**Final Document**

CIS 4911 - Senior Project (U01)

Event Driven Cloud Computing

Cory McAn

April 24, 2015

Jason Dettbarn

# 

# Copyright and Trademark Notices

All Addigy specific logos and slogan material is copyright Addigy, Inc. All other materials developed for this project are available open source under MIT License:

<http://opensource.org/licenses/MIT>

# Abstract

This document will help to outline and explain all of the functional components for Event Driven Cloud Computing. The document will be broken down into different chapters each of which will explain different aspects of the project.

The first chapter will introduce the problem while providing the scope of the system, as well as any definitions that will be used throughout the document. The second chapter talks the limitations of the current system and provide alternative solutions to the problem. The third chapter talks about Project organization and development of tasks and milestones. The fourth chapter talks about the requirements that the new system will need in order to solve the problem. The fifth chapter talks about the design of the new system and any subsystems that are needed in order to create the whole system. The sixth chapter talks about the design of the system in detail while focusing on the system models. Finally chapter seven talks about the system validation tests that were performed on the new system.

# 

[1. Introduction](#h.vloxe4snkv8d)

[1.1 Problem Definition](#h.xamyk1sa2frx)

[1.2 Scope of System](#h.xsggis6ax9na)

[1.3 Overall Development Methodology](#h.21cm2j9j4mnf)

[1.4 Terminology](#h.6v0kcvdcffo)

[1.5 Overview of The Document](#h.u4srmhpvqtu6)

[2. Feasibility Study](#h.9bma6ypkcecv)

[2.1 Description of Current System](#h.vbcn4qgnn9r1)

[2.2 Description of Alternative Solutions](#h.jx7e4t7m54cv)

[2.3 Recommendation](#h.qshsam552hfq)

[3. Project Plan](#h.qun6kp8m45i5)

[3.1 Project Organization](#h.9fnho69aojyb)

[3.1.1 Project Personnel](#h.xim3m6r90e0l)

[3.1.2 Hardware and Software Resources](#h.mj5ijln47lee)

[3.2 Identification of Tasks, Milestones, and Deliverables](#h.w33zqynytt87)

[3.3 Cost of The Project](#h.20t9h1w5s7lr)

[4. System Requirements](#h.z9m8cj7bsdto)

[4.1 Functional and Nonfunctional Requirements](#h.8rjzdvrzrpyx)

[4.2 Requirements Analysis](#h.lppxo95xh6mg)

[4.2.1. Scenarios](#h.hsf7v0f2xcz1)

[4.2.2. Use Case Model](#h.gs2sbx274odq)

[4.2.3. Static Model](#h.8pc3uwix4tn9)

[4.2.4. Dynamic Model](#h.mo31nla0422b)

[5. System Design](#h.xcan711lp35n)

[5.1 Overview](#h.xl0rpoe0qvod)

[5.2 Subsystem Decomposition](#h.90r1qg8nmnfu)

[5.3 Hardware and Software Mapping](#h.7of0ffw7fejz)

[5.4 Persistent Data Management](#h.82t3tcb208at)

[5.5 Security/Privacy](#h.7h3f67lpaerm)

[6. Detailed Design](#h.9kmlbh37m0gn)

[6.1 Overview](#h.ud2t7j8yi7ft)

[6.2 Static Model](#h.o2rt6stmk8jg)

[6.3 Dynamic Model](#h.xh64ohaavxuh)

[6.4 Code Specification](#h.jrsr94huemw0)

[7. System Validation](#h.f35gia8qxd0t)

[8. Glossary](#h.912xodnsah2y)

[9. Appendix](#h.ss4pjnfiy0d7)

[9.1 Appendix A - Project Schedule](#h.w9r1h4yhfx9g)

[9.2 Appendix B - All Use Cases](#h.of362didayyz)

[9.3 Appendix C - User Interface Design](#h.4se1u0mvch6j)

[9.4 Appendix D - Analysis Models](#h.ampnq1mjsf1v)

[9.5 Appendix E - Design Models](#h.fa3dm9mj98t2)

[9.6 Appendix F - Documented Class interface](#h.1rg3i6lemahy)

[9.7 Appendix G - Diary of Meeting and Task for Entire Semester](#h.ktc5kyvllelj)

# **1. Introduction**

This document will serve as the final document which will outline the complete development cycle from start to finish. Contained in this document is all of the information collected and developed for Event Driven Cloud Computing, an IT monitoring and alerting system that sends real – time alert messages to IT Admins, and without requiring servers.

The following is the information you will be able find in this document: feasibility study done on the benefits of creating a new dashboard from scratch, the project plan aimed to keep the project within a 2 month timeline, the system requirements that needed to be met when creating the backend audit processing, the system design which shows all of the major subsystems in the implementation, more in depth design details showing the class interfaces for each subsystem, and finally the steps taken to validate the whole system.

## **1.1 Problem Definition**

In the world of IT, many professionals lack the tools necessary to manage Mac machines and respond to alert scenarios on their client machines. This project aims to accomplish this in an efficient and cost effective manner, using cutting edge technology that is far superior to the typical server implementation.

Currently audit data is processed by a remote server using polling to get updated information for machines. This implementation allows audit data to simply be pushed on demand and processed, without any need for running servers, and also able to process many audit files at once.

## **1.2 Scope of System**

The scope of the system for this is focused on the backend functionality of audit processing in the S3 bucket using event triggered Lambda Functions. There is also a need for a dedicated console window which can display the alert messages output by the lambda functions that are configured for alerting.

## **1.3 Overall Development Methodology**

The overall development methodology that was approached in this project was an agile and flexible development cycle. The primary features of the project were able to be completed thanks to this approach.

## **1.4 Terminology**

* AWS (Amazon Web Services) – The cloud infrastructure where client computer audit data will be stored.
* Audit Data – Data from client computers compiled using the client side application for Addigy that contains information related to hardware and software configurations on the machine.
* S3 Bucket – The storage that AWS uses for all files. To be used to house the client audit data.
* Lambda Function – Code written in Node.js that runs when AWS recognizes a specific event. Many functions can run simultaneously and will live until they have completed execution.
* JSON – File format that all client audit data will be stored in. Data is encapsulated for easy access using brackets to group related data.

## **1.5 Overview of the Document**

The following is the information you will be able find in this document:

* Feasibility study done on the benefits of a backend audit processing service
* The project plan that describe the development lifecycle
* The system requirements that needed to be met when creating the new dashboard
* The system design which shows all of the major subsystems in the new implementation
* More in depth design details showing the class interfaces for each subsystem
* The steps taken to validate the whole system.

# **2. Feasibility Study**

## **2.1 Description of Current System**

Currently the system has a web dashboard that displays all relevant computer information in a given company. There are multiple views available on the page such as the company view which shows an overview of all computers in the company and statistics regarding processor, memory and disk usage. There is also the individual view for each machine which shows this information in reference to that machine.

Some of the current limitations of the system are:

* reliance on a remote server for pulling machine data
* updates to the dashboard are through Javascript events on the page, so machine information will only update when the page is open
* current machine image is saved on remote server, which if there was enough load, could fill the disk on the server

## **2.2 Description of Alternative Solutions**

* Remote Server: Currently the system uses a remote server for storage and pulling of data for company computers. As mentioned above, this would cause problems if the server receives too many requests. This alternative would also be more costly than a cloud implementation.
* Creation of a Messaging Service: Rather than using PubNub for event notifications, we could build our own event messaging service. However that would require additional time and resources, greatly extending the scope of the project.

## **2.3 Recommendation**

Based on this analysis, the team recommends using AWS with Lambda event triggers and an S3 bucket to do the computation for the service in the cloud. The messaging service can be replaced with one of the many free services that are currently available, in our case we are using PubNub for alert messaging.

# **3. Project Plan**

This project is to be organized and developed using a variety of different tools gears towards an agile development approach. By using an agile development approach, it ensures that the project is constantly being updated and available to deploy at any time during its life cycle.

## **3.1 Project Organization**

The project is to be split up and organized based, again, on an agile development approach. The following series of steps will be taken in order to make sure that the development lifecycle is productive with very little downtime.

1. **Scope**
   1. Meet with client and discuss all of the functional requirements, non-functional requirements, and end goals for the project
   2. Define any additional resources needed for the completion of the project
   3. Obtain all of the resources necessary to realize the project
2. **Analysis**
   1. Define any software and hardware specifications needed in order to develop the product taking into account the functional requirements that need to be realized.
   2. Meet with development team to discuss the possibility of the requirements
   3. Discuss the financial feasibility for the project with the client
   4. Meet with the development team to organize and distribute the requirements needed based on individual strengths and weaknesses.
   5. Develop the project’s delivery timeline
   6. Obtain approval to proceed with the project
   7. Provide the tools necessary to tackle the project using agile development principles.
3. **Design**
   1. Review any software needed to complete the application.
   2. Develop functional specifications and prototypes based on functional specifications
   3. Review prototypes with client and record feedback
   4. Incorporate any feedback while trying to avoid any scope creep
   5. Obtain approval for completion

## **3.1.1 Project Personnel**

Cory McAn is the sole developer, tester and documentation creator.

My Product Owner is Jason Dettbarn of Addigy.

## **3.1.2 Hardware and Software Resources**

The resources we will be using for this project are:

* AWS
* Lambda
* S3
* PubNub
* Linux VMs for Demos
* HTML, CSS and Javascript for Demo
* Node.js code for Lambda Functions
* Windows/Mac/Linux for development

## **3.2 Identification of Tasks, Milestones, and Deliverables**

Based on our User Stories, we are able to determine some discrete tasks:

Tasks:

* List the individual audit files in the S3 bucket
* read individual files using AWS GET
* store the audit data in a consolidated file, such that all audit files in bucket are in consolidated
* parse consolidated data to see old alerts and compare
* update alerts with audit data change on new data
* use PubNub to send alerts to a console for the admin to view
* create a configuration file that can modify alert thresholds, variables, and whether to alert on an alert type
* add antivirus log quarantines to the possible alerts
* modify consolidation and alerting code to be able to handle multiple simultaneous audit alerts
* create dedicated HTML page for viewing alerts in real-time

We feel that the following events will be milestones in our project:

* Lambda triggers for S3 bucket event related to new data
* Consolidating all audit files in order to see previous audit state
* Alerting on Battery, Harddisk, Firewall and Antivirus statuses
* PubNub messages to notify of changed data
* HTML page updates with new alerts for demo

At the end of the semester we will be delivering the following features:

* Completed consolidation and audit alerting code
* Cloud based event triggers to avoid remote server
* PubNub notifications go to authorized users when data updates/thresholds are reached
* Display alerts in a dedicated HTML page
* Alerts on thresholds defined in config file that resides in bucket
* Alerts related to Battery, Harddisk, Firewall and Antivirus statuses

## **3.3 Cost of The Project**

|  |  |
| --- | --- |
| **System** | **Cost** |
| PubNub | FREE |
| AWS S3 | FREE |
| Lambda | FREE to try, charged based on usage of functions, costs in References |
| Remote Server | FREE (Already Built) |
| Creation of Messaging Service | Expensive, cost would include server and time |

# **4. System Requirements**

The system proposed will run in the cloud using Amazon Web Services, allowing for greater computational speed and load. In this section we will detail the specific requirements of the system and use diagrams to illustrate these requirements.

## **4.1 Functional and Nonfunctional Requirements**

1. **Functional Requirements**

* The system shall run in the cloud using AWS, S3, and Lambda.
* The system shall consolidate all audit files per organization into one file.
* The system shall notify IT Admins when certain thresholds are reached using PubNub.
* The system shall only alert IT Admins on NEW alerts, by checking the previous alert status.
* The system shall have a backend configuration file that can modify how alerts are searched for in the audit data and turn alerts on/off.
* The system shall have a front end display of real time alerts for demo, possibly to be integrated.

1. **Non-functional requirements:**

* The system shall use HTML, Node.js, CSS, and JavaScript.
* The system shall be able to interpret JSON data, the format of audits.

## **4.2 Requirements Analysis**

In this section, we will look at the different fictional scenarios that could occur while using the Addigy Web Dashboard. Then we will examine the Use Case Model of the dashboard. Lastly, we will go over the static and dynamic models of the software.

## **4.2.1. Scenarios**

1. Individual computer user makes change on their local machine. Data is pushed to S3 bucket via Facter running on local machine. S3 bucket runs Lambda function to consolidate all audit files for that organization.
2. IT Admin runs audits for an organization. Many audit files are pushed into S3 simultaneously. The IT Admin can then use the Dedicated Console to view the alerts related to those audits.
3. IT Admin wants to specify how alerting is done for a given organization. He navigates to the S3 Bucket for that organization and opens the \_config.json file. He is able to change the values of the harddisk alert so that he only receives alerts from this organization when any organization audits have a hard disk usage percentage above 85%. He runs audits for the organization and only sees hard disk alerts for usage >= 85% in the Dedicated Console. This applies to the other types of alerts as well.

## **4.2.2. Use Case Model**

Refer to **Appendix A** and **Appendix B**.

The use cases show the flow of events and actors involved in each scenario. Typically the IT Admin starts the auditing process and the individual client’s machine data is put into the S3 bucket, where Lambda functions are triggered and execute.

**4.2.3 Static Model**

Refer to **Appendix C**.

The static model shows the different classes in the Event Driven Cloud Computing system, their variables and methods, and the relationship between them.

# **4.2.4. Dynamic Model**

Refer to **Appendix D**.

The dynamic model shows the sequence diagrams for Event Driven Cloud Computing. It shows the entire auditing process in each use case and the methods that are used to accomplish them.

# **5. System Design**

This section explains the detailed breakdown of the system into its components.

## **5.1 Overview**

Event Driven Cloud Computing is a back end audit processing service that contains many different systems in the Lambda functions that process the audits. It follows a three-tier architecture that separates the Presentation, Logic and Data Tiers, shown in the System Design diagram in **Appendix C**.

The presentation tier consists of:

* PubNub Console displays the alert messages on the developer console for the initial PubNub implementation.
* Dedicated Console displays the alert messages on a dedicated front end demo page that color codes the alert message output based on the name of the machine posting the alerts.

The logic tier consists of the primary Lambda code that processes the audits when triggered by an audit put event. It is composed of:

* ConsolidateAuditFiles which puts all of the current audit data in the bucket into one file so that we can maintain the previous audit state.
* TriggerOnThreshold is the original alerting function that has alerts for Harddisk and Firewall status.
* TriggerOnNewThreshold is the expanded functionality of the previous function that now checks the previous audit state to alert only on NEW alerts.
* SendPubNubAlert triggers alerts on Harddisk and Firewall, and then sends those messages to the PubNub developer console.
* ConfigAlerts has the same functionality as above but now with the added ability to modify how alerts work. The \_config.json file allows the backend admin to change the search variables, threshold values, and whether or not to send an alert on a given static.
* TriggerOnBatteryThreshold has the previous functionalities and also alerts on battery health and percentage.
* AlertAntivirusLog has the previous functionalities and also alerts on Antivirus quarantine status.
* MultipleRunTester has the previous functionalities and has been tweaked to work with up to 11 test audits being uploaded at once in a mass upload.

The data tier consists of the S3 Buckets and the Addigy API, which are already created and needed no modification or design. The S3 buckets hold the audit information as it is pushed in to be processed, and the Addigy API is the service that facilitates the push of the audit data into S3.

## **5.2 Subsystem Decomposition**

The **PubNub developer console** was already created and did not need any modification.

The **Dedicated console** was created in HTML and CSS, with a javascript portion that processes the PubNub messages coming in on the specified channel and color codes the output based on hostname.

All of the **Lambda functions** that make up the Logic Tier were written in Node.js, as at the time of development, AWS Lambda required Node.js code for Lambda Functions. Each of these functions accomplishes a set of tasks needed to process the audit files and give the IT Admin some meaningful output regarding the alerts.

**PubNub** is a messaging service that allows online communication using a Publish-Subscribe architecture. In our case, we only require the publishing aspect in the Lambda code, and in the Dedicated Console we subscribe to the publish channel to view the messages. By subscribing to a channel, a machine connects a TCP socket to PubNub through which it receives any messages published to the channel. The message will be delivered to all subscribers, and can have many subscriber channels open at once.

The **Addigy API** interacts with the client machine and runs the audit, at which point the audit is put into the S3 Bucket for that organization. The Addigy API was already built and I leveraged it to obtain the audit data for use in my Lambda Functions.

## **5.3 Hardware and Software Mapping**

The hardware and software that were used in this project were integrated in the following ways:

* **AWS Cloud Infrastructure**
* The S3 Bucket is configured to run code in response to certain events. In our case it is configured to run the specified Lambda Function on any “Put” of a file into the organization Bucket. This means that after a file has been uploaded, the Lambda function will run. This infrastructure serves as a ‘server-less server’ that can handle new jobs on demand whenever there is new audit data to process.
* **Server Machine**
* We use a linux virtual machine to host all of the project files, and to host the demo page. The server runs an Apache server. This server runs the Dedicated Console page when navigating to the server address.

## **5.4 Persistent Data Management**

This application currently does not handle any persistent data. All audit data is pushed into the buckets and can be managed by the back end IT Admin that has control of the S3 buckets. The PubNub Keys live inside of the configuration file in the bucket so that the admin can control which PubNub Channel they want to publish their alerts to.

## **5.5 Security/Privacy**

This application currently does not handle authentication as it is concerned with the backend processing of audit data. The IT Admins that would be accessing this data already have the credentials necessary to view it.

Data encryption and security is handled by AWS and that is why it was a prime candidate for this project.

The default Linux settings were used in the VM that was provided. No other modifications were made.

# 

# **6. Detailed Design**

This section explains the details of the subsystems that make up Event Driven Cloud Computing.

**6.1 Overview**

The Overall System uses a Model-View-Controller design with the Dedicated Console, S3 Buckets and Addigy API.

The **PubNub developer console** was already created and serves as a view.

The **Dedicated console** was created in HTML and CSS, with a javascript portion that processes the PubNub messages coming in on the specified channel and color codes the output based on hostname. This console serves as a view for the IT admin to see the alert messages.

All of the **Lambda functions** that make up the Logic Tier were written in Node.js, as at the time of development, AWS Lambda required Node.js code for Lambda Functions. Each of these functions accomplishes a set of tasks needed to process the audit files and give the IT Admin some meaningful output regarding the alerts. The functions that send messages require the PubNub API to publish to the desired channel. These functions are the logic that run in the S3 Bucket which can be considered the model in our architecture.

**PubNub** is a messaging service that allows online communication using a Publish-Subscribe architecture. In our case, we only require the publishing aspect in the Lambda code, and in the Dedicated Console we subscribe to the publish channel to view the messages. By subscribing to a channel, a machine connects a TCP socket to PubNub through which it receives any messages published to the channel. The message will be delivered to all subscribers, and can have many subscriber channels open at once.

The **Addigy API** interacts with the client machine and runs the audit, at which point the audit is put into the S3 Bucket for that organization. The Addigy API was already built and I leveraged it to obtain the audit data for use in my Lambda Functions. The Addigy API is the Controller that interacts with the Web Dashboard (previous semester project, integration was desired but we lost a team member) and the Individual Client Machine to get the audit pushed into the S3 Bucket.

**6.2 Static model**

## In this section I will describe the different parts of the system with subsystem specific class diagrams. Refer to **Appendix C** for the complete Class Diagram of the system.

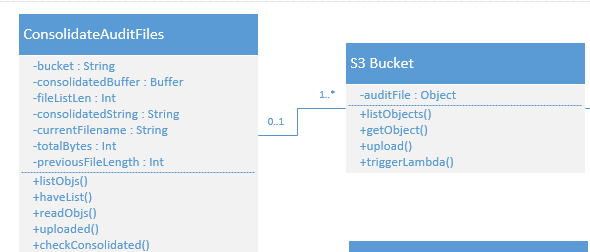
The S3 Bucket – ConsolidateAuditFiles Class Diagram shown in Appendix C shows the relationship between the S3 Bucket and the ConsolidateAuditFiles Lambda Function.

The S3 bucket maintains the event triggers that run the Lambda Functions, contains the auditFiles, and has all the functions for listing the files in the bucket, getting the files in the bucket, uploading files, and triggering the functions.

ConsolidateAuditFiles stores information related to the bucket and the consolidatedString (all of the audit files in the bucket stored in a JSON format for creating the new consolidated file). It uses the functions provided by S3 to list, get, read, and upload files in the bucket.

ConsolidateAuditFiles leverages the functionalities provided by the S3 bucket to create the consolidated audit file:

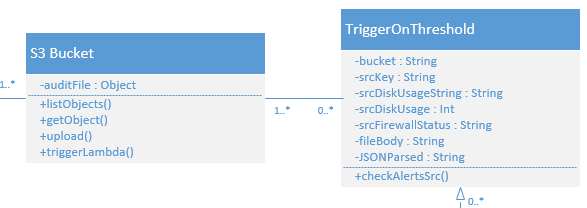
* listObjs: Lists all the files in the bucket, which allows us to get the list of the files in the bucket for traversal.
* haveList: Callback that runs after S3 successfully returns the list of objects to ensure we have it before moving on.
* readObjs: The function that actually reads the data in each file and appends them together to create the consolidated audit file.
* Uploaded: Callback that runs after successfully uploading the consolidated file to the S3 bucket. Prints a success message and kills the Lambda Function.
* checkConsolidated: function that runs at the beginning of the function execution that checks whether the file that started the Lambda Function execution is the consolidated file or an audit file. If it’s the consolidated we kill the execution to avoid wasted computation.



The S3 Bucket – TriggerOnThreshold Class Diagram shown below shows the relationship between the S3 Bucket and the TriggerOnThreshold Lambda Function.

TriggerOnThreshold maintains the srcKey which is the name of the file that was put and triggered the function, the srcDiskUsage which is the value of the diskUsage from the audit file, the srcFirewallStatus which is the value of the firewallStatus from the audit file, the fileBody which is the actual audit file text, and JSONParsed which is the audit file in JSON form to allow for searching of the JSON variables in the audit file. It supports the method:

* checkAlertsSrc: The alert checking function for the put file. In this iteration it checks for Harddisk and Firewall information in the audit file, handles errors, and stores the alert information for output in the console at the end of execution.

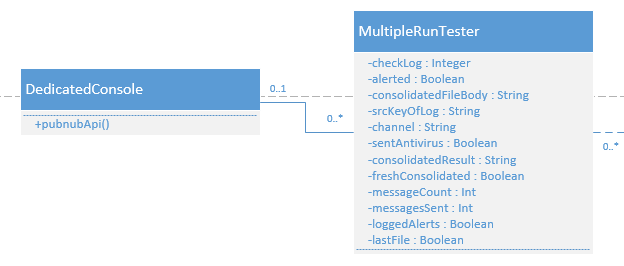


The Dedicated Console – MultipleRunTester Class Diagram shown below shows the relationship between the Demo Console view and the Lambda Function that handles batch uploads of audit files to S3.

The Dedicated Console subscribes to the channel specified and waits for messages on the channel. The page remains blank besides the top and bottom bars and the images until the messages flow in, and when they do it color codes the output of the messages based on the name of the computer for which the alerts are sent.

The MultipleRunTester has many considerations for handling batches of files. It requires many different checks to ensure computation time is not wasted, such as checking for antivirus logs, the configuration file, and the consolidated file on the initial run. If the Source File is any of those, we exit execution to save computation. The MultipleRunTester goes through every file in the bucket, first checks the configuration of the alerts. Then it checks for Antivirus, Firewall, Harddisk, Battery Health and Battery Status alerts, and sends those messages to the PubNub channel specified in the configuration file. Once all the messages for a given source file have been sent, we end execution. The last file (by name) in the bucket will consolidate all the audit files, and a timer is used to ensure that it happens after all the alerts have been sent.

This implementation could be improved with a small database that keeps track of which lambda function is the last to execute, and only on the last function to execute do we consolidate the files in the bucket. In future versions this will certainly be a desired feature.



**6.3 Dynamic model**

The dynamic functionalities in this project reside in the Lambda Functions, where there are many decision making pieces to process the audit files and prepare for alerting.

Addigy API

This controller drives the putting of audit files into the S3 bucket. It was unmodified during the project and serves as the starting point for audit processing.



Dedicated Console

The demo console utilizes the Javascript framework to intercept the PubNub Messages and display them in a cleanly formatted way.

The demo console remains blank until it receives messages on the PubNub channel, then it populates the empty space with the messages in div containers on the screen.



ConsolidateAuditFiles

This function is triggered on a file put event by the Addigy API Controller, and will consolidate all the audit files in the bucket when triggered. It uses the functions provided by S3 to list, get, read, and upload files in the bucket.



MultipleRunTester

This function is triggered on a file put event by the Addigy API Controller, handle batches of audit files on execution. It uses all of the other Lambda Functions in one streamlined function to send all the alert messages for the batch of audits to the PubNub channel specified in the configuration file.



## 

**6.4 Code Specification**

Dedicated Console

This view only has the logic associated with subscribing to the channel and stylizing the message boxes with color. Otherwise it is entirely CSS and HTML. The control object is the Javascript framework.

Addigy API

The details for the Addigy API are hidden from the user of this software, and as such there is no code specification for it. It is used only to provide the system with the input of audit files.

S3 Bucket

The inner workings of the S3 functionality are hidden from the user of this software, all operations on bucket contents are accessed through the APIs within S3. The class interface in the class diagram shows those functions that are used in this software. See **Appendix C**.

Lambda Functions

The Lambda Functions each have their own class interfaces as defined in the Class Diagram in **Appendix C**. Each Lambda function is pure code that runs on events, so these classes only exist during the execution of each function.

# **7. System Validation**

The system validation for the Event Driven Cloud Computing was done manually due to the nature of the audit files and the need to upload them to the S3 Bucket for testing. Due to this I created a folder called TestData that has the test batches of files for use with the MultiRunTester. I used an actual audit and edited the values to create each of the new audits for use in the batch tests.

**7.1. Subsystem Tests**

|  |  |
| --- | --- |
| Identifier | PubNubTest |
| Purpose | To ensure alert messages are received on the PubNub console |
| Setup | Configure publish keys in the Lambda code |
| Input | New Audit uploaded into S3 bucket with event trigger active and Lambda Function set |
| Expected Output | On PubNub Developer console, after subscribing to the channel, expect alert messages related to audit file that was uploaded to S3 |

|  |  |
| --- | --- |
| Identifier | PubNubAndConfigTest |
| Purpose | To ensure alert messages are received on the PubNub console |
| Setup | Configure publish keys in the config file |
| Input | New Audit uploaded into S3 bucket with event trigger active and Lambda Function set |
| Expected Output | On PubNub Developer console, after subscribing to the channel, expect alert messages related to audit file that was uploaded to S3 |

|  |  |
| --- | --- |
| Identifier | MultipleRunTesterTest |
| Purpose | To ensure alert messages are being sent to dedicated console for demo |
| Setup | Use same channel keys for publish/subscribe in config file/demo page |
| Input | New Audits uploaded into S3 bucket with event trigger active and Lambda Function set |
| Expected Output | On the demo page, color coded alert messages pop up as they come in |

**7.2. System Tests**

|  |  |
| --- | --- |
| Identifier | ConsolidateTest |
| Purpose | To ensure all audit files are consolidated in the bucket after uploading new audit files |
| Setup | Configure bucket event trigger to consolidation code |
| Input | New Audit uploaded into S3 bucket with event trigger active and Lambda Function set |
| Expected Output | Inside the S3 bucket where we uploaded the new file, we should see \_consolidated.json. We can open and check that the values match the files that are currently in the bucket. |

|  |  |
| --- | --- |
| Identifier | TriggerOnThresholdTest |
| Purpose | To ensure alert messages are being sent for Harddisk and Firewall alerts |
| Setup | Configure bucket event trigger to TriggerOnThreshold code |
| Input | New Audit uploaded into S3 bucket with event trigger active and Lambda Function set |
| Expected Output | In the CloudWatch logs of the lambda function, the alerts will be visible in the console output. |

|  |  |
| --- | --- |
| Identifier | TriggerOnNewThresholdTest |
| Purpose | To ensure alert messages are output for only NEW alerts |
| Setup | Configure bucket event trigger to TriggerOnNewThreshold code |
| Input | New Audit uploaded into S3 bucket with event trigger active and Lambda Function set |
| Expected Output | In the CloudWatch logs of the lambda function, the alerts will be visible in the console output. |

|  |  |
| --- | --- |
| Identifier | BatteryAlertTest |
| Purpose | To ensure alert messages are received on the PubNub console related to Battery Health and Status |
| Setup | Configure publish keys in config file, set bucket event trigger to TriggerNewPubNubConfigAndBatteryLambda code |
| Input | New Audit uploaded into S3 bucket with event trigger active and Lambda Function set |
| Expected Output | On PubNub Developer console, after subscribing to the channel, expect alert messages related to audit file that was uploaded to S3 with Battery Health and Battery Status messages |

|  |  |
| --- | --- |
| Identifier | AntivirusAlertTest |
| Purpose | To ensure antivirus alert messages are received on the PubNub console |
| Setup | Configure publish keys in the config file and set event trigger to alertAntivirusLogLambda |
| Input | New Audit uploaded into S3 bucket with event trigger active and Lambda Function set |
| Expected Output | On PubNub Developer console, after subscribing to the channel, expect alert messages related to audit file that was uploaded to S3 and any antivirus alerts related to the logs in the bucket |

**7.3. Evaluation of Tests**

|  |  |
| --- | --- |
| **Identifier** | **Result** |
| PubNubTest | PASS |
| PubNubAndConfigTest | PASS |
| MultipleRunTesterTest | PASS |
| ConsolidateTest | PASS |
| TriggerOnThresholdTest | PASS |
| TriggerOnNewThresholdTest | PASS |
| BatteryAlertTest | PASS |
| AntivirusAlertTest | PASS |

# 

# **8. Glossary**

**Audit:** Detailed machine hardware and software information that is collected via the Addigy software on each client machine.

**Facter**: Tool used by Addigy software on client machine that collects the audit data to be put into the S3 bucket for processing.

# 

# **9. Appendix**

## **9.1 Appendix A - Project Schedule**

(Gantt chart in Gantt.pdf, too large for document)

## **9.2 Appendix B - All Use Cases**

**Use case name**

ConsolidateAuditFiles

**Participating Actors**

* IT Admin
* Mac Addigy Client for Individual Computers

**Flow of Events**

1. IT Admin submits an audit request for a given client computer.
2. Request is processed by Addigy service.
3. Mac client runs the facter audit for the given machine.
4. Audit file is uploaded to the S3 Bucket.
5. If audit files for this organization have been consolidated, we are done.
6. If audit files have not been consolidated S3 triggers the lambda function ConsolidateAuditFiles.
7. ConsolidateAuditFiles returns the consolidated audit file into the S3 bucket.

**Exit Condition**

All files in the bucket are now consolidated into one file for archiving and use by TriggerOnThreshold.

**Exceptions**

Cannot read the list of objects, cannot read any individual files, cannot upload to the bucket.

**Functional Requirements**

Runs in the S3 bucket that calls the Lambda function.

All audit files in the current bucket will be uploaded to S3 into one file.

**Nonfunctional Requirements**

Each audit file will be separated by a new line character.

**Use case name**

TriggerOnHarddiskThreshold

**Participating Actors**

* IT Admin
* Mac Client

**Flow of Events**

1. IT Admin submits an audit request for a given client computer.
2. Request is processed by Addigy service.
3. Mac client runs the facter audit for the given machine.
4. Audit file is uploaded to the S3 Bucket.
5. S3 bucket triggers lambda function TriggerOnThreshold.
6. Function checks new audit file data for harddisk usage data token.
7. If above the desired threshold, create a console log of the alert.
8. If no threshold reached, log 'no alert'

**Exit Condition**

If threshold reached, log the alert. Otherwise, log ‘no alert’ to console.

**Exceptions**

Cannot read audit file. (Incorrect file format, must be JSON)

**Functional Requirements**

Runs in the S3 bucket that calls the Lambda function.

Prints alert data to console for viewing in S3.

**Nonfunctional Requirements**

None.

**Use case name**

TriggerOnNewThreshold

**Participating Actors**

* IT Admin
* Mac Client

**Flow of Events**

1. IT Admin submits an audit request for a given client computer.
2. Request is processed by Addigy service.
3. Mac client runs the facter audit for the given machine.
4. Audit file is uploaded to the S3 Bucket.
5. S3 bucket triggers lambda function TriggerOnNewThreshold.
6. Function checks new audit file data for harddisk usage token and stores it.
7. Function checks new audit file data for firewall status token and stores it.
8. Function checks old audit data in consolidated file for harddisk usage token and stores it.
9. Function checks old audit data in consolidated file for firewall status token and stores it.
10. If above the desired threshold and no alert existed previously, create a console log of the alert.
11. If firewall status is unchanged or there exists a previous firewall alert, log 'no new alert'.
12. If no harddisk threshold reached or there exists a previous harddisk alert, log ‘no new alert’.

**Exit Condition**

If threshold reached, log the alert. Otherwise, log ‘no alert’ to console.

**Exceptions**

Cannot read audit file. (Incorrect file format, must be JSON)

**Functional Requirements**

Runs in the S3 bucket that calls the Lambda function.

Prints alert data to console for viewing in S3.

**Nonfunctional Requirements**

None.

**Use case name**

SendPubNubAlert

**Participating Actors**

* IT Admin
* Mac Client

**Flow of Events**

1. IT Admin submits an audit request for a given client computer.
2. Request is processed by Addigy service.
3. Mac client runs the facter audit for the given machine.
4. Audit file is uploaded to the S3 Bucket.
5. S3 bucket triggers lambda function TriggerOnNewThreshold.
6. After lambda function returns with alert information, run SendPubNubAlert.
7. Function retrieves publisher and subscriber information.
8. Function parses alert data into message to be viewed by Admin.
9. Function subscribes to the PubNub channel.
10. Function sends message on the PubNub channel.
11. Channel forwards alert message to be viewed by Admin on WebDashboard.

**Exit Condition**

Alert successfully sent by function. PubNub channel displays message on console.

**Exceptions**

Alert cannot be sent.

Cannot connect to message channel.

**Functional Requirements**

Runs in the S3 bucket that calls the Lambda function.

Sends PubNub message on correct channel and received by Admin.

**Nonfunctional Requirements**

None.

**Use case name**

ConfigAlerts

**Participating Actors**

* IT Admin
* Mac Client

**Flow of Events**

1. IT Admin submits an audit request for a given client computer.
2. Request is processed by Addigy service.
3. Mac client runs the facter audit for the given machine.
4. Audit file is uploaded to the S3 Bucket.
5. S3 bucket triggers lambda function ConfigAlerts.
6. First function parses the PubNub alert configuration and gets the keys for publisher and subscriber, then it gets the “activated” value to determine if sending PubNub alerts is turned on.
7. Next function parses firewall alert configuration and gets the firewall name, value to alert on and activated value.
8. Next function parses harddrive alert configuration and gets harddrive name, value to alert on and activated value.
9. These flags are passed to triggerOnNewAlert and the alerts are logged to console and PubNub if active.

**Exit Condition**

Activated alerts show in console and in PubNub if active.

**Exceptions**

Alert cannot be sent.

Cannot connect to message channel.

Bad consolidated file, cannot be parsed.

**Functional Requirements**

Runs in the S3 bucket that calls the Lambda function.

Config.json sits in S3 bucket and determines alert behavior.

**Nonfunctional Requirements**

None.

**Use case name**

TriggerOnBatteryThreshold

**Participating Actors**

* IT Admin
* Mac Client

**Flow of Events**

1. IT Admin submits an audit request for a given client computer.
2. Request is processed by Addigy service.
3. Mac client runs the facter audit for the given machine.
4. Audit file is uploaded to the S3 Bucket.
5. S3 Bucket triggers lambda function triggerOnBatteryThreshold.
6. Function gets battery config from ConfigAlerts for values to alert on.
7. Function parses old audit file in consolidated for relevant variables.
8. Function parses new audit file for relevant variables.
9. Function calculates battery status based on variables in audit files.
10. Function gets battery health information from new audit.
11. All alert information is formatted for output.
12. Calls logAlerts function to output alerts to console and PubNub if activated.

**Exit Condition**

Antivirus alerts collected and ready to be sent or printed to console.

**Exceptions**

Bad consolidated file, cannot be parsed.

**Functional Requirements**

Runs in the S3 bucket that calls the Lambda function.

Battery Config will be in configuration file in bucket.

**Nonfunctional Requirements**

None.

**Use case name**

AlertAntivirusLog

**Participating Actors**

* IT Admin
* Mac Client

**Flow of Events**

1. IT Admin submits an audit request for a given client computer.
2. Request is processed by Addigy service.
3. Mac client runs the facter audit for the given machine.
4. Audit file is uploaded to the S3 Bucket.
5. S3 Bucket triggers lambda function AlertAntivirusLog.
6. Function checks config file for antivirus configuration and gets the search string to find in the antivirus logs.
7. Function checks antivirus log related to audit file that triggered AlertAntivirusLog.
8. Function parses log for all relevant lines, an example would be if search string was “quarantine” it would find all the lines that contain the word quarantine and format them for output.
9. Function passes alerts to triggerOnNewAlert to output alerts to console and PubNub if activated.

**Exit Condition**

Antivirus alerts collected and ready to be sent or printed to console.

**Exceptions**

Alert cannot be sent.

Cannot connect to message channel.

Bad consolidated file, cannot be parsed.

**Functional Requirements**

Runs in the S3 bucket that calls the Lambda function.

Antivirus log will have same name as audit file with “\_log” after the filename.

**Nonfunctional Requirements**

None.

**Use case name**

MultipleRunTester

**Participating Actors**

* IT Admin
* Mac Client

**Flow of Events**

1. IT Admin submits multiple audit requests simultaneously for a given organization’s computers.
2. Request is processed by Addigy service.
3. Mac client runs the facter audit for the given machines.
4. Audit files are uploaded to the S3 Bucket.
5. S3 Bucket triggers lambda function MultipleRunTester for each audit file put into bucket.
6. Function runs all functionalities from previous user stories on the audit files. Alerts will be stored related to whichever alerts are turned “on” in the config file by using the ConfigAlerts function.
7. Alerts are sent to the PubNub channel configured in the config file.
8. Alerts can be viewed in the PubNub channel to easily see all the alerts for the organization.

**Exit Condition**

Alerts for this organization are ready to be viewed in PubNub channel

**Exceptions**

Alert cannot be sent.

Cannot connect to message channel.

Upload error.

**Functional Requirements**

Runs in the S3 bucket that calls the Lambda function.

**Nonfunctional Requirements**

Messages must include hostname in output.

**Use case name**

DedicatedConsole

**Participating Actors**

* IT Admin
* Mac Client

**Flow of Events**

1. IT Admin submits multiple audit requests simultaneously for a given organization’s computers.
2. Request is processed by Addigy service.
3. Mac client runs the facter audit for the given machines.
4. Audit files are uploaded to the S3 Bucket.
5. S3 Bucket triggers lambda function MultipleRunTester for each audit file put into bucket.
6. Alerts are sent to the PubNub channel configured in the config file.
7. Alerts will be displayed in a dedicated console window on an HTML page that is linked to the PubNub channel in the config file, for easy viewing on demo day.

**Exit Condition**

Alerts for this organization are ready to be viewed in Dedicated Console which displays PubNub channel

**Exceptions**

Alert cannot be sent.

Cannot connect to message channel.

Upload error.

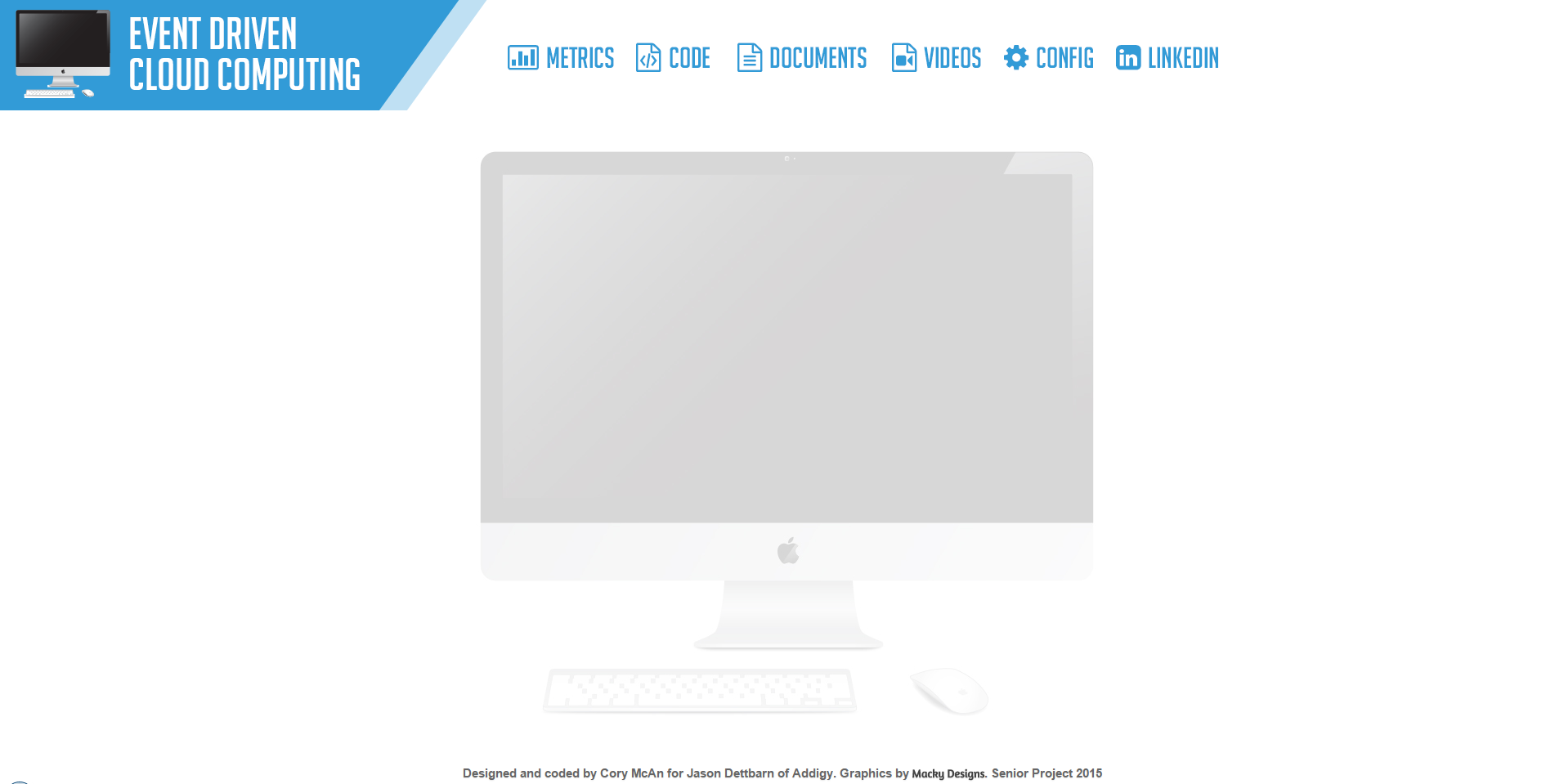
**Functional Requirements**

Runs in the S3 bucket that calls the Lambda function.

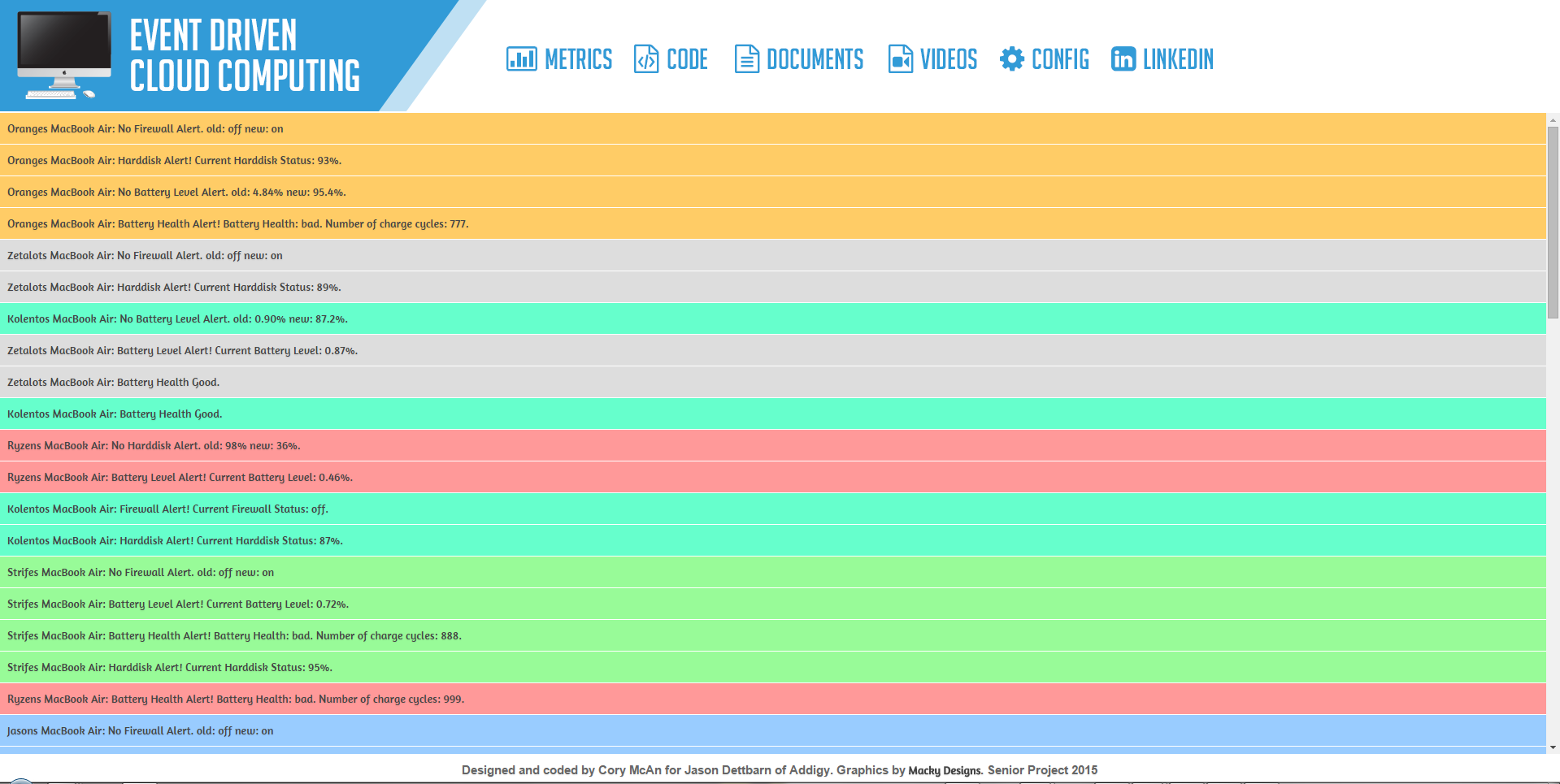
**Nonfunctional Requirements**

Messages must include hostname in output to identify which computer has the alert

## **9.3 Appendix C - User Interface Design**



Above: Before receiving any alerts.



Above: After receiving alert messages. Color coded based on hostname.

## 

## **9.4 Appendix D - Analysis Models**





**SEQUENCE DIAGRAMS**

1. **ConsolidateAuditFiles**



1. **TriggerOnThreshold**
2. **TriggerOnNewThreshold**



1. **SendPubNubAlert**



1. **ConfigAlerts**



1. **TriggerOnBatteryThreshol**
2. **AlertAntivirusLog**



1. **MultipleRunTester**



1. **DedicatedConsole**



## **9.5 Appendix E - Design Models**C:\Users\Cory\Desktop\posterImages\sysDesign.PNG

## 

## **9.6 Appendix F - Documented Class interface**

Addigy API

This controller drives the putting of audit files into the S3 bucket. It was unmodified during the project and serves as the starting point for audit processing.



Dedicated Console

The demo console utilizes the Javascript framework to intercept the PubNub Messages and display them in a cleanly formatted way.

The demo console remains blank until it receives messages on the PubNub channel, then it populates the empty space with the messages in div containers on the screen.



## **9.7 Appendix G - Diary of Meeting and Task for Entire Semester**

|  |  |
| --- | --- |
| Team Members: | Cory, Alex, Jason |
| Date: | Jan 20th 2015 |
| Discussions: | We will be building upon the previous web dashboard and adding/modifying functionalities to work with Amazon Web Services, specifically S3 and Lambda functions. We did a general overview of the technologies and discussed meeting times, typically Tuesday and Sunday. We also discussed potential roles regarding how the work would be split up, currently it appears that Cory will deal with backend and Alex will deal with front end. Google documents were set up for the Requirements Document and Feasibility Study & Project Plan. |

|  |  |
| --- | --- |
| Team Members: | Cory, Alex, Jason |
| Date: | Jan 27th 2015 |
| Discussions: | In person meetup, went over tasks for Sprint 1 in detail. Jason created an overview diagram to help us understand the structure of the framework. |

|  |  |
| --- | --- |
| Team Members: | Cory, Alex |
| Date: | Feb 2nd 2015 – Feb 13th 2015 |
| Discussions: | Daily Scrums, discussed what needed to be done and created our scrum cards. |

|  |  |
| --- | --- |
| Team Members: | Cory, Alex, Jason |
| Date: | Feb 17th 2015 |
| Discussions: | Sprint 1 review. Neither of us completed a user story, still having issues with the concepts and the new languages we needed to learn. Sprint 1 retrospectives. |

(At this point, Alex dropped the course, I did scrum cards on my own each day but they will not be listed here as they are not technically ‘meetings’)

|  |  |
| --- | --- |
| Team Members: | Cory, Jason |
| Date: | Feb 24th 2015 |
| Discussions: | In person meetup to go over Sprint 2 progress. Showed Jason what I have completed so far, asked questions regarding implementation. Got feedback, discussed future implementation possibilities. |

|  |  |
| --- | --- |
| Team Members: | Cory, Jason |
| Date: | Mar 3rd 2015 |
| Discussions: | In person meetup to discuss Sprint Presentation. More information regarding infrastructure and implementation. Retrospectives for Sprint 2. |

|  |  |
| --- | --- |
| Team Members: | Cory, Jason |
| Date: | Mar 17th 2015 |
| Discussions: | Online meetup to discuss Sprint 3 progress. Doing well, discussed impediments and results. More background regarding future sprints implementations. |

|  |  |
| --- | --- |
| Team Members: | Cory, Jason |
| Date: | Mar 24th 2015 |
| Discussions: | In person meetup to discuss Sprint 3 presentation. Discussed techniques for getting better scores in presentations, such as making a demo video. Discussed a better solution for IDE than the one I had been using. Got access to PubNub for next user stories. |

|  |  |
| --- | --- |
| Team Members: | Cory, Jason |
| Date: | Mar 31st 2015 |
| Discussions: | Online meetup to discuss Sprint 4 progress. Discuss impediments and potential solutions. Weighing the options against each other. Discuss what we need for demo day and Sprint 4 report. |

|  |  |
| --- | --- |
| Team Members: | Cory, Jason |
| Date: | April 7th 2015 |
| Discussions: | Online meetup to discuss Sprint 4 presentation. Jason happy with feedback and progress on user stories. Discuss in more detail demo day materials and figure out how to accomplish demo day dedicated console page. |

|  |  |
| --- | --- |
| Team Members: | Cory, Jason |
| Date: | April 14th 2015 |
| Discussions: | Online meeting to discuss Sprint 5. Currently things are going smoothly, demo is almost complete and we are up to speed. Discuss generally how demo day usually goes and plan for what to present at showcase. |

1. **References**

<http://aws.amazon.com/lambda/pricing/> for pricing information related to AWS and Lambda Functions.