OmegaGraf

by

Jacob Colvin, Matthew Currie & Dylan Owen

Submitted to
The Faculty of the School of Information Technology
In Partial Fulfillment of the Requirements for
The Degree of Bachelor of Science
In Information Technology

Jacob Colvin 4/13/2020
Date

Matthew Currie 4/13/2020
Date

Dylan Owen 4/13/2020
Date

Tony Iacobelli 4/27/2020
Tony Iacobelli, Faculty Advisor Date

University of Cincinnati
College of
Education, Criminal Justice, & Human Services
April 2020

Table of Contents

Illustrations

Tables

Figures

Abstract

According to the IEEE, agile methodologies have driven organizations to alter their infrastructure at an ever-increasing rate. Thus, it is important to also accelerate the deployment times for external monitoring systems to provide fast and continuous feedback to security, development and infrastructure teams. VMware's vCenter is a product that, according to VMware's hardening guides, requires external monitoring from the moment it is put into production. Existing solutions (e.g. SexiGraf, psview, and checkmk) do not adhere to infrastructure-as-code principals, and are instead deployed as complex, monolithic and non-integrating applications. In response to this need, we built OmegaGraf, a one-click vCenter monitoring solution. OmegaGraf is an entirely open-source, containerized solution, configured and deployed through a simple web interface, providing dynamic dashboards, granular time series data, environment aggregation, and alerting. Best of all, OmegaGraf is deployable in under five minutes, allowing IT professionals to provide value to their organization faster than ever before.

1.0 Introduction

1.1 Problem Statement

According to the Institute of Electrical and Electronics Engineers (IEEE), agile methodologies have driven organizations to alter their infrastructure at an ever-increasing rate (Diaz 2019). Due to this acceleration, IT professionals are often pressured to stand up new products and services at speeds aligning with agile timescales, which may lead to neglect in areas such as monitoring and compliance. Setting up monitoring meeting the product's specifications can be both difficult and time consuming, and neglect is not immediately seen by the rest of the business.

vCenter is one such solution that can be very difficult to correctly monitor, as there are many sources for metrics, and organizations often have multiple instances running concurrently. VMware themselves state that external, unified monitoring is a requirement for hardened environments, yet vCenter does not include tooling to accomplish this. In production, unmonitored vCenter environments may have security alerts, threshold issues and other problematic events go completely unnoticed.

Although many tools exist to add this functionality, they often charge exorbitant sums for much needed features. Open-source tooling, such as Zabbix, Nagios, or TICK, have steep learning curves, and tend to specialize to such a degree that multiple products are needed to gain a holistic view. Configuration is complicated and requires editing multiple configuration files to get tools up and running. Learning how to configure and collect the

data you need is time-consuming and difficult. If a modular, containerized and single-interface solution were developed, it would allow for a fast and easily configurable monitoring solution. If this solution was also able to integrate with corporate infrastructure, far fewer vCenter environments would go unmonitored.

1.2 Solution

OmegaGraf is a one-click, modular solution that immediately sets up probes and begins monitoring your vCenter environment. This application has a clean, simple interface that orchestrates multiple containers, which will collect, store, and display metrics from any vCenter environment. OmegaGraf is free, open source, and will utilize exclusively open-source software. Through integrations with other open-source products, OmegaGraf provides dynamic dashboards, granular time series data, environment aggregation, alerting, and much more.

Existing solutions to this problem (e.g. SexiGraf, Opsview, and checkmk) do not adhere to infrastructure-as-code principals, and are instead deployed as complex, monolithic, and non-integrating applications. OmegaGraf, however, is a modular solution which uses an independent container for each service. This architecture is much more manageable; services can easily be upgraded, moved, and restarted with minimal impact. In addition, this setup allows organizations to integrate the near-instant OmegaGraf setup with pre-existing graphing services they might already have running in their current environment.

Due to its ease of use, flexibility with other tools, and incredibly fast deployment time, OmegaGraf aims to ensure that no vCenter environment will lack adequate monitoring when deployed to production.

1.3 Project Goals

OmegaGraf provides a simple vCenter monitoring solution in under 5 minutes. It is modularly designed from the ground up with both enterprise and enthusiasts in mind and is completely open source. OmegaGraf consists of multiple components, including a management interface which will allow users to deploy our solution without having to understand Docker, edit config files, or perform time-consuming technical tasks.

Minimal prerequisites are required, and should consist only of a few shell commands, a host system, and, of course, a vCenter environment that needs to be monitored. We provide simple, step-by-step and easy to follow documentation, which should allow any technician to quickly setup and deploy a monitoring solution with OmegaGraf.

1.4 Report Overview

The following outlines in detail the process and results of our project. This includes our project's concept, its design objectives, the methodology and technical approach we took, technical specifications, our testing results, problems we encountered, and key takeaways. These sections should give a detailed look into the specifics of OmegaGraf's development.

2.0 Discussion

2.1 Project Concept

2.1.1 Project Overview

OmegaGraf is a container-based application built with the intention to deploy a monitoring solution for vCenter environments quickly, efficiently, and with little interaction during the process. On initial setup, the user is greeted with a simple, single-page setup web interface to make any changes to the default configuration that they want. Once confirmed, the web interface pushes an application programming interface (API) call to the backend, which then uses the parameters specified to automatically deploy a monitoring environment with only a button press.

Although the application automatically chooses specifications by default, the user can make adjustments to these through optional subcategories on the web interface. This way, the application provides simplicity for users who just want to click “Deploy” and get started, but also provides complexity for users who have specific needs such as an already existing environment.

2.1.2 Project Source

OmegaGraf was inspired by a community driven VMware monitoring appliance called SexiGraf. While it had a very interesting premise, reception was very negative due to it targeting mostly enthusiasts. The naming and general style of the project caused it to completely miss a potentially large enterprise audience. Even for home users, the project lacked critical features, and was deployed as a monolithic appliance.

OmegaGraf starts at the source problem and delivers a much cleaner, more streamlined approach. It also targets enterprise and home users alike, as they tend to share the same needs in this space. Most user groups have more important or interesting tasks they would prefer to be performing, rather than setting up graphing for their solution.

2.2 Design Objectives

2.2.1 Goals and Requirements

Our team planned to develop a functional vCenter monitoring solution, deployable through a web interface. Our solution was designed to take no more than 5 minutes from clicking “Deploy” to having a fully configured and operational monitoring solution.

OmegaGraf meets all criteria through the following requirements:

- ➔ Solution collects, stores, and displays common vCenter metrics
- ➔ Deployment takes under 5 minutes for one vCenter instance
- ➔ Database has support for external Grafana instances
- ➔ Solution supports multiple vCenter instances, viewable from a single frontend
- ➔ Autorun installer application with a visually appealing graphical interface
- ➔ Frontend written primarily in TypeScript
- ➔ Backend written primarily in C#, Shell, and PowerShell
- ➔ Software security meets government standards NIST S.P. 800-37 and 800-53

These requirements provide a fast, simple monitoring solution that provides holistic insights into any vCenter environment.

2.2.2 Abandoned Goals

Along the process of development, some features were determined to offer too little in relation to the effort required. Over time, we decided to drop two different goals, each of which we had initially listed as “optional”.

We had initially intended to offer monitoring of ESXi hosts, which would provide some additional bare-metal metrics to our system. However, we chose to drop this optional requirement, as ESXi only serves metrics via SNMP. Thus, this capability would require an entire additional layer on our application. Additionally, we discovered that vCenter already provides metrics from individual hosts, so the requirement was somewhat redundant.

Additionally, we had initially considered supporting high availability (HA) configurations, to provide some resilience to our system. However, HA setups with our database simply consist of the pipeline being entirely replicated. If a user desired a HA setup, they could simply run OmegaGraf a second time on a different host. However, standing up a usable setup from these two instances would require configuration of one or more HA proxies, something that is entirely out of our project’s scope. Thus, this optional requirement was dropped as well.

2.3 Methodology / Technical Approach

2.3.1 Design Requirements & Procedures

Our initial requirements were critical, especially in the early stages of OmegaGraf's development. Through extensive prerequisite research, we knew exactly what technologies would work for us, as well as what would be feasible to complete. We kept our goals very modest and attempted to create a minimum viable product (MVP) as fast as possible, without increasing our scope. At the same time, we always kept extensibility at the forefront of our minds , knowing that we would most likely want to eventually exceed our requirements.

Shortly after we initially released our design requirements, we had a project name, a logo, an interface mockup, and documentation of our envisioned user experience. We have followed these initial items very closely through the entire development lifecycle, which has reduced the amount of time spent rewriting or redesigning our solution to levels approaching zero.

Due to our careful limitations of scope and focus on these initial requirements, documents and concepts, we were able to release an MVP before our development cycle reached the halfway point. We believe that this has significantly increased the quality of our current product, as it has allowed much more time for quality testing, feedback and improvement. Additionally, a focus on extensibility has allowed us to introduce many additional features that were not present in our initial requirements.

2.4 User Profile

Project
OmegaGraf

User Profile
➔ Anyone using the VMware vCenter ecosystem

Software, Interface, and Related Experience
OmegaGraf users should have at least minimal experience with CLIs, vCenter, and other VMware solutions used in their environment. Additionally, users should have an intermediate knowledge of both Docker and NFS.

Experience with Similar Applications
OmegaGraf users may be experienced with the following similar applications:

- ➔ VMware built-in monitoring
- ➔ checkmk
- ➔ Nagios
- ➔ Opsview
- ➔ SexiGraf
- ➔ TICK
- ➔ Zabbix

Task Experience
The user will start by copying a command posted at omegagraf.github.io, which will set up their virtual machine and download the latest version of OmegaGraf. This will be a guided process and the user will only need to run a single command. After the installation is complete and OmegaGraf is started, a URL will be provided. The user will then connect to the OmegaGraf frontend via their web browser of choice. OmegaGraf will present the user with a simple interface, allowing them to choose their vCenter instances, and change any options (if desired). The user will click the green “Deploy” button. The OmegaGraf service will now build the monitoring environment. A loading screen will be shown, indicating that progress is being made. A few minutes later, the user will be presented the application URLs, and they will be able to login to their graphing application of choice.

Frequency of Use
OmegaGraf itself will only need to be used once per environment. If the environment is changed, OmegaGraf can be used again to reconfigure. The containers that OmegaGraf creates will be used on a daily to hourly basis, depending entirely on user preference.

Key Project Design Requirements that the Profile suggests
- ➔ Optional granular configuration parameters
- ➔ Support for common Linux distros
- ➔ Simple, easy to use web UI
- ➔ Meets NIST 800-37 & 800-53 requirements

2.5 Use Case Diagram

Figure 1 presents the use case for OmegaGraf. The use case shows all the different interactions a user type has when accessing the application.

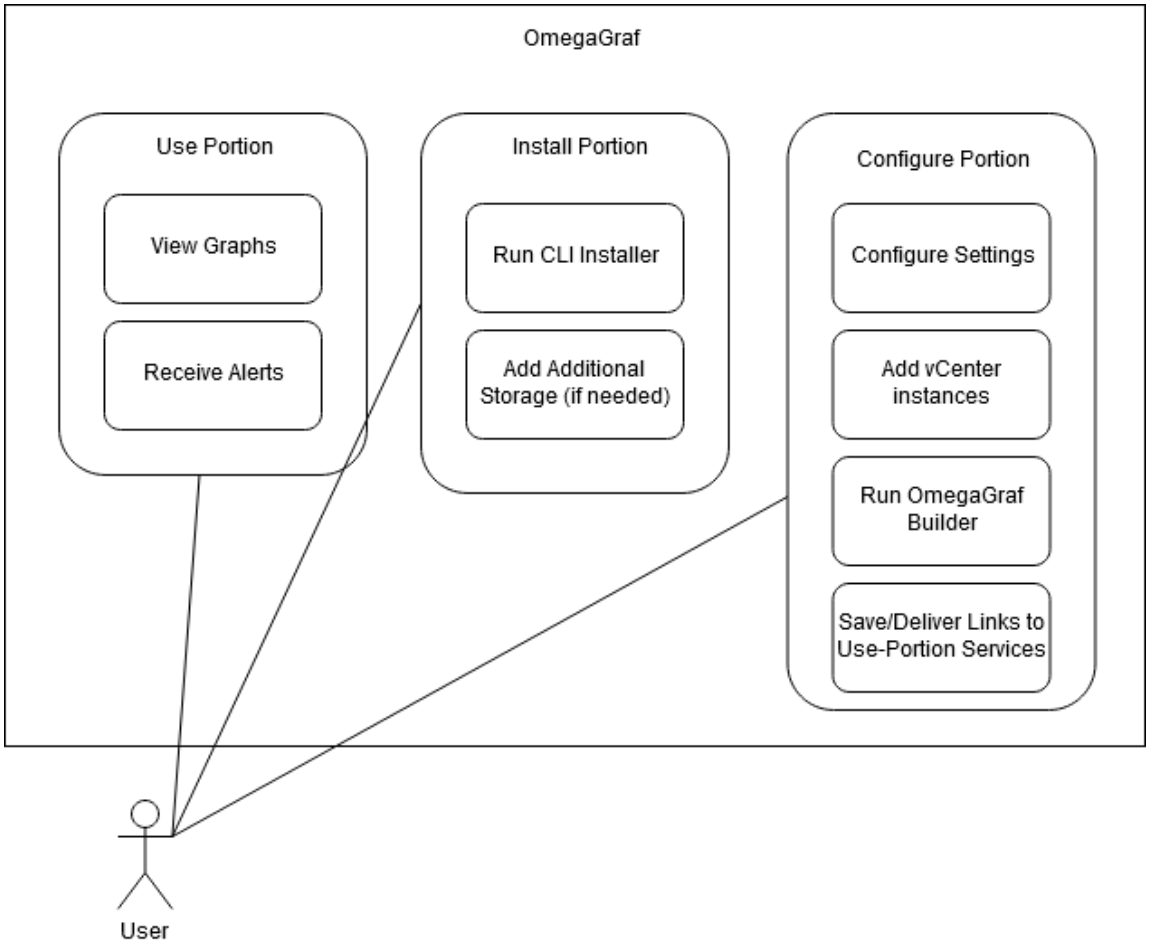


Figure 1: Use case diagram

2.6 Technical Architecture

Figure 2 depicts the multiple technologies utilized by OmegaGraf and shows how they interact with one another on a given network. Dotted lines depict a logical and/or non-physical association.

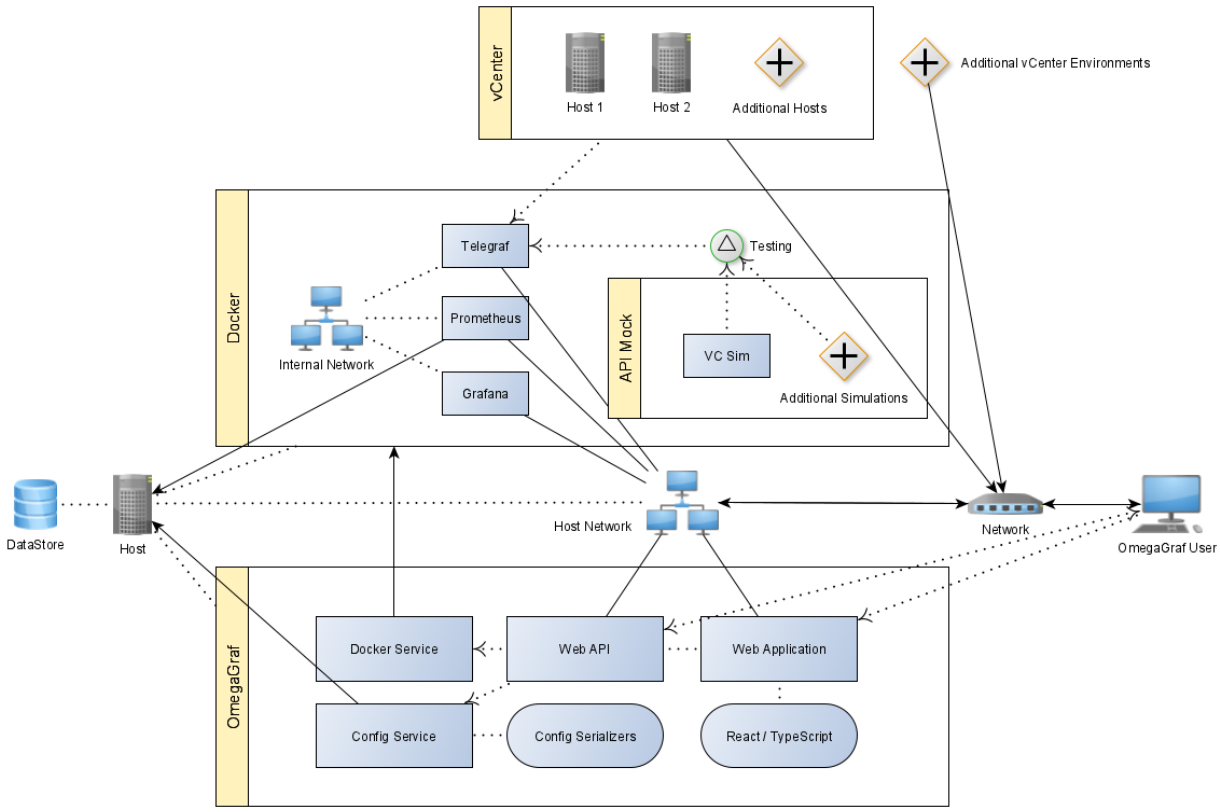


Figure 2: Technical architecture diagram

2.6.1 Setup

Initial setup of OmegaGraf is performed through one of our installation shell scripts. We have scripts for popular Linux distributions, such as Ubuntu/Debian. Minor versions are automatically detected, and small alterations will be made to optimize the process. These scripts first assist the user with adding and installing necessary dependencies, such as docker-ce. The install script will then offer to download and run the latest release of OmegaGraf, after dependencies are finished installing. Figure 3 shows the initial portion of the setup on Ubuntu 18.04.

```
$ bash <(curl -s https://raw.githubusercontent.com/OmegaGraf/OmegaGraf/master/install/debian-amd64.sh)
>> installing OmegaGraf for ubuntu/bionic 18.04
[sudo] password for user: ************
Fetched 3666 kB in 8s (476 kB/s)
Reading package lists... Done
Building dependency tree
Reading state information... Done
Reading package lists... Done
Building dependency tree
Reading state information... Done
ca-certificates is already the newest version (20180409).
curl is already the newest version (7.58.0-2ubuntu3.8).
software-properties-common is already the newest version (0.96.24.32.12).
apt-transport-https is already the newest version (1.6.12).
gnupg2 is already the newest version (2.2.4-1ubuntu1.2).
0 upgraded, 0 newly installed, 0 to remove and 150 not upgraded.
Would you like to enable SSL? (y/n) y
OK! I will be installing certs and running on HTTPS port 5001.
>> installing microsoft tooling for developer certs
Preparing to unpack packages-microsoft-prod.deb ...
Unpacking packages-microsoft-prod (1.0-ubuntu18.04.2) over (1.0-ubuntu18.04.2) ...
Setting up packages-microsoft-prod (1.0-ubuntu18.04.2) ...
```

Figure 3: Capture of running the setup script

2.6.2 App Service & API

The heart and soul of OmegaGraf is the application service running on a Docker host. This service is a single, portable file, and is compatible with Windows and most Linux distributions. The service, once started, finds and hooks into the host's Docker socket, allowing it to create and otherwise orchestrate containers as needed. An API endpoint is

exposed, allowing the user to make the necessary calls to create and configure containers via OmegaGraf's web interface. Internally, various serializers allow the API to bind configuration data to objects that represent container configuration files, which may then be written at the container bind locations. Figure 4 depicts this process.

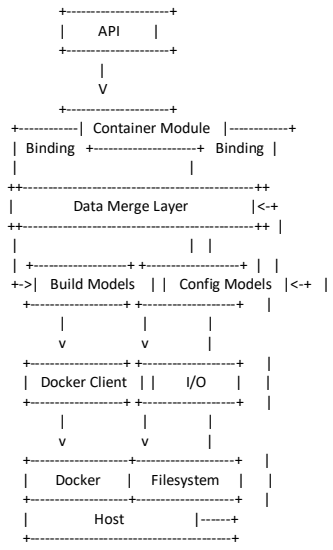


Figure 4: Diagram of the application service

2.6.3 Web Interface

OmegaGraf's app service hosts a user interface (UI), which is, by default, exposed over HTTPS port 5001. (Or HTTP port 5000, if the host does not have a certificate installed.) The UI is a React single-page-application (SPA), which allows OmegaGraf to update the web interface's state, without any page loading. This amounts to a great user experience, since there are essentially zero wait times through the entire process. The SPA makes regular changes to the application state, depending on the user's preferences. Feedback will also be constantly provided, since we can regularly query OmegaGraf's app service in a way that is completely transparent to the end user. In essence, the interface is dynamically updated with status details as orchestration progresses.

The interface (see Figure 5) itself consists of three main elements. First, there is a login form, which will prompt users to enter their unique key (provided by OmegaGraf on launch). Once the key has been validated, the user is directed to a setup form, which allows the user to input their vCenter credentials. This form also is used to allow the user to customize any settings as they see fit, though this is completely optional. Once the user submits the form, the user is shown regular progress updates until orchestration is complete.

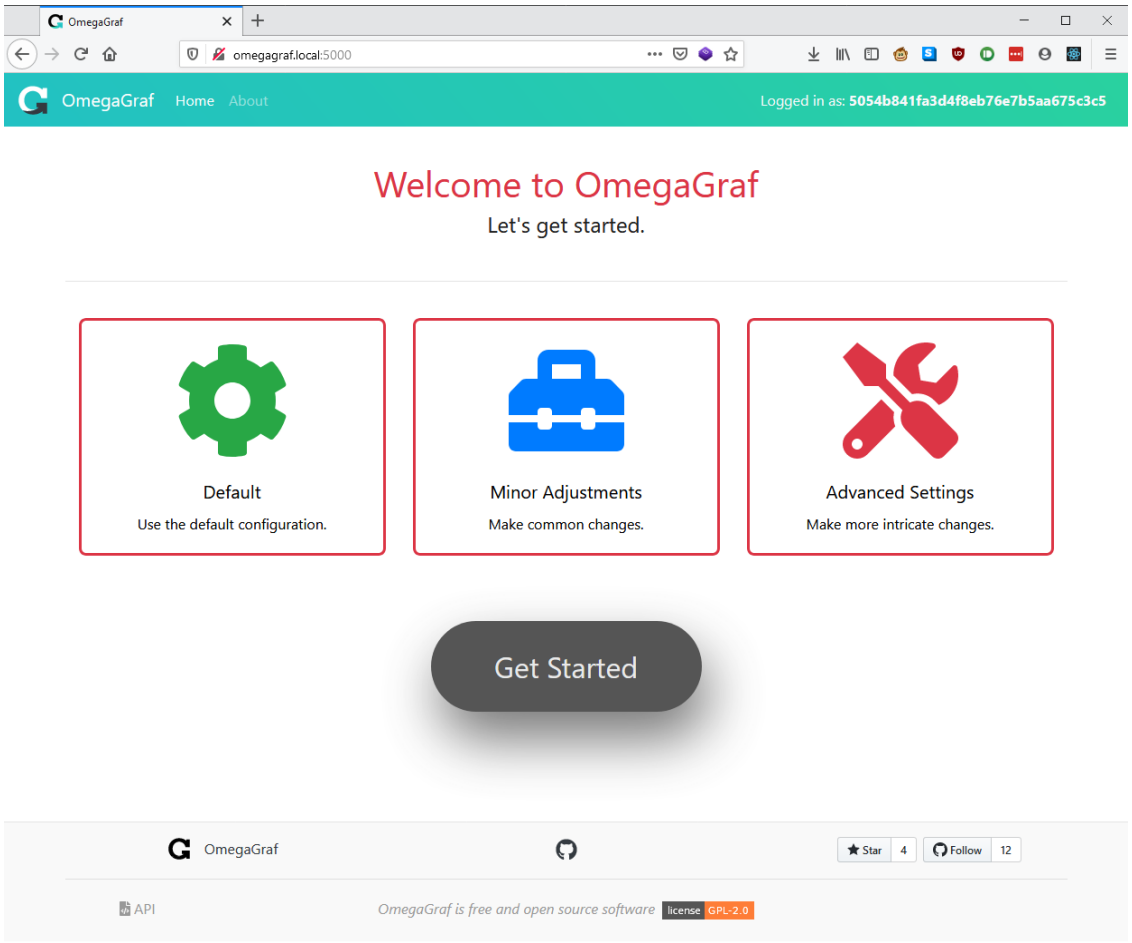


Figure 5: Homepage of the OmegaGraf UI

2.6.4 Containers

The only permanent part of OmegaGraf is the containers, which will continue to collect, store and display metrics after all other services are terminated. In production, we use three containers.

1. Telegraf

Telegraf is a lightweight service that routinely scrapes any number of vCenter endpoints. It then stores and exposes those metrics on an internal endpoint, until they are considered stale.

2. Prometheus

Prometheus is a minimal and extremely efficient time-series database, which routinely crawls metrics endpoints according to an internal configuration file. This is configured to crawl Telegraf. Once an endpoint is crawled, the metrics are stored in Prometheus until the end of retention.

3. Grafana

Grafana is the frontend for our data (see Figure 6). It displays graphs via querying Prometheus, using PromQL. Grafana also has an API, which OmegaGraf uses to automatically add Prometheus, as well as several curated dashboards for visualizing the metrics collected from vCenter.

Figure 6: Default overview of Grafana deployed by OmegaGraf

In development, we have an additional container which we created using vmware/govmomi, called VCSim. This is a simulation of the vCenter API, complete with simulated hosts, clusters, virtual machines (VMs), and more. This allows us to simulate an environment consisting of any number of vCenter instances, and other objects existing inside those instances. Users of OmegaGraf have an option in the web interface to enable these simulations, in case they would like to demo the solution.

2.6.5 Networking

The majority of OmegaGraf's networking is completely internal. OmegaGraf creates a custom Docker network, allowing containers to pass data between themselves without being exposed to the host network. The only container that must be exposed is the graphing component, Grafana. This container binds a single port, which is used for

accessing the graphing interface. Prometheus, our time-series database, may also be exposed, so that it may be queried using an external graphing tool.

2.6.6 Security

Security is something that we consider each step of the way while developing OmegaGraf. Alongside making the application compliant with the National Institute of Standards and Technology (NIST) regulations, we also wanted to prove compliance. This way, users can see every step OmegaGraf takes to meet government standards and that the application is secure enough to deploy on any system. OmegaGraf is developed to have minimal external network exposure. By default, data is only ever transported outside of Docker during interaction with Grafana. Minimizing the amount of data users send over the network means less exposure, resulting in better security for OmegaGraf. As the data is transported between the server and client, the communication is encrypted using HTTPS and SSL protocols. To help keep the data secure, the application generates and prints a key to be used as a login while visiting the web UI. This is more secure than using accounts, quicker and easier to manage for administrators, and it simplifies setup for the end user.

2.7 Testing

2.7.1 Overview

For testing procedures, our main methods of approach were as follows:

- ➔ UI testing via Azure DevOps Pipeline
- ➔ Backend unit and integration testing via Nunit in Travis

- ➔ vCenter mock container testing via DockerHub CI
- ➔ Static code analysis via SonarCloud and DeepScan
- ➔ Manual testing via an amalgamation of scenarios and test cases

One important piece of the process is making sure each iteration is seen through several pairs of eyes - the Lead Developer, the Test Lead, and an additional test user, such as a stakeholder. This allows the program to be tested under use through different people who may work differently based on their use cases. For example, a developer might know exactly what they're looking for as a result inside of their product, but an end user may expect something different or use the program differently.

Our test cases attempt to reach for any possible configuration. To start, we want to write one test case for "ideal use." This includes the use of a Windows 10 computer running Chrome using the application's default settings and everything filled properly. In addition, we have test plans for irregular or error-prone usage. This includes leaving a field blank, having incorrect data populated, setting an incorrect username or password, or any combination of these. This gives us an idea of how our application runs, not just in ideal scenarios, but also in situations where users may not correctly use the interface.

For developing and recording our plans we used Azure DevOps Test Plans (see Figures 7 and 8). The process is as follows:

1. Lead Developer and Test Lead work to create test cases for QA testing.
2. Lead Developer, once having features initially developed, will run through preliminary scenarios. This will allow the team to catch possible bugs early on.
3. Once the previous tests pass, they will deliver the tested version to Test Lead.
4. Test Lead will run through all test cases, documenting bugs encountered.

5. Test Lead will give the test report to the Lead Developer for patching.
6. Steps 2-5 are repeated until no further bugs are encountered.
7. Once the cycle is passed, a release is sent to a test user for testing.

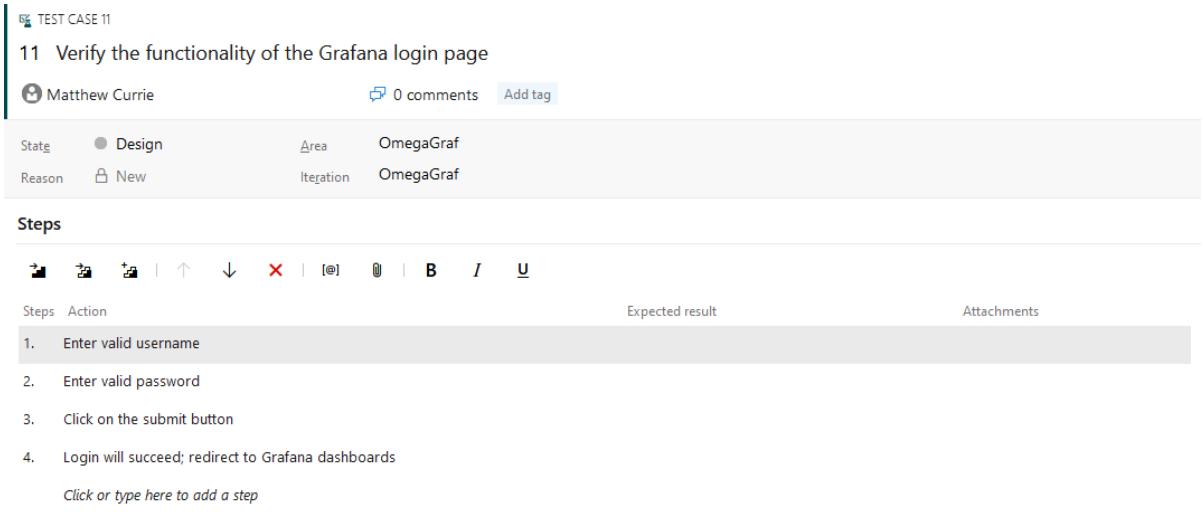


Figure 7: Example of a functionality test case

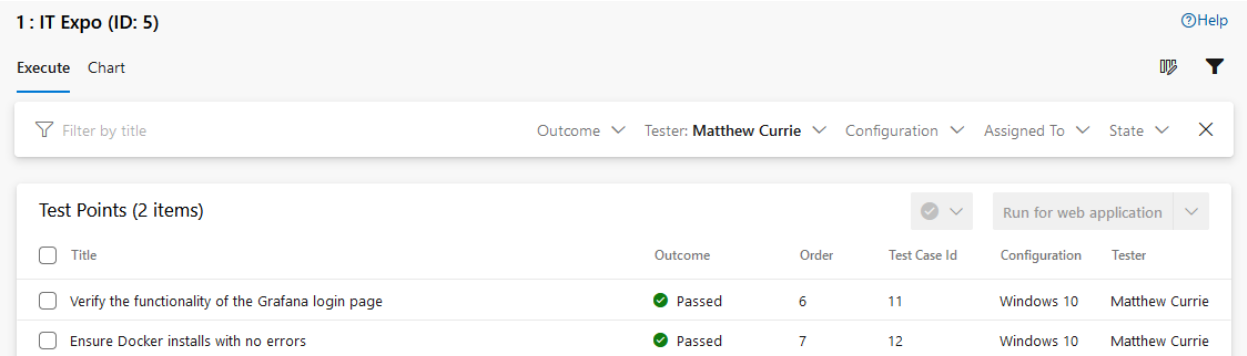


Figure 8: Example of a test plan outcome

As part of our testing, we ran automated tests of our backend, mostly covering integration with the Docker client. Figure 9 shows an example of this testing.

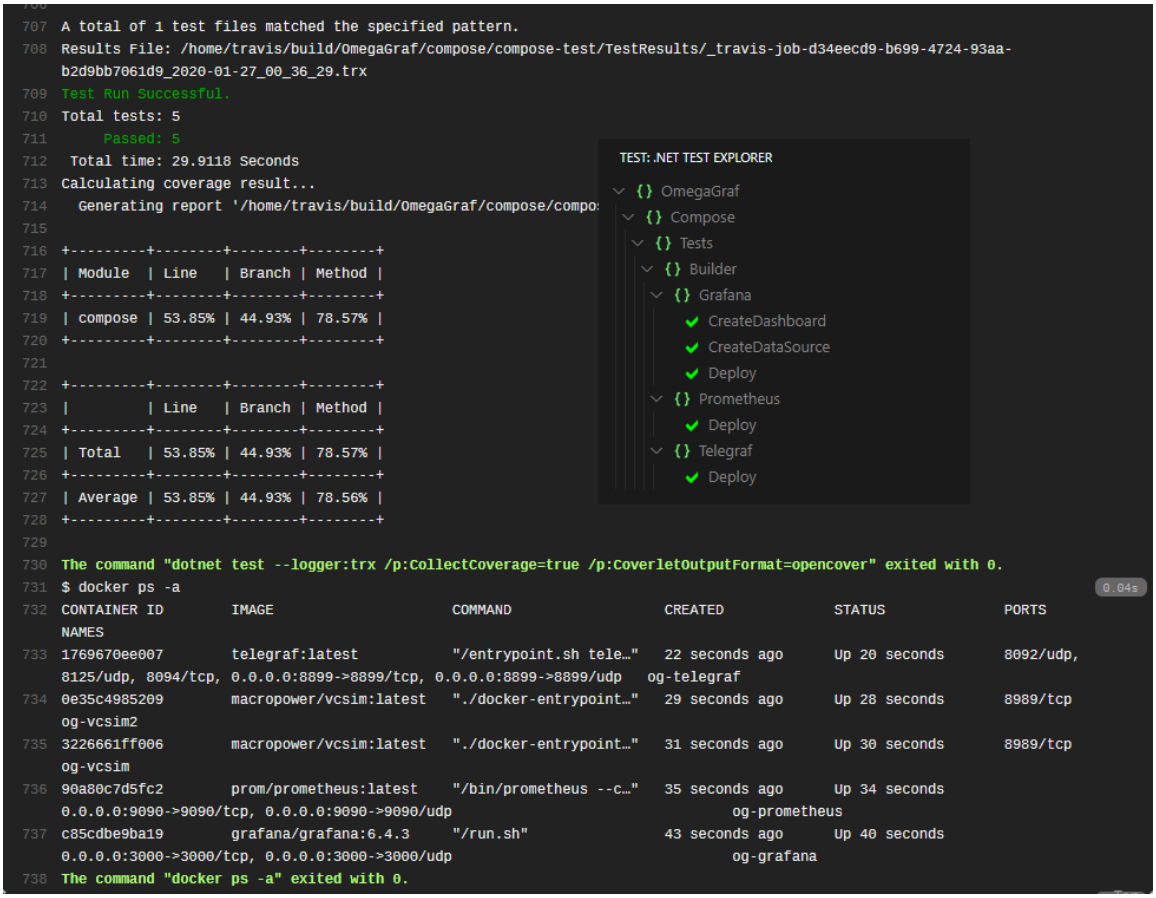


Figure 9: Example of backend automated test results

Figure 10 shows a report generated by a SonarCloud agent which runs as a part of our build pipeline in TravisCI. This report lists potential vulnerabilities and bugs it encountered during the build, as well as metrics around code coverage and duplication. This report is marked “passed” when the count of discovered bugs and vulnerabilities are both zero.

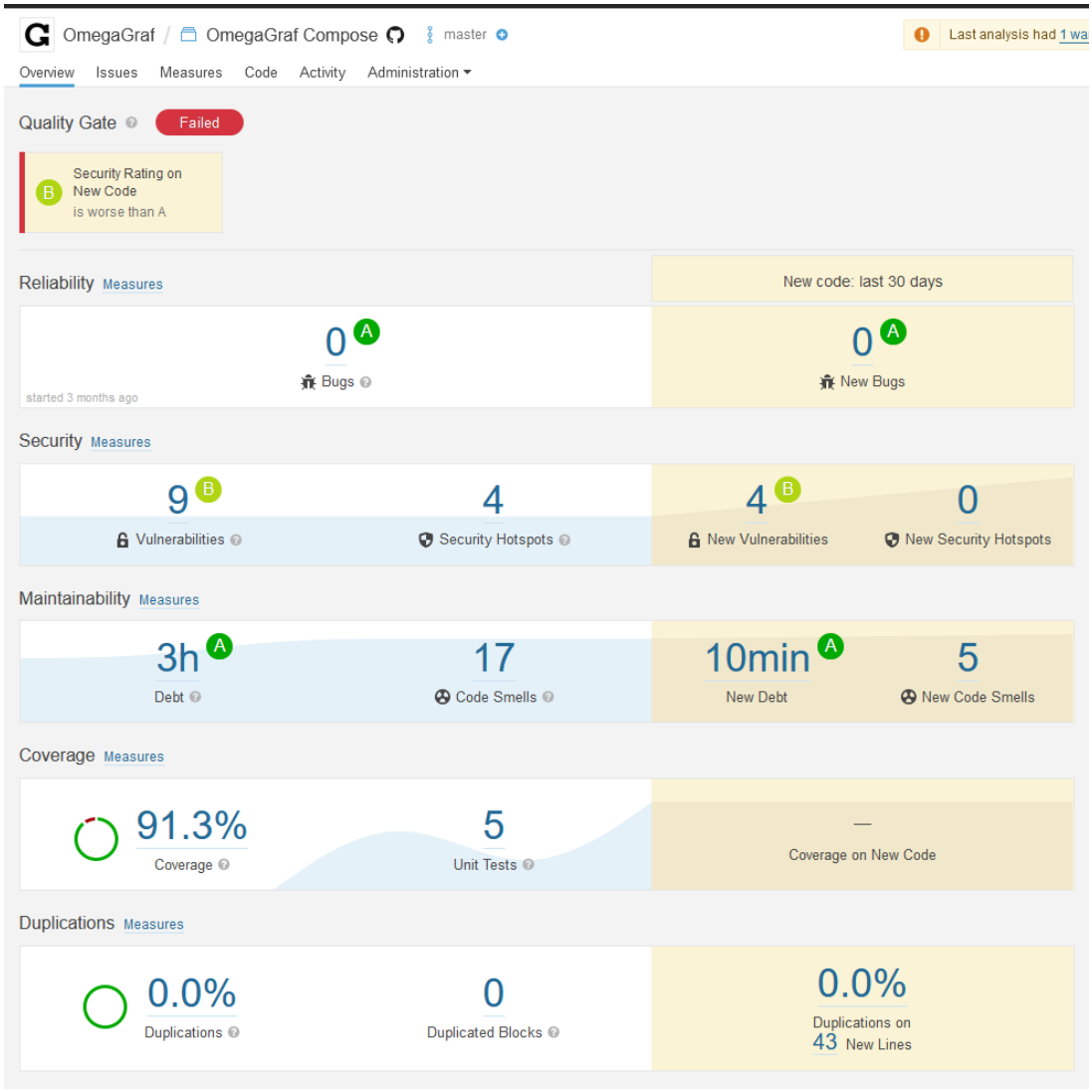


Figure 10: Example of backend code quality scan results

Figure 11 shows a report generated by DeepScan, which runs on each pull request. This report is marked “good” when the count of discovered issues is zero.

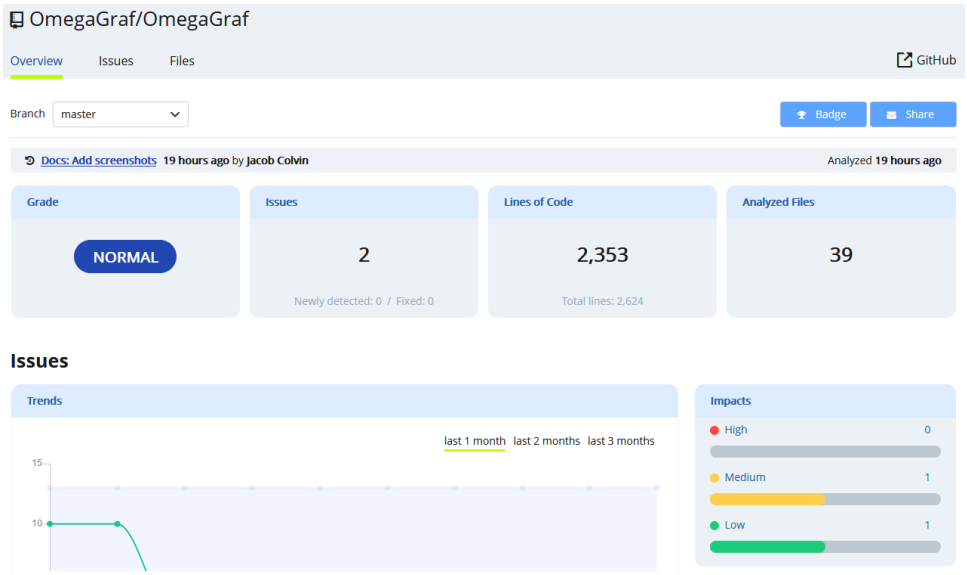


Figure 11: Example of UI code quality scan results

2.7.2 Scope of Testing

Our test plan attempts to test all major and minor use cases through the OmegaGraf API and UI. This includes combinations of sending a manual API call, filling out default settings, simple settings, advanced settings, and combinations of the aforementioned. Because these are the avenues of use for OmegaGraf, testing these should cover most of our bases. Figure 12 lays out an outline of what our test plan considers.

1.0 Functionality Testing

TEST ID	SCENARIO	DESCRIPTION
FN_01	Verify the installation functionality of OmegaGraf	Ensure Docker installs with no errors
FN_02	Verify the installation functionality of OmegaGraf	Ensure Grafana installs with no errors
FN_03	Verify the installation functionality of OmegaGraf	Ensure Prometheus installs with no errors
FN_04	Verify the installation functionality of OmegaGraf	Ensure Telegraf installs with no errors
FN_05	Verify the installation functionality of OmegaGraf	Ensure VCSim installs with no errors
FN_06	Verify the login functionality of OmegaGraf initial login page	Test a valid secure code
FN_07	Verify the login functionality of OmegaGraf initial login page	Test an invalid secure code
FN_08	Verify the login functionality of OmegaGraf initial login page	Test a valid vCenter URI
FN_09	Verify the login functionality of OmegaGraf initial login page	Test an invalid vCenter URI
FN_10	Verify the login functionality of OmegaGraf initial login page	Test using vCenter Simulator
FN_11	Verify the header functionality of OmegaGraf initial login page	Verify About page link redirects correctly
FN_12	Verify the header functionality of OmegaGraf initial login page	Verify Display page link redirects correctly
FN_13	Verify the login functionality of Grafana login page	Test a valid username and valid password.
FN_14	Verify the login functionality of Grafana login page	Test an invalid username and valid password.
FN_15	Verify the login functionality of Grafana login page	Test a valid username and invalid password.
FN_16	Verify the login functionality of Grafana login page	Test an invalid username and invalid password.
FN_17	Verify the configuration of data sources	Check configured data sources import correctly
FN_18	Verify the configuration of data sources	Check configured data sources add and display correctly.
FN_19	Verify data displays in Grafana dashboard	Confirm dashboard populates graph from data source
FN_20	Verify instance filter in Grafana dashboard	Confirm instance filters correctly add and/or remove instance categories.
FN_21	Verify the functionality of temporary credentials	Confirm temporary password is accepted.
FN_22	Verify the functionality of temporary credentials	Check that temporary password expires.

2.0 Security Testing

TEST ID	SCENARIO	DESCRIPTION
SC_01	Verify the password requirements of Grafana dashboard	Verify passwords cannot be less than 8 characters.
SC_02	Verify the password requirements of Grafana dashboard	Verify incorrect username / password displays 'Invalid Credentials' to protect existing user accounts
SC_03	Verify the password requirements of Grafana dashboard	Verify incorrect username / password displays 'Invalid Credentials' to protect existing user accounts
SC_04	Verify previous accessible pages cannot be accessed after logging out.	Confirm going back to previous pages does not display any data after logging out.
SC_05	Verify the functionality of temporary credentials	Confirm temporary password is accepted.
SC_06	Verify the functionality of temporary credentials	Check that temporary password expires.
SC_07	Verify secure communication	Confirm website loads when using HTTPS
SC_08	Verify secure communication	Confirm website will not load when using HTTP

3.0 User Acceptance Testing

TEST ID	SCENARIO	DESCRIPTION
UA_01	Download OmegaGraf and complete setup using script.	Ensure all OmegaGraf components install with no errors
UA_02	Add a new data source to the Grafana dashboard	Import a new data source into the dashboard
UA_03	Add a new data source to the Grafana dashboard	Add a new vCenter to the dashboard
UA_04	Reset account credentials	Change password to a new one.
UA_05	Configure data source in Grafana dashboard	Apply a filter to only view (memory usage)

Figure 12: List of test cases

2.7.3 Objectives

Our main objective was to ensure that our application has proper handles for anything that a user can throw at it. This is covered through several smaller objectives:

- ➔ Each piece of user interaction must be tested.
- ➔ Each test must be run through different credential scenarios; a.k.a. Ensuring authentication.
- ➔ Any bug must be reported in a Bug Report in Azure DevOps, eventually compiled into an overarching GitHub issue.
- ➔ Each bug must be accounted for; either fixed outright or acknowledged and addressed.
- ➔ Each test must pass.
- ➔ All these objectives must be reached by Tech Expo.

2.7.4 Logging Tests and Procedures

Throughout our testing, the use of Azure DevOps tools will help us log UI testing results. On every major release, all team members are to run all tests in our Azure DevOps UI project. These results are recorded and tracked in a progress report that should reflect our burndown rate (see Figure 13).

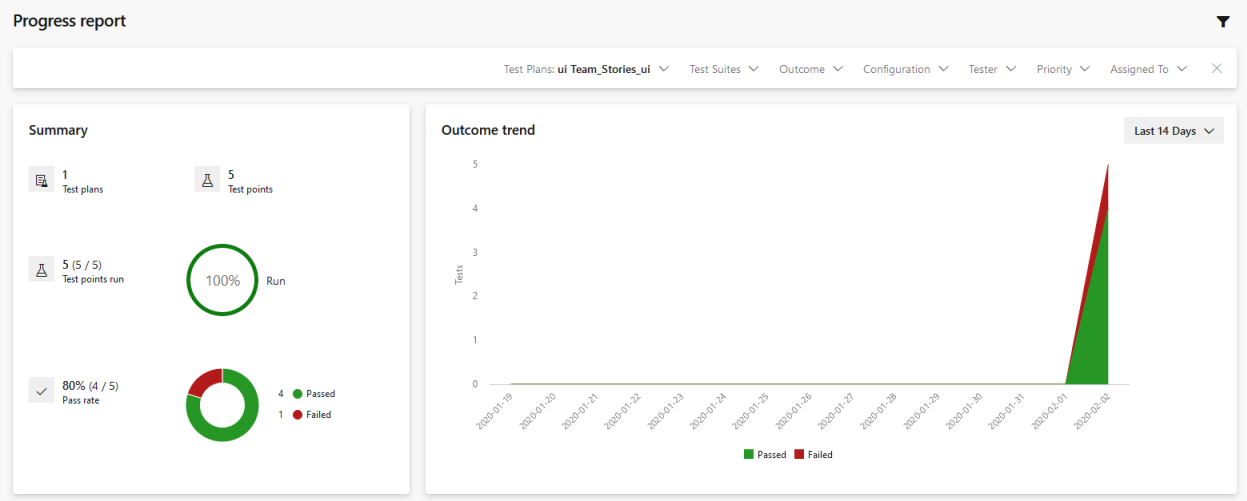


Figure 13: Example of Azure test outcome trend report

On every commit or PR to any of our GitHub repositories, automated testing is performed. This includes backend unit testing, a code quality scan, and compilation testing. When a commit adds a new feature to the application, new test cases are written to ensure the functionality of the new feature is fully tested and works as intended. Test cases have been divided into three primary categories: Functionality, Security, and User Acceptance. Functionality cases are testing the functionality of the UI. Security cases deal with securing the environment and correcting flaws in any security mechanisms. User Acceptance cases are tasks that an end user can accomplish. These cases are tested by

someone outside of the development and testing teams to see how an outside user interacts with the application.

If any issues are discovered, team members are to create an Issue in our project's GitHub repository (see Figure 14). Team members are to label these issues as either "bug", "enhancement", or "documentation". All bugs are to be placed in an "Expo" milestone and must be resolved before IT Expo 2020. Enhancement and documentation requests may be placed under "Expo", "Expo-Optional" or "Post-Expo", depending on whether they are needed to meet our project's requirements.

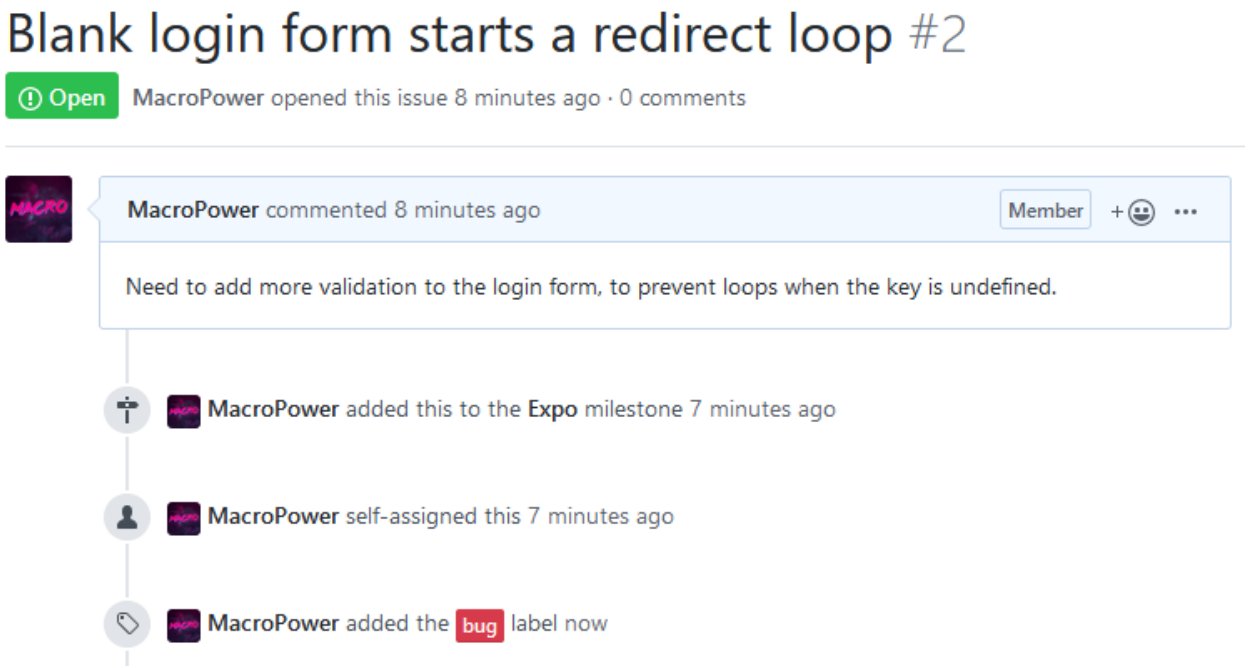


Figure 14: GitHub Issue

Updates to the issue will be posted in the issue comments. Once an issue has been fully addressed, it will be automatically closed by the related code commit or pull request.

2.7.5 Test Results

Figures 15-17 list the complete results of our testing of functionality. Functionality tests look at the main actions that a user will want to perform while using the application. Some cases are tests multiple times for data that is true, and once again when the data is false. For example, when logging into the application with the secure code, we also verified that it wouldn't take mismatching codes.

TEST ID	SCENARIO	DESCRIPTION	TEST STEPS	EXPECTED RESULT	ACTUAL RESULT	STATUS (PASS/FAIL)
FN_01	Verify the installation functionality of OmegaGraf	Ensure Docker installs with no errors	1. Download from GitHub 2. Open PowerShell/CMD 3. Run command .\OmegaGraf.exe	Program will install the UI and backend successfully; OmegaGraf will display secure code.	Executable successfully installed Docker with no errors.	Pass
FN_02	Verify the installation functionality of OmegaGraf	Ensure Grafana installs with no errors	1. Download from GitHub 2. Open PowerShell/CMD 3. Run command .\OmegaGraf.exe	Program will install the UI and backend successfully; OmegaGraf will display secure code.	Executable successfully installed Grafana with no errors.	Pass
FN_03	Verify the installation functionality of OmegaGraf	Ensure Prometheus installs with no errors	1. Download from GitHub 2. Open PowerShell/CMD 3. Run command .\OmegaGraf.exe	Program will install the UI and backend successfully; OmegaGraf will display secure code.	Executable successfully installed Prometheus with no errors.	Pass
FN_04	Verify the installation functionality of OmegaGraf	Ensure Telegraf installs with no errors	1. Download from GitHub 2. Open PowerShell/CMD 3. Run command .\OmegaGraf.exe	Program will install the UI and backend successfully; OmegaGraf will display secure code.	Executable successfully installed Telegraf with no errors	Pass
FN_05	Verify the installation functionality of OmegaGraf	Ensure VCSim installs with no errors	1. Download from GitHub 2. Open PowerShell/CMD 3. Run command .\OmegaGraf.exe	Program will install the UI and backend successfully; OmegaGraf will display secure code.	Executable successfully installed VCSim with no errors.	Pass
FN_06	Verify the login functionality of OmegaGraf initial login page	Test a valid secure code	1. Enter valid secure code 2. Click on Submit button	Login will succeed	Login succeeds; redirects to Welcome page	Pass
FN_07	Verify the login functionality of OmegaGraf initial login page	Test an invalid secure code	1. Enter invalid secure code 2. Click on Submit button	Login will fail	Login fails; displays "Unauthorized"	Pass

Figure 15: Functionality testing pt. 1

FN_08	Verify the login functionality of OmegaGraf initial login page	Test a valid vCenter URI	1. Enter valid vCenter URI 2. Enter valid vCenter Username 3. Enter valid vCenter Password 4. Click on Deploy button	Login & deployment creation will succeed	Deployment succeeds; visit http://localhost:3000 and sign to view graphs	Pass
FN_09	Verify the login functionality of OmegaGraf initial login page	Test an invalid vCenter URI	1. Enter invalid vCenter URI 2. Enter valid vCenter Username 3. Enter valid vCenter Password 4. Click on Deploy button	Login & deployment creation will fail	Deployment fails; displays message "Error creating container, please check server logs"	Pass
FN_10	Verify the login functionality of OmegaGraf initial login page	Test using vCenter Simulator	1. Check box to Use a vCenter Simulator 2. Enter a valid Number of Simulated Instances (x >=1) 3. Click on Deploy button	Creation will succeed; containers will be running.	Deployment succeeds; visit http://localhost:3000 and sign to view graphs	Pass
FN_11	Verify the header functionality of OmegaGraf initial login page	Verify Home page link redirects correctly	1. Navigate to https://localhost:5001 2. Click About button 3. Click Home button	Button will redirect to main home page	Home page loads and displays "Welcome to OmegaGraf" with initial config options	Pass
FN_12	Verify the header functionality of OmegaGraf initial login page	Verify About page link redirects correctly	1. Navigate to https://localhost:5001 2. Click Display button	Button will redirect to About page	About page loads and displays session data, sim data, and global data	Pass
FN_13	Verify the login functionality of Grafana login page	Test a valid username and valid password.	1. Enter valid username 2. Enter valid password 3. Click on Submit button	Login will succeed; redirect to Grafana dashboards	Login succeeds; redirects to Grafana dashboard	Pass
FN_14	Verify the login functionality of Grafana login page	Test an invalid username and valid password.	1. Enter invalid username 2. Enter valid password 3. Click on Submit button	Login will fail	Login fails; displays error message "Invalid username or password"	Pass
FN_15	Verify the login functionality of Grafana login page	Test a valid username and invalid password.	1. Enter valid username 2. Enter invalid password 3. Click on Submit button	Login will fail	Login fails; displays error message "Invalid username or password"	Pass
FN_16	Verify the login functionality of	Test an invalid username and	1. Enter invalid username 2. Enter invalid password 3. Click on Submit button	Login will fail	Login fails; displays error message	Pass

Figure 16: Functionality testing pt. 2

	Grafana login page	invalid password.			"Invalid username or password"	
FN_17	Verify the configuration of data sources	Check configured data sources import correctly	1. Click on Configuration > Data Sources 2. Check for data sources	Data sources should be listed from initial setup.	Displays current data sources	Pass
FN_18	Verify the configuration of data sources	Check configured data sources add and display correctly.	1. Click on Configuration > Data Sources 2. Click on add data source 3. Select source to add (for this example, use TestData DB) 4. Name the data source 5. Click Save & Test 6. Click on Configuration > Data Sources	Data source should be added and listed in Data Sources.	Displays message "Data source is working"; New imported data source is listed in Data Sources under Configuration section	Pass
FN_19	Verify the configuration of data sources	Check configured data sources can be removed correctly.	1. Click on Configuration > Data Sources 2. Click on data source to edit. 3. Click on Delete button	Data source should be removed from listed in Data Sources.	Displays message "Data source deleted" Data source is removed from listed Data Sources under Configuration section	Pass
FN_20	Verify data displays in Grafana dashboard	Confirm dashboard populates graph from data source	1. Click on Dashboards > Manage 2. Select Dashboard you would like to view. 3. Wait for data to populate (if necessary).	Data will successfully import and populate graph.	Data shows up and is plotted on a graph in a different color for each system.	Pass
FN_21	Verify instance filter in Grafana dashboard	Confirm instance filters correctly add and/or remove instance categories.	1. Click on Dashboards > Manage 2. Select a filter based on Starred or by Tag 3. Select Dashboard you would like to view. 4. Wait for instance to display (if necessary).	Data will only be shown for the selected instances.	Data shows up only for dashboards that meet filter criteria.	Pass
FN_22	Verify instance filter in Grafana dashboard	Confirm instance filters correctly add and/or remove instance categories.	1. Click on Dashboards > Manage 2. Select & apply a filter 3. Click the filter you would like to remove 4. Wait for instance to display (if necessary).	Data will only be shown for all available dashboards.	Data shows up only for all available dashboards.	Pass

Figure 17: Functionality testing pt. 3

Figure 18 displays the security test cases that we utilized. These cases would often test security functions and requirements that would be necessary to meet the NIST 800-37 and 800-53 requirements.

TEST ID	SCENARIO	DESCRIPTION	TEST STEPS	EXPECTED RESULT	ACTUAL RESULT	STATUS (PASS/FAIL)
SC_01	Verify the password requirements of Grafana dashboard	Verify passwords cannot be less than 6 characters.	1. Login to Grafana 2. Go to Profile > Change Password 3. Enter password 5 or fewer characters	Password change should fail.	Displays error message "New password is too short"	Pass
SC_02	Verify the password requirements of Grafana dashboard	Verify incorrect username / password displays 'Invalid Credentials' to protect existing user accounts	1. Enter incorrect username. 2. Enter valid password 2. Verify incorrect login displays	'Invalid Credentials' message should display	Displays error message "Invalid username or password"	Pass
SC_03	Verify the password requirements of Grafana dashboard	Verify incorrect username / password displays 'Invalid Credentials' to protect existing user accounts	1. Enter valid username. 2. Enter incorrect password 2. Verify incorrect login displays	'Invalid Credentials' message should display	Displays error message "Invalid username or password"	Pass
SC_04	Verify previous accessible pages cannot be accessed after logging out.	Confirm going back to previous pages does not display any data after logging out.	1. Access a dashboard in Grafana. 2. Log out of Grafana dashboard 3. Attempt to access previous dashboard	Attempt to display dashboard should fail; access denied.	Redirection fails; returns to login page.	Pass
SC_05	Verify secure communication	Confirm website loads when using HTTPS	1. Open web browser 2. Navigate to https://localhost:5001	Website login will load	Website loads; redirects to login page	Pass
SC_06	Verify secure communication	Confirm website will not load when using HTTP	1. Open web browser 2. Navigate to http://localhost:5001	Website login will not load; display error connection not secure	Website does not load; empty response	Pass

Figure 18: Security testing

Figure 19 shows the tasks we gave to external users while testing OmegaGraf. All test users would be given the scenario below and we would record whether they were able to complete the task (pass) or not.

TEST ID	SCENARIO	DESCRIPTION	STATUS (PASS/FAIL)
UA_01	Download OmegaGraf and complete setup using script.	Ensure all OmegaGraf components install with no errors	Pass
UA_02	Add a new data source to the Grafana dashboard	Import a new data source into the dashboard	Pass
UA_03	Add a new data source to the Grafana dashboard	Add a new vCenter to the dashboard	Pass
UA_04	Reset account credentials	Change password to a new one.	Pass
UA_05	Configure data source in Grafana dashboard	Apply a filter to only view (memory usage)	Pass

Figure 19: User acceptance testing

Figure 20 shows static code analysis results, as well as unit test results, for the OmegaGraf backend. This report is passed as the number of bugs and vulnerabilities is zero.

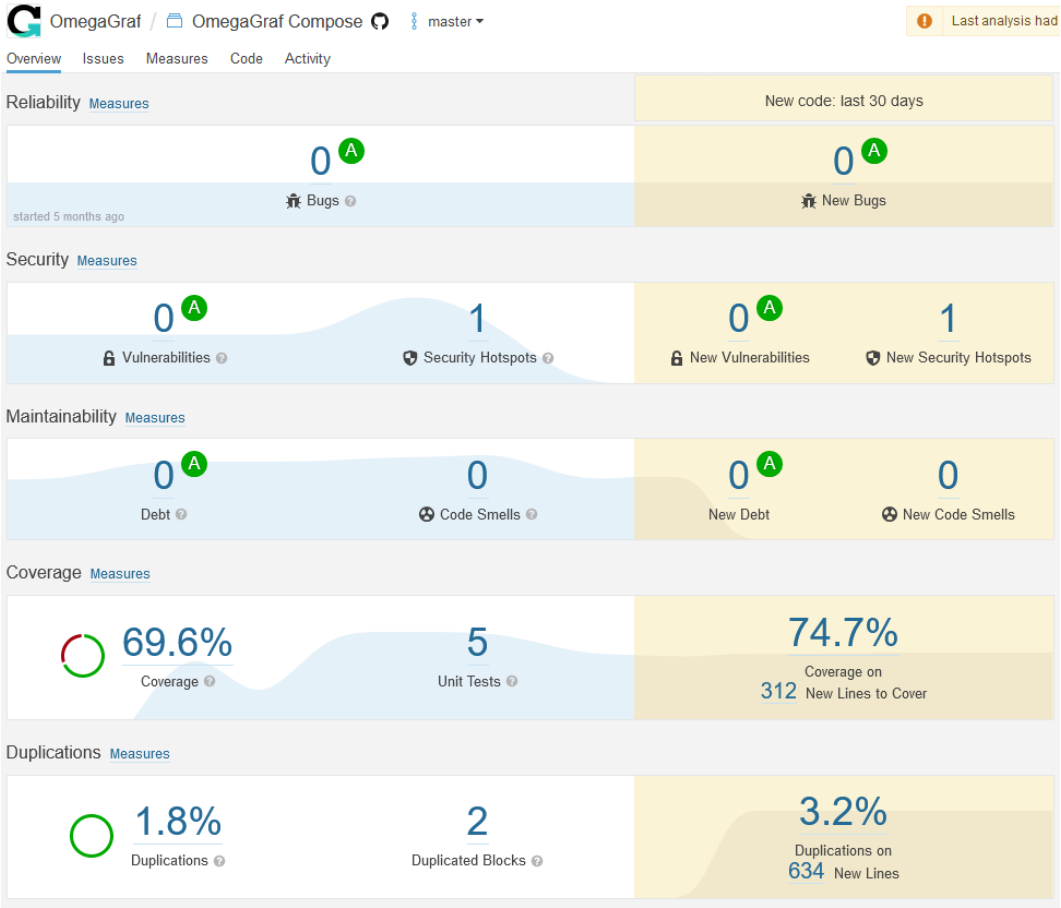


Figure 20: Backend code automated testing and analysis

Figure 21 shows static code analysis results for the OmegaGraf frontend. This report is passed as the number of issues is zero.

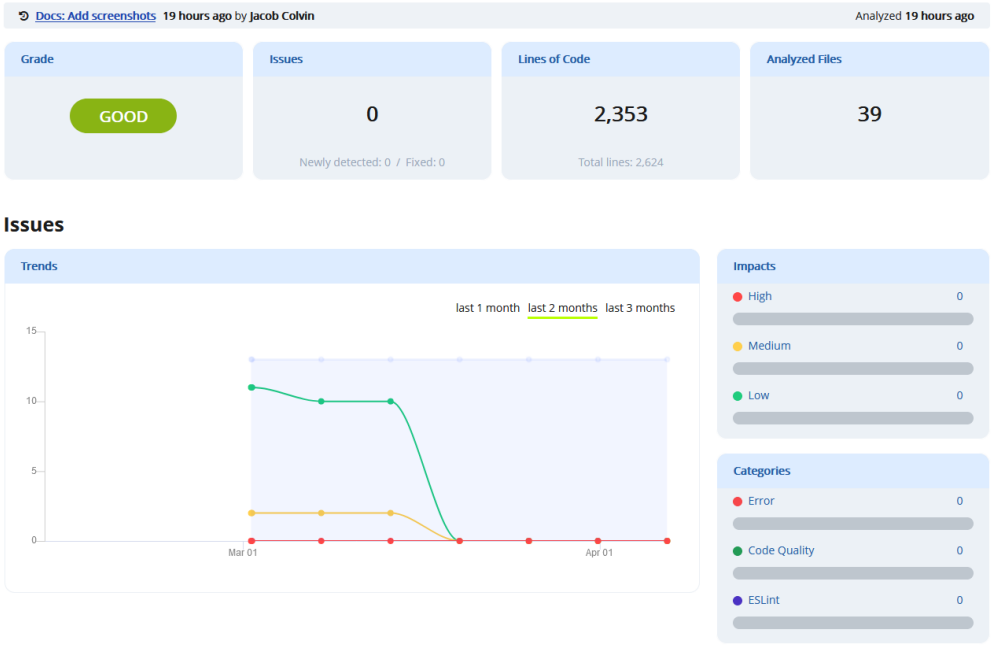


Figure 21: Frontend code automated analysis

2.7.6 Learnings During Testing

One major issue we have had is in using too many frameworks for testing. Some of this is unavoidable, given that OmegaGraf is a full-stack application consisting of shell scripts, multiple Docker containers, an API, and a UI. However, we could have aggregated more items into Azure DevOps. Just for instance, our backend uses TravisCI for automated testing, which in our experience has simply been an inferior version of Azure DevOps. TravisCI has the same functionality in terms of automating builds and tests, however it lacks a lot of management surrounding the overall application. Although we appreciated the major capabilities Azure DevOps offered, our setup with TravisCI remained the way we do automated testing due to familiarity.

2.8 Budget

To calculate our budget, we must first understand the cost of materials and cost of labor. Because the application itself uses exclusively free and open-source software, the software cost is zero. The hardware cost, however, is more complex; because the software is hosted locally on the user's hardware, the cost of hosting varies per environment. While the entire vCenter environment is necessary, it is a sunk cost and is not influenced in any way by OmegaGraf. There is a small upkeep cost, due to the increased power consumption of running an additional VM. However, this is impossible to estimate due to varying costs of power, as well as Ohm's Law, which dictates that power is equal to capacitance times voltage squared (Intel 2004). Essentially, upkeep of our VM is dependent on the existing utilization of the host (see Figure 22).

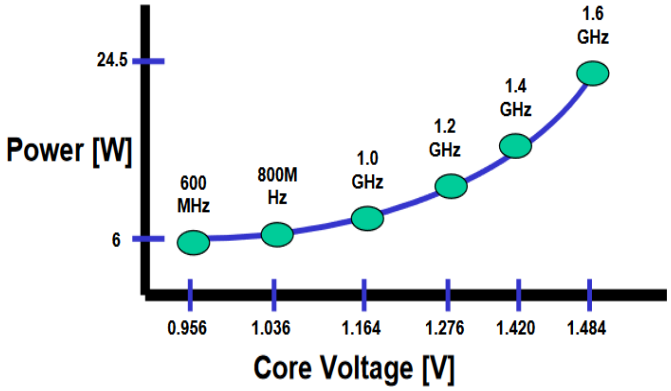


Figure 22: Power and processing (Intel 2004)

Although actual labor costs of the project are zero due to the development being a part of our team's Senior Design project for a Bachelor of Science degree in Information Technology, we can estimate the cost if it were developed by paid developers. Our team consists of three developers working 18 hours per week over a period of 30 weeks.

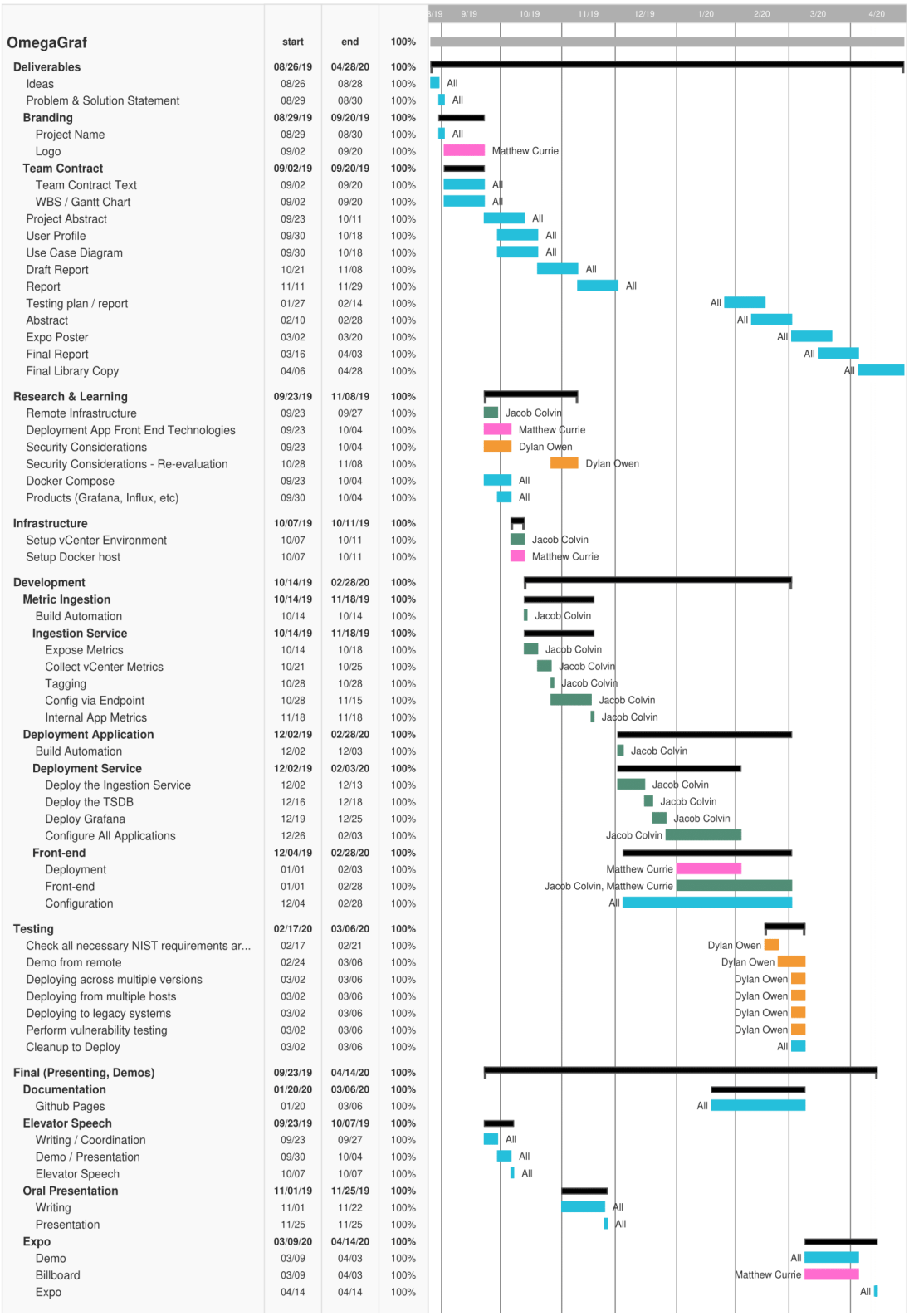
According to ZipRecruiter, the median hourly wage of a developer in Cincinnati is $40/hour (ZipRecruiter 2019). These values are used to estimate a salary figure, which can be seen in Table 1.

Category	Item	Description	Estimated Cost	Actual Cost
Hardware	vCenter Server	End user's vCenter server(s)	Previously Incurred	Previously Incurred
Hardware	Linux VM	Machine to host OmegaGraf containers	*	*
Software	Telegraf	Open-source software	$0	$0
Software	Prometheus	Open-source software	$0	$0
Software	Grafana	Open-source software	$0	$0
Software	Docker (Community)	Open-source software	$0	$0
Labor	3x Developers	Wages (theoretically paid to developers)	$64,799.99	$0

Table 1: Project budget

2.9 Gantt Chart

Figure 23 displays the schedule used during the development of OmegaGraf.



OmegaGraf	start	end	100%
Deliverables	**08/26/19**	**04/28/20**	**100%**
Ideas	08/26	08/28	100%
Problem & Solution Statement	08/29	08/30	100%
Branding	**08/29/19**	**09/20/19**	**100%**
Project Name	08/29	08/30	100%
Logo	09/02	09/20	100%
Team Contract	**09/02/19**	**09/20/19**	**100%**
Team Contract Text	09/02	09/20	100%
WBS / Gantt Chart	09/02	09/20	100%
Project Abstract	09/23	10/11	100%
User Profile	09/30	10/18	100%
Use Case Diagram	09/30	10/18	100%
Draft Report	10/21	11/08	100%
Report	11/11	11/29	100%
Testing plan / report	01/27	02/14	100%
Abstract	02/10	02/28	100%
Expo Poster	03/02	03/20	100%
Final Report	03/16	04/03	100%
Final Library Copy	04/06	04/28	100%
Research & Learning	**09/23/19**	**11/08/19**	**100%**
Remote Infrastructure	09/23	09/27	100%
Deployment App Front End Technologies	09/23	10/04	100%
Security Considerations	09/23	10/04	100%
Security Considerations - Re-evaluation	10/28	11/08	100%
Docker Compose	09/23	10/04	100%
Products (Grafana, Influx, etc)	09/30	10/04	100%
Infrastructure	**10/07/19**	**10/11/19**	**100%**
Setup vCenter Environment	10/07	10/11	100%
Setup Docker host	10/07	10/11	100%
Development	**10/14/19**	**02/28/20**	**100%**
Metric Ingestion	**10/14/19**	**11/18/19**	**100%**
Build Automation	10/14	10/14	100%
Ingestion Service	**10/14/19**	**11/18/19**	**100%**
Expose Metrics	10/14	10/18	100%
Collect vCenter Metrics	10/21	10/25	100%
Tagging	10/28	10/28	100%
Config via Endpoint	10/28	11/15	100%
Internal App Metrics	11/18	11/18	100%
Deployment Application	**12/02/19**	**02/28/20**	**100%**
Build Automation	12/02	12/03	100%
Deployment Service	**12/02/19**	**02/03/20**	**100%**
Deploy the Ingestion Service	12/02	12/13	100%
Deploy the TSDB	12/16	12/18	100%
Deploy Grafana	12/19	12/25	100%
Configure All Applications	12/26	02/03	100%
Front-end	**12/04/19**	**02/28/20**	**100%**
Deployment	01/01	02/03	100%
Front-end	01/01	02/28	100%
Configuration	12/04	02/28	100%
Testing	**02/17/20**	**03/06/20**	**100%**
Check all necessary NIST requirements ar...	02/17	02/21	100%
Demo from remote	02/24	03/06	100%
Deploying across multiple versions	03/02	03/06	100%
Deploying from multiple hosts	03/02	03/06	100%
Deploying to legacy systems	03/02	03/06	100%
Perform vulnerability testing	03/02	03/06	100%
Cleanup to Deploy	03/02	03/06	100%
Final (Presenting, Demos)	**09/23/19**	**04/14/20**	**100%**
Documentation	**01/20/20**	**03/06/20**	**100%**
Github Pages	01/20	03/06	100%
Elevator Speech	**09/23/19**	**10/07/19**	**100%**
Writing / Coordination	09/23	09/27	100%
Demo / Presentation	09/30	10/04	100%
Elevator Speech	10/07	10/07	100%
Oral Presentation	**11/01/19**	**11/25/19**	**100%**
Writing	11/01	11/22	100%
Presentation	11/25	11/25	100%
Expo	**03/09/20**	**04/14/20**	**100%**
Demo	03/09	04/03	100%
Billboard	03/09	04/03	100%
Expo	04/14	04/14	100%

Figure 23: Gantt chart

2.10 Problems Encountered

On the web UI side of development, one of the roadblocks encountered was the initial inexperience Matt Currie had with React development. Although he had experience with web framework development, overcoming the syntax of React and JSX made the initial push a bit hard to get through. Luckily, with some trial and error, a Codecademy course and a consultation with a React developer friend, we were able to get through.

We had several more technical issues that would appear from time to time, mostly related to choices we had made early in the design process. For the most part, these were relatively simple to address and could be easily mitigated by small tweaks or workarounds. We had a more difficult time working around one race condition that appeared, mostly because it only manifested with certain hardware. Luckily, by further developing our logging capabilities, we found that our issue was occurring around Grafana configuration. These types of issues can be difficult to identify, but once pinned down, we were able to add some additional logic to prevent these circumstances.

2.11 Future Recommendations

With essentially any product of significance, there are always processes that could have been improved in retrospect. For OmegaGraf, this was likely in the area of automated testing. At the conclusion of the spring semester, OmegaGraf is lacking in this area, and contains only a few of what we would classify as integration tests, which only covered some of the most important aspects of backend functionality. Additionally, the OmegaGraf API and UI have no automated tests. Overall, this has added significant pain to our release

process, as each build needs to be reviewed manually. One thing that has been pointed out to us is that negligence in this area will likely affect our ability to collaborate with other developers, as they will be relying on our testing to ensure that any of their code changes are working as intended. For this reason, moving forward, automated testing will be one of our main priorities.

Proving and documenting the NIST compliance was a large task that seemed very daunting to take on. Since testing is also more towards the end of a project's development cycle, it can seem like a time crunch. We found that a big thing that helps is to do research in preparation. For Dylan, it would have helped to go over the NIST guidelines and documentation more early on and keeping notes about potential issues that may arise.

While testing, there would sometimes be errors when targeting packets during vulnerability testing. A lot of time was spent having to look up the related syntax errors and noting them for future reference. Alongside this, there was often troubleshooting and dealing with false positives that would occur while performing vulnerability testing. By going over the security documents earlier and creating tests based on the NIST compliance checklist, we could have allowed for more time to plan everything out and make sure every aspect is tested, working, and secure.

3.0 Conclusion

3.1 Lessons Learned

We initially planned on using more custom products but ended up bringing in Telegraf to our tech stack instead. We learned not to overly rely on ourselves to create everything by hand, and that using other already existing open-source products is completely fine, if it improves the project's overall quality, and saves time and money.

One other lesson learned was the importance of having good logging built into products. When testing weekly builds, Matt Currie and Dylan Owen would occasionally have trouble with deploying new versions of the OmegaGraf environment. However, with logging fully developed, troubleshooting these problems became a much faster and simpler task.

3.2 Abilities / Skills Developed

For the preliminary development of the frontend, Matt learned the basics in React development. Since he was originally primarily experienced with infrastructure work, his experience was primarily in shell scripting, not JavaScript or TypeScript. He used this project as an opportunity to research into React.

During the second half of the semester, we got a lot more into testing the application's security. Dylan had been familiar with port scanners and vulnerability exploiting tools from what he had learned in his college classes, but when it came to scripting certain tasks, there was a lot of additional troubleshooting that went into it. Using Metasploit's pre-configured exploits would work well, but creating custom payloads tailored to target

the application proved to be much more difficult. Many times, false positives would be displayed due to a minor misconfiguration or other issues would occur due to incorrect syntax. Using learning resources such as Lynda and other educational videos, Dylan was able to learn a lot about penetration testing.

While Jacob Colvin was already familiar with much of the technology stack, he had never acted as a developer in a team, working on a structured project. He found that his development work greatly benefited from having design documents, such as style guides and mockups, fully defined before starting with more technical tasks. By adhering to agreed-upon requirements, regularly delivering updates, and making changes where needed, the efficiency of development can be drastically increased.

3.3 Improvements Since Fall

The following list describes some of the largest features we have added to OmegaGraf since the fall semester:

- ➔ Home page that allows users to launch several guided setups.
- ➔ Detailed deployment status page that dynamically links to services.
- ➔ Security features to comply with NIST 800-37 & 800-53.
- ➔ Fully typed configuration syncs between the API and UI.
- ➔ Self-monitoring of OmegaGraf's components.
- ➔ Seven fully functional, pre-built dashboards for OmegaGraf's data.
- ➔ Command line arguments and documentation for advanced users.
- ➔ Multi-level logging system.
- ➔ Advanced, repeatable, and helpful input validation.

At the conclusion of the fall semester, OmegaGraf had an MVP ready. However, we still made very significant progress on the solution, mostly regarding the UI. In fact, the majority of our code commits were made during the spring semester of 2020, amounting to ~250 of around 350 total. In addition to the listed improvements, we have made numerous fixes in the areas of design, compatibility, and stability, as well as improvements to the overall design and user experience of OmegaGraf.

3.4 Learnings from Presentations

Likely the most daunting part in preparing for our presentations was crafting a demonstration. We had a very short amount of time to demonstrate a solution that, while simple at its surface, is very technically complex. With complexity comes some uncertainty, despite our general confidence with the stability of the solution. Additionally, we cannot fully demonstrate our Grafana dashboards without at least several minutes of data loaded. We initially proposed using data from a previous installation, however this caused numerous issues as our test environments were re-generated, disassociating all existing time-series. Additionally, keeping an environment spun down for longer than five minutes will mark metrics as stale.

Our solution to this problem was two-fold. First, we had multiple members with an environment fully ready to demonstrate, in case of any unforeseen technical difficulties. Second, we used a two-stage approach for our demonstration. We used one VM that was ready for an OmegaGraf deployment, in addition to a second VM with OmegaGraf already

deployed. This way, we could demonstrate the setup and deployment process, before switching to a fully developed instance with several hours of data loaded.

3.5 Closing Remarks

Creating OmegaGraf has been a fantastic learning experience. At this semester's close, we believe we have developed a product that will truly be useful to many people in the IT industry. We are incredibly proud of the results of all our efforts and will continue to maintain and make improvements to OmegaGraf after graduation.

4.0 References

Diaz, Jessica, Jorge E. Perez, Miguel A. Lopez-Pena, Gabriel A. Mena, and Agustin Yague. 2019. "Self-Service Cybersecurity Monitoring as Enabler for DevSecOps." IEEE Access 7: 100283-100295.

“How Much Do Software Developer Jobs Pay per Hour?” *ZipRecruiter*. Accessed November 10, 2019. https://www.ziprecruiter.com/Salaries/How-Much-Does-a-Software-Developer-Make-an-Hour.

Intel. “Enhanced Intel SpeedStep Technology for the Intel Pentium M Processor.” White Paper, no. 301170 (March 2004). http://download.intel.com/design/network/papers/30117401.pdf.

“Prepare for Your Containerization Project.” Smarter with Gartner. Accessed October 13, 2019. https://www.gartner.com/smarterwithgartner/prepare-for-your-containerization-project/.

Quinn, Stephen D, Murugiah Souppaya, Melanie Cook, and Karen Scarfone. “National Checklist Program for IT Products - Guidelines for Checklist Users and Developers.” NIST Special Publication 800-70. Revision4. (February 2018). https://doi.org/10.6028/nist.sp.800-70r4.

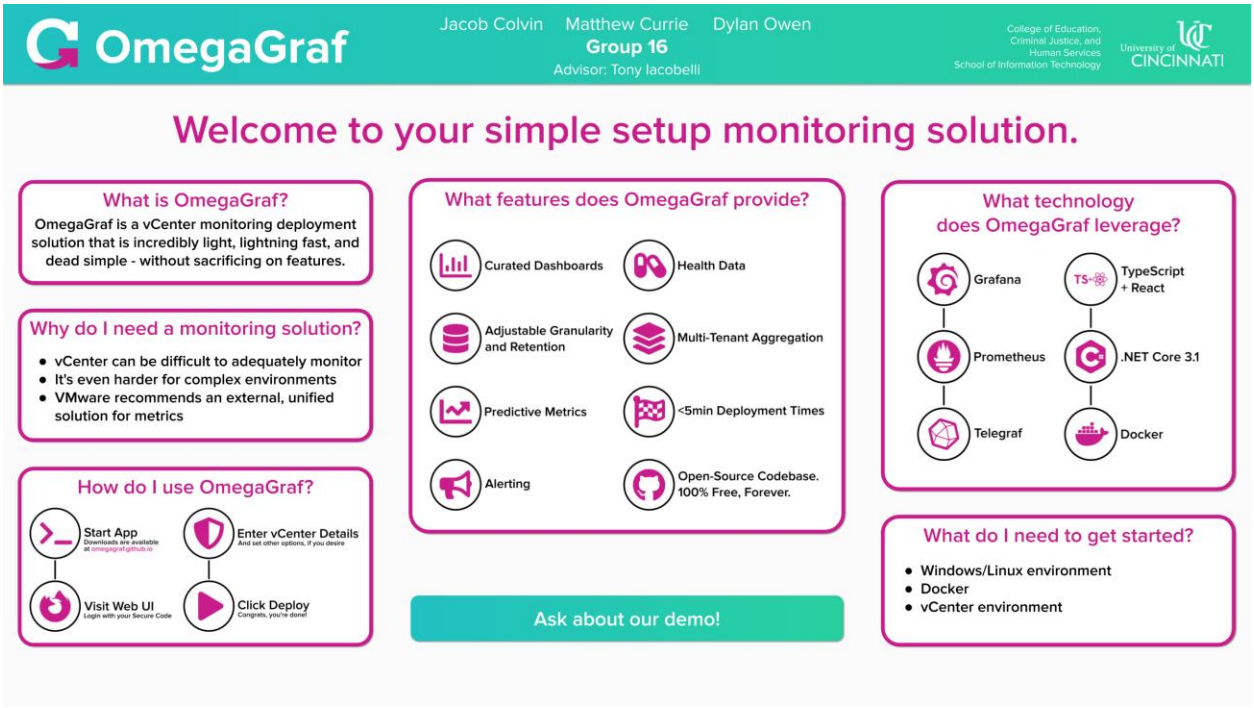
“Reading Room.” SANS Institute: Reading Room - Analyst Papers. Accessed October 13, 2019. https://www.sans.org/reading-room/whitepapers/analyst/guide-virtualization-hardening-guides-34900.

“r/Sysadmin - SexiGraf Is Now Available.” reddit. Accessed October 13, 2019. https://www.reddit.com/r/sysadmin/comments/3nl1ju/sexigraf_is_now_available/.

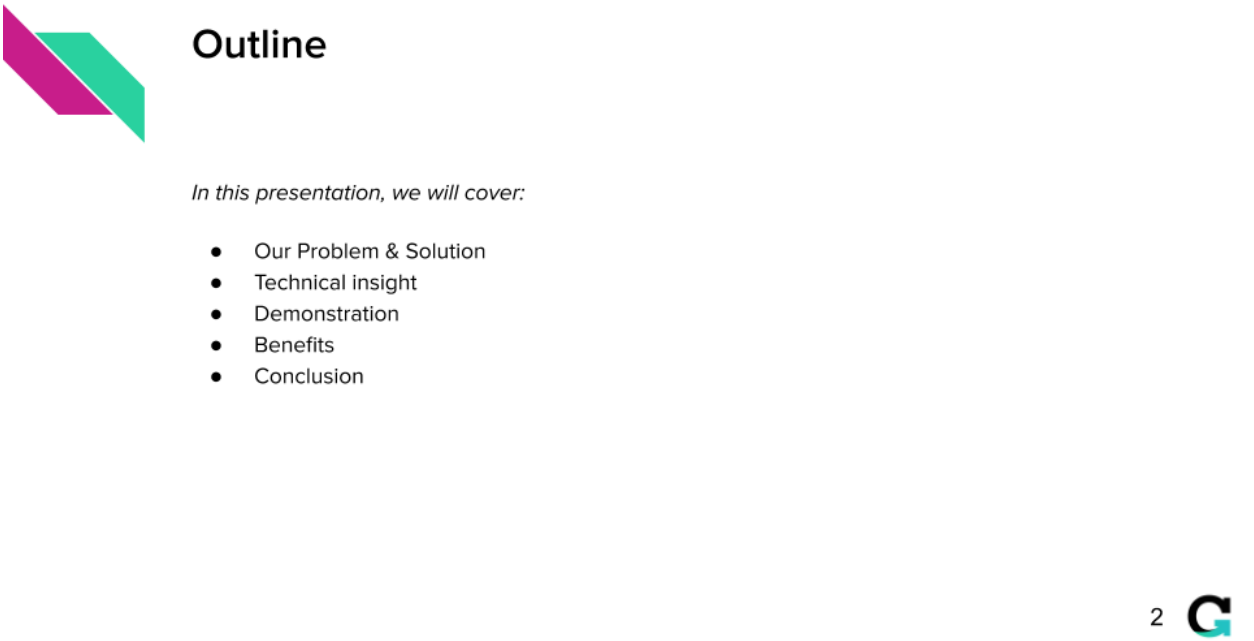
“Security Hardening Guides - VMware Security.” *VMware*. Last modified October 7, 2019. Accessed October 13, 2019. https://www.vmware.com/security/hardening-guides.html.

5.0 Appendix

Appendix A: IT Expo Poster



Appendix B: Final Release Presentation



Problem & Solution

The Problem

- vCenter can be difficult to adequately monitor
- It's even harder for complex environments
- VMware recommends an external, unified solution for metrics

Our Solution → OmegaGraf - Fast & Easy vCenter Monitoring Deployment

+ Deploys a clean, simple, and modular monitoring solution
+ Requires little interaction and minimal expertise
+ Begins monitoring all your environments within just 5 minutes

3

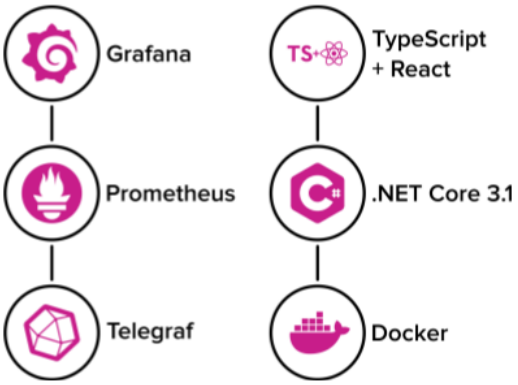
Technology

OmegaGraf configures and deploys a containerized software stack.

Through a web interface, OmegaGraf:

- Generates a configuration
- Sends data to a docker-ce interface
- Orchestrates container deployment

OmegaGraf and all the technologies it uses are all completely free and open-source on GitHub!



4

Benefits

- Open source - completely free
- Can save admins hours and even days of work
- Plug and play with your existing solutions
- Its simplicity can save organizations from not having monitoring due to the time required to set up

OmegaGraf

Demonstration

Conclusion

Download and give it a try!

https://OmegaGraf.github.io

OmegaGraf

Thank You

Appendix C: Source Code

Repository	**Link**
OmegaGraf	https://github.com/OmegaGraf/OmegaGraf
OmegaGraf - Website	https://github.com/OmegaGraf/omegagraf.github.io
vCenter Simulator (vcsim)	https://github.com/OmegaGraf/docker-vcsim