Daniel Rajchwald
Andrew Reece

This is a record of the work we've done on our project.

09 NOV andrew:
set up a to-do list organized by the big categories of work to be done. [link]

10 NOV andrew:
spent the past couple of days figuring out how to customize the twitter streaming api, and how to get it to play well with python (and with spark, kind of). reading lots and lots on twitter streaming, spark streaming, and how/when python has a usable interface. wonder why python lags in the dev queue behind java and scala for spark?

just discovered in the spark fine print that, as of 1.5.1, spark streaming doesn't have a python implementation of its direct-to-tweet-stream ingestion service. bummer. thought that would be a major wrench in our plans...
but today i found this workaround, which uses the `stream=True` property on a `requests.get()` object to act as a generator, which can then feed into a Spark DStream. (blog writeup of that project here.)

that works, i tested it out and i can get tweets outputted from an RDD collect(). but it's a little hinky the way it's set up. it basically initializes empty RDDs and then uses a transform to pass them through the incoming stream from requests. it's not bad, but it's not great as an implementation. i think that's due to a combination of the author being creative but not totally savvy, mixed with pyspark just not really being designed to handle custom input streams.

11 NOV andrew:
pyspark **can** handle Kafka feeds, which i had been avoiding because it sounded scary. but today i took the dive and waded through all of the kafka docs to figure out how it all works. it's basically a fancy queue with good tolerance for distributed inputs and outputs, and it works well in the apache ecosystem. so i am still using the requests module to pull off tweets one by one from the streaming feed i have set up. but now that goes into a kafka producer. (there's a kafka-python wrapper.) next step is to hook up a spark DStream as a consumer.

- there's a lot on kafka-->spark (example in java),
- and on twitter-->kafka,
- and even some on twitter-->kafka-->spark,
- but almost nothing on **python(**twitter-->kafka-->spark**)**.
- this is python(kafka-->spark), which will be useful.
- (there's also python(twitter-->spark), as linked above, but we can do better than that, i think.)

- **python(twitter-->kafka-->spark)-->d3** seems to be totally undocumented (or un-googlable)
- there are a few examples of spark-->d3 out there (with an interesting use of spark-as-a-service (git repo here), which i hadn't seen before - that example is in scala though).
- This example (also Scala) goes kafka-->spark-->hbase-->d3, which is a nice reference. not sure if it's justified to add another middleman (ie hbase) into the mix for the level of data we're ingesting, but it's a good thought in case we need it. i also have never worked with hbase, which ups the intimidation factor a bit. i am just starting to feel good about kafka!)
- here are the official spark repo's examples for pyspark streaming use cases.
- we should be able to piece it all together.

Note: Kafka topics can be deleted with the terminal command:
`kafka-topics.sh --delete --zookeeper localhost:2181 --topic my-topic`
This only works if you have delete.topic.enabled=true in the server.properties config file.

Update: Apparently the .jar file for KafkaUtils is not included with the Spark 1.5.1 install, even though pyspark has a module that can use it. This took awhile to figure out. Now I have it working from spark-submit, but I still can't figure out how to add the jar to SparkConf() to make it work in ipython notebook. This is not a major issue, but a little bit annoying.

Update: We have a working pipeline up to spark output, sort of. Right now we have the Twitter streaming API feeding into `requests.get()`, which writes to a running Kafka producer (basically a streaming queue that listens for incoming data). Spark starts up as a separate script, and uses `KafkaUtils.createDirectStream()` to tap in directly to the Kafka stream. `createDirectStream` outputs a DStream, which can be operated on more-or-less like a normal Spark RDD. from there I apply some simple maps to filter down to the username and tweet in each entry, and then we print that output to stdout in the terminal. This works.

There are still some things to do before I feel confident saying that the pipeline is secure and functional up to this point:

1. **How much data can this Dstream/RDD thing see at any given moment?** Look into the BATCH_INTERVAL and BLOCKSIZE arguments that the hacky twitter-spark repo used. It's nice that we can print, but that just prints everything as it comes in. We want to collect aggregate measures per <time period> in order to get sentiment scores, etc. We also want to be able to pick the most-extreme-sentiment tweet in a given 1- or 5-minute window (this is an early idea but we should have the capability). So, figure out how much the stream RDD can get its hands on.
2. **How much needs to be done in Spark, versus in D3?** Would it be enough to compute sentiment scores per tweet, flag the extreme-sentiment tweets, etc, and then write to cassandra/hbase/flat file, and then leave to D3 to handle the final aggregation? D3 is pretty fast and by that point it would really just need to take an average. (We might

either record per-tweet sentiment sum, along with number of words, and then use that to take averages in D3, or we could record per-partition averages and word counts, and then weight them together in D3. In the most-work-for-spark case, we'd find a way to get all the data for a given epoch, compute the scores, and then D3 would just need to to pick up the final numbers. It'd need to do more than that anyway, I guess, as we want the ability to pull out full texts of tweets and other details. So, you need to think about how and where to store the tweets you're parsing in Spark.)

3. **How does partitioning work here? Also, is multi-thread concurrency helpful?** It's a bit cramped on my local machine because I think zookeeper/kafka take up one or two cores, and I only have 4. That means Spark only has 2-3 cores max to work with (and we probably want one free for system ops). But we should be able to assign 2-4 processes per core, if we're not multi-threading. So, figure out how much you have available in terms of resources, and how to allocate with this streaming spark regime.
4. **Look into using SQLcontext() for processing data**. This could make the sentiment scoring go a bit more smoothly. Also, how does **toPandas()** work?
5. **Figure out an intelligent way to stop the streaming context (shut down).** We currently have the python program that feeds the Kafka producer running for a fixed amount of time (set at 2 minutes right now). But when it stops giving new tweets, it's still giving timestamps (can we suppress this?), and so Spark Stream still has something coming in. Even if it didn't, I think it'd still just keep going until it received a command to stop. There's an `.awaitTermination()` method, which can take <number of seconds> as an argument. But if you don't pass it anything, how does it know when to stop? You can also just call `ssc.stop()` (ssc=spark streaming context), but you need something to trigger that at the right time. There may be some way to use `count()` to pay attention to how much new data are coming into the RDDs at any given time, as per this thread. I read it a couple of times and didn't really understand it, so there's some googling to be done there.
6. Once you have all this figured out, then would be a good time to do a quick check-in on AWS and make sure you can get all this up and running on an EMR cluster. might also want to bake an AMI that has all the stuff you need ready to go.

12 NOV andrew

Do we have an intelligent way to remove stop words from tweets before scoring on sentiment? Is that normally done?

For calling the Twitter-->Kafka feeder:

```
python /Users/andrew/git-local/twitter-in.py
```

For calling the Kafka-->Spark reader:

```
/usr/local/bin/spark-submit --jars
/Users/andrew/git-local/spark-streaming-kafka-assembly_2.10-1.5.2.jar
/Users/andrew/git-local/spark-output.py
```

Can we switch to Zeppelin from iPython Notebook? Seems like it's designed to integrate Spark…

15 NOV andrew

To-do (little steps):

- ~~**Learn how streaming spark saves to file**~~
- ~~**Cross check search-terms.txt with this list of candidate social media tags**~~
- ~~**Figure out how to adjust the batch size of each stream pull (eg. try to grab X tweets per file write)**~~
    - The second argument to pyspark.StreamingContext() is the batch duration, ie. the number of seconds one DStream collects from the input pipe.
    -

Update: .saveAsTextFiles() saves each batch to a separate file. Which is kind of annoying, as we want D3 to read from the output, which should just be one json or csv file. (FWIW we can write to a single file by doing collect() first, but that kind of ruins the streaming aspect.) So now I'm looking into ways to save the output of the Spark stream to a database of some kind.

Current options appear to be:

- Save to SQL db, then use a PHP script to collect the SQL, which D3 calls. By calling the .php file, it runs the PHP script and then reads the output. This is very cool, I had no idea this was possible! Eg. d3.json('myscript.php',data()...)
    - Here's a link for Spark Stream → SQL (in Scala but still good PoC)

- Save to a "real-time" NoSQL-style db like HBase, then use the HBase REST API to call down as JSON. This is sexier and fits in the Spark ecosystem better, but I already know SQL and PHP, and I don't know HBase. So, I'll have a look but I'm not going to slog to figure it out. It looks like there are a few examples online, which is good.

- This post mentions it's better to have a middleman service interact with HBase then allow the client to directly query it, which I'm guessing means something like setting up a Flask app. That's a little annoying, another layer…

Update: I'm going to try using AWS SimpleDB (SDB). It's nothing fancy, just a schema-less database that seems to handle concurrent writes well enough. By calling `.foreachRDD(lambda rdd: rdd.foreachPartition(write_to_db))` after we're done ingesting, filtering, cleaning, and analyzing for sentiment, we write each partition's data to SDB, which persists on the AWS cloud. We'll still need Python/Boto to call it down as JSON for D3 to ingest as a document, but that's not so bad, I can run the service from my personal web server. (I think I recall setting up Flask on AWS is a pain? Should look into it again though.) Instead of a PHP file, we can hit the Flask server from D3 with a

`d3.json("/flask/directory/function/trigger", function(data) {...})` and we should be good to go.

Re: SDB, it's not the sexiest option, but I moved away from HBase (sexier) because I didn't want to have to have too many services running on my master instance all at once (I already have zookeeper and kafka and spark). I think zookeeper is designed to handle a bunch of services well, but I don't really understand how it works, and have mostly been treating it as a black box that lets me get Kafka up and running. Unless I get really inspired I'm going to leave all that as-is.

I also realized writing to an SQLite dump or even a normal SQL database won't work because of the concurrent writes that each worker will want to do. I'd need a local SQLite file on each worker, which wouldn't fly, and normal SQL doesn't handle concurrent writes. AWS SDB is the simplest setup that handles concurrency. The data we'll be storing isn't massive once Spark is done with it, so I don't need to worry too much about the schema or SQL vs NoSQL, etc.

16 NOV andrew:

To-do:

(1) ~~Set up Boto calldown of SDB on Flask instance (on personal server)~~
(2) ~~Write basic D3 script that hits Flask to get SDB JSON, show something in HTML~~
    (a) Once we have this, we will have completed the barebones full pipeline. That means we have full data flow from Twitter Stream-->Kafka-->Spark-->SDB-->D3.
(3) Next step is to make sure we can replicate all this on an AWS/EMR cluster.
    (a) Re: baking AMI with zookeeper, kafka, etc etc...maybe now is a good time to look into Chef? Ask Suchin about this. We need an AMI and startup script that will get everything running automatically.
(4) Look into setting num partitions and DStreams
(5) Build in fault tolerance and 503/420 error handling from Tweet Stream. You want to automatically reconnect in case the stream drops, but not so fast that you piss off Twitter. See their guidelines on this.

Once you have all that done, ie. robust full pipeline, then you can start with the next big projects:

(1) Designing output visuals
    (a) Map with tweet pings?
    (b) Running seismograph per candidate?
(2) Incorporating LabMT sentiment scoring into Spark parsing
(3) Porting the streaming version to static Spark
(4) Grail: Integrate both static and streaming as different kafka feeds, all going into the same Spark context
(5) Need to think about how to tell app which debate/timestamp you want to look at. How should it know to grab the stream for a real-time analysis, vs grab archive only for a past analysis?

(6) How will we store data so that users can go back in time when using the app? (Eg. what happened 10 min ago?) How can we provide functionality to make use of this storage?
(7) Move AWS creds to .aws/credentials and out of run.py on server

Update: We have the full skeleton pipeline working. That's exciting, but the next step is still structural - get everything set up on AWS, bake an AMI (and startup script), build in fault tolerance. Hold off on num partitions for now.

We also need to think about how we'll hook up multiple kafka feeds.

- In realtime, we have the every-minute s3 write from gardenhose, which isn't a stream like the Stream API is, but can still be treated like one (I think), since it's updating every minute. We might not need to use this, as the Stream API will deliver every tweet within our search parameters, so long as we don't exceed our rate limit. I think we are permitted to ingest about 1% of all Tweets (although Twitter no longer publishes those numbers), and seeing as we're only asking for a small subset of english language tweets we might actually get every tweet we want. In that case, we don't need the minute-by-minute gardenhose. Now that I think about it, I feel comfortable moving ahead on this assumption -- that all debate tweets within our search parameters are <= our rate limit on the stream API. That relieves us from having to hook up multiple Kafka producers.
- In retro-analysis, eg. the sep 16 GOP debate, we will only use the archived tweets, and no stream at all. So here, we don't actually need Kafka at all, we can just load directly from s3 (i think we can even load an entire set of s3 files with wildcard matching via Spark and treat it as one RDD. That's nice.

Make sure you record how many tweets are coming in per batch-size, so you have a number to report for your writeup (ie. handled X tweets per second for X minutes with efficient near real-time processing)

Thoughts on chef: Is it worth the learning overhead?
We need an AMI that loads with:

Zookeeper,
Kafka,
Spark, and
Python
(we may not need Anaconda, since it's only scipy, and then pandas, that makes for difficult installation).
A shell script that:
starts up Zookeeper,
opens a Kafka Producer and topic, and
starts hitting the Twitter Stream.
We also need it to start up the Spark analysis script (with a Kafka consumer).

That is a lot to load from scratch each time, so we want an AMI and startup script to run automatically. It looks like there's a way to do this with jar files on the web console, and I know it can be done with Chef, but I don't really want to learn how to use chef nor fiddle with jars. Look into other solutions, they must be out there.

Update: Looks like we may be able to at least start up the EMR cluster we want, complete with custom bootstrap installs, using boto. Who knew! Haven't checked yet to see if we can also bake the resulting AMI and reload. The newest version of EMR (4.x) seems to have moved away from talking about AMIs, and now describes everything in terms of "Releases". You can still bootstrap actions, though.

17 NOV andrew

Apparently boto3 is the new boto. I had thought it was just a python3 version developed concurrently, but it works for all pythons 2.6+ and is the only boto currently with ongoing development. This is important for us because only boto3 includes an argument in the EMR setup for AWS's new `release-label` parameter (it used to be `ami-version`).

Update: Minor annoyance - if I want to spin up a cluster with some nodes on spot pricing using Boto, then I need to know roughly what the going rate is for the kind of instance I want. (Usually I spin things up from the web console, where going rates are easily seen.) This has led me on a bit of a chase, but I finally found that the ec2 client for aws cli (as well as its boto counterpart, thankfully) has a `describe_spot_price_history` function that will return recent spot price rates. So before spinning up a cluster, I'll run this function to get the recent 10 or 20 spot prices for the instance type I want, average them, and put in my bid for, say, 20% higher than the average. (Maybe with a minimum of 0.03 or something low like that, in case the spot price is really really low, like less than 0.01...then 20% wouldn't do much.)

See 18 NOV update on this.

Update: This looks like a really useful walkthrough for setting up Zookeeper/Kafka on an AWS cluster. This may be a case where we actually use the full 3-level master/core/task node structure, as we could have zookeeper on the master, kafka on one core, spark master on another core, and spark workers as the task nodes? You should look into this more - it may not actually be necessary to put kafka and spark on the same cluster. You could have a spark cluster and a zookeeper-kafka cluster. Or maybe have multiple master nodes on a single cluster? This slide deck has a sketch of a similar architecture (Spark/Kinesis)

Update: If you set "Requester pays" on an s3 bucket, you need to use `aws s3api` instead of `aws s3`. If you try to go in through normal s3 cli and you're not the bucket owner (eg. I tried to add Daniel) it will reject those credentials. If you want to download something with s3api, you need to use `--request-payer requester` flag and string. Otherwise if you just want to list a bucket you don't need that flag, but you still need to use `s3api`. I think if you're the bucket owner you can use either `s3` or `s3api`.

18 NOV andrew:
Mainly struggling with AWS linux and bootstrap builds today.

Among other things, default python is 2.6 (or at least it is for previous-generation instances like m1.small, which i'm using for testing). 2.7.9 is actually installed on these instances, but I need to run `sudo alternatives --set python /usr/bin/python2.7` to get it to switch.

Also, this weird thing just happened where boto3 stopped working, which in turn seemed to be related to a pip version issue. But when I upgraded pip, it didn't upgrade to /usr/bin but /usr/local/bin. When I tried to do pip install --upgrade boto3, it barfed due to no sudo. And apparently sudo doesn't have /usr/local/ in its path. (I didn't know that sudo uses a different $PATH than non-sudo commands.) Luckily there's an SO post for this, and it showed how I can just run sudo `which pip` install…, which works. Sheesh!

In other news, I have a bootstrap script that installs zookeeper and kafka and a few other things (upgrades python, installs boto3, etc). There's a Kafka server.properties file which I've put up in our s3 bucket, and the startup bash script will grab that and replace the default one.

> NOTE: I think we need the Kafka server startup and the Zookeeper server startups to run in separate bash scripts, as they occupy a terminal instance once they're running.

Do we need to run Kafka on a core node, or can we just run on the master? If we have a master machine with enough available processes, we should be able to run spark, zookeeper, and kafka all on it, and use the core-level nodes all for spark. This would save at least one step, which is finding a way to identify the ip address of the master node during the bootstrap process, as we'd need that (I think) to tell a Kafka server on a core-level node where the zookeeper is. This requires accessing the `aws emr` cli (either by bash or boto) and finding a clever way to pull out only the master node ip address. That's non-trivial, so I'm going to try to put Kafka on the master.

Note: Looks like m1.mediums come with JDK 1.7 installed. I think that should be fine...do we need 1.8 for the kind of sparky stuff we want to do?

Update (spot pricing): Now our cluster initialization script is smart enough to find the AWS region (eg. us-east-1a) with the lowest average spot price over the past hour, and it enters its bid accordingly at a 20% markup from the average. In case the spot price average is really low, like less than 1 cent, it sets a minimum bid at $0.02 (because 20% of, say, 0.001 doesn't do much). **We may want to significantly boost our minimum bid on actual debate nights,** as it's possible server prices will be higher then if more people are doing stuff on AWS.

Update: Now using 4 startup scripts (3 bootstrap, 1 “step”). The bootstrap scripts upgrade to the latest python and install zookeeper and kafka, and then start zookeeper and kafka instances running. The step script starts a kafka topic called “tweets”.

Let’s figure out how to use Ganglia. The new AWS Releases don’t let you install it the way they used to, but this fresh SO post (from yesterday) has a bootstrap script on s3 that does it: `s3://support.elasticmapreduce/release/4.x/ganglia/install_ganglia_emr-4.0.0.rb`

Note: If you start up a free tier instance in EC2, it will put you on a VPC. You need to go into your VPC Services tab on the AWS console, go to “Your VPCs”, select all VPCs and then do Actions-->Edit DNS Hostnames and select “yes”. (This is not an issue when starting an EMR cluster - but you spent a bit of time today in regular EC2, and came across this issue.)

Update: Getting bootstrap and step scripts to work on EMR is frustrating. If there’s a problem it just hangs and doesn’t let you know (not even in the s3 log files). Slowly making progress.

As I understand it, bootstrap actions are applied to all nodes in a cluster, and so it makes sense to use bootstrap scripts for things like making sure every node has the right python modules installed. On the other hand, step actions are only applied to the master node, and these are usually jobs that you give to the master node to delegate throughout its cluster, once all the initial setup is complete.

But I’m pretty sure I just tried to do a series of step actions (so, master-only) and they failed because the non-master nodes said they couldn’t find the s3 .sh scripts. That’s weird. So I’m going back and using the “Run if” utility in bootstrap, which lets me set a condition for running a script. (The most common usage, and the example provided in the run-if source code, is to check if a node is master or not, which makes me a bit suspicious about the official AWS blog that says steps only run on the master. Or maybe it’s just that step actions start on the master, but delegation is occurring in some way I don’t understand? Anyway, running the scripts again with this run-if approach now.

Update: Since it takes 20-30 minutes to provision resources each time I start up a cluster on spot pricing, I’ve been using the time to debug the cluster startup script I wrote with boto3. (So far I’ve been starting clusters with the web console.) I think I finally have it working now, which is good. The bootstrapping section is still failing.

Update: I know why the bootstrap is failing, at least I know one reason. The run-if ruby script that the AWS EMR folks provided is purposely busted now that they’ve moved into AWS release 4.x. I did find a bespoke version of the “run-if” command (for Zookeeper, no less), hiding in the Accumulo bootstrap script, and so I’m trying a modification of that as a way of doing master-node-only installs. (It uses some bash-fu to determine whether we’re on the master node.)

Reminder: Make sure to check that the Release Version of EMR you're running has Spark 1.5+.
Reminder: Install Ganglia via bootstrap!

Update: I figured out why the Kafka server startup script kept failing. It always gave me the same error, which I couldn't make sense of:

```
Exception in thread "main" java.io.IOException: Error opening job
jar: /mnt/var/lib/hadoop/steps/s-SVZXPM6KC74B/kafka-start.sh
      at org.apache.hadoop.util.RunJar.run(RunJar.java:160)
      at org.apache.hadoop.util.RunJar.main(RunJar.java:136)
Caused by: java.util.zip.ZipException: error in opening zip file
      at java.util.zip.ZipFile.open(Native Method)
      at java.util.zip.ZipFile.<init>(ZipFile.java:215)
      at java.util.zip.ZipFile.<init>(ZipFile.java:145)
      at java.util.jar.JarFile.<init>(JarFile.java:154)
      at java.util.jar.JarFile.<init>(JarFile.java:91)
      at org.apache.hadoop.util.RunJar.run(RunJar.java:158)
      ... 1 more
```

I was also puzzled that my `/home/hadoop/ directory`, where I thought I'd instructed the bootstrap scripts to install `zookeeper/` and `kafka/`, was always empty. It turns out that EMR stashes any bootstrap sequences in their own folders in `/mnt`, `/mnt/var/lib/bootstrap-actions/<bootstrap#>`. So kafka was in `/mnt/var/lib/bootstrap-actions/3`, and zookeeper was in `/mnt/var/lib/bootstrap-actions/2`. But my scripts starting up kafka assumed that we were all in the same directory (which I thought was `/home/hadoop`). I adjusted the bootstrap scripts to explicitly move all the stuff I want into the `/home/hadoop` directory. Tomorrow morning I'll see if that solves my problems. Buggy day!

20 NOV andrew:
Once cluster bootstraps correctly:

1. ~~Check to make sure pyspark is installed and SparkContext() is available.~~
    a. If so, remove findspark and pyspark imports from spark-output.py
2. ~~Try starting kafka-server.sh with screen. This may prevent the problem it seems to have with starting kafka-topic afterward?~~
    a. NB: Screen can only run from tty, so bootstrap bash commands can't use it
    b. The answer was to use () and & together. See this lifesaver of a post.
3. ~~Download kafka-topic.sh to a local folder in bootstrap and run the .sh as a step~~
4. Make sure the version of Spark that loaded has pyspark.streaming.KafkaUtils
5. Make sure you're running kafka on localhost:9092...or if not, make sure that Spark and its children can all get to Kafka with the line: `KafkaUtils.createDirectStream(ssc, ["tweets"], {"bootstrap.servers": host_port})`

    a. You can use the private or public IP of the master node instead of localhost if need be
6. Same thing goes for twitter-in.py, `kafka = KafkaClient('localhost:9092')`

So the main next step is to make sure twitter-in and spark-output are working the same as they do on your local machine.
Once that's working, send in twitter-in and spark-output as step jobs to the cluster startup script (in that order, after everything else is set up).

Important point to consider:
For streaming/real-time, this approach is all well and good. We can start up the AWS cluster shortly before a debate, give it time to bake and fire up all the nodes, and then shut it down afterward. But what about the static case? We want this to be a website people can just visit and interact with normally, not have to wait 20 min each time they want to use it. **What we really need Spark for in the static case is the number crunching, not the serving of content.** Once all the sentiment data and topics and whatever else have been computed, we have it all in a database and can serve normally with Flask and the client-end D3 app - we don't need Spark. So we should crunch for each debate with the tweet archive, store the data, and then that's the end of Spark for the archives. We might have a cron job in place that runs the python boto script that starts the EMR cluster at a set point before each debate, in the streaming case.

Remember ps aux | grep zoo !

Update: Our talk with Kevin cleared up (hopefully) our expectations for our project's performance goals. To recap, our proposal (under Objectives 4.c) reads:

> A note on "speedup": The performance gains we address with parallelization in this project are not so much a measure of quantity as category. Without parallelization, the amount of incoming data (on the order of 1e2-1e3 semi-structured JSON files per second) would make this project intractable. Even so, we will still employ best practices when computing in Spark, eg. minimizing shuffles and calls to collect().

In other words, this project is not so much about comparing speed gains as it is making good use of Spark and its architecture to let us do this real-time debate tracking we're interested in. It could be that it's possible to configure all this to run on a single machine, with some really interesting optimization that makes it also able to run - but that's not the focus of our project. We're implementing a parallelized streaming input parser that does some basic (or in the case of LDA, not-so-basic) computations and feeds the results to an interactive web app. That all is laid out pretty well in our proposal, and we're planning on sticking to those goals. It's a lot of work as it is!

We should also note that the architecture we've built here is scalable to extremely large input streams, which is another advantage of designing with Spark/Kafka streams.

Update: I think I found out why the Steps always fail. They are actually looking for a .jar file, as in a java archive! (The language of bootstraps and steps is always "jar" but usually you can pass it whatever - shell, ruby, etc. Good news is that there seems to be a "command-runner.jar" that lets you execute shell scripts...I found evidence of this on the very last line of this page (btw documentation for AWS bootstrapping is really bad). So I'm trying it out now. Fingers crossed.

Update: Finally! Kafka topic is up and running on startup now. The command-runner was a big piece of the puzzle, but there were a few more:

- The shell scripts I ran as actions (kafka-start and kafka-topic) both needed execute permissions on owner and only had rw- (so chmod 755 in the updates.sh bootstrap fixed that)
- The other big problem was that starting the kafka server (which is the action we take before instantiating a kafka topic) was eating up the bash prompt, meaning it took up the entire foreground and disallowed further commands. I eventually found this post, which notes that combining () and & lets you run a command in the background and then execute commands subsequently in the same bash session. Once I did that, the second step action (starting the kafka topic) ran successfully.