How I built a Scalable Web-Scraper with AWS

My proposal for a fully managed cloud-based Python scraper

Aaron Langley Jul 29, 2020 · 7 min read

How I imagine my AWS console… (Photo by Patryk Grądys on Unsplash)

Early last year, I found a crazy deal on a used car from Craigslist. After double checking that it was not a scam (it was not). I naturally went straight to Kelly Blue Book to get the "real value" of my new car — but this got me thinking:

It seems that the first resource they must use is Craigslist — since this is the most popular peer-to-peer used car listing platform (and *only* major platform, until recently). if this is the case, Kelly-Blue-Book must somehow have access to Craigslist data. Craigslist does not offer an API, however, nor do they open their datasets (assuming they have any) to the public. This left me with a thought that perhaps companies like KBB are scraping some of their data (probably a safe guess).

Therefore, I decided to collect data to build my own Kelly-Blue-Book-esque platform. To do this, I needed to build a scraper with these requirements:

1. Easily-scalable and dynamic scraper to accommodate the many Craigslist cities, listings, and listing-types.
2. Managed processing of hundreds of jobs per day — staggered and carefully scheduled in order to not DDoS Craigslist servers (or get blocked by them).
3. Monitoring of each job to immediately notify me of any runtime failures.
4. And lastly, CI/CD to manage rapid deployments of all services involved.

Here's a high-level diagram of my solution:

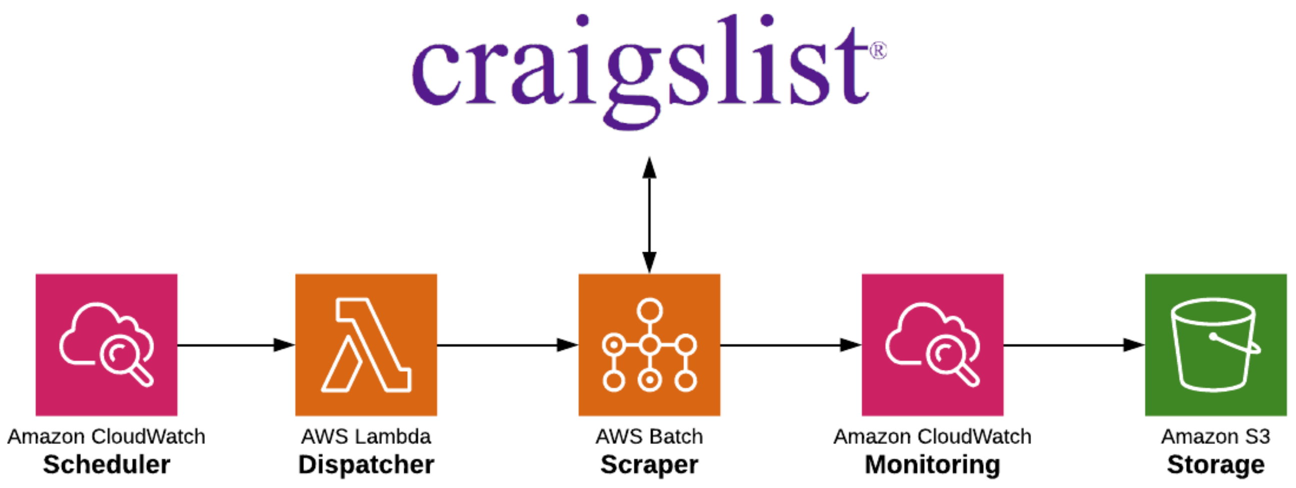


Image by author

Essentially, I needed to build an ETL pipeline to collect Craigslist data on a daily basis, in order record the lifespan and events (price changes, de-listings) of every car listing.

later — keep reading!).

Disclaimer: Please understand that this project is built for research only. While it has the capability of hitting Craigslist with thousands of requests in a very short time (or crashing your computer), I have been careful to limit the number of requests in a given time by strategically spacing the jobs so as to not spam/DDoS Craigslist servers. **If you are scraping Craigslist (or any site for that matter), be sure to read and adhere to the robots.txt.**

Let's dive right in…

While I won't be going step-by-step through the the whole process tutorial-style; I do want to give you an idea of how the scraper itself works.

The scraper is run inside a Docker container — the code itself is very simple, you can find the whole project here. It is built in Python and uses the BeautifulSoup library.

There are several environment variables passed to the scraper. These variables define the search parameters of each job. Essentially, the container's lifecycle follows these three steps:

1. The container is passed a few variables, primarily, the City/search-region, and the vehicle Make.
2. The container searches Craigslist based on parameters given and generates structured results from the HTML.
3. The container sends these results to S3 in CSV format.

Process and Infrastructure

Here's the cool part…

Now let's go over the extraction/scheduling process in detail.

1. Every day, my AWS **CloudWatch** event rules trigger lambdas to dispatch scraping jobs for different sets of states and vehicle makes.

the environment's configuration (compute environment determines job concurrency — read more about AWS Batch here).

4. **CloudWatch** processes and notifies on failures as Batch sends status updates.

5. At the end of each Batch job, the scraper will send a newly generated CSV of search-results to my **S3** bucket under the naming convention: searchtype_make_city_timestamp.csv. I chose to put the "make" before "city" since the make is more important to the price and lifespan of the vehicle listing.

Here's a high-level diagram of the scraping process:

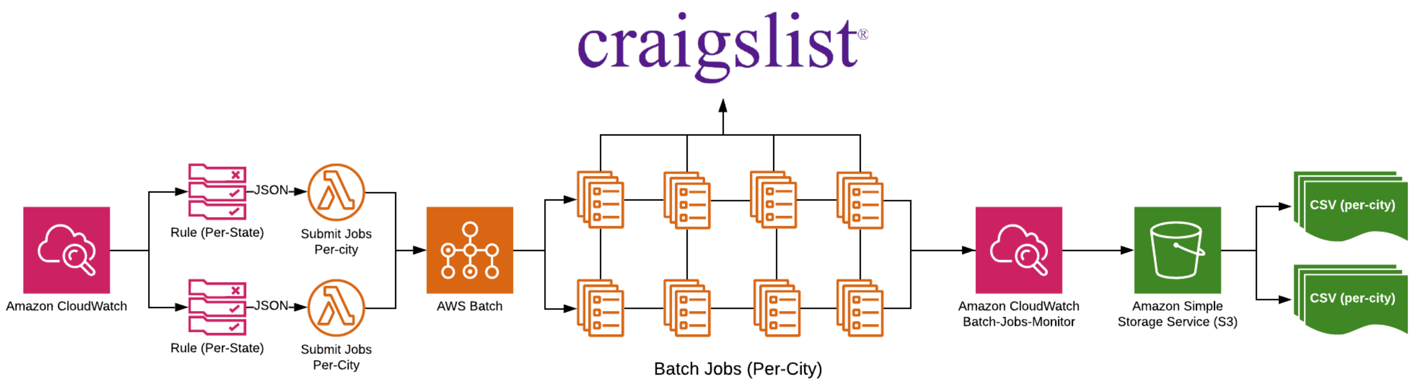


Image by author

As you see from the diagram, I am using CloudWatch, Lambda, Batch, S3. I am also using SNS for notifications triggered by the "Batch-Jobs-Monitor."

Here is my reasoning for the services I chose:

CloudWatch has "Rules" which behave as Cron Jobs and can pass JSON payloads to lambda functions. This enabled me to submit multiple CloudFormation Templates with job parameters in JSON format as the to be executed according to the pre-determined Cron expressions.

Lambda is fantastic for the smaller tasks (dispatch and notifications), and integrates easily with almost every Amazon service.

speed-up each job, I decided on multithreading the requests in my python scraper. This meant I needed a service that would allow for long runtimes, and a high I/O rate. Lambda was out of the question due to these conditions (and Lambda does not support Python's multi-threading packages). I found that AWS Batch was perfect — I was able to configure the compute environments to fit my needs (and budget), while having an easily maintainable Dockerized scraper in a pay-per-use service. Another major perk is that AWS Batch uses ECS to run the jobs, therefore, every time a new queue is processed, AWS Batch spins up new EC2 instances — **with new IP addresses** (essentially rotating IP's).

S3… Well S3 is a quick and inexpensive way to temporarily store my semi-structured scraper-results in an accessible location. I did have to be thoughtful of my object-naming standards. S3 buckets can get messy with hundreds of thousands of objects if naming-standards are not taken into consideration— the AWS SDK's do not provide robust object-searching capabilities. Soon, I will be migrating my data out of S3 and into a structured relational database.

SNS is great for the few text-notifications I need to send myself — and easily integrates with CloudWatch and Lambda.

CI/CD overview

This part was important to me, mainly because there were many moving parts to my scraper — and the AWS console can be tedious.

I used both **CodePipeline** and **TravisCI** for commit-triggered building/deployment. This was mainly for my own experimentation. I found that TravisCI is able to do everything I needed, so I will be moving my CodePipeline projects to TravisCI and save that extra $1 per month 😉

I used the **Serverless Framework** to handle Lambda deployments via TravisCI's build-scripts.

The python scraper is containerized and deployed to Amazon's **ECR**, where the image is referenced/launched as needed by AWS Batch.

I will admit, I took a hacky approach to my Cron Event-Rules: I wrote **CloudFormation** templates with the event payload embedded as JSON.

```
        "group1a": {
            "Type": "AWS::CloudFormation::Stack",
            "Properties": {
                "TemplateURL": "./scraperCronStack.json",
                "Parameters": {
                    "description": "group 1a cron(0 10 2/2 * ? *)",
                    "cronExpression": "cron(0 10 2/2 * ? *)",
                    "jsonInput": "{\"states\":[\"New York\",\"Oregon\",\"Utah\",\"Ohio\
                }
            }
        }
    }
}
```

SampleCLEventRule.json hosted with ❤ by GitHub view raw

The escaped "jsonInput" is a little messy, but I am able to easily add or remove jobs via these templates. For example, if I wanted to add another scraping job, I would just add "group2a" with new "jsonInput" and "cronExpression" values.

I should note the "TemplateURL" routes to an event-rule CloudFormation template pointing to a single Lambda function which is configured to receive the "jsonInput" and dispatch jobs to AWS Batch accordingly.

When a change is needed, the templates are modified and committed to Github. As soon as the pull-request is merged to my master branch, TravisCI updates the CloudFormation stack with the latest changes. This process makes it extremely simple for me to scale-up or scale-down my Scalable-Scraper.

Now for the Whole Shebang

…and here is my whole solution — from CI/CD to text-alerts:

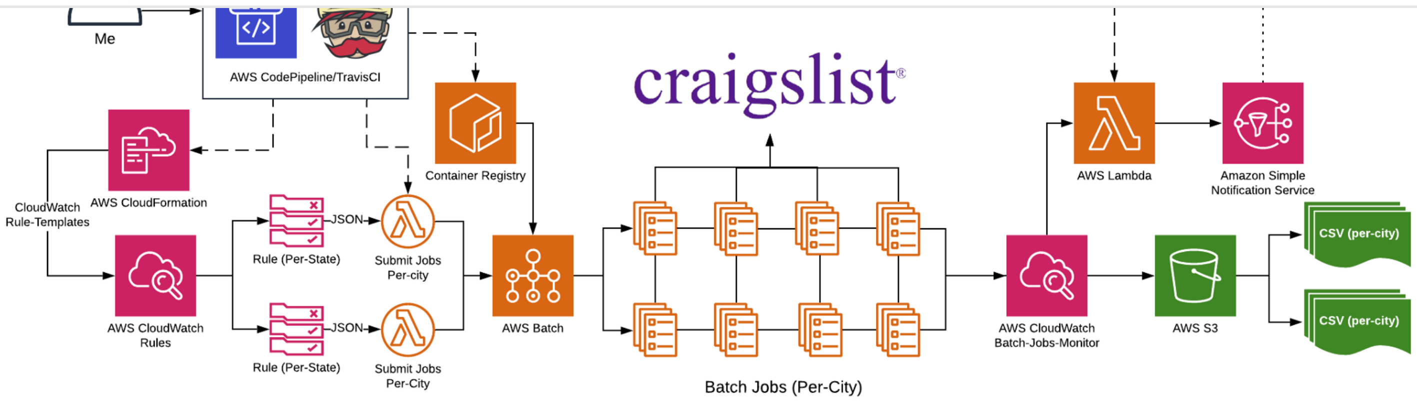


Image by author

Stay tuned!

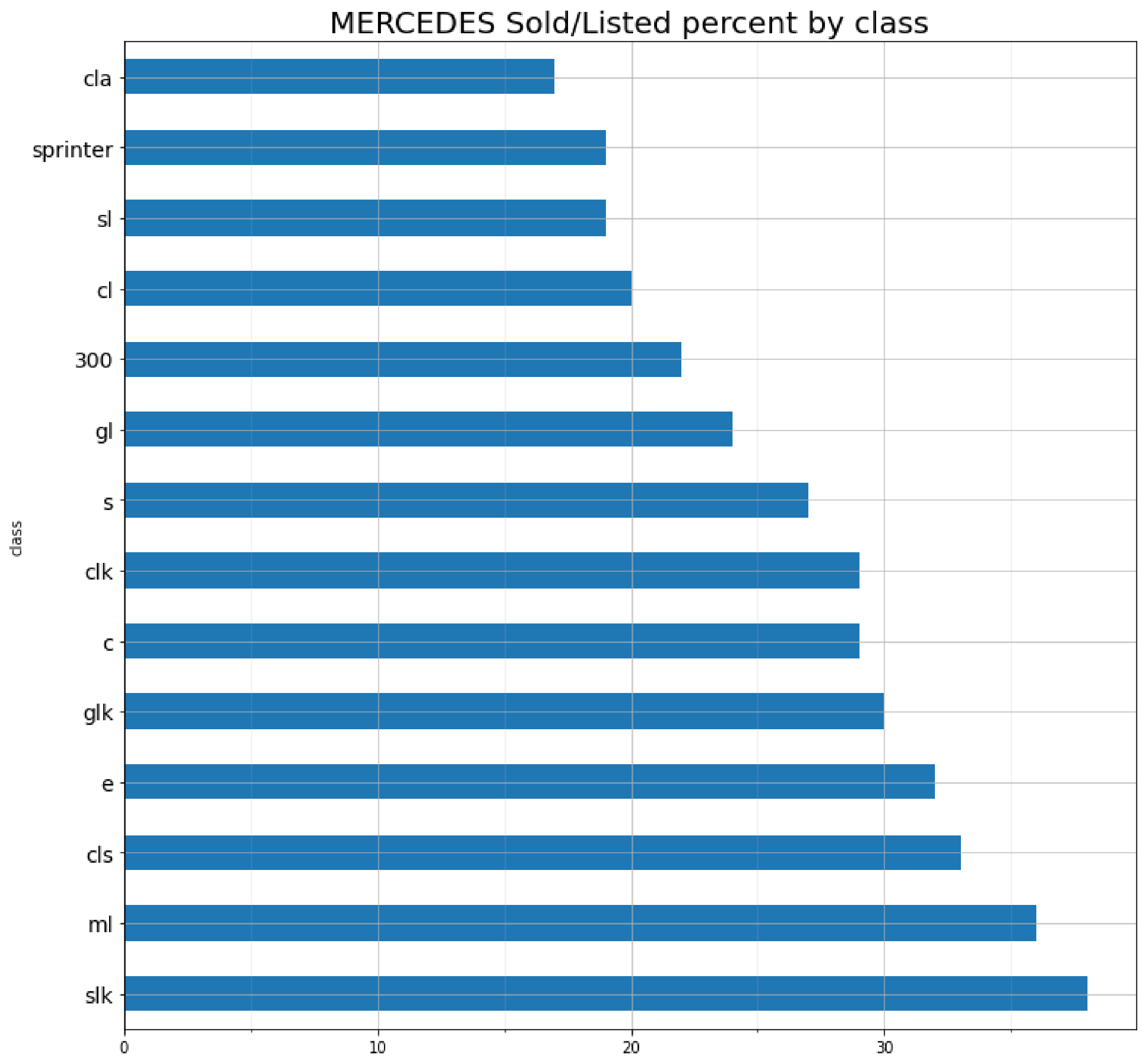
Do you want to find out how to get a terrific deal (you wouldn't believe the price) on a beauty like this? Or possibly you are more interested in the project itself…

1998 Mercedes SLK 230 (supercharged), 50k miles, hardtop convertible (everything works perfectly). Yes it's mine. (Image by author)

Mercedes-Benz data (I have about 6 months of data now!). Guess what I found?

According to this (small) dataset, my SLK has the highest selling rate of all Mercedes-Benz listings on Craigslist!

Check it out:



There's the SLK with a ~38% selling rate. Graph created by author.

That's some cool data, right?

I would love to hear your thoughts on this solution — there is more than one way to skin a cat. Leave comments below!

Cheers!

— Aaron Langley

Web Scraping Python AWS Ci Cd Pipeline Etl