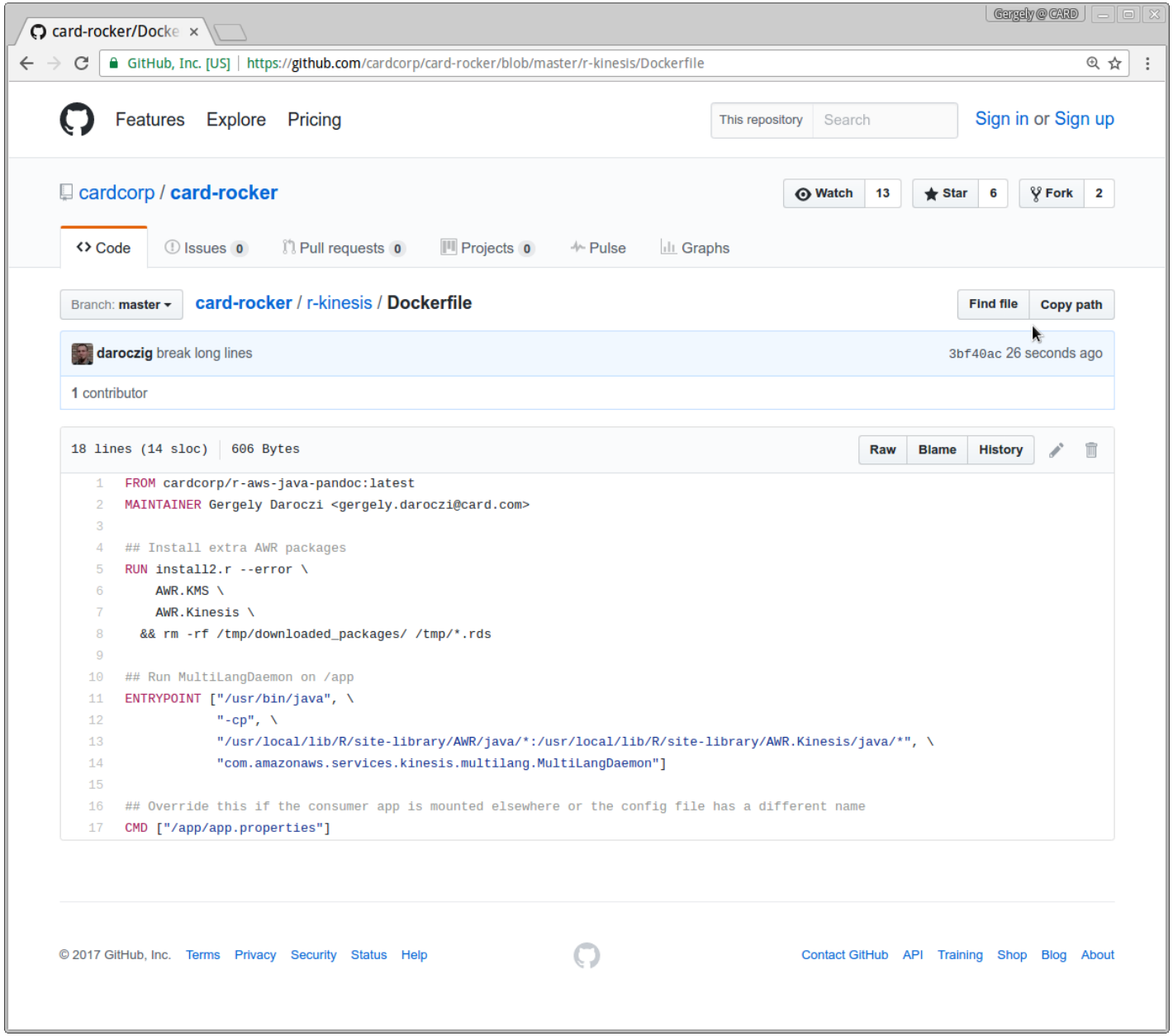


Let's run it in AWS!

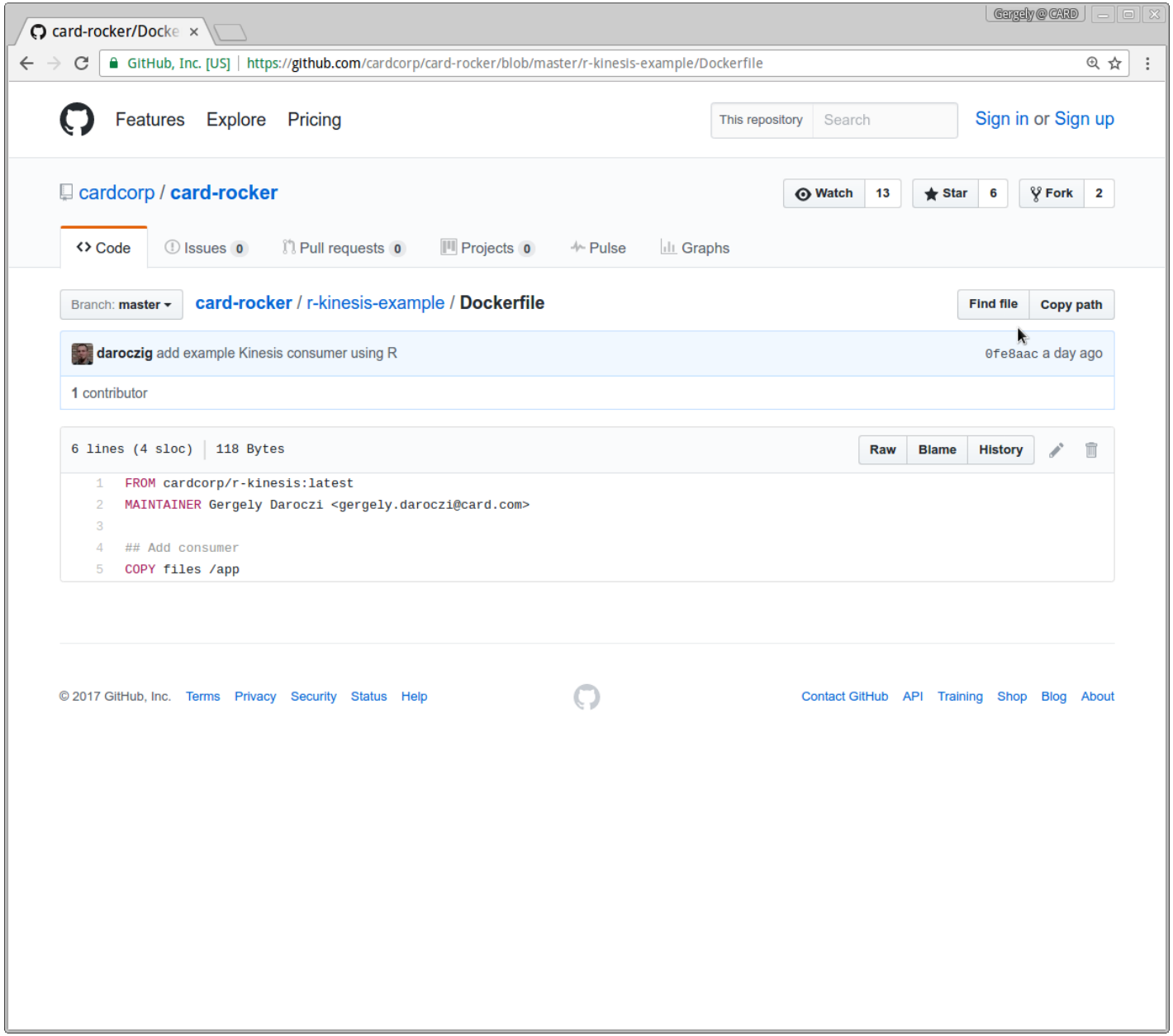
EARL
ENTERPRISE APPLICATIONS OF THE R LANGUAGE

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)

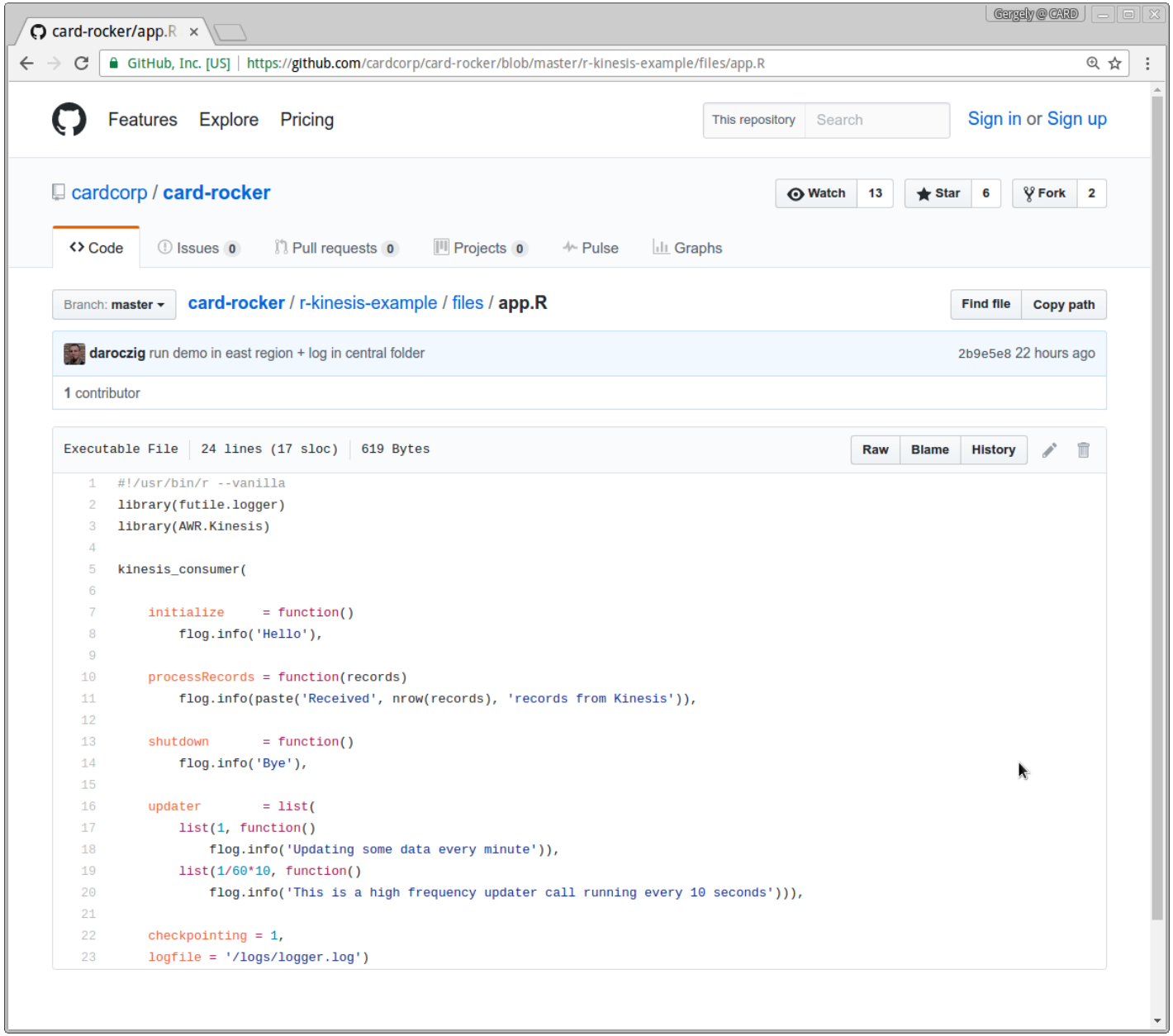
Dockerize your Kinesis Consumer



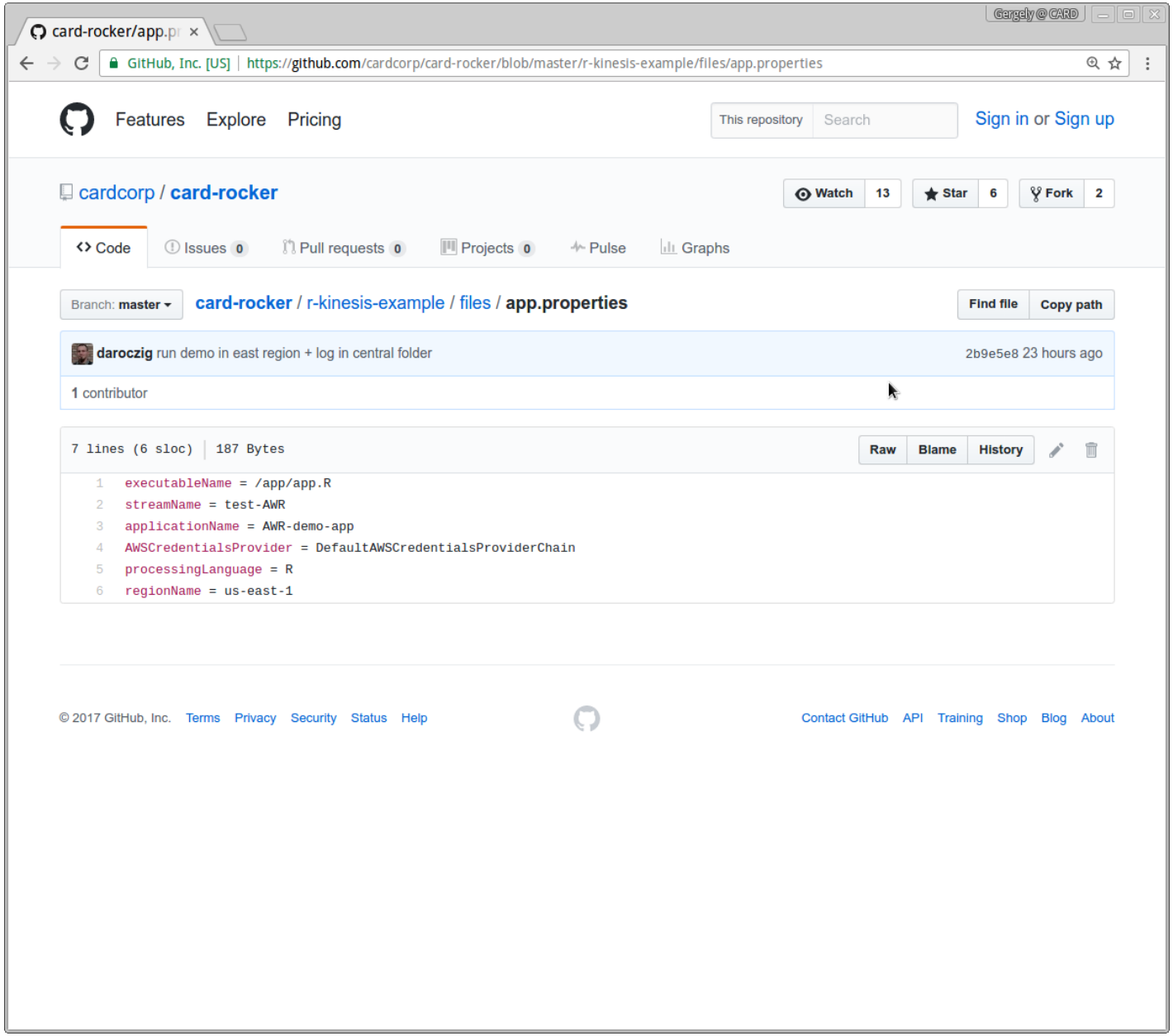
Dockerize your Kinesis Consumer



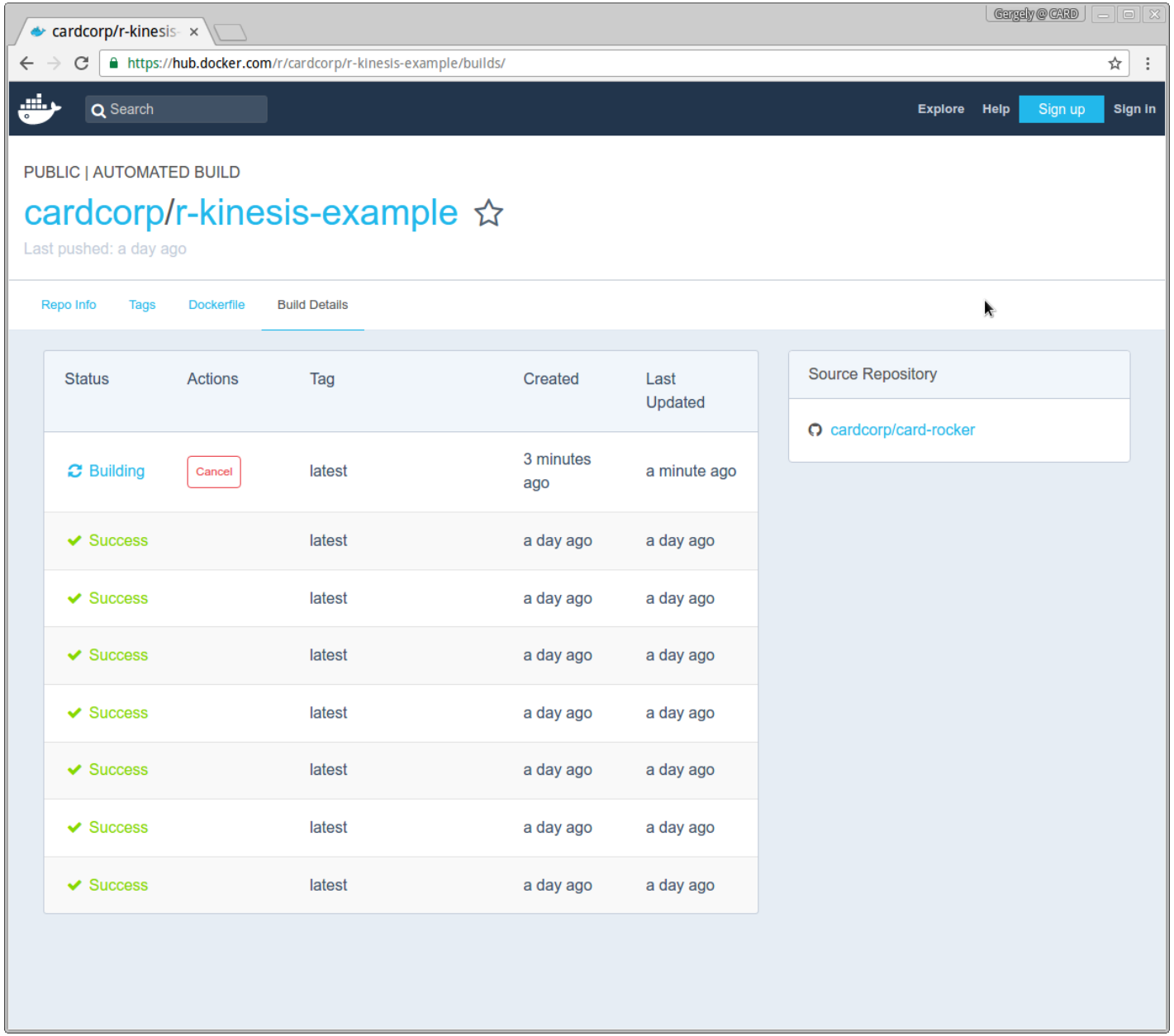
Dockerize your Kinesis Consumer



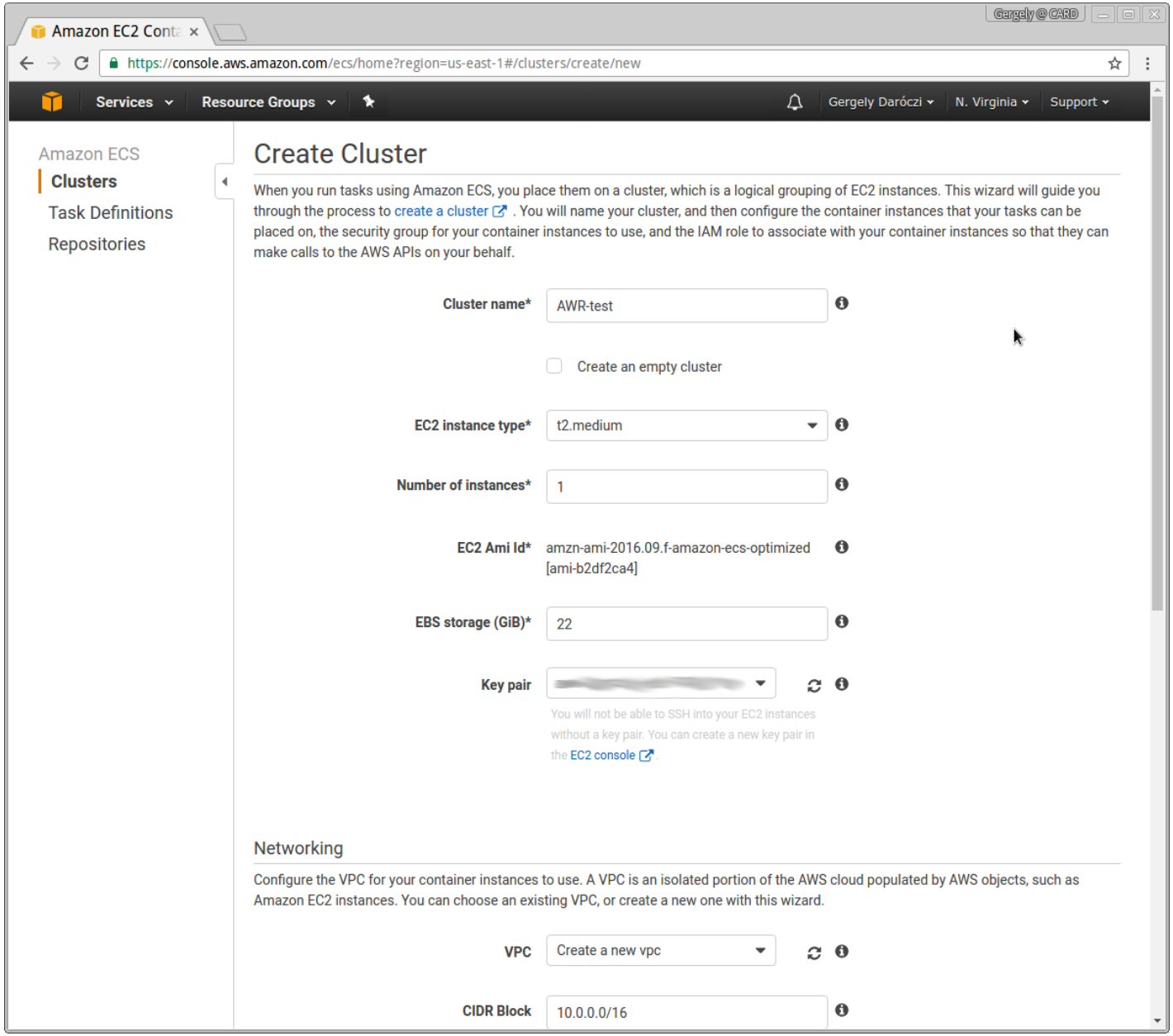
Dockerize your Kinesis Consumer



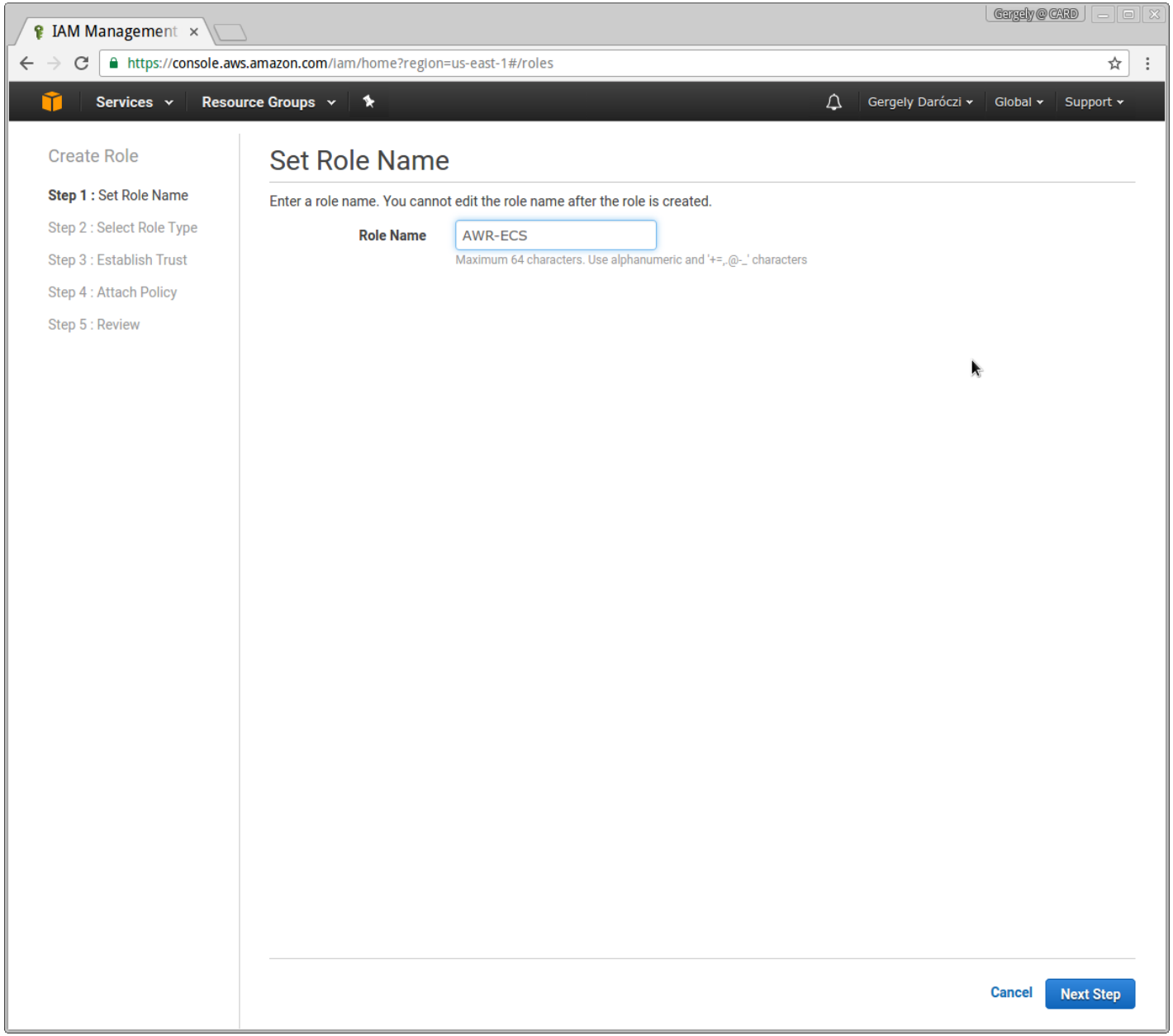
Put it on Docker Hub



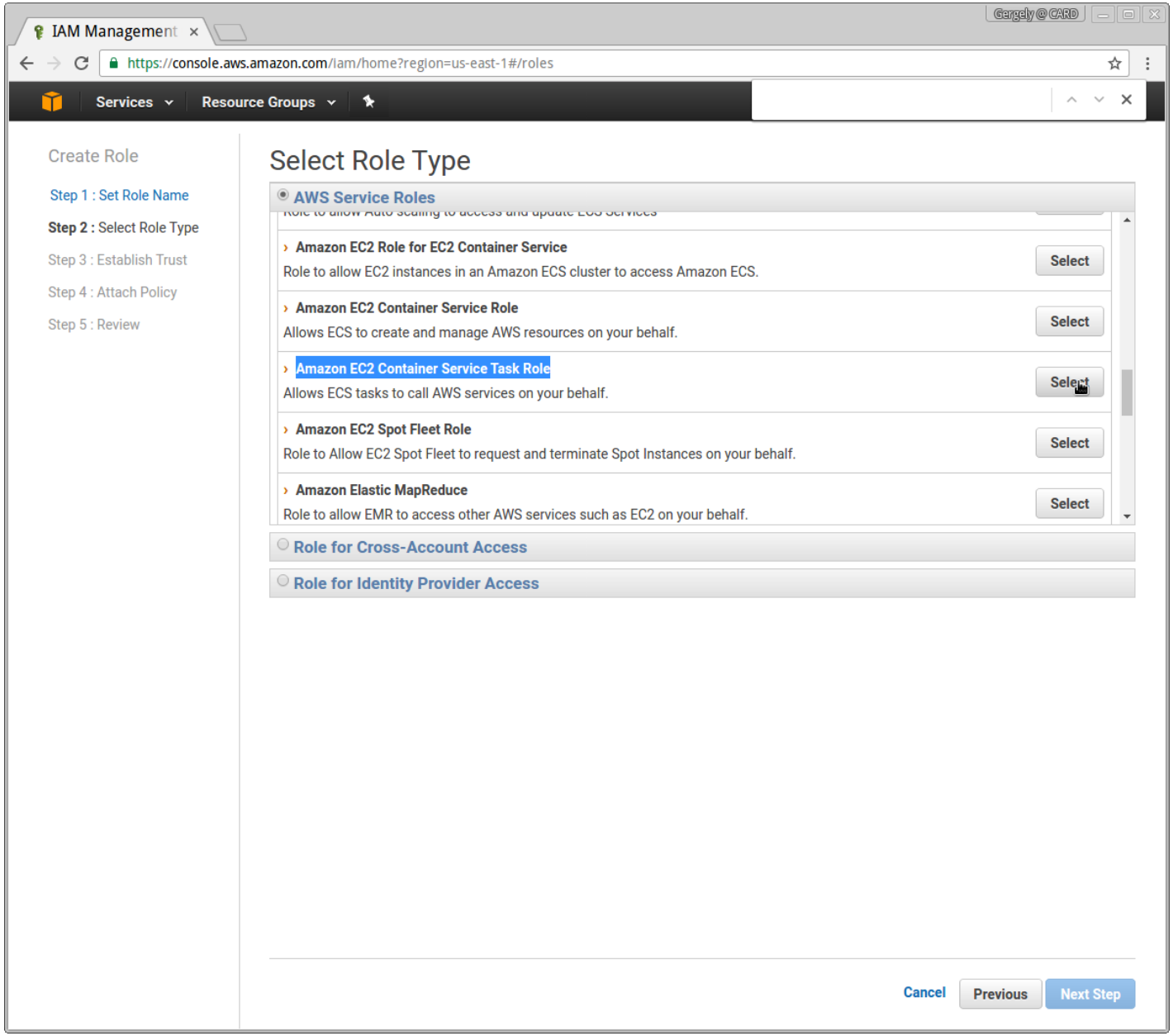
Create an ECS cluster



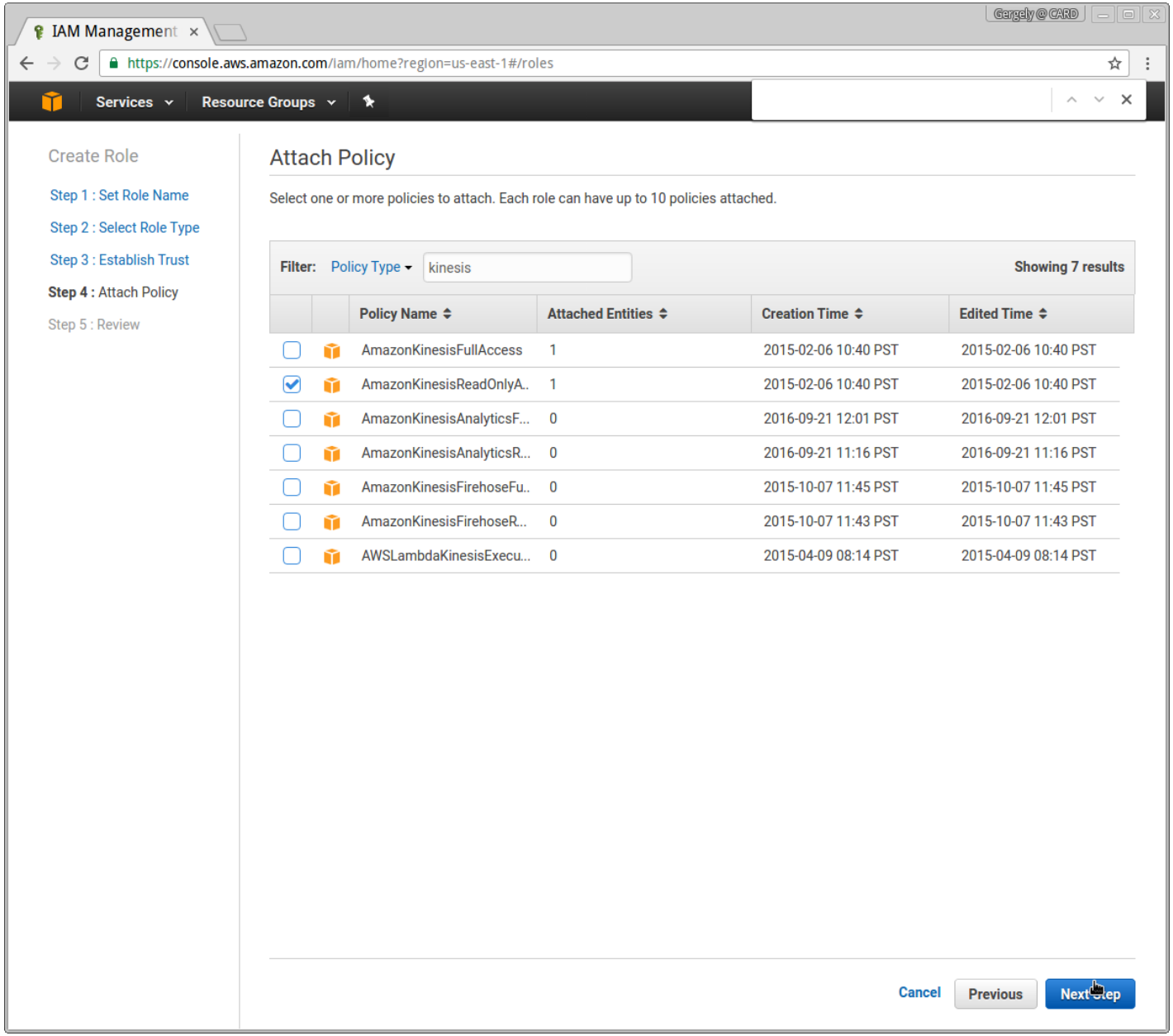
Create ECS Task Role



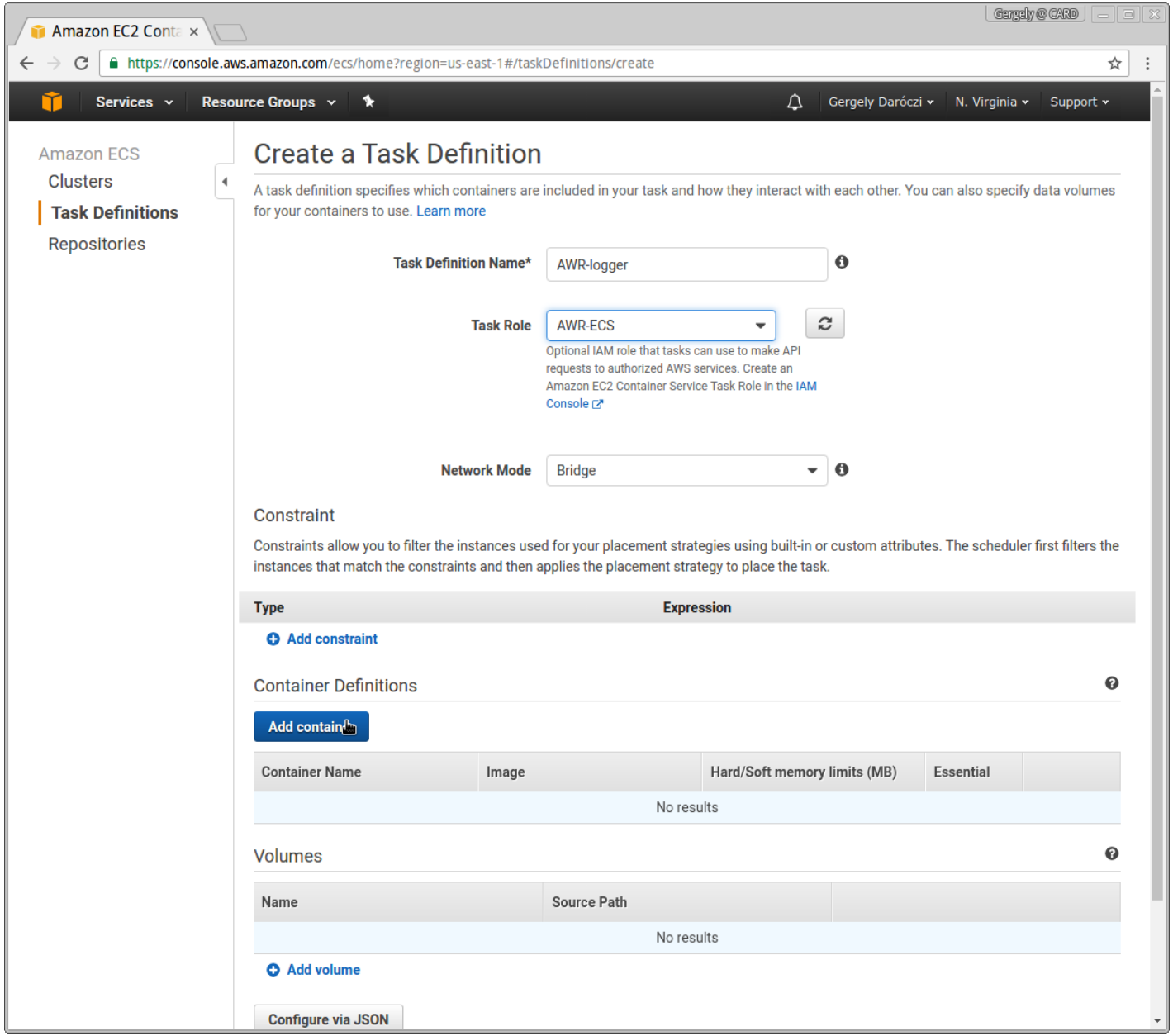
Create ECS Task Role



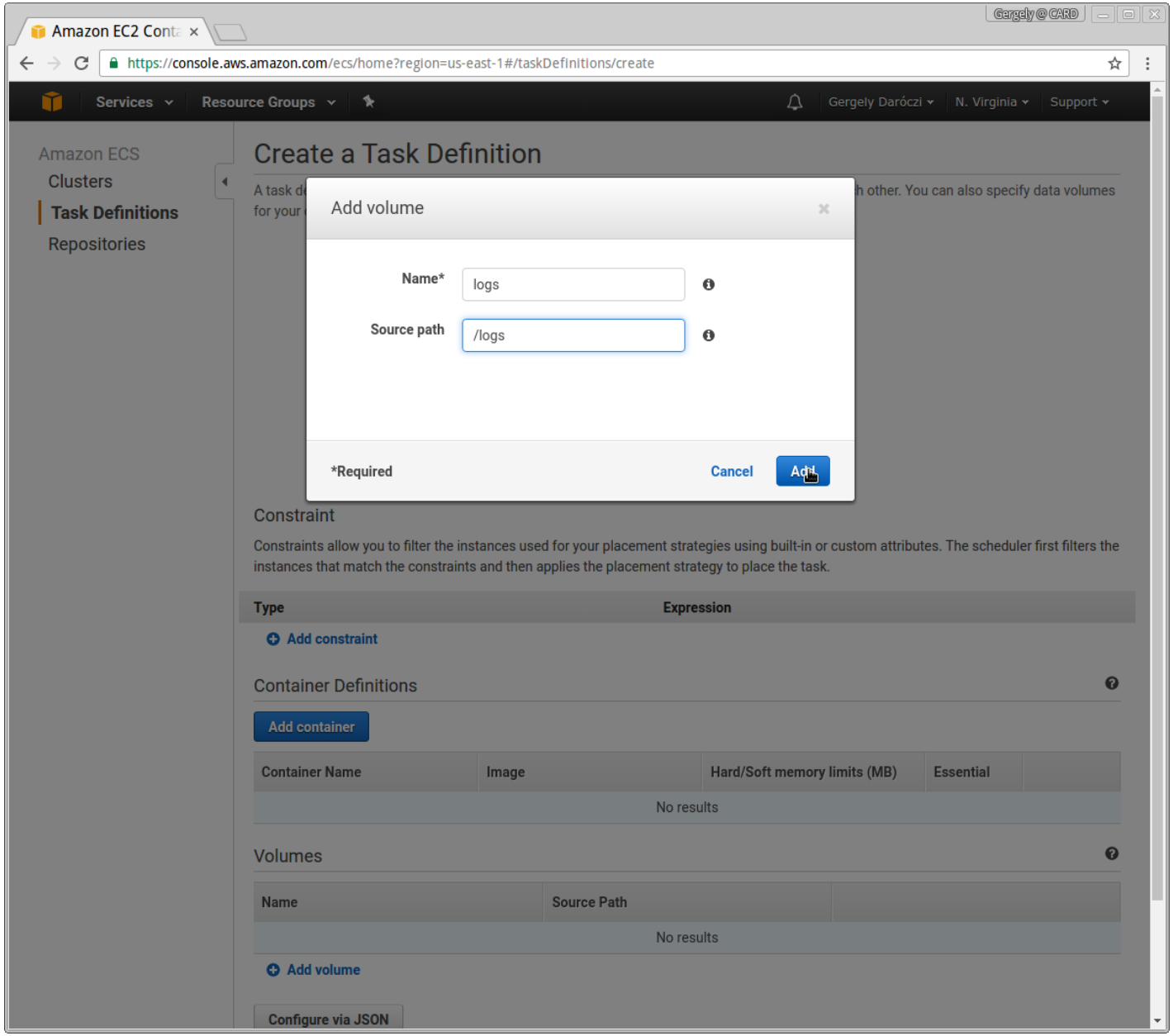
Create ECS Task Role



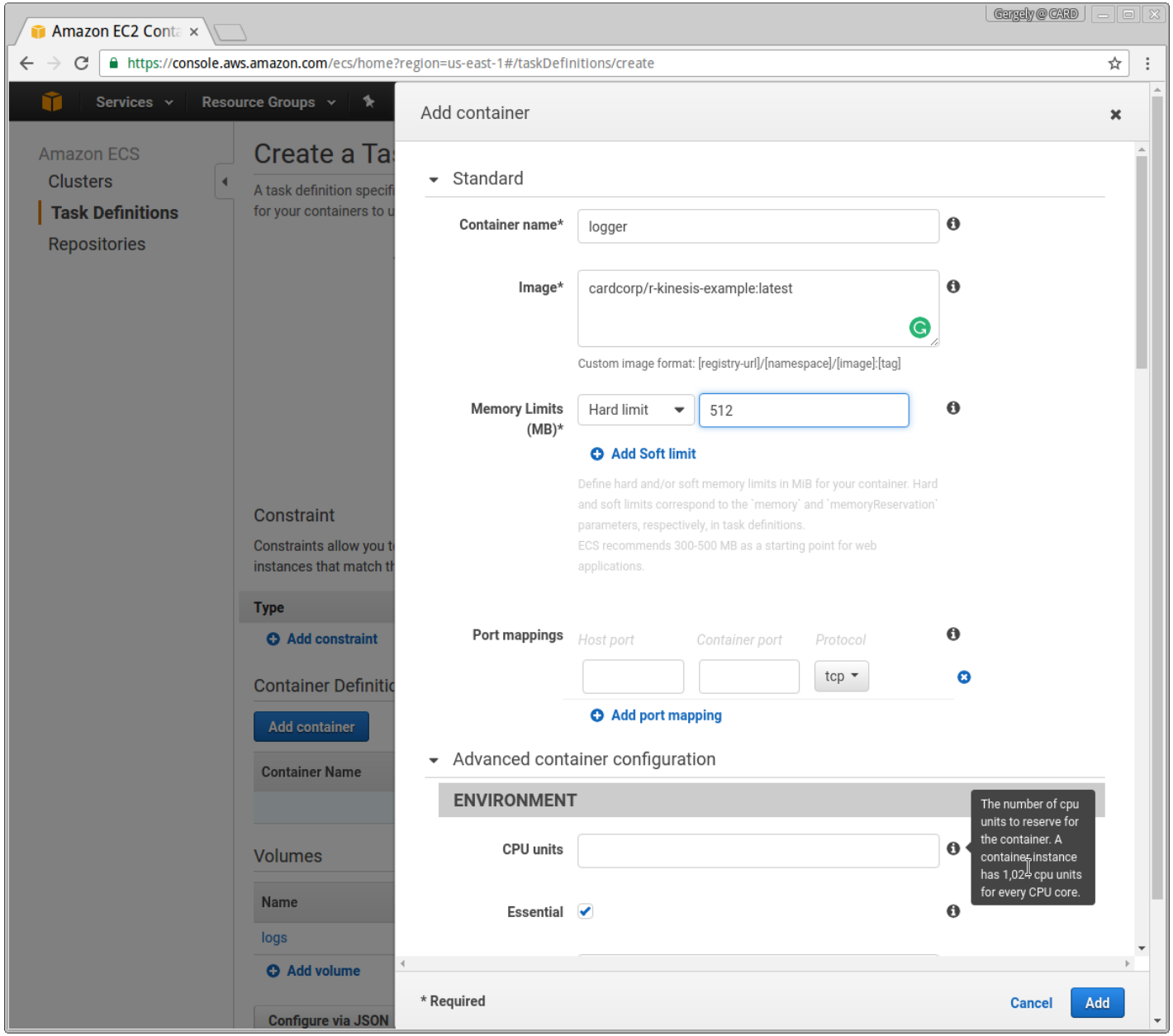
Create a Task definition



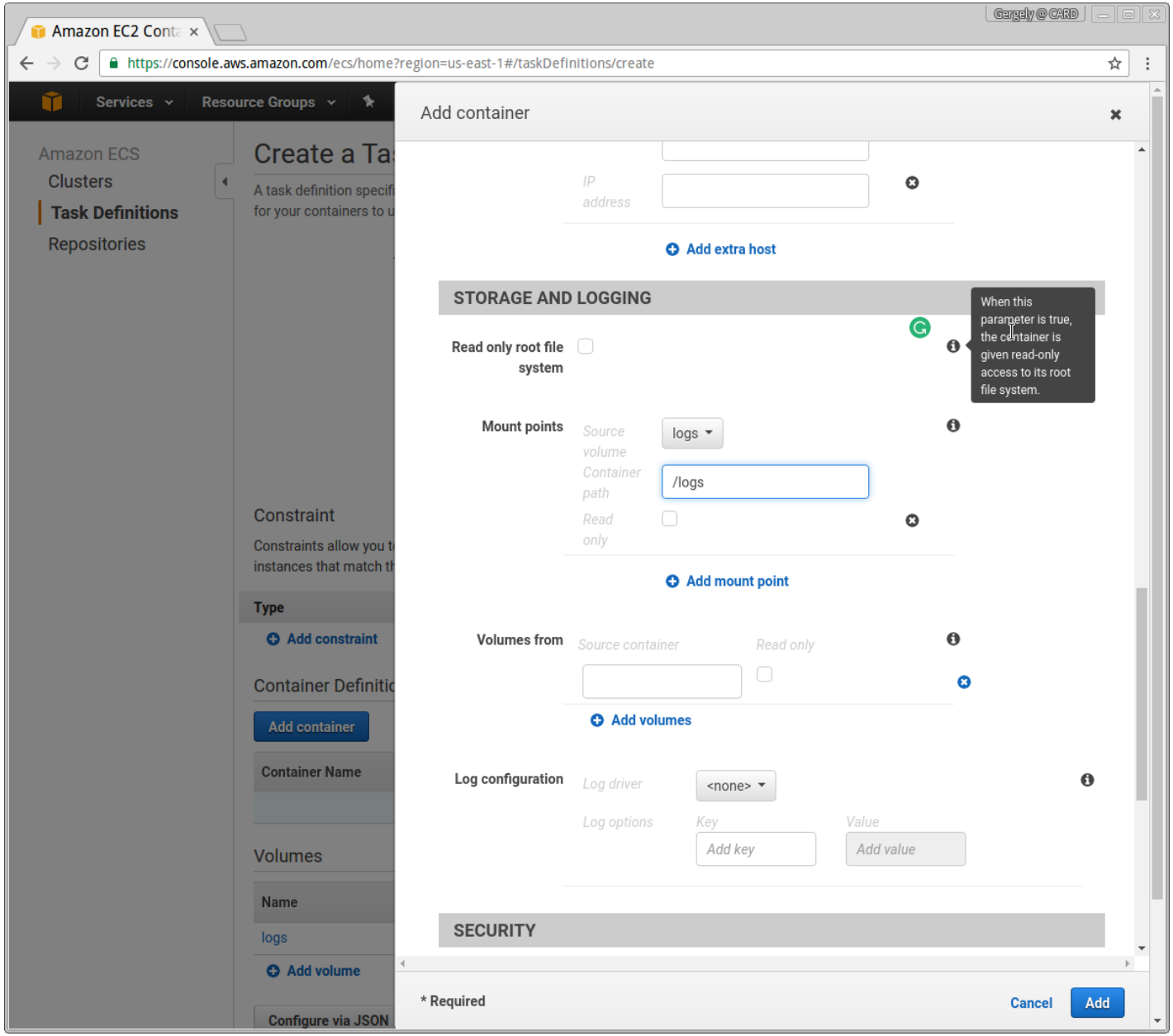
Create a Task definition



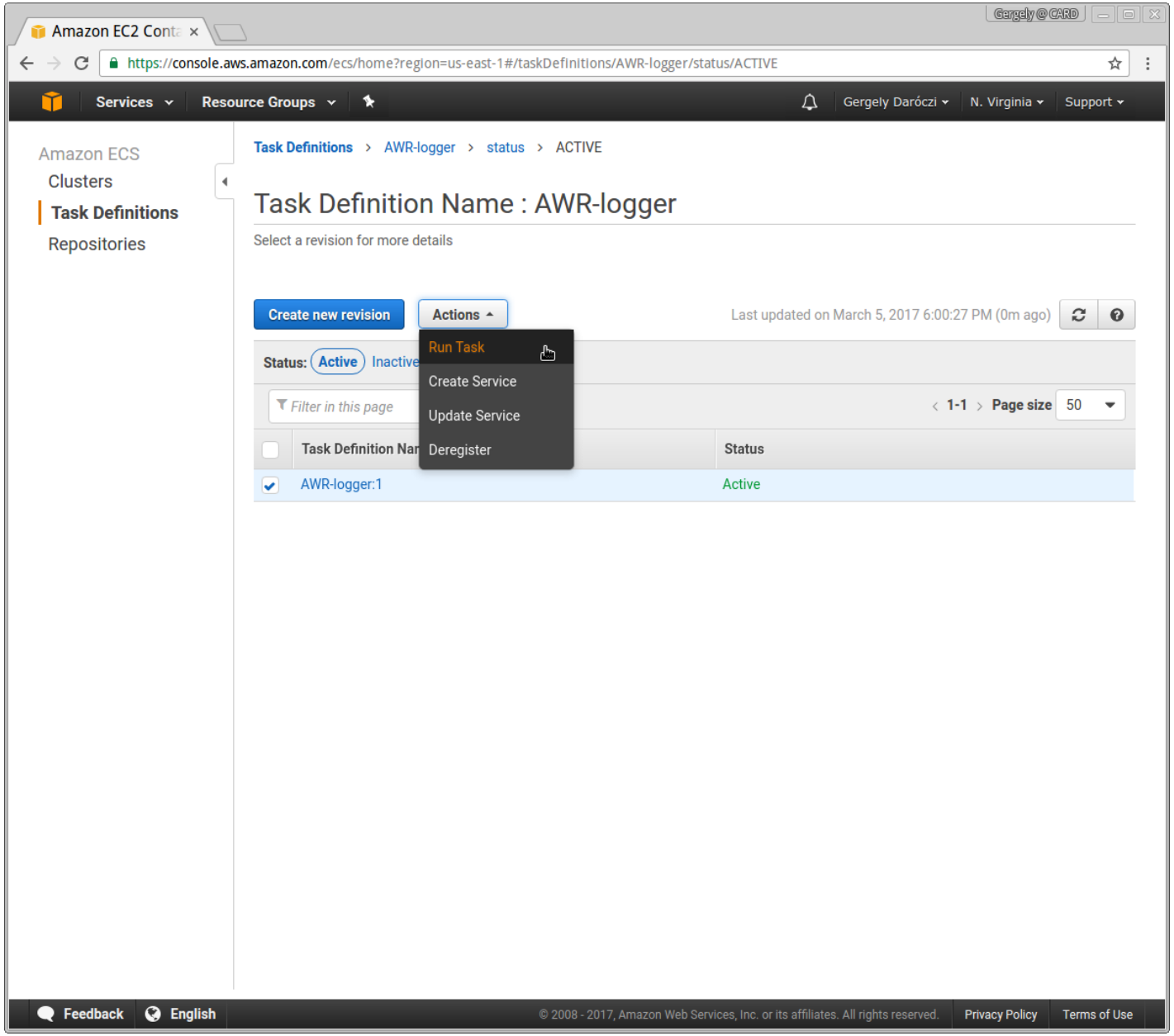
Create a Task definition



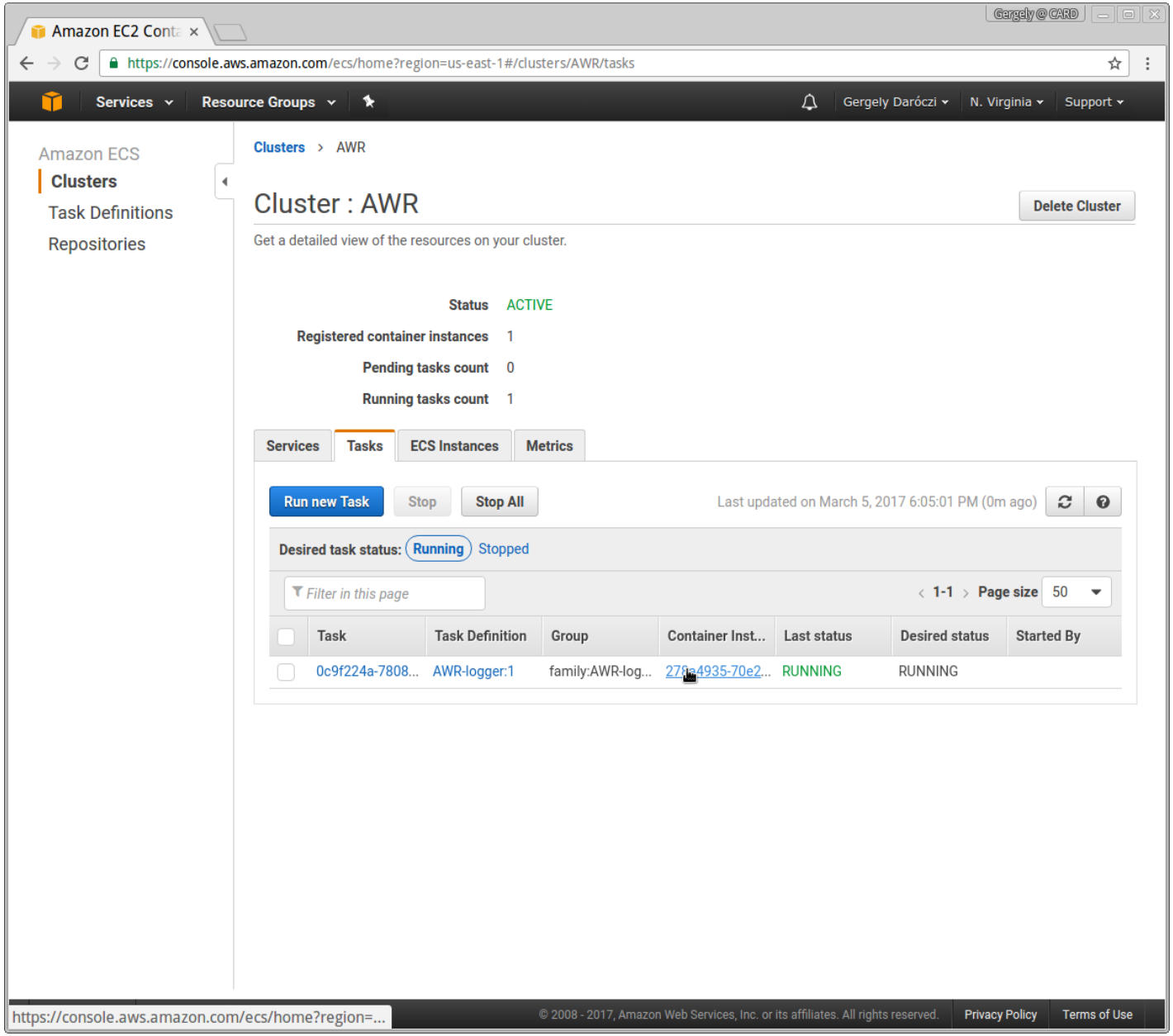
Create a Task definition



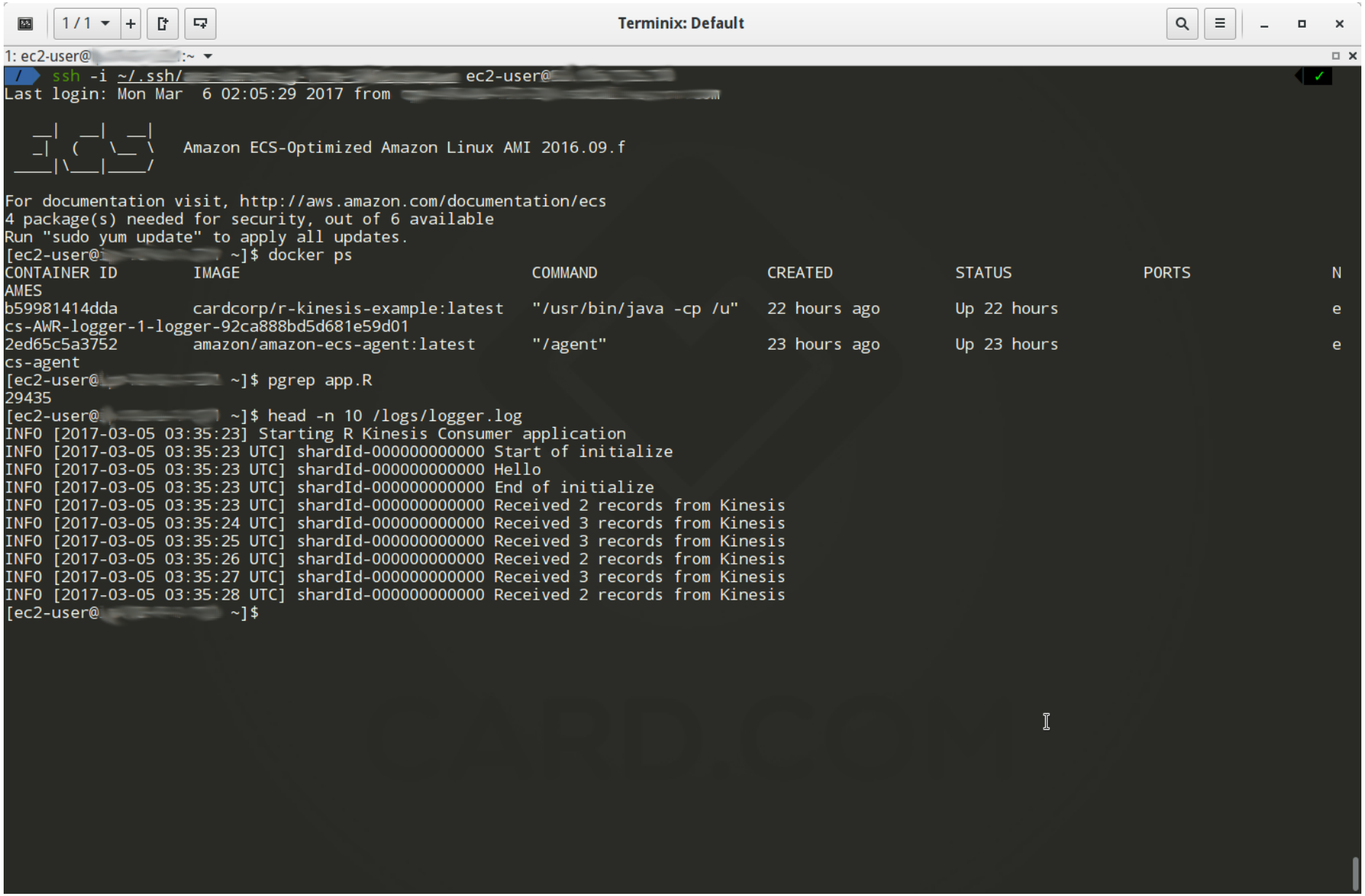
Run the ECS Task



Run the ECS Task

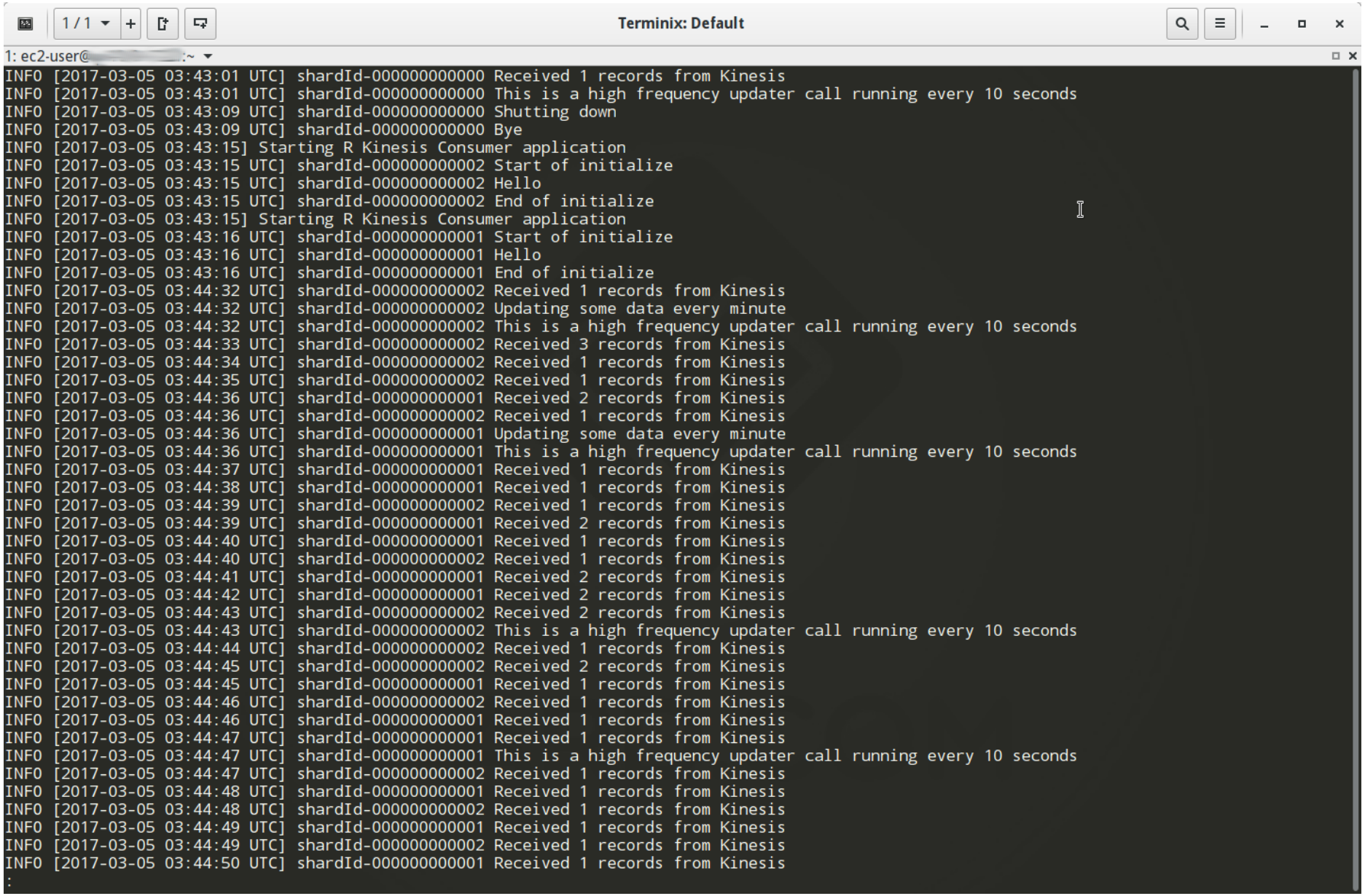


Run the ECS Task



Scaling the Kinesis Consumer up

EARL
ENTERPRISE APPLICATIONS OF THE R LANGUAGE



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 commiting credentials etc to the repo

Problems:

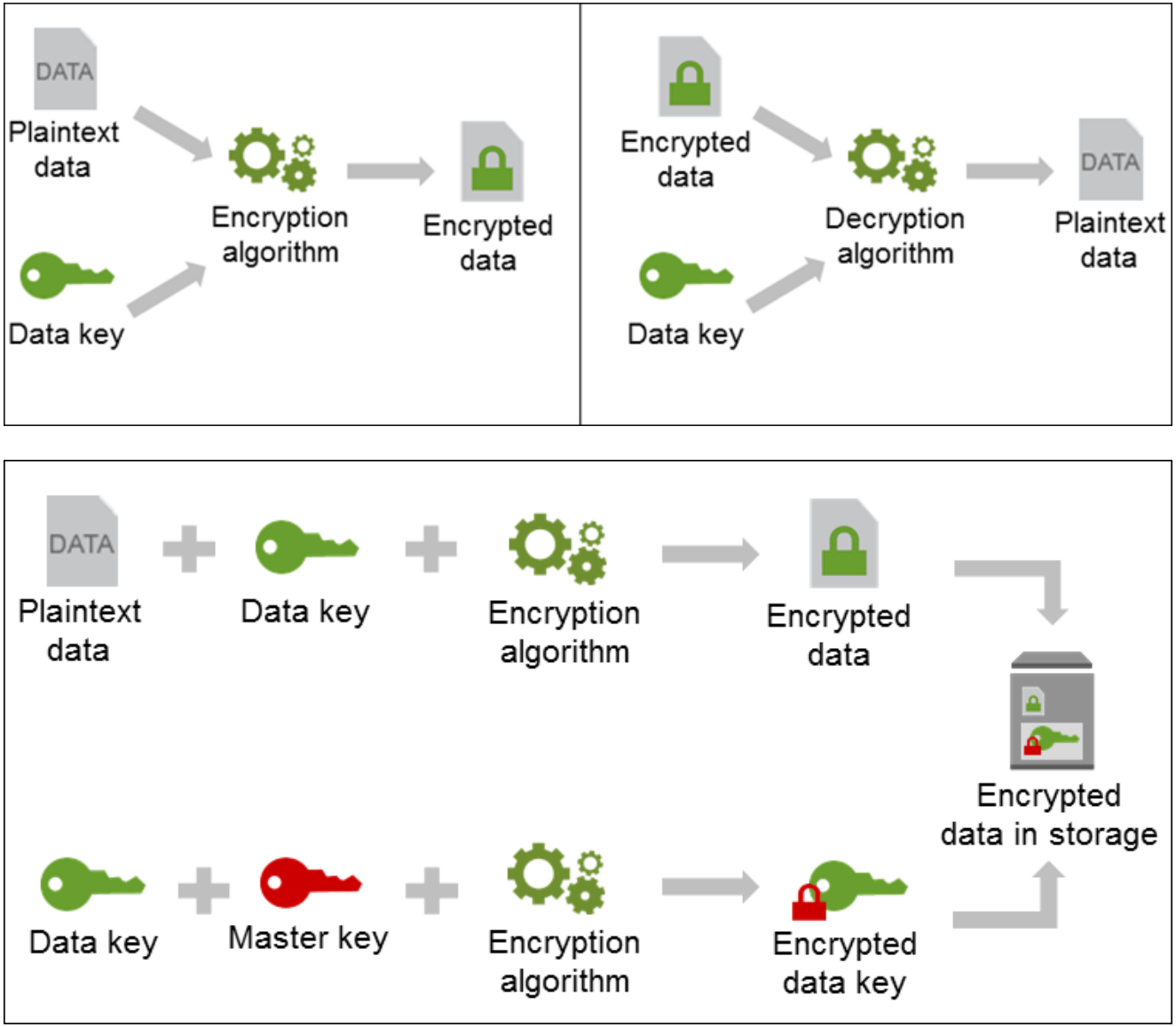
- How to store credentials in the Docker images?
- Where to store the Docker images?

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 commiting 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

EARL
ENTERPRISE APPLICATIONS OF THE R LANGUAGE

- encrypt up to 4 KB of arbitrary data:

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

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

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

- generate a data encryption key:

```
> 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?

EARL
ENTERPRISE APPLICATIONS OF THE R LANGUAGE

```
## 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
## https://github.com/sdoyen/r_password_crypt/blob/master/crypt.R
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))
```

Example "Production" Consumer App

EARL
ENTERPRISE APPLICATIONS OF THE R LANGUAGE

```
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('AQECAHiiz4GEPFQLL9AAON5TY/lDR5euQQScpXQU9iYTn+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

EARL
ENTERPRISE APPLICATIONS OF THE R LANGUAGE

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
```

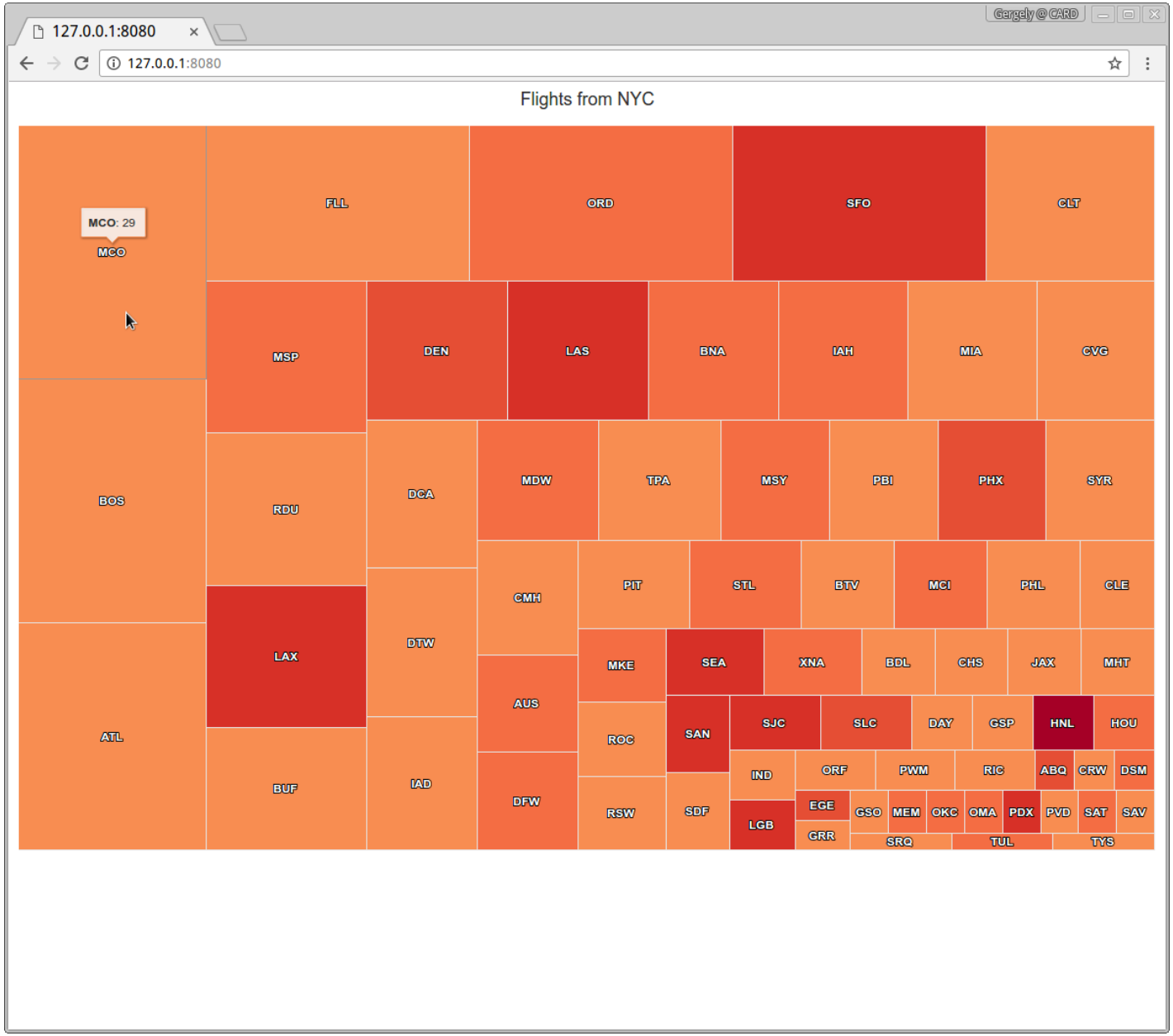
Build and push to ECR:

```
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
```

Shiny Dashboard

```
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)
```

Shiny Dashboard



Technical Details

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

```
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

What's Next?


```
> library(rJava)
> kc <- .jnew('com.amazonaws.services.s3.AmazonS3Client')
> kc$getS3AccountOwner()$getDisplayName()
[1] "foobar"
```


AWR
Because "S"
is so 1992.