Scaling Up with New Technologies: A Case Study in the Application of Containerization and Microservice Technology

A Major Qualifying Project report:

Submitted to the Faculty of

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the degree of

Bachelor of Science

by

Brittany Goldstein

Date: March 23, 2021

Project Sponsor: Johns Hopkins Applied Physics Laboratory

Sponsor Liaison: Dr. Jason Miller

Approved:

Professor George Heineman, Major Advisor

Abstract

Johns Hopkins University Applied Physics Lab (JHU/APL) sponsored this project with the goal of identifying means for increasing scalability of processing, through the targeted use of technologies such as containerization and microservices, producing a guide to streamline their continuous integration and delivery processes. This project looks into best code practices using these technologies and associated strategies such as autonomic computing to develop code more efficiently and of higher quality.

I. Introduction

A software project cycles through seven main stages: ideation, defining, designing, development, testing, deployment, and maintenance [1]. As projects expand in size and intricacy, this cycle can be slowed down by increasingly complex testing and debugging, resulting in the need for higher volumes of communication between developers and increasing the time from ideation to deployment. At the same time, a project's popularity and customer base tends to grow with the more features it has to offer, increasing the demand to deploy new updates as the development cycle slows. Companies use a variety of strategies to maintain this cycle's efficiency under increased pressures.

Johns Hopkins University Applied Physics Lab (JHU/APL) has sponsored this project with the goal of of identifying means for increasing scalability of processing, through the targeted use of technologies such as containerization and microservices. As a trusted technical expert for the government, JHU/APL works on some of the most critical problems facing our nation [2]. Developing and deploying high-quality solutions to these problems in a timely fashion is important to achieving their mission. The objectives for this project are:

1) Research a number of container-based technologies and select one for extensive investigation for potential use at JHU/APL.
2) Investigate best practices for Continuous Integration (CI) to maximize productivity and increase quality in software development projects.
3) Investigate best practices for Continuous Delivery (CD) to increase scalability in software development projects.
4) Develop a case study using a "plain vanilla" application whose architecture is closely related to JHU/APL's existing software applications.

5) Educate software engineers on the CI/CD process, which can have a steep learning curve, especially for domain-specific engineers.
6) Adapt the plain vanilla application to allow for autonomic computing on Amazon Web Services.

II. Background

Best practices for software development and deployment have evolved quickly since the invention of the computer and the internet. Microsoft deployed their first software on a CD-ROM in 1987 [3]. This required companies to upload their software to the CD-ROM and distribute the CD-ROM to stores, and then users to physically go to the store and purchase it. These software licenses were called shrink-wrap licenses, because by opening the plastic packaging around the CD-ROMs, users would automatically accept the terms of the software's license [4]. Now, the majority of computers no longer include a CD-drive to read CDs and install their contents, as software deployment has become more sophisticated.

The advancement of the internet has made it increasingly easy to deploy and access software. This has aided the rise in popularity of Unix's Tape Archive (TAR) format, which was released in 1979 [5]. A tarball is a file containing other files that can be compressed into many formats such as .tar.gz or .tgz. This is a common format for companies to distribute their code over the internet. When a user decompresses the tarball, they then must build and install the source code on their machine prior to the software's use [6]. Since Unix developed this tool, Linux, MacOS, and Windows have all developed or integrated additional tools to allow users to extract and compress tar files [7].

A. Virtual Machines

In the 1960's, IBM began working on virtualization to make deployment easier for their customers [8]. Over time, this has evolved from user-level virtualization, to application virtualization, to hardware virtualization. Virtual machines allow companies to run isolated instances of different operating systems on the same machine's hardware, saving them the cost of buying multiple machines with different hardware configurations. The user can scale the amount of resources allocated to the virtual machine based on its needs. This allows software developers to run their code in a variety of different environments. Today, users can deploy full virtual machines to other machines or the cloud for development, testing, and running purposes.

B. Containers

Recently, many companies have shifted their projects to containers due to the portability, modularity, and scalability they provide, which make them ideal for deployment [9]. Containers hold everything an application needs to run, so that the user does not need to install anything additional on their host machine. This allows developers to easily deploy containers to other machines without worrying about the host machine's configurations. Additionally, developers can model the environments where their applications will be deployed for testing and debugging purposes.

Compared to virtual machines, containers require less resources and have a faster startup time, increasing project efficiency [10]. This is because each container uses the host's operating system instead of creating an operating system for each virtual machine. Although containers' smaller size make them easier to transport and faster to startup, it also makes them more vulnerable to security threats. Containers are less isolated from the rest of the host machine, so if the host machine or another container becomes infected, the chances of the container becoming infected are high.

There is an Open Container Initiative (OCI) to create industry standards for container formats and run-times with which most containerization software systems either meet or are working towards becoming compatible [11]. The OCI is sponsored by Docker and CoreOS rkt, two of the leading containerization software applications, and promotes a) the ability to run an image (the precursor to a container) without additional parameters and b) standard image specifications. The image specifications include an image manifest, file system serialization, and an image configuration.

1) Microservices: Many containerization software applications encourage users to only assign a single responsibility (and ideally a single process) to a container [12]. This requires applications with multiple responsibilities to be restructured into multiple containers communicating with one another through an architecture called microservices. Microservices can be highly beneficial in large projects due to the ability to deploy sections of the project without making changes to the project in its entirety [13]. This makes the application easier to maintain and test and reduces extraneous communications between development teams working on separate features. However, this does require more work in the initial stages to develop well-defined individual application program interfaces (APIs) for intercommunication between microservices. Without well-defined APIs, microservices can become heavily co-dependent on one another, difficult to maintain, or highly individualized - all things the structure is should minimize [14].

In some cases, it is beneficial to contain multiple processes in a single container, called a fat container [15]. This includes when:

- multiple processes communicate through shared memory, and
- one process either controls or produces one or more other processes.

2) Cloud-Based Computing Services: There are many cloud-based computing services available for use by developers that offer a Container as a Service (CaaS) design [16]. CaaS design allows customers to host, manage, and run their containers on a cloud-based system. Due to the cloud-based nature, the hosting services can scale up based on a project's needs [17]. If they need less resources, the company would not have to pay for those resources.

C. Continuous Integration and Delivery (CI/CD)

Continuous integration (CI) is the process by which developers automatically build, test, and merge their code changes [18]. This allows developers to simultaneously work on different features of the same software while maintaining an up-to-date working version of the code.

Continuous delivery (CD) is the ability to deploy code to users quickly despite changes to the system [19]. This requires that each project have a substantial set of unit tests for automatic code testing at regular intervals - typically daily or when new code is pushed - to ensure that new code does not break the existing systems. Although producing the unit tests can come at a higher initial labor cost to developers, they tend to save more time long term by finding bugs as they occur. Ideally, unit tests should cover 70% to 80% of the code; however, this percentage is dependent on the software's subject matter and the risk associated with a bug in the code [20]. Code coverage also does not indicate whether the unit tests are testing cases indicative of the variety of cases that the software will face in the field, so it is important that multiple developers knowledgeable of the code aid in the development of the unit tests to ensure that all edge cases are covered. Otherwise, the unit tests will not perform their function. Only when code has been thoroughly tested, should it move from the development branch to the deployment branch. These two branches ensure that there is always a deployable version of the code.

D. Scalability of Processing

If implemented correctly, containers, microservices, cloud-based applications, and continuous integration and delivery can all help increase scalability of processing. Scalability of processing refers to the ability

of a company, its systems, and project architecture to keep up with the growth of a software project and its subsequent demands. This can be divided into internal growth and external growth.

1) Internal Scalability: Internal scalability is the ability of the developing team and its systems to handle the increased burden created by growth in software demand. In cases where this requires adding new developers to the team, they should be able to quickly begin working on the software systems without experiencing a steep learning curve or having to undergo extensive training, which removes other developers' focus away from their assigned tasks.

2) External Scalability: External scalability is the ability to deploy the project to as many customers as needed to meet demand without placing additional strain on the developers or their hardware's capabilities. This can be achieved through autonomic computing, a field of development where computers manage themselves and their resources, thereby taking the strain off of developers [21]. Autonomic computing includes automatically responding to varying amounts of demand by varying resource usage accordingly, detecting and preventing threats to the system's security, and reconfiguring the system based on new information and optimal performance.

III. Methodology

This project was divided into three main phases:

1) The first phase involved investigating and selecting both a container-based technology and cloud-based technology. These selected technologies were used in later phases to make recommendations to the sponsor on how to adapt software projects to a container-based approach.
2) The second phase of the project focused on creating a simple "plain vanilla" project to mimic the architecture of some of JHU/APL's projects. We applied a container-based structure to the existing architecture using the technologies chosen in phase one to learn how to best adapt JHU/APL's software. This involved developing recommendations on how to containerize software projects to ease the learning curve.
3) The third phase originally was supposed to focus on researching best practices for CI/CD and applying them to the plain vanilla project to streamline development and deployment. However, discussions with another team at JHU/APL revealed that they had been working to develop more effective CI/CD practices during the same timeline as this project. In order not to duplicate

their work, phase three shifted to focus on researching and applying best practices for autonomic computing, using Amazon Web Services (AWS) and the plain vanilla project as a model.

Further details about how each phase was completed can be found below.

A. Phase 1: Comparison of Container-Based Technology

This project compared the technology available on the market for containerization and cloud-based API services to inform its recommendations to the sponsor. As different technologies offer different features, we wanted to ensure that the software we chose to apply to the plain vanilla project best met the sponsor's needs.

1) Container-Based Technology: The four leading container technologies on the market are Docker, CoreOS rkt, Apache Mesos, and LXC [22]. However, LXC only creates system containers, which are containers that host the entire system including the operating system, similar to a virtual machine [23]. As the sponsor is looking for an application-based container, LXC was eliminated from consideration.

Further research into Apache Mesos revealed that although it has some overlapping capabilities with Docker and Rkt, its primary purpose is for cluster management, not application-based container development [24]. Among the many workloads that Apache Mesos manages, it does have a container orchestration extension called Marathon that can run on its base framework. This allows users to simultaneously orchestrate Docker containers, Kubernetes, Java applications, distributed data services and much more. If we attempted to use Apache Mesos to meet the sponsor's needs, we would ultimately end up needing to use Docker as well.

The two remaining container technologies, Docker and CoreOS rkt, were compared based upon market popularity, image format, container security, OCI compliance, microservice capabilities, continuous integration and delivery features, ease of debugging, and additional features/partnerships with other software applications for development [Fig. 1, 2]. Docker was chosen for further investigation and use in the remainder of this project due to its:

- **Popularity:** It was used to create the majority of containers in 2018 and many large companies use it to implement their software [22] [25]. This suggests that it has an easier learning curve or higher quality features that make it more desirable.

FEATURES	DOCKER	COREOS RKT
Popularity	Comprised 83% of Containers in 2018 [22]	Comprised 12% of Containers in 2018 [22]
Major Customers	PayPal, eBay, BBC News, Spotify, Lyft, Expedia, Groupon, GE Appliances, and Uber [25]	CA Technologies, Verizon, Viacom, Salesforce.com, and DigitalOcean [25]
Image Format	Docker [22]	Docker and appc [22]
Security	1) 'docker scan' command uses snyk to identify security vulnerabilities and notify the user of their location and magnitude [26]. 2) Containers have reduced root capabilities, to prevent an attacker from infecting the host through the container [27]. 3) Users can configure Docker to only run signed images.	Rkt signs and verifies containers by default for an extra layer of security [28]. Additionally, all containers are fetched as a non root user, so if a container was hacked, the hacker would not have have root permissions on
OCI Compliant	yes [29]	yes [29]
Microservice Orchestration	The docker-compose feature allows users to create a docker-compose.yml file to configure multiple containers as services which depend on each other [30]. If needed, users can create multiple YML files that can be built in a hierarchical fashion.	Rkt groups containers into pods which share resources and information. The user can configure the pods in a hierarchical structure [28].
Continuous Integration/ Continuous Delivery	Docker allows the user to configure automated builds, GitHub and Bitbucket integration, automated tests, webhooks, and slack notifications [31].	None
Debugging Containers	Node JS, Python, and .NET Core applications in Docker containers can be debugged using Visual Studio's debugger [32]. If the user's application is not in one of these languages, they can use command line operations to pause the container and execute commands within it to debug [33].	Rkt provides commands for users to read container logs through systemd or stdout/stderr [34].

Fig. 1. Comparing Docker and CoreOS rkt (part 1)

Fig. 2. Comparing Docker and CoreOS rkt (part 2)

- **Security:** Since JHU/APL does sensitive work, it is important for developers to be able to secure the containers from outside attacks. Docker containers have many design specifications to ensure this including [27]:
  - Kernel namespaces to isolate processes in a container from the host machine and other containers.
  - Individualized network stacks so that each container only has access to its own ports and interfaces.
  - Control groups which monitor resource distribution on all containers and will attempt to block a Denial of Service attacks if they notice a spike in resource usage.
  - Reduced root privileges in containers so that if an attacker manages to infect a container, it would be more difficult for them to breach the host machine and cause harm. There is a set of 14 root privileges that are provided by default to 'root' users in a container, any desired

additional features must be configured.

  - Docker Content Trust Signature Verification which allows developers to sign their images and clients to verify that images they run are signed by the developer. By default, images are not signed; however, the image signature can be configured and enabled in the dockerd binary and daemon.json.

Additionally, Docker's partnership with Snyk allows developers to pinpoint the location of vulnerabilities in a container and their severity, so that vulnerabilities can be triaged [26]. Docker is also compatible with security applications such as TOMOYO, AppArmor, SELinux, GRSEC, and PAX and is the first container application to gain the Security Technical Implementation Guide Certification from the Defense Information Systems Agency [27] [42]. This guide teaches developers how to strengthen their containers to protect against malicious attacks. Through the guide's certification process, the government has verified that Docker is safe to use on JHU/APL's government contracts, if they choose. However, Docker is not immune to security vulnerabilities. Docker allows containers to share a file system with their host machine. This is controlled by the Docker daemon. To prevent a malicious user from gaining access to this gateway, Docker recommends that only trusted users be given access to the Docker daemon. Additionally, the REST API endpoints used by Docker command line to communicate with the Docker daemon uses a Unix Socket, whose access permissions can be reconfigured. Docker also requires that the REST API endpoints are secured with HTTPS and certificates.

- **Support for Microservice Architecture:** Docker aids multi-container application development through the docker-compose feature. Docker-compose allows users to define a series of services in a docker-compose.yml file [30]. These services each have their own Dockerfile, container, environment, and ports. Using the docker-compose "up" command, Docker builds services based on their dependent containers. Docker also allows multiple docker-compose.yml files in a project, for different stages of development or different environments. These can be built in a hierarchical manner to allow for one base application that can be customized.
- **CI/CD Features:** Docker Hub has several CI/CD capabilities built-in, including automatic builds, build testing, slack integration, and webhooks [31]. Users can configure webhooks to trigger a new image build, or any other event of their choosing, upon receiving a code push [43]. If this image

build is successful, it can then automatically be pushed to the Docker Hub registry [44]. There is an 'Autotest' feature on every Docker Hub registry which, when enabled, runs user-made unit tests on the source code when a change to the repository is initiated [45]. These unit tests are configured as a service in a docker-compose.test.yml file. Additionally, users can connect their Docker Hub repository to a slack channel to receive automatic updates when a build passes or fails [46].

- **Ability to be Debugged:** Docker is capable of debugging NodeJS, Python, and .NET Core applications in containers using Visual Studio's debugger [32]. For applications written in other languages, Docker has several commands to aid a user in debugging an already running container, including [33]:
  - the attach command which outputs the live results of stdout within the container,
  - the exec command which allows the user to run additional commands,
  - the pause command which pauses a running container, and
  - the top command which provides information about all the processes running in the container.
- **Partnerships with Container as a Service Providers:** Docker's partnerships with Amazon Web Services (AWS) and Microsoft Azure make it easy for users to build and deploy containers to cloud-based hosting services in only one or two commands [36] [37]. Previously, users would have to provision spaces for their projects on the AWS and Microsoft Azure end prior to returning to the Docker command line and deploying their containers. This process is now automatic, increasing project efficiency.
- **Docker-Provided Resources:** Docker has put a lot of effort into creating many additional resources to aid developers using their core Docker Engine. Docker Hub, Docker Swarm, and Docker Cloud ease the process of storing, accessing, orchestrating, and hosting containers, freeing up developer time for coding.

2) Cloud-Based Technology: In addition to container-based technologies, this project examined cloud-based computing services on the market for hosting, running, managing, orchestrating, and deploying containers in a streamlined fashion. The three major cloud-based computing services on the market are Amazon Web Services (AWS), Google Cloud, and Microsoft Azure. However, Google Cloud was eliminated from consideration due to its incompatibility with Docker compared to the other two choices. Since Docker was already chosen as the container-based technology for further investigation, it is

FEATURES	AMAZON WEB SERVICES (AWS)	MICROSOFT AZURE
Major Customers	Duolingo, Samsung, GE, Cookpad, Intel, Snap, Intuit, GoDaddy, and Autodesk [48] [49]	eBay, Boeing, Samsung, GE Healthcare, BMW, and Travelocity [50]
Security	AWS's Identity and Access Management (IAM) allows users to create other users and groups and set their permissions to access AWS resources [51]. AWS also supports multi-factor authentication.	Azure Active Directory allows users to create different roles, each with specific permission relating to different container registries [52]. Azure does support multi-factor authentication.
Container Registry	AWS's Elastic Container Registry (ECR) supports Docker container storage, management, and deployment [53]. Repositories are organized using namespaces, making it easy for team collaboration. Additionally, ECR is integrated with a bunch of third-party tools including RedHat OpenShift, TigeraSecure, and WeaveCloud.	Azure's Container Registry supports building, storing, securing, scanning, replicating, and managing Docker and OCI supported images [54]. It includes automatic container building and patching when base images are updated or new code is committed and task scheduling for future builds/updates to images.
Container Orchestration Service	AWS's Elastic Container Service (ECS) + AWS Fargate is a serverless container orchestration service that allows users to manage container availability and scale [48]. It allows users to schedule jobs/containers based on resource demands and container availability requirements. It also allows users to test a new version of a service prior to switching out the old version.	Azure Container Instances allows users to run and deploy containers without managing servers [54]. It is able to provision more containers based upon resource and computation demands to the system.
Application Network	App Mesh coordinates network traffic between containers [55]. It collects traffic monitoring data and provides services to reroute traffic to manage loads and monitor the health and status of containers.	Service Fabric aids in the orchestration and scalability of multi-container projects and microservices [56]. It allows projects to be run from any machine and supports any language. It scales up to thousands of machines.
Kubernetes	Amazon Elastic Kubernetes Service (EKS) allows users to automatically run, manages, and scale their own Kubernetes. [57] Since EKS is a Kubernetes conformant, users do not have to install or operate a Kubernetes control plane.	Azure Kubernetes Service (AKS) supports server-less Kubernetes and integrated CI\CD [58].

Fig. 3. Comparing Amazon Web Services and Microsoft Azure

important that the cloud-based technology chosen is compatible. Although Google Cloud does allow Docker containers to be stored and deployed on its framework, the initial set up places a much higher burden on developers than AWS and Microsoft Azure. Developers must create a Compute Engine instance and install Docker on it, prior to uploading their Docker image [47]. In comparison, Docker's partnerships with AWS and Microsoft Azure allow the same steps to automatically be completed through a couple lines from Docker command line [36] [37].

The two remaining cloud-based computing services, AWS and Microsoft Azure, were compared based upon customer base, security, container registry, and container orchestration service [Fig. 3]. Our research revealed that there were no major differences in features between AWS and Azure in the areas relevant to this project. Both services had a comparable service in each of the categories analyzed that could be used to achieve the same effect. Therefore, we let the sponsor choose their preference. They chose AWS, because it has a longer standing reputation in the field (having been on the market for four more years than Azure) and the sponsor currently utilizes AWS for other related applications which may make the transition easier [59].

B. Phase 2: Plain Vanilla Project

JHU/APL works on many complex software projects, whose architecture we aimed to mimic with our plain vanilla project to create a base for adopting container-based technology. Through several iterative discussions with the sponsor, we decided upon a Python application that would read in JSON files representative of cars and return the car's average length [Fig. 4]. The application is user configurable and screens cars based upon a discrete feature, such as make, model, or color.


```
{"color":"blue","make":"toyota","model":"tundra","length_measurement_ft":
22.461144419949264,"length_measurement_uncertainty_ft":1.5,"width_ft":7,
   "speed_mph":35,"license_plate":"ABC1234","position":{"latitude":25,
  "longitude":25,"artificial_processing_time_sec":0.05,"time": "2020-10-05T12:34:45.789012"}}
```


Fig. 4. An Example Car JSON

1) Formulating the Project: Due to the limited time frame of this project, we tried to keep the plain vanilla project as simple as possible, while still including all the necessary features to ensure that it would be a representative case study. The sponsor's desired features included user configurable settings, data screening, state estimation, aggregation of results, output formatters, and a health and status monitor.

This application was first built as a series of processes and helper files running on the localhost machine. The application was broken up into multiple processes to aid in the division of the application into multiple services later on. For the purposes of this paper, the terms services and containers are synonymous. Communication between the processes was brokered by Apache ActiveMQ which was downloaded onto the local machine [60]. The files necessary for the application are:

- **amqsend.py:** A STOMP ActiveMQ producer. It connects to a specified host, port, and topic and sends the input message to all of the connected consumers on the same topic.
- **amqecho.py:** A STOMP ActiveMQ consumer. It listens for messages on a specified host, port, and topic and stores any incoming messages to a queue.
- **main.py:** This process continuously checks for two things (1) updates to the user specified configuration file and (2) new car JSON files in the queue. It then screens the car JSON files based on the user configurations and sends only cars that meet these specifications to the estimator.
- **data_extractor.py:** This file has helper functions that parse the user defined configuration file and car JSON files. It expects each line of the configuration file to be in the format of "<type> <specifier>" where type is either color, make, or model and specifier is along the lines of blue, Toyota, and Camry, respectively.
- **Car.py:** This is a Python class that stores the make, model, length, and color information for each car. It also stores the time each car enters a process, so that process latency can be tracked.
- **AvgCar.py:** Continuously receives car JSONs and calculates the average length of cars based on specified fields.
- **Output.py:** Continuously receives updates on the average length of cars in user specified fields and prints them out to the screen.

The components of this application were divided into services to be compatible with the container/microservice architecture [Fig. 5]:

- **Data Broker:** This project used a pre-made Docker container image containing Apache ActiveMQ accessible on Docker Hub. This was retrieved by running `docker pull rmohr/activemq` in the command line. To prevent communication streams from getting mixed up, multiple topics are used to differentiate communications between different services and of different message types.

  The data broker was placed in its own service instead of individually being installed on every other

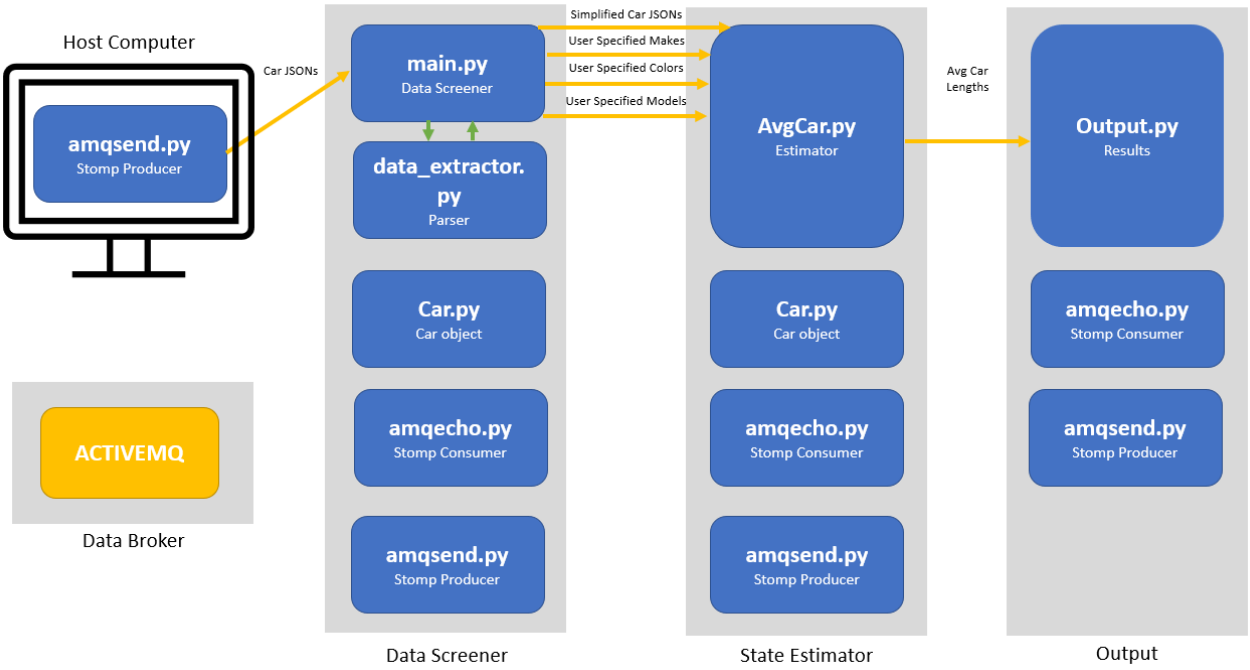

Fig. 5. Plain Vanilla Application: The gray boxes represent the Docker containers. The blue boxes are python files. The interconnections between containers are Stomp sockets passing through the ActiveMQ data broker.

service to reduce redundancy and centralize communication. Installing ActiveMQ on each individual container would make each container a fat container, despite the conditions for a fat container not being met, and require greater networking to connect the application.

- **Data Producer:** The car JSON files were individually sent to the application by the user using the Stomp ActiveMQ producer. To send a file, the producer expects the data broker's IP address (-i) and port number (-p), the connection topic (-t), and the path to the car JSON file (-f) via the command line. The topic must match the one provided to the data screener.
- **Data Screener:** An ActiveMQ listener received these JSON files and extracted and stored the necessary information in an internal data structure. This is passed to the data classification process which regularly pulls from the user configuration process to identify which cars should be passed along for further processing by the state estimator (their features match the features specified in the configuration file). In order to run, it expects the IP address (-i) and port number (-p) to connect to data broker, topic (-t), subscriber ID (-s), and the path to the configuration file (-f) to be specified via the command line. The topic and subscriber ID create a consumer to consume car JSON messages from the host computer. This topic must match the host computer's amqsend.py topic.

- **State Estimator:** The state estimator averages the lengths of the cars passed to it based on their specific features. For example, if the user specified “make Honda” and “make Toyota” in the configuration file, the state estimator would calculate the average length of all Hondas and the average length of all Toyotas separately. It would then aggregate these results to find the average length of all Hondas and Toyotas. Additionally, we added some unnecessary matrix multiplication to mimic the amount of processing that data in JHU/APL’s systems undergo. The service expects the IP address of the data broker (-i) via the command line. It assumes that the data broker’s communication will all be through the 61613 port as all the socket connections in the service use the Stomp protocol.
- **Output Handler:** This service updates the live output of the average length of cars by user defined feature, along with the overall average length of cars processed as time passes. Theoretically, these values should be more extreme at the beginning with fewer data points, but smooth out as the application receives more data. Like the state estimator service, this service only expects the data broker’s IP address (-i) via the command line.

2) Docker Setup: The project was developed on a machine with a Windows 10 Home operating system. Docker Desktop was installed following Docker’s documentation [61]. This required enabling the Windows Subsystem for Linux and installing a Linux Distribution for Windows, in this case, we chose Ubuntu [62]. Docker Desktop includes the Docker Engine, Docker Command Line (CLI) Client, Docker Compose, Notary, Kubernetes, and Credential Helper. Docker commands were run in the Windows Powershell.

3) Dockerfiles: Dockerfiles can be thought of as instruction manuals that a Docker engine uses to construct Docker images [63]. They were created for each of the application’s services, with the exception of the ActiveMQ service whose image was pulled fully constructed from Docker Hub [35] [Fig. 6]. Each line of a Dockerfile has an instruction followed by arguments. The following are valid Dockerfile instructions:

- `FROM`: The FROM directive indicates what pre-existing docker image the Dockerfile should build this Docker image upon, called its base image. Since all of these services were developed in Python, they were built on the python:3.8 docker image, which provides access to the Python version 3.8 programming language (ex: `FROM python:3.8`). More Docker images can be found at Docker Hub for this purpose [35]. There are base Docker images for most programming languages and operating systems. Each Docker image only has one base image, but it is possible to have a Dockerfile which

builds multiple Docker images and therefore has multiple FROM commands.

- `RUN`: The RUN directive tells Docker to run a command (shell format) or executable (exec format). The default shell format is /bin/sh -c on Linux and cms /S /C on Windows, but the user can configure the command to run in a different shell, such as bash, by changing the first argument (ex: `RUN /bin/bash -c <command>`) [63]. If the user is trying to run an executable, the format is `RUN ["executable", "argument1", "argument2"]`. A Dockerfile can include as many lines with the RUN directive as the creator deems necessary.

  The RUN directive is used to install any dependencies that the files in the container require to run. Since our containers used the STOMP protocol for communication between the data broker and containers, the stomp.py library was required on all containers. To do this, we used the command `RUN pip install stomp.py`. The RUN directive is also helpful for creating a directory in each container for storing the source code (ex: `RUN mkdir -p /code`). This directory can be set as the working directory in the following step.
- `WORKDIR`: The WORKDIR directive sets the working directory for commands in the Dockerfile. If no working directory is set, the container creates a default working directory. However, best practice is to always set the working directory, so that the developer can always ensure that everything in the container is located where it should be, especially if a program relies on specific paths.
- `USER`: The USER directive instructs the Dockerfile to run all the subsequent commands as the specified user. To create a Windows user in a Docker Container: `RUN net user /add <user>`. The container's current user can then be set with `USER <user>`.
- `COPY`: The COPY directive loads files and directories into the Docker image. The first parameter is the file/directory path and name on the host computer and the second parameter is the destination file/directory path on the Docker container. This can be used for as many files as needed to run the service. For example, `COPY amqecho.py /code/amqecho.py`. If no file path is specified on the host machine, it assumes that the file is located in the same directory from which the container's Dockerfile is being built. If the file or path contain spaces, then the command can be put into square brackets and quotes as: `COPY ["amq echo.py" "/code/amqecho.py"]`. Additionally, the developer can set the file's ownership to be one of the developer defined groups or users with the - - chown parameter (ex: `COPY --chown=<user>:<group> <files> <dest>`). If the

developer would like to protect against specific files being accidentally copied, they can specify this in a .dockerignore file [64]. This is similar to a .gitignore file and should be placed in the same directory where the Docker image will be built. Like a .gitignore file, the developer can list UNIX regular expressions in the .dockerignore file which will be checked against any file copied into the Docker image to make sure there is no match.

- `ADD`: The ADD directive performs the same operations as the COPY directive with broader capabilities. In addition to files, it allows the developer to upload files directly from URLs and automatically upload and extract tar files to a destination. Despite its broad capabilities, the ADD directive should not be overused. Best practice states that the COPY directive should be used for files and directories still as it is more direct in meaning [65]. Additionally, if the file being uploaded from the URL is a .zip, then it is better to use the RUN directive with the curl command. Otherwise, this would require two commands in the Dockerfile - an ADD command to upload the .zip file and a RUN command to extract the .zip - compared to one, increasing the size and complexity of the Docker image. For uploading tar files, the ADD directive is the best command to use as it does both the uploading and extracting in one step. The format follows the same as the COPY directive: `ADD --chown=<user>:<group> <src file/directory/URL> <dest>` with the option to add square brackets and quotes if there are spaces in the source or destination [63].
- `EXPOSE`: The EXPOSE directive instructs the specified ports in the container to be opened to listen for traffic from other machines in the network at run time. There is the option to specify the protocol as well. If the protocol is not specified, it is assumed to be tcp. As the data broker, the ActiveMQ container exposes many of its ports to listen for different types of traffic. Since STOMP traffic is normally conducted over port 61613/tcp, the ActiveMQ Dockerfile includes `EXPOSE 61613/tcp`.
- `ENV`: The ENV directive allows the developer to set environment variables. The format is `ENV <variable name>="<variable value>"`.
- `LABEL`: The LABEL directive sets Docker image metadata. A Docker image can have multiple labels or none at all. The format is `LABEL <label1_name>=<value> <label2_name>=<value> ...`. The values and label names only need to be in quotes if there is a space in them. It is important to note that Docker images automatically inherit the labels from their base image.
- `HEALTHCHECK`: The HEALTHCHECK directive runs a developer-specified command regularly to

check on the status of the container. The container's initial status is starting. When the container passes a health check, it becomes healthy. If it fails several consecutive health checks, its status changes to unhealthy. The developer can choose to specify how often the health check is run (–interval), duration of a health check before it times out (–timeout), how long after the container's initialization to start health checks (–start-period), and number of retries before declaring a container unhealthy (–retries). The default values are 30 seconds for the interval and timeout, 0 seconds for the start period, and 3 retries. The format is `HEALTHCHECK --interval=DURATION --timeout=DURATION --start-period=DURATION --retries=N CMD <command> <parameter1> ....`

- `VOLUME`: The VOLUME directive mounts a directory so that it can be shared between a group of containers and its data can remain available even after the container stops running. Once a volume is initialized in a container the first time it is run, every subsequent time it is run, the same volume is used to access and store data. The format is `VOLUME <directory>`.
- `ONBUILD`: The ONBUILD directive is used to add commands that will only be run when the current Dockerfile is used as a base image for another Docker image. The format is `ONBUILD <DOCKERFILE DIRECTIVE> <arg1> ....`
- `CMD`: The CMD directive can only be used once at the end of a Dockerfile. This runs the main process of the service. The components are placed within square brackets. The first element is a command or executable. The remaining elements are any parameters that are associated with the command. For example, `CMD ["python3", "/code/Output.py", "-i", "activemq"]` runs Output.py using python3, which was loaded using the FROM command. The third and fourth arguments indicate to Output.py that the host name for connecting to the data broker is activemq.

```
FROM python:3.8
RUN mkdir -p /code
WORKDIR /code
ENV BROKER activemq
RUN pip install stomp.py
RUN pip install pyvim
COPY Output.py /code/Output.py
COPY amqecho.py /code/amqecho.py
COPY amqsend.py /code/amqsend.py
CMD python3 /code/Output.py -i ${BROKER}
```

Fig. 6. Output Service Dockerfile

It is important to remember that the container is acting as an isolated process, so it needs to be given all the necessary files and tools to run. This does not just mean the main process, but every file or dependency that file depends on and so forth. However, there is a fine line between adding everything the container needs to run and adding too much. A container should only include the bare minimum of what is necessary to run as it is supposed to be a portable, easy to debug, and fast to build and run.

Although pyvim is not a necessary dependency for any of the files or processes in any of the containers, it was installed on all of them. This allowed us to open files and make minor changes while the containers were running during the debugging process, since no editor naturally exists on a container. However, once the Docker container is working, it is better to rebuild the image without this line, to reduce the size of the image and the container.

4) Building a Container: To build a container using a Dockerfile, run the command:

`docker build -t <image>:<tag> -f <Dockerfile> .`

This prints out the Dockerfile's step by step execution, so that if it errors, we can identify what caused the issue. The tag is the image version. It is optional, but most docker images have a "latest" tag, indicating that the Docker image is the latest version of the project.

The '.' at the end of the Docker build command indicates the location of the Docker context. If there is only a . and no path, then the Docker context is the current directory where the Docker build command is run. When the Docker build command is run, all the files within the Docker context and its sub-directories are sent to the Docker Daemon [66]. The Docker Daemon listens for requests from the Docker client and manages all the images, containers, volumes, and networks [67]. This is to provide the Docker Daemon with all the potential files that could be copied/added in the Dockerfile. If the Docker context is large, even if not all of the files are added in the Dockerfile, the Docker image and container will be much larger. This increases build, push and pull time. To increase efficiency, the Docker context should only contain the necessary files. It should be noted that although the .dockerignore file keeps specific files from being added to the Docker image, it does not prevent the files from being added to the Docker context [63].

To pull a pre-built image from Docker Hub, user the command: `docker pull <image name>`.

5) Running Individual Containers: Once an image is built, a container can be run from it at any time. To view the images loaded/built on the host machine, use the command: `docker images`. This brings

up the image names, their tags, ids, when they were created, and size. To remove an image from the host machine, use the command: `docker rmi <image id>`.

To run a container based on an image, the user must use `docker run`. This command has many options; the most important ones to know are container name (–name), ports (-p), and detached state (-d) [68]. The user also has the option to override any of the Dockerfile specifications. The container name should only be specified if absolutely necessary for the application's functioning. Overspecification of container names prevents multiple copies of the same container being created to handle high volume of data. One case where the container name should be specified is the data broker, as every other container must connect to it and needs to know its host name. In this application there is only one data broker; however, if an application had multiple data brokers, it would be important to make sure each data broker had a unique name and producers and consumers on the same topic were connected to the same data broker. The ports option binds a host machine port to one of the container's ports. The detached state automatically stops a container when it exits, which prevents errored containers from continuing to run. An example of all these options put together is `docker run –d –-name <container name> –p <host machine port>:<container port> <image name>`.

A user has the option to expose or bind ports or sometimes do both. It is important to understand the difference between the two options to ensure proper port configurations. Ports are normally exposed in the Dockerfile; however, this can be overridden in the `docker run` command just like any other Dockerfile configuration. Exposing a port in a container allows the container to listen for traffic on that port. This is necessary in cases such as the data broker where the container is expecting a lot of traffic on specific ports.

Port binding normally happens during the Docker run step. This maps the container's port to a port on the host machine. Therefore, if the container is not on the same network as the host machine, it can still receive communications from the host machine through that port. To confirm which ports a container has open and is listening on, use `docker port <container name>`.

Once the Docker container is running, the inside of the container can be accessed using Docker command line or `docker exec`. If using Docker Desktop, the running container can be selected, followed by the Docker command line button in the top right corner to open up a bash window inside the container. This bash window can be used for debugging purposes. The same bash window can be accessed by

running `docker exec -it <container id> /bin/sh`. If an editor, such as vim, is installed in the image via the Dockerfile, then the user can use it to examine the file structure in the container while it is running, edit it, and run new commands. However, these changes will not persist once the container is removed, so they should be limited and transferred to the original files once the user decides to keep them.

To list all the containers on the host machine, there are two commands the user can use: `docker container ls` or `docker ps`.

Before removing a container from the host machine, it must be stopped. Otherwise, it can cause an error in Docker. This error can normally be resolved by restarting the Docker Engine. To stop a running container, use the command: `docker stop <container name>`. To then remove the container, use the command: `docker rm <container name>`.

6) Connecting the Containers: Once all the services were decided, we had to figure out how to connect them. We first attempted running only the ActiveMQ service with the rest of the processes running on localhost. This required opening up the ports that ActiveMQ was listening on our host machine. STOMP's default port is 61613, which we opened to listen for both incoming and outgoing traffic. When running the ActiveMQ service, we bound the internal 61613 port to the host machine's 61613 port using the -p directive as seen: `docker run -p 61613:61613 rmohr/activemq`. With this setup, all of the processes successfully ran on the local machine and all communication was directed through the data broker container. The next step was to determine how to launch the other processes as containers and connect them to the data broker.

7) Docker Networking: Docker networking enables the communication between Docker containers required to maintain the microservices approach. The four containers in the plain vanilla project were run on the same Docker network to allow constant communication between the services [Fig. 7]. All Docker containers are assigned an IP address upon run time which allows them to ping and connect with other Docker containers. If their container name is unique, it can also act as a domain name. The container's ports are automatically open to other Docker containers on the same network. To allow the host computer to send traffic into the network, we bound ActiveMQ's 61613 port to the host computer's 61613 port.

The networks currently running on the Docker engine can be viewed using the command `docker network ls`. Prior to adding any networks, there should be three networks in this list: bridge, host, and

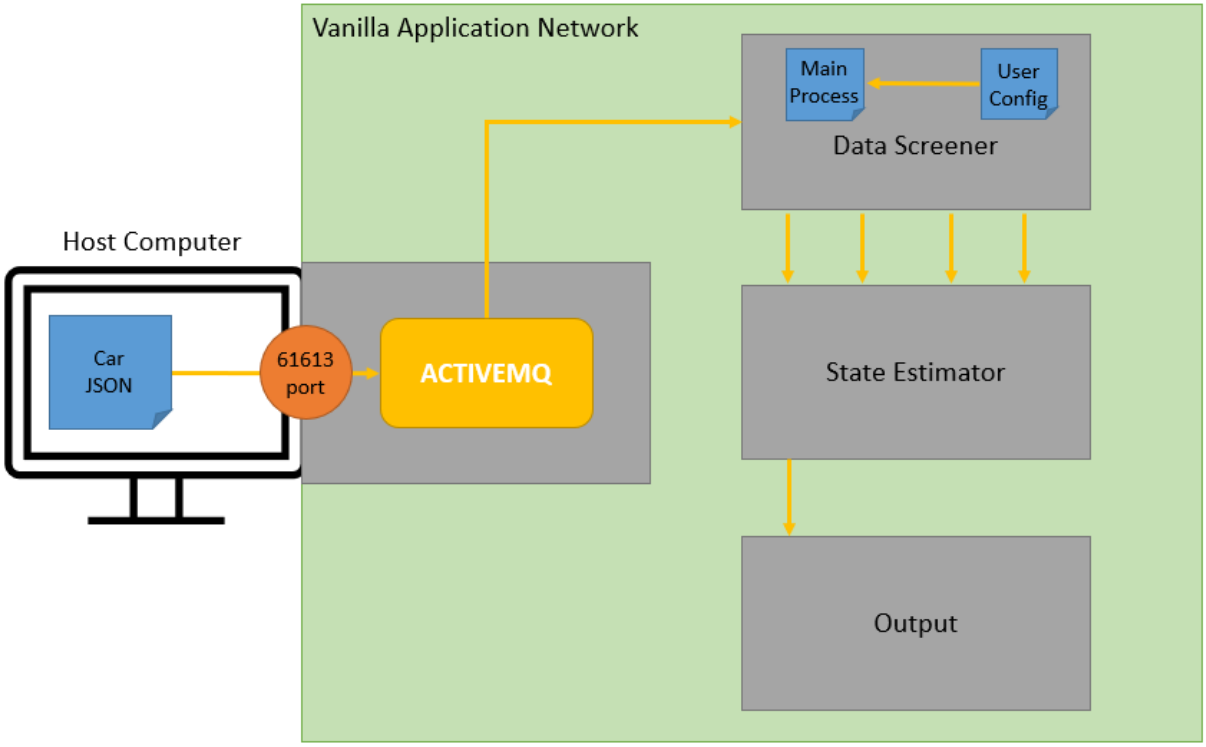

Fig. 7. The Plain Vanilla Application's Networking Configuration: The ActiveMQ connections are represented by yellow arrows with each producer pointing to each consumer. ActiveMQ identifies unique connections through specified topics. The bound 61613 port allows car JSON messages to pass from the host machine into the application network.

none. The bridge network is the only fully functional network [69]. To connect all the containers, they all must be added to the same network. To create a network, the command `docker network create --driver bridge <network name>` was used. The bridge network is the default driver for any new network and does not have to be specified.

Once the network is created, the containers have to be added to it. If the containers are already running, they can be added using `docker network connect <network name> <container name>`. If the containers are not running, the Docker run command has a –network option that allows the developer to specify which network they would like the container to be added to: `docker run -dit --name <container name> --network <network name> <image name>`. However, this can only be used once in the run command, so if the user would like to connect the container to multiple networks they have to run the previous command for any additional networks. The container's addition to the network can be confirmed with `docker network inspect <network name>`. This lists all the

containers in the network and their IP addresses along with other relevant information.

Once there are multiple containers in the network, they should be able to ping each other with `docker exec` using their host names or their IP addresses. The host names are the container names themselves. The ActiveMQ databroker requires that only the ActiveMQ container expose its 61613 port. Every other container does not have to configure its ports, as the STOMP connections are solely being hosted by the ActiveMQ container on port 61613. If for some reason, the other containers did require a listener of some type, specific ports on each container could be exposed by altering the Dockerfile or using the –expose option at run time. The command `docker port <container name>` makes it easy to view which ports are open on each Docker container.

Once all the containers are in the network, they can all communicate with one another, but they cannot communicate with machines outside of the network. To allow us to send Car.JSON files from the host machine into the system without setting up an entirely separate container, we bound ActiveMQ's 61613 port to the host machine's 61613 port upon run time. This allows ActiveMQ to receive traffic from the host machine via its 61613 port. The host machine could then locally send Car.JSON files through the command: `python amqsend.py –i localhost –p 61613 –t test –f Car0001.JSON`. It is important to note that the Car.JSON files are sent to the local host machine's 61613 port rather than the ActiveMQ machine's 61613 port. This is due to the fact that the two machines are not on the same network and cannot ping each other. Their only connection is this port through which ActiveMQ is listening to all traffic on the localhost machine and vice versa.

C. Phase 3: Autonomic Computing

1) Continuous Integration and Delivery: As mentioned previously, we met with a representative from JHU/APL working on improving the CI/CD process at the beginning of phase three. Their efforts were in parallel with the duration of this project and achieved many of the continuous integration and delivery goals set forth at the project's beginning. However, when talking with them, they indicated that there were two places for possible improvement of their CI/CD systems:

1) Incorporating static code analysis into the continuous integration pipeline
2) Streamlining their artifact management systems

Prior to shifting focus in phase three, we investigated the feasibility of a tool called SonarQube to achieve this first goal. SonarQube checks for bugs, security vulnerabilities, and code smell - places where

the code is confusing or difficult to maintain [70]. Upon finding one of these issues, it rates it on a scale of one to five. SonarQube is compatible with Java, C#, C, C++, JavaScript, TypeScript, Python, Go, Swift, COBOL, Apex, PHP, Kotlin, Ruby, Scala, HTML, CSS, ABAP, Flex, Objective C, PL/I, PL/SQL, RPG, T-SQL, VB, VB6, and XML. It can be integrated with the Gitlab CI/CD pipeline and other CI/CD pipelines to automatically check new pushes, pull requests, branches, and merges.

To examine the plain vanilla project using SonarQube, we first ran the local SonarQube server. The instructions to host/run this server are on a publicly accessible Docker image. We pulled this image with `docker pull sonarqube:latest`. We then ran the local server using the command: `docker run -d --name sonarqube -e SONAR_ES_BOOTSTRAP_CHECKS_DISABLE=true -p 9000:9000 sonarqube:latest`. Since the local SonarQube runs on the 9000 port, we had to open this port on our host machine to allow traffic. The first time this is run locally, the server requests that the user change the password. Using the interface located at http://localhost:9000, we were able to create a project key called “MQP”.

SonarQube then requires a specific scanner to be run on the code depending on the language type - there are seven main scanners including a default scanner. They can be run using provided docker images or by installing the scanners locally on the user’s machine via downloadable zip files. We were unable to get the Docker images of the scanners to work, because it could not locate the sonar_project.properties file in the project root directory among other issues when running the Docker image. However, we were successful in running the locally installed SonarScanner. Once the contents were extracted from the zip file, the machine’s environment variables had to be updated to include a pathway to the scanner’s binary. The scanner was run with the command: `Sonar-scanner.bat -D sonar.login="<username>" -D sonar.password="<password>" -D sonar.projectKey=<projectkey> -D sonar.projectBaseDir="<path>"`. The results of the plain vanilla project’s analysis were then viewable at http://localhost:9000. Each file was graded based on the number/severity of the issues and the amount of time SonarQube estimated the issues would take to correct. This information aids the user’s prioritizing of the changes.

One benefit of SonarQube is that it allows users to add additional rules for code analysis of the following languages: C#, COBOL, Flex, Java, JavaScript, PHP, PL/SQL, PL/I, Python, RPG, VB.NET, XML [70]. This would allow JHU/APL to enforce and maintain their own style guides.

Through discussions with the sponsor, the second goal was not further investigated due to the inability to fully test any discoveries with JHU/APL's CI/CD systems in progress. At this point, the project shifted to focus on applying autonomic computing to the plain vanilla application.

2) Autonomic Computing: The goal of this phase was to modify the existing plain vanilla application to automatically scale up resources when high traffic flow is detected. Specifically, we wanted to automatically scale up the Screener service during periods of high influx traffic to handle car JSON parsing, while only using one of every other service type to process the data parsed. Although this required many pieces of the AWS infrastructure, the main technology used was Amazon's Elastic Container Service (ECS) which allows users to create clusters of containers.

To access our AWS account, we used both the online AWS console and AWS Command Line Interface (CLI). After installation on our local machine, the AWS CLI had to be configured to work with our account by loading config.txt and credentials.txt files [71] [72]. Additionally, the account logged us out after every 12 hours and `aws ecr get-login-password | docker login --username USER --password PASSWORD ACCOUNT_ID.dkr.ecr.REGION.amazonaws.com` was required to re-login on the local machine. Using the CLI, we could access the AWS account from our local command window.

AWS Infrastructure

We began by configuring the network for the AWS version of the plain vanilla application. Amazon's Virtual Private Cloud (VPC) acts as a private network on which to develop applications [73]. Within it, there are multiple availability zones, the exact number depending on the region being used for development. We created four subnets within our personal VPC, two on each of the availability zones, with one in each zone having public access.

We then created an ECS cluster called cars via the AWS console [74]. ECS is automatically supposed to generate the specified number of ECS instances upon the creation of the cluster. These ECS instances are linked to an autoscaling group which automatically generates new instances if the old ECS instances are terminated. However, we ran into issues where the created ECS instances were not linked to the cluster.

There are several ways to approach fixing this. The first method involved creating an ECS instance through its components. At its core, ECS instances are Amazon Elastic Compute Cloud Instances (EC2) running an Amazon ECS container agent. Therefore, we tried to launch an EC2 instance and run a

container agent on it [75]. To launch an EC2 instance, we had to specify an Amazon Machine Image (AMI), which acts a model for the EC2 instance [76]. We chose the Amazon Linux AMI because it provided access to the Linux operating system, Python, mySQL, and AWS CLI. By adding a key pair to an EC2 Instance, upon its creation, we could conceivably connect to its inner configurations files once it was launched and modify them to link it to our cluster. Users can generate key pairs on AWS and download the private keys to their local computers. Without the correct key pair, an EC2 instance would deny us access for not having the appropriate credentials [77]. However, key pairs are not required when creating AWS resources if the user does not expect to alter the resource in the future. To launch the ECS instances, we had to ssh into it. This required altering the ECS instances' security group to allow incoming traffic on port 22 from the ssh protocol [78]. After following all these steps, we still had issues launching our ECS instances and accessing their configuration files, most likely because we chose the wrong AMI or made a mistake in configuring the network.

The second method involved trying to determine what was blocking the cluster-generated ECS instances from registering with the cluster. Since both ECS instances had public IP addresses, they needed access to the internet to register. However, we discovered that we were missing an internet gateway to the VPC. Once we created this gateway and added it to the routing table, the ECS instances began to register [79].

After the ECS cluster was registering the ECS instances, we had to work towards creating cluster services. Cluster services are comparable to microservices within the cluster. Prior to launching autoscaling cluster services, there are multiple steps:

1) Creating new AWS Identity and Access Management (IAM) Roles to authorize further development
2) Uploading Docker images to the AWS Elastic Container Repository (ECR)
3) Configuring a task definition for each cluster service
4) Creating a network load balancer (NLB) and target group to reroute traffic based on the number of Screener services and traffic density
5) Configuring each cluster service

We first created an ecsTaskExecutionRole to allow our ECS containers permission to make API calls. Example API calls include retrieving Docker images from the ECR and sending information to awslogs. This role was created via the AWS IAM console and the AmazonECSTaskExecutionRolePolicy was attached [80]. Policies define what a role can do and multiple policies can be added to the same role, if

needed.

To run our previously created Docker images on the cluster, we had to upload them to the ECR. This had to be done through the CLI. Within our command window, we created a repository for each individual Docker image: `aws ecr create-repository --repository-name NAME --image-scanning-configuration scanOnPush=true --region REGION`.
We then modified our Docker image tag to include our aws account information: `docker tag DOCKER_IMAGE ACCOUNT_ID.dkr.ecr.REGION.amazonaws.com/DOCKER_IMAGE`.
Finally, we pushed the Docker image to the repository we created: `docker push ACCOUNT_ID.dkr.ecr.REGION.amazonaws.com/DOCKER_IMAGE`.

Each cluster service runs one or multiple tasks. These tasks are defined by a task definition. Task definitions can be created via the AWS console or an uploaded JSON file. Task definitions define all the Docker containers each service relies on and their dependencies. In our case, we only ran one Docker image within each service, but we could have ran more. There are two types of task definitions/services: EC2 and Fargate. EC2 tasks require EC2 instances to run and are for resource heavy applications that run constantly [81]. In contrast, Fargate tasks are only supposed to run as needed for testing and do not require EC2 instances. In theory, we wanted to use Fargate tasks because we only needed to run our application while testing, and did not need to consume resources otherwise. However, Fargate tasks limit the choices in networks to awsvpc, thereby limiting the docker configurations for the containers on the network. This was something that we did not want to have to deal with, so we chose to proceed with EC2 tasks.

Once the task definition type is decided upon, the task definition must define the network mode, task role, task execution role, containers, and app mesh (if applicable). The network mode choices include Bridge, Host, awsvpc, and none [82]. We tried to use the awsvpc network, because it is the most flexible for autonomic computing - it allocates an elastic network interface for each task, which allows it to dynamically reassign port host mappings as needed. In comparison, the Host network uses an EC2 instance's network interface and the Bridge network uses the container interface.

The task definition requests a task role and task execution role. Since we are not attempting to make API calls to any other AWS services through our task definitions, we do not need a task role [82]. However, the task execution role is the ecsTaskExecutionRole we made earlier.

The Docker images used to form the containers are passed in to the task definitions by ids registered to each image uploaded to the ECR. Since a container consists of a Docker image plus all the specifications to run the image, the task definition includes port mappings, environment variables, hostnames, and more [82]. A task definition also records dependencies between any of the containers, if multiple containers exist in the task. Additionally, there is the option for AWS to produce cloud logs for each container in the task by choosing a check box under container definitions -> auto-configure cloud watch logs. These logs show what is happening inside the container if it exits prematurely.

To scale up the Screener service, there needs to be a load balancer to reroute traffic to all of the containers running the Screener task. There are three main types of load balancers to choose from: Application, Network, and Classic [83]. The application load balancer can only direct HTTP traffic, making it incompatible for our application which uses TCP traffic. The classic load balancer is a deprecated load balancer that only works with the older version of ECS clusters. This made the network load balancer the best choice for use with our application as it supports both our clusters and TCP traffic. The load balancer requires a target group indicating where to send the traffic [84]. In our case, it is the containers in the Screener service. The Screener service is placed in an autoscaling group with a policy to scale up (add more containers) when its average CPU usage is higher than a specified percentage. The incoming traffic port must also be defined in the load balancer and the security group was modified to allow traffic via that port. This load balancer is linked to the Screener service upon its creation.

Using the task definitions, we launched EC2 cluster services. Each cluster service was assigned a name, task definition, number of tasks, VPC, and subnets [85]. We chose to make each service of type Replica with only one task. Replica services spread the specified number of tasks across the number of containers within the cluster, while Daemon services keep only one copy of the task on each container [86]. Unlike Daemon services, Replica services can be autoscaled. Each cluster service, with the exception of the Screener service, was placed within its own autoscaling group with the desirable number of containers set to one. This ensured that a new container would automatically be created if a container was stopped.

Docker Container Changes

When switching to the AWS platform, alterations had to be made to the Docker containers to be compatible with autonomic computing. To centralize data among all the containers, we created a mySQL database on the AWS Relational Database Service (RDS). This database holds the most recent user

configurations and the specifications of cars that have passed screening. If traffic increases and new containers are launched, the new containers can access older containers' data through this database. The database also allows data to persist even when a container goes down. This also removed the need for producers and consumers connecting the Screener and Estimator services, as they both could individually update and poll the database for more information.

Due to the flexible nature of autonomic computing, values relating to resources and networking cannot be hard coded within the Docker containers, as multiple copies of the same Docker container would then be fighting over those communication pathways and resources. This was not an issue when running the Docker containers locally with only one copy of each one.

The first place where this became a concern was the individual consumer's subscriber IDs within each container. For the ActiveMQ data broker, each consumer needs a unique subscriber ID to identify it. These values were originally hard coded, as we knew exactly how many consumers the containers were using and the IDs of each of them. However, prior to transferring the container's images to AWS, we had to update the code to randomly generate subscriber IDs between 0 and 1,000,000 every time a new consumer is created. Due to the large range, the odds of the same number being generated more than once while the application is running is minimal.

We also updated the Dockerfiles to include environment variables so that we did not have to hard code the values for connecting to the ActiveMQ container or mySQL database. Instead, the ActiveMQ hostname, mySQL database hostname, and mySQL database permissions could be passed into the Screener, Estimator, and Output services, as needed, through their task definitions.

Resolving Cluster Errors

Despite following all the steps to add our services to the cluster, we ran into two major issues revolving around ActiveMQ, which made it seemingly incompatible for use with Amazon's Elastic Container Service and autonomic computing:

1) only the tasks within the ActiveMQ service would progress to RUNNING status, and
2) ActiveMQ's subscriber model did not allow for a load balancer to be added.

There are eight states an AWS task can achieve with the ideal one being RUNNING [87]. The tasks in the Output and Estimator services consistently remained at PENDING or PROVISIONING before timing out and STOPPING. By looking at the logs, we were able to see that the consumers within the Output

and Estimator containers could not connect to ActiveMQ. By connecting to the ECS instances through ssh (using the key pair we specified during its creation), we were able to see that the services, and therefore containers, were not always being added to the same ECS instance. When originally making the cluster, we specified that we would like two ECS instances, not realizing that those instances have no way of communicating with each other. Therefore, we altered the autoscaling group for the cluster's ECS instances so that the desirable number of ECS instances was one, causing one ECS instance to terminate and all containers to move to the other instance. Despite this, the containers still continued to produce logs indicating that they could not connect to ActiveMQ.

On the advice of an Amazon representative, we then changed the network mode in each services' task definition to Bridge as that is the closest comparable network to the Docker networks we created locally in phase 2. Those networks were based on a bridge network driver. However, this is also the most rigid of all the options, which makes it less desirable for autonomic computing. This did not fix the issue. We determined it was because of the difference between the Bridge network mode and the user created Docker networks. User created Docker networks can connect to each other by name or alias, while the default Bridge network mode can only connect to other containers using IP addresses. Since every time the AWS task stops and reconnects the container IP address changes, it is not possible to pass the ActiveMQ container IP address as an environment variable to the Screener, Estimator, and Output services. The only way to fix this without removing ActiveMQ would be to move all the containers into the same task definition, and therefore, service, thereby removing the microservice architecture and defeating the purpose of switching to containers.

The second issue had to do with ActiveMQ's subscriber model. ActiveMQ is a data broker that passes messages back and forth between producers and consumers. It has defined ports for each protocol type - 61613 being the port for the STOMP protocol. Producers send information to this port, while consumers subscribe to information from this port. The information a consumer receives is determined by its "topic". Each consumer only subscribes to one topic, while each producer must specify a topic when sending each piece of information. A consumer only receives the information on its topic. This setup does not mesh well with a load balancer which expects a port number which it can listen to and divide all traffic from which it hears among the containers in its target group. There is no way to give ActiveMQ's port number to the load balancer, because that would require also specifying the topic that the load balancer's target

group is trying to listen to, which is not possible in AWS's current setup.

Due to the multiple issues with using ActiveMQ, we theorized a method of running the application without it. In this method, the ActiveMQ and Output services would be removed entirely, and the Screener service would be broken into two services: UserConfig and Screener [Fig. 8]. The UserConfig service would take over the Screener's responsibility of checking the user configuration file (whose path would be passed via environment variable) every couple minutes to determine if there had been any changes, and updating the mySQL database to reflect the user's wishes. This needs to be compartmentalized from the rest of the Screener, because otherwise the application would be wasting resources checking the user configuration file far more often than needed as the number of Screener services scaled up.

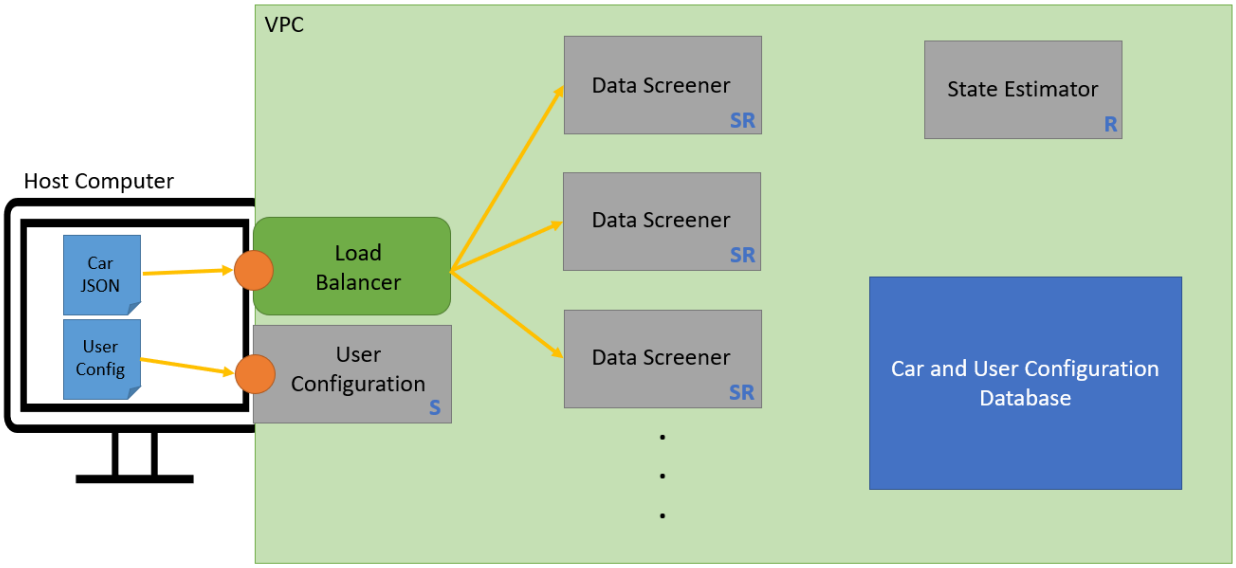

Fig. 8. The theorized alternative AWS networking configuration for the plain vanilla application without ActiveMQ. The blue S's and R's in the corner of each service denote whether the service stores information or retrieves information from the database. The orange circles represent two ports on which the load balancer and User Configuration service are listening to the host computer.

The Screener service would be responsible for screening car JSON files based on the user configurations that they read from the mySQL database. Any cars that passed the user configurations would be added to the mySQL database. To enable on demand scaling, the Screener service would be added to the target group of a network load balancer, which would split the incoming traffic among all its targets. When the CPU usage of each Screener service surpassed a specified percentage, the number of Screener services would automatically increase. The network load balancer would read the car JSON in through a dedicated TCP socket. A security group for the network load balancer would allow all TCP traffic to come in via

the socket's specified port, but no outgoing traffic.

The Estimator service can then access the database to retrieve the user specifications and the cars that passed screening. For each user specification, it would calculate the average car length of the cars meeting that specification and print it out to the console's logs. This would eliminate the need for the Output service as there would be no reason to send print statements to a service just to print them out. Additionally, the Estimator and Output services had been connected by ActiveMQ, so if we did keep the Output service, we would have to create a socket to connect them.

Based on the struggles we experienced trying to adapt the plain vanilla application to AWS, we suggest designing applications with the goal of autonomic computing in mind in the future. This means emphasizing maximum abstraction so that individual containers can spin up and terminate at will without impacting the overall application. Additionally, understanding the system requirements is necessary prior to determining the container's communication methods. Otherwise, issues such as the ones we ran into with ActiveMQ and the load balancer may persist.

IV. Future Work

Time prevented us from implementing our theoretical model of a scalable version of the plain vanilla application without ActiveMQ. Future steps would include implementing this model and also further investigation into the limits of AWS and ActiveMQ. AWS provides a service called AmazonMQ, which has the same functionality as ActiveMQ. Theoretically, since AWS provides both the ECS Container Service and AmazonMQ functionality there should be a way, though possibly less than ideal, to get the two services to interact.

Additionally, the plain vanilla application is a simplistic model of the applications JHU/APL develops on a regular basis. This was a conscious decision based on time constraints; however, a more accurate model would factor in time-based data latency, confusion matrices, and multi-hypothesis decision makers. Future work would include learning the optimal methods to replicate this work on a more complex and authentic application.

References

[1] What is the software development life cycle? [Online]. Available: https://phoenixnap.com/blog/software-development-life-cycle

[2] The johns hopkins university applied physics laboratory. [Online]. Available: https://www.jhuapl.edu/

[3] A history of the cd-rom drive. [Online]. Available: https://www.gcis.co.uk/a-history-of-the-cd-rom.html

[4] S. J. Spooner, “The validation of shrink-wrap and click-wrap licenses by virginia’s uniform computer information transactions act,” *Richmond Journal of Law & Technology*, vol. 7, no. 3, p. 27, 2001.

[5] (2020, 06) .tarfile extension. [Online]. Available: https://fileinfo.com/extension/tar

[6] J. Price. (2009, 06) How to install software from a tarball in linux. [Online]. Available: https://www.maketecheasier.com/install-software-from-a-tarball-in-linux/

[7] F. Wahab. (2018, 07) How to use tar on windows 10. [Online]. Available: https://www.addictivetips.com/windows-tips/use-tar-on-windows-10/

[8] S. Conroy. (2018, 01) History of virtualization. [Online]. Available: https://www.idkrtm.com/history-of-virtualization/

[9] J. Lin. (2019, 03) Deploying a scalable web application with docker and kubernetes. [Online]. Available: https://medium.com/better-practices/deploying-a-scalable-web-application-with-docker-and-kubernetes-a5000a06c4e9

[10] A. Pollock. (2020, 09) Virtualization vs. containerization. [Online]. Available: https://www.liquidweb.com/kb/virtualization-vs-containerization/

[11] About the open container initiative. [Online]. Available: https://opencontainers.org/about/overview/

[12] H. Lai. (2015, 01) Baseimage-docker, fat containers and ”treating containers as vms”. [Online]. Available: https://blog.phusion.nl/2015/01/20/baseimage-docker-fat-containers-treating-containers-vms/

[13] C. Richardson. Pattern: Microservice architecture. [Online]. Available: https://microservices.io/patterns/microservices.html

[14] L. Rosenstock. (2019, 02) Designing apis for microservices. [Online]. Available: https://stoplight.io/blog/designing-apis-for-microservices/

[15] Devops. (2015, 05) Containers should be fat or thin? [Online]. Available: https://www.boynux.com/containers-fat-thin/

[16] E. Baez. (2019, 04) Containers as a service: A complete guide to understanding. [Online]. Available: https://www.scalyr.com/blog/containers-as-a-service-complete-guide/

[17] What is containers as a service (caas)? [Online]. Available: https://www.ibm.com/services/cloud/containers-as-a-service

[18] What is ci/cd? [Online]. Available: https://www.redhat.com/en/topics/devops/what-is-ci-cd

[19] What is continuous delivery? [Online]. Available: https://continuousdelivery.com/

[20] S. Cornett. (2013) Minimal acceptable code coverage. [Online]. Available: https://www.bullseye.com/minimum.html#:~:text=Code%20coverage%20of%2070%2D80,higher%20than%20for%20system%20testing.

[21] A. Banafa. (2016, 07) What is autonomic computing? [Online]. Available: https://www.bbvaopenmind.com/en/technology/digital-world/what-is-autonomic-computing/#: :text=Autonomic%20computing%20is%20a%20computer’s,maintenance%20such%20as%20software%20updates.

[22] B. Doerrfeld. (2019, 01) 5 container alternatives to docker. [Online]. Available: https://containerjournal.com/topics/container-ecosystems/5-container-alternatives-to-docker/

[23] C. Tozzi. (2017, 07) What’s the future of lxc, docker’s semi-forgotten stepmother? [Online]. Available: https://containerjournal.com/features/whats-future-lxc-dockers-semi-forgotten-step-mother/

[24] A. Abdelrazik. (2017, 07) Docker vs. kubernetes vs. apache mesos: Why what you think you know is probably wrong. [Online]. Available: https://d2iq.com/blog/docker-vs-kubernetes-vs-apache-mesos

[25] (2020, 06) Docker vs coreos rkt. [Online]. Available: https://www.upguard.com/blog/docker-vs-coreos#: :text=CoreOS%20Rocket%20(rkt)%20is%20the,inherent%20in%20Docker’s%20container%20model

[26] J. Armstrong. (2020, 05) Snyk and docker partner to secure containerized applications. [Online]. Available: https://snyk.io/blog/snyk-docker-secure-containerized-applications/?utm_medium=Partner&utm_campaign=Docker-partnership-launch-05-19-2020

[27] Docker security. [Online]. Available: https://docs.docker.com/engine/security/

[28] Overview. [Online]. Available: https://coreos.com/rkt/?utm_source=thenewstack&utm_medium=website&utm_campaign=platform

[29] Faq. [Online]. Available: https://opencontainers.org/faq/

[30] (2020, 01) Defining your multi-container application with docker-compose.yml. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/multi-container-microservice-net-applications/multi-container-applications-docker-compose

[31] Pricing and subscriptions. [Online]. Available: https://www.docker.com/pricing

[32] (2020, 01) Debug containerized apps. [Online]. Available: https://code.visualstudio.com/docs/containers/debug-common

[33] M. Betz. (2016, 03). [Online]. Available: https://medium.com/@betz.mark/ten-tips-for-debugging-docker-containers-cde4da841a1d

[34] rkt commands. [Online]. Available: https://coreos.com/rkt/docs/latest/commands.html

[35] Docker hub. [Online]. Available: https://www.docker.com/products/docker-hub

[36] C. Puccio. (2020, 07) Aws and docker collaborate to simplify the developer experience. [Online]. Available: https://aws.amazon.com/blogs/containers/aws-docker-collaborate-simplify-developer-experience/

[37] P. Yuknewicz. (2020, 05) Microsoft and docker collaborate on new ways to deploy containers on azure. [Online]. Available: https://azure.microsoft.com/en-us/blog/microsoft-and-docker-collaborate-on-new-ways-to-deploy-containers-on-azure/

[38] Docker swarm. [Online]. Available: https://www.sumologic.com/glossary/docker-swarm/#: :text=A%20Docker%20Swarm%20is%20a, join%20together%20in%20a%20cluster.&text=The%20activities%20of%20the%20cluster,are%20referred%20to%20as%20nodes.

[39] (2016) Docker cloud. [Online]. Available: https://www.docker.com/sites/default/files/Docker%20Cloud.pdf

[40] Using rkt with systemd. [Online]. Available: https://coreos.com/rkt/docs/latest/using-rkt-with-systemd.html

[41] rkt and selinux. [Online]. Available: https://coreos.com/rkt/docs/latest/selinux.html

[42] A. Clemenko. (2019, 08). [Online]. Available: https://www.docker.com/blog/docker-enterprise-first-disa-stig-container-platform/

[43] P. Ram. (2018, 04) What is a webhook? [Online]. Available: https://codeburst.io/what-are-webhooks-b04ec2bf9ca2

[44] Set up automated builds. [Online]. Available: https://docs.docker.com/docker-hub/builds/

[45] Automated repository tests. [Online]. Available: https://docs.docker.com/docker-hub/builds/automated-testing/

[46] Set up docker notifications in slack. [Online]. Available: https://docs.docker.com/docker-hub/slack_integration/

[47] Containers on compute engine. [Online]. Available: https://cloud.google.com/compute/docs/containers

[48] Amazon elastic container service. [Online]. Available: https://aws.amazon.com/ecs/?c=cn&sec=srv

[49] Amazon elastic kubernetes service. [Online]. Available: https://aws.amazon.com/eks/?c=cn&sec=srv&whats-new-cards.sort-by=item.additionalFields.postDateTime&whats-new-cards.sort-order=desc&eks-blogs.sort-by=item.additionalFields.createdDate&eks-blogs.sort-order=desc

[50] D. Sudhanshu. Who are biggest customers of the microsoft azure platform? [Online]. Available: https://www.cisin.com/coffee-break/Enterprise/who-are-biggest-customers-of-the-microsoft-azure-platform.html

[51] Amazon identity and access management (iam). [Online]. Available: https://aws.amazon.com/iam/

[52] (2020, 03) Azure security baseline for azure container registry. [Online]. Available: https://docs.microsoft.com/en-us/azure/container-registry/security-baseline

[53] Amazon elastic container registry. [Online]. Available: https://aws.amazon.com/ecr/

[54] Container instances. [Online]. Available: https://azure.microsoft.com/en-us/services/container-instances/#features

[55] Aws app mesh features. [Online]. Available: https://aws.amazon.com/app-mesh/features/

[56] Service fabric. [Online]. Available: https://azure.microsoft.com/en-us/services/service-fabric/

[57] Amazon eks features. [Online]. Available: https://aws.amazon.com/eks/features/

[58] Azure kubernetes service. [Online]. Available: https://azure.microsoft.com/en-us/services/kubernetes-service/

[59] C. Preimesberger. (2019, 08) Aws vs azure. [Online]. Available: https://www.eweek.com/cloud/at-a-high-level-in-the-cloud-aws-vs-azure

[60] "Apache activemq." [Online]. Available: https://activemq.apache.org/download.html

[61] Install docker desktop on windows home. [Online]. Available: https://docs.docker.com/docker-for-windows/install-windows-home/

[62] (2020, 09) Windows subsystem for linux installation guide for windows 10. [Online]. Available: https://docs.microsoft.com/en-us/windows/wsl/install-win10

[63] Dockerfile reference. [Online]. Available: https://docs.docker.com/engine/reference/builder/

[64] madflojo. (2017, 05) Leveraging the dockerignore file to create smaller images. [Online]. Available: https://rollout.io/blog/leveraging-the-dockerignore-file-to-create-smaller-images/#: :text=dockerignore%20File-,The%20.,of%20a%20docker%20build%20command.

[65] N. Janetakis. (2017, 05) Docker tip #2: The difference between copy and add in a dockerfile. [Online]. Available: https://nickjanetakis.com/blog/docker-tip-2-the-difference-between-copy-and-add-in-a-dockerile

[66] Best practices for writing dockerfiles. [Online]. Available: https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

[67] Docker overview. [Online]. Available: https://docs.docker.com/get-started/overview/#: :text=The%20Docker%20daemon%20(%20dockerd%20)%20listens,daemons%20to%20manage%20Docker%20services.

[68] Docker run reference. [Online]. Available: https://docs.docker.com/engine/reference/run/

[69] Network tutorial standalone. [Online]. Available: https://docs.docker.com/network/network-tutorial-standalone/

[70] Sonarqube documentation. [Online]. Available: https://docs.sonarqube.org/latest/

[71] Installing, updating, and uninstalling the aws cli version 2. [Online]. Available: https://docs.aws.amazon.com/cli/latest/userguide/install-cliv2.html

[72] Configuring the aws cli. [Online]. Available: https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-configure.html

[73] Vpc and subnets. [Online]. Available: https://docs.aws.amazon.com/vpc/latest/userguide/VPC_Subnets.html

[74] Creating a cluster. [Online]. Available: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/create_cluster.html

[75] Launching an amazon ecs container instance. [Online]. Available: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/launch_container_instance.html

[76] Amazon machine images (ami). [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html#amazon-linux

[77] Amazon ec2 key pairs and linux instances. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-key-pairs.html

[78] Security groups for your vpc. [Online]. Available: https://docs.aws.amazon.com/vpc/latest/userguide/VPC_SecurityGroups.html#SecurityGroupRules

[79] Internet gateways. [Online]. Available: https://docs.aws.amazon.com/vpc/latest/userguide/VPC_Internet_Gateway.html#working-with-igw

[80] Amazon ecs task execution iam role. [Online]. Available: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/task_execution_IAM_role.html

[81] Ec2 or aws fargate? [Online]. Available: https://containersonaws.com/introduction/ec2-or-aws-fargate/

[82] Task definition parameters. [Online]. Available: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/task_definition_parameters.html

[83] Load balancer types. [Online]. Available: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/load-balancer-types.html

[84] Create a network load balancer. [Online]. Available: https://docs.aws.amazon.com/elasticloadbalancing/latest/network/create-network-load-balancer.html

[85] Creating a service using the new console. [Online]. Available: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/create-service-console-v2.html

[86] Amazon ecs services. [Online]. Available: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/ecs_services.html

[87] Task lifecycle. [Online]. Available: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/task-lifecycle.html