Project Solution - Final

Monday, November 12th, 2018

Group 12

Christian Salcedo
Amber Usry
Clay Fulcher
Jacob Walch

Table of Contents

Abstract

Reports from the company's remote warehouse security system find that crucial points around the warehouse are vulnerable due to fluctuating battery levels. The company has tasked its in-house software team to write a data reader that reports said battery levels which are received hourly by the security vendor. The data will be received as CSV files and processed through the software team's automated reporting system which will filter and pinpoint vulnerable spots that are below the 5.0 voltage battery level in the final file format. This project report will detail all that is and was needed to complete the order.

Feasibility & Cost

Necessity

The need to complete a reporting program for security purposes pertaining to the status of battery checkpoints is paramount to ensure protection of the company's assets within the remote warehouse. Future ensurement of warehouse assets will secure its market value with customers and rake in future profit. On a social standpoint it is crucial to guarantee security with customers that buy assets from the warehouse and protect the company from any legal ramifications that may come due to loose security in faulty battery levels across the warehouse. The completion of this project is absolutely necessary.

Financing

Considering the financial aspect of the project, we have elected to go with open source software. Open source platforms meaning the software will cost us nothing to procure and operate, aside from company time and current resources including in-house hardware. The software we plan to utilize includes the following: Pandas, PuTTY, PuTTYgen, WinSCP, Google Drive, Notepad++, and Red Hat Enterprise Linux (RHEL7) as our OS on the EC2 instance.

EC2 is free for most tiers, and taking into consideration the low amount of data this project will encompass*, the team doesn't have an expectation of being charged for Amazon's services. The threshold for an EC2 instance is 750 hours of usage per month during the free tier period, and we do not anticipate exceeding this given that we will only be running one instance at a time totaling a maximum of 744 running hours**.

Current company hardware will be used including two seperate labs that contain Python, PuTTY, PuTTYgen, and WinSCP installed on the computers. No additional hardware will need to be purchased for the completion of this project.

During the completion of this project, Pandas implementation became too high of a risk to continue with implementation plans as it caused server crashes on multiple ec2 instances. Overall it was unnecessary to parse out such small CSV files and a secondary solution of parsing within the custom in-house built script was implemented using basic Python statements such as FOR and IF loops. This added an additional cost of time.

*(<1KB of data per file transfer * 24 transfers per day * a maximum month of 31 days = at most 744 KB transferred per month, well within the 8GB maximum of AWS free tier limit)
**(24 hrs*31 days)

Technical

Due to all hardware being in-house, there will be no need to go further than what is available in the company's facilities. All software including Pandas which is a powerful Python library used to auto parse the csv files, PuTTY to connect to our instance, PuTTYgen to generate a ppk file for a secure connection, WinSCP for file transfer, Google Drive for file storage, Notepad++ as a code editor, and AWS Free tier EC2 instance to provision a server. The software team will be developing a custom, in-house Python script for the necessary file transfer; it is within the team's expertise to create it. Overall, by technical means, a solution is viable.

Social and Cultural Viability

Regarding the cultural and social viability of the project, we recognize that after the application is constructed and is fully online, there will need to be appropriate documentation provided for posterity. Because our team will not always be around to operate and maintain the system, we can produce said documentation as it pertains to each level of the system's operation. This process will be completed simultaneously with our project's construction and eventual implementation, so that it is accurate specifically to the current build that will eventually take shape.

Sizing & Scope

Hardware

As it relates to sizing, we have access to two computer labs with multiple desktop setups running Windows 10 to complete our work. Because a majority of the work will be completed utilizing cloud services, we do not have to worry about infrastructure within these labs such as creating physical and logical diagrams.

Software

One of the software team members will offer their company email to make an account with AWS. This will include credit card information as mandatory by the AWS contract and to ensure that even if the program will be running under the free tier threshold, payments can be made if the instance goes over the free tier limit.

Time/Personnel

The team will consist of 4 individuals: Christian Salcedo (Programmer), Amber Usry (OS Implementation), Jake Walch (Technical Writer), and Jordan "Clay" Fulcher (OS Backup Implementation/Technical Writer). The timetable for Go-live will approximately take one month.

Capacity

It is estimated that throughout a single year (this includes leap years) there will be a total of 8,784 requests, all for files that are recorded hourly each under 1 kb in size.

366 days (leap year) * 24 requests (hourly) = 8,784 rpy (requests per year)

We are collecting 24 of these CSV files per day, seven days a week, meaning that our total weekly data collection comes out to 168 kb. Each hourly file should not exceed 1kb, most especially when the stored file will parse out all good battery readings.

Pricing

The testing phase for this project solution will run a week using the AWS EC2 free tier and have determined that our cost for procurement and use of their cloud services will cost us $0.00 for the first year. Once all the bugs have been confirmed and the team knows for sure that maintenance costs will not be an issue, plans with AWS for the following years will include a yearly flat fee of $59.00. These calculations were determined from using the AWS Simple Monthly Calculator.

Buy vs. Build

Recommendations

Since we are using AWS, this project will require a combination of the two, we will have very limited control over what customizations we can do for the project. We “buy” the hosting on AWS and we “buy” the services that they provide as we need them. We “build” the machine by choosing exactly what we want to go into the instance.

As far as power goes, we will be utilizing a mid-tier instance within AWS to power our service. We are utilizing Python to create a script that gets the file from the security company’s web endpoint and posts it to our end-user-interface to view battery levels.

Specifically, the program will be using the most updated version of Pip along with a request library available for Python to get the URL endpoint, store it in our database on AWS and, finally, post it to the second provided endpoint as well as the web app we are in the process of constructing.

On the front end-side, we will be utilizing a template that we procured two years ago for another project. This template makes use of node.js, HTML, CSS, as well as a variety of other languages to create an aesthetically pleasing web app that will display the information requested, as well as provide a 'history' functionality that the end-user can use to track battery life over a period of hours, days and weeks. It is our belief that this added functionality will give the client a greater insight into their battery life, allowing them to appropriately schedule changes and replacements as needed.

All plans for front-end development were dropped mid construction due to necessity of project requirements overshadowing unnecessary features that were not mission critical.

Size of Software

Our software will be minimal in size and will not exceed 1 gigabyte of storage on our AWS instance. This is because the data that we are getting, storing, and posting is a very small size and won't require a lot of processing or coding to get it to work in the way the project specifies.

In its entirety, our finished product will consist of a fully-functioning web app running on our AWS instance in the cloud. The server will be running through and maintained by Amazon, so our costs will be contained to what AWS charges us for this service. The web app will consist of the template we have already procured, combined with the Python script that will pull the information from the established endpoint, parse it, and then post it for consumption by our endpoint.

Again the front-end development was scrapped and AWS charges thus far have been $0.61 for the month of September and $3.71 for the month of October both accumulated due to the elastic computing cloud fees.

Project Plans & Scheduling

Labor

The project will take a month to complete from the start date of October 1st 2018. Due to the size of the project we will not need to hire outside help.

Our team consists of four IT professionals, all with different skill sets and methods of production. Leading the project will be Jake Walch, and he will also serve as a technical writer for all project documentation and post-mortem resources for posterity. Christian Salcedo will take the lead on back-end programming with Python, ensuring that the program works as designed and installing any needed modules within the Linux OS. Amber Usry will be assisting with the testing and quality assurance of the back-end programming, automation of the script as well as procurement and maintenance of the ec2 instance. Jordan "Clay" Fulcher will be the head of research for the entirety of the project and constructed the LAMP stack on the server.

While these roles are all firmly established in the team, our workflow allows for a more nuanced and mixed delegation of tasks, so that each member of the team is able to put some work into all facets of the end product.

Plans

Below we have included a list of project plans that each team member will be responsible for, as well as a brief description of why each of those plans is necessary.

- Walch
    - Software maintenance plan
        - Appropriate documentation must be developed and updated so that future operators of the system can correctly maintain it. Our software consists of a LAMP server, CSV library, request library, datetime (for UTC) and crontab and will be updated as needed.
    - Risk management plan
        - In the event something in the system fails, we must have a plan of action to address any and all emergencies that we can possibly identify. We identified problems that would appear during normal running of the instance and removed them to reduce the risk of the final product suffering in quality.
    - Infrastructure Maintenance Plan
        - We need a plan to make sure our servers, desktops, instance backups, and security are maintained on a weekly basis or as needed. We are using an automation process that needs to run at a certain time every day and we need to make sure that that happens.

- Salcedo
    - Software development plan
        - This plan will ensure that the software is developed with the ability to complete all functionality required. This will be a custom script with documentation noted throughout and tested thoroughly.
    - Deployment plan (software)
        - We are utilizing multiple third party software in conjunction with our proprietary scripts so we must plan to ensure all these elements work together so we have enough time for testing and quality assurance.
    - Procurement Plan
        - Amazon is free to procure and anything that may exceed the free tier limit will be . We also downloaded other free packages and libraries including as Python and Pandas.
- Usry
    - Quality assurance plan
        - We developed the code in such a way that errors with automation and the scripts themselves could be identified quickly, easily, and tested repeatedly, staying with our plan to test every hour on the hour in between code edits. This made the QA process run smoothly and efficiently.
    - Disaster recovery
        - This is necessary in case disaster strikes and it becomes necessary to start the project from virtual scraps. After we implemented updates (scripts, automation) we took images of the instance so that if anything were to go wrong (system failure, etc.) we would have builds to fall back on as well as documentation throughout.
    - Training Plan
        - After we were finished with the work, we assembled a training plan. Even though the process is automated, we will train on how we went through the files and parsed them as well on how we automated the uploading and downloading processes. This was done for future programmers and for the employees that will interact with this file on a daily basis to know how to access and read it so they can perform their job duties as well.

- Fulcher
    - Data migration plan
        - We must have a plan in place to ensure the data we are handling is migrated in the correct format to the correct place. We have an

automation process that moves data from place to place and it needs to run 24/7 and we need to be sure it is done correctly.

- Installation plan (hardware)
  - As we made use of mostly cloud services for our hardware, our installation plan involved only installing the initial LAMP server. This was necessary for the completion of the project and should never be overlooked.

Project Schedule

As this project is relatively small, we do not anticipate for it to be a long, drawn out process. However, we do want to make sure we spend adequate time on it. We plan to spend 1 month on this project to make sure any and all complications have been worked out.

We intend to build the majority of the functionality and aesthetic of the application early on, leaving room for documentation and best practices to be written and checked for quality. We expect this part of the process to go smoothly as we have scheduled for plenty of time for us to work out the kinks in the program and make adjustments as needed.

Again all front-end development was inevitably scrapped.

Schedule

Below is the tentative official project schedule (schedule to some adjustments).

	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
Oct 1	Completed getting data from endpoint	Initialize documentation on Python script			Completed research on PANDAS and complete initial testing on storing data	Georgia Southern vs. South Alabama	SCRUM weekly meeting 2208
Oct 7	Initial version of web application is up and running on AWS instance	~~Begin documentation of frontend~~	Completed work on data storage procedures	Continued documentation on Python script and backend processes	~~Begin integration of frontend and backend products~~		SCRUM weekly meeting 2208
Oct 14		Documentation on integration of services				NM State vs. Georgia Southern	SCRUM weekly meeting 2208
Oct 21	Begin testing for automated GET requests paired with web app			Finalize documentation of Python script	~~Finalize documentation on node.js frontend~~		SCRUM weekly meeting 2208
Oct 28	Finalize documentation on integration of product	Begin final testing and quality assurance	Testing/QA	Georgia Southern vs. App State	Testing/QA		Project complete meeting (post mortem)

Hardware & Software Procurement

For this project, we were fortunate to be able to use the resources in the IT Lab to complete our project. As such, we did not need to create and submit a formal RFP to get software in order to accomplish our goals. As such, there was no formal submission/approval process for us to complete as we are working autonomously as a team. With that said, we have quantified all Hardware and Software resources we

have used to complete this assignment and have organized them in a list here, organized by order of importance.

Hardware

- HP z240 Desktop Machine (IT Lab 2208) - LAMP Implementation
- Lenovo Thinkcentre M-Series Intel Core i7-4770 CPU - Script Testing and Automation
- AWS Cloud Infrastructure (server and storage space)
- 2013 Macbook Air (Jake's PC)
- Samsung T5 Portable SSD 250GBs

Software

- Windows 10 OS
- Mac OSX
- Google Chrome, Google Drive tools
- AWS Management Console
- PuTTYgen
- PuTTY
    - crontab
    - sudo yum/yum
    - pip/pip3
    - -- user
    - install
    - update/upgrade
- Notepad ++
- WinSCP
- Python 3.4 & Libraries:
    - Pandas (abandoned during implementation)
    - csv
    - requests
    - datetime
    - os.path (abandoned during development)

Custom Software

During the development process we identified that we would need to do some automation for our system to run without the need for human intervention. We created and tested the manual process for carrying out our objective and later decided to create a script in Python that would parse this process to fit the needs of the warehouse maintenance crew. In order to automate it, the Linux OS crontab capability was used to target the Python script.

The script that we initially developed gets the file from the endpoint and does nothing else. At first we were going to use pandas to parse the data to be posted; we ended up having to utilize a combination of crontab and Python statements to automate and parse the relevant data respectively.

Specifically, we utilized FOR/IF statements and the Append function to parse the files; request is used to GET the files from the endpoint as well as POST them; when the front-end was in development, we intended on using os.path to replace the current file with the most-current version. However, we scrapped the front-end system during the development process in order to focus on critical functions on the back-end.

Configuration Management

When we began our work, we started on Python 2.7. When we realized that Pandas would require Python 3.4 for it to work before we could begin using it, we downloaded it and updated our code accordingly; we stored correct responses and incorrect responses to help troubleshoot problems in the code.

At a certain point during testing, we were not receiving a response back, so we downloaded an example output file from the security vendor as well as our output files, and compared the two using Notepad++ to diagnose the error. We used the information we got out of that testing method to help us fix our code and get back on the right track.

As a policy within our group, we designated Christian and Amber as the arbiters of our configuration management. If any member of the team had a change they wanted to make to the code, either Amber or Christian had to sign off on it before it was implemented for testing. We did this as these members of our team were the most technically familiar with the code and could make educated judgements on what stood to be potentially broken by a particular change.

Version Control

Our code was stored on Google Drive so that all of us have access to the code at the same time. We worked primarily on our version that required Pandas to be implemented for our parsing; the other version requires manual parsing and did not make use of Pandas. After realizing that Pandas may not play nice with us after experiencing instance failures, Christian copied the version, creating a separate version that could make use of another method of parsing in the event that the first version did not pan out. We ultimately went with the standard Python parsing method as denoted above.

At first our plan was to utilize Pandas in our parsing process; however, our instance crashed three times during the installation of Pandas and had to be recreated from an image Amber took early on of the instance. These repeated failures illustrated to us that Pandas would not be viable for the completion of our objective.

Google Drive afforded us the ability to store multiple scripts to address this problem and work on them simultaneously. Because of this, we weren't relying on a single build of the project to function and could make adjustments to each version as we saw fit.

Data Migration

The migration plan involved the process of getting the file and parsing it to be posted. The CSV file is a comma delimited file, so we had to take that into account when we considered how to pull the data, parse it, and export it back out with the correct battery listings.

We needed to specify the timezone that the data was being pulled from, as this would affect the time stamp that it would ultimately be paired with. Due to the security vendor's server being located in a timezone that is four hours off from our server, the datetime Python library was imported to make use of the "*utcnow"* function which coincided with the timezone of the security vendor's server. This means that when we grabbed the latest file from the web endpoint, we would grab the latest readings instead of readings that occurred 4 hours ago.

In addition to delimiting the time, our plan for data migration included the parsing functionality of the CSV file. The files would be imported, the battery levels that had fallen below 5.0 would identified, and those values would be exported as a CSV file to the correct endpoint.

Verification and Testing

We first aimed for confirmation (through manual testing) that the code would be able to pull and post the CSV files to the correct locations. We manually ran the files through Python 3.4 and printed the results to check our code. After succeeding in that regard, we moved on to automation.

Testing

From the outset, we had planned on using the Pandas library to parse the files for us. We experienced numerous roadblocks including issues simply downloading it; once we successfully installed the library, it crashed our instance three times in succession.

After it was evident that using Pandas would continue to produce problems with our system, we abandoned it and manually all necessary parsing inside the python script.

Testing and quality assurance took place over the course of three days. The first took place on October 29th, and tests were executed at 1AM, 2AM and 3AM (UTC). At this point we had not completely figured out how to parse correctly, so our results were incomplete. The second set of tests occurred on October 30th, at 3AM, 9PM, 10PM, 11PM and 12AM (UTC). As well, these were executed manually because our automation process had not yet been finalized and at this point we have successfully parsed the file. Finally, at 2AM on October 31st, the system successfully began executing on its own and ran from that point forward for the remainder of the time.

Verification

The system would need to notify the operator(s) when a particular battery level fell below 5% and display those results so that the maintenance crew onsite could appropriately manage or replace batteries as needed.

Our system, through the use of Python scripts and AWS infrastructure, gets the files (battery levels) from the specified endpoint; the system checks these levels with regard to whether they have depleted below this threshold; the system posts the results of this check to an endpoint to be checked by the maintenance crew so that action can be taken accordingly.

By our metrics, we have verified that our product accomplishes desired goals of our customer with regard to what they requested. By allowing ourselves a few days to verify our code and ensure successful execution of our scripts, we were able to have the system fully automated about 24 hours prior to the go-live.

Maintenance Plan

For our maintenance plan, we have designated Amber as the primary monitor of the system. In the subsequent months to the go-live, she will ensure the data types, timestamps and exports of the system correspond with the desired outcomes of the remote warehouse team.

During testing, we made sure that it was running at least 24 hours prior to the deadline; no maintenance was scheduled for the first seven days of November in order to allow the system to run as intended, eliminating the concern of an interruption to business continuity.

We used the results.php link provided to us to periodically check our automated code to make sure everything was running smoothly. In order to minimize costs on AWS, we limited our count of instances running to one at a time in order to stay within the free-tier limitations. This ensured that our project did not exceed our budget projections that we created at the start of the project.

Risk Assessment

We used reliable applications including AWS, PuTTY, and WinSCP to help us minimize risk. As for Python, while 2.7 was reliable, we went ahead and updated to the newest stable version to reduce any problems we might have had. We wanted to be proactive in this endeavor so as to minimize risks before they happened as opposed to responding to them after they happened.

Pandas presented us with a much higher risk as it was identified early on that the library would sometimes prevent us from accessing our instance or cause it to crash entirely. We moved on to a much lower risk of parsing the CSV file in the code itself using basic statements and functions.

Throughout the development and implementation process, we kept detailed notes on what worked and what didn't so that we would have a reference for any problems we ran into or any issues that might pop up in the future. This included heavy commentary within the python script of what nearly each line did.

Overall, the risk involved with this project was low; we merely needed to check the battery levels to confirm whether they were depleted or not and did not deal with any direct risk. With this being said, we identified the potential risk of our system not functioning as intended; failure to correctly report battery levels for the remote warehouse team correctly and on time could result in an interruption to business continuity for our customers, which would affect our relationship with them financially and socially.

Disaster Recovery Plan

Many times throughout the process of testing, the Pandas library caused our instance to crash, or sometimes become unusable. Thankfully, we could just use a backup of our instance to start over and try to fix the issue. We would create a new version of the backup after major updates so that if a problem occurred after a major update, we could go back to that instance before we made the changes to see what happened.

We also used Google Drive as a form of disaster recovery for our code itself. Using our risk assessment notes, we developed an emergency plan to deal with any potential issues we may have encountered (Internet outage, AWS failure, corrupted files, etc.). At the outset of the project we created a shared folder and made it accessible to each of our email addresses so that we could each view and edit files at the same time.

As a further failsafe protocol, we migrated our files to Chris' personal hard drive at the end of every group session we conducted. This way, in the event of a catastrophic failure or if we were to lose access to our Google Drive, we would still have our build saved on this external drive.

Quality Assurance Records

As denoted above, we conducted three series of tests between October 29th and October 31st in order to verify the system worked as intended and could handle the file retrieval, data migration, data parsing, and posting required.

- October 29th at 1AM, 2AM and 3AM (UTC)
    - Process: This was done using the custom in-house python script. It was called to execute through the ipython command prompt. To do this the tester had to navigating the directory to where the script was stored locally within the computers files and enter the command “ipython filename”. The filename at this time was test1.py so the correlating command entered was “ipython test1.py”.
    - Result:The file was retrieved from the endpoint posted by the security vendor and posted to our company’s respective endpoint manually. However the document contained data migration errors, and was not able to post all information exactly how it was given.At this time our team was also attempting to post all data that was given by the security vendor instead of only posting the prevalent data, that being the battery levels that fell below 5.0
        - Corrections Made: Data migration was corrected to read the “id” numbers as a string so that they would appear as “001” instead of how they were currently being posted, as “1”. At the time, the PANDAS library was being utilized and later dropped.
            - Tester: Christian

- October 30th 3AM, 9PM, 10PM, 11PM and 12AM (UTC)
    - Process: This was completed using the custom in-house python script. It was called to execute through the ipython command prompt(manually). To do this the tester had to navigating the directory to where the script was stored locally within the computers files and enter the command “ipython filename”. The filename at this time was test2.py so the correlating command entered was “ipython test2.py”.
    - Result: The code was retrieved correctly and posted with previous data migration errors resolved.
        - Corrections Made: Automation of the script still needs to be completed.
            - Tester: Christian and Amber

- October 31st at 2AM (UTC
    - Process: In this testing phase, the PANDAS library was dropped as the parsing tool and instead a small for/if loop that split the lines, stored all the columns of the CSV files, set a parameter to store only strings under 5.0 within the split line containing the battery levels, and ultimately appended the columns to the split lines for a simple, cohesive CSV file that contained only the locations of batteries in danger. The custom in-house python script was uploaded to the ec2-instance using WinSCP. This testing was to ensure the successful implementation of automation through crontab. To make the crontab iterate at the rate desired ”crontab -e” was entered to open up the editor. To initiate the script hourly at 15 minutes past the hour using the python 3 and including the direct file path to the script that we wish to run : “15 * * * python3 home/ec2-user/test3.py”.
    - Result: The code correctly ran every hour after the 15th minute as intended.
        - Corrections Made: Script ran successfully and no corrections needed to be made.
            - Tester: Christian and Amber

Works Cited

Geitgey, Adam. "Quick Tip: The Easiest Way to Grab Data out of a Web Page in Python." *Medium*, Augmenting Humanity, 3 June 2017, medium.com/@ageitgey/quick-tip-the-easiest-way-to-grab-data-out-of-a-web-page-in-python-7153cecfca58.

"The Pandas Project¶." *Pandas: Powerful Python Data Analysis Toolkit - Pandas 0.22.0 Documentation*, pandas.pydata.org/about.html.

Planning a Programming Project." *Khan Academy*, Khan Academy, www.khanacademy.org/computing/computer-programming/programming/good-practices/a/planning-a-programming-project.

"Red Hat Customer Portal." *A Brief History of Cryptography - Red Hat Customer Portal*, access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/7.0_release_notes/index.

https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/7.0_release_notes/index

PuTTY. (2018, October 18). Retrieved from https://en.wikipedia.org/wiki/PuTTY

https://en.wikipedia.org/wiki/PuTTY

Pandas (software). (2018, November 06). Retrieved from https://en.wikipedia.org/wiki/Pandas_(software)

https://en.wikipedia.org/wiki/Pandas_(software)

Calculator.s3.amazonaws.com. (2018). *Amazon Web Services Simple Monthly Calculator*. [online] Available at: https://calculator.s3.amazonaws.com/index.html [Accessed 12 Nov. 2018].

https://calculator.s3.amazonaws.com/index.html

AWS Service Terms. (n.d.). Retrieved from https://aws.amazon.com/service-terms/

https://aws.amazon.com/service-terms/