cloudera

Serverless for data scientists

Mike Lee Williams • @mikepqr • mlw@cloudera.com

github.com/williamsmj/serverless-for-data-scientists

I'm Mike, and I'm a research engineer on Cloudera's machine learning services team, Fast Forward Labs.

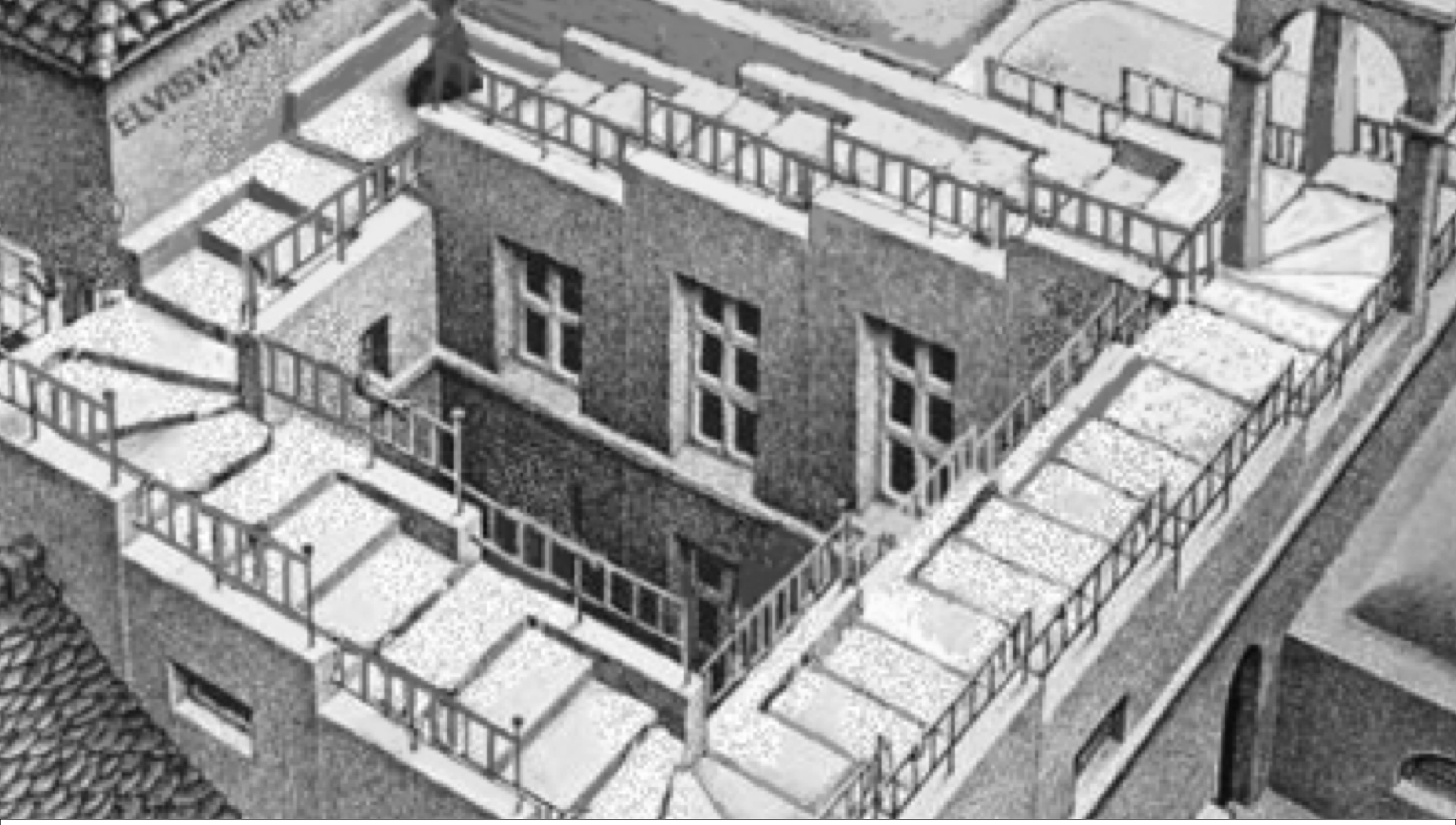
Today I'm going to talk about serverless. If you work in operations or mainstream engineering then you're probably familiar with this idea already. I hope from today's talk you'll come away excited about its application to data science and machine learning.

If you're a data scientist, then I hope this will be a useful and interesting tour of an idea that's newish to you. If you're currently constrained by your computing resources, serverless is a light-weight alternative to setting up a cluster (or waiting for someone else to set up a cluster).

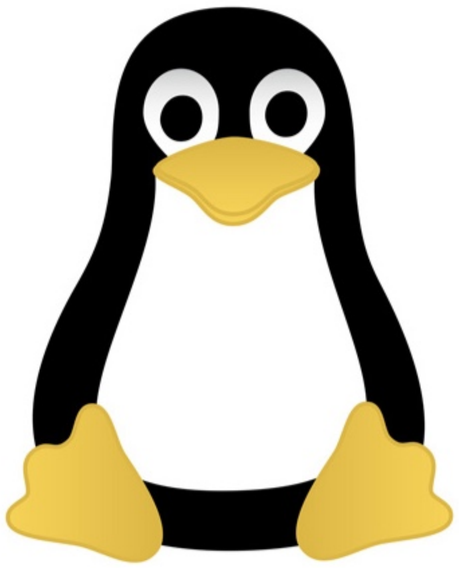
These are computers. It’s generally agreed that they were a bad idea.

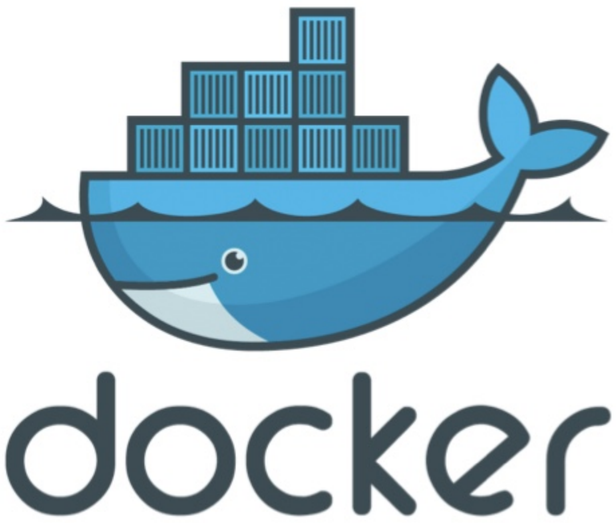
One of the wonderful things about the cloud is that the actual hardware is somebody else's problem. You're paying Amazon or Google or Microsoft to deal with that. They do a really good job, to the extent that you can largely forget about the physical infrastructure. From the point of view of the cloud user, it's like getting rid of your hardware completely.

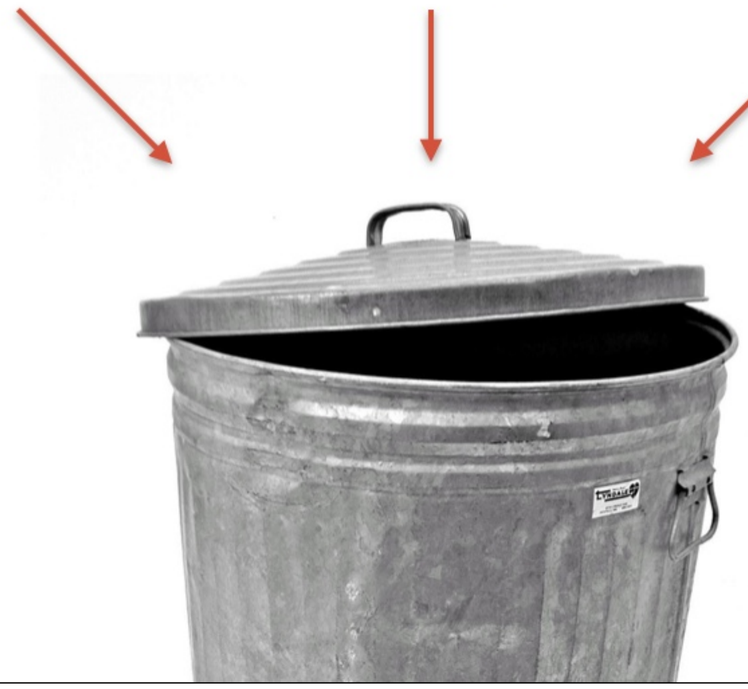
But moving to the cloud does create a problem. Not always, but it has a tendency to be a very efficient way of setting fire to money. One of the reasons for that is, despite the fact that we associate the cloud with the ability to scale dynamically, inevitably we end up with underutilized long-lived resources such as EC2 instances.



And we still have the problem of installing, configuring, securing and maintaining our operating system or database or whatever. This is a shame because for many of us it’s not what we want to do with our time. And for most of our employer’s it’s not how they make money. They make money by deploying business logic.







So what if we could not only get rid of our physical infrastructure, but also the next layer up: the operating system or the container. If we could do that then we could focus entirely on the business logic.

2 + 2 = ?

4

37 + 5 = ?

42

That is the claim, at least of serverless. Serverless works like this. A request come in to some kind of gateway service. This summons a machine out of thin air and deploys your application on that machine. The new machine figures out the response, sends it back, and then dies a noble death. Another request arrives and the same thing happens. Crucially, between these two requests, you have no infrastructure.

Or more generally, the amount of infrastructure you have scales almost linearly with your usage, and that linear scaling has an intercept of zero!

By the way, “serverless” is a really bad name, because you still have servers. In fact, you have effectively infinitely many of them. (In that sense, it’s a bit like “nonparametric statistics”, which is “nonparametric” in the sense that you have, well, lots of parameters, rather than in the usual sense of the word “non”, which is that that you have, well, none.)

AWS Lambda

Google Cloud Functions

Azure Functions

OpenFaaS

Generally, in order to derive the maximum benefits of serverless (i.e. making the operations problems somebody else's), you need to give money to one of the big cloud providers to use their serverless platforms. These are pretty much your options.

The odd one out here is OpenFaaS, which I won't be talking about, but if you have private infrastructure or particular needs, it might be worth looking into.

```python
# zappa/app.py
import datetime
from flask import Flask
app = Flask(__name__)

@app.route("/time")
def time():
    return str(datetime.datetime.now())

if __name__ == "__main__":
    app.run()
```

So, that sounds great, but what can we actually do with this effectively infinite supply of short-lived machines.

Let’s start with a simple flask application that tells the user the time. If you haven’t seen flask before, there’s some boilerplate, but the important thing is the time function (which returns the time!) and the line immediately before it, which is a decorator. That decorator tells flask that if someone visits the URL /time, it should respond with the result of the function.

```
$ python app.py
 * Serving Flask app "app" (lazy loading)
 * Environment: production
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

```
$ curl 127.0.0.1:5000/time
2018-08-15 21:09:46.450550
```

We can “deploy” this app locally by running python app.py. And if we visit localhost we can get the result. Apparently I created this slide at 21:09.

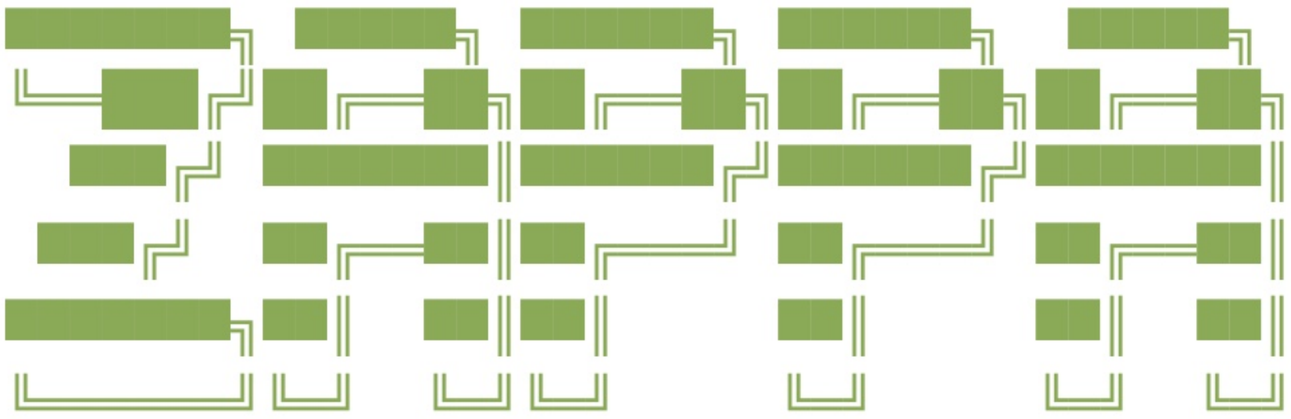
But assuming your laptop is firewalled sensibly, and you have aspirations to occasionally put it to sleep, this is not a good place for the app to live long term.

Serverless use case #1

Web operations for a low, low price

Which brings us to the first, and most prominent use case for serverless. Let’s deploy our time server on AWS Lambda.

```
$ ls
venv/     app.py

$ source venv/bin/activate && pip install flask

$ zappa init
```



```
Welcome to Zappa! Zappa is a system for running server-less Python web
applications on AWS Lambda and AWS API Gateway. This `init` command will
help you create and configure your new Zappa deployment. Let's get
started!
```

There's a lot of fiddly book-keeping involved in Lambda deployment, but the good news is you can avoid it all by using one of a number of command line tools.

I recommend a tool called zappa. I like zappa because it's written in python, and at least in principle it's agnostic about which of the serverless platforms you deploy to (although in practice it only works on AWS Lambda).

The best known alternative is called "serverless" confusingly, but the installation instructions mention npm and I'm like: not today, satan).

AWS has their own tool called Chalice, which I've never used, but it doesn't support a use case I'll mention later, and of course it doesn't and will never support any target other than AWS Lambda.

Here's how you use zappa. All this code is on the GitHub repo I mentioned at the start. First you create and activate a fresh virtualenv, and install your dependencies. In this case that's flask. Then run zappa init and you get a bunch of questions (you can accept the defaults).

```
$ cat zappa_settings.json
{
    "dev": {
        "app_function": "app.app",
        "aws_region": "us-east-1",
        "profile_name": "default",
        "project_name": "basic",
        "runtime": "python3.6",
        "s3_bucket": "zappa-ycr5dtiyk"
    }
}
```

That init command creates a configuration file that looks like this. Note it’s auto populated “app.app”. That’s the module and object corresponding to our flask application.

```
$ zappa deploy
Calling deploy for stage dev..
Downloading and installing dependencies..
 - sqlite==python36: Using precompiled lambda package
Packaging project as zip.
Uploading basic-dev-1534393616.zip (6.2MiB)..
100%|████████████████████████████████████████| 6.54M/6.54M [00:05<00:00, 902KB/s]
Scheduling..
Scheduled basic-dev-zappa-keep-warm-handler.keep_warm_callback with expression
rate(4 minutes)!
Uploading basic-dev-template-1534393629.json (1.6KiB)..
100%|████████████████████████████████████████| 1.59K/1.59K [00:00<00:00, 3.29KB/s]
Waiting for stack basic-dev to create (this can take a bit)..
100%|████████████████████████████████████████| 4/4 [00:09<00:00,  4.89s/res]
Deploying API Gateway..
Deployment complete!: https://ucyxlhvz7f.execute-api.us-east-1.amazonaws.com/dev
```

Then we do zappa deploy, and here's where the zappa shines. It's taking care of a huge amount of fiddly stuff here. Most notably, it's creating a zip file that contains both our application and its dependencies (as defined by what it finds installed in the virtualenvironment), and copying that to AWS. At the end of all the book-keeping, we get a public URL we can use and share to replace 127.0.0.1.

```
$ curl https://ucyxlhvz7f.execute-api.us-east-1.amazonaws.com/dev/time
2018-08-16 04:30:59.270893
```

And it works!

Here's the time on the server, which is 4:30am, rather than 9:30 at night, because AWS runs in UT.

In the notes for this talk I show how to upgrade this deployed application to give allow the timezone as a URL parameter. zappa makes that easy, and crucially it has a tail command that consolidates the logs and errors from the components of the deployed application in your terminal, which makes debugging easier.

Serverless use case #2

Cron jobs for a low, low price

Like I say, web operations is the perhaps the best known use of Lambda. But let’s look at another: regularly recurring jobs.

∂wx DTLA
@dwxdtla

Friday. Partly cloudy overnight. High of 109 (much warmer than yesterday).

7:30 AM - 6 Jul 2018

I created a couple of twitter bots, dwxdtla, and dwxnyc, that tweet today's weather compared to yesterday's each morning. These are actually Lambda functions that trigger on a regular schedule.

```
$ cat zappa_settings.json  # see github.com/williamsmj/dwx
{
    "dev": {
        "profile_name": "default",
        "s3_bucket": "zappa-abcdefg123",
        "events": [{
           "function": "tweet.check_time_and_post",
           "expression": "cron(30 * * * ? *)"
        }],
        "environment_variables": {
            "DS_KEY": "...",
            "TW_CONSUMERKEY": "...",
            "TW_CONSUMERKEYSECRET": "...",
            "TW_ACCESSTOKEN": "...-...",
            "TW_ACCESSTOKENSECRET": "...",
            "DWX_LATITUDE": "40.782222",
            "DWX_LONGITUDE": "-73.965278",
            "DWX_TZ": "US/Eastern"
        }
    }
}
```

I deployed these functions using zappa. Their configuration files illustrate a couple of useful zappa features in bold here. First, you can use a cron-like syntax to create recurring jobs, which map to functions. Second, you can export configuration to Lambda instances in the form of environment variables.

I was going to create a San Francisco bot for this talk, but Twitter changed the rules and I'm in a queue now to be revalidated. But if you want to create one of these bots for your location, or you're interested in cron-like serverless in general or Twitter bots in particular, check out the GitHub repo for this project.

So we can deploy web applications and run cron jobs. Great. What's the advantage? Well, there are a couple of big ones from the point of view of the operations and traditional software engineering communities that are worth mentioning: you can focus on business logic, and you get scalability for free.

But the advantage that most interests me is this one. Because we only pay for our code while it's running, and when it's running we pay in proportion to its usage, without unutilized overhead, it's often cheap! We'll dig into what that means for data science in a second, but let me give you an example from a traditional web app deployment.

Adam Pash Follow
Director of Engineering at Postlight. Formerly Gawker Media, And then I was like, Lifehacker.
Jun 21, 2017 · 7 min read

Serving 39 Million Requests for $370/Month, or: How We Reduced Our Hosting Costs by Two Orders of Magnitude

When I joined Postlight as an engineer last year, my first task was a big one: Rewrite the Readability Parser API. For those unfamiliar with the Readability Parser, it was the API that powered the popular Readability read-it-later app, along with many other services and apps across the web. The Parser API accepted a link to any article on the internet, then returned a structured representation of that article—extracting content from the chaos of the web. (What did Readability have to do with Postlight? That's a different story.)

The reasons for the rewrite were threefold:

Postlight's Readability API cost about $10,000/month on EC2. They moved it over to AWS Lambda and now it costs $370/month.

Now it's not all sweetness and light. There are disadvantages to this approach from the point of view of web operations. Some of those are shared with microservices in general (complexity, debugging is tricky), and some of those are shared with all cloud deployments (vendor lock-in). For the purposes of the rest of this talk, neither of those is a huge deal, but it's important to acknowledge them.

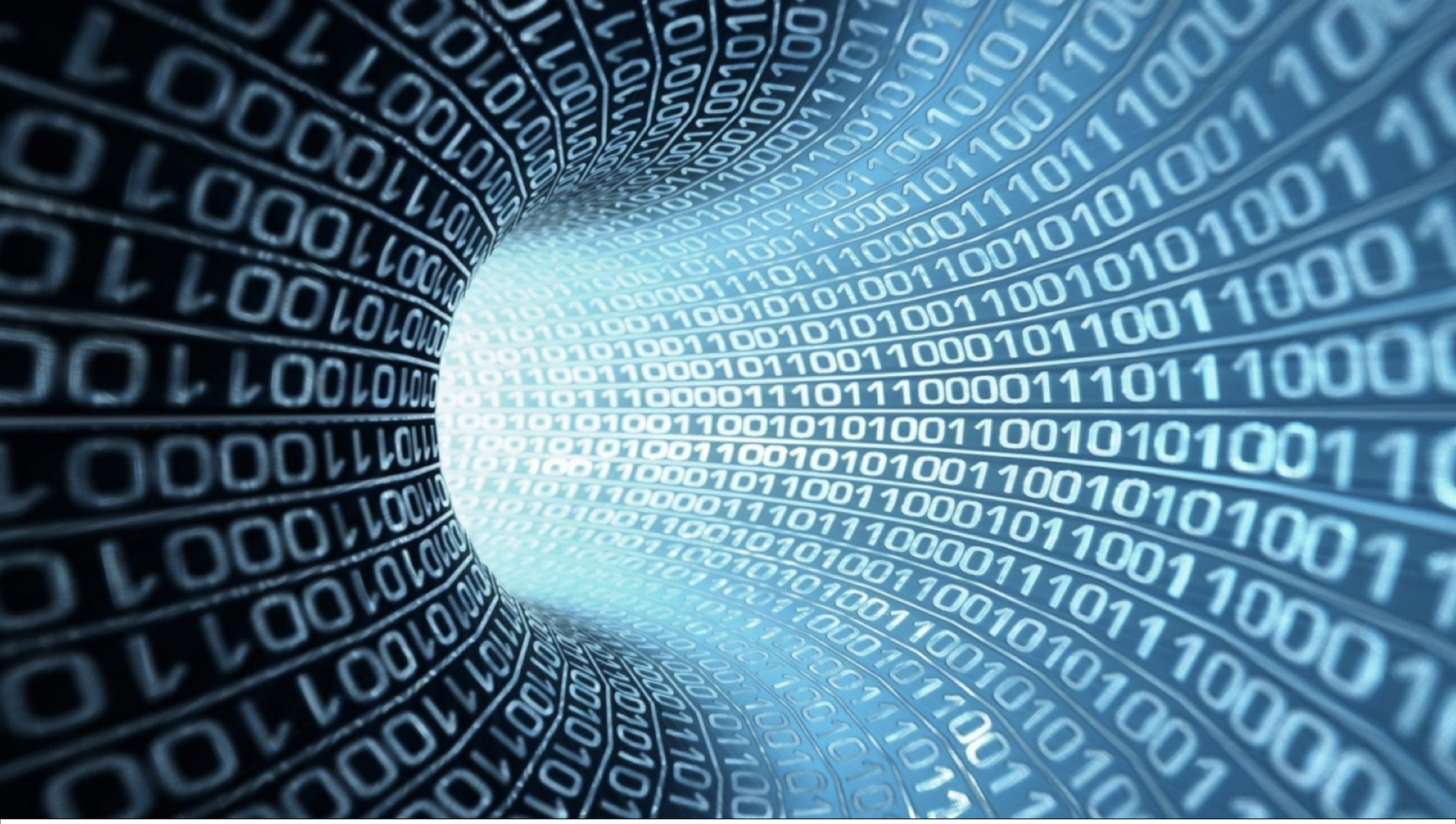
What is a huge deal, and this is not so much a disadvantage as a set of restrictions inherent in serverless, is the nature of these machines that pop in and out of existence.

The first of these restrictions is that these are underpowered machines. They have no more than 1.5GB of RAM and a tiny amount of local storage (500MB).

Second, they are living on borrowed time. Your function must complete in 300 seconds.

Those two are kind of arbitrary, and they vary by cloud provider, and if you are up for running your own serverless platform using OpenFaaS they can be lifted.

But the unavoidable restriction, that is inherent in the premise of serverless, is that these short-lived machines have no state. They have no record of previous instances. They even have no direct way of communicating with other running serverless instances. So, side effects like posting to Twitter or writing to a database aside, these serverless instances only support pure functions.



But this talk is about data science. Here's a picture of data science to indicate that we are moving into a new phase of the talk.

We've got these machines that have no internal state and are not very fast. Potentially we've got a lot of them. And we only pay for them while we're using them. This raises interesting possibilities for data scientists!

Computer Science > Distributed, Parallel, and Cluster Computing

Occupy the Cloud: Distributed Computing for the 99%

Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, Benjamin Recht

(Submitted on 13 Feb 2017 (v1), last revised 7 Jun 2017 (this version, v2))

Distributed computing remains inaccessible to a large number of users, in spite of many open source platforms and extensive commercial offerings. While distributed computation frameworks have moved beyond a simple map-reduce model, many users are still left to struggle with complex cluster management and configuration tools, even for running simple embarrassingly parallel jobs. We argue that stateless functions represent a viable platform for these users, eliminating cluster management overhead, fulfilling the promise of elasticity. Furthermore, using our prototype implementation, PyWren, we show that this model is general enough to implement a number of distributed computing models, such as BSP, efficiently. Extrapolating from recent trends in network bandwidth and the advent of disaggregated storage, we suggest that stateless functions are a natural fit for data processing in future computing environments.

The rest of this talk is about those possibilities, and a proof of concept tool called pywren that was first described in this paper. If you haven't read a CS paper before (or for a while), I highly recommend this paper! But we're going to look at pywren, which is a tool that maps functions across parameters using Lambda as the computational backend.

```
>>> def square(x):
...     return x * x

>>> parameters = [0, 1, 2, 3, 4, 5]

>>> [square(x) for x in parameters]
[0, 1, 4, 9, 16, 25]

>>> mapped = map(square, parameters)

>>> mapped
<map at 0x114254400>

>>> list(mapped)
[0, 1, 4, 9, 16, 25]
```

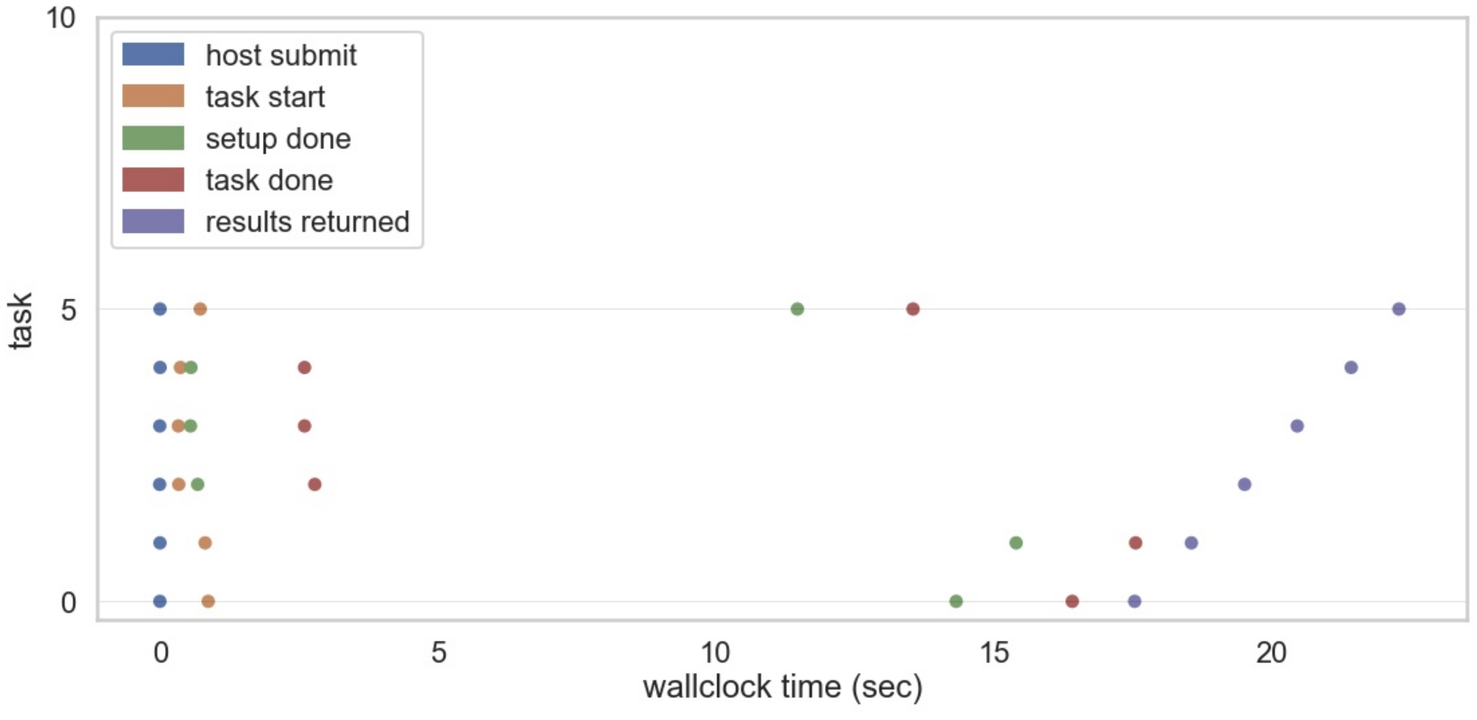
What does map "functions across parameters mean". Here's what it looks like locally. The usual approach is to use a list comprehension. You can also use map, which is considered unpythonic, but I want to introduce it here.

In python 3, map gives you back a slightly weird object. But no worries. If you call list on it, you "consume" that intermediate object and get the answer.

(You might be thinking, by the way, that you should use python 2. In which case, if you're a data scientist then I want to draw your attention to fact that a little library called numpy will no longer support python 2 for new versions starting in ... 4 months!)

```
>>> import pywren

>>> pwex = pywren.default_executor()

>>> futures = pwex.map(square, parameters)

>>> futures
[<pywren.future.ResponseFuture at 0x11422ffd0>,
 <pywren.future.ResponseFuture at 0x11422f588>,
 <pywren.future.ResponseFuture at 0x11422f358>,
 <pywren.future.ResponseFuture at 0x11422f748>,
 <pywren.future.ResponseFuture at 0x10bbfff98>,
 <pywren.future.ResponseFuture at 0x11422f470>]

>>> futures[0].result()
0

>>> [f.result() for f in futures]
[0, 1, 4, 9, 16, 25]
```

Here's how you'd apply that same computation using pywren's map function, which is almost but not quite a drop in replacement for the regular map function. Behind the scenes, it sets up 6 AWS Lambda instances and returns "futures" objects. Don't worry about what these are. They have a result method that gives us the answer, so again, we have a slightly indirect way to get the answer.



What's actually going on when we use pywren? First pywren serializes the function and the data and puts it on s3, and invokes the lambdas. That's "host submit".

The lambdas start (that's "task start"). They have to do some setup, which involves pulling and deserializing the job from s3 and installing an anaconda runtime. That's "setup done".

They compute the result and write it to s3 ("task done") where it waits for us to call the result method. When we call that method, we download the result to our client.

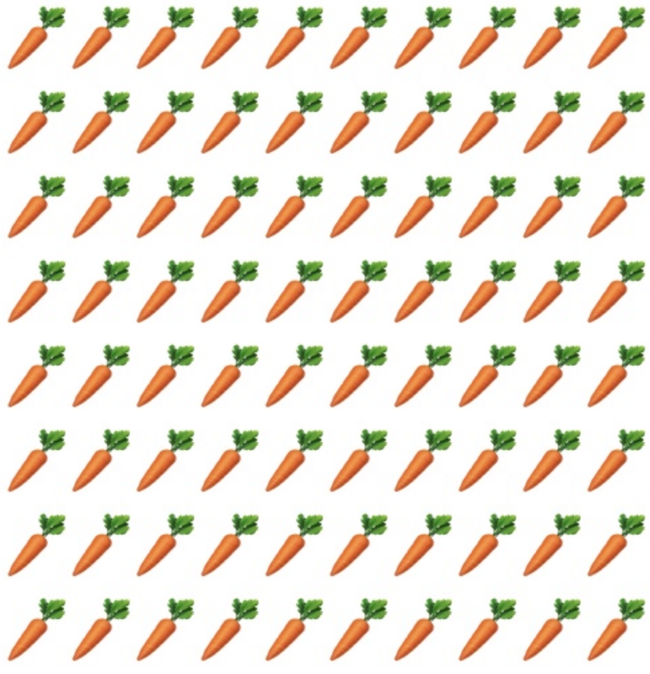
```python
if USE_PYWREN is True:
    def map(fxn, args):
        futures = pywren.default_executor.map(fxn, args)
        return pywren.get_all_results(futures)
```

Notice that, those funny map and future objects aside, we don’t need to change much code. In fact, if you wanted to globally use pywren for mapped computation, you could take advantage of the dynamic nature of python and just redefine the map builtin to be one that used pywren.

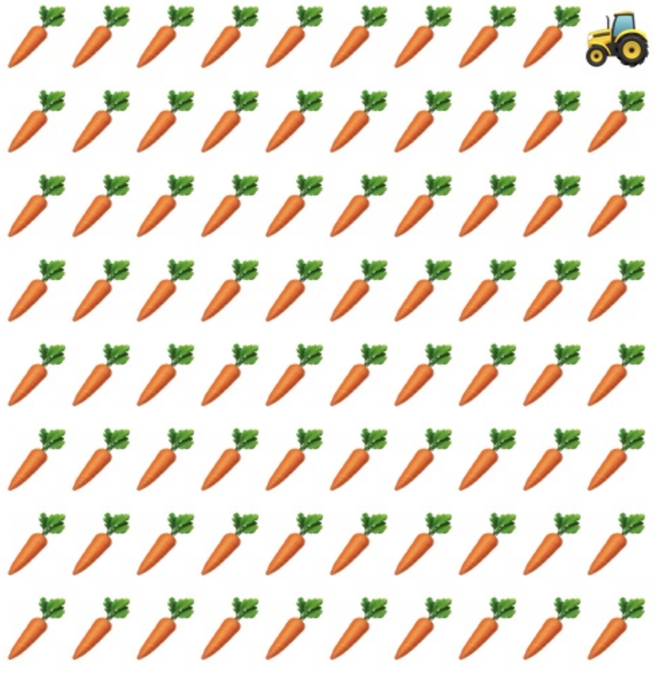
Serverless use case #3

Embarrassingly parallel jobs

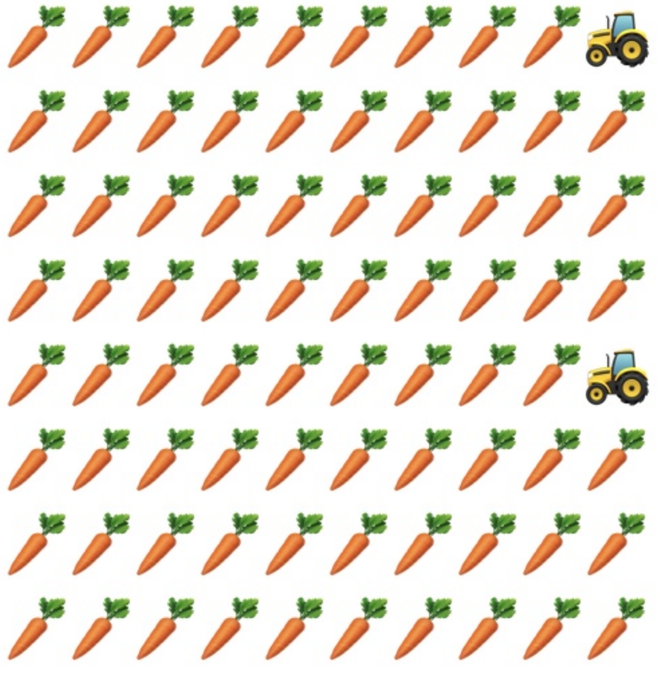
What we just saw was an example of our third use case. What is it about this job that made it a candidate for pywren? It was embarrassingly parallel. What does that mean?



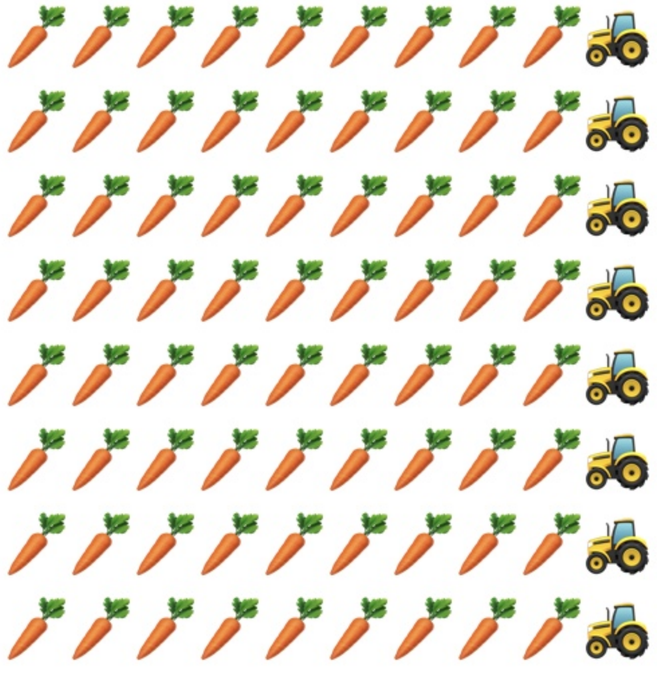
Imagine a field of carrots. Eight rows, each containing ten carrots.



I have a harvester that can pick one carrot per second. It’s obviously going to take 80 seconds to harvest the carrots.



What if I have two harvesters. Then the harvesters can work independently and harvest the carrots in half the time.



And if I have eight harvests I can get it done in 10 seconds. I get a speedup in direct proportion to the number of workers because the tractors can work independently. This is not like co-authoring a book, where the two authors need to exchange information regularly about what they've done so far. The tractors are truly independent. There is no inter-worker communication. That's embarrassingly parallel.

Anything in python that you'd normally implement as a list comprehension usually has this characteristic. Some for loops have side effects and are not embarrassingly parallel, but if you can convert a for loop to a comprehension, it's likely embarrassingly parallel too.

```
# original idea from
#   https://blog.seanssmith.com/posts/pywren-web-scraping.html
#
# this example in full (and the NIPS dataset)
#   github.com/williamsmj/nips.json

>>> def scrape_batch(batch):
...     for paper in batch:
...         paper["abstract"] = scrape_abstract_for_one_paper(paper)
...     return batch

>>> batches = [[paper1, paper2, ...], [paper101, paper201, ...], ...]

>>> pywren.get_all_results(pwex.map(scrape_batch, batches))
```

So here's another example from my own work. I wanted scrape the abstracts for the many thousands of machine learning papers from a conference website. Doing this on my laptop took a long time because I was scraping them one at a time (or, if I wanted to get clever, I could have written some asynchronous code, or used multiprocessing). But even then I would still have been constrained by the speed of my internet connection.

By shifting this map to Lambda with pywren, I could work in parallel and also benefit from AWS's faster internet connection!

Note here I'm not mapping a function that scrapes one paper over a list of papers. I'm mapping a function that scrapes a batch of papers over a list of batches. I'm doing that because scraping one paper is relatively quick, which means that if I map them one at a time then the overhead of setting up Lambda instances dominates. By accumulating lots of pages until that 10 second overhead is worth it, I maximize my speed up.

This is pseudocode. The full code (and the resulting dataset) is on GitHub if you're interested.

http://www.bradfordlynch.com/blog/2017/05/28/ComputeOnLambda.html

305 Million Solutions to The Black-Scholes Equation in 16 Minutes with AWS Lambda

Originally Posted: May 28, 2017

The research I'm working on involves estimating a firm's probability of default over a variety of time horizons using the Merton Distance to Default model. The dataset contains daily financial information for more than 24,000 firms over the past 30 years. Given that I am calculating the probability of default over five time horizons, applying the Merton model will require solving the Black-Scholes equation roughly 305 million times. Luckily, the model is easily parallelized because the only data needed for the model, aside from the risk-free rate, is firm specific. This post shows how the Python library Pywren can leverage AWS Lambda to run hundreds of models in parallel, achieving a **270x speed-up** over a quad-core i7-4770, with minimal changes to the simulation code. If you are interested in learning more about the model, see my post about implementing the model in Python.

Here's another example, of parallelizing some pretty beefy numerical calculations. The Black-Scholes equation is a partial differential equation that describes the evolution of a stock option. Bradford Lynch wanted to solve it for many millions of configurations. Again, with only a couple of changes to his code, he was able to get a 270x speedup.

On the GitHub link at the start of this talk there are also examples of long-lived embarrassingly parallel ETL loads such as video encoding. Those are cool, but I'd call them data engineering rather than data science.

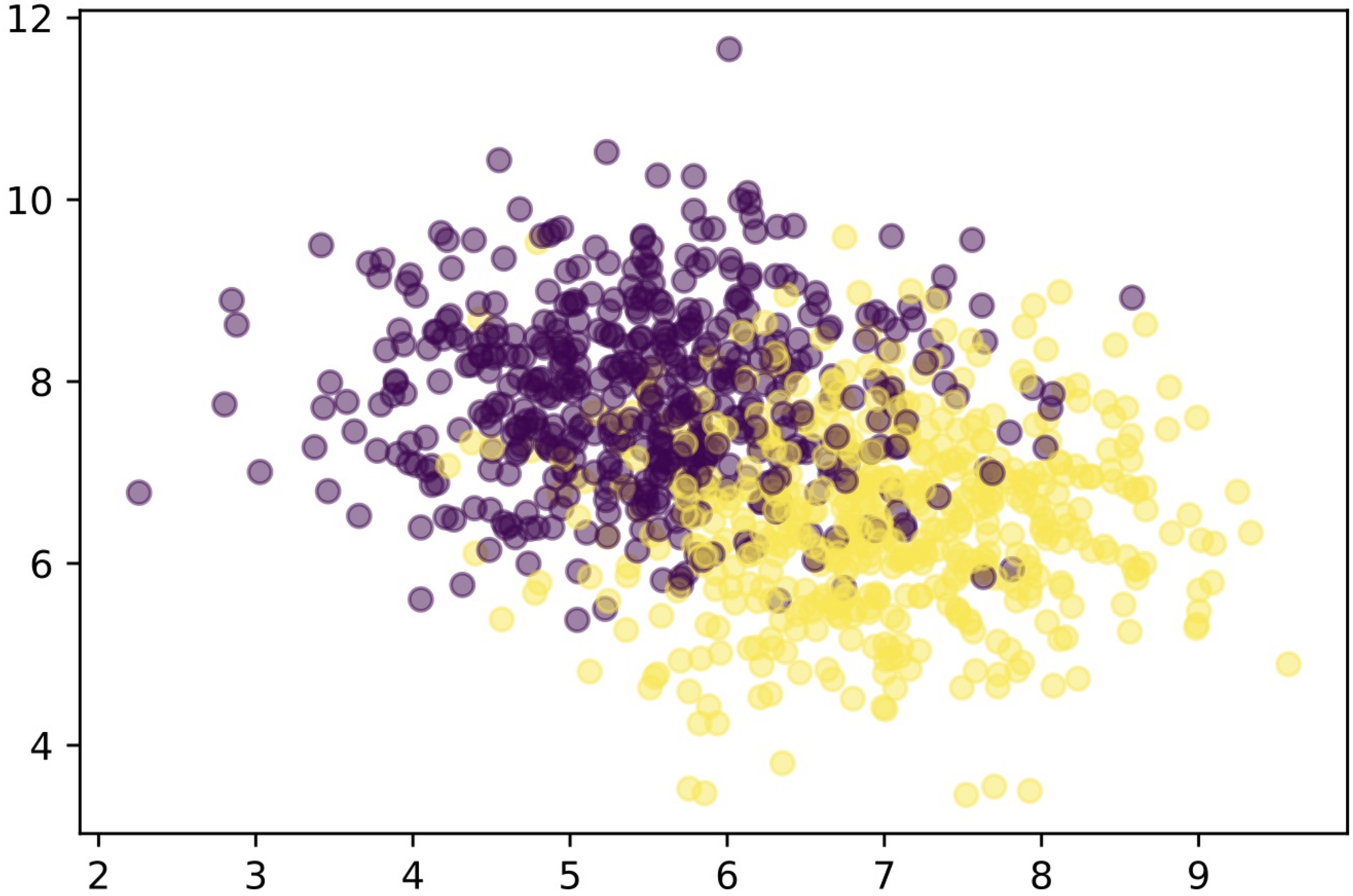
I find using Lambda interactively with pywren, for things like web scraping, or notebook-like exploration, more compelling. The reason is, it's in that interactive mode that Lambda's cost savings are most apparent. Think about data exploration in a notebook: the vast majority of that time is spent thinking, not waiting for the computer. And if that expensive computer is always on (like a spark cluster), that's a waste of money. A cluster that can be spun up and destroyed quickly (which is effectively what pywren offers), is super compelling to me as a data scientist!

Serverless use case #1 and #3

Hyperparameter optimization (pywren) and model serving (zappa)

Finally I want to talk about machine learning! In general, machine learning algorithms are a little tricky to parallelize. Distributed gradient descent for example is a famously subtle algorithmic problem, even before you get to the question of the system you're distributing over.

But one part of the ML workflow that's a good fit for pywren out of the box is hyperparameter optimization. And I want to close the loop by looking again at web operations with zappa, and showing how we data scientists can throw off the shackles of our engineering overlords and put our own models into production.



So! We have some training data. Two classes. Two features.

```
>>> from sklearn.neighbors import KNeighborsClassifier

>>> classifier = KNeighborsClassifier(n_neighbors=5)

>>> classifier.fit(Xtrain, ytrain)

>>> classifier.score(Xtest, ytest)
0.838
```

We can train a model for this data locally like this. I'm using K-nearest neighbors, which gives me 83.8% accuracy. KNN classifies a point by looking at the labels of the K points closest to it in the training set and taking a vote. Very simple.

That number K is a "hyperparameter". A hyperparamater is a magic number required by a ML algorithm that you have to choose on a problem by problem basis. In this case, we need to choose how many nearby training samples are going to get to vote on the classification of our new point.

The usual way to do this is a simple search: you try all the possibilities, and you pick the one that gives the most accurate model.

```
>>> all_hyperparams
[{’n_neighbors': 1},
 {'n_neighbors': 2},
 {'n_neighbors': 3},
 {'n_neighbors': 4},
 {'n_neighbors': 5},
 {'n_neighbors': 6},
 {'n_neighbors': 7},
 {'n_neighbors': 8}]

>>> def train_model(hyperparams):
...     classifier = KNeighborsClassifier(**hyperparams)
...     classifier.fit(Xtrain, ytrain)
...     return classifier
```

That approach is embarrassingly parallel, so we can use pywren. First we have to set up a list of all the values of K we want to try. Then we have to write a function we can map over this list of hyperparameters that returns a trained model.

```
>>> futures = pwex.map(train_model, all_hyperparams)

>>> classifiers = pywren.get_all_results(futures)

>>> for hyperparams, classifier in zip(all_hyperparams, classifiers):
...     print(hyperparams, classifier.score(Xtest, ytest))

{'n_neighbors': 1} 0.758
{'n_neighbors': 2} 0.782
{'n_neighbors': 3} 0.814
{'n_neighbors': 4} 0.82
{'n_neighbors': 5} 0.838
{'n_neighbors': 6} 0.838
{'n_neighbors': 7} 0.842
{'n_neighbors': 8} 0.828
```

Now we can send this computation off to AWS. Note here that pywren doesn't only send back simple results like numbers. It can send back a rich python object with attributes and methods, in this case trained scikit-learn classifiers. And we can look at the accuracy of each model and choose the best. I've taken the liberty of uploading a pickle of that best model to a public URL, which allows me to do my final trick!

```python
# modelserver/app.py
url = "https://s3.amazonaws.com/modelservingdemo/classifier.pkl"
r = requests.get(url)
classifier = pickle.loads(r.content)

@app.route("/predict")
def predict():
    X = [[float(request.args['feature_1']),
          float(request.args['feature_2'])]]
    label = classifier.predict(X)
    if label == 0:
        return "purple blob"
    else:
        return "yellow blob"
```

This is a flask application that downloads a scikit-learn model from a public URL when it's launched. It has a predict endpoint that expects two URL parameters and returns a prediction. I deployed it to Lambda using zappa before this talk.

https://cpj4teujk3.execute-api.us-east-1.amazonaws.com/dev/predict?feature_1=9&feature_2=6

What it's good for

Web ops
Cron jobs
Embarrassingly parallel computation
...i.e. bursty stuff

So, that was kind of a grab bag. Let me give you some high level take-aways.

We've seen serverless is good for deploying websites and running cron jobs. And we saw at length that it's good for a certain kind of computation, namely embarrassingly parallel stuff. It's good for these things in the sense that it's relatively easy to set up and cheap to use.

The principle to take away from this is what these things have in common — they're bursty. That's the reason they're cheap on Lambda. They're bursty in the sense that they sometimes require significant resources that Lambda can scale up to accommodation. But also in the sense that they sometimes require zero or almost zero resources. Your web product is new, your cron job isn't on, or you're working interactively, and you're now staring at your notebook trying to figure out what these results mean. And on serverless you're not paying for that thinking time.

The great thing about serverless is not so much that it's great when you're using it (although the ability to scale up very large is powerful), but that it's great when you're not using it!

What it's not good for

Map-shuffle-reduce
Linear algebra (yet)
Atomic tasks that last longer than 300s
Training neural networks

Now, I'm not an operations person, so I'm not going to speak to its disadvantages for web applications. But I want to flag up some limitations for data science use.

First, pywren really only works for embarrassingly parallel stuff, i.e. when the workers don't need to communicate with each other. Some map-reduce tasks look like this, but many have a shuffle step, where the results are aggregated and manipulated and potentially distributed back out for further work. The pywren team have demonstrated workarounds that make pywren a possible alternative to hadoop and spark that involve writing intermediate results to s3. But at that point pywren is no longer the drop-in replacement for the python map function. You need to think deeply about the structure of your algorithm in order to use it.

By the same token, many linear algebra operations can be parallelized for huge speedups, but the tricks involved mean that the resulting algorithms are often not embarrassingly parallel. That means that, out of the box, vanilla pywren can't help. With that in mind, keep an eye on the pywren project. They're working on a project that gives you arrays and array operations that feel like numpy, but that use Lambda as the computational backend.

We saw with the web scraping example that you sometimes need to accumulate many tasks to make the overhead worth it. But if your atomic tasks are already over that 300 second limit, Lambda can't help.

github.com/williamsmj/serverless-for-data-scientists

Mike Lee Williams • @mikepqr • mlw@cloudera.com

Remember the pywren paper was called “Distributed computing for the 99%”.

What I understand by that is that the data science 1% operate in companies that are at a scale where they very rarely don’t require most of their computing resources. Global consumption at these companies varies, but it doesn’t go down to zero. Or if it does, their peak usage is such that’s it’s cheaper to just buy the dang cluster. And the data scientists have data engineering, tools and operations support to take care of those resources. Serverless in the sense I’ve presented it today, and in the sense I think the pywren folks have in mind, is perhaps not for those people.

But, I guess the high level take away from this talk, is that if you’re operating independently, and you don’t have the money, the engineering support, or the patience for heavyweight solutions, take a look at zappa, pywren and the serverless ecosystem. Even if you do have those resources, check it out. It’s a lot of fun!