Data Aggregation Reference Architecture

George Ludwig, Solutions Architect - Global Alliances
gludwig@indeed.com
version 1.1 June 22, 2018

Background

There are a number of Applicant Tracking Systems (ATSes) that have a distributed information architecture. In other words, the ATS joba data is located in multiple locations, each of which has its own address, and potentially requires its own authorization credentials. We refer to these as 'endpoints'. In some cases, the endpoints correspond to regional data centers owned by the ATS. In other cases, the ATS customer runs the ATS software onsite, effectively making each ATS customer an endpoint. But in all cases, in order for Indeed to accept the jobs data of the ATS, the jobs data must be aggregated to a single XML file that contains all of the ATS data, accessible via FTP.

Purpose

The goal of this project is to provide a blueprint for how an ATS can easily implement an inexpensive, fault-tolerant solution to the Jobs Data Aggregation problem.

Technical Overview

The reference architecture is designed to be as reliable as possible while also being as cost-efficient as possible. In order to provide a solution that is usable by all ATSes, the system is designed to be run on Amazon Web Services.

To address cost-efficiency, the system is designed to be run using t2.micro compute instances, the second-least expensive compute option. With that in mind, the processes that extract and aggregate the jobs data do not rely on loading all jobs data in to memory at once. A single t2.nano instance serves as the data aggregator, and a second t2.nano instance serves as the FTP server.

The launch configurations for these instances start with a standard AMI for Amazon Linux, and have boot scripts that automatically update the system, installs and configures software, mounts the Elastic File System, and downloads code and configuration data from a private S3 bucket in order to configure the Data Aggregator server and the FTP server. See appendix for script examples.

To address availability, both the data aggregator and FTP server are launched into singleton autoscaling groups. This means that there will always be one-and-only-one instance of the data aggregator, and one-and-only-one instance of the FTP server. Each instance is assigned an Elastic IP address, so that in the case that an instance is replaced due to a failed health check, its replacement will maintain the same IP address. Also, both the data aggregator instance as well as the FTP server instance will use the same Elastic File System mount point for data storage. In this way, the two instances share data, and also preserve that data in the case of an instance termination/restart.

To ensure that the servers are healthy, there are two health checks. The first health check is the standard EC2 health check built-in to AWS, which monitors for system-level errors with hardware or software. The second set of health checks are custom-made, one for the aggregation server and the other for the FTP server. These health checks are triggered as cron jobs.

The aggregation server health check is a basic connectivity check to make sure the aggregation server is able to connect to all the endpoints. The FTP server health check connects to the FTP server and retrieves a directory listing to ensure there are at least n files, typically 1. If a server fails one of the custom health checks, the server is marked “unhealthy”, and is then taken offline by the auto-scaling group, and a new instance brought online.

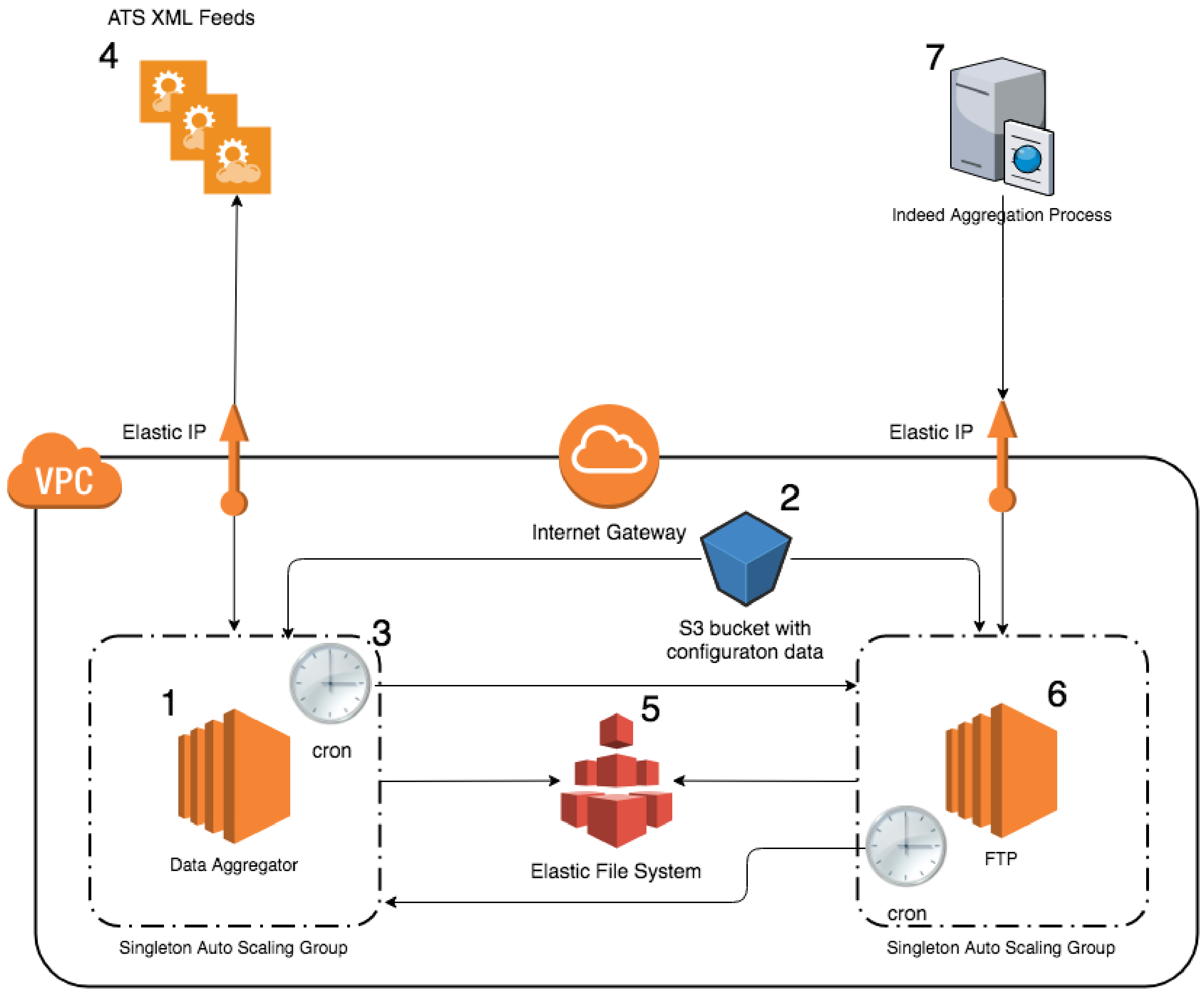
The retrieval/aggregation of the endpoints is triggered by a cron job that simply triggers the GetJobs endpoint on the aggregation server. The aggregator knows that when this URL is invoked, it should iterate through all the endpoints to get the jobs data and save it to a temp file. After all the endpoints are harvested, the temp files are concatenated in to a single file, and the resulting file is date-time stamped with the date-time of completion (in milliseconds).

All cron jobs, including the health checks as well as the GetJobs initiator, are run on both the aggregator as well as FTP servers. This is for redundancy. The are staggered so as to not execute at the exact time, For example, the health checks run every 10 minutes, but are offset by 5 minutes on the other server. In this way, the health checks run every 5 minutes. Similarly, the GetJobs initiator is run every two hours, with a one hour offset on the other server. The result is that the jobs data is aggregated hourly.

Since the FTP server mounts the same data partition as the data aggregator, there is no need to copy or move the files to make them available to the FTP server.

Architecture Diagram

This architecture diagram illustrates the major components of the system:



1. The Data Aggregator - This is a lightweight server written in Java that uses Jetty to create the GetJobs endpoint as well as the HealthCheck endpoint. The GetJobs endpoint will read an endpoint configuration file that contains connection information for all the endpoints, as well as extra data like a comma-delimited list of email address for error notifications. Based on the endpoint configuration file, it will retrieve data from all the endpoints and concatenate it to a single XML file, after which it runs the file through a prettyprint utility to make it sure it’s properly formatted.

2. S3 bucket - This S3 bucket contains all of the resources that are copied to the aggregator and FTP server when they are first instantiated. It includes: API configuration

file, cron configuration for the ftp server, cron configuration for the aggregator, jar file with FTP and aggregator health checks, prettyprint.py source code and requirements.txt, jar file with the JobAggregator server, and the vsftpd configuration file.

3. Cron jobs: These include the aggregator health check, the FTP server health check, and the GetJobs initiator. All of the cron jobs are run on both the aggregator as well as the FTP server, with a time offset so that the same jobs are not run at the same time.
4. ATS XML feeds - These are the endpoints from which jobs data is retrieved.
5. Elastic File System - This is the files system that is mounted by both the data aggregator as well as the FTP server. Being elastic, it grows and shrink based on usage. Since both machines mount the same file system, there is no need for the aggregator to send the jobs XML data to FTP server in order to make it available.
6. The FTP server - this is the industry standard vsftpd program for FTP.
7. Indeed's aggregation operations.

Operational Costs

There are two components to the operations costs: 1. Computing Resources, and 2. Data Transfer.

The cost for computing resources, including the EC2 instances and Elastic File System, can be as little as $10/month. If there are many API endpoints, it may be necessary to upgrade the aggregator instance type. The FTP instance type can be a t2.micro, possibly t2.nano, regardless of the size of the jobs XML file.

The cost for data transfer is directly proportional to the size of the jobs XML file. AWS only charges for data egress, which is $.09/GB. As an example, assume your jobs XML file is 1GB. Indeed will pull this file via the FTP server 4x/day. The monthly data cost would be about $11.

In total, you can expect this system to cost as little as $20/month, depending on the size of your jobs XML. Even if the jobs XML file was 3GB, total operational costs would only be around $40/month.

Next Steps

We are currently creating resources to make the deployment of this solution as painless as possible. The end goal is to provide sufficient resources such that the ATS need only implement a couple of classes in Java that do the actual job data retrieval from a given endpoint, at which point the entire infrastructure can be deployed via CloudFormation template. Ideally, the entire process from start to finish should take an ATS engineer no more than two days to implement.

The technical resources, in the form of source code, documentation, CloudFormation template, etc, are available on GitHub at https://github.com/indeedalliances. The project that will painlessly walk you through a demo deployment of the reference architecture is located at https://github.com/indeedalliances/FeedAggregationReferenceArchitecture.