**AWS Essentials for Data Science: Compute**

**Understanding and deploying EC2 and Lambda services**

So you’ve built a cool app and want to show it off to the world. Maybe it’s an AI that generates [cat pictures from scribbles](https://affinelayer.com/pixsrv/), a [viral LinkedIn post generator](https://viralpostgenerator.com), or an [English to RegEx translator](https://www.autoregex.xyz/). You want a user to just click a link and immediately start interacting with your app, rather than needing to…

This “immediate interactivity” is going to require a **server**, which takes user requests (e.g., cat scribbles) and *serves* responses (e.g., AI-generated cat images). You *could* use your personal laptop, but it’ll stop serving requests when it goes to sleep or turns off, and a sophisticated hacker could probably steal your private data. The cherry on top is that your hard drive might melt if your computer tries serving too many requests at once!

Unless you like hacked, melted laptops, you’ll probably want to [rent a server from the cloud](https://towardsdatascience.com/aws-essentials-for-data-science-why-cloud-computing-141cc6cee284). While you sacrifice some control by not having access to the physical machine, you’ll abstract away a lot of configuration and maintenance you likely don’t want to deal with anyway. And if you’re willing to pay a bit more, you can easily rent a machine — or several — that are significantly stronger than your laptop. So how do we get started?

We previously covered a [high-level overview](https://towardsdatascience.com/aws-essentials-for-data-science-why-cloud-computing-141cc6cee284) of the cloud, as well as a tutorial on [storing data](https://towardsdatascience.com/aws-essentials-for-data-science-storage-5755afc3cb4a). But what about the *engines* of the cloud? In this final post, we’ll cover two compute-focused **Amazon Web Services**. We’ll start with the fundamental cloud building block, **EC2**, before moving on to server-less computing with **Lambda**.

**Background**

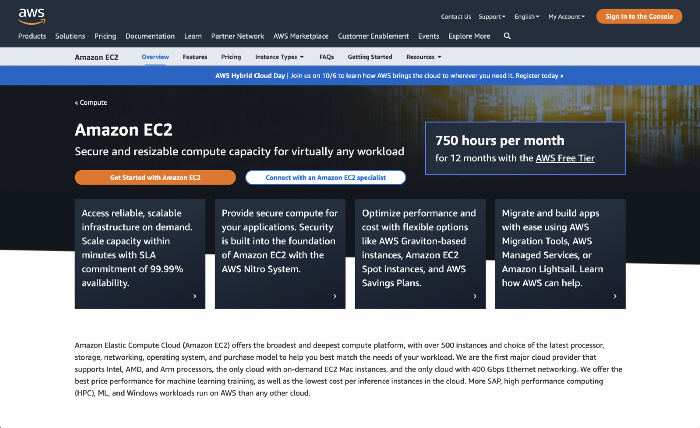
The holiday season is a recurring chaotic time for retailers. Q4 accounts for **a staggering 33–39%** of [Macy’s](https://ycharts.com/companies/M/revenues) and [Kohl’s](https://ycharts.com/companies/KSS/revenues) yearly revenues, for example, and even with Prime Day in the summer, [Amazon’s](https://ycharts.com/companies/AMZN/revenues) Q4 is still around 31%. Much of this holiday rush [takes place online](https://www.cbre.com/insights/articles/omnichannel-what-is-the-share-of-e-commerce-in-overall-retail-sales), translating to *a lot more users* spending *a lot more time* on stores’ websites.

Put yourself in the shoes of an Amazon infrastructure engineer in October 2005, a few years after the [dot-com bubble](https://en.wikipedia.org/wiki/Dot-com_bubble), but before the cloud industry really started. You know you have to do *something* to handle the tsunami of traffic on the horizon: the last thing you want is for the site to be down, [millions of dollars of sales slipping by](https://www.independent.co.uk/news/business/amazon-down-internet-outage-sales-b1861737.html) as frustrated shoppers switch to another website. [1]

One way to handle the additional load is to *buy more computers*. (There are indeed [stories of early Amazon engineers](https://open.spotify.com/episode/14LmWeOMRZysw2i2vYSOuw?si=ce630660e3b44461) preparing for the holidays by buying the most powerful servers they could find and crossing their fingers!) These extra servers should indeed handle the spike in traffic. But when the holiday buzz ends, that extra compute will end up sitting around unused until the next holiday season. [2]

The alternative would be to *rent* compute somehow. Ideally, resources would **elastically** and **automatically** increase and decrease to your immediate needs, rather than needing to guess ahead of time. You’d abstract away the physical hardware, instead just dipping into a “pool” of resources.

Amazon Web Services was born out of needs like these in the fledgling internet: dynamically accessing the compute resources you need, when you need them. We’ve [already covered](https://towardsdatascience.com/aws-essentials-for-data-science-storage-5755afc3cb4a) one of their fundamental *storage* offerings: Amazon Simple Storage Service (S3), a catch-all Dropbox analogue for storing data. But let’s now turn to Amazon’s fundamental *compute* offering: **Elastic Compute Cloud.**



Screenshot by author

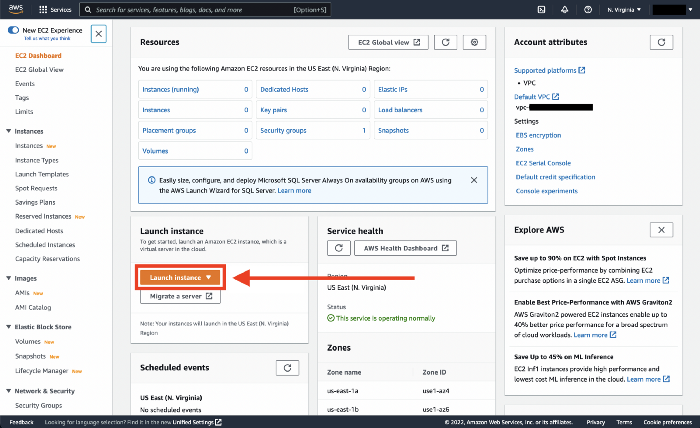
**EC2: Elastic Compute Cloud**

We can use **Amazon EC2** to access the fundamental building block of the cloud: the **virtual server**. Data centers are filled with servers, which are [logically partitioned](https://en.wikipedia.org/wiki/Logical_partition) into virtual servers, allowing multiple people to simultaneously and independently use the hardware.

One server could be simultaneously running simulations for a weather forecast, fetching data from multiple databases, sending the HTML for a dozen webpages, and more. Importantly, this physical server would be abstracted away from its users beyond the configurations of their virtual servers, letting them focus on their applications.

At AWS, virtual servers are called **EC2 instances**. Released in 2006, EC2 was [one of Amazon’s first cloud services](https://aws.amazon.com/blogs/aws/aws-blog-the-first-five-years/) and has grown to be a [central component of the tech stacks](https://aws.amazon.com/ec2/customers/) of Netflix, Pinterest, Lyft, and many others. EC2 instances are modular and configurable, allowing users to optimize for compute, memory, GPU, storage, or a combination depending on their needs. A GPU-optimized instance could be used for training machine learning models, for example, while a storage-optimized instance could host a database.

Let’s now create an EC2 instance to take a closer look. We [log into our AWS account](https://aws.amazon.com/login), then navigate to EC2 from the menu of services. We should see something like the image below. Let’s click on the Launch instance button and begin.



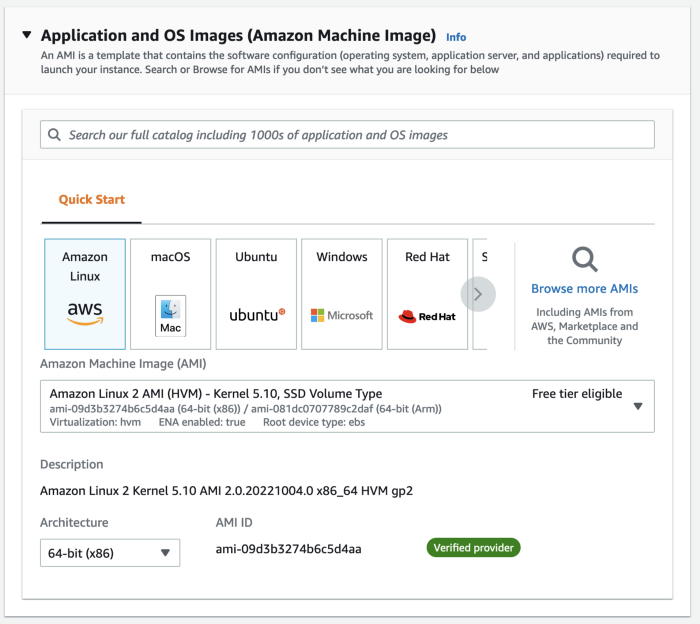
Screenshot by author

**Set up**

**AMI: Amazon Machine Image**

The first thing we’ll do when launching an EC2 instance is select the [**Amazon Machine Image**](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html). An AMI specifies the basic software configuration our instance: the operating system, [application server](https://www.gartner.com/en/information-technology/glossary/application-server), and applications required for your server to run. AMIs are like [Docker images](https://www.tutorialspoint.com/docker/docker_images.htm): **reusable templates** that let us create the exact environment we want each time.

The default AMI comes with a Linux kernel optimized for EC2, [the system and service manager *systemd*](https://en.wikipedia.org/wiki/Systemd), [the GCC compiler](https://en.wikipedia.org/wiki/GNU_Compiler_Collection), and other very low-level software. We could create our own AMI if we had strong opinions about how to optimize for our use case. But this is an intro tutorial, so let’s just choose the default Amazon Linux 2 AMI.



Screenshot by author

**Instance Type**

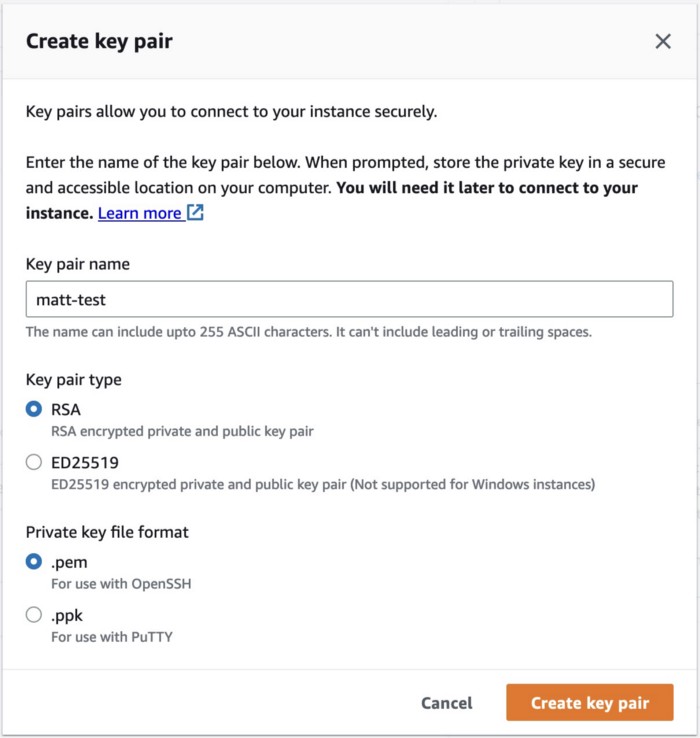
Next up is **Instance Type**, where we choose the hardware for our server. We won’t want to deviate from the t2.micro option, which is covered by the Free Tier. In a production setting, we could decide to optimize for **CPU** (for running a [wide range of system operations simultaneously](https://www.weka.io/learn/hpc/cpu-vs-gpu)), **GPU** (for machine learning or graphics processing), **storage** (for slow reads and writes of persistent data), **memory** (for [fast reads and writes of volatile data](https://www.backblaze.com/blog/whats-diff-ram-vs-storage/)), or some combination.

We can’t change the instance type once we launch our instance, so make sure you don’t accidentally click [the one that charges $31.21 per hour](https://www.todayilearnedcloud.com/Amazon-EC2-How-Much-Does-The-Most-Expensive-Instance-Cost/)! Triple-checking that we’ve selected t2.micro, we can continue to the next step.

**Key Pair**

We’ll now create a key pair. [AWS uses public-key cryptography](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/get-set-up-for-amazon-ec2.html) to secure the login information for EC2 instances. Rather than a password, we’ll use the key pair to remotely log into our instance via [SSH](https://www.techtarget.com/searchsecurity/definition/Secure-Shell).

We’ll click on Create a new key pair. We then give it a name and stick with the RSA and .pem defaults. (Select PuTTY if you’re using Windows.)



Screenshot by author

Once we click Create key pair, Amazon will save a public part of our key, and our computer will download the private key. Make sure you don’t lose this .pem (or .ppk) file, as we’ll use it to identify ourselves when remotely accessing the EC2 instance.

**Network settings**

We now set the rules for how to access our EC2 instance via the internet. For this demo, let’s just click Select existing security group, then our default VPC ([virtual private cloud](https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html)). If you followed along in the previous [AWS Storage post](https://medium.com/towards-data-science/aws-essentials-for-data-science-storage-5755afc3cb4a), you’ll have already tinkered with the inbound and outbound access rules for this VPC.

**Configure storage**

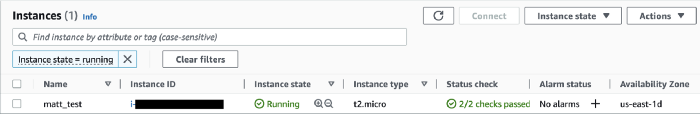
We’ll leave our storage config at the default values, which are well within the Free Tier limits. Any data we write to our instance will be deleted once our demo is over; if we cared about persisting this data, we could click Add new volume to reserve an [Amazon Elastic Block Storage (EBS)](https://aws.amazon.com/ebs/) volume and save our data there. But let’s stick with the root volume for now.

**Advanced details**

We’ll skip this section for our demo. But this is where we can specify configurations like using on-demand [spot instances](https://aws.amazon.com/ec2/spot/), shutdown and [hibernate](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/Hibernate.html) behavior, whether we want detailed [Amazon CloudWatch](https://aws.amazon.com/cloudwatch/) logs, and more.

**Connecting to our instance**

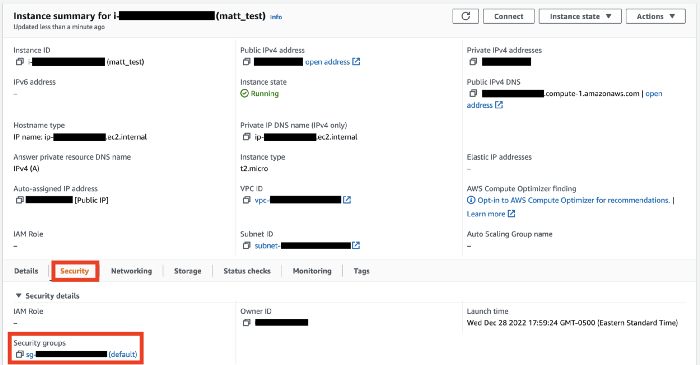
Our instance will be available within a few minutes after we hit Create. On the EC2 home page, we can then click on Instances and see something like the image below.



Screenshot by author

Let’s now connect to our instance. We’ll use [**SSH**](https://www.techtarget.com/searchsecurity/definition/Secure-Shell), a network protocol that enables secure communication over an unsecured network (like the internet). Once we’ve SSH’d into our instance, we will be able to control the machine as if we were in a Terminal on our laptop.

The first thing to do is **modify the instance’s security group to allow inbound traffic.** On our instance page, we’ll click on the Security tab at the bottom, then the link for the instance’s security group.



Screenshot by author

On the Security Group page, click Edit inbound rules, then Add rule. Select **SSH** for Type, then **My IP** for Source. Finally, click Save rules.

So let’s try connecting to our instance now. The command is ssh followed by the Public [IPv4 DNS](https://www.cloudns.net/blog/what-is-ipv4-everything-you-need-to-know/), available on the Instance page. You can also go to Connect > SSH client to get the address.

We get a Permission denied error because we haven’t passed in our private key. Let’s change directories to the one with our key and try again.

So far so good, but then we get another error:

The error is that our private key has [permission code 644](https://chmodcommand.com/chmod-0644/), which means that anyone can read the file. This is the [default access level](https://www.namecheap.com/support/knowledgebase/article.aspx/400/205/file-permissions/) for new files, but AWS considers this too insecure: anyone with your key could impersonate you.

So we’ll need to modify the file privacy to make it more secure. To do so, we use [**chmod**](https://en.wikipedia.org/wiki/Chmod) to change the read and write permissions of the file. Specifically, we’ll change the file so that **the only valid action is a read by the owner (us).** Even if someone copied our private key to another computer, or a different user on our network somehow found the key, the file wouldn’t open because that person isn’t the owner. As an additional precaution, we remove our write access as well.

The chmod code for [this permission level is 400](https://chmodcommand.com/chmod-400/): owner can only read (4), security group can’t read/write/execute (0), and others can’t read/write/execute (0). Let’s therefore run this command in the Terminal:

Now when we try to connect, we succeed:

We’re in! To the left of our cursor we should see something like [ec2-user@ip-xx-xx-xx-xxx ~]$, where we previously just saw $. Let’s now run some quick commands to explore the instance.

Python already comes installed, which is convenient! Let’s download a [sample Python file](https://raw.githubusercontent.com/mgsosna/code_samples/master/calculate_mean.py) from GitHub and run some basic calculations on our EC2 instance. We’ll use curl to download the file from the URL, then pass in some arguments to calculate\_mean.py to get their average value.

Let’s now train a random forest classifier on some generated data. We’ll download numpy, pandas, and scikit-learn, open Python, generate the data, then create the model.

We’ll first download the necessary libraries.

We can now run Python, generate our data, train the model, and make some predictions. We’ll use np.random.normal for the features and np.random.choice for the labels.

Ta da! You’ve (technically) trained a machine learning model in the cloud. Since we’ve hit the pinnacle of EC2 use cases (😜), let’s sever our SSH connection and terminate our instance.

Now on our Instance page on the EC2 page, we can click on Instance state > Terminate instance. Note that we’ll lose our Python libraries and the calculate\_mean.py file, since the instance’s data will be wiped as it’s made available for someone else to use. If we want to hold onto the instance a little longer, we can click Stop instance instead.

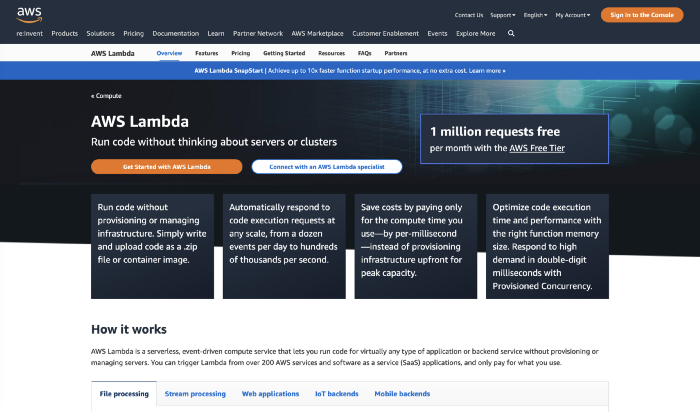
**Beyond the basics**

So we just created an EC2 instance, downloaded files from the internet, and ran some Python code. While this is awesome, we haven’t experienced anything that we couldn’t run on our own laptop, which is likely more powerful than a t2.micro server. **So what value are we really getting from EC2?**

The first thing to note is that **there are a wide range of EC2 options beyond the Free Tier.** If we wanted to run simulations for a research paper, we could simply select an EC2 instance with more CPU or GPU than our laptop. This would get the job done more quickly, especially if we don’t have a decent computer or can’t dedicate all of its resources to the simulations. (It would also prevent damaging your own laptop, which I accidentally did in college with a horribly inefficient R script!)

But more importantly, we need to remember that **EC2 instances are just *building blocks*.** For our research simulations, it may be more efficient to rent two or three instances and parallelize the calculations. If we’re using an EC2 instance to host a Flask API for our website, when traffic grows we can simply duplicate the instance and add a load balancer to distribute traffic. Better yet, [we can automatically scale](https://aws.amazon.com/ec2/autoscaling/) the number of instances up and down to meet demand, letting us focus more on our actual application. Q4, while still a demanding time for retailers, has become so much more manageable with cloud computing.

All this abstraction is a tremendous step forward from the internet of the early 2000’s. And yet… sometimes even a virtual server isn’t flexible or scalable enough for our use case. In that case, we may want a “server-less” option like **Amazon Lambda.**



Screenshot by author

**Lambda**

**When is EC2 not the right choice?**

With EC2, we’re given an entire virtual server. AWS gives us incredible flexibility in defining what this server will look like, letting us tinker with every hardware and software detail imaginable. This post’s EC2 setup section for a simple demo, for example, was dozens of lines long!

This flexibility means we can find some combination of EC2 instances that will meet essentially any task we can imagine. **But this puts the responsibility on *us* to make sure we’re configuring our instances properly.** This configuration may be straightforward if we’re performing a single task like training a machine learning model. But optimizing our resource usage becomes much more challenging when we’re juggling multiple instances, *especially if we have multiple instance types*. If our app is clunky, should we add more instances to host our database, fetch data from the database, or display it on the frontend? And how does this change as traffic to our app changes?

Another issue is that **our instance is only accessible while it’s running.** Just like how our laptop won’t respond to inputs while it’s sleeping or turned off, we can’t use our EC2 instance unless it’s running — ***and charging us for its usage.*** This is fine if we’re constantly sending requests to our instance. But if we expect to use our instance sporadically, then it can be expensive and inefficient to keep the instance running while we’re not using it.

**What is Lambda?**

In 2014, Amazon released [**Lambda**](https://aws.amazon.com/lambda/), pushing the abstraction of the cloud to a new level with ***server-less* computing**. In contrast to EC2, we’re given *zero* information about the machine(s) running our code. Our [only tunable parameter](https://aws.amazon.com/lambda/pricing/) is the amount of memory our code is allowed to use.

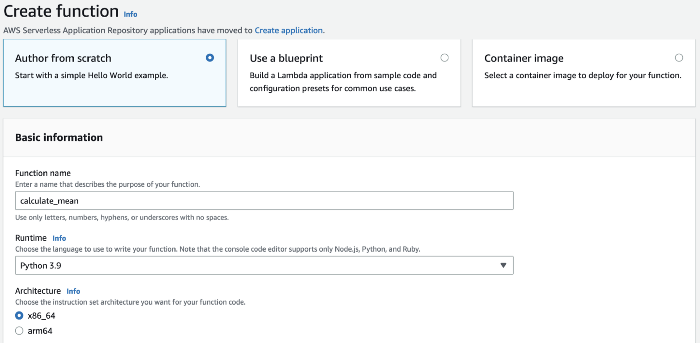
Lambda executes code in response to *events* and automatically manages the underlying compute resources. If our code isn’t actively running on a machine, we’re not paying for anything. This is ideal for one-off actions like [writing user actions to a database](https://docs.aws.amazon.com/lambda/latest/dg/services-apigateway-tutorial.html) or [generating predictions from an ML model](https://aws.amazon.com/blogs/compute/pay-as-you-go-machine-learning-inference-with-aws-lambda/), but we can even build a [microservice-based web app](https://aws.amazon.com/getting-started/hands-on/build-serverless-web-app-lambda-apigateway-s3-dynamodb-cognito/) or [real-time streaming data processor](https://aws.amazon.com/blogs/compute/building-serverless-applications-with-streaming-data-part-1/).

Our workflow in Lambda will involve setting up ***functions*** that we then *trigger* from other services. No need to stress about configuring hardware, ensuring its security patches are up to date, and efficiently utilizing our resources.

The one major downside is that **because we don’t have a dedicated virtual server waiting for requests, there is a brief delay when we first begin triggering our functions**, as Lambda needs to find and reserve the necessary resources for us. Scattering our application across many Lambda functions, as opposed to one monolithic EC2 instance, can also **significantly increase the complexity of our app** as it grows. But these are often small prices to pay for the convenience of having the hardware completely abstracted away from us.

**Creating a function**

Let’s explore Lambda by setting up a function that returns the mean of a set of numbers. In the AWS Console, we start by navigating to the Lambda homepage. We then click the big orange Create function button. We’ll stick with the “Author from scratch” option, then give our function a name (calculate\_mean) and select the Python 3.9 runtime. Then we scroll down and click Create function.



Screenshot by author

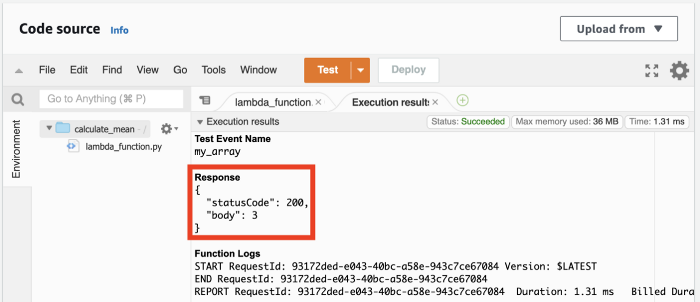
We’re then taken to the Lambda function page. Scrolling down to the Code tab, we see a basic template provided for our function:

And in the Test tab, we see a simple JSON input we can use to test our function.

We’ll want to change both of these for our calculate\_mean function. Let’s start by changing the test input to an array of numbers like below. Give your test a name (e.g., my\_array) and hit Save. You should see a green banner at the top that says, *“The test event* ***my\_array*** *was successfully saved.”*

Now let’s go back to the Lambda function and change it to the code below. Specifically, we remove the json import, extract the nums field from event, and change the return body field to calculate the mean. In a production setting, we’d want to add error handling for empty arrays, arrays with non-numbers, etc., but this is fine for now. [3]

Save your function (⌘ + s, or File > Save), then click Deploy. You should see a green banner that says, *“Successfully updated the function* ***calculate\_mean****.”* Now hit Test, which should generate some logs like below. The import thing is that the response body field should be the mean of the array of numbers in your test file (for us 3, the mean 1 to 5).

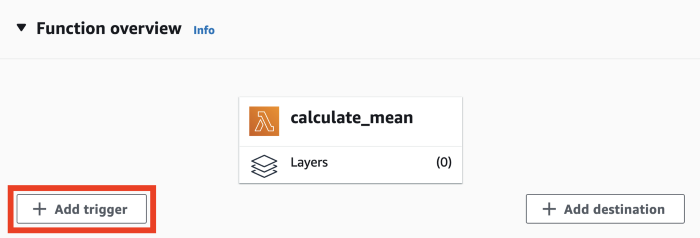


Screenshot by author

Let’s try this now from the **AWS CLI**. (If you haven’t downloaded the CLI, you can follow [these steps](https://medium.com/towards-data-science/aws-essentials-for-data-science-why-cloud-computing-141cc6cee284).) Open a Terminal window, create an input JSON file, and send the file to your Lambda function. Lambda will return the response as a file, output.json here, which we can then view.

**Triggering via API Gateway**

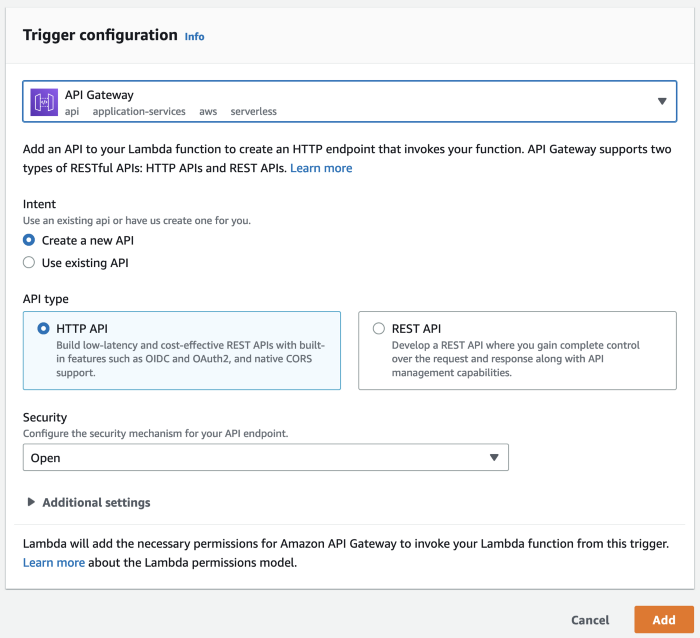
Congrats, we have a Lambda function! But we can only really interact with this function in the Lambda console and on our personal computer. What we really want is **to be able to trigger this function from anywhere.** So back in the browser, let’s scroll to the top of the calculate\_mean function page, where we can see the triggers (and destinations) for our function.



Screenshot by author

When we click Add trigger, we’re taken to a drop-down menu with an astonishing number of services, from [Alexa](https://aws.amazon.com/alexaforbusiness/), [AWS IoT](https://aws.amazon.com/iot/), and [DynamoDB](https://aws.amazon.com/dynamodb/), to non-AWS services like [Auth0](https://auth0.com/), [Datadog](https://www.datadoghq.com/), and [Shopify](https://www.shopify.com/). We could have Auth0 trigger a Lambda function that writes to a database whenever a user logs in, for example, or [Amazon SNS](https://aws.amazon.com/sns/) send us a notification any time a new file is uploaded to one of our S3 buckets.

Let’s choose [**API Gateway**](https://aws.amazon.com/api-gateway/) to create an HTTP endpoint for our function. This will let us invoke our Lambda function from any code that can send an HTTP request. We’ll select Create a new API, HTTP API for API type, and Open for Security. Then click Add.



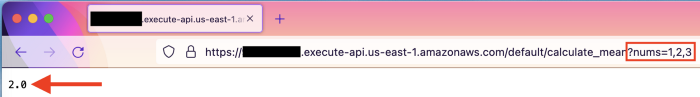
Screenshot by author

We should now see API Gateway as a trigger for our calculate\_mean function. The endpoint should look something like <https://xx.execute-api.us-east-1.amazonaws.com/default/calculate_mean.>

If we click on the link, we’re taken to a page that just says “Internal Server Error.” This is because our Lambda function is configured to expect a JSON with the field nums, but our browser’s HTTP request doesn’t have this field. An easy way to pass in this field is as a [**query string parameter**](https://www.positly.com/support/query-string-parameters/), where *the URL itself* has nums and the array of values. So instead of /default/calculate\_mean, we would have something like /default/calculate\_mean?nums=1,2,3.

So let’s modify our Lambda function to accept URL parameters. We’ll add a try-except block, where we first try pulling the nums param from the query string (event['queryStringParameters']). We convert the string to a list of strings, then cast each number to float. If the user passes in a JSON (as we did previously), we just pull the nums array from the event object.

Don’t forget to save the function and deploy it. Wait a few seconds, then copy your API endpoint into the URL address of a new tab on your browser. Add ?nums=1,2,3 to the end of the URL and hit enter.



Screenshot by author

Awesome! Let’s try one last thing. Open a Python window on your local machine (or your EC2, if you’re feeling fancy) and run the following:

We see that we can even pass in floats via the URL string to our function, and that we’re able to access it from anywhere. Congrats! 🎉

**Cleaning up**

Our demos are over, so let’s make sure to delete everything we created to avoid incurring charges. We should **start with our API endpoint, which is currently available for anyone on the internet to use** — if someone guesses our URL, they could start sending hundreds or thousands of requests per second, which we’ll have to pay AWS for.

So let’s navigate to **API Gateway** in the AWS Console and delete our API endpoint. You click on the button next to calculate\_mean-API, then Actions > Delete.

Next up is the Lambda function. Navigate to **Lambda** in the AWS Console and delete the function.

Finally, if you haven’t already, terminate your EC2 instance. Navigate to **EC2** in the console, stop your instance if you haven’t already, and then terminate it. You can also go to your security group and remove your IP address from the inbound traffic rules.

**Conclusions**

In this post, we explored two Amazon Web Services for compute: **EC2** and **Lambda.** We saw how **Amazon EC2 lets us reserve a virtual server with whatever exact configuration we’d like**, tailoring our resources to our use case. We created a server and SSH’d into it after modifying the read permissions of our private key. We then downloaded files from the internet, ran a Python script, and trained a classifier on some generated data.

We then contrasted EC2 with **AWS Lambda, Amazon’s “server-less” approach to compute.** We created a function for calculating the mean of an array of numbers and invoked the function from the console and AWS CLI. We then added an AWS API Gateway trigger, modified our function to take query string parameters, and called the function from our browser and local Python environment.

Between our [intro,](https://towardsdatascience.com/aws-essentials-for-data-science-why-cloud-computing-141cc6cee284) [storage](https://towardsdatascience.com/aws-essentials-for-data-science-storage-5755afc3cb4a), and compute posts, we’ve used seven fundamental services: IAM, S3, RDS, DynamoDB, EC2, Lambda, and API Gateway. While Amazon offers [over 200 services](https://www.aboutamazon.com/what-we-do/amazon-web-services), many build on these core services. With the knowledge we’ve gained in this series, we’re well equipped to begin leveraging the cloud for our needs.

Thanks for reading!

Best,  
Matt

**Footnotes**

**1. Background**

In researching for this post, I found plenty of interesting statistics about how expensive it can be for a popular website to be unresponsive or unavailable. Some of the more interesting stats:

* [Taobao](https://en.wikipedia.org/wiki/Taobao), a Chinese online shopping platform, had a 20-minute crash during Singles’ Day in 2021 that may have cost [**several billion dollars**](https://queue-it.com/blog/singles-day-statistics/) in sales.
* A webpage that loads within two seconds has an average [bounce rate](https://en.wikipedia.org/wiki/Bounce_rate) of 9%. That number [**jumps to 38%**](https://www.pingdom.com/blog/page-load-time-really-affect-bounce-rate/) when the webpage takes five seconds to load.

**2. Background**

For hyper-growth early Amazon, the extra compute purchased during the holiday season would eventually just serve the normal business needs as the company grew. But for most companies, this extra compute would be a hindrance most of the year.

As a side note, there’s a common narrative that AWS spun out of Amazon trying to utilize all this “extra compute” sitting around Q1-Q3. They had all these unused servers, so why not just let customers use them? My [favorite rebuttal](https://open.spotify.com/episode/14LmWeOMRZysw2i2vYSOuw?si=ce630660e3b44461&nd=1) of this narrative is that when Q4 came up the following year, Amazon obviously couldn’t just terminate all those customers to take back their servers! Amazon would be stuck needing to buy a bunch more servers again.

**3. Creating a function**

We can argue whether error handling should be within the Lambda function or at callers to the function. There are tradeoffs: adding error handling to the function means the checks are in one place, reducing mental load on developers interacting with the function. But checks make the function heavier — they need to run *every* time the function is called.

We can keep the function lightweight by having callers guarantee valid inputs. But enforcing this responsibility is challenging, especially if the codebase is large and developers don’t necessarily have the full context of the system. This also means there’s no standardization on what checks are run and how errors are communicated back to the user.

One intermediate option could be to have checks performed by [one Lambda function that then passes valid inputs to the actual function](https://stackoverflow.com/questions/31714788/can-an-aws-lambda-function-call-another). This increases the overall system complexity, but it allows for both reduced mental load on callers and a lightweight function. Ultimately, the right approach depends on your use case.