Raster processing on Serverless Architecture

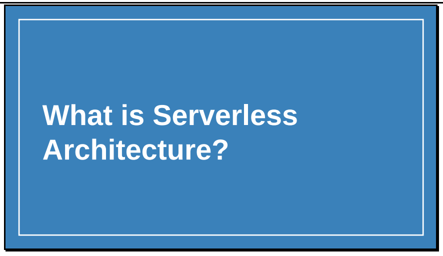
Using python tools on AWS Lambda

Good afternoon, thanks for coming to my talk on Raster Processing on Serverless Architecture. I'm going to give an introduction to what Serverless Architecture means, what the implementation looks like on The Amazon Web Service platform and show that despite significant restrictions it could be a performant and cost effective way to expose raster processing services. I've got a lot of material to work through so I'm going to try and talk really fast, but I'll be around for the rest of the conference and would be happy to get into more details with anybody.

Hello.

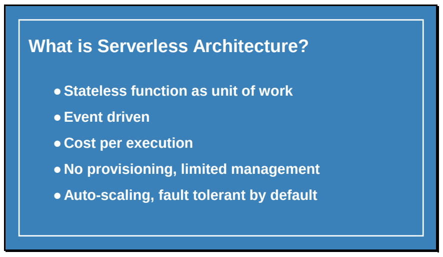
azavea

My name is Matt McFarland and I'm a software developer in Philadelphia. I work at a company called Azavea which does geospatial web application development and data analytics - and the inspiration for this talk came out of the work we do there.



What is Serverless Architecture?

So what is Serverless Architecture? The very short answer is: it's a cloud computing platform. It executes your code and it's distinguished by having managed infrastructure that abstracts away a server in the traditional sense. There are, of course, still servers executing your code: very sophisticated servers running sophisticated software - so it's helpful to be aware of *how* the infrastructure operates, but you don't have to manage it or, in most cases, stay up at night worrying about it.



What is Serverless Architecture?

- Stateless function as unit of work
- Event driven
- Cost per execution
- No provisioning, limited management
- Auto-scaling, fault tolerant by default

It’s an execution model which removes the need to pre-determine the compute capacity necessary to execute a task at scale. The platform manages the underlying resources needed.

A function is the unit of work in this Architecture. There’s no expectation that there will be a SINGLE function in your code, just that that there will be a single function that is invoked as an entrypoint and that it’s stateless. There’s no capacity for long running processes, so you’re not running a web server or listening to socket connection. The platform will invoke your function when it’s time to execute, and when that function exits, your code is no longer running and any state loaded is lost.

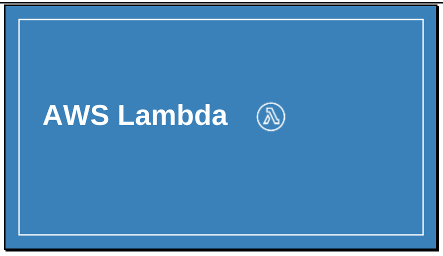
The function is invoked by means of an event being triggered, meaning that you can register a function to be executed when something happens in your system.

The cost of a function is only for the actual time executing it. There is no pre-purchased capacity, like in a traditional VM/EC2 instance. This can be very cost effective if you expect your function to have periods of being under-utilized or idle. If nothing invokes your function for the weekend, you don’t pay anything for having the potential capacity.

The big deal is that you don’t create, provision or manage the infrastructure on which the function is executed. It’s not an exact analogy, but it may be helpful to visualize your function existing as a container image, like a Docker Container. In that analogy, you’re providing the contents of the container and the platform is handling the infrastructure and scheduling of it in real time.

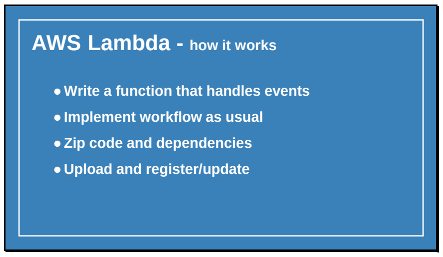
That resource management also involves scaling your function in response to multiple events. As events stream in, the platform will create instances of your function to respond to them - the function itself is not aware of the load the service is under, each is simply responding to a single event. If 10 simultaneous requests arrive, 10 instances are spun up. In practice, an instance usually sticks around for a few minutes, so it can be reused - but that’s an optimization the platform is making, you should always consider you function ephemeral.

With a serverless architecture, you’re no longer engineering the load balancing, scaling policies or failover rules that are otherwise necessary for a resilient application.



AWS Lambda

AWS Lambda is the incumbent Serverless provider, though similar functionality can be found on other platforms.



AWS Lambda - how it works

- Write a function that handles events
- Implement workflow as usual
- Zip code and dependencies
- Upload and register/update

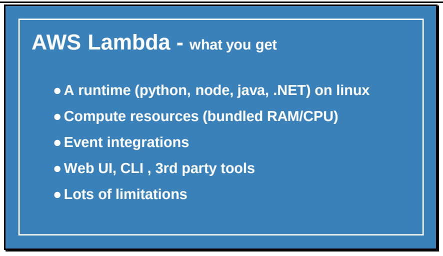
It operates just like the overview I gave. At a high level, in the context of python, it works like this:

1st: You write your handler function: this receives the event and has to conform to a signature dictated by the platform.

Next, implement your logic as you would normally, just remember it needs to be stateless so lean on external services for storage and things like that. Write it so you can run and test locally.

Choose your dependencies judiciously, because you have to zip them up with your code and create a deployment bundle - the size of which has implications for the performance of the function.

And the last step is to create the Lambda resource via CloudFormation, along with any additional AWS resources (like API Gateway endpoints, s3 buckets). Upload the whole lot and wait for the stack to be created.

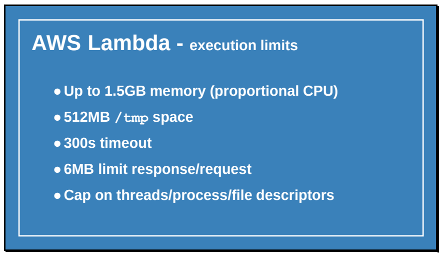


AWS Lambda - what you get

- A runtime (python, node, java, .NET) on linux
- Compute resources (bundled RAM/CPU)
- Event integrations
- Web UI, CLI , 3rd party tools
- Lots of limitations

Here are some specifics on what you can configure for the service:

- You can chose the runtime your function operates on: python, node, java and .NET - and you do have access to a portion of the linux filesystem.
- You also chose the coarse resources available to your function: the amount of memory from 128MB to 1.5GB. The CPU increases proportionally with memory, but is not set explicitly.
- There are a catalog of events from within the AWS ecosystem that you can register your Lambda function with, as well as invoke it programmatically yourself. Some examples are when an object is added to an S3 bucket, a record inserted into a DynamoDB table, or an HTTP request to the API Gateway.

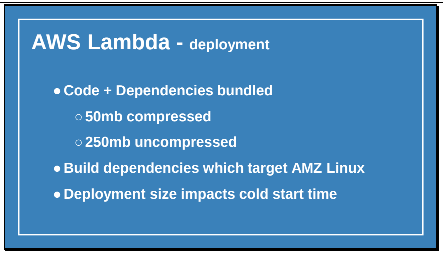


AWS Lambda - execution limits

- Up to 1.5GB memory (proportional CPU)
- 512MB `/tmp` space
- 300s timeout
- 6MB limit response/request
- Cap on threads/process/file descriptors

So you trade control and responsibility over a lot of DevOps work to get a scalable, fault-tolerant compute environment without a lot of engineering effort - but you have to live within the rules.

No long running jobs, no memory hungry jobs, no jobs that directly return large payloads and nothing that writes anything substantial to disk. It's a challenging environment to do computation, especially with rasters - but these are all limitation that can be worked around.

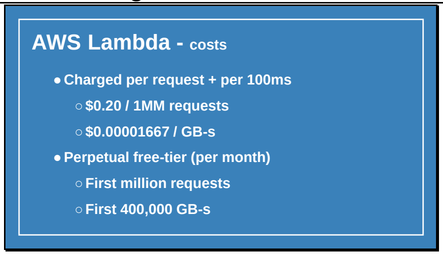


AWS Lambda - deployment

- Code + Dependencies bundled
  - 50mb compressed
  - 250mb uncompressed
- Build dependencies which target AMZ Linux
- Deployment size impacts cold start time

As I mentioned before, when you deploy a lambda function you are responsible for delivering the code and all of its dependencies outside of the standard library. This means you are not apt-get, yum or pip installing anything as part of a provisioning step. If your function depends on anything it must be in the deployment package and compiled to run on AMZ Linux.

That deployment bundle can only be 50MB compressed, and the overall size of the bundle affects what is called the "cold start" time, which is the time it takes to spin up an instance of a function to respond to an event. The larger the bundle, the more time it takes for Lambda to create it. You don't incur this latency when there is a recent instance already available, so counterintuitively a higher use app will have less latency than one with lower usage.



AWS Lambda - costs

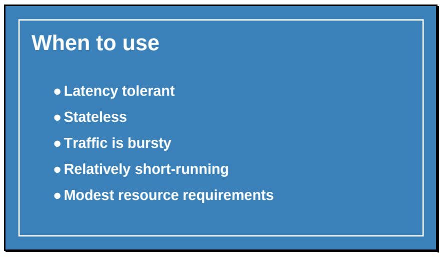
- Charged per request + per 100ms
  - $0.20 / 1MM requests
  - $0.00001667 / GB-s
- Perpetual free-tier (per month)
  - First million requests
  - First 400,000 GB-s

A few words on costs. You are charged for each request as well as for each 100 millisecond block of time that it executes, and the price increases with higher memory configurations.

When considering the free tier - a low use function could conceivably be run at no cost. However, for a very high use app, the costs are essentially unbounded.

An example straight from the pricing page:
"If you allocated 512MB of memory to your function, executed it 3 million times in one month, and it ran for 1 second"
The price is about $20

Keep in mind, If you are relying on other AWS services, you're responsible for those costs as well. AWS pricing can be tricky to navigate, so make sure you do your homework.

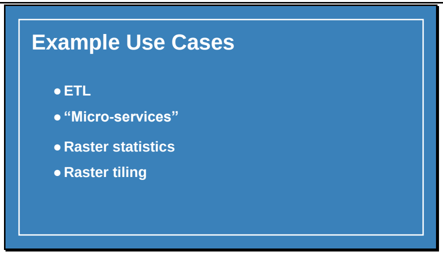

When to use

- Latency tolerant
- Stateless
- Traffic is bursty
- Relatively short-running
- Modest resource requirements

So how do you know if it's an appropriate service for your application?

Latency tolerant: Expect some variability in start up time due to cold starts. With the code I'm working with, I can see up to 4 seconds cold start + the time it takes for the function to actually execute.
(Bursty): If your traffic is constant or predictable, you may be able to engineer solutions that are more cost effective or robust. But when there's times of low usage that may spike suddenly, per-execution billing is helpful.
Short Running: 5 mins is excessive for web requests, but not a lot for some heavy computational loads, especially given the low resources available to Lambda

If you don't have access to a lot of operations or infrastructure experience, this could be an easy way to provide scalability and fault tolerance.


Example Use Cases

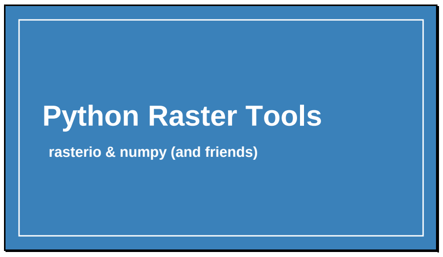
- ETL
- "Micro-services"
- Raster statistics
- Raster tiling

Common use cases are doing ETL data processing pipelines, like imports and exports based on a request or an upload. Or to implement micro-services: these small, modular network services that are loosely coupled and can scale independently of other services in a system.

Two additional use cases I'll focus on are working with raster data sets. In the projects I work on at Azavea, it's typical to need to perform on the fly zonal statistics for a user supplied geometry. Something like reporting on the distribution of land cover values within a watershed. Or creating a priority raster based on a weighted overlay operation between two source rasters.

To support that kind of analysis with a visualization, we'll often generate raster map tiles.

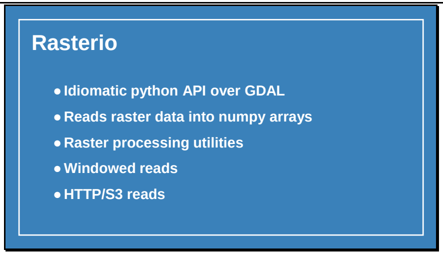
This usually involves a robustly engineered solution to provide processing outside of our web servers, that can scale to meet demand, run in multiple availability zones and yet not come at too burdensome an operating cost. A tall order, but one that actually fits the use case of AWS Lambda well.



Python Raster Tools

rasterio & numpy (and friends)

The critical piece that enables the actual raster processing are a suite of open source Python libraries: rasterio, developed by the folks at MapBox, and numpy. One could do a really great talk on just these libraries, but in the interest of time, I'm going to have to do a brief and sadly inadequate overview so I can talk about how to get them running in AWS Lambda.



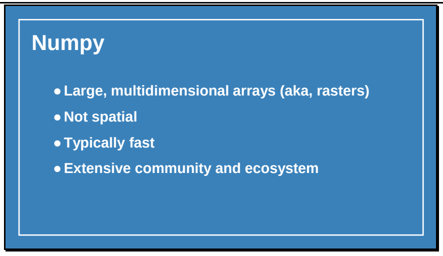
Rasterio

- Idiomatic python API over GDAL
- Reads raster data into numpy arrays
- Raster processing utilities
- Windowed reads
- HTTP/S3 reads

Rasterio is really the foundation of this work. It's a nice, pythonic API over GDAL to do I/O on rasters + additional raster processing routines

It reads raster data in as numpy arrays to allowing further processing and helps mask the arrays so you don't process NODATA values or values not contained in a zonal geometry.

What makes it especially suited to run on Lambda is the ability to do windowed reads, meaning it can start and stop reading bytes at defined offsets within a file and it can read these windows over a network protocol like S3.

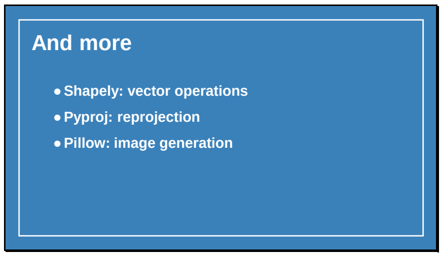


Numpy

- Large, multidimensional arrays (aka, rasters)
- Not spatial
- Typically fast
- Extensive community and ecosystem

Numpy is a python library for working efficiently with large, multidimensional arrays. A raster is really just a 2d array of cell values where the cells are essentially georeferenced.

It's not a GIS library, but with the right techniques you can perform traditional raster operations using the numerical operators in python.



And more

- Shapely: vector operations
- Pyproj: reprojection
- Pillow: image generation

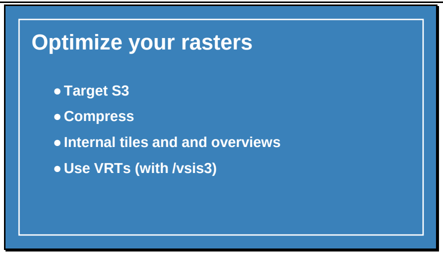
There's actually a number of additional libraries to support raster processing on Lambda, a few notable mentions include

- Shapely, for vector manipulation
- Pypoj, for vector reprojection for the same
- And pillow, which is a python image library - we'll use that to make png tiles on the fly from our rasters

---

How it's done

So now I'm going to try to cram all that information together and show how to set up a Lambda function to do zonal raster calculations and tiling.



Optimize your rasters

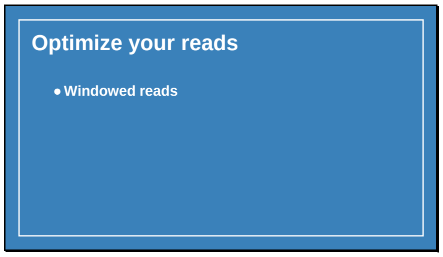
- Target S3
- Compress
- Internal tiles and and overviews
- Use VRTs (with /vsis3)

The first task is to optimize your raster for this environment.

You'll want to target s3 for storage. The rasters is not going in your deployment bundle and we're only going to want part of them at a time. Since you pay for storage monthly and you want to give rasterio the fastest reads possible with the fewest bytes on the wire, you have an incentive to use aggressive compression options. We've been able to reduce a national 10m DEM dataset from around 7-800GB from the original .IMG format to 250GB of GeoTiffs while improving read speeds over the network.

GeoTiffs also support an internal tiling scheme that rasterio will take advantage of when doing windowed reads which has a huge impact on read time.

It also plays nicely with Virtual Datasets (VRTs), so if you're dealing with a very large dataset, you can chunk it up, and create a VRT which points to the individual files on S3.   Under heavy loads, this can improve read requests because S3 will automatically partition your buckets over its internal storage structure.

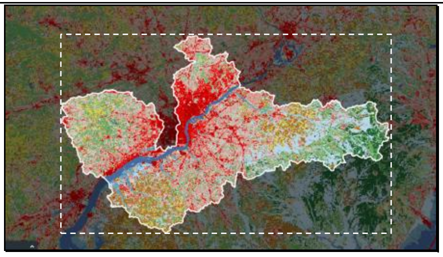


Optimize your reads

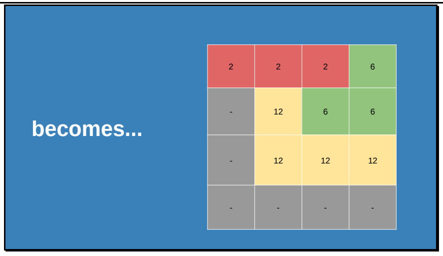
- Windowed reads

Now that your rasters are in top shape, the next thing to focus on is optimizing the techniques you use to read it in.

Doing windowed reads over HTTP is what really enable this architecture to work. A quick overview, a windowed read is essentially defining a rectangle over a larger raster and reading in just the bytes that coincide with that rectangle. On Lambda we use this technique to read a polygonal area from large raster stored on s3 so we're not loading data we don't need or won't fit in our environment.



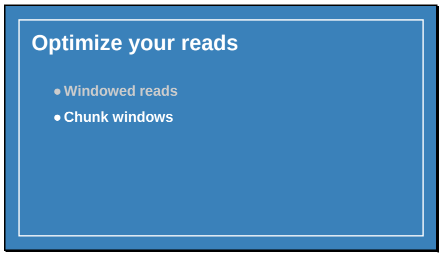
Here's what that looks like visually. I've got a national scale land use raster, and defined a region from a polygon for which I want to generate some statistics. We define a bounding box over that geometry and that rectangle identifies the strip of bytes that we're going to read out of the raster.



becomes...

2	2	2	6
-	12	6	6
-	12	12	12
-	-	-	-

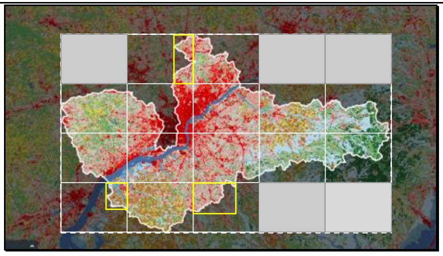
Since we're using rasterio, It's read in as a 2D numpy array. The window read was rectangular, but we masked the elements which did not overlap with the actual geometry shape, so that those values are not included when we do certain numpy operations.



Optimize your reads

- Windowed reads
- Chunk windows

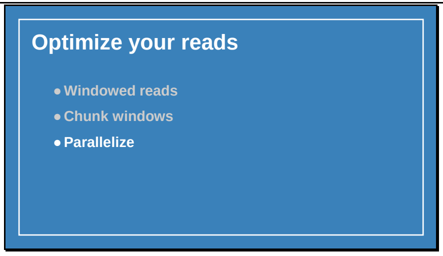
So what if the window size would pull in data that exceeds the memory available to your function? Or, even if it doesn't exceed the memory, doing something like generating statistics is a very parallelizable task. The technique I use to accommodate those scenarios is to define a polygon area size threshold, and for any input that exceeds that threshold I chunk the initial polygon into smaller geometries which you can then read in one at a time while accumulating the statistics.



To demonstrate this from our previous example, we can create a grid over our original bounding box, creating a bunch of new polygons.

- We then test each of these new polygons to see if they intersect at all with the actual geometry, and discard them if they don't
- We also reduce the size of the new bounds by defining a new window based on the intersection of the grid and the original geometry (I highlighted just a few here in yellow)

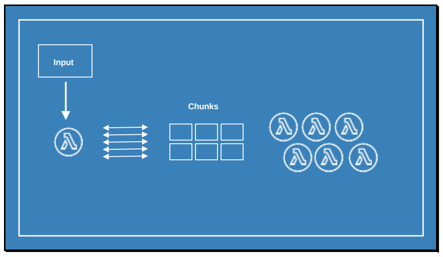
Now instead of one large window read, we have a several small ones and potentially significantly fewer meaningful bytes to read in. If the operation is fast enough, you can just execute the read within a python generator so they yield out sequentially and you never have more than one chunk of data in memory at a time.



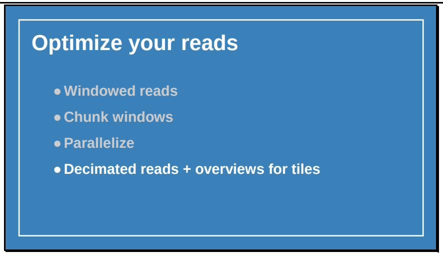
Optimize your reads

- Windowed reads
- Chunk windows
- Parallelize

Or, you can try to parallelize the task. You *can* do multiprocessing in Lambda, but it's a bit of of anti-pattern.



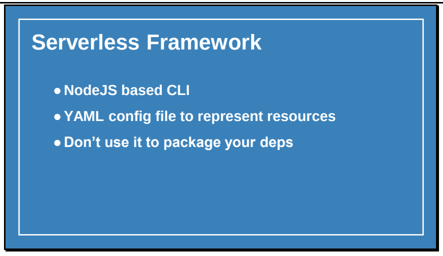
Instead, you treat the function itself as the unit of parallelization. Split up the work as described and asynchronously invoke your function passing each a single window. Lambda will spin up as many instances as are invoked so in practice you can have massively parallelized and distributed computation over your rasters, without managing any kind of costly cluster. There is, of course, some overhead when invoking functions, so you'll have to find that inflection point where it's worthwhile for your data.



Optimize your reads

- Windowed reads
- Chunk windows
- Parallelize
- Decimated reads + overviews for tiles

The last trick for optimizing your reads when you're generating visual map tiles is to do what rasterio refers to as a decimated read. Since if you have a national scale raster and say you want to create a tile at zoom level 0, the whole country is going to fit in that tile and that would be a big, expensive read and you don't need most of that data anyway. Instead, by restricting the size of the array you're reading into (probably 256x256 for a standard map tile), it does nearest neighbor resampling *as* it reads so you end up with a light tile read that's fast at any scale, and also makes use of overviews if they've been generated.

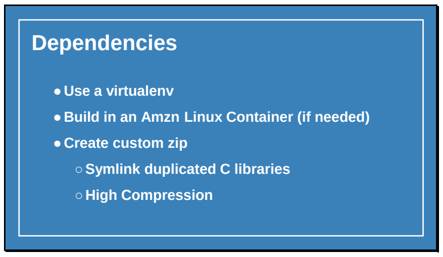


Serverless Framework

- NodeJS based CLI
- YAML config file to represent resources
- Don't use it to package your deps

Managing the AWS resources with CloudFormation is difficult and I like to use a tool called Serverless Framework instead. It's a command line tool which uses a concise YAML configuration file to represent everything you'd put in your CloudFormation template. It creates and updates your stacks, so you can use it to define and deploy your functions.

It also really nicely packages up all of your dependencies and code into a bundle, but don't use that feature because we can optimize that as well.. In fact, if you do use that feature and have loaded all the libraries that I've mentioned in this talk, it'll fail because the package size exceeds the bundle size limits.

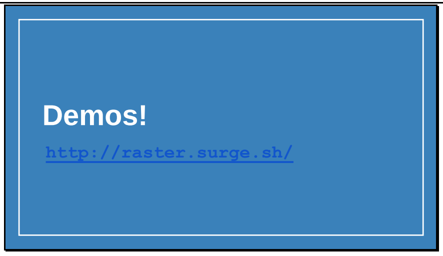


Dependencies

- Use a virtualenv
- Build in an Amzn Linux Container (if needed)
- Create custom zip
  - Symlink duplicated C libraries
  - High Compression

So how do you manage dependencies? First, pip install everything you need into a virtualenv. You don't want any unintended modules to be included. If your libraries require building C libraries, make sure you execute that script in an Amazon Linux docker container so that it's built against the right distro. For the packages I've mentioned: rasterio, shapely and numpy which do require C libraries, the package authors have conveniently built "manylinux" distributions, which package up a binary which is suitable for running in Lambda - so you don't have to do anything special.

You can exclude certain packages that are pre-installed on the python lambda runtime, like boto3 and its dependencies. Shapely and rasterio also have some duplicated C libraries between them. As part of your script, you can actually replace one version with a symlink to the other and by using the appropriate flag (-y) to the zip command, the package will maintain that symlink in the zip and also once extracted on Lambda. This can take MB off of your bundle size. I also use high compression options here to keep the zip as small as possible.

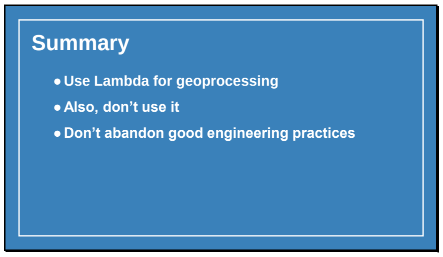


Demos!

http://raster.surge.sh/

Now, some demos. It would be a hypocritical of me to talk up the auto-scaling features of Lambda and then not share my hastily prepared demos to the audience. So feel free to poke around while I'm going through these and see if I come to regret that decision.

At worst, my requests are going to get queued up and things may slow down, but there are no servers I'm managing that I'm worried about crashing.



Summary

- Use Lambda for geoprocessing
- Also, don’t use it
- Don’t abandon good engineering practices

So I hoped I demonstrated that you can use lambda for raster geoprocessing - it’s a really interesting platform that I think can cheaply provide a lot of robustness for your applications, and not just with python. Azavea maintains a Scala raster processing library called GeoTrellis which we’ve also gotten to work in the environment.

However, it’s not a silver bullet and sometimes it’s more cost effective for you use dedicated compute resources and do the engineering to scale it. It’s not always going to be the most performant solution either, so it depends on if the tradeoffs are worth it for your use case.

And lastly, just because it’s a managed service doesn’t mean you should abandon proper engineering practices. Put a CloudFront distribution in front of you s3 buckets, cache results so they can be reused. A serverless function can complement your system but doesn’t replace the need for a proper system architecture.