Machine Learning System Architectures for Secure Data Analytics in Agriculture Applications

By

NOAH R. TARR

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Electrical and Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Venkatesh Akella

Mason Earles

Chen-Nee Chuah

Committee in Charge

2022

Abstract

I propose four network architectures for improving security in machine learning model training where training data must remain secure from the model trainer and the model must be secure from the data owner. The architectures are numbered 1 through 4 with the first being the least secure and 4 being the most. Architecture 4 is the final iteration implemented in practice and facilitates end-to-end security for the Provider and one or more Consumer's.

Each architecture is comprised of multiple processes running concurrently and performing distinct tasks. The processes are deployed in docker containers with strictly defined networks to allow communications between processes only where necessary to uphold security objectives. Techniques such as attestation and cryptography are applied among various other technologies including FarmStack Trusted Connectors and Docker. All of which were employed to reduce data and application security threats further.

The important contributions

- Plausible architectures, which provide security to the Provider of the raw data and Consumer of that data, for training models in the agricultural industry.
- A previously untested application of FarmStack Trusted Connectors.
- The development of the Internal Data Handler (IDH), which replaced disk storage as an internal cryptographic file storage server.

My results prove that of the range of architectures described in this paper, the only worth deploying are architectures 2 and 4. Architecture 2 has far from the same security guarantees as architecture 4 but is approximately 3.7 times faster than architecture 4 in its current state. Architecture 4, while slower, addresses all objectives of the threat model. There are many objectives in the threat model not addressed by Architecture 2. Hence, Architecture 4 is recommended over Architecture 2.

Contents

List of Figures

List of Tables

1 Introduction

The new and quickly expanding field of smart farming, or precision agriculture, has resulted in a plethora of ground-breaking and innovative technologies. Many farmers are embracing the incorporation of the Internet of Things, Artificial Intelligence and Robotics in agriculture [1]. Nevertheless, uneasiness exists in conventional farmers over the use of these technologies and the data they produce [2]. Farmers concern for their data is not without reason as it may contain proprietary information like farming practices, growing techniques, crop yield and more [2, 3]. Furthering the research and development of technologies for the agricultural industry will depend on the sharing of this conceivably private data.

Collections of agricultural data may be obtained by farmers over years of working with the same agricultural entities (plants, animals, etc). Data collected from consistent entities periodically can hold substantial value in the field of machine learning. Further, model developers are regularly in need of new data for training. These concepts boiled down to the following two questions.

- How can a farmer in possession of an abundance of data distribute it securely to others without fearing a breach in security?
- How can a machine learning model architect in search of data for model training obtain high-security agricultural data from farmers?

The proposed architectures were conceptualized to address these questions.

In this paper I investigate how the owner of a secure data set may deploy a compute instance containing a controlled network architecture that enables one or more machine learning (ML) model owners to utilize the data in training their models while simultaneously maintaining security for both the data set and model training application. The motivation for my research is to alleviate security risks that previously prohibited the sharing and utilization of secure agricultural data.

I propose four secure data exchange architecture's that have been implemented and tested along with one additional theoretical network architecture. These architecture's range in security, throughput, and complexity, but each maintains the purpose of providing the highest level of security consistent with the capacity of its environment and incorporated processes.

The core research of this paper is centered around security. The throughput of each architecture will be discussed, but it is of less significance because fewer resources were consumed to improve the speed of the individual processes that define each network. Moreover, the throughput of some of these processes can surely be improved.

We use two key terms to define the user ends of these network's. Namely, the agricultural data owner or farmer is coined the *Provider*, and the ML model owner is the *Consumer*. The word secure, in the context of this paper, means: 1) The Provider's data is concealed from consumer before, during and following its use; 2) The Consumer's proprietary code is safe from the Provider and their application cannot be run without strict approval. Similar principles are shared by other data exchange networks [4, 5]. These two security expectations will not be fully met until architecture 4. While usage of the data by the simulated consumer in my proposal is model training on agricultural data, the same technologies and theory can be applied to facilitate the use of other types of secure data in many potential operations.

I present the architectures from least secure with lowest complexity to most secure with highest complexity. In doing so, I hope to demonstrate that while the complexity perceived between architectures 1 and 4 seems significant, architecture 4 is achieved by merely adding one or two individual processes from one architecture to the next. The processes that shape the architectures will be deconstructed and discussed. I have conducted my research and analysis under the guise of being as close to reality as possible. This includes selecting a model for testing that was actively being developed on real agricultural data. I opted to use FarmOS, a web interface for handling agricultural data, as the central manager for all the data used in testing. Some of the data used for testing is secure data but some is not. Satellite data was pulled from the open-source Sentinel-2 data set and used in training. However, I chose to consider all data secure and requiring the same level of security. The reasoning for this will be discussed more in the chapter 3 (Approach).

The purpose of presenting a range of architectures is to address the contrasting levels of security a farmer may expect when providing their data to outside parties. I predict, corresponding to alternate data sets containing information with varying degrees of security, certain providers may require a high level of security for the data they hold while others will prefer faster turnarounds of their data use over stringent security. Similarly, Consumers may expect a greater level of security for their model applications and could utilize a similar network on their own compute instance when using lower security data from a Provider.

All of the architectures are designed using the same general set of technologies. FarmStack

Trusted Connector's are used in order to secure the Provider's data while it is in transit from its origin database to the compute instance. The architecture of each network heavily depends on Docker. Each distinct process is contained a docker image. These processes are interconnected through custom docker networks. Multiple networks are used between all of the processes, but each is given stringent policies which only allowed external network access if absolutely necessary. Processes with potential security risks, like the Consumer's training application, are deployed on strict networks with no external access effectively restricting them from potentially sending any information to outside sources.

An important consideration, which isn't addressed until the fifth architecture, is the security risk present when using unencrypted data on any remote cloud compute instance. Cloud compute instances, like AWS EC2 in our research, are considered honest-but-curious which means they have the ability to snoop data actively on their servers regardless of how small likelihood. Even in the third and fourth architectures presented in this paper in which the Provider's data is never written to disk unencrypted, the data still exists unencrypted in memory while being used giving the service providers an opportunity to copy it if they chose to. Nevertheless, this issue may be for the most part disregarded when utilizing a major service provider such as "Amazon Web Services," "Oracle Cloud Infrastructure," "Google Cloud Platform," or "Microsoft Azure" among others.

An expectation consistent in all proposed architectures is that the results of the Consumers training application should be reviewed by the Provider prior to the Consumer receiving them. This must be done to ensure the Consumer did not merely package the Providers data and attempt to output them as results.

However, the Consumer may adjust their training application to only output the information they need without necessarily outputting data that could be revealing to proprietary information about the model architecture. For example, outputting only the resulting trained weights reveals little about the architecture of the model and the Provider can easily validate that none of their data is contained. The outputs of the model used in this study are the best acquired state checkpoint during training and a csv file comprised of the train and validation loss over all training epochs. It is possible to reverse engineer the resulting state checkpoint to obtain the characteristics of certain layers from the model's architecture, but that excludes all layers without weights such as vital activation layers.

The first architecture is used as a baseline for comparing the security and throughput in the other phases of our network. This baseline is the simplest architecture with the highest training throughput, but is considered secure for only the Provider and insecure for the consumer. The only security risk addressed is in the transportation of the Provider's data. Any additional security comes from the analysis of the Consumers docker image for vulnerabilities by the Provider prior to running the network.

The second architecture improves security by leveraging more Docker features. The Consumer's initial application is divided into separate docker containers and a new docker network is used to further isolate the Consumers code. The core training application should be obfuscated by the Consumer prior to dockerizing it concealing their proprietary code. This obfuscation step is maintained for all of the remaining architectures.

The first significant change is made in the third architecture with the incorporation of a cryptographic file storage server which replaces writing to and from the physical disk for all processes. Instead of accessing data on disk directly, requests are made to this server to both write and read data files. This architecture supports the Provider giving open access to the instance during training. The only access that must be restricted is memory access due to the secure data traveling unencrypted in memory. This is certainly more secure then architectures one and two, but see's a decline in throughput due to the time required to encrypt, decrypt and transfer the data over sockets.

The fourth architecture is the final architecture implemented and tested. It incorporates the use of an "Provider Data Handler" and "Consumer Data Handler." These new processes have little to no affect on the throughput of training and further allow the same instance to be shared by multiple Consumer's. The security improvements may be leveraged if the instance is being shared by Consumers, but also makes the pipeline of the entire network a closed loop with no interaction required in the instance following the networks deployment. This is achieved by these handlers which send training outputs directly to the Provider's personal instance for validation then directly to the Consumer's personal instance following approval. An attestation handshake technique is also used by these handlers to ensure the results are going to the correct external source. The security objectives in the threat model for this research were all addressed in architecture 4.

To alleviate the obstacles of both the honest-but-curious service provider and unencrypted data accessible in memory, I contribute the theory of a fifth architecture. Here I briefly describe the utility and security potentials of a new type of compute environment known as a Trusted Execution Environment (TEE). AWS have deployed their own version of a TEE coined Nitro Enclave's. Through the use of these in the future, it will be possible to reconfigure and simplify the proposed architectures to perform all computation and temporary data storage inside an enclave. Unfortunately, enclaves are not currently an option due to their highly restricted hardware access. It is not currently possible to interact with a graphics card from inside an enclave therefor making them unusable for high density graphical model training. New research has shown promising results in the area of secure GPU interaction in a TEE [6]. However, this does not apply to AWS enclaves as was used in this research. The proposed architecture 4 is the a promising alternative as a secure end-to-end network for agricultural machine learning model training.

In this paper I aim to supply farmers with a method for sharing their secure data. Furthermore, decisions in software to use and how to structure pieces of the proposed network have been driven by our goal of making this proposal as close to reality as possible in this simulated environment. This includes the choice to use AWS as a service provider as they are the most used cloud services provider globally [7]. I have also opted to use a popular web-based farm management tool known as FarmOS to format and manage the data I use for demonstrating the network. The network architectures proposed and model training medium for their use are comparable to real-world applications and will provide farmers with a mechanism of distributing their data without the fear of releasing secure information.

2 Background

The rate at which data is collected and stored is increasing exponentially from year to year. In 2014, [8] presented that certain big-data analytics repositories exceeded exabytes. This figure is significantly smaller then what is seen today. [9] stated in 2019 that 90% of the data in the world, at that point, was generated starting in 2017 and was then compounding at a rate of 2.5 quintillion (10^{30}) bytes per day. Today, the practice of collecting vast quantities of data at an exceedingly fast rate has been adopted in just about every corner of the digital world and it is surely not going to waste. Big data has provided many benefits to society including modeling the spread of diseases, crime prediction and financial market analysis [10]. However, two paramount problems with big data are the potential for data leakage and the question of who owns that data [9, 10, 2].

Concern for the security and proper handling of data has been a heightening concern of both individuals and institutions. Stories covering unprecedented data leaks by large organizations, like Adobe in 2013 [11] and a number of educational institutions since 2005 [12], appear too regularly in the news. Worse than the unintentional leaks is the multitude of instances where sensitive data was voluntarily released or sold. A recent and well known example is the controversial 2018 case where 50 million Facebook users personal data was misused by Cambridge Analytica after Facebook sold it to them [2].

Public awareness was raised by the Facebook scandal and other global publications leading to new and stricter data privacy laws around the world [2]. Regarding agricultural data, however, [2] observes, ”for some time now the issue of data misuse and controversy has been well known to farmers and the agricultural community.” Back in 2014, Monsanto, an American agrochemical and agricultural biotechnology corporation, left customers credit card information and other private data exposed on their servers [13]. Another case similar to the 2018 Facebook case in that data was willfully exchanged was *Haff Pultry v Tyson et al.*, in 2017. 38 chicken farmers raised a class action lawsuit against a number of chicken processing companies, including Tyson Foods Inc., ”for allegedly [selling] production data (e.g. grower payments, broiler weights, type of feed and medicine used, and transportation costs) to 3rd parties” in order to keep their prices below competitive levels [2]. These reoccurring instances of cooperate negligence and misuse of data are bound to increase the levels of mistrust the public holds toward cooperation’s holding their data.

The need for more data is only increasing as new technologies enter the market. There has been a recent explosion of digital technologies in agriculture systems giving rise to the relatively new and booming field of precision agriculture (or smart farming) [2, 3, 1]. Precision agriculture, defined in [14], is "the application of technologies and principles to manage spatial and temporal variability associated with all aspects of agricultural production for the purpose of improving crop performance and environmental quality." Autonomous tractors, new genotypes, improvements to pest and weed management, and refined fertilizer use are just a few currently underdevelopment precision agriculture technologies [3]. These technologies present the potential for an exciting future in agriculture, but successfully deploying them will require both the production and utilization of massive new stores of data. This use of data is especially due to the use of machine learning models in many cutting-edge technologies which require large data collections to train [1].

A principal issue precipitating the unnecessarily large volume of data produced today is that much of it may not be discoverable, interpretable and/or reusable [1, 3]. Copious amounts of data are produced by agricultural research institutions but is most often stashed in private repositories due to the secure nature of the data [1]. Furthermore, researchers may be unsure of what data will be useful resulting in data that is stored raw and unformatted and, therefore, ultimately unusable [15]. My proposed research addresses the issue of discoverability in that its primary goal is to facilitate the sharing of secure data. I adopted an open-source software, built and designed by a community of farmers, known as FarmOS, which is a web interface for farmers to enter and maintain agricultural data and information. Furthermore, they have developed a custom data structure for maintaining the data which enables easy interpretation and is reusable for others who may use that data. FarmOS have not only defined data structures for many agricultural entities, but made it trivial to expand them in order to develop custom data structure's using their schema.

Agricultural data privacy laws surely have a ways to go in order to protect farmers from untrustworthy institutions. This issue, however, paired with the problem of data being leaked in breeches has led to the extensive research and development of new technologies which put precedence on data security and allow for operations, especially in machine learning, to be performed without compromising security. These new technologies include trusted execution environments (TEE), cloud operated machine learning as a service and fully homomorphic encryption among others [16, 17, 18, 19]. In this paper I express a collection of architectures built on a combination of some of these technologies as well as others for trusted and secure machine learning model training. A key software utilized is FarmStack. Digital Green is developing FarmStack as a peer-to-peer network protocol which secures data in transit through periodic attestation, network policy enforcement and endpoint application enforcement. Providing agricultural data owners with the means to allow others to use their data securely is the primary goal of the proposed data sharing and model training network architectures.

2.1 Related Works

OmniLytics, proposed by [4], is "a blockchain-based secure data trading marketplace for machine learning application." It shares the key principle of security for both the data from the model owner and model security from the data owner. OmniLytics, however, is purposed exclusively for machine learning applications while my architectures may be manipulated for different uses. Moreover, OmniLytics differs from my approach through its use of functional encryption (FE) to secure data.

Another project with the akin fundamentals of data security is Agora [5]. Agora is similar to OmniLytics in that it is a data exchange marketplace running atop a blockchain utilizing smart contracts and crypto tokens for security and verification. Also shared with Omnilytics is Agora's FE use to secure data. Agora allows for more applications then OmniLytics by relying on a central "Data Broker" who essentially runs functions on the encrypted data and sells the functional output to consumers.

The primary contrast between my method and both Agora and OmniLytics is the facet of it being a marketplace. My proposal is unique in that it is not a marketplace at all but a self contained peer-to-peer network. I do not address any method or escrow style system for ensuring anything beyond the security of the Provider's data and Consumer's model. Further, the architectures I propose are purposefully malleable to allow alterations to fit distinct needs.

3 Approach

In this chapter, I will provide all the research and development applied to deliver the aforementioned secure data training network architectures. To begin, I will contribute my expected use case for this research and the resulting network architectures followed by a description of the machine learning model training application used to demonstrate these networks. Immediately following, I will briefly discuss the data used to train the model. The next few sections will comprise of the technologies used in these network architectures with documentation, explanation and the use-case of each. The remainder of the chapter will contain documentation on the architectures. I will briefly supply a description of the network architectures from an external perspective excluding each instances internal network. Finally, each architecture will have a dedicated section, appearing from least to most secure (1 - 5), in which I will discuss the architectures themselves in granular detail, such as their contained processes, and the comprehensive depictions of their data routes.

3.1 Threat Model

The threat model proposed here is only completely address by architecture 4. Architecture 1 addresses very few aspects of the threat model because it is a baseline for comparing the security improvement for the others. Architectures 2 and 3 increasingly address more facets of the threat model. What each architecture resolves from the threat model and what it is missing will be discussed in that architectures respective section in chapter 5. For now, I will only provide the complete threat model which is shown in table 3.1.

3.2 Use Cases

The aim of the architectures presented in this paper is to enable secure agricultural data owners to distribute their data for training machine learning models. This results in different use cases for the Provider of the data and the Consumer of that data.

3.2.1 Consumer Use Case

There are certainly few use cases for data that can be used but not seen. Model training only requires the use of the data for a limited time and the results of the training depend on the architecture of the model. Training a model does rely on model developers viewing the data. Direct view of the data may aid the model developer in better tuning their training parameters, but this could also be achieved through the provision of detailed meta data prior to training

Table 3.1. The Threat Model

Each column has a *Security Objective*, related *Risks*, and *Solutions* to the risks

	1	**2**	**3**
Security Objectives	Data secure in transit over internet from source to shared instance.	All data (raw, generated, training) secure on shared instance.	Training results are the only data that can be moved off the instance and only once when training completes.
Risks	Data Interception during internet transfer	High density data must be stored on disk	Consumer's application could export data to external server
	IP Spoofing of Shared Instance	Consumer can read disk data	Consumer's application could store Provider's data as fake training results.
			Data Interception during internet transfer
			IP Spoofing of Consumer or Provider Instances
Solutions	FarmStack Trusted Connector	Internal Data Handler encrypts all data before writing it to disk rendering it unreadable.	Consumers application has no direct external network access
			Provider must review and validate results before they are exported to Consumer
			All data is encrypted before traversing external networks
			Attestation is used in verifying destination instances

	4	**5**
Security Objectives	Consumer's proprietary code is secure from Provider	Container's cannot perform unspecified communication with other containers or machines
Risks	Provider can access code after pulling Consumers docker image.	If Consumer's application container has docker.sock access it can communicate with any other running container.
		Container's sharing the same network can always communicate with each other
Solutions	Consumer's proprietary code is obfuscated	The only container's with access to docker.sock are the Provider and Consumer ends of the FarmOS Connector. Both are owned by the Provider, so they are not a security risk.
	Consumer's application runs an attestation processor through CDH to CDC. It will only run if the handshake is valid.	A web of docker networks was deployed so that containers only share a network with another container that they need to communicate directly with.

	6	**7**
Security Objectives	Containers and their internal processes cannot be altered after they are started	Consumer's unknown training application cannot pose any malicious threats.
Risks	Consumer could alter container code or inject new code into running container	Consumer's application could be malware
Solutions	All containers file-systems are set to read-only	Container filesystem is read-only. It only has access to a single read-only disk volume which doesn't contain secure data. It cannot communicate with host machine.
	Consumer's shared instance role is read-only	

by the data owner. Moreover, obtaining accurate training results on data that is concealed is contingent to thorough meta data. Investigating the proper meta data writing practices for data sets used in machine learning is not discussed in this paper but it is related and an active research topic[20, 21].

Beyond the structural purpose of the proposed network architectures, this paper is dedicated to training a model on agricultural data specifically. This, in part, is due to the less strict nature of security for this data. While agricultural data may be considered secure data, the level of that security is more to the discretion of the data owner rather then other types of data, for instance, medical and financial data. For this reason, an architecture may be selected based on the discretion of the data owner who may opt for more or less security.

All things considered, the following list contains the expected characteristics of the perceived use case for these architectures.

- ML Model Research and Development
- Agricultural Data
- Data that is not Readily Available

3.2.2 Provider Use Case

There is a plethora of use cases a Provider may have for one or more of the individual processes used in defining the proposed secure data networks. However, monetary gain is the chief reason a Provider may choose to use a network which hides data access. For any other feasible reason in distributing their data, the Provider would almost certainly need to allow the viewing of the data.

3.3 ML Model

While the focus of this paper incorporates Machine Learning, model training specifically is not in scope. Therefore, I will describe my reasoning behind selecting the model chosen for testing the architectures and provide enough background to comprehend its purpose and relation to my research. In chapter 4 (results), I will provide a compact selection of training and validation loss values. However, the only significance of these results to my research is that they are consistent across all architectures. Consistency in the training results across each distinct architecture and when trained on a normal machine outside of any architecture proves that the expected behavior that the proposed network's have no effect on the training results.

3.3.1 Selecting a model for testing

Rather then using a preexisting model for analysis, I believed it was better to use a model that matched the expected Consumer use case of these architectures. When I began this research, another member of my lab, Hamid Kamangir, was actively in development of a geospatial remote sensing model for grape yield estimation on vineyards. This model fit well within my conceived use cases as it was actively in development and required agricultural yield data from vineyards which is not freely available. Moreover, it can take a considerable amount of time to train a model on high-density geospatial data. The lengthy training time was a benefit to my research because it enabled clearer throughput data comparisons between each distinct network architecture. In other words, minimal slowdowns in a processes contained in a particular architecture would be amplified during the complete training-cycle making it easier to compare.

3.3.2 Model Definition

The chosen model for testing was a custom deep learning architecture comprised of UNet and Convolution LSTM trained on geospatial imagery. This model is being developed as a generalized yield predictor for many different grape cultivars across different farms. The only change I made to the model after it was given to me was narrowing the cultivar scope for training from nine cultivars down to one. The outcome led to better training results and reduced training time. Even with a single cultivar, this model could take upwards of three to four hours to train in the third and fourth network architectures compared with 30 minutes to an hour in the first two, so throughput analysis was still very clear.

3.4 Data

Four years of yield observation's from the Livingston vineyard paired with time series Sentinel2 satellite images of the farm over the same time period formed the data set used for training the model. The Sentinel2 data set is freely available to the public. However, I chose to regard the satellite imagery as secure data that would be packaged with the Provider observation data under the pretense that certain Provider's may wish for complete anonymity including the location their data set was acquired.

See table 3.2 for specifics on the dataset including quantities and data sizes. The "Used" file count and size refers to the data that was used during the network. This includes years 2016, 2017 and 2019. The 2018 data was used as the test data set after the model was already trained. The training set is made up of data years 2016 and 2019. The 2017 data is used as the validation set.

Table 3.2. Raw Data Set Specifics

Year	*2016*	*2017*	*2018*	*2019*
Yield Data File Count	60	60	56	44
Yield Data Size (Mb)	81.98	67.13	69.51	55.01
Satelite Image Count	15	15	15	15
Satelite Image Size (Mb)	94.38	94.78	94.43	94.36
Total File Count Per Year	75	75	71	59
Total Size Per Year (Mb)	176.36	161.90	163.94	149.37
Total File Count	**280**		**Used Total File Count**	**209**
Total Data Size (Mb)	**651.58**		**Used Total Data Size (Mb)**	**487.64**

3.4.1 Simulating Data Collection

The data used for training the model was provided as a source for research in my lab at UC Davis. However, I felt it was useful to simulate the collection of this data to promote the true utility of the architectures. I also considered the fundamental issue in the disorganized and unformatted storage of many raw data sets. I chose to use the farm management application FarmOS to address both the simulation of data collection and storage of the data in a operable format. Paired with FarmOS as the interface, I opted to use an AWS S3 Bucket to store the raw data files after they had be attached to logs through FarmOS. I will discuss both AWS and FarmOS more in depth in the section 3.5.

3.5 Technologies Used

In the next section 3.7 I will begin discussion of the network architectures, but first, I will describe the distinct technologies used in their deployment. The two core technologies utilized in the development of these networks were Docker and FarmStack Trusted Connectors. Furthermore, FarmOS and AWS were two utilities that played key roles.

3.5.1 Docker

Docker is used extensively in the deployment of these networks. Some of this research would not be possible without it. Specifically, FarmStack Trusted Connectors, as will be discussed next, rely on Docker as a method to enforce application-to-application security. Docker is an extensive technology which offers many services. Here, I will only discuss the aspects used directly in this paper.

Docker container's are a virtualization technology somewhat resembling very lightweight virtual machine's (VM). However, they differ in that docker containers are pre-packaged application environments running on top of a hypervisor, the Docker Engine, directly on top of a machine's operating system (OS). Most VM's are essentially copies of an entire OS running on top of a hypervisor on top of the primary OS [22].

3.5.1.1 Docker Images and Containers

Docker containers are deployed from docker images. Docker images are created using dockerfiles which are a set of ordered commands which first specify another image as a base and any data or information you want to pass to the container. When creating an image, the Docker Engine takes all the information in image file and combines them in a set of hashed layers in which the Docker Engine, or more specifically the Docker Daemon (dockerd), packages with a deployment engine on top [23]. Dockerd is the core process that manages all containers and the Docker Engine contains both dockerd and its CLI (docker). Each image is assigned an id which is created by hashing its layers with SHA-256. Therefor, all images have there own unique id's.

Containers are launched by specifying an image. When launched it is a standalone entity independent from the image. It contains its own file system, IP, ports, and OS all managed by the Docker Engine. Multiple containers can be run side by side using the same or distinct images. Each container is assigned a unique ID when launched that can be used to reference it. Docker containers can be stopped and started, but remain independent until removed.

A containers file system differs from the host instance's file system in that it does not persist between resets. The data contained in the containers file system is still in the host file system, but it is managed by the Dockerd and will be removed when the container is terminated. Container file system data is also only accessible by the root system user, so it cannot be tempered with by non-root users.

3.5.1.2 Docker Networks

Docker networks are another virtualization technology possible by the dockerd. When dockerd is run it initializes a bridge network with the host to enable communication. It is also possible to remove a container from the network entirely, but it would then not be able to communicate at all. While containers run as independent entities with a uniquely assigned IP on the bridged dockerd network by default, custom networks can be deployed as well utilizing one of the network driver options available or with a custom driver. Drivers option built into the dockerd include the default bridge, host, overlay, ipvlan, macblan or none [24]. However, all the containers in my architectures run on the bridge network.

Further, these networks can be both internal or external. Communication between containers can be routed through the host by linking a containers port to a host port. Containers on the same machine and the same docker network can communicate directly through the dockerd without forwarding the containers ports to host ports. Containers on the same network can also be called by their name rather than IP. This simplifies communication and improves scalability as IPs are unique to containers and not images.

3.5.1.3 Docker Compose

Docker Compose does not add any new features to docker. Rather, it is a utility that simplifies the deployment of docker services using yaml files. Their custom yaml schema translates directly to docker commands and essentially streamlines the startup of multi-container applications. Once the file is written, you can spin all defined services up or tear them all down with a single command. It is a very useful tool for systems like the architectures I present, which use multiple docker containers simultaneously with specific properties, as it makes it possible to initialize the entire network with a single command.

The properties defined in nearly all service's in each architecture's Docker Compose are the following.

1. image: the docker image name for the service
2. env_file: path to file containing environment variable definitions for container
3. ports: list of all port mappings from host to container
4. networks: list of all networks shared with the service

5. extra_hosts: list of mappings of external IP's to reference names

6. depends_on: list of other services that must start before this one

7. read_only: sets the containers file system to read-only

8. volumes: list of local or remote directory paths whose contents are shared with the service

    (a) ':ro' at end of volume mapping: set volume to read only

9. shm_size: allotment of memory shared with the host system

3.5.2 FarmStack Trusted Connector

The Trusted Connector is an an open-source peer-two-peer (P2P) network protocol powering the secure transfer of data. It also guarantees application-to-application security beyond the P2P connectivity and attestation. I utilized the FarmStack team's implementation of the Trusted Connector. FarmStack has been utilizing this technology to facilitate security for farmers data while in transit from the farmer to its destination. The original publisher of the Trusted Connector was Fraunhofer under their Industrial Data Spaces initiative. "The Trusted Connector is an IoT edge gateway platform ... [that] is a reference implementation of the International Data Spaces security architecture and trust profile" [25]. It was developed using the Spring Boot architecture and provides: message routing and conversation through a custom Apache Camel component; application isolation through containers; data flow and usage control through IDSCP2 [25].

As Trusted Connectors are a new technology, I will go into greater detail here on how they work. I will start by providing an overview of the network of a single one-directional connector. Then, I will deconstruct the individual processes it contains and the technologies that make it possible. Trusted Connectors can also be deployed to permit bi-directional data flow. A bi-directional Trusted Connector is effectively the same as two one-directional connectors layered on top of each other sharing the same process instances. These will also be shown as the primary Trusted Connector used in the proposed architectures is bi-directional.

3.5.2.1 Effective Architecture

The high-level architecture of a one-directional FarmStack Trusted Connector can be seen in figure 3.1. It is made up of 4 individual Docker Containers. These are the "provider_app",

”provider_core”, ”consumer_core”, and ”consumer_app.” The consumer_core is the broadcasting server and the provider_core is the client.

The provider_app and consumer_app are the source and destination points of the Trusted Connector respectively. The provider_app does the initial selection and any pre-processing of the data then uploads it to the provider_core. After the provider_core receives the complete data file, it constructs a message and sends it to the consumer_core. The consumer_app is the application that will receive the providers data from the consumer_core to use it within defined bounds.

The provider_core and consumer_core perform routing verification and act as the transport layer in the connector. They handle all the data routing between the provider and consumer applications as well as periodic attestation for continuous validation. While there is communication between the provider_core and consumer_core in both directions, the connector is considered one-directional because the communication between the provider and consumer applications follows a one directional path, as shown by the arrows in figure 3.1.

A simplified flowchart of the Trusted Connector can be seen in figure 3.2. This figure provides a good understanding of what is happening inside the provider and consumer core processes to an exceptionally basic degree. The provider_core is an IDSCP2 client which forms a secure tunnel with consumer_core. The consumer_core is an IDSCP2 broadcasting server. The provider as the client and consumer as the server can be reversed. However, the layout shown is the common structure for FarmStack Trusted Connectors and is what I used in the proposed architectures.

The data flow for a one directional connector from the Provider to Consumer resembles the following given the consumer is the broadcasting server and the provider is the client. The consumer_core is started first and begins listening for connection requests for clients. Next, the provider_core is started, and it searches for the consumer_core server to establish a connection.

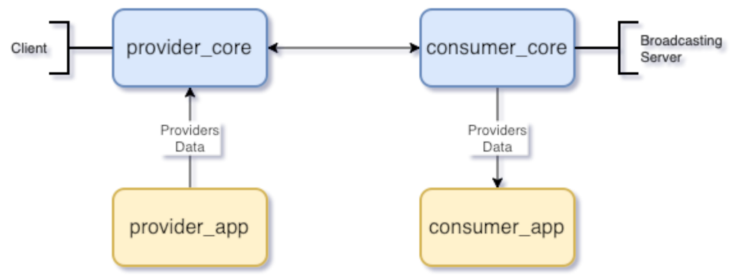


Figure 3.1. One-Directional Trusted Connector Overview

When the connection is made, the provider_core initiates an attestation handshake to validate the authenticity of the consumer_core. Once the handshake is approved, data will flow from the provider_app to the provider_core then through the consumer_core to finally the consumer_app.

The difference of the bi-directional connector is that data can also flow from the consumer_app back to the provider_app. For this reason, we denote the data flowing between the two ends of the connector as requests in figure 3.3. The setup is still the same for the most part, but additional logic is required in the connector to allow for the reverse transfer. It is also important to note that in a bi-directional connector, the sense of a *provider* and *consumer* is essentially lost. I still use this terminology in figure 3.3, but it doesn't hold the same value because any data could be flowing between the two sides. However, the consumer_core is usually deployed as the broadcasting server, hence the notation in the figure.

In the remainder of this section I will only discuss the one-directional connector. Once the methodology of the one-directional connector is comprehended, the bi-directional can be

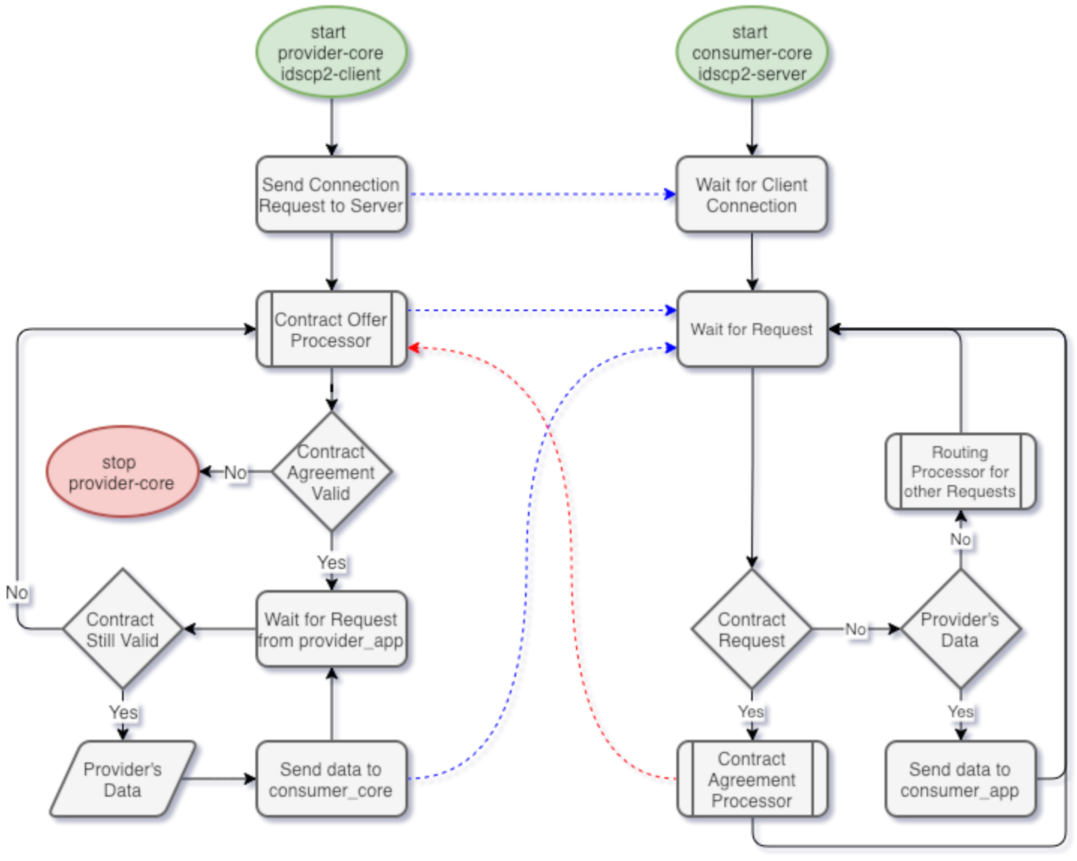

start provider-core idscp2-client
start consumer-core idscp2-server
Send Connection Request to Server
Wait for Client Connection
Contract Offer Processor
Wait for Request
stop provider-core
No
Contract Agreement Valid
Yes
Routing Processor for other Requests
No
No
Contract Still Valid
Wait for Request from provider_app
Contract Request
No
Provider's Data
Yes
Yes
Yes
Provider's Data
Send data to consumer_core
Contract Agreement Processor
Send data to consumer_app

Figure 3.2. One-Direction Trusted Connector Simplified Flowchart

deployed. As stated previously, this is because the bi-directional is essentially two one-directional connectors running in parallel with directions reversed but using the same core processes. Later in this chapter I will provide a diagram depicting the Trusted Connector used in the proposed architectures. That connector facilitates requesting various types of data including raw text objects and complete files from a web server database.

3.5.2.2 Connector Opposed to Direct Data Transfer Rate

In generic terms, direct data transfer means a consumer is downloading data directly from its source. This source could be anywhere data is normally stored like a personal machine, local or remote server, etc... The data is already stored at that source, so only one download step is required.

Trusted Connectors route data through multiple processes before the consumer may download it. This is shown in the steps below.

Step wise data transfer through connector

1. provider_app downloads the data from source.
2. provider_app uploads the data to the provider_core.
3. provider_core constructs a message and sends it to the consumer_core.
4. consumer_app downloads the data through the consumer_core from the provider_core.

Data transfer through Trusted Connectors is expected to be two times slower than direct data transfer. This is a consequence of the way data is transmitted through Apache Camel[1].

This slowdown is evident in the steps listed above. Rather than a direct transfer, which requires a single download, transfers through connectors require an additional download and

[1]Read section 3.5.2.4 for more info on Apache Camel)

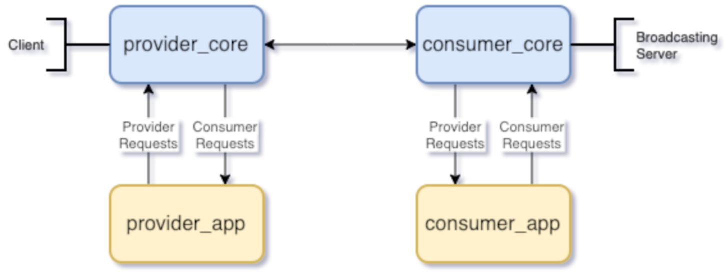


Figure 3.3. Bi-Directional Trusted Connector Overview

upload. These are steps 1 and 2 in the list above. The upload time is negligible because the provider_core and provider_app are on the same machine. Nevertheless, the additional remote download step means there are twice as many downloads when using a connector over direct. Hence, the expected total transfer time per file is approximately doubled.

The first download of the data by the provider_app is because the data is stored on a server that is remote to the Provider's instance containing the provider_app and provider_core. If the data was stored locally on the instance that contained the provider app and core, the transfer time would not be doubled because there would only be a single download.

3.5.2.3 IDSCP2 Network Protocol

The underlying network protocol beneath the Trusted Connector, empowering the secure channel communication, is the Industrial Data Spaces Communication Protocol 2 (IDSCP2). It is a TLS/TCP/IP based transportation layer protocol which, at its base, is a complex event driven finite state machine including ten individual states and 24 possible transitions events per state (240 transitions total) [26]. A copy of the simplified IDSCP2 finite state machine, taken from [26], can be seen in figure 3.4. The second step on the provider_core side of Figure 3.2 is the "Contract Offer Processor". This block implies that the heavy lifting done by the IDSCP2 protocol prior to the transfer of data between the provider and consumer. That is the end-to-end security through remote attestation and application verification. The following is a statement from the official IDSCP2 Repository [26].

> The goal of the IDSCP2 handshake is to reach the STATE_ESTABLISHED, which allows the IDSCP2 communication with the remote peer. To achieve this state, first the handshake has to be started ..., valid DATs have to be exchanged and verified, and finally the RA [Remote Attestation] drivers have to verify each other's trusted state.
> During the handshake, as well as after the protocol is in STATE_ESTABLISHED, re-attestations and fresh DAT requests can be triggered to keep the communication trusted over time.

This complex state machine enables many customizable security measures through the use of IDS Usage Control Policies. These policies facilitate repeated periodic verification, application enforcement (discussed more in 3.5.2.5), time restricted use and more. These policies must be

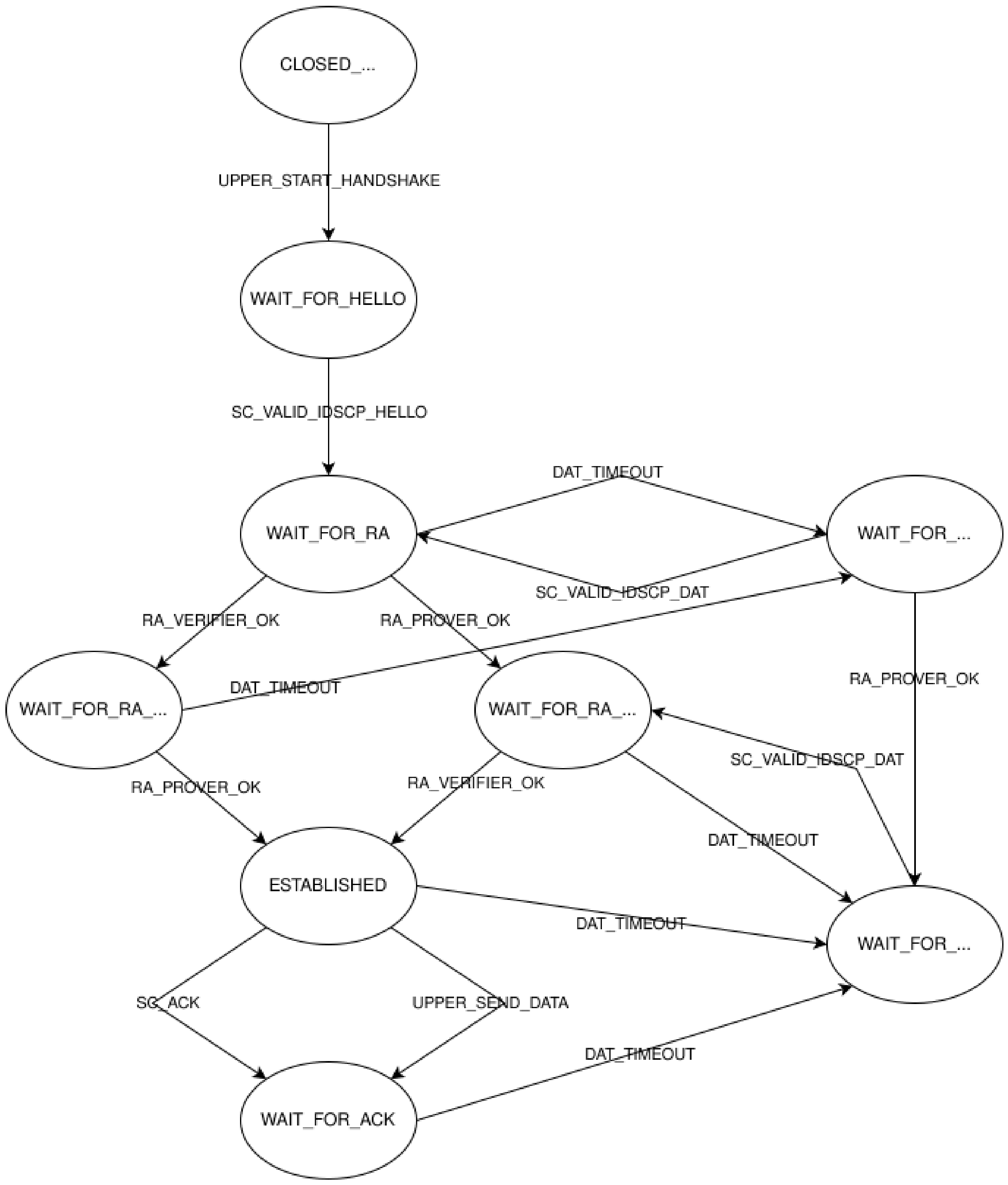


Figure 3.4. Simplified IDSCP2 Finite State Machine [26]

created for route definition and are written in the IDS Usage Control Language. This language is based on the Open Digital Rights Language (ODRL).
A complete list of current connector supported IDS Usage Control Policy Patterns [25].

- Allow the Usage of the Data - Unrestricted access to the data
- Connector-Restricted Data Usage - Allows data usage for a specific connector
- Interval-Restricted Data Usage - Allows data usage within a specific time interval
- Duration-Restricted Data Usage - Allows data usage for a specified time period
- Restricted Number of Uses - Allows the usage of the data n times
- Use Data and Delete it After - Allows data usage in a specified time interval with the restriction to delete it at a specified time stamp
- Local Logging - Allows data usage if logged
- Remote Notifications - Allows data usage with notification messages

In the proposed network architectures, the "Connector-Restricted Data Usage" and "Local Logging" are used. In practice, the "Duration-Restricted Data Usage" could also be employed if the provider had an estimate on the duration of the model training.

Essentially, ODRL is a base level vocabulary of general machine readable contracts with a base level data model. IDS then further extends the ODRL descriptions to define usage policies and an enforcement methodology through contracts. There are 3 types of these IDS contracts: requests, offers, agreements. A policy is final when all 3 of these contracts are successful over a connector.

3.5.2.4 Apache Camel

Apache Camel is an open-source integration framework empowering custom routing and mediation rules. A custom component was written for Apache Camel by IDS for their Trusted Connector. In order to deploy a Trusted Connector, it must be given route definitions to know how and where to route data. Both the provider and consumer core require custom routes for how to handle data to send and received data. Apache Camel can be written in multiple different data serialization languages (XML, YAML, DSL), but XML was used for the routing in this proposal. Therefor, below I provide example Apache Camel route definition files in XML. See figure 3.2 for a high-level flowchart of these routes.

An example Provider route definition file may include, but is not limited to, the following:

1. Processors - Used by routes for processing requests
   - Type Extraction Processor
   - Contract Offer Creation Processor
   - Contract Agreement Receiver Processor
   - Resource Update Creation Processor
2. Routes
   - Contract Offer and Validation Route – See Example Code in *Provider - Contract Offer and Validation Route*
   - Send Data Route – See Example Code in *Provider - Send Data Route*

Provider - Contract Offer and Validation Route The following enumerated list provides a description of the numbered lines in the example xml *contract offer and validation* route [27].

1. repeatCount - Specifies a maximum limit of number of fires.
2. The containerUri constant is a requirement of the contract offer and should be the docker image URI for the Consumers application. This URI is contained in the contract agreement and compared with the actively running Consumer Application. If they do not match, the contract offer fails.
3. The Processor interface processes the message into correct format.
4. Accepts incoming message to process.
5. Checks message and it's type based on its headers.
6. The Content Based Router from the EIP patterns allows you to route messages to the correct destination based on the contents of the message exchanges.

7. Returned contract agreement was invalid.

```xml
<route>
   <!-- timer creates a loop -->
1. <from uri="timer://contractRequest?repeatCount=1" />
2. <setProperty name="containerUri"><constant>
     https://hub.docker.com/.../sha256-31dd26dc5f7c0a43...eca727b4afefdc6fc
   </constant></setProperty>
3. <process ref="ContractOfferCreationProcessor" />
4. <to uri="idscp2client://consumer-core:29292"/>
5. <process ref="TypeExtractionProcessor"/>
6. <choice>
     <when>
       <!-- Check for valid contract message  -->
       <simple>
         \${exchangeProperty.ids-type} == 'ContractAgreementMessage'
       </simple>
       <process ref="ContractAgreementReceiverProcessor"/>
     </when>
     <otherwise>
7.     <log loggingLevel="ERROR" message="Invalid Agreement Found"/>
     </otherwise>
   </choice>
</route>
```

Provider - Send Data Route The following is an example xml *send data* route including brief comments to describe specific lines [27].

```xml
<route id="sendData">
  <!-- <from uri="direct:update"/> -->
  <from uri="file://deploy/?fileName=sample_data_file.csv&amp;charset=utf-8&amp;noop=true"/>
  <!-- ensure that contract agreement is made before sending-->
    <delay asyncDelayed="true">
    <constant>10000</constant>
  </delay>
  <log message="Sending data."/>
  <!-- Parse CSV -->
  <unmarshal>
    <csv delimiter="," useMaps="true"/>
  </unmarshal>
  <!-- Serializes maps to JSON -->
  <marshal> <json/> </marshal>
  <process ref="ResourceUpdateCreationProcessor" />
  <to uri="idscp2client://consumer-core:29292?awaitResponse=true&amp;
    connectionShareId=ucConnection&amp;sslContextParameters=#clientSslContext&amp;
    useIdsMessages=true"/>
  <!-- Sets the correct header for the text answer "OK" -->
  <setHeader name="Content-Type">
```

```
    <constant>application/json</constant>
  </setHeader>
</route>
```

An example Consumer route definition file may include, but is not limited to, the following. Note that the Consumer only needs one route in this case. This one route can handle both the contract agreement and response as well as accept and route provider data to the consumer app endpoint.

1. Processors - Used by routes for processing requests
    - Type Extraction Processor
    - Contract Offer Processor
2. Routes
    - Receive Data Route – See Example Code 3.5.2.4

Consumer - Receive Data Route The following list provides a description of the numbered lines in the example xml *receive data* route [27].

1. Checks message and it's type based on its headers.
2. The Content Based Router from the EIP patterns allows you to route messages to the correct destination based on the contents of the message exchanges.
3. Check for valid contract message.
4. Process only if property is ContractOfferMessage
5. Process only if property is ResourceUpdateMessage
6. If received header doesn't match to a known header.

```
<route>
  <from uri="idscp2server://0.0.0.0:29292?
    sslContextParameters=#serverSslContext&amp;useIdsMessages=true"/>
1.<process ref="TypeExtractionProcessor"/>
2.<choice>
```

```
    <when>
3.    <simple>\${exchangeProperty.ids-type} == 'ContractOfferMessage'</simple>
      <log message="### Handle ContractOfferMessage ###"/>
4.    <process ref="ContractOfferProcessor"/>
    </when>
    <when>
5.    <simple>\${exchangeProperty.ids-type} == 'ResourceUpdateMessage'</simple>
      <log message="### Handle ResourceUpdateMessage ###"/>
      <setHeader name="Content-Type">
        <constant>application/json</constant>
      </setHeader>
      <to uri="http://consumer-app:8081/post_data"/>
    </when>
    <otherwise>
6.    <log loggingLevel="ERROR"
        message="Expected ContractOfferMessage or ResourceUpdateMessage"/>
    </otherwise>
  </choice>
</route>
```

3.5.2.5 Docker

Trusted Connectors rely on Containers. Docker was used as the container system in this proposal, but they also support TrustMe OS containers. TrustMe OS is described in [25] as a ”highly secure container solution for embedded and mobile systems”. However, TrustME OS was not used in this proposal, so it will not be discussed further. Many aspects of Docker are utilized in the Trusted Connector including networks, image hashes, and shared volumes.

Docker networks are used to link the individual containers together. A single connector uses three networks generically named, provider-internal, consumer-internal and ids-wide. provider-internal covers the provider_core and provider_app containers. consumer-internal similarly contains consumer_core and consumer_app. ids-wide links the provider_core and consumer_core. In this setup, the only communication between the provider and consumer application must be routed through the core connector processes. This can be seen in figure 3.1.

Shared volumes are used to share a folder between the host machine and containers. These shared folders are to provide the key-pairs for the attestation drivers, routing definitions and log directory to the connector cores. They are shared through volumes because this information should be private and, therefore, should not be stored in the docker images.

Image hashes are significant in the ability to provide app-to-app security. Each containers SHA256 hash is unique to that image alone. Any alteration to the code contained in the image will result in a different hash. This means after a consumers application is agreed upon by the

Consumer and Provider then containerized, the consumer is incapable of incorporating anything else into that image. For this reason, the Consumers application should be reviewed by the Provider prior to deploying the connector to ensure it does not contain anything malicious.

Other Docker utilities are used as well, but they do not require further documentation here.

3.5.3 FarmOS

FarmOS is an open-source web-based application for farm management, record keeping and planning. It is utilized in this proposal for two key reasons. First, it follows my goal of providing these architectures in a real world scenario. Moreover, this is an application used by farmers to store potentially secure data which they may want to share with others. Second, the developers of FarmOS have defined general purpose data structures for formatting agriculture data.

They provide a set of predefined data structures organized into the categories assets, quantities and logs. Assets represent things being tracked or managed. Quantities represent quantitative data in granular units referenced as single data points within a log. Logs represent events that take place in relation to assets and other records.

Each category has further predefined types for that category such as harvest, observation and seeding for logs. The team has also enabled the extension of these data structures for custom use cases. As mentioned in chapter 2 (background) of this paper, a major problem with agricultural data sets today is the disorganized and inconsistent formatting of data records between farmers. While they do not fit *every* agricultural institutions potential data formatting requirements, these FarmOS data structures are a solid start in providing a streamlined human and machine readable format for agricultural data.

In the proposed architectures, Harvest and Observation logs were used to store yield data and satellite images as attached files respectively. Each Harvest was named by the farm block the attached yield data referenced and contained geospatial data formatted as geojson which outlined the physical geographical location of that block. The Observation logs similarly contained geospatial locations for the farm, but the locations were for larger portions of the farm which contained multiple blocks. Each log also contained other attributes such as the timestamp the data was collected at. When the logs are pulled, all of this information is considered private to the provider and can not be revealed to the Consumer. However, the consumer application can parse and process this information automatically without the Consumer seeing it, as the Consumer is provided access to the empty data structures containing this data as meta-data for their application.

3.5.4 AWS

Amazon Web Services (AWS) was used extensively in this proposal. For the architectures specifically, I used AWS EC2 instances for cloud computing and AWS S3 Bucket's for data storage. I chose AWS because it is currently the most used remote cloud computing service and in-part due to the theoretical fifth network architecture which utilizes AWS Nitro Enclaves.

3.5.4.1 EC2 and S3

A single EC2 instance was used to host the Shared Instance which contained each secure network architecture. I used the *g4dn.xlarge* instance type to support the network. While g series cards are notably more expensive then other instance types, it was required for GPU access as the model training application desperately needs a GPU for training.

g4dn.xlarge Hardware Specifications

- 1 GPU
- 4 vCPUs
- 16Gb Memory

As will be shown in the chapter 4 (results), a better instance could be used to improve some results as architectures 3 and 4 both consume upwards of 99% of the available instance RAM.

A single S3 bucket was used to store all of the yield and satellite imagery data files attached to each log in FarmOS.

3.5.4.2 Nitro Enclave

AWS Nitro Enclave are Trusted Execution Environment's (TEE) which "are fully isolated virtual machines [that] have no persistent storage, no interactive access, and no external networking" [28]. Essentially, Nitro Enclaves are self-contained virtual machines that use cryptographic attestation when any data is sent to or pulled from it. All communication with an enclave goes through the single local channel that is opened to its parent ec2 instance when the enclave is initialized. Apart from this channel, enclaves are completely isolated from outside interaction. All information on Nitro Enclaves comes from the AWS documentation 3.5.4.2.

Each time an Enclave is spun up it generates a new private and public key pair. The enclave public keys are managed by the AWS Key Management Service (KMS). This is the KMS of the account that owns the EC2 instance which contains the enclave. This key must then be used to encrypt all data that goes in to the enclave. The enclave decrypts this data using its stored private key. There is more to it, but enclaves are not used in practice in this proposal. They are just briefly mentioned in the architecture 5 theoretical documentation.

Similar to virtual machines, Nitro Enclaves cannot actively share their parent instances memory and processor like docker containers. Instead, they required hardware to be partitioned and dedicated to the enclave while it is running. This poses the small issue of being forced to allot enough processing power to the enclave to support its contained application at peek its peek requirements. Containers do not share this problem as they use only what they require at any given instant. [28]

Their are many use cases for enclaves due to their guaranteed strengths in security. However, there are a number of limitations to enclaves in their current state. The limitations most relevant to this paper are its inability to utilize a GPU and only a single enclave may be initialized per EC2 instance.

In promoting their secure and isolated infrastructure, they are only provided with essential hardware [28]. The hardware deemed essential for enclaves excludes GPU's which poses an issue for applications that require GPU capabilities. Due to this, Nitro Enclave's cannot be used as environment's for graphical machine learning model training in their current state.

3.6 Overview of the Architecture's

The secure training network architectures contained in this paper do not have accompanied applications that are available. However, the structures and details described here on each architecture can be used as references for new implementations. [2]

Background information on the individual technologies used and other knowledge required in comprehending the architectures was detailed in this chapters previous sections. In this section, I will provide an outline for the remainder of this chapter and list the contents in each of the architecture sections. The following sections will include a description of external instance layouts and a deconstruction of each of the first four architectures in order of complexity and the theoretical architecture 5.

[2]Digital Green's FarmStack team is actively developing an application to easily create Trusted Connectors through a UI. [27]

The sections describing the proposed network architectures will be organized as follows.

1. General Description – A brief overview of the network
2. Core processes – Each of the named boxes in the data flow and network diagrams
3. Data flow – Stages in the flow of data through the network split into to sections. 1) Container Data Flow 2) Complete Data Flow
4. Docker networks – Description of each architectures Container and Network Structure diagram
5. Docker Compose – The docker-compose file used for the architecture with documentation

3.6.1 Diagrams

Core-process and sub-process diagrams depicting process structure and connections are presented for each architecture. Use figure 3.5 as a key for both of these diagrams. The sub-process diagrams will be come first to provide a complete view of the architecture. The core-process

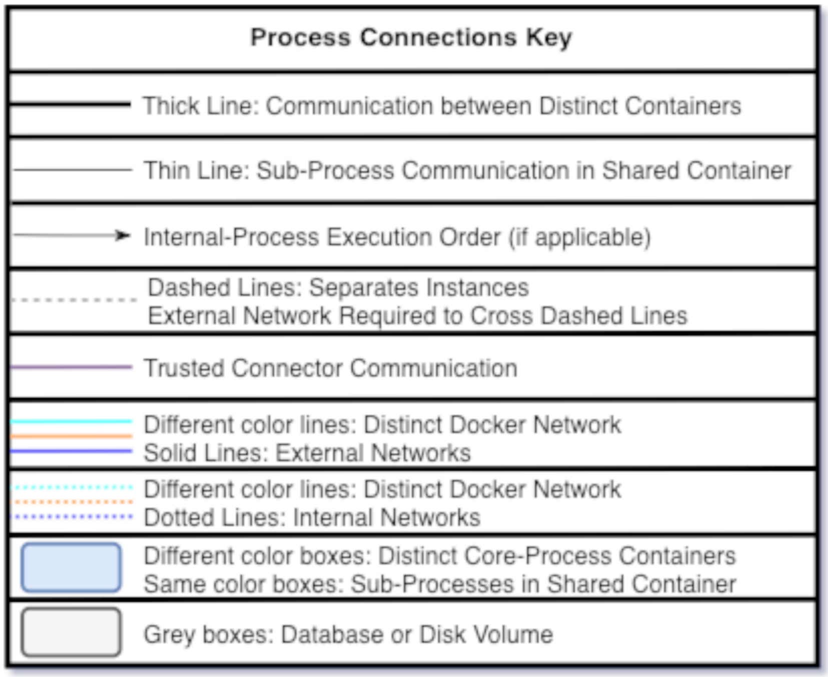

Process Connections Key
Thick Line: Communication between Distinct Containers
Thin Line: Sub-Process Communication in Shared Container
Internal-Process Execution Order (if applicable)
Dashed Lines: Separates Instances
External Network Required to Cross Dashed Lines
Trusted Connector Communication
Different color lines: Distinct Docker Network
Solid Lines: External Networks
Different color lines: Distinct Docker Network
Dotted Lines: Internal Networks
Different color boxes: Distinct Core-Process Containers
Same color boxes: Sub-Processes in Shared Container
Grey boxes: Database or Disk Volume

Figure 3.5. Key for *Complete Process* and Process Architecture Diagrams

diagrams will come toward the end of each section when discussing an architectures docker networks. The sub-process and core-process figures are captioned *Complete Process Structure* and *Container and Network Structure* respectively.

The complete process architecture figures show the entire architecture including the sub-processes in each core-process, data transfer routes between sub-processes, and distinct compute instances. Arrows instead of lines connecting sub-process blocks, in these diagrams, denote their order. These diagrams are comprehensive structural portrayals of all architectural components.

The container and network structure diagrams will depict all containerized services, the docker networks used to link the containers, connections between services, and all compute instances with their hosted processes. A key for the contained docker networks is included in the top right corner of each of the core-process figures. This key maps line styles and colors between processes in the diagram to specific docker networks. These diagrams are useful in comprehending the docker-compose shown at the base of each architectures dedicated section.

I include flowcharts for each process and the notable sub-processes. See figure 3.6 for the flowchart key. Each flowchart will be included once when the represented process is first included in an architecture. While certain processes share the same name between architectures and perform the same general task, their methodology changes significantly. For these processes, I will include the flowchart representative of that specific architecture when described.

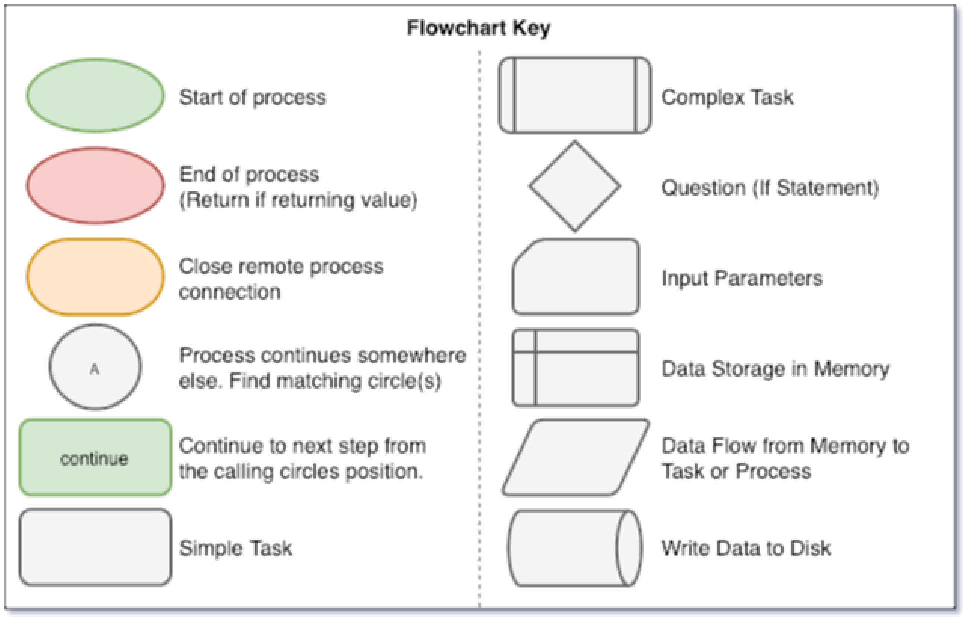

Flowchart Key
Start of process
End of process (Return if returning value)
Close remote process connection
A
Process continues somewhere else. Find matching circle(s)
continue
Continue to next step from the calling circles position.
Simple Task
Complex Task
Question (If Statement)
Input Parameters
Data Storage in Memory
Data Flow from Memory to Task or Process
Write Data to Disk

Figure 3.6. Key for Process Flowcharts

3.6.2 Tables

Each architecture will contain two tables which outline the data flow path. The tables are *Container Data Flow* and *Complete Data Flow* and they list the general flow of data and the type of data (Raw, Training, Results) at a numbered step. These tables are simplified meaning they do not show the exact paths the data will follow because they do not include loops or repetitive data requests. For example, during the model training phase of the architecture, training images and labels are requested thousands of time before before the results are obtained. These tables do not show this. Instead, training data is requested as one step and results are ready and a small number of steps following.

The data flow steps do not need to be complete to comprehend the flow. These tables provide the primary path data will take through all processes from the architectures start to its finish. The *Core-Process Data Flow* table shows how the data courses through complete process containers. The *Sub-Process Data Flow* table depicts the flow of data through all processes independent of their container.

Definitions of Data Flow Table Columns

- Data Type: The type of data flowing in that step. This changes as the data flows through the network.
- Origin: The Container process the data resides at the start of the step
- Destination: The Container process the data resides at the end of the step
- Sub Process (Complete Data Flow only): The data will reside in is this process which is a sub-process of the container listed directly to the left in the Destination column.
  - When the destination is "Disk" or "*X* File System" (X is the name of the container), this column contains the sub-process cause the virtual memory write. This is always the same sub-process from the step directly preceding the step containing the write.

The term "Disk" as the origin or destination denotes the data was stored in the instances file system through a shared volume. Alternatively, the term "*X* File System" where X is the name of a container denotes the data was stored in the containers file system.

3.6.3 Expectations for Network Architectures 2, 3 & 4

There are some expectations regarding the use of architectures 2, 3 and 4. The purpose of the proposed architecture 5 is to alleviate some of the security risks present in the following expectations. Architecture 1 addresses very few security risks as it is used as a baseline to compare the throughput and security levels of the other architectures. All security risks will be discussed more thoroughly in chapter 5 (discussion), but I will mention some here as they relate to the design of the architectures. The list below has general information pulled from the threat model in section 3.1 above.

The following list includes the expectations consistent through architectures 2, 3 and 4.

- The compute instance containing the training network architecture is shared by the Provider and Consumer. However, the Provider is the root user and is the only user who may start and stop processes and view active memory. The Consumer, on the other hand, is a non-root user who has read-only system access. The Consumer may traverse directories and view files on disk, but cannot make any changes.
- Memory is always vulnerable, so in no case should the Consumer have access to pulling data from memory for view.
- Consumers proprietary code is obfuscated.
- Training results should be reviewed by the Provider prior to the Consumer receiving them. Given the Consumers training app is obfuscated, this is the only way the Provider can guarantee the Consumer has not stolen their data.
- The Consumers application should run in a container restricted to internal network access.

3.7 External Instance Architecture

The process architectures discussed traverse over more then just a single compute instance. Connections are formed between some processes which extend to external instances as well. The two external instance layouts possible between the proposed training architectures are shown in figures 3.7 and 3.8. Figure 3.7 depicts the remote instance network for architectures 1, 2 and 3. These architectures only support a single consumer at a time. Architecture 4 and the theoretical architecture 5 both support n consumers simultaneously as depicted in figure 3.8. The only bound on the number of consumers is the compute power of the instance.

In architecture 1 through 3, the only remote instance connected to the Cloud Compute Instance is the Provider's Personal Instance, which is required to supply the network with the Provider's data. The assumption is that the Provider's Personal Instance has direct access to the FarmOS database containing the data. In our case, this database is an AWS S3 Bucket. The shared cloud compute instance contains the remainder of the architecture. All results of the training application's in these architectures are written directly to disk following the completion of the training network.

Architectures 4 and 5 support multiple consumers due to the addition of two processes coined External Data Handler's. These processes will be discussed in greater detail later, but in short, they export the results directly to the provider and consumer external instances rather then writing them to disk. In these architecture, certain processes may be shared between active Consumers while other processes run in parallel each dedicated to an individual Consumer. Once results are exported to a consumer, processes related specifically to that consumer terminate while the other services can remain running to be used by additional Consumers.

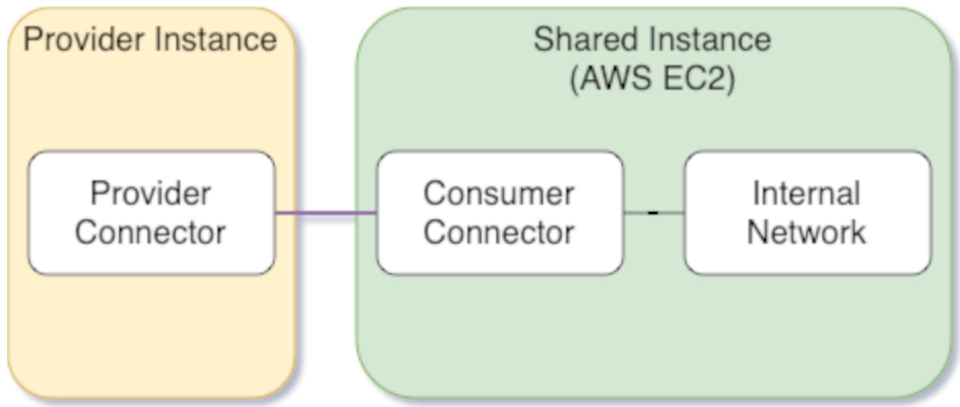


Figure 3.7. High Level Instance View - Architectures 1 through 3

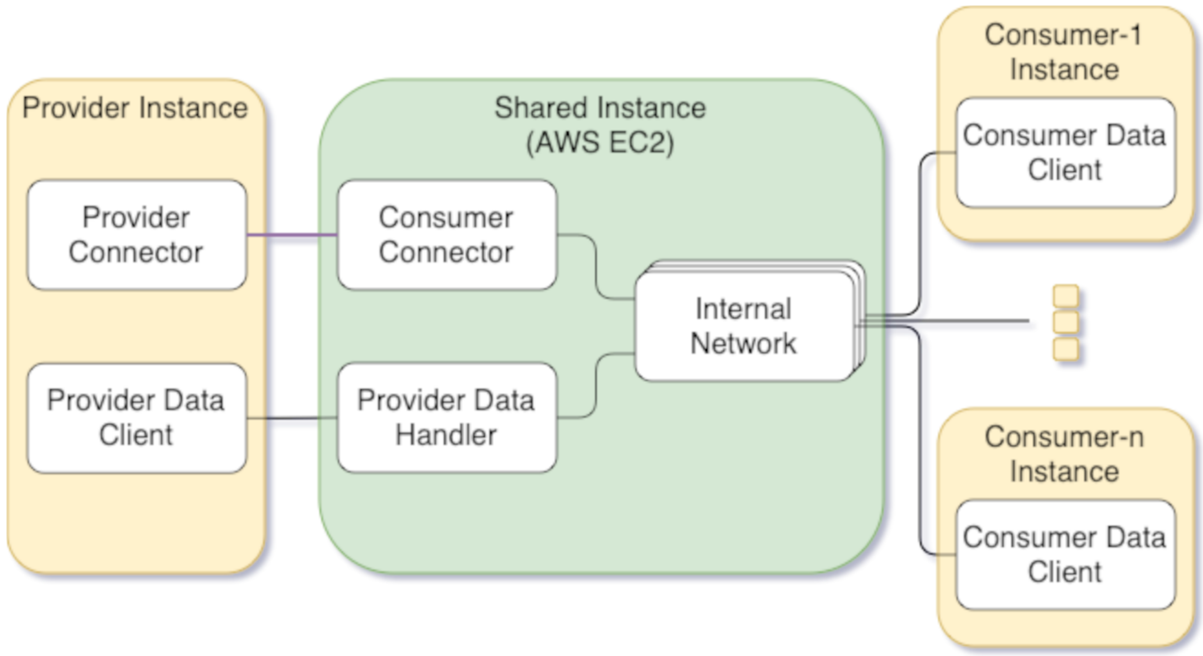


Figure 3.8. High Level Instance View - Architecture 4

3.8 Network Architecture 1: Baseline

This network architecture is used as a baseline for comparing the security and throughput of the remaining architectures. For this reason, this network is deployed with no security apart from the FarmOS Connector securing the Providers data while it is in transit from their personal instance to the EC2 compute instance containing the network. While it is trivial to add additional layers of security to meet certain objectives in the threat model, I opted to assume those layers did not exist here and instead provide architecture 2 which contains those traits.

Figure 3.9 shows this architectures complete process structure.

3.8.1 Core Processes

This architecture contains two core processes: FarmOS Connector and Batch Processing Application. The FarmOS Data Request is a sub-processes contained in the Batch Processing Application. In all further architectures, it is separated into a standalone process. Furthermore, I will discuss it in greater detail in the architecture 2 section when it is a core-process rather then now.

3.8.1.1 FarmOS Connector (FosC)

The same FarmOS Connector is used throughout all 4 architectures. It is a bi-directional connector using standard HTTP protocol for communication. The previously shown bi-directional connector diagram in figure 3.3 was expanded to provide the structure of the FosC. This is shown in figure 3.10. I did not make a flowchart for the FosC as it is very similar to figure 3.2, but with additional routing services added for consumer to provider requests that can flow in

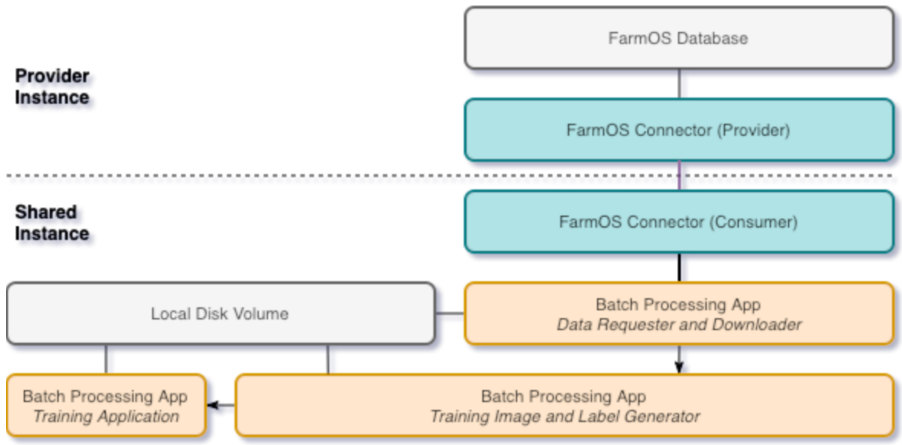


Figure 3.9. Network Architecture 1 - Complete Process Structure

both directions. The IDSCP2 data usage policies assigned to the FosC were "Local Logging" and "Remote Notification." These permit the usage of the data with the requirement that the consumer_core logs all connector related events locally and notifies the provider_core of these events in real-time.

Process Steps

1. provider_app pushes all the logs as a JSON object from the FarmOS database to the consumer_app over the connector.

2. consumer_app publishes the logs to an internal endpoint which can be requested from other internal applications.

    - The logs contain all the data and data references mentioned in section 3.5.3.
    - The requesting application in this network is the BPA, but in subsequent networks it is the FarmOS Data Requester.

3. consumer_app publishes a second endpoint internally on the shared instance for requesting files from the provider_app.

    - These files are requested using the ID's FarmOS assigns them.
    - The log object initially published contains these ID's.

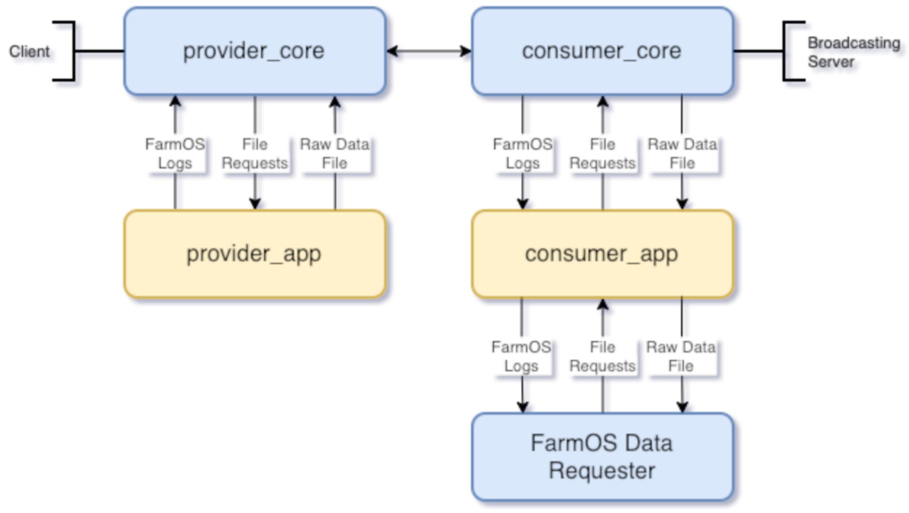


Figure 3.10. FarmOS Connector Architecture and General Request Flow

4. consumer_app's file endpoint receives an individual request for every raw data file. The following steps are followed for each request.

   (a) consumer_app forwards file requests back through the connector to the provider_app.

   (b) provider_app returns the requested file data with headers containing the name.

   (c) consumer_app forwards the file data back to the requesting internal process.

FarmOS Connector - Docker Compose The FarmOS Connector has two of it's own docker-compose files for the provider and consumer sides of the connector. It uses a separate compose from the main network because connectors some time to fully initialize. This is especially true in the case of a bi-directional connector like this one. Moreover, in architecture 4 which supports many consumers simultaneously, the connector must remain running while the compose files for other consumers application are started and stopped.

Provider Docker Compose This same docker-compose file is used by the provider in all architectures.

```
services:
  provider-core:
    image: farmstack/trusted-connector:1.0.0
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - ./logs/provider/:/root/log/
      - ./allow-all-flows.pl:/root/deploy/allow-all-flows.pl
      - ./settings2.mapdb:/root/etc/settings.mapdb
      - ./provider-keystore.p12:/root/etc/provider-keystore.p12
      - ./truststore.p12:/root/etc/truststore.p12
      - ./provider-routes.xml:/root/deploy/provider.xml
    extra_hosts:
      - consumer-core: 22.222.222.22 # consumer-core External IP
    ports:
      - 8989:8989  # Port for farmos-provider to push data
      - 29292:29292 # Port for consumer-core to send file requests
    networks:
      - ids-wide
      - farmos-provider-internal

  farmos-provider:
    image: farmstack/farmos-provider:1.0.0
    ports:
      - 5000:5000
    networks:
      - farmos-provider-internal
```

```

networks:
  ids-wide:
    driver: bridge
  farmos-provider-internal:
    driver: bridge
    internal: true
    name: farmos-provider-internal
```

Consumer Docker Compose - Architecture 1 There is a slight adjustment made in this docker-compose file for subsequent architectures. The change is noted by a comment in the code and it is removing the `network_mode: host` line from the `farmos-consumer` service so that only processes on the `farmos-consumer-internal` network can access the *farmos-consumer* app's log and file endpoints.

```
services:
  consumer-core:
    image: farmstack/trusted-connector:1.0.0
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - ./logs/consumer/:/root/log/
      - ./allow-all-flows.pl:/root/deploy/allow-all-flows.pl
      - ./settings2.mapdb:/root/etc/settings.mapdb
      - ./consumer-keystore.p12:/root/etc/consumer-keystore.p12
      - ./truststore.p12:/root/etc/truststore.p12
      - ./consumer-routes.xml:/root/deploy/consumer.xml
    extra_hosts:
      - provider-core: 11.111.111.11 # provider-core External IP
    ports:
      - 29290:29290
    networks:
      - ids-wide
      - farmos-consumer-internal

  farmos-consumer:
    image: farmstack/farmos-consumer:restapi
    network_mode: host # Only present in architecture 1
    ports:
      - 29291:29291
    networks:
      - farmos-consumer-internal

networks:
  ids-wide:
    driver: bridge
  farmos-consumer-internal:
    driver: bridge
```

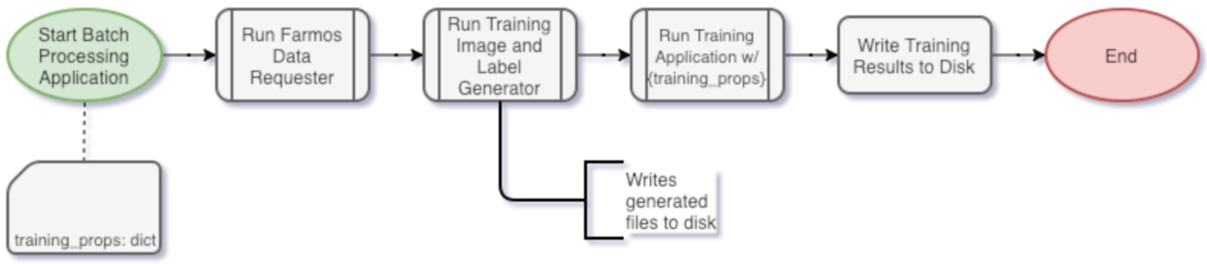

Start Batch Processing Application
Run Farmos Data Requester
Run Training Image and Label Generator
Run Training Application w/ {training_props}
Write Training Results to Disk
End
training_props: dict
Writes generated files to disk

Figure 3.11. Network Architecture 1 - Batch Processing Application

```
    name: farmos-consumer-internal
    internal: true
```

3.8.1.2 Batch Processing Application (BPA)

The BPA in this architecture contains the sub processes for the entire training network apart from the Connector. It contains the FarmOS Data Requester and Downloader (Requester), Training Image and Label Generator (Generator), and Training Application (Trainer) as sub processes. See figure 3.11 for detailed process steps. The Generator and Trainer do not have accompanied flowcharts because they are out of scope for this discussion and they were not my work but the work of a colleague's [29]. The Requester, however, will be addressed further in section 3.9 on architecture 2 when it is a standalone process. See figure 3.14 for the Requester flowchart.

Process Steps

1. Pull data (logs) from FarmOS connector.
2. Scrape all needed data from logs.
3. Request all raw data files required for training using their file ID's scraped from logs.
4. Write data files directly to disk for later access by Batch Processing Application.
5. Generate training images and labels from raw data stored on disk.
6. Write training labels and images to disk.
7. Run training application on training labels and images.
8. Training results are temporally stored in memory until the Trainer is complete.

9. When the Trainer sub-process completes, all results are written to disk.

3.8.2 Data Flow

The data flow for this architecture is the shortest because it contains the fewest processes. These tables will continue to grow as more processes are Incorporated in the following architectures.

Tables 3.3 and 3.4 show the core-process and sub-process data flow respectively. See section 3.6.2 for a detailed description on the structure of each table.

Table 3.3. Architecture 1 - Container Data Flow

Step	Data Type	Origin	Destination
1	Raw	FosC-Provider	FosC-Consumer
2	Raw	FosC-Consumer	BPA
3	Results	BPA	Disk

Table 3.4. Architecture 1 - Complete Data Flow

Step	Data Type	Origin	Destination	Sub Process
1	Raw	FosC-Provider	FosC-Consumer	FosC-Consumer
2	Raw	FosC-Consumer	BPA	Data Requester
3	Raw	BPA	BPA	Data Downloader
4	Raw	BPA	Disk	Data Downloader
5	Raw	Disk	BPA	Generator
6	Training	BPA	Disk	Generator
7	Training	Disk	BPA	Trainer
8	Results	BPA	Disk	Trainer

3.8.3 Docker Networks

See figure 3.12 for this architectures network diagram. While the FosC has its own internal networks, the remainder of this architecture uses no additional custom docker networks. Rather, this architecture uses a the default host network deployed by the Dockerd. This network links the BPA container directly to the host machines network. Hence why this diagram contains a dashed box surrounding around all processes rather then linking them directly as will be shown in the remaining architectures. Putting containers on the host network is a security risk because it means the same network used by the host machine is mirrored to all attached containers. This means all ports forwarded in the containers will be forwarded on the host network and they have direct external network access through the host. This security problem will be addressed in architecture 2.

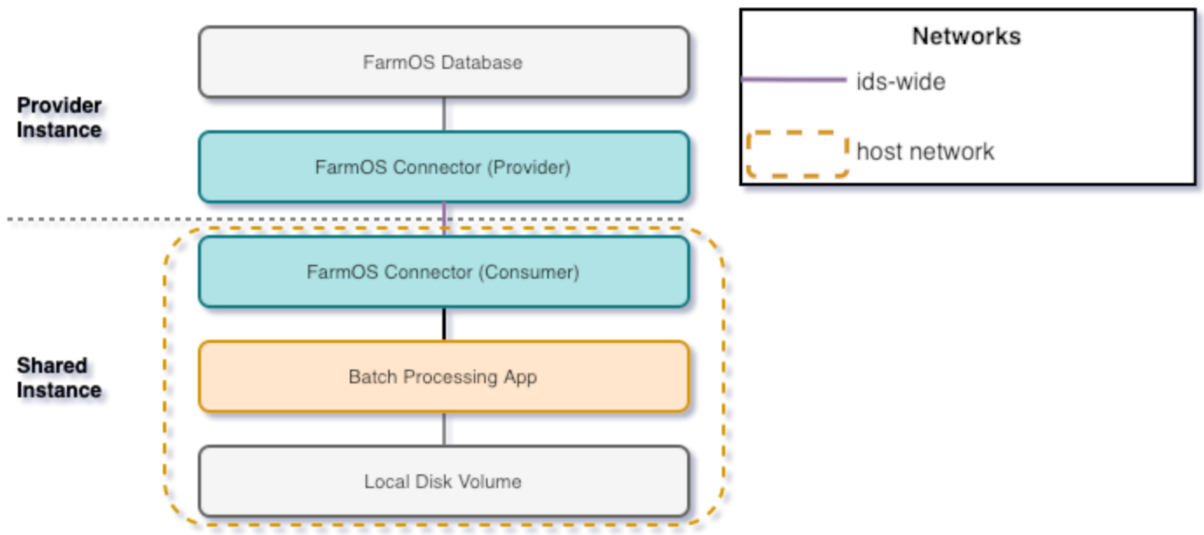


Figure 3.12. Network Architecture 1 - Container and Network Structure

3.8.4 Docker Compose

Below is the docker-compose file for this network architecture. It is the shortest and simplest compose file of all the architectures because it only uses a single core process and does not use any custom networks outside of the FosC.

```
services:
  batch_processing_app:
    image: consumer:architecture1_bpa
    env_file:
      - .env
    volumes:
      - ./dlvenv/dlvenv:/dlvenv
      - ./data:/app/data
      - ./results:/app/results
    network_mode: host
```

The `dlvenv` volume contains the environment files. They are stored in a volume on disk rather then in the Docker Container because geospatial machine learning packages are very large taking up over 9GB of space.

The `data` volume is where the Requester stores all raw data after pulling it from the FosC. The training labels and images from the Generator are also stored in the `data` volume.

The `results` volume is used to store the training results.

3.9 Network Architecture 2: Utilizing Docker

In Network Architecture 2, I expand the overall security by dividing the processes from Architecture 1 into separate standalone processes and applying some of Docker's networking features. This architecture provides both improved security for the provider's data and security for the consumer's training application through obfuscation. When code is obfuscated, it is made illegible and exceedingly difficult to reverse engineer while still maintaining all process integrity. In other words, it does the same thing(s) it was originally written to do, but the code itself cannot be read. The consumers training application is assumed to be obfuscated in all remaining network architectures.

The consumers code can no longer be analyzed by the provider for security risks because it is obfuscated. However, the BPA, which contains the Trainer, is set to strictly internal network access, so it cannot export any of the providers data. Moreover, the training results are written to disk for the provider to review prior to giving them to the consumer. This is so the provider can ensure none of their data was copied and marked as training results.

Figure 3.13 shows this architectures complete process structure.

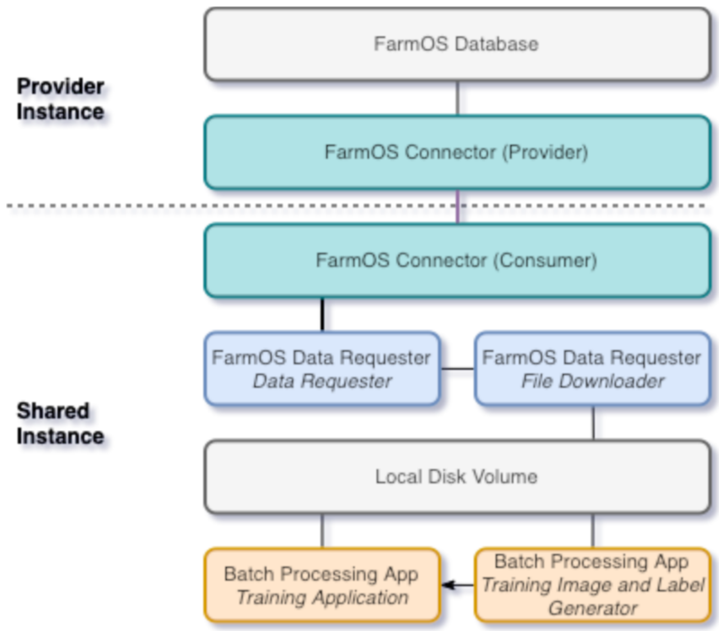


Figure 3.13. Network Architecture 2 - Complete Process Structure

3.9.1 Core Processes

Three processes make up this architecture. The FosC and BPA, as in architecture 1, and the FarmOS Data Requester (Requester). The Requester has been divided from the BPA in architecture 1.

3.9.1.1 FarmOS Data Requester

This Requester is essentially identical to the BPA sub-process Requester from architecture 1 apart from an additional step added at its conclusion to signal the BPA that all data has been pulled. When the Requester process completes it creates a file as a signal to the BPA that it has finished downloading all raw data files from the FosC. See figure 3.14 for the logical flowchart. The top section in the flowchart figure is the Requester objects most general steps. The middle and bottom sections show the breakdown of the format_logs and get_and_store_data methods respectively.

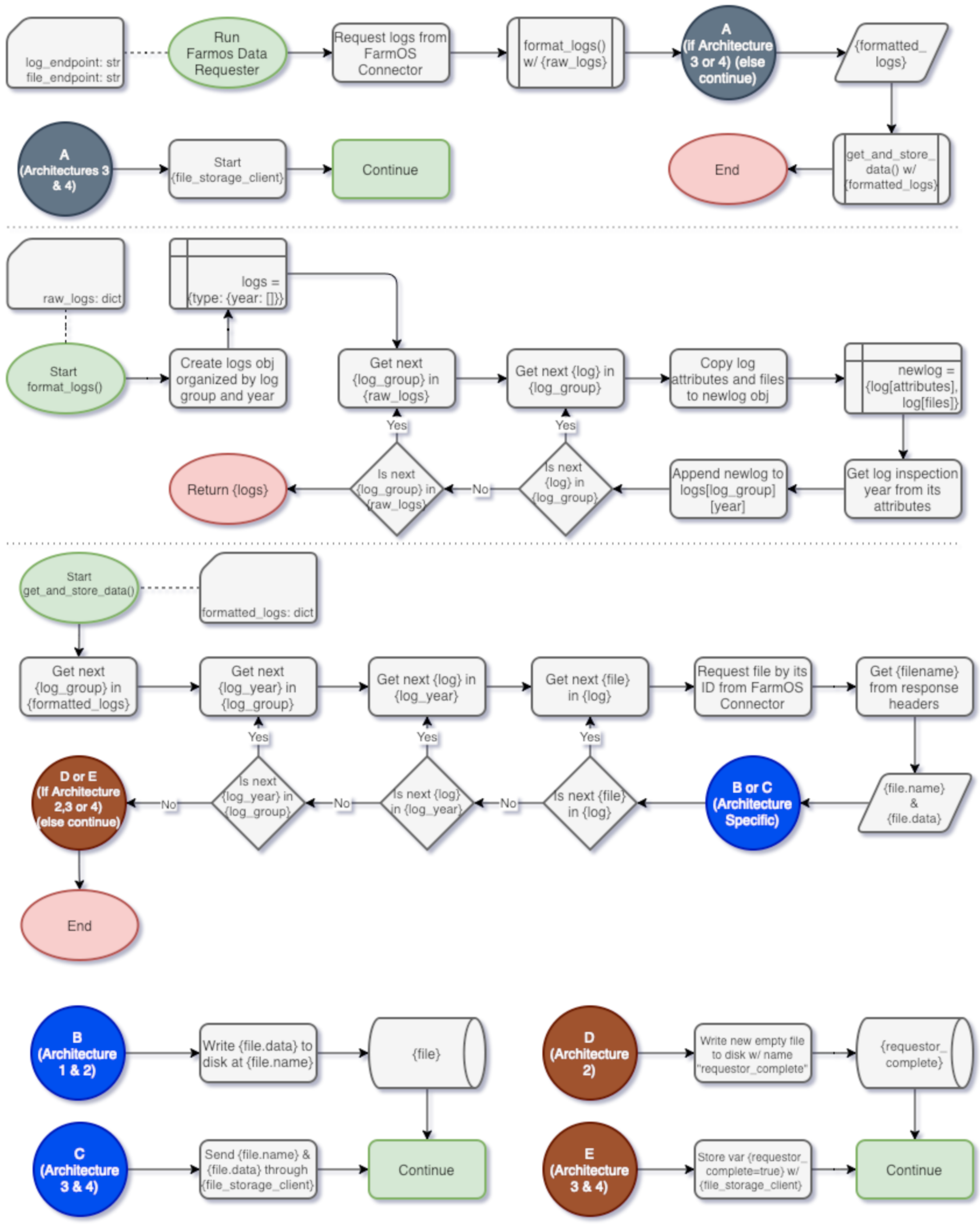
Process Steps

1. Pull raw data logs from FarmOS connector.
2. Scrape geospatial location data and associated data file id's from each log and store in a new log object.
3. Request each raw data file required for training by their scraped file ID's.
4. Write each requested data file directly to disk for later access by BPA.
5. Create new empty file on disk at root of data file storage directory with name "requester_complete." The BPA is watching for this file and when created it signifies to BPA to start.

3.9.1.2 Batch Processing Application

The logic for this architectures BPA has a few key differences from the architecture 1 BPA. The most evident of these changes are the separation of the Requester and the addition of a new first step that checks for the "requester_complete" file.

The BPA and Requester processes are started at the same time, but the BPA must not start generating training images and labels until all raw data is on the shared instance disk. Hence, the reason it must wait for the file to be created as a signal to begin. Other methods of signaling



log_endpoint: str
file_endpoint: str
Run Farmos Data Requester
Request logs from FarmOS Connector
format_logs() w/ {raw_logs}
A (if Architecture 3 or 4) (else continue)
{formatted_ logs}
get_and_store_ data() w/ {formatted_logs}
End
A (Architectures 3 & 4)
Start {file_storage_client}
Continue
raw_logs: dict
logs = {type: {year: []}}
Start format_logs()
Create logs obj organized by log group and year
Get next {log_group} in {raw_logs}
Get next {log} in {log_group}
Copy log attributes and files to newlog obj
newlog = {log[attributes], log[files]}
Get log inspection year from its attributes
Append newlog to logs[log_group] [year]
Is next {log} in {log_group}
Is next {log_group} in {raw_logs}
Yes
Yes
No
Return {logs}
Start get_and_store_data()
formatted_logs: dict
Get next {log_group} in {formatted_logs}
Get next {log_year} in {log_group}
Get next {log} in {log_year}
Get next {file} in {log}
Request file by its ID from FarmOS Connector
Get {filename} from response headers
{file.name} & {file.data}
B or C (Architecture Specific)
Is next {file} in {log}
Is next {log} in {log_year}
Is next {log_year} in {log_group}
Yes
Yes
Yes
No
No
No
D or E (If Architecture 2,3 or 4) (else continue)
End
B (Architecture 1 & 2)
Write {file.data} to disk at {file.name}
{file}
C (Architecture 3 & 4)
Send {file.name} & {file.data} through {file_storage_client}
Continue
D (Architecture 2)
Write new empty file to disk w/ name "requestor_complete"
{requestor_ complete}
E (Architecture 3 & 4)
Store var {requestor_ complete=true} w/ {file_storage_client}
Continue

Figure 3.14. FarmOS Data Requester

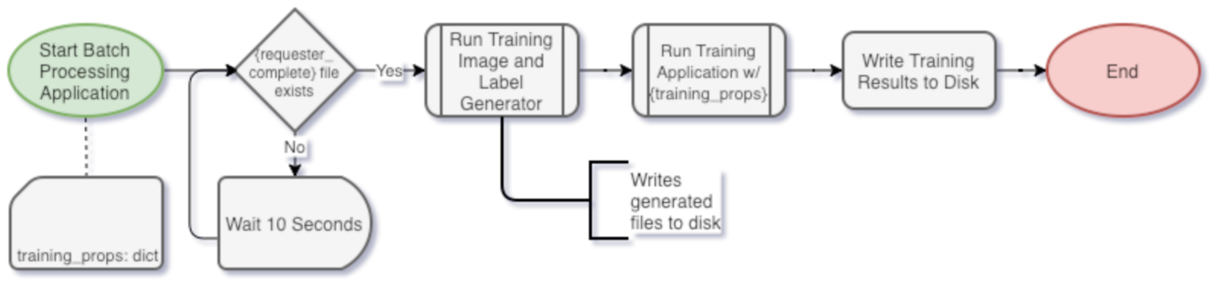

Start Batch Processing Application
training_props: dict
{requester_complete} file exists
Yes
No
Wait 10 Seconds
Run Training Image and Label Generator
Writes generated files to disk
Run Training Application w/ {training_props}
Write Training Results to Disk
End

Figure 3.15. Network Architecture 2 - Batch Processing Application

could have been used such as socket communication (as in later architectures). I opted to use the simple file creation and watching method because it produces no additional security risks and is trivial to incorporate into the existing processes.

The final change is the aforementioned obfuscation of the BPA. The BPA is considered to be the Consumers proprietary code. The Provider is the root user on the shared instance meaning without obfuscation the Provider could appropriate the Consumers code. Obscuring the code from the Provider clearly opens new security risks to the Providers data. While the BPA's code is proprietary, its results are not. This will be discussed further in chapter 5 (discussion).

Process Steps

1. Continuously check for the "requester_complete" file at the root of raw data storage directory on disk.
2. When file is seen in directory, start the Generator.
3. Generate training images and labels from raw data stored on disk.
4. Write training labels and images to disk.
5. Run training application on training labels and images.
6. Training results are temporally stored in memory until the Trainer is complete.
7. When the Trainer sub-process completes, all results are written to disk.

3.9.2 Data Flow

The complete data flow is very similar to the architecture 1 complete data flow which is expected as the routing between all sub processes is the same for the data itself. The container flow has increased due to the separation of the Requester into a new container.

Tables 3.5 and 3.6 show the core-process and sub-process data flow respectively. See section 3.6.2 for a detailed description on the structure of each table.

Table 3.5. Architecture 2 - Container Data Flow

Step	Data Type	Origin	Destination
1	Raw	FosC-Provider	FosC-Consumer
2	Raw	FosC-Consumer	Requester
3	Raw	Requester	Disk
4	Raw	Disk	BPA
5	Results	BPA	Disk

Table 3.6. Architecture 2 - Complete Data Flow

Step	Data Type	Origin	Destination	Sub Process
1	Raw	FosC-Provider	FosC-Consumer	FosC-Consumer
2	Raw	FosC-Consumer	Requester	Data Requester
3	Raw	Requester	Requester	File Downloader
4	Raw	Requester	Disk	File Downloader
5	Raw	Disk	BPA	Generator
6	Training	BPA	Disk	Generator
7	Training	Disk	BPA	Training Application
8	Results	BPA	Disk	Training Application

3.9.3 Docker Networks

See figure 3.16 for this architectures network diagram. Rather than connecting the containers directly to the host machines network, this architecture utilizes the custom docker network `farmos-consumer-internal`. It is used to link the FosC-consumer and Requester processes allowing communication between the two. The BPA is not given network access in this architecture because it does not require it. This is significantly more secure then the previous architecture as it completely removes the BPA's ability to export any data to a remote machine.

3.9.4 Docker Compose

This network architecture has a more complex docker-compose file compared to architecture 1. This is a result of multiple services and docker utilities. The volumes are the same as the volumes used in architecture 1. The BPA uses the same volumes as architecture 1, but the `dlvenv` volume is set to read-only (:ro). The Requester only requires the `data` volume because it doesn't need the geospatial packages in the `dlvenv` volume or the `results` volume for the training results.

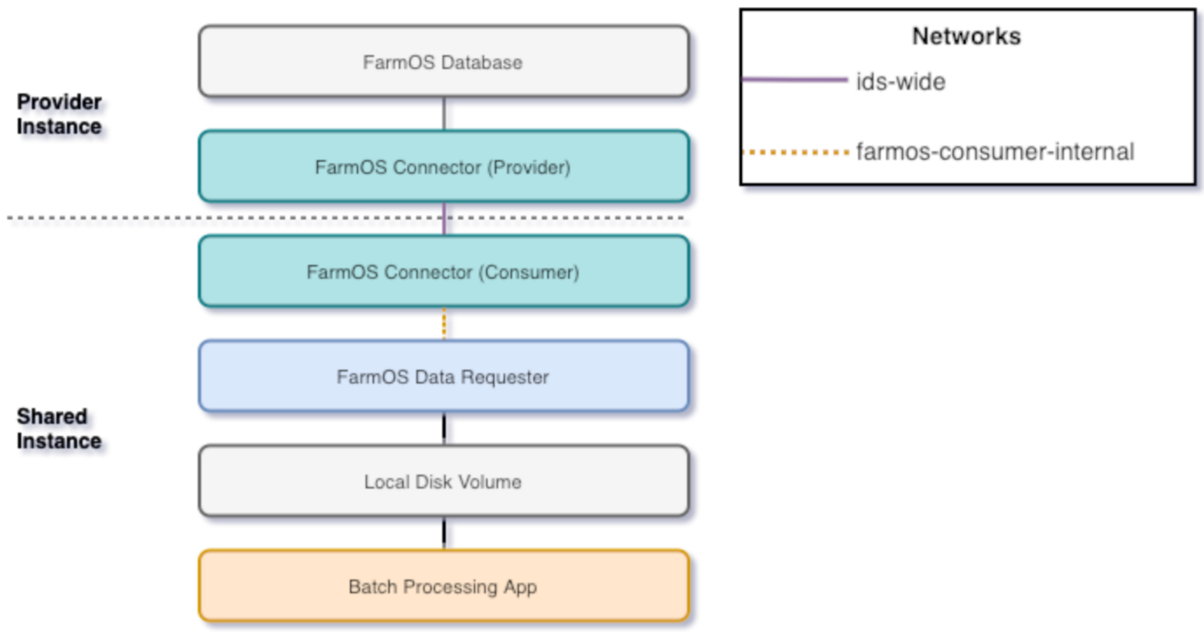


Figure 3.16. Network Architecture 2 - Container and Network Structure

Each of the containers file-systems are also set to read-only in this architecture. All services in remaining architectures will be set to read-only. Setting a container to read-only hardens its security. If a malicious source were to breach the container they would be unable to alter anything in the containers file system.

Note that `external: true` under the `farmos-consumer-internal` network means that it is already deployed on the Dockerd, not that it has internet access.

```
services:
  requester:
    image: consumer:architecture2_data_requester
    env_file:
      - requester.env
    networks:
      - farmos-consumer-internal
    volumes:
      - ./data:/app/data
    read_only: true;

  batch_processing_app:
    depends_on:
      - requester
    image: consumer:architecture2_bpa
    env_file:
      - bpa.env
    volumes:
```

```yaml
      - ./dlvenv/dlvenv:/dlvenv:ro
      - ./data:/app/data
      - ./results:/app/results
    read_only: true;

networks:
  farmos-consumer-internal:
    external: true
```

3.10 Network Architecture 3: Cryptographic File Storage Server

Architecture 3 contains substantial changes from the base architecture. These changes provide improvements to security for the provider. Consequently, these changes also reduce the networks throughput compared to the first two architectures. In this architecture, data is no longer written to disk unencrypted during training. This includes the Provider's raw data, training images and labels, and results.

Any data that must be written to disk is a member of one of two groups: internal data or external data. Internal data will never leave the machine and should persist only for the duration of the training network. The internal data group includes raw data and generated training data. External data will eventually be exported from the machine following training. The training results are external data.

Internal data can be saved and pulled an unlimited number of times during training. It is the utility data that is passed between processes and is required to obtain the final trained model and results. Internal data does not include any results. External data can be stored, but cannot be retrieved until after the model is trained. All managed data is removed from disk at the networks completion except the encrypted external data which is compressed and encrypted.

Figure 3.17 shows this architectures complete process structure.

3.10.1 Core Processes

Four total core processes constitute this network architecture. It contains the FosC, BPA and Requester as in architecture 2. Additionally, it contains the Internal Data Handler (IDH).

Two methods used often in flowcharts throughout the remaining architectures are the `Send` and `Recv` methods. They portray the functionality for sending and receiving data respectively. I present these now in figure 3.18. They will not be discussed hence forth.

3.10.1.1 Internal Data Handler

The IDH works as a subsidiary to processes in the network for disk reads and writes. Moreover, the IDH encrypts all data before writing to disk, so no active data is ever viewable on disk during the duration of the training network.

The IDH is composed of two layers: the server and the processor. The server is responsible for accepting connections, processing requests and routing data to/from the processor. The processor must encrypt incoming data and write it to disk, read requested data from disk and

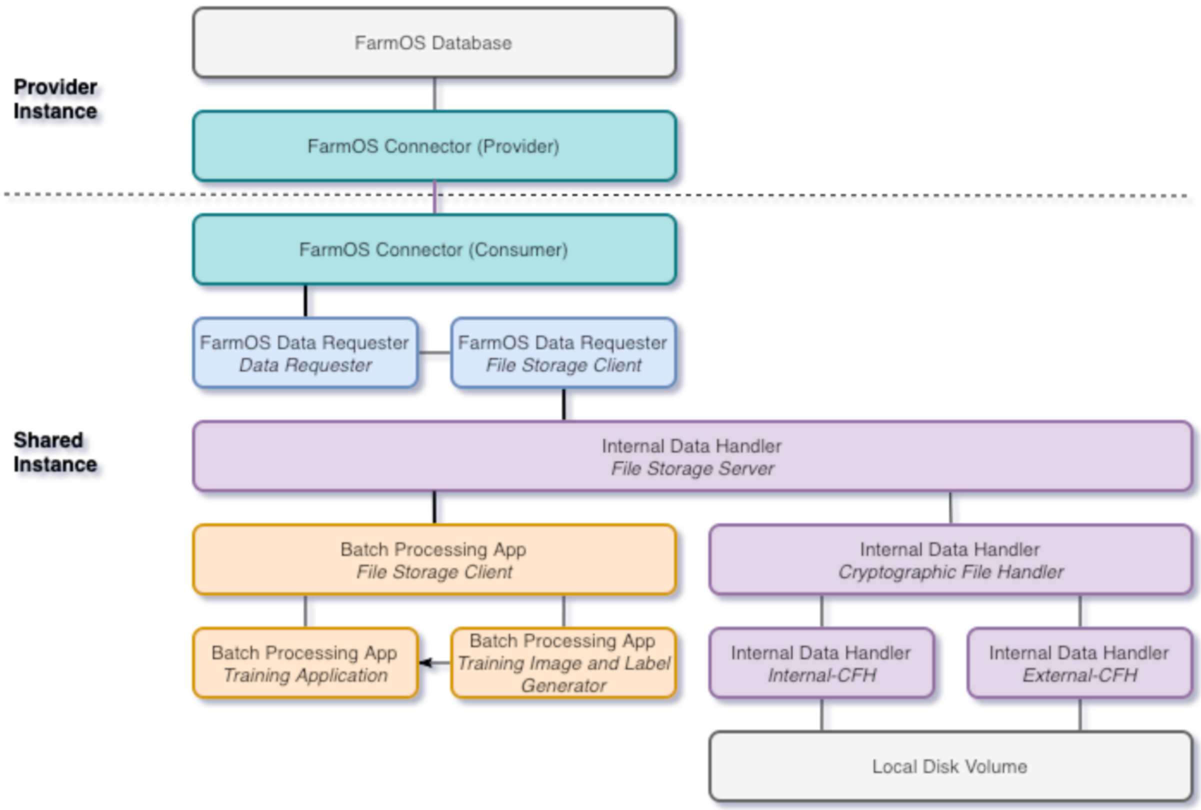

FarmOS Database
Provider Instance
FarmOS Connector (Provider)
FarmOS Connector (Consumer)
FarmOS Data Requester *Data Requester*
FarmOS Data Requester *File Storage Client*
Shared Instance
Internal Data Handler *File Storage Server*
Batch Processing App *File Storage Client*
Internal Data Handler *Cryptographic File Handler*
Batch Processing App *Training Application*
Batch Processing App *Training Image and Label Generator*
Internal Data Handler *Internal-CFH*
Internal Data Handler *External-CFH*
Local Disk Volume

Figure 3.17. Network Architecture 3 - Complete Process Structure

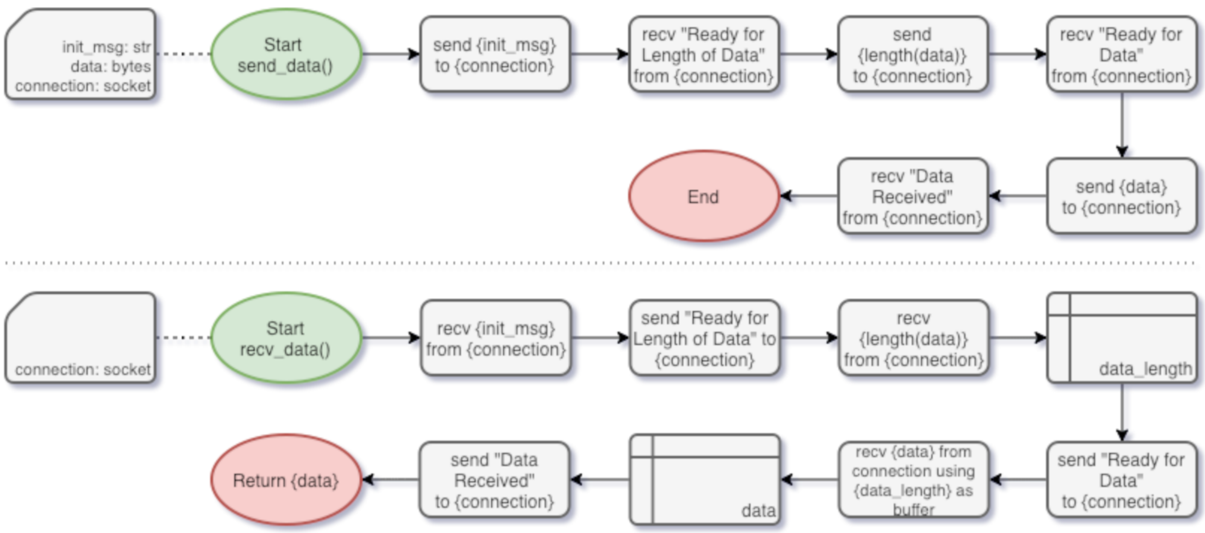

init_msg: str
data: bytes
connection: socket
Start send_data()
send {init_msg} to {connection}
recv "Ready for Length of Data" from {connection}
send {length(data)} to {connection}
recv "Ready for Data" from {connection}
send {data} to {connection}
recv "Data Received" from {connection}
End
connection: socket
Start recv_data()
recv {init_msg} from {connection}
send "Ready for Length of Data" to {connection}
recv {length(data)} from {connection}
data_length
send "Ready for Data" to {connection}
recv {data} from connection using {data_length} as buffer
data
send "Data Received" to {connection}
Return {data}

Figure 3.18. Send and Receive Data Flowcharts

decrypt it, maintain a record of all stored data and preserve an AES encryption key.

The File Storage Server (FSS) is the server layer and two Cryptographic File Handler's (CFH) form the processor layer. The FSS and CFH are not core-processes, but layers in the IDH.

3.10.1.2 Cryptographic File Handler

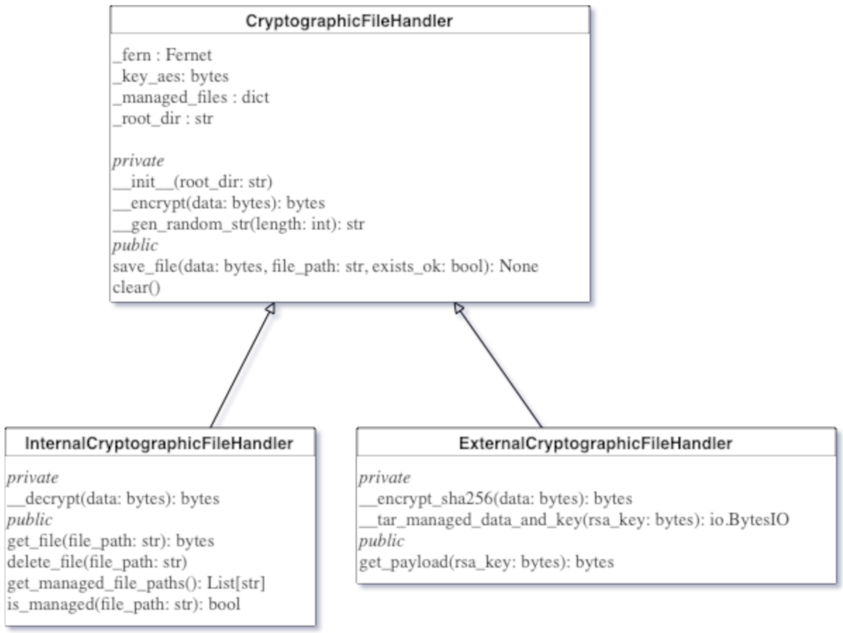
The internal-CFH (iCFH) and external-CFH (eCFH) collectively assume the processor layer of the IDH. Each CFH is necessary to meet the individual requirements of the internal and external data groups respectively. The two CFH's contain some of the same requirements so they implement the same base. The relationship's between the CFH objects can be seen in their UML class diagram in figure 3.19. The set of methods for each the CFH objects mimic the required steps for a very basic file store with additional cryptography tools. RSA and AES cryptographic utility functions are appended to the set of common file store functions. The cryptography functions are used during the save and get file methods. Encryption for save and decryption for get.

Methods that make up a file store include, writing, reading and deleting files, obtaining a list of contents by name, clearing all memory, etc... The iCFH holds methods for all of these methods and they work as they are named with encryption steps inserted in `save_file`, `get_file` and `get_payload`. The eCFH, however, has only a single `get_payload` function.

Both the iCFH and eCFH utilize AES encryption. 128-bit AES encryption in cipher block chaining (CBC) mode is used to encrypt all the stored data. AES was chosen because it demonstrated the best encryption throughput at 12.9 KB/ms which was better significantly better than RSA and DES techniques [30]. While the decryption throughput for RSA is better, the time saved during encryption with AES outweighed RSA. AES is the only encryption used for internal data. In architecture 3, external data is also only encrypted with AES. Architecture 4 further applies 256-bit RSA encryption to the external AES key.

See figure 3.20 for the CFH base class flowchart. It contains functional steps for the constructor, `save_file` and clear methods. The constructor saves the storage path for the encrypted files, generates a 128-bit AES key, and deploys a fernet object to use for AES encryption and decryption.

save_file takes the files name and data as two parameters. The file name may be a complete file path or any file-system parse-able string. First, it generates a random string to use as the



CryptographicFileHandler
_fern : Fernet
_key_aes: bytes
_managed_files : dict
_root_dir : str
private
__init__(root_dir: str)
__encrypt(data: bytes): bytes
__gen_random_str(length: int): str
public
save_file(data: bytes, file_path: str, exists_ok: bool): None
clear()
InternalCryptographicFileHandler
private
__decrypt(data: bytes): bytes
public
get_file(file_path: str): bytes
delete_file(file_path: str)
get_managed_file_paths(): List[str]
is_managed(file_path: str): bool
ExternalCryptographicFileHandler
private
__encrypt_sha256(data: bytes): bytes
__tar_managed_data_and_key(rsa_key: bytes): io.BytesIO
public
get_payload(rsa_key: bytes): bytes

Figure 3.19. Cryptographic File Handler - Class Diagram

file's name when stored on disk. It AES encrypts the file data and writes the encrypted data to the saved storage path with the random string as its name. The files true name is used as a key and the random name a value in the managed_files object.

clear deletes all managed data from disk and removes all keys from managed_files. It can be called anytime, but is called in a hook when the process is ended. If the process terminates before *clear* is able to run the encrypted file data will remain on disk. However, the AES key used to encrypt the data will be gone rendering the files inaccessible.

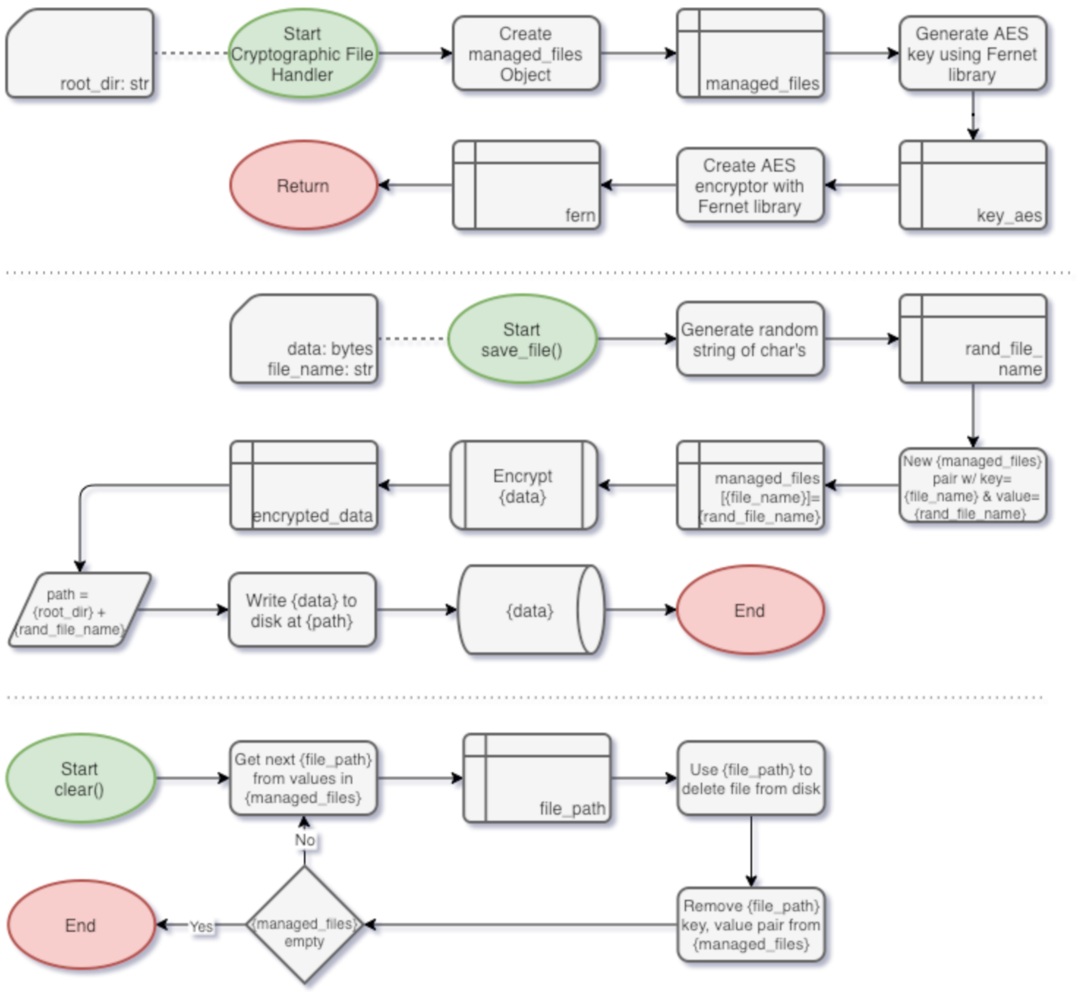
The iCFH's methods reflect stored individual file accessibility. This resembles a true data store. Contrarily, the eCFH is a vault allowing storing of files and collecting all at once, but no individual file access.

The iCFH contains the `delete_file`, `is_managed`, `get_file` and `get_managed_file_paths` functions. The iCFH's AES encryption key lives only in memory and, therefor, is lost at its termination. The flowchart for the iCFH can be seen in figure 3.21.

delete_file deletes the passed file from disk and removes its key from the managed_files object.

get_managed_file_paths returns all the key's in the managed_files object (true file names).

The eCFH is for handling external data. External data will only be pulled once at the end



root_dir: str
Start Cryptographic File Handler
Create managed_files Object
managed_files
Generate AES key using Fernet library
key_aes
Create AES encryptor with Fernet library
fern
Return
data: bytes
file_name: str
Start save_file()
Generate random string of char's
rand_file_name
New {managed_files} pair w/ key={file_name} & value={rand_file_name}
managed_files[{file_name}]={rand_file_name}
Encrypt {data}
encrypted_data
path = {root_dir} + {rand_file_name}
Write {data} to disk at {path}
{data}
End
Start clear()
Get next {file_path} from values in {managed_files}
file_path
Use {file_path} to delete file from disk
Remove {file_path} key, value pair from {managed_files}
{managed_files} empty
No
Yes
End

Figure 3.20. Cryptographic File Handler - Base Class

of the network when the model is trained. All the encrypted data managed by the eCFH is combined into a single data archive (tar) aliased *payload*. This payload also includes the AES key. In architecture 3, the AES key is attached to the payload unencrypted, so the Provider can decrypt the results and review them before giving them to the Consumer. After all the external data is in the payload, the payload is compressed and returned. Those steps form the ***get_payload*** function shown in figure 3.22.

3.10.1.3 File Storage Server

The FSS is a multi-threaded socket server. The main thread accepts connections and creates sub threads to process them. The sub threads receive the connection's request then extract a *client* and *method*. The *client* is representative of the internal and external data groups. Therefor, the value of *client* in the request must either be *internal* or *external*. The FSS uses the *client*

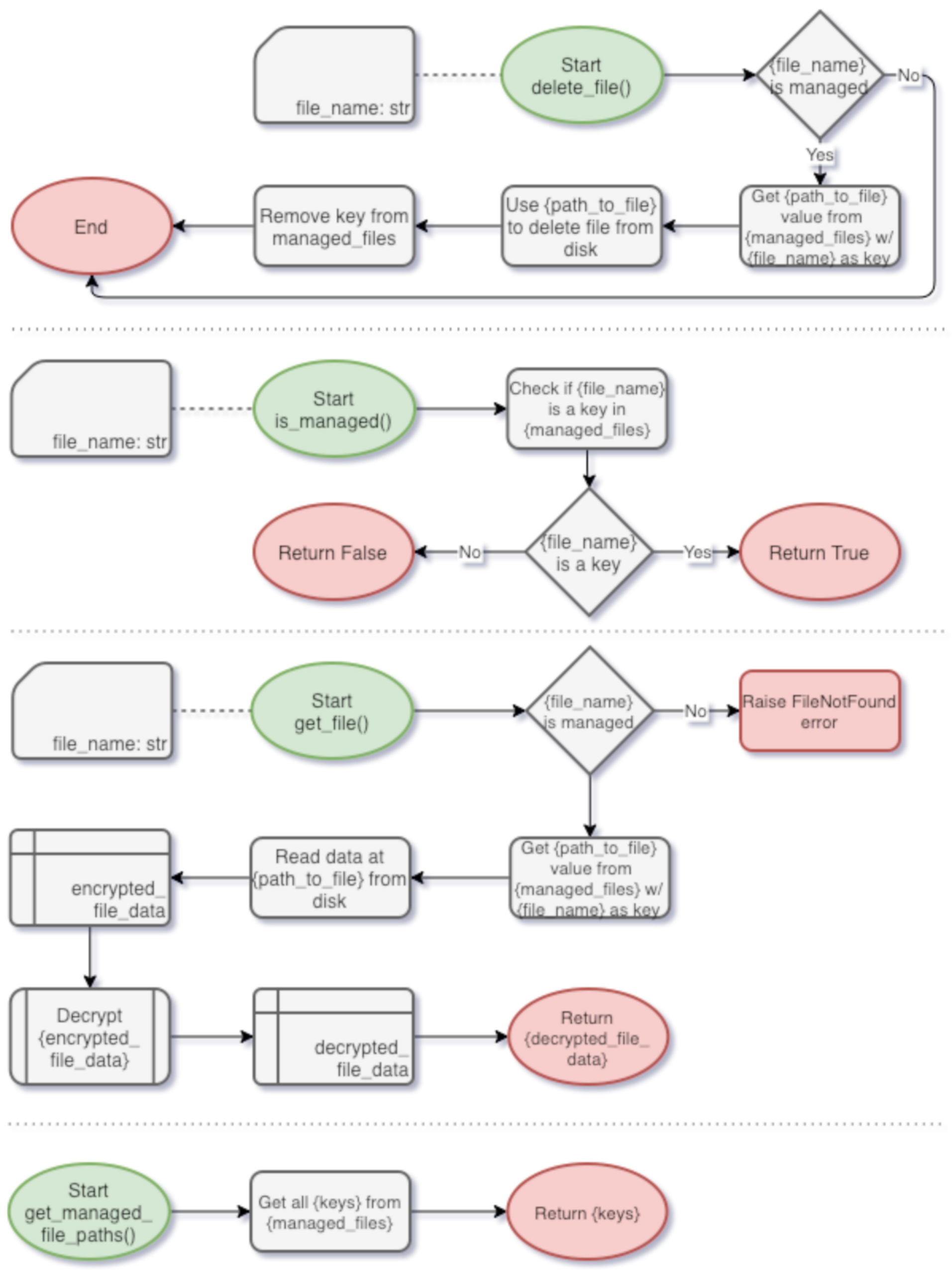
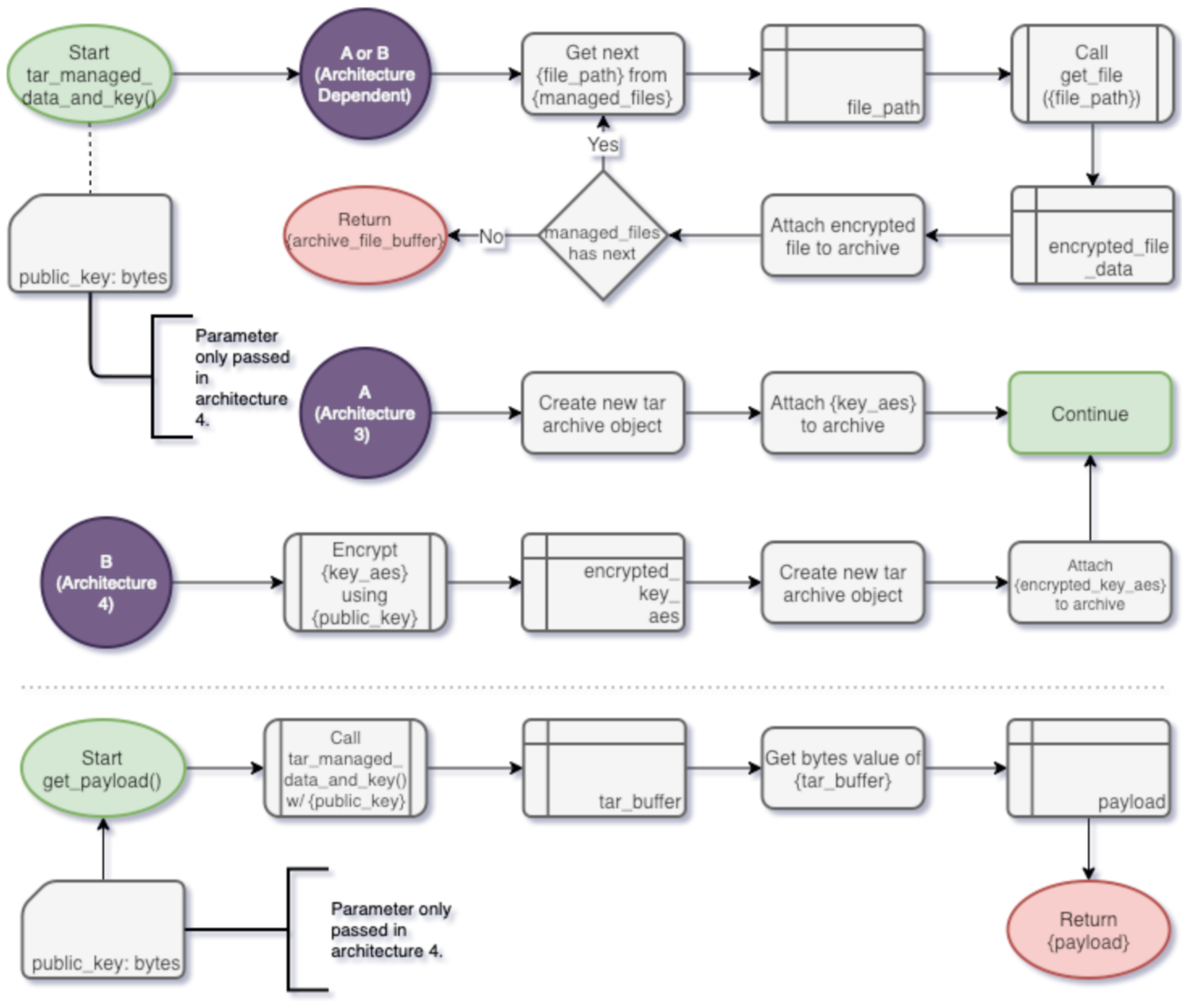

file_name: str
Start delete_file()
{file_name} is managed
No
Yes
Get {path_to_file} value from {managed_files} w/ {file_name} as key
Use {path_to_file} to delete file from disk
Remove key from managed_files
End
file_name: str
Start is_managed()
Check if {file_name} is a key in {managed_files}
{file_name} is a key
No
Yes
Return False
Return True
file_name: str
Start get_file()
{file_name} is managed
No
Raise FileNotFound error
Get {path_to_file} value from {managed_files} w/ {file_name} as key
Read data at {path_to_file} from disk
encrypted_ file_data
Decrypt {encrypted_ file_data}
decrypted_ file_data
Return {decrypted_file_ data}
Start get_managed_ file_paths()
Get all {keys} from {managed_files}
Return {keys}

Figure 3.21. Cryptographic File Handler - Internal Data Sub Class



Start tar_managed_data_and_key()
A or B (Architecture Dependent)
Get next {file_path} from {managed_files}
file_path
Call get_file ({file_path})
public_key: bytes
Return {archive_file_buffer}
No
managed_files has next
Yes
Attach encrypted file to archive
encrypted_file_data
Parameter only passed in architecture 4.
A (Architecture 3)
Create new tar archive object
Attach {key_aes} to archive
Continue
B (Architecture 4)
Encrypt {key_aes} using {public_key}
encrypted_key_aes
Create new tar archive object
Attach {encrypted_key_aes} to archive
Start get_payload()
Call tar_managed_data_and_key() w/ {public_key}
tar_buffer
Get bytes value of {tar_buffer}
payload
public_key: bytes
Parameter only passed in architecture 4.
Return {payload}

Figure 3.22. Cryptographic File Handler - External Data Sub Class

to route the data to either the iCFH or eCFH. The *method* must be one of: *GetFile*, *PutFile*, *PutVar*, *GetVar*, *ListDir*.

The FSS flowchart depicts the unique steps followed depending on the *client* and *method* contained in the connection's request. The diagram is too large for a single page, so it is split into figures 3.23 and 3.24. Figure 3.24 contains the architecture dependent tasks for the *GetFile* method specifically.

The ***GetFile*** request returns either the requested files data, if client is internal, or the payload if external. The result of getting the payload in this architecture is the payload being written to disk.

The ***PutFile*** request routes the passed by file name and data to its respective CFH.

The ***PutVar*** request saves the passed variable name and value as a key value pair in an object in memory.

The ***GetVar*** request pulls the value of the passed variable name from memory and returns it to the connection.

The ***ListDir*** request returns an object containing a list of files and directories from the given file path. The FSS generates this object by pulling and parsing all managed file paths from the iCFH.

File Store Client is referenced as a sub-process in this architecture and architecture 4. This denotes the processes which will communicate with the IDH.

3.10.1.4 FarmOS Data Requester

See figure 3.14 for the FosC flowchart. Architectures 3 and 4 use the same Requester, and it differs from architecture's 1 and 2 by use of the IDH. Rather then writing files to disk after pulling them from the FosC, it sends them to the IDH. It also does not write a file to disk to denote its completion. Instead, it stores a variable in the IDH that can be requested from the BPA.

Process Steps

1. Send "requester_complete" variable to IDH with value "false".
2. Pull raw data logs from FarmOS connector.
3. Scrape geospatial location data and associated data file id's from each log and store in a new log object.
4. Request each raw data file required for training by their scraped file ID's.
5. Send each data file to IDH.
6. Send "requester_complete" variable to IDH with value "true".

3.10.1.5 Batch Processing Application

Similar to the Requester, the BPA in this architecture differs in the use of the IDH. When the BPA starts it pulls the *requester_complete* variable from the IDH. When the IDH returns true for this var the BPA starts the Generator. The Generator pulls all raw data files from the IDH and sends all generated labels and images to the IDH. The Trainer then pulls the training labels and images from the IDH and sends its results to the IDH.

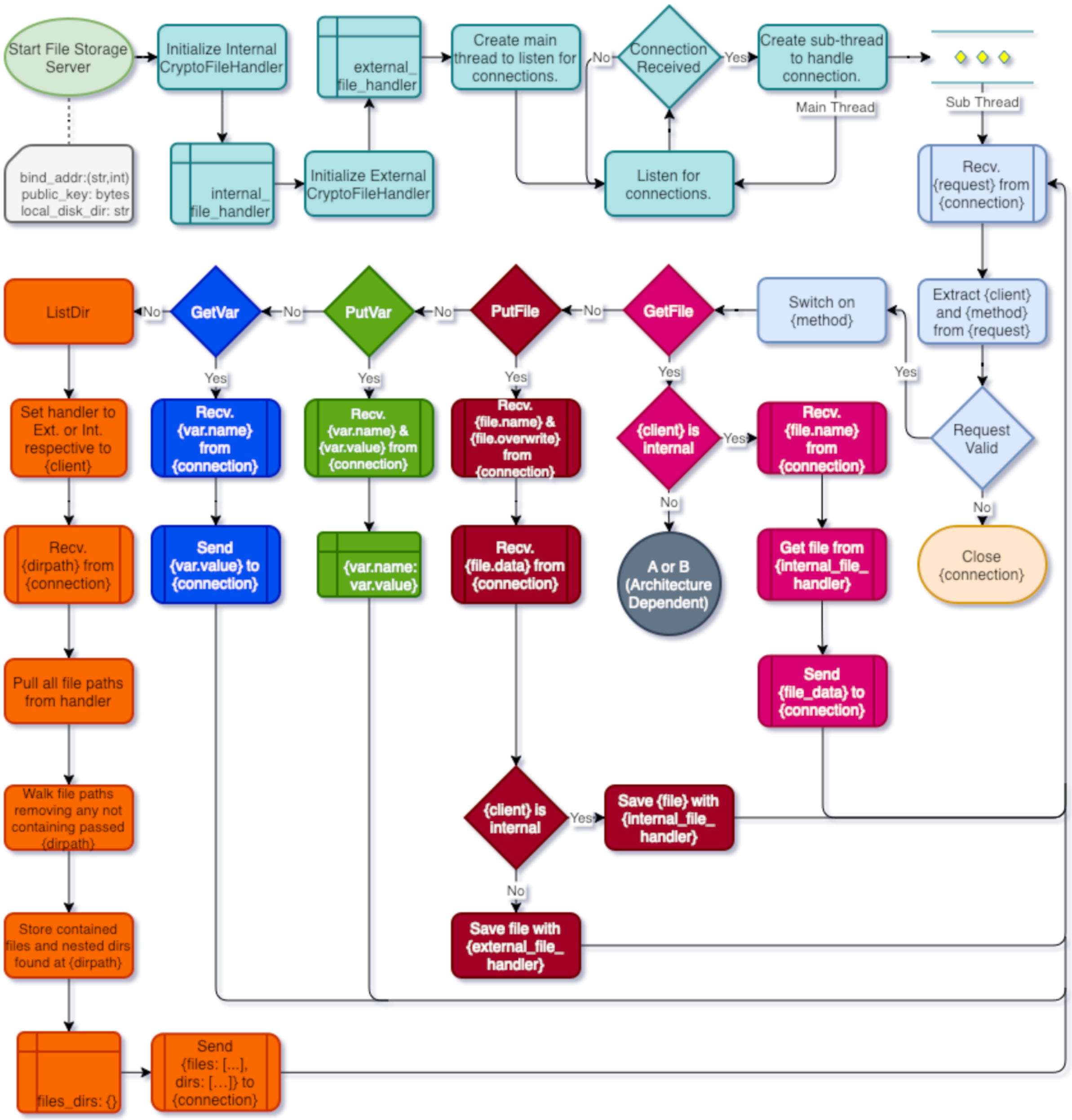
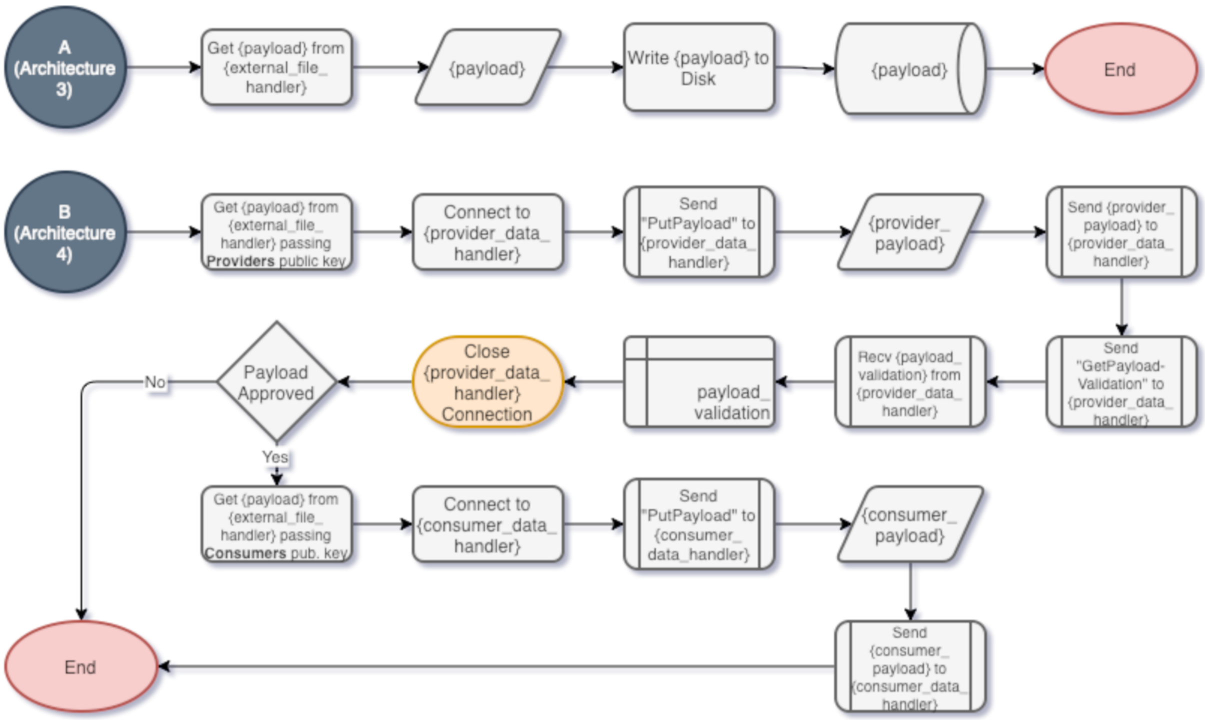
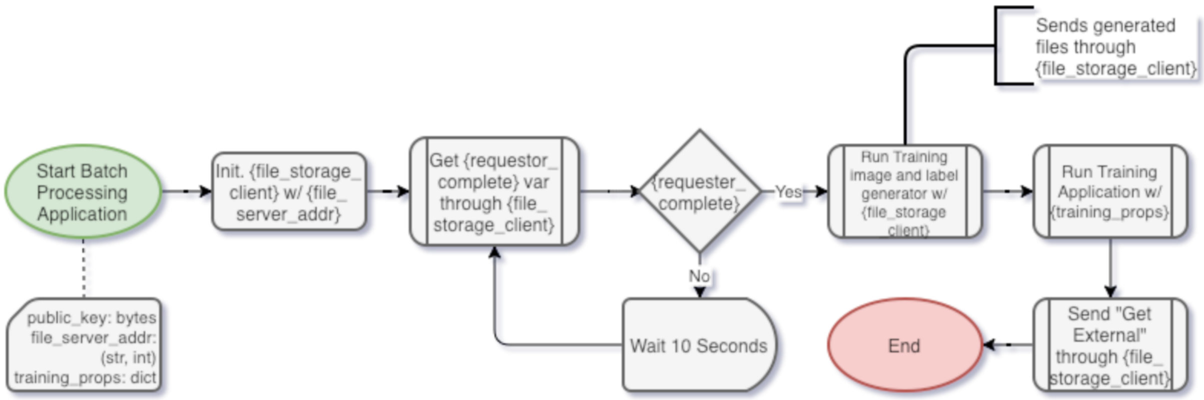

Start File Storage Server
Initialize Internal CryptoFileHandler
external_ file_handler
Create main thread to listen for connections.
No
Connection Received
Yes
Create sub-thread to handle connection.
Main Thread
Sub Thread
bind_addr:(str,int) public_key: bytes local_disk_dir: str
internal_ file_handler
Initialize External CryptoFileHandler
Listen for connections.
Recv. {request} from {connection}
ListDir
No
GetVar
No
PutVar
No
PutFile
No
GetFile
Switch on {method}
Extract {client} and {method} from {request}
Yes
Yes
Yes
Yes
Yes
Set handler to Ext. or Int. respective to {client}
Recv. {var.name} from {connection}
Recv. {var.name} & {var.value} from {connection}
Recv. {file.name} & {file.overwrite} from {connection}
{client} is internal
Yes
Recv. {file.name} from {connection}
Request Valid
No
No
Recv. {dirpath} from {connection}
Send {var.value} to {connection}
{var.name: var.value}
Recv. {file.data} from {connection}
A or B (Architecture Dependent)
Get file from {internal_file_ handler}
Close {connection}
Pull all file paths from handler
Send {file_data} to {connection}
Walk file paths removing any not containing passed {dirpath}
{client} is internal
Yes
Save {file} with {internal_file_ handler}
No
Store contained files and nested dirs found at {dirpath}
Save file with {external_file_ handler}
files_dirs: {}
Send {files: [...], dirs: [...]} to {connection}

Figure 3.23. File Storage Server



A (Architecture 3)
Get {payload} from {external_file_handler}
{payload}
Write {payload} to Disk
{payload}
End
B (Architecture 4)
Get {payload} from {external_file_handler} passing Providers public key
Connect to {provider_data_handler}
Send "PutPayload" to {provider_data_handler}
{provider_payload}
Send {provider_payload} to {provider_data_handler}
Send "GetPayload-Validation" to {provider_data_handler}
Recv {payload_validation} from {provider_data_handler}
payload_validation
Close {provider_data_handler} Connection
Payload Approved
No
Yes
Get {payload} from {external_file_handler} passing Consumers pub. key
Connect to {consumer_data_handler}
Send "PutPayload" to {consumer_data_handler}
{consumer_payload}
Send {consumer_payload} to {consumer_data_handler}
End

Figure 3.24. File Storage Server - Architecture Specific *GetFile* Method Steps



Sends generated files through {file_storage_client}
Start Batch Processing Application
Init. {file_storage_client} w/ {file_server_addr}
Get {requestor_complete} var through {file_storage_client}
{requester_complete}
Yes
No
Run Training image and label generator w/ {file_storage client}
Run Training Application w/ {training_props}
Wait 10 Seconds
End
Send "Get External" through {file_storage_client}
public_key: bytes
file_server_addr: (str, int)
training_props: dict

Figure 3.25. Network Architecture 3 - Batch Processing Application

Process Steps

1. Continuously request the "requester_complete" variable from the IDH.

2. When IDH returns true, start the Generator.

3. Generate training images and labels requesting raw data from the IDH as needed.

4. Send training labels and images to IDH as they are generated.

5. Run Trainer on pulling training labels and images from IDH as needed.

6. Training results are sent to IDH.

7. When Trainer completes, "GetFile" is sent to IDH.

3.10.2 Data Flow

The data flow tables for this architecture have increased considerably. There are 14 steps in the container data flow and 30 steps in the complete data flow. These increases come from the new IDH process. Every time a file needs to be written to or read from the IDH, additional encryption or decryption steps are required in the route. This only time data is written directly to disk in this architecture is the final step when the results payload is ready. All other disk writes are now contained only in the IDH's File System.

While not reflected in the data-flow tables, training images and labels are read thousands of times during training. These additional steps (especially encryption/decryption) have a substantial impact on the overall throughput of the architecture.

Tables 3.7 and 3.8 show the core-process and sub-process data flow respectively. See section 3.6.2 for a detailed description on the structure of each table.

Table 3.7. Architecture 3 - Container Data Flow

Step	Data Type	Origin	Destination
1	Raw	FosC-Provider	FosC-Consumer
2	Raw	FosC-Consumer	Requester
3	Raw	Requester	IDH
4	Encrypted Raw	IDH	IDH File System
5	Encrypted Raw	IDH File System	IDH
6	Raw	IDH	BPA
7	Training	BPA	IDH
8	Encrypted Training	IDH	IDH File System
9	Encrypted Training	IDH File System	IDH
10	Training	IDH	BPA
11	Results	BPA	IDH
12	Encrypted Results	IDH	IDH File System
13	Encrypted Results	IDH File System	IDH
14	Payload	IDH	Disk

Table 3.8. Architecture 3 - Complete Data Flow

Step	Data Type	Origin	Destination	Sub Process
1	Raw	FosC-Provider	FosC-Consumer	FosC-Consumer
2	Raw	FosC-Consumer	Requester	Data Requester
3	Raw	Requester	Requester	File Storage Client
4	Raw	Requester	IDH	File Storage Server
5	Raw	IDH	IDH	Cryptographic File Handler
6	Raw	IDH	IDH	Internal Cryptographic File handler
7	Encrypted Raw	IDH	IDH File System	Internal Cryptographic File handler
8	Encrypted Raw	IDH File System	IDH	Internal Cryptographic File handler
9	Raw	IDH	IDH	Cryptographic File Handler
10	Raw	IDH	IDH	File Storage Server
11	Raw	IDH	BPA	File Storage Client
12	Raw	BPA	BPA	Generator
13	Training	BPA	BPA	File Storage Client
14	Training	BPA	IDH	File Storage Server
15	Training	IDH	IDH	Cryptographic File Handler
16	Training	IDH	IDH	Internal Cryptographic File handler
17	Encrypted Training	IDH	IDH File System	Internal Cryptographic File handler
18	Encrypted Training	IDH File System	IDH	Internal Cryptographic File handler
19	Training	IDH	IDH	Cryptographic File Handler
20	Training	IDH	IDH	File Storage Server
21	Training	IDH	BPA	File Storage Client
22	Training	BPA	BPA	Training Application
23	Results	BPA	BPA	File Storage Client
24	Results	BPA	IDH	File Storage Server
25	Results	IDH	IDH	Cryptographic File Handler
26	Results	IDH	IDH	External Cryptographic File Handler
27	Encrypted Results	IDH	IDH File System	External Cryptographic File Handler
28	Encrypted Results	IDH File System	IDH	External Cryptographic File Handler
29	Payload	IDH	IDH	Cryptographic File Handler
30	Payload	IDH	IDH	File Storage Server
31	Payload	IDH	Disk	File Storage Server

3.10.3 Docker Networks

See figure 3.26 for this architectures network diagram. The `internal_net` docker network is new in architecture 3 and is shared by the BPA, Requester and IDH. The Requester and BPA sharing this network means they have the ability to communicate with one another. However, no communication ever occurs between these two processes, so there is no line shown between them in figure 3.26.

The Consumer could attempt communication with the Requester in their custom BPA but it would be futile. The Requester is designed for the single basic purpose of requesting data from the FosC and storing it in the IDH. Essentially, it just requests then forwards data along a strictly defined path. Their are no openings in its logic that could allow any additional functionality including potential communication requests from the BPA.

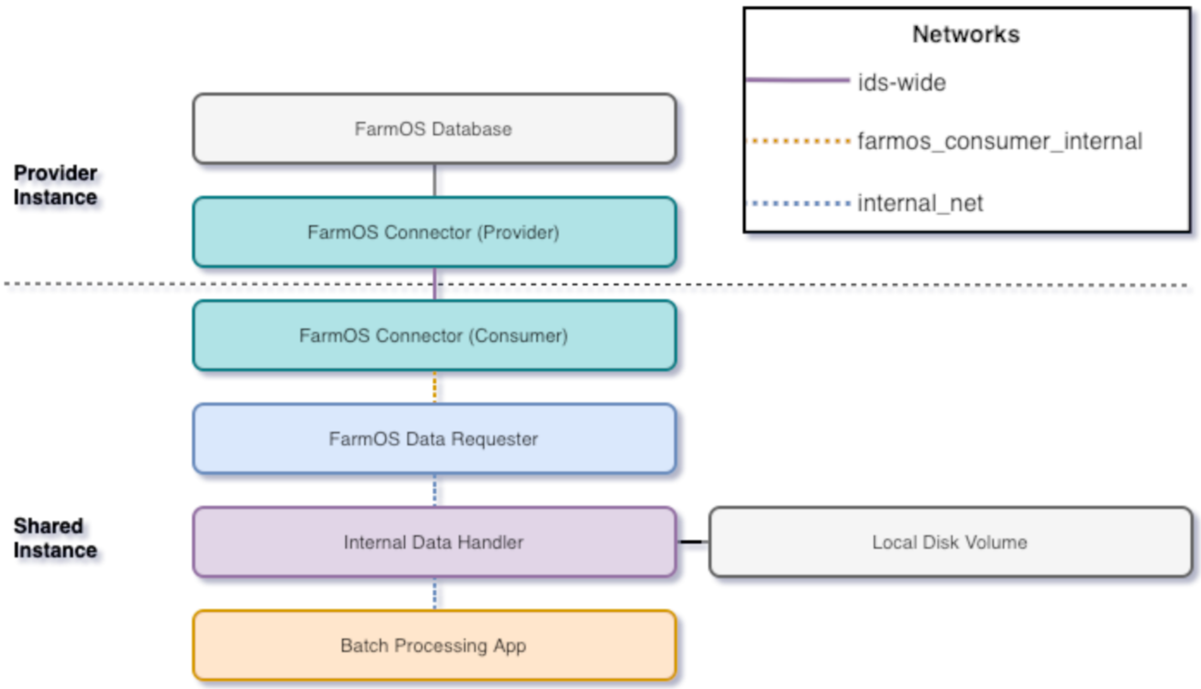


Figure 3.26. Network Architecture 3 - Container and Network Structure

3.10.4 Docker Compose

The addition of the IDH results in significant changes to the compose file. The first notable change is in the shared volumes for the Requester and BPA services. The incorporation of the IDH alleviates disk storage requirements for both the Requester and BPA. The Requester no longer has any local disk access and the BPA had all of its volumes except its environment packages removed.

Notice that the requester having its shared volumes removed means it is incapable of writing anything to disk. The containers file-system is read only and it has no shared volumes. This is consistent with its purpose of purely requesting then forwarding data. This is true with the BPA as well. Its file-system and only shared volume are both read only.

The definition of the `internal_net` is contained in lines 46-48 of the compose. Setting it to internal restricts the services that do not extend additional networks to communication within `internal_net` only.

The IDH is set to a specific IP on line 10. The IP is then mapped to the key *internal_data_handler* on line 18 and 38. Enforcing an IP address then mapping it as shown is done so the Requester and BPA can always refer to the key as the address of the IDH server. The IP can be changed, but, as long as the mapping is correct, the logic for the IDC sub process in the Requester and BPA never needs to change because it can always refer to that key. Strickly setting it to an IP also means that if another service (e.g. the BPA) in the same internal network were to attempt spoofing the IP for any malicious reason, a resulting error would be thrown causing the IDH to stop and therefor ending the entire network architecture.

The final noteworthy component is line 251. There is a port mapping but only to an `internal_net` port. This can be observed in that there is a number on the right side of the colon but not the left. If there were a number on the left side of the colon it would mean that the port was mapped from the container to the host network. However, these processes only require communication within each other and not to any outside sources that would require opening a host port.

```
1  services:
2    internal_data_handler:
3      image: secure_training_network:architecture3_internal_data_handler
4      env_file:
5        - internal_data_handler.env
```

```yaml
    ports:
      - :9090
    networks:
      internal_net:
        ipv4_address: 192.168.100.100
    volumes:
      - ./payload_storage:/app/payload_storage
      - ./keys:/app/keys:ro

  requester:
    depends_on:
      - internal_data_handler
    image: consumer:architecture3_data_requester
    env_file:
      - requester.env
    networks:
      - farmos-consumer-internal
      - internal_net
    extra_hosts:
      internal_data_handler: 192.168.100.100
    read_only: true;

  batch_processing_app:
    depends_on:
      - requester
    image: consumer:architecture3_bpa
    env_file:
      - bpa.env
    networks:
      - internal_net
    extra_hosts:
      internal_data_handler: 192.168.100.100
    volumes:
      - ./dlvenv/dlvenv:/dlvenv:ro
    read_only: true;

networks:
  farmos-consumer-internal:
    external: true
  internal_net:
    driver: bridge
    internal: true
```

3.11 Network Architecture 4: External Data Handlers

External Data Handlers (EDH) are introduced in Architecture 4. The two EDH's are the Provider Data Handler (PDH) and the Consumer Data Handler (CDH) and they are designed to transmit training results to their respective external client instance. The PDH and CDH improve the security for Provider's data and Consumer's model respectively. This architecture is the most secure of all 4 discussed and meets all criteria defined in the threat model in section 3.1.

In this architecture, training results are no longer written to disk unencrypted. Instead, they are first passed to the Provider's instance for validation. The Provider's validation of the results entails verifying the results do not contain secure data. Moreover, the results should only contain the trained model and any additional files the Provider may have previously agreed on (e.g. training loss data). If approved by the Provider, results are transferred to the Consumer's instance marking the completion of the network architecture. If the results are not approved, the results payload is deleted and the CDH is prematurely terminated.

This architecture enables multiple Consumer's to utilize the Provider's data from the same machine. The only limit on the number of Consumer's that could concurrently use the Provider's data from the same instance is the processing power of the instance itself. Specifically, the instances memory allotment, GPU processing power and CPU capabilities.

Figure 3.27 shows this architectures complete process structure.

3.11.1 Core Processes

The Provider External Data Handler (PDH) and Consumer External Data Handler (CDH) are the new processes in architecture 4. All other processes match those from Architecture 3 but with alterations. Each EDH is a server which exists on the shared instance. They also have a paired client for external communication. The PDH and CDH use the Provider Data Client (PDC) and Consumer Data Client (CDC) respectively. Each server and client duo uses attestation to validate their connection prior to any data transmission.

The incorporation of the EDH processes is what enabled this architecture to concurrently support n data Consumer's. This is not supported in architecture 3 because it writes the results payload to disk following training. In this architecture, no data is ever written to disk unencrypted including the payload, so there it is no longer possible for users on the shared instance to copy others data unknowingly.

Similar to the IDH's FSS, the PDH and CDH server's are multi-threaded. Moreover, they receive connections and handle them using sub-threads. These can be seen in figures 3.28 and 3.30.

3.11.1.1 Provider Data Handler

The purpose of the PDH was the facilitate the Provider's validation of the Consumer's training results. See figure 3.28 for the Provider Data Handler.

Process Steps

1. Verify connection with an external requesting process is the PDC.
2. Receive training results payload from IDH following training completion.
3. Pass the payload to the PDC.
4. Wait for payload approval from client.
5. Forward approval to IDH.

The most important of the process steps, with regards to security, is step 1. It was accomplished using an attestation handshake processor. See the steps for this process in figure 3.28 at the yellow circle marked *B: Execute Attestation Handshake*. Essentially, it generates and

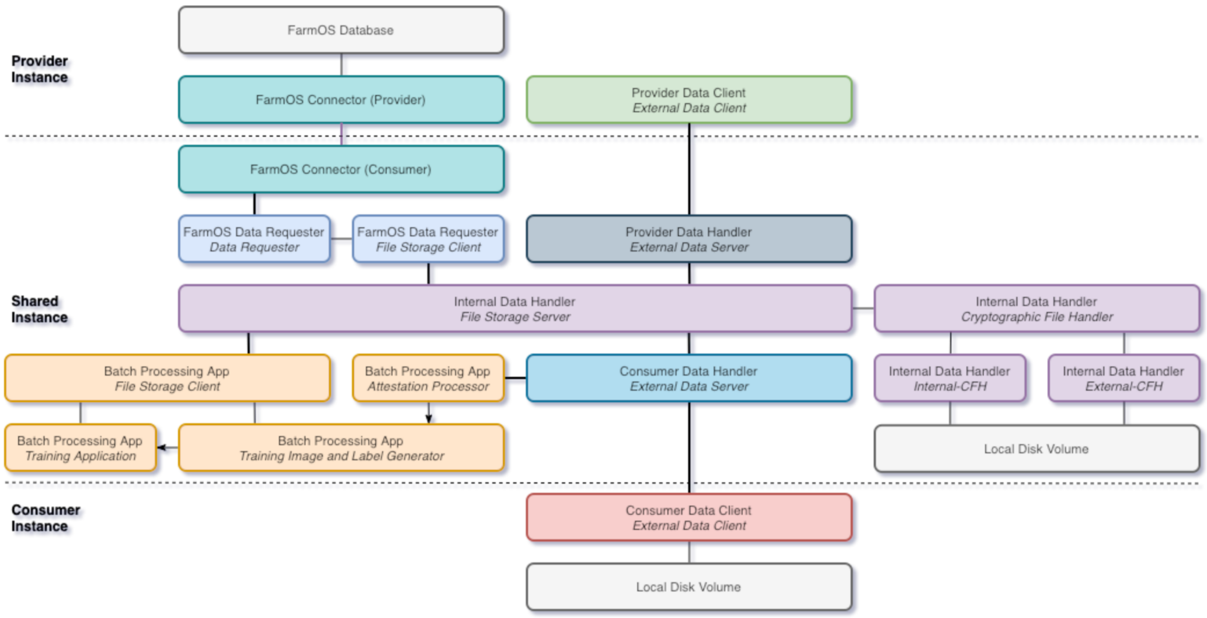


Figure 3.27. Network Architecture 4 - Complete Process Structure

encrypts a random token using the Provider's public key, sends this encrypted key to the client, waits for the client to return the key encrypted with the Provider's private key, decrypts the returned key using the Provider's public key and compares the decrypted key with the original. If the keys match, the handshake passes and the client connection is kept alive. If the keys do not match, the connection is closed.

Step's 2 through 5 are encompassed in the request types (*PutPayload, GetPayload, PutPayloadValidation, GetPayloadValidation*) in figure 3.28. The requests for *PutPayload* and *GetPayloadValidation* are sent, in that order, by the IDH following training completion. Hence, these requests only assert the connection is from an internal source. The *GetPayload* and *PutPayloadValidation* requests come in that order from the PDC, thus the Attestation Handshake process must be run for them.

The PDH completes after the *GetPayloadValidation* is requested and successfully fulfilled.

3.11.1.2 Provider Data Client

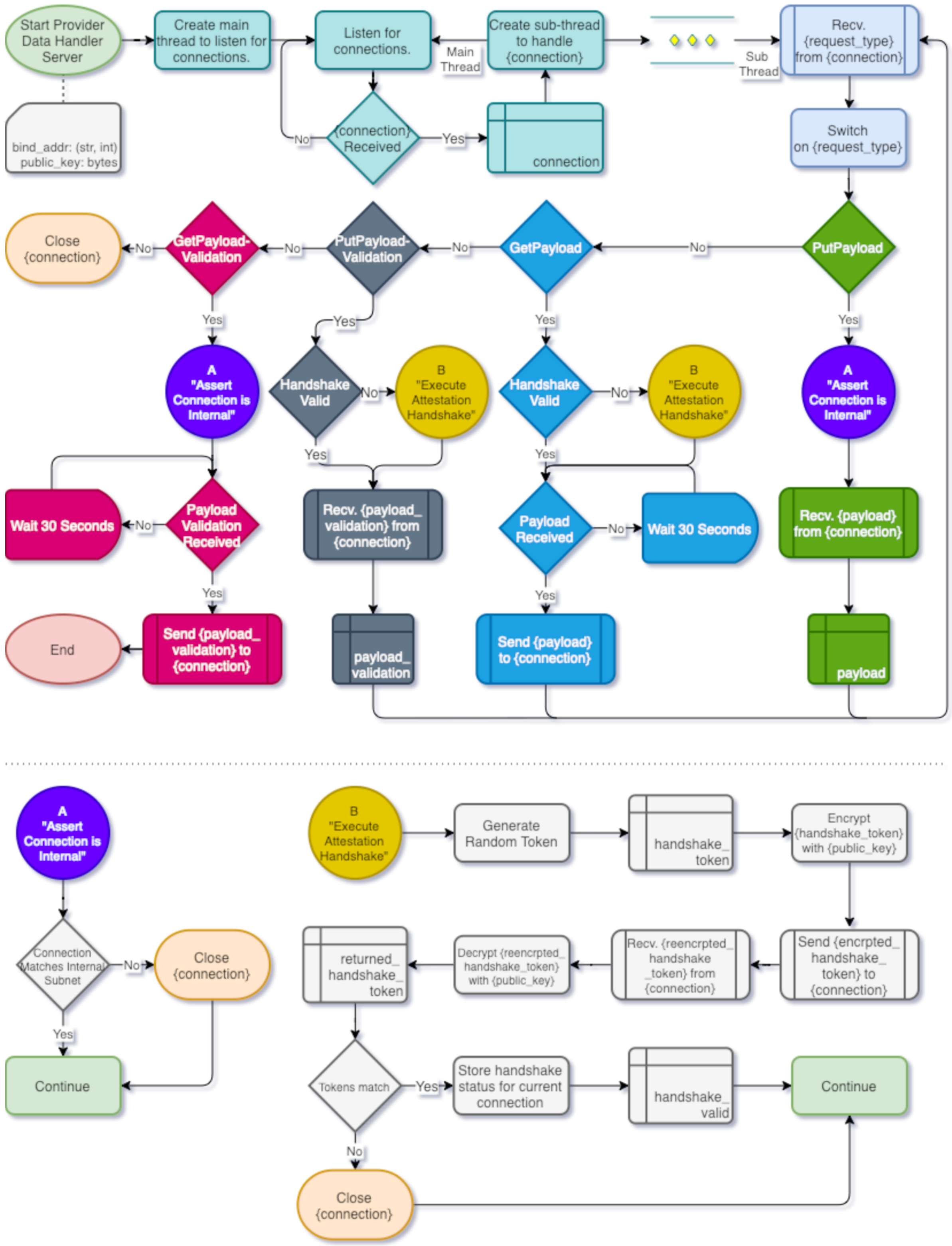
The complete purpose of the PDC is to succeed in the attestation handshake check and run the *GetPayload* and *PutPayloadValidation* commands. The PDC flowchart is shown in figure 3.29. The most noteworthy step in the PDC is when it waits for user input that signifies the validity of the payload. When the payload is received from the PDH it is decrypted and written to disk. The PDC then begins waiting for the user input.

3.11.1.3 Consumer Data Handler

The CDH is considered an EDH like the PDH due to its primary purpose of transmitting remote data. However, its internal procedures and requirements are very different from the PDH.

1. Support the handshake between the BPA and CDC.
2. Receive the payload from the IDH following provider verification.
3. Pass the payload to the CDC.
    - This step only occurs of the payload was approved in step 2.

The attestation handshake in step (1) is used for more then just remote client authentication. It is also used to improve the security of the BPA. This will be discussed more in the BPA subsection 3.11.1.5 below. The CDH receives an encrypted token from the BPA and forwards it to the CDC. It then waits for the CDC to return an encrypted token which is then returned back



Start Provider Data Handler Server
bind_addr: (str, int)
public_key: bytes
Create main thread to listen for connections.
Listen for connections.
Main Thread
Create sub-thread to handle {connection}
Sub Thread
Recv. {request_type} from {connection}
{connection} Received
No
Yes
connection
Switch on {request_type}
PutPayload
GetPayload
PutPayload-Validation
GetPayload-Validation
Close {connection}
Yes
A "Assert Connection is Internal"
Handshake Valid
B "Execute Attestation Handshake"
Wait 30 Seconds
Payload Validation Received
Send {payload_ validation} to {connection}
End
Recv. {payload_ validation} from {connection}
payload_ validation
Payload Received
Send {payload} to {connection}
Recv. {payload} from {connection}
payload
Connection Matches Internal Subnet
Continue
Generate Random Token
handshake_ token
Encrypt {handshake_token} with {public_key}
Send {encrpted_ handshake_ token} to {connection}
Recv. {reencrpted_ handshake _token} from {connection}
Decrypt {reencrpted_ handshake_token} with {public_key}
returned_ handshake_ token
Tokens match
Store handshake status for current connection
handshake_ valid

Figure 3.28. Provider Data Handler

to the BPA. These steps are accomplished jointly between the *ExecHandshake* and *GetPayload* requests which are called by the BPA and CDC respectively.

Step (3) above denotes the stage when the CDH exports the payload to the CDC.

The *PutPayload* request type is used by the IDH to pass the results payload to the CDH. This occurs after IDH receives approval to export the payload. This is listed as step 2 above.

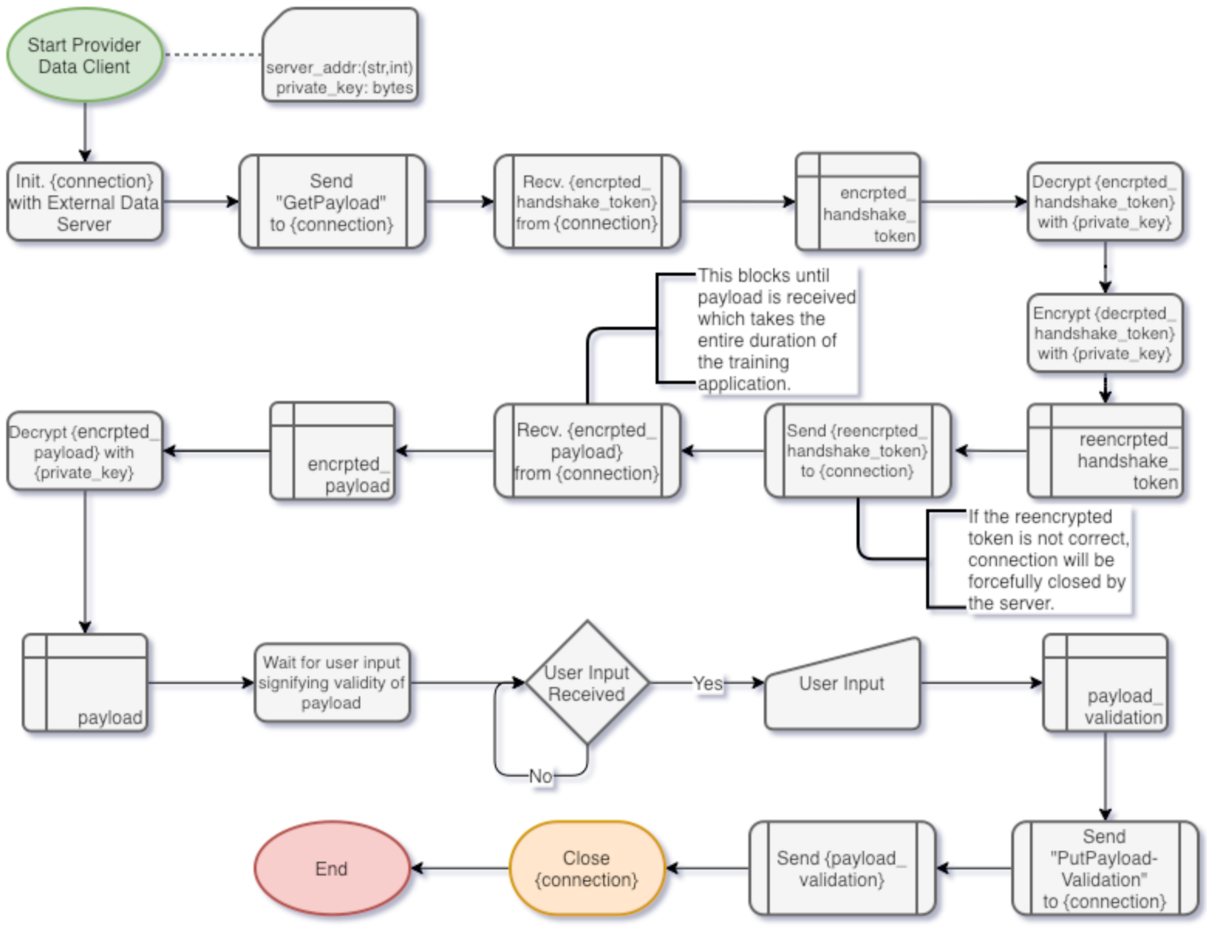
3.11.1.4 Consumer Data Client

Similar to the PDC, the CDC was designed specifically to communicate with the CDH, but also to work with the Consumer's BPA through the CDH. The PDC's objectives are listed below.

1. Correctly answer the attestation handshake request to signal the BPA to start.

2. Receive the payload from the CDH.

3. Decrypt payload and write to disk.



Start Provider Data Client
server_addr:(str,int)
private_key: bytes
Init. {connection} with External Data Server
Send "GetPayload" to {connection}
Recv. {encrpted_handshake_token} from {connection}
encrpted_handshake_token
Decrypt {encrpted_handshake_token} with {private_key}
Encrypt {decrpted_handshake_token} with {private_key}
This blocks until payload is received which takes the entire duration of the training application.
reencrpted_handshake_token
Send {reencrpted_handshake_token} to {connection}
If the reencrypted token is not correct, connection will be forcefully closed by the server.
Recv. {encrpted_payload} from {connection}
encrpted_payload
Decrypt {encrpted_payload} with {private_key}
payload
Wait for user input signifying validity of payload
User Input Received
Yes
No
User Input
payload_validation
Send "PutPayload-Validation" to {connection}
Send {payload_validation}
Close {connection}
End

Figure 3.29. Provider Data Client

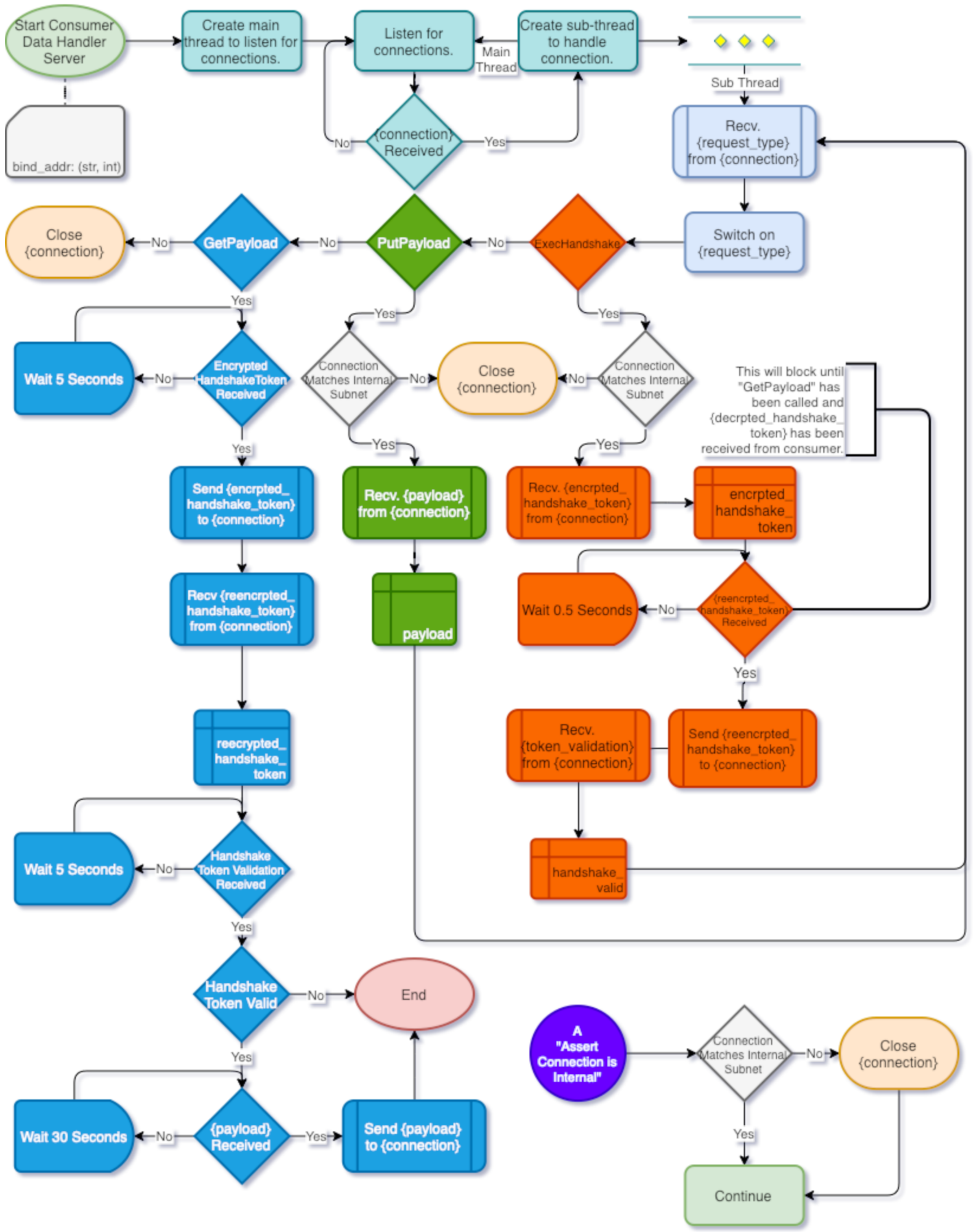
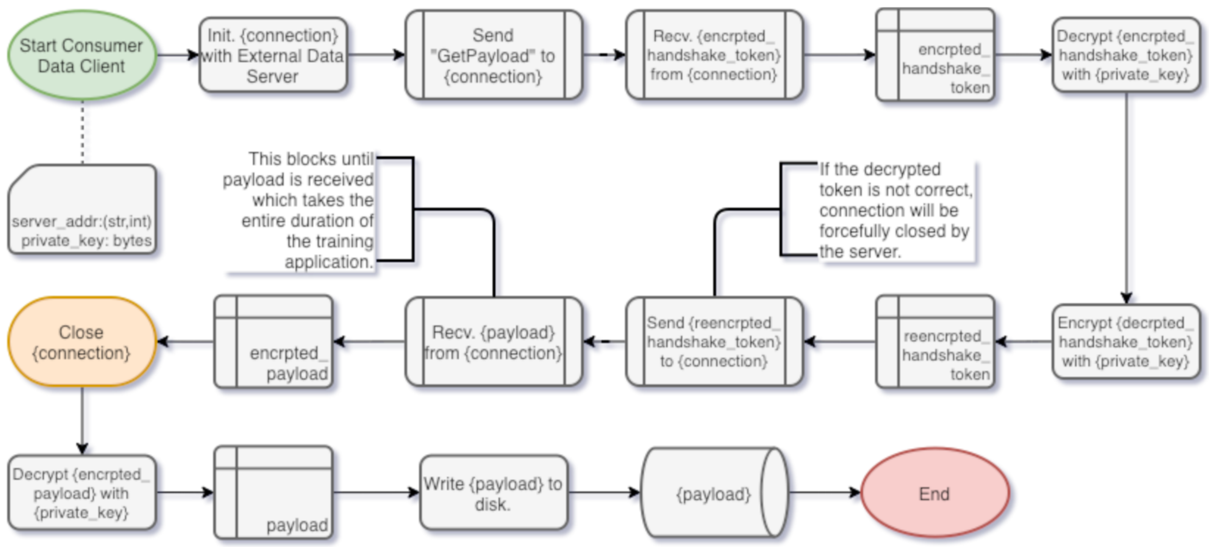

Start Consumer Data Handler Server
bind_addr: (str, int)
Create main thread to listen for connections.
Listen for connections.
Main Thread
Create sub-thread to handle connection.
Sub Thread
{connection} Received
No
Yes
Recv. {request_type} from {connection}
Switch on {request_type}
ExecHandshake
PutPayload
GetPayload
Close {connection}
Wait 5 Seconds
Encrypted HandshakeToken Received
Connection Matches Internal Subnet
This will block until "GetPayload" has been called and {decrpted_handshake_token} has been received from consumer.
Send {encrpted_handshake_token} to {connection}
Recv. {payload} from {connection}
Recv. {encrpted_handshake_token} from {connection}
encrpted_handshake_token
Recv {reencrpted_handshake_token} from {connection}
payload
Wait 0.5 Seconds
{reencrpted_handshake_token} Received
reecrypted_handshake_token
Recv. {token_validation} from {connection}
Send {reencrpted_handshake_token} to {connection}
handshake_valid
Handshake Token Validation Received
Handshake Token Valid
End
{payload} Received
Send {payload} to {connection}
Wait 30 Seconds
A "Assert Connection is Internal"
Continue

Figure 3.30. Consumer Data Handler



Start Consumer Data Client
Init. {connection} with External Data Server
Send "GetPayload" to {connection}
Recv. {encrpted_ handshake_token} from {connection}
encrpted_ handshake_ token
Decrypt {encrpted_ handshake_token} with {private_key}
server_addr:(str,int) private_key: bytes
This blocks until payload is received which takes the entire duration of the training application.
If the decrypted token is not correct, connection will be forcefully closed by the server.
Close {connection}
encrpted_ payload
Recv. {payload} from {connection}
Send {reencrpted_ handshake_token} to {connection}
reencrpted_ handshake_ token
Encrypt {decrpted_ handshake_token} with {private_key}
Decrypt {encrpted_ payload} with {private_key}
payload
Write {payload} to disk.
{payload}
End

Figure 3.31. Consumer Data Client

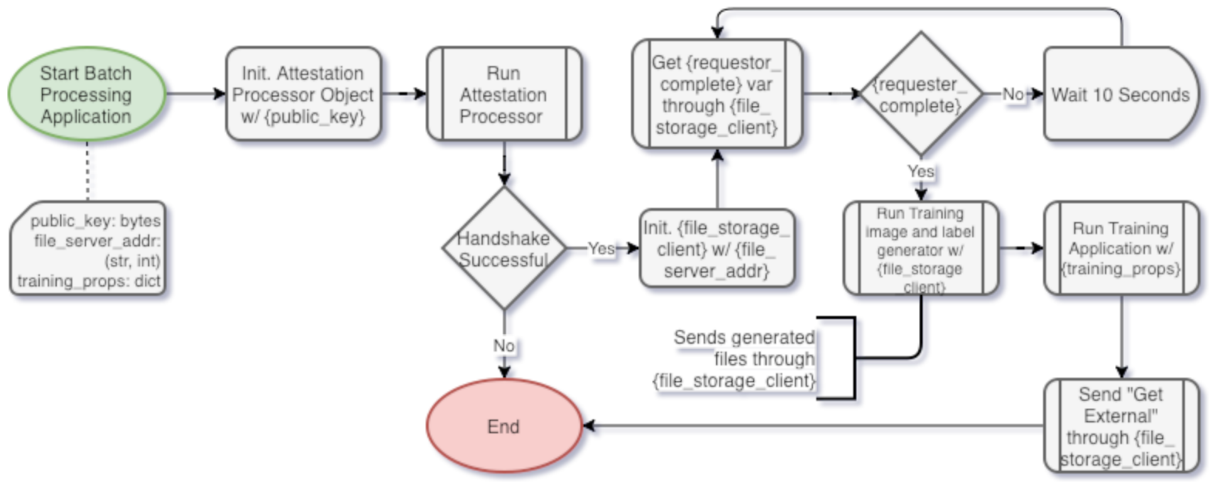
Step (1) will be discussed more in subsection 3.11.1.5, but if step (1) above were to fail, the BPA would never run on the shared compute instance. This improves the security of the Consumer's training app because it asserts that it is only being run when the Consumer approves it.

The final steps (2 & 3) are to receive the payload, decrypt it and write it to disk. Step (2) marks the end of the architecture internal to the shared instance and step (3) marks the complete architectures conclusion.

3.11.1.5 Batch Processing Application

An additional layer of security is added to this architectures BPA. That is an attestation handshake between the BPA and the Consumer's remote instance. The BPA has no external network access, so the CDH is used as an intermediary in the handshake process. As in architectures 2 and 3, the BPA is entirely obfuscated. Moreover, if the Provider were to copy the application they could run it, but could not alter it or see its contents. Incorporating the remote attestation handshake as the first step in the BPA means even if the Provider copied the BPA they would be unable to run it because they could not decrypt the randomly generated key. Even if they were to pull the unencrypted key directly from memory, the BPA expects the key to be returned encrypted with the Consumer's private key which the Provider would have no access to.

Beyond the attestation processor, the BPA follows the same steps as architecture 3. See figure 3.32 for this architectures BPA flowchart.



Start Batch Processing Application
public_key: bytes
file_server_addr: (str, int)
training_props: dict
Init. Attestation Processor Object w/ {public_key}
Run Attestation Processor
Handshake Successful
Yes
No
End
Init. {file_storage_client} w/ {file_server_addr}
Get {requestor_complete} var through {file_storage_client}
{requester_complete}
No
Wait 10 Seconds
Yes
Run Training image and label generator w/ {file_storage client}
Run Training Application w/ {training_props}
Sends generated files through {file_storage_client}
Send "Get External" through {file_storage_client}

Figure 3.32. Network Architecture 4 - Batch Processing Application

3.11.2 Data Flow

The addition of the PDH, PDC, CDH, and CDC in this network only extend the data flow from the end of training compared with the previous architecture. Moreover, they cause new steps in the data flow that replace the final ”write payload to disk” step from architecture 3. Notice that there are no writes directly to the shared instances disk. There is only a single disk write in the Complete Data Flow for this network and that occurs on the Consumer’s instance. All other disk writes have been replaced with writes to the IDH’s file system for data on the Shared Instance.

Tables 3.9 and 3.10 show the Container and Complete data flow respectively. See section 3.6.2 for a detailed description on the structure of each table.

In the Container Data Flow table in table 3.9, the above mentioned new steps start at step 14 and continue to the end. An additional 7 steps in the Container Data Flow may seem significant. However, as will be shown in the results chapter, the time added in fulfilling these steps in negligible in comparison to the runtime of the entire architecture. The only step which would extend runtime is step 17 and it would extend the runtime by an unknown length. This step blocks all further steps until the Provider reviews the payload and submits their approval. The resulting extension in throughout relying entirely on the Provider in this step. It is impossible to gauge how much time it will take for the Provider to review it. For this reason, I neglect the time altogether in my results.

The data flow in both of these tables relies on the payload being approved in the Provider in step 14. If the payload was disapproved, the Container Data Flow and Complete Data Flow would end at lines 18 and 35 respectively.

The steps in the Complete Data Flow in table 3.9 have increased by 10 in comparison to architecture 3. The time it takes for the user input in step 32 is neglected as a slowdown because the time it takes is no fault of the networks.

Table 3.9. Architecture 4 - Container Data Flow

Step	**Data Type**	**Origin**	**Destination**
1	Raw	FosC-Provider	FosC-Consumer
2	Raw	FosC-Consumer	Requester
3	Raw	Requester	IDH
4	Encrypted Raw	IDH	IDH File System
5	Encrypted Raw	IDH File System	IDH
6	Raw	IDH	BPA
7	Training	BPA	IDH
8	Encrypted Training	IDH	IDH File System
9	Encrypted Training	IDH File System	IDH
10	Training	IDH	BPA
11	Results	BPA	IDH
12	Encrypted Results	IDH	IDH File System
13	Encrypted Results	IDH File System	IDH
14	Provider Payload	IDH	PDH
15	Provider Payload	PDH	PDC
16	Payload	PDC	PDC File System
17	Payload Approval	PDC	PDH
18	Payload Approval	PDH	IDH
19	Consumer Payload	IDH	CDH
20	Consumer Payload	CDH	CDC
21	Payload	CDC	Disk

Table 3.10. Architecture 4 - Complete Data Flow

Step	**Data Type**	**Origin**	**Destination**	**Sub Process**
1	Raw	FosC-Provider	FosC-Consumer	FosC-Consumer
2	Raw	FosC-Consumer	Requester	Data Requester
3	Raw	Requester	Requester	File Storage Client
4	Raw	Requester	IDH	File Storage Server
5	Raw	IDH	IDH	Cryptographic File Handler
6	Raw	IDH	IDH	Internal Cryptographic File handler
7	Encrypted Raw	IDH	IDH File System	Internal Cryptographic File handler
8	Encrypted Raw	IDH File System	IDH	Internal Cryptographic File handler
9	Raw	IDH	IDH	Cryptographic File Handler
10	Raw	IDH	IDH	File Storage Server
11	Raw	IDH	BPA	File Storage Client
12	Raw	BPA	BPA	Generator
13	Training	BPA	BPA	File Storage Client
14	Training	BPA	IDH	File Storage Server
15	Training	IDH	IDH	Cryptographic File Handler
16	Training	IDH	IDH	Internal Cryptographic File handler
17	Encrypted Training	IDH	IDH File System	Internal Cryptographic File handler
18	Encrypted Training	IDH File System	IDH	Internal Cryptographic File handler
19	Training	IDH	IDH	Cryptographic File Handler
20	Training	IDH	IDH	File Storage Server
21	Training	IDH	BPA	File Storage Client
22	Training	BPA	BPA	Training Application
23	Results	BPA	BPA	File Storage Client
24	Results	BPA	IDH	File Storage Server
25	Results	IDH	IDH	Cryptographic File Handler
26	Results	IDH	IDH	External Cryptographic File Handler
27	Encrypted Results	IDH	IDH File System	External Cryptographic File Handler
28	Encrypted Results	IDH File System	IDH	External Cryptographic File Handler
29	Provider Payload	IDH	IDH	Cryptographic File Handler
30	Provider Payload	IDH	IDH	File Storage Server
31	Provider Payload	IDH	PDH	External Data Server
32	Provider Payload	PDH	PDC	External Data Client
33	Payload	PDC	PDC File System	External Data Client
34	Payload Approval	PDC	PDH	External Data Server
35	Payload Approval	PDH	IDH	File Storage Server
36	Encrypted Results	IDH File System	IDH	External Cryptographic File Handler
37	Consumer Payload	IDH	IDH	Cryptographic File Handler
38	Consumer Payload	IDH	IDH	File Storage Server
39	Consumer Payload	IDH	CDH	External Data Server
40	Consumer Payload	CDH	CDC	External Data Client
41	Payload	CDC	Disk	External Data Client

3.11.3 Docker Networks

As the network diagram in figure 3.26 shows, many additional docker networks are incorporated in this architecture. Of the four supplemental docker networks and as their names reflect, two are specific to the PDH and the other two are for the CDH. Each EDH requires two networks because one is for internal communication and the other for external. Its vital that each of the external network not be shared with other internal processes as it would give them external network access as well opening holes in security. The new internal docker networks are `provider_internal_net` and `consumer_internal_net`. The new external networks are `provider_net` and `consumer_net`.

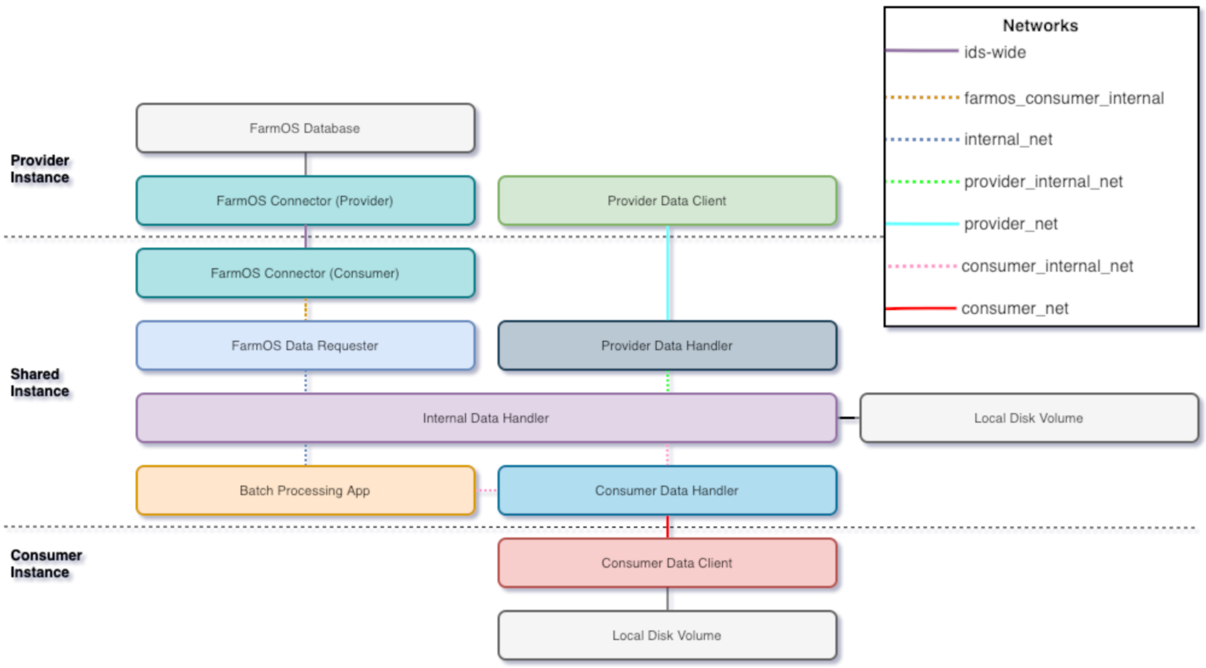


Figure 3.33. Network Architecture 4 - Container and Network Structure

3.11.4 Docker Compose

This compose file is clearly the most extensive given the large number of services running independently. However, there are no docker features used here that have not already been shown and discussed in prior architecture docker compose docs.

Noteworthy additions are on lines 7 and 21 which are in the `provider_data_handler` and `consumer_data_handler` services respectively. These lines includes port mappings between both the container's networks and the host network. In prior networks, all port mappings were internal only, so they only had an internal mapping (e.g. a port on the right side of the colon only). Here is shown a mapping to the host network as well because both EDH's require external network access which can only be accomplished through port mapping through the host network. This is why the use of attestation in these processes is necessary.

Also notice that the shared volume has been removed from the `internal_data_hander`. The payload is no longer written to disk in this architecture, so the volume used for saving the payload in architecture 3 could be removed here.

```
 1 services:
 2   provider_data_handler:
 3     image: secure_training_network:provider_data_handler
 4     env_file:
 5       - provider_data_handler.env
 6     ports:
 7       - 9091:9091
 8     networks:
 9       provider_net:
10         provider_internal_net:
11           ipv4_address: 192.168.200.100
12     volumes:
13       - ./keys/provider_public.pem:/app/key.pem:ro
14     read_only: true;
15 
16   consumer_data_handler:
17     image: secure_training_network:consumer_data_handler
18     env_file:
19       - consumer_data_handler.env
20     ports:
21       - 9092:9092
22     networks:
23       consumer_net:
24         consumer_internal_net:
25           ipv4_address: 192.168.300.100
26     volumes:
27       - ./keys/consumer_public.pem:/app/key.pem:ro
```

```
    read_only: true;

  internal_data_handler:
    depends_on:
      - provider_data_handler
      - consumer_data_handler
    image: secure_training_network:architecture4_internal_data_handler
    env_file:
      - internal_data_handler.env
    ports:
      - :9090
    networks:
      internal_net:
        ipv4_address: 192.168.100.100
      provider_internal_net:
      consumer_internal_net:
    extra_hosts:
      provider_data_handler: 192.168.100.200
      consumer_data_handler: 192.168.100.300
    volumes:
      - ./keys/provider_public.pem:/app/keys/provider_public.pem:ro
      - ./keys/consumer_public.pem:/app/keys/consumer_public.pem:ro

  requester:
    depends_on:
      - internal_data_handler
    image: consumer:architecture4_data_requester
    env_file:
      - requester.env
    networks:
      - farmos-consumer-internal
      - internal_net
    extra_hosts:
      internal_data_handler: 192.168.100.100
    read_only: true;

  batch_processing_app:
    depends_on:
      - requester
    image: consumer:architecture4_bpa
    env_file:
      - bpa.env
    networks:
      - internal_net
      - consumer_internal_net
    extra_hosts:
      internal_data_handler: 192.168.100.100
      consumer_data_handler: 192.168.100.300
    volumes:
      - ./dlvenv/dlvenv:/dlvenv:ro
    read_only: true;

```

```yaml
networks:
  farmos-consumer-internal:
    external: true
  internal_net:
    driver: bridge
    internal: true
    ipam:
      config:
        - subnet: 192.168.100.0/24
  provider_net:
    driver: bridge
  provider_internal_net:
    driver: bridge
    internal: true
    ipam:
      config:
        - subnet: 192.168.200.0/24
  consumer_net:
    driver: bridge
  consumer_internal_net:
    driver: bridge
    internal: true
    ipam:
      config:
        - subnet: 192.168.300.0/24
```

3.12 Network Architecture 5: Theoretical Enclave Architecture

Architecture 5 has not been implemented, so it is described here from a purely theoretical perspective and is considered *future work*. Implementing this architecture in practice is not possible for the reasons mentioned previously in section 3.5.4.2. I provide the theory for this architecture as it may be possible in the future if AWS expands Nitro Enclaves to allow the usage of GPU's and to host more then one enclave one an ec2 instance. If the instantiation of this architecture were possible, it would be the most secure for both the Provider's Data and Consumer's Model Application.

In architectures 3 and 4, the security of data was improved using the IDH which encrypted data before writing it to disk. However, the data was not encrypted while actively in memory. Enclaves address this security risk. The RSA keys generated in the enclaves are used to encrypt data before it leaves the Provider's instance and the enclave itself. Before data reaches the Shared Instance through either the FosC or the enclaves, it is encrypted. Data is only decrypted with inside an enclave and new data is always encrypted before leaving an enclave.

3.12.1 Core Processes

The Complete Process Structure for this architecture can be seen in figure 3.34. This figure includes two processes bordered with a dashed line rather then solid. This is to highlight that these processes would be contained in Nitro Enclaves rather then containers.

FarmOS Connector The FosC-provider is changed in this architecture to encrypt data before transferring it. The provider_app of the FosC is where the change would be made. After pulling data from the FarmOS Database, it would need to call the KMS api to encrypt the raw data with the Generator's public key. All further functionality is the same as previous architectures FosC.

FarmOS Data Requester The Requester in this architecture would be reverted to the Requester used in architecture 2 because it would write the data directly to disk rather then to the IDH. This is possible because the data is already encrypted when the Requester receives it.

Payload Connectors Two new Trusted Connectors are shown in figure 3.34. The *Payload Validation Connector* (PVC) would replace the PDH and PDC from Architecture 4. Similarly, the *Payload Delivery Connector* (PDCo) would replace the CDH and CDC. These connectors are not limited to this architecture and could have been deployed in architecture 4. The only reason they were not was because the EDH's were sufficient and time was limited.

Notice that the consumer ends of both the PVC and PDCo are on the Shared Instance in figure 3.34 rather then the Provider and Consumer's instances. This is intentional as the payload originates on the Shared Instance and needs to be sent to the external instances. Further, The PVC and PDCo would not start with the rest of the architecture as they would consume resources unnecessarily. They would instead be spun up when training was nearly complete.

The PVC performs exactly the same duties as the PDH and PDC. To support these requirements, the PVC would need to be a bi-directional connector.

Unlike the PVC, the PDCo's tasks differ from those of the CDH and CDC. The PDCo does not need to support an attestation processor between the Consumers application and their instance. The complete purpose of the PDCo would be to send the Payload to the Consumer's Instance. Hence, the PDCo would be a one-directional connector.

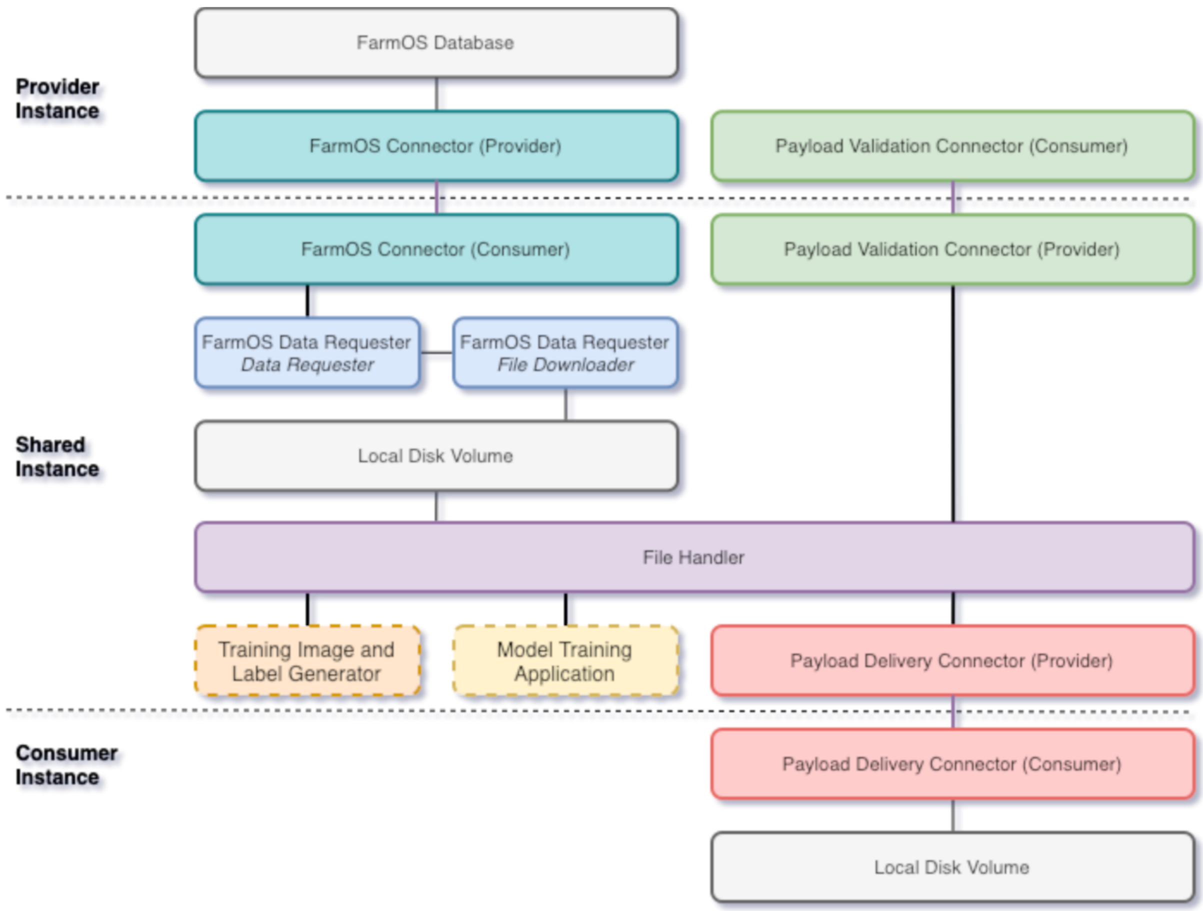

Provider Instance
FarmOS Database
FarmOS Connector (Provider)
Payload Validation Connector (Consumer)
Shared Instance
FarmOS Connector (Consumer)
Payload Validation Connector (Provider)
FarmOS Data Requester Data Requester
FarmOS Data Requester File Downloader
Local Disk Volume
File Handler
Training Image and Label Generator
Model Training Application
Payload Delivery Connector (Provider)
Consumer Instance
Payload Delivery Connector (Consumer)
Local Disk Volume

Figure 3.34. Network Architecture 5 - Complete Process Structure

File Handler The File Handler would be similar to the IDH, but differ enough to warrant a name change for clarity. It would be the client for communicating with the enclaves over their vsock (Virtual Machine Socket) channels. It would be a stripped down version of the IDH with all the internal-CFH functionality removed. Beyond communication with the enclaves, it would need to maintain the results payload and pass it to the PVC then PDCo in the same order as the IDH in architecture 4.

3.12.2 Enclaves Processes

The Generator and Trainer are the two processes that would run in enclaves. These processes perform all of the data processing and generation. If they could be hosted in enclaves, no secure data would ever need to be on the shared instance unencrypted. The logic for the Generator and Trainer would remain the same apart from some additional steps that would need to be incorporated as required by enclaves.

The Generators public key must be used on the Provider instance to encrypt the raw data before it is sent over the connector.

The Trainers public key must be passed to the Generator. The Generator uses this key to encrypt the training images and labels it generates from the raw data.

3.12.3 Data Flow

The first stage in this network is getting each enclaves public keys in place for encrypting data. This is done using KMS and therefor is not shown in the data flow table show in table 3.11.

This architectures Data Flow table shows almost all data as encrypted in the data type row. This is an improvement compared to all prior architectures. The only steps in the entire table including unencrypted data are on rows 15 and 20. These steps, however, are not performed on the shared instance but on the Provider and Consumer's instances respectively. The only times when data would not be encrypted on the shared instance is between rows 6 and 7 as well as between rows 10 and 11. Both of these points are when the data is in the enclaves.

Table 3.11. Architecture 5 - Data Flow

Step	**Data Type**	**Origin**	**Destination**
1	Encrypted Raw	FosC-provider	FosC-consumer
2	Encrypted Raw	FosC-consumer	Data Requester
3	Encrypted Raw	Data Requester	Data Downloader
4	Encrypted Raw	Data Downloader	Disk
5	Encrypted Raw	Disk	File Handler
6	Encrypted Raw	File Handler	Generator
7	Encrypted Training	Generator	File Handler
8	Encrypted Training	File Handler	Disk
9	Encrypted Training	Disk	File Handler
10	Encrypted Training	File Handler	Trainer
11	Provider Payload	Trainer	File Handler
12	Consumer Payload	Trainer	File Handler
13	Provider Payload	File Handler	PVC-provider
14	Provider Payload	PVC-provider	PVC-consumer
15	Payload	PVC-consumer	PVC File System
16	Payload Approval	PVC-consumer	PVC-provider
17	Payload Approval	PVC-provider	File Handler
18	Consumer Payload	File Handler	PDCo-provider
19	Consumer Payload	PDCo-provider	PDCo-consumer
20	Payload	PDCo-consumer	Disk

4 Results

In this chapter, I will provide results obtained through during testing of each of the proposed training networks. I will discuss these results in the the follow discussion chapter 5. The chapter contains sections titiled ”Model Training”, ”Hardware Utilization”, ”Runtime and Throughput” and ”Security.” They are ordered from least to most significant.

4.1 Validation of Trained Models

Machine learning model design and inspection is outside the scope in this paper, so I will not be addressing it. The quality of the model is not of interest, only that it is reproducible. For the purposes of this research, it is only important that the results stay consistent through each architecture. The security network encasing the training application should have no affect on the training of the model.

I performed two distinct inferences on the trained models to ensure their consistency. I obtained the mean absolute error (MAE) and mean absolute percent error (MAPE) for the yield prediciton over dates between April 1 and July 15. These dates are the dates the data used for training was collected and encompass the grape farming season. The timeseries graphs for MAE and MAPE are contained in figure 4.1.

I also recorded the train and validation loss over each epoch during training. These values are plotted per epoch to reveal the models loss curve. The loss curve is shown in figure 4.2;

The architectures performed as intended. Hence, the trained models resulting from each architecture performed identically apart from the purposefully generated randomness most often present in model training. As the results were nearly identical, I provide a single figure for the timeseries and one more for the loss curve.

4.2 Hardware Utilization

I present the data in this section in two groups. The first group is the utilization during Architectures 1 and 2. The second group contains Architecture 3 and 4. They are grouped based on their use of the IDH. The only noticeable change in hardware use is when the IDH is used for file storage in place of disk. The Memory and GPU consumption in architecture's 3 and 4 decreased, on average, by 70% and the CPU use increased by 20% as a direct result of the IDH. The increased time required to move data to and from the IDH causes the utilization of the shared instances hardware components to vary more then when data is managed on disk

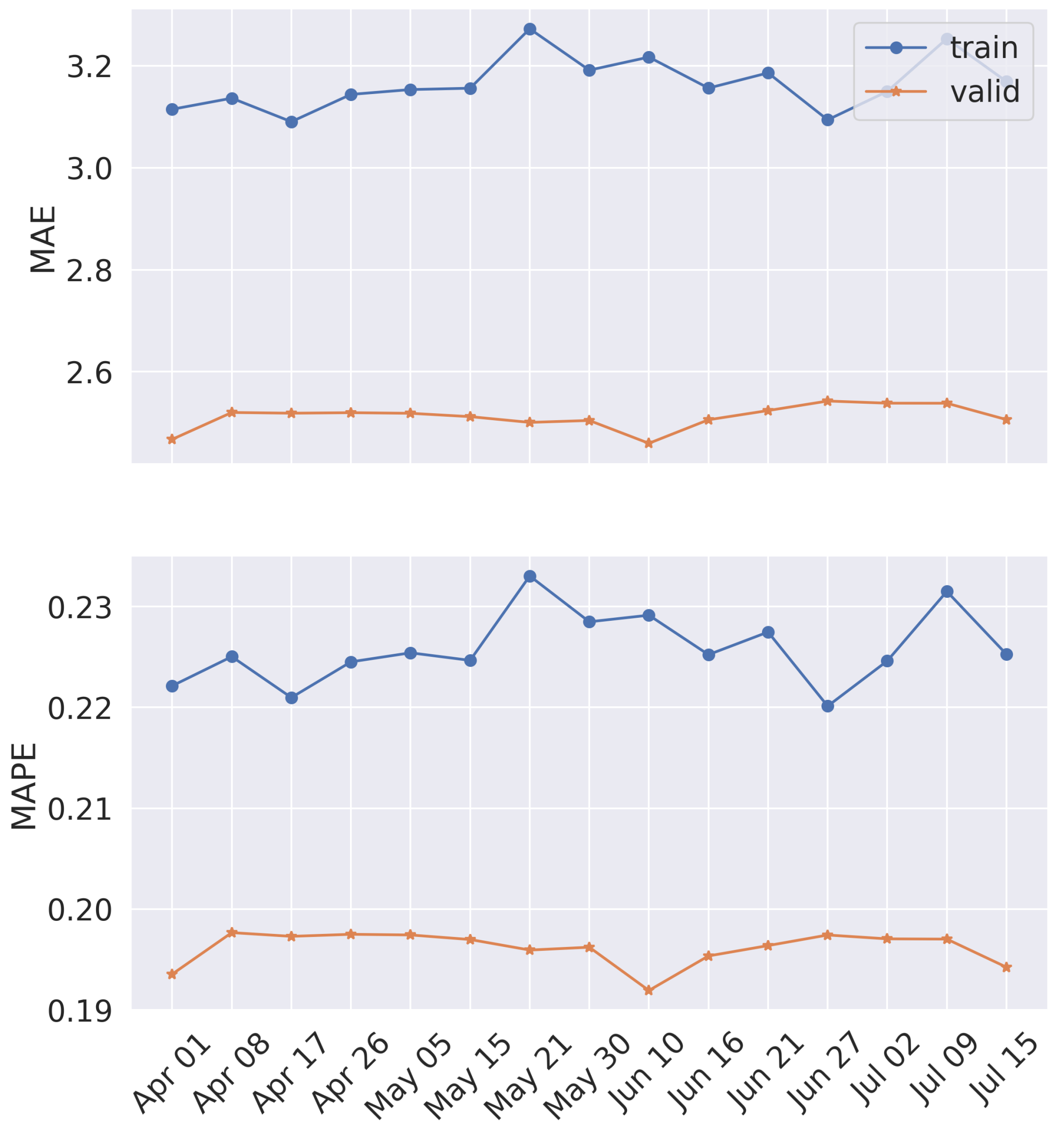


Figure 4.1. ML Model - Timeseries Data

directly.

The instances Memory, CPU and GPU were considered in hardware utilization. Apart from amplitude, the theoretical percent use of each hardware component is expected to follow the trapezoidal curve shown in figure 4.3. The starting point in the time range is the time the architecture was initialized. The ending time is architecture dependent and matches either the time when the payload was written to disk or delivered to the consumer instance. The recorded Memory usage data is zeroed at the value it was at prior to the start of the network. Therefor, the usage percentage for Memory is with respect to the memory available at the start of training.

Figures 4.4 and 4.5 contain plots of the experimental hardware usage percentage over the duration of the training network. The first noticeable difference between each experimental graph is how the curve's in the former graph are much smoother then the latter. This is made more clear by the 4th order moving averages of each curve shown in the bottom of each of the figures. The IDH causes the usage of hardware to be erratic with a much larger deviation in from point to point. The standard deviation and average for the % usage of the hardware component

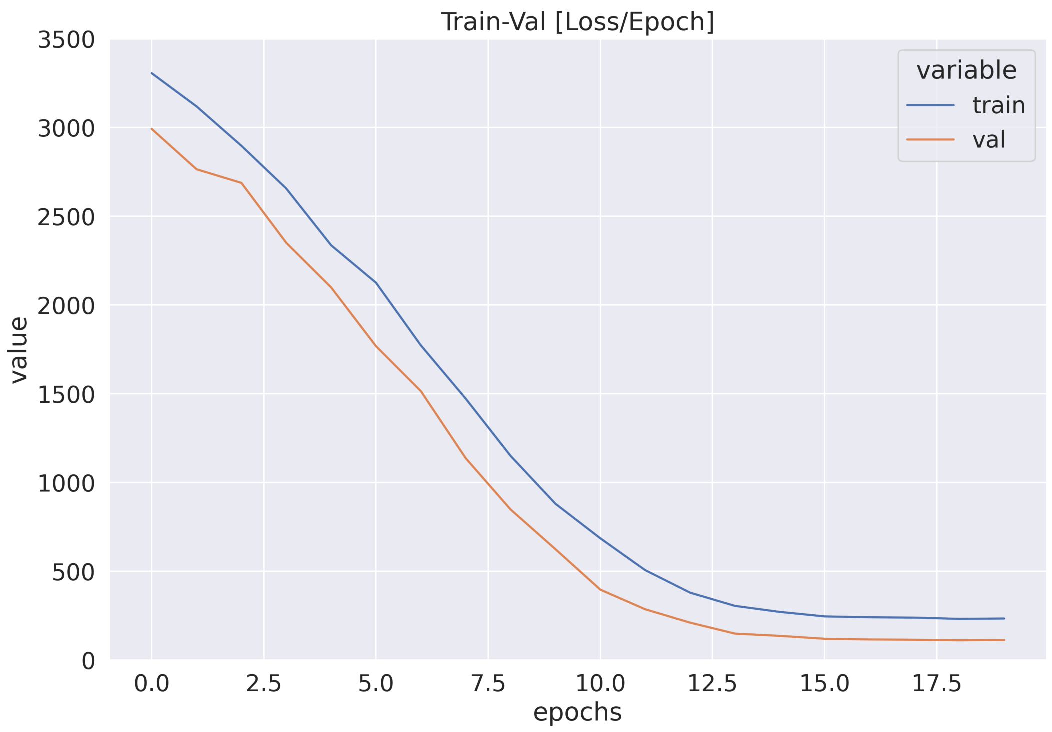


Figure 4.2. ML Model - Loss Curve

curve's in each group can be seen in table 4.1.

Table 4.1. Standard Deviation and Average for % of Hardware Usage

	Standard Deviation			**Average**		
Data Storage	*Disk*	*IDH*	%-Diff	*Disk*	*IDH*	%-Diff
Memory	6.499	11.384	75%	76.96	16.09	-79%
CPU	1.730	2.382	38%	32.63	51.45	58%
GPU	7.092	13.022	84%	94.17	18.87	-80%

4.3 Performance

In this section, I will provide the runtime data of the architectures and throughput calculations for their processes. Runtimes are compared between the core-processes in each architecture. I will compare and contrast the Requester, Generator and Trainer processes first because these processes are contained in all 4 architectures. I will then provided analysis of the IDH using data pulled from the Requester and Trainer while they used the IDH in architecture 3 and 4. Finally, I will provide the results of the EDH's.

I mentioned in the Introduction Chapter of this paper that not much time was spent optimizing the throughput of each process. I want to reiterate this here as this is just an initial implementation. Future work could be spent optimizing to likely lead to much better results especially with respect to the IDH.

I will discuss throughput results of the individual processes, but these comparisons are made

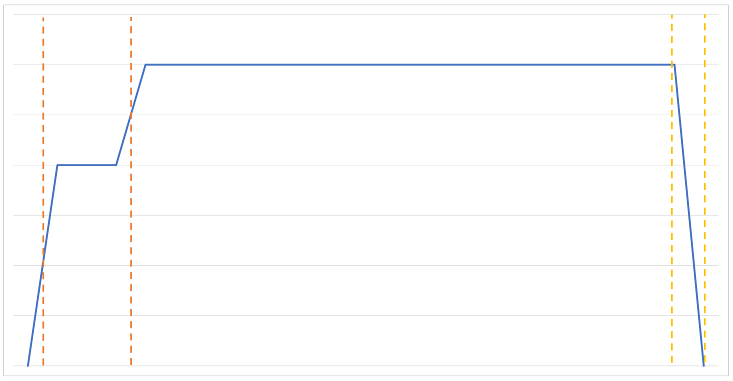


Figure 4.3. The expected graph shape (excluding amplitude) of the Hardware utilization of all hardware components across all architectures.
The area between the two left-most dashed lines is when the Generator is executing. Between the right-most dashed lines is when training is complete so the use quickly falls to zero.

between the same two groupings as in the hardware utilization data results. Architectures 1 and 2 are grouped together as they store and manage data on disk. Architecture 3 and 4 are grouped because they each use the IDH for data storage. The reason for the grouping is the runtimes for processes in each group are the same, as will be shown. The IDH is the single process that has any considerable affect on architecture runtimes as it increases the cycle time and decreases the throughput of the processes that interact with it.

Throughput is measured as data per second. The meaning of data and the throughput overall changes between processes. The exact meaning will be mentioned in the throughput results for a given process. Throughput in displayed in two units: Mb/sec and $Files/sec$. The $Files/sec$ unit is less significant as the files are not all the same size. In fact, files here incorporates both the

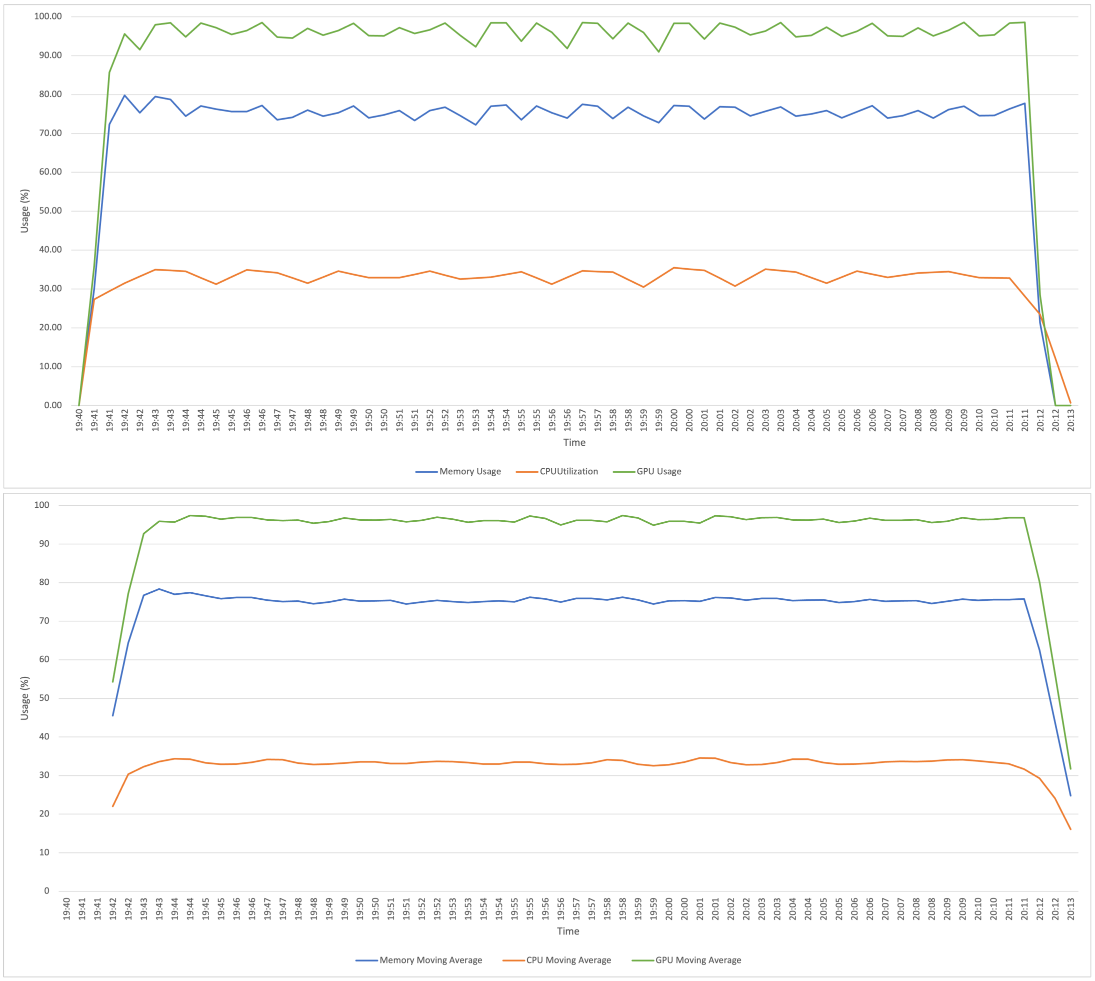


Figure 4.4. Hardware utilization when Data is stored on Disk. (Architecture 1 & 2)
Top: Experimental Values
Bottom: 4th Order Moving Average of Experimental Values

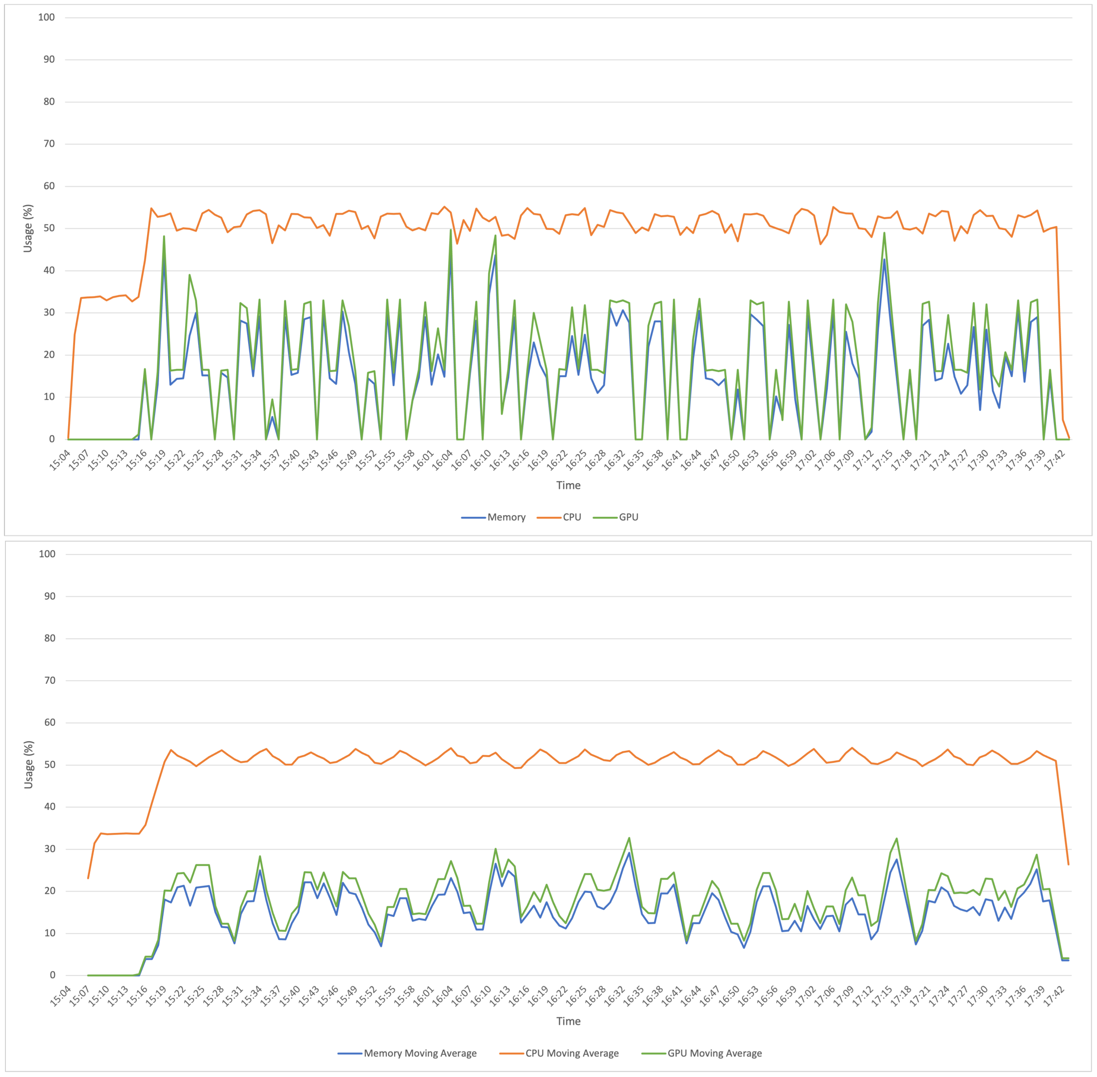


Figure 4.5. Hardware utilization when Data is stored in IDH. (Architecture 3 & 4)
Top: Experimental Values
Bottom: 4th Order Moving Average of Experimental Values

Yield Data Files and Satellite Image Files which are very different in size. There is a standard deviation between the Yield Data file sizes of $826.0Kb$. Satellite Images sizes are much closer in size and have a negligible deviation. However, the Satellite Images are much larger pulling the average file size up significantly making the $Files/sec$ throughput appear smaller then expected. I will still include both values for all throughput results, but consider these details when viewing.

4.3.1 FarmOS Connector

The FosC is used in all processes and remains exactly the same in each. The initialization time of the FosC has no impact on the runtime of the architectures. It is a process which each architecture requires to run, but the FosC does not require the architecture to run. Due to each architectures reliance, it must be spun-up prior to the initialization of any given architecture. Therefor, has no impact on complete runtime of the network with respect to its spin-up time.

The use of the FosC increases the Requester's runtime and decreases its throughput. This is in comparison to the Requester bypassing the connector and pulling data directly from the Provider. See section 4.3.3 on the Requester below for insight into the results of the requester.

Their is a known issue with the consumer_app where files must be re-requested if the transfer fails. Further, when a fail occurs there is hang time before the file is re-requested. If the file is not re-requested, it will hang forever.

Consider a file failing near the end of its transfer. The entire process must rerun for that file making it take 4 times what it takes to transfer that file directly. This does not include the time the provider_core is hanging after a failed transfer. The total runtime affect is more significant for the larger files (satellite images) which were found to fail more often and, in some cases, multiple times in a row.

4.3.2 Architecture Independent Processes

The Requester, Generator and Trainer processes are considered architecture independent because they are used in all 4 architectures. Figure 4.6 shows the runtimes of all processes in a single graph. This graph is to show how the total runtimes of each architecture compares to their components. For a better visual on each distinct process, see figure 4.7. Additionally, figure 4.8 contains the percent difference of the Generator and Trainer runtimes in Architectures 2, 3 and 4 from Architecture 1. All values are shown in table 4.2.

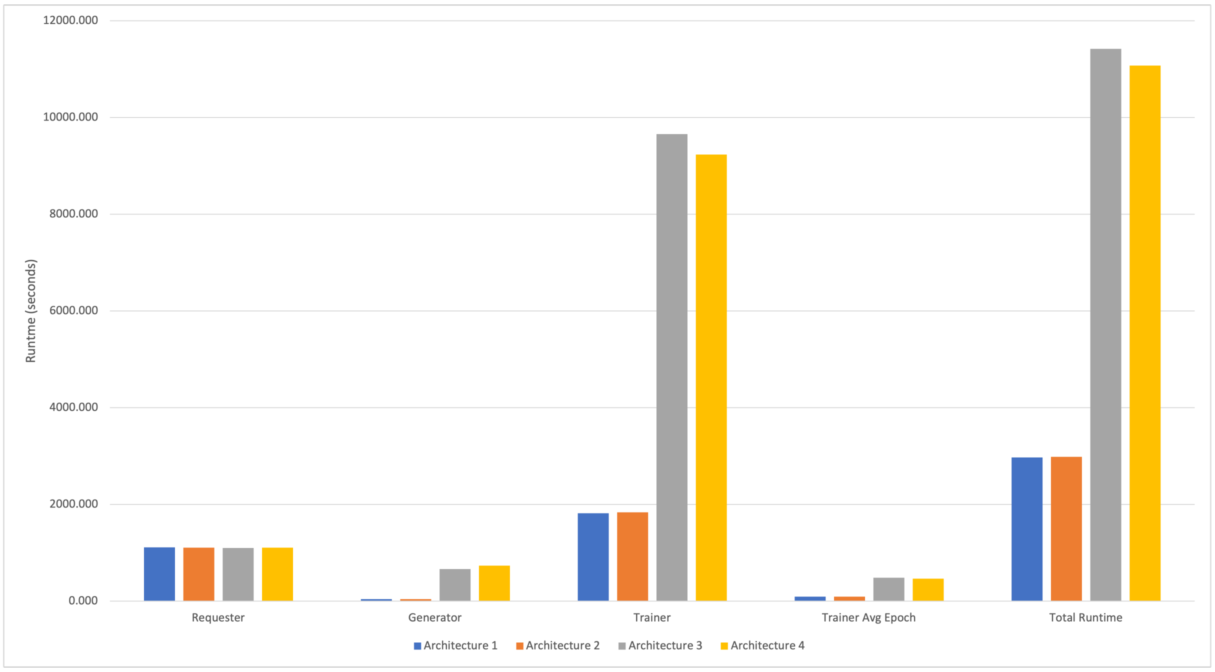


Figure 4.6. Collective Process Runtimes per Architecture

Table 4.2. Collective Process Runtimes per Architecture

	Architecture 1	**Architecture 2**	**Architecture 3**	**Architecture 4**
Requester	1111.118	1104.560	1098.351	1104.735
Generator	42.703	41.558	662.363	734.779
Trainer	1813.557	1836.771	9658.734	9233.215
Avg Epoch	90.390	91.457	482.461	461.226
Total	2961.614	2975.250	11409.934	11064.044

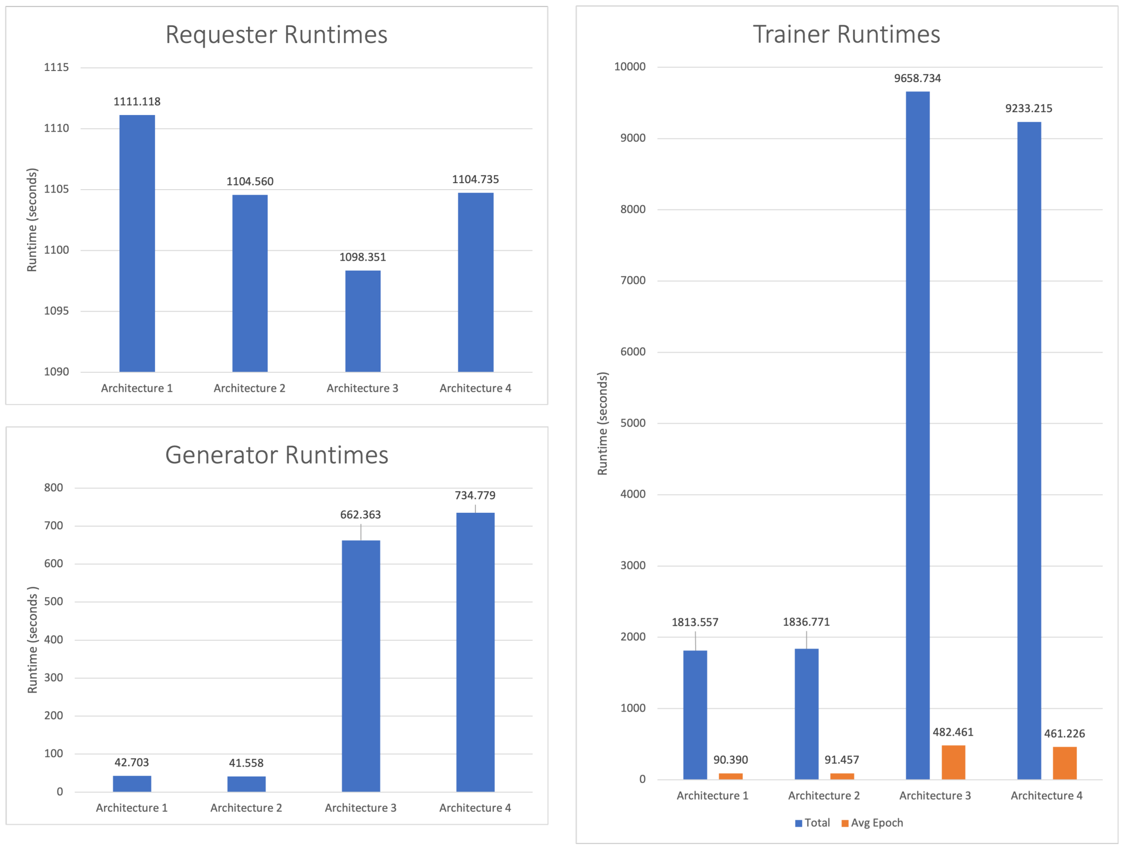
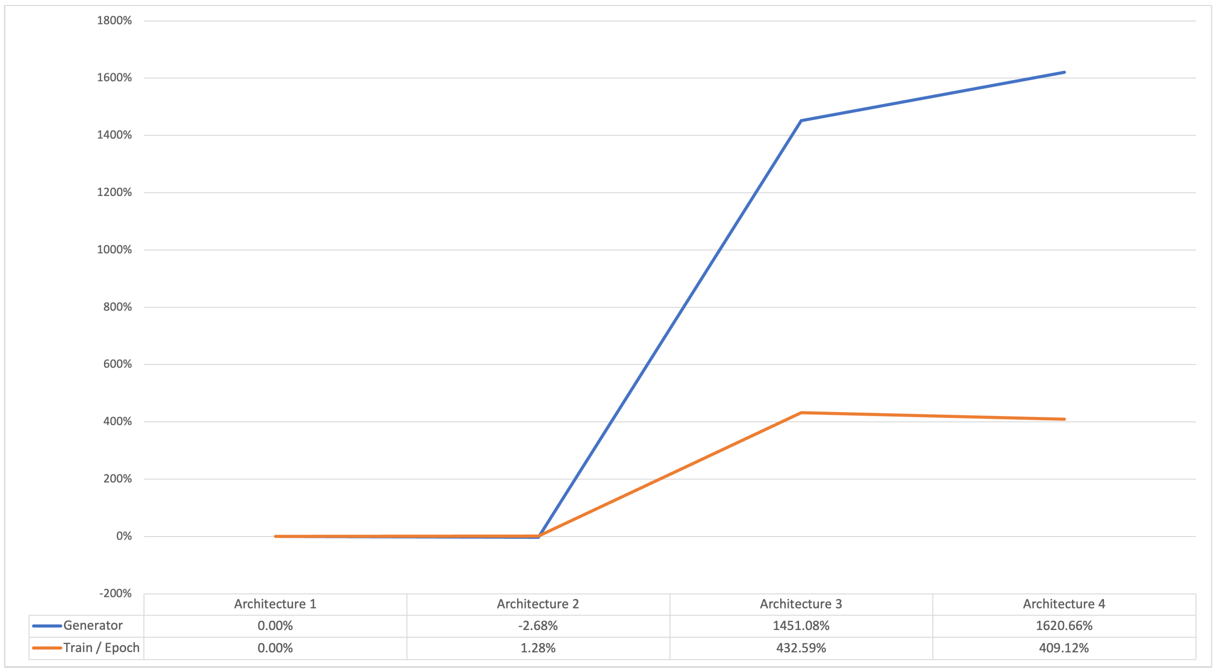


Figure 4.7. Individual Runtimes of Architecture Independent Processes across all Architectures



	Architecture 1	Architecture 2	Architecture 3	Architecture 4
Generator	0.00%	-2.68%	1451.08%	1620.66%
Train / Epoch	0.00%	1.28%	432.59%	409.12%

Figure 4.8. The Percent Change of the Generator and Trainer runtimes from the Baseline Architecture 1 to Architecture 2, 3 and 4

4.3.3 Requester

Values discussed on the Requester in the section can be found in table 4.3.

Runtime The runtime measurements and throughput calculations for the Requester remain consistent through all four architectures. The runtimes stay so consistent that their average of 1104.68 seconds have a standard deviation of just 5.22 seconds. As shown in figure 4.7, the Requester's runtime is consistent regardless of the IDH being used for file storage or the disk. Figure 4.6 shows how close these values are together when comparing them with the differences in the runtimes of other processes. In all processes but this one, there is clearly a substantial increase in total runtime between architectures 2 and 3 when the IDH is Incorporated.

Throughput The Requester's purpose is to retrieve the raw data from a remote source and store it on the shared instance. Therefor, the throughput is the measurement of data quantity pulled from its source and stored on the instance per second. The graph to the right of figure 4.9 contains the calculated throughput values. The Requester stays consistent through all architectures, so I only display a single value averaged across all architectures for each throughput unit.

FosC vs Direct Data The Requester experiences a slowdown from the use of the FosC as was mentioned in subsection 4.3.1 above. Figure 4.9 contains the averages of these two runtimes graphed and table 4.3 contains the exact values. The Requester reveals a runtime increase of approximately 305% when data is pulled through the FosC rather then directly from the Provider. Further, the throughput decreases by approximately 75% when pulling data through the FosC.

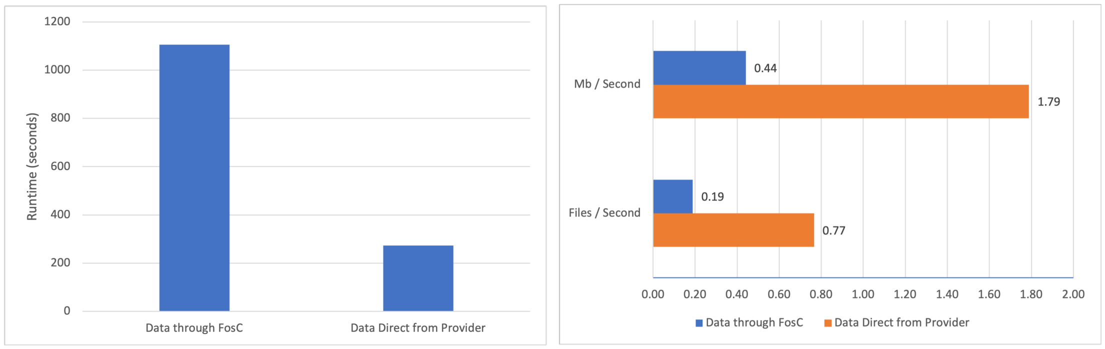


Figure 4.9. Requester when Pulling Data through the FosC and from the Provider Directly.
Left: Runtimes
Right: Throughput

Trusted Connectors use Apache Camel for data transfer and routing which is expected to be two times slower. The core causation's to the Requester taking four time longer when using the FosC is a problem with the applications representing the consumer_app and provider_app. See section 4.3.1 above for information on this issue.

Table 4.3. Requester Results Data for Runtime Measurements & Throughput Calculations

	Data through FosC	**Data Direct from Provider**
Time to Pull all Files (s)	1104.68	272.66
Runtime %-diff	305.2%	-75.3%
Raw Data Files Pulled	209.00	209.00
Data Pulled (Mb)	487.64	487.64
Files / Second	0.19	0.77
Mb / Second	0.44	1.79
Throughput %-diff	-75.3%	305.2%

4.3.4 Generator

The values contained in this section related to throughput of the Generator can be found in tables 4.4 and 4.5. The throughput calculations in table 4.4 rely on the values in table 4.5.

Runtime The generator shows the largest increase in runtime in architecture 3. The runtime increases by approximately 1451% in architecture 3 compared to architecture 1 and 2. This is shown in figure 4.8. The IDH is the cause of the increase and I will address why the change is so drastic in section 4.3.6.

See the "Generator Runtimes" graph in figure 4.7 to see the average runtimes for each architecture. The Generator's runtime is expected to match between architectures 1 and 2 as well as between architecture 3 and 4. The only variation in this is primarily a direct result of the IDH with some error coming from unintended noise.

Throughput The Generator uses the raw data to generate the training labels and images. Hence, the throughput of the Generator is measured as data generated per second. The generators throughput results are shown in figure 4.10.

For architectures with data stored on disk, the throughput includes the time required to read the raw data off disk and write the generated data to disk.

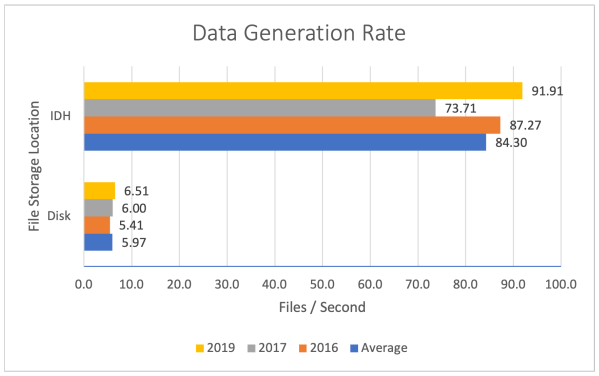
For architectures that use the IDH for data storage, the throughput includes the time to request the data from the IDH and to deliver the generated data to the IDH. Delivering the data to the IDH does not include the time it takes for the IDH to encrypt and write the data to disk. It is only the time required for the IDH to receive the generated bytes. Requesting data from the IDH, however, includes the time it takes for the IDH to decrypt the data on disk then deliver it. The time added in awaiting the raw data file decryption is the bottleneck of the generator when using the IDH.

Table 4.4. Generator Throughput Results Data

	Data Stored on Disk			Data Stored in IDH		
Raw Data Year	*2016*	*2017*	*2019*	*2016*	*2017*	*2019*
Runtime (s)	14.97	19.62	12.80	241.50	241.06	180.68
Throughput (files/s)	8.02	6.12	6.87	0.497	0.498	0.487
Avg Throughput (files/s)		**7.00**			**0.494**	
Throughput (Mb/s)	87.27	73.71	91.91	5.41	6.00	6.51
Avg Throughput (Mb/s)		**84.30**			**5.97**	
Throughput %-Diff		**1311.05%**			**-92.91%**	

4.3.5 Trainer

I will provide results for the runtime of the Trainer, but I will not do any throughput analysis. I did not make the training application which is the core of the Trainer process and is the source of all changes in throughput. Therefor, throughput of the Trainer is outside the scope.



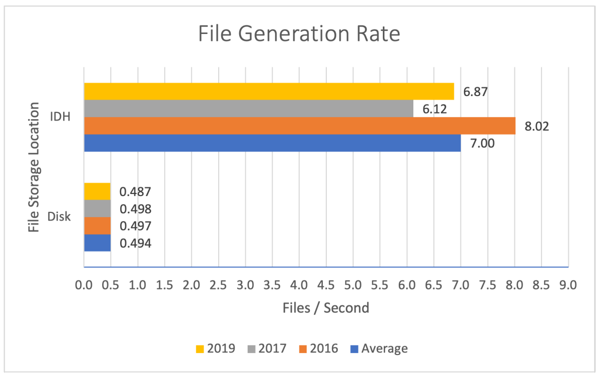


Figure 4.10. Generator Throughput for the Three Years of Raw Data Used in Creating Training Images and Labels
Left: The Data Generation Rate
Right: The File Generation Rate

Table 4.5. Training Data Created by the Generator

	Labels	*Images*	*Total Files*	*Data Size (Mb)*
2016	60	60	120	1306.26
2017	60	60	120	1446.29
2019	44	44	88	1176.82
Total	164	164	328	*3929.37*

Runtime There are two values for the runtime of the Trainer: total runtime and epoch runtime.

The training is performed over 20 epochs. The epoch runtime is the average runtime of all 20 epochs in a given run. The average epoch time for each run were then averaged over all test runs for each architecture to obtain the final value. The total Trainer runtime is just slightly higher (8.17 seconds on average) then the total time for all 20 epochs to complete. This 8.17 seconds is essentially just the time it takes to initialize the training app. The runtime values are shown in the *Trainer* row of table 4.2.

Like the Generator, the runtime for the Trainer stays relatively consistent between the architectures storing data on disk and architectures using the IDH. The greater runtime for the trainer in architectures 3 and 4 is a fault of the IDH transfer speeds. See section 4.3.6 for more detail. The values between the two sets of using the IDH or not should theoretically be the same. The visible variation that can be seen in figures 4.6 and 4.7 comes from random noise resulting in inconsistent data transfer rates.

4.3.6 Internal Data Handler

Mentioned above a number of times is the considerable affect the IDH has on runtimes. The IDH is the bottleneck in architectures 3 and 4. Specifically, the IDH increases runtime and decreases throughput in the Generator and Trainer processes, drastically.

While the IDH is the bottleneck, it is also a necessity. The IDH solves many of the security issues that were present in Architecture 2. The IDH's security impacts will be discussed further in section 4.4 below.

Runtime The IDH does not have a runtime as other processes due. The runtime of the IDH is just the runtime of the entire network which is not informative.

Throughput The IDH is a local server for storing data. Receiving data from a client then adding it to the store and sending requested data from the store to a client are the main two procedures performed by the IDH. These are referred to as Receiving and Sending, respectively. The throughput of sending/receiving is measured as data transferred out/in of the IDH per second. Their is no mention of architectures in these results as the IDH throughput is independent of the architectures.

The Send and Receive procedures each contain a set of core steps directly related to their throughput. These are broken down in detail in section 3.10.1.1, but I'll include simplified versions here to assistance in the inference.

- "Receive" Steps
  1. Receive Data Push Request from Client
  2. Receive File Data from Client
  3. Close Connection with Client
  4. Encrypt File Data
  5. Write Encrypted File Data to Disk
- "Send" Steps
  1. Receive Data Pull Request from Client
  2. Read File from Disk
  3. Decrypt File Data
  4. Send Decrypted File Data to Client
  5. Close Connection with Client

Notice how the client connection in "Receive" is closed in step 3. Moreover, the client connection does not have to persist while data is encrypted and stored. In the "Send" steps, however, the client connection persists through all steps including decrypting the data. This is required as the data does not live unenecrypted, so the client must wait for the data to be decrypted first. As a result, the Receive rate is considerably faster than the Send rate. These rates can be scene in the graphs contained in figure 4.11.

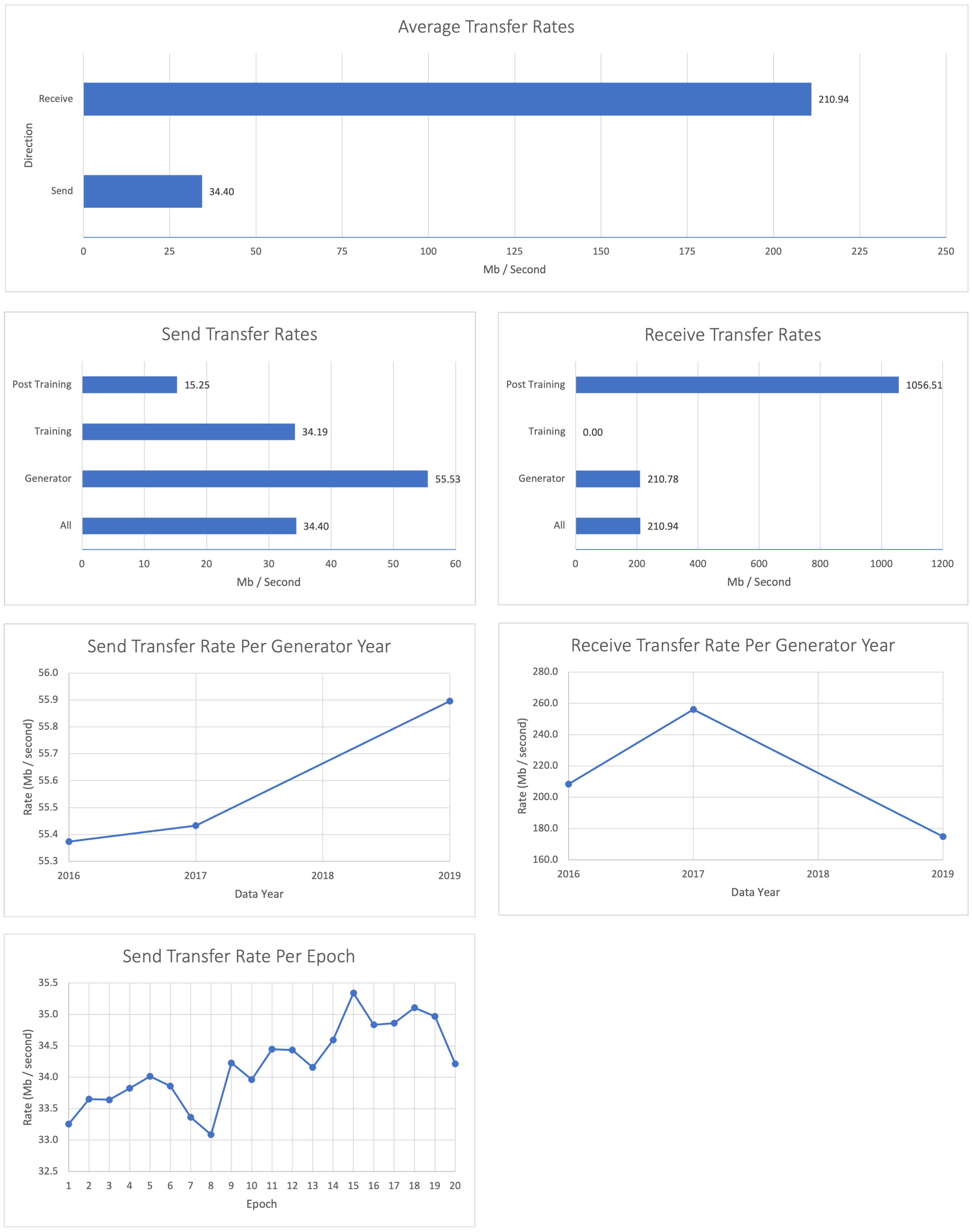


Figure 4.11. Send and Receive Transfer Rates for the IDH

The transfer rates are determined across 3 distinct stages in the network as listed in figure 4.11. These are the Generator, Training, and Post Training stages. No data is sent to the IDH during the training phase, hence the value of zero in the "Receive Transfer Rates" graph. [1]

The rates in the Post Training stage are skewed in comparison to the other stages because a very small amount of data is transfer over a small amount of time. Furthermore, the Post Training occurs over less then a second, so it is unable to reach a steady state. This is clear in the minute affect the rates occurring in this stage have on the averages. See the "Send Transfer Rates" and "Receive Transfer Rates" graphs in figure 4.11 for visuals of this.

Use table 4.6 to understand the impact each stage has on the average transfer rate for either Sending or Receiving data. The quantity of data for a given stage and transfer direction has the biggest impact because it is scales with the amount of time spent making that request for that stage.

Send Transfer Rate

Generator This rate is much greater then the average because this process is synchronous meaning the IDH only sends a single data object at a time unless during training in which five threads are sending requests simultaneously. It also takes time for the generator to generate the training label and image after it requests raw data, so the IDH is not receiving a flood of constant requests like occurs during training.

Training This is nearly the average as the majority of the data requests are sent to to the IDH during this stage for training image and labels. Its value is lower then the generator because the training process uses 5 asynchronous workers which are constantly requesting large streams of data. There is also a much smaller break between requests because the time between requests is much less than during the generator.

Post Training The only file sent in the Post Training stage is the Payload.

Receive Transfer Rate

Generator This transfer rate is nearly equal to the average, but is significantly lower then the send rate during Post Training phase. It is lower because the generator sends a much larger quantity of data during Image and Label generation to the IDH. There are also simultaneous data requests being sent by the Generator to the IDH after it sends the generated data objects.

[1]The zero was not considered when determining the average Receive Transfer Rate.

Table 4.6. Send and Receive Requests from and to, respectively, the IDH

	IDH Send Data		**IDH Receive Data**	
	Request Count	*Data Sent (Mb)*	*Request Count*	*Data Received (Mb)*
Generator	2624	15705	328	3929.37
Training	81364	963993	0	0
Post Training	1	7.45	7	3.70

Moreover, other resources are being used by the IDH to decrypt and send data back to the Generator while it is trying to receive the new data objects, encrypt them, and write them to disk.

Post Training The 7 files received in the Post Training stage make up the payload. It is four random vectors generated for this training run, two csv files containing the validation and training loss values and the trained model state dictionary itself.

4.3.7 External Data Handlers

The results for both EDH's are good. Their affect on the runtime is completely negligible considering they collectively make up less then 0.5% of architecture 4's total runtime. However, as aforementioned in section 3.11, the time it takes for the Provider to review the payload and validate it is completely neglected. It is impossible to know how long the provider may take and, therefor, is outside the architectures control and not considered in its evaluation.

4.4 Security

In this section I will address securities threats and how they were addressed. The solutions to security risks may be changes to an architectures docker network, docker container properties, applied techniques and/or tools (e.g. obfuscation), sub-processes incorporated into existing core-processes, or completely new core-processes. I will refer to the threat model in table 3.1 frequently.

This subsections below are titled by the security objects listed in the threat model. They are ordered numerically by the numbers at the top of each column in the threat model table. For each I will mention the risks and precisely the object or technique that solved the risk. In chapter 5 (Discussion), I will review which architectures address each security objective. I will also cover Architecture 5 in Chapter 5 (Discussion) as it is theoretical and their are no test results from it. However, I will present which additional objectives would be solved by it.

4.4.1 Security for Provider's Data in Transit

Before training in the network can even begin, the data must first be able to travel securely from the Provider's instance to the shared instance. Hence, the guaranteed security of the Provider's raw data while it is in transit from the Provider's personal instance to the shared instance is the first required security objective.

Risks Data can be intercepted in a variety of ways. There is address spoofing to imitate the original destination of the data and network sniffing to access packets as their transferred among others.

Solution The purpose of the FosC was to solve this security risk. The FosC ensures the data is going to the correct application and validates the source of the request through an attestation handshake.

4.4.2 All Training Related Data on Shared Instance Secure

All data actively on the shared instance must be secure and inaccessible to non-admin users. This includes the Provider's raw data, data generated by the Generator and the training results. Training results should only be accessible off the shared instance.

Risks Large high density data files must be stored on disk. If the data is written to disk in its raw form it may be viewed and potentially stolen by other users on the shared instance.

Solution The IDH was designed to address this risk. The IDH encrypts all data before it is written to disk. When encrypted, all the data written to disk is rendered unreadable by anyone on the instance. To enforce the use of the IDH, docker properties were leveraged to set all container, besides the IDH, to read-only file system and read-only shared volumes. Read-only for both ensures the contained application cannot write anything to disk.

4.4.3 Only Training Results may leave the Instance

This objective is a continuity of the previous security one. The network must ensure the only data that may ever leave the shared instance disk is the training results. All raw data and training data will be on the shared instance during the duration of the training network and self delete at its completion. The raw data specifically may stay on the instance in the event of multiple consumers sharing the instance with the Provider.

Risks This threat has the largest number of related risks because there are possible sources of risk both internal and external to the network.

The training results could be intercepted while there are being exported from the shared instance to either the Provider's or Consumer's. This is essentially same threat as from the first security objective in section 4.4.1. The only difference being that this deals with the results payload leaving the shared instance rather then data coming into the instance. Moreover, the data could be intercepted during remote transfer and the data could be transferred to a different source that is spoofing their address to imitate the remote instance.

The processes developed by and coming from the Consumer could be themselves malicious. Especially the Trainer which is obfuscated. The Consumer's Generator and Trainer both have the ability to export data to a remote source.

The Trainer poses an additional substantial hazard in saving data as "Training Results" to eventually be sent to the Consumer. The Trainer could simply mark the Provider's secure data as "Training Results" and send them to the IDH.

Solution The PDH and CDH use attestation to validate the request sources. The data is also encrypted end-to-end through each EDH and client to defend against interception.

The only containers given network access are the FosC, PDH and CDH. Furthermore, both Consumer applications (Trainer, Generator) have no internet capabilities. They can only network with internal Containers.

The PDH was deployed in architecture 4 so the Provider could first evaluate the Payload before it was delivered to the Consumer. This was the only feasible solution in the scope of presented research.

4.4.4 Consumer's Code is Secure

This Architecture is based in not only security for the Provider and their data, but also security for the Consumer's proprietary code. The Consumer's code should only be executable with the consent of the Consumer. The Provider should not have the ability to read the Consumer's code even as an admin of the instance.

The Consumer's proprietary code primarily refers to the Trainer. However, the Consumer may wish for security of their Generator in certain situations.

Risks The Provider could pirate the Consumer's code from the shared Docker Images. Provider may also keep the image to run later without the Consumer's approval.

Solution The Consumer should obfuscate their application(s) prior to containerizing them. Obfuscated code can be executed but is unreadable and nearly impossible to reverse engineer.

To address the risk of the Provider's capability to store and execute the Consumer's image(s), the CDH's attestation process was altered and connected to the Trainer. Unlike the handshake process of the PDH which occurs only between the PDH and PDC, the CDH only facilitates a handshake between the Trainer and CDC. See section 3.11.1.4 for greater detail on this process. The Trainer cannot execute until its handshake with the CDC passes.

4.4.5 No Undefined Communication between Containers

Container's should not be able to perform unspecified communication with other containers in the architecture.

Risks The docker.sock is used by the Dockerd to communicate with all containers on the machine. Container's with access to this socket are able to communicate with all other containers on the machine as an admin.

Container's on a given docker network are able to communicate with all other containers on that network. This is especially a risk if all containers are connected to a default network like the "Host" or "Bridge".

Solution The only container's with access to the docker.sock are the Provider and Consumer ends of the FosC. These containers are both owned by the Provider, so the only potential risk is from external malicious sources. However, Trusted Connectors have extensive security features to protect from external threats [25].

Many custom docker networks were deployed so containerized processes can only communication with required containers (e.g. Trainer to IDH). The web of docker networks in Architecture 4 can be viewed in figure 3.33.

4.4.6 Security from Consumer's Code

The training network must be able to prevent any threat presented by the Consumer's code. The Consumer must provide the Trainer and Generator to the Provider. Therefor, the architecture must be able to protect itself, the data and the compute instance from the Consumer's containerised app's.

Risks The Consumer's code is obfuscated, so it is impossible for the Provider to know if the application's are what the Consumer states it is. Further, the Trainer and/or Generator could be malware or something else with a malicious purpose.

Solution These risks have already been solved in the solutions to prior security objectives. The Consumer's containers cannot pose any hazard's. Their file-systems are set to read-only and the volumes they have access to are read-only. The use of custom docker networks means they cannot present any threats to other containers or the host machine.

4.4.7 Containers cannot be Modified after Starting

Container's must not be alterable after they start. This is to prevent the Consumer's code from generating new scripts (or similar) inside their Container's that may pose a threat.

Risks Consumer could alter container code or inject new code into a running container from their containerized apps.

Solution The file-systems for containers are read-only and any volumes they may share are also read only. These have been mentioned as the answers to prior objectives as well, hence this objective is already upheld.

5 Discussion

In this chapter, I will discuss the Approach and Results chapters including reasoning's and decisions made in the research. More, I will report on both the pros and cons of key facets throughout the paper.

My research was centered around addressing security and less around throughput of the processes. I found it more important to ensure a final architecture that was completely secure with respect to the threat model objectives. With a known level of security reached, future effort could be dedicated to improving the architectures speed. Hence, in this chapter I propose many features of the architecture that could see throughput improvements, among other enhancements, through further exploration.

If I do not mention a specific architecture in this chapter, then I am talking about architecture 4 as it is the most secure of the architectures.

5.1 Comparing the Architectures

I presented some of the results in two groups; one group included architectures 1 and 2 and was defined by there use of the disk for direct and unencrypted data storage. The other group contained architectures 3 and 4 which both used the IDH for secure encrypted data storage. The results showed essentially the same runtime and throughput for processes in the same group, but extensive differences between groups. All throughput decreases came almost entirely from the use of the IDH in place of the disk.

I demonstrated four architectures to show a range of improvements in security while concurrently expecting a decreasing range in throughput. I hoped this would enable a choice of which architecture to deploy to practice based on security and runtime requirements. Conversely, the results proved that it is useless to ever deploy architectures 1 or 3 because their paired architectures, 2 and 4 respectively, offer then same throughput while improving security.

I expected architecture 1 to be replaced by architecture 2, but was surprised to find architecture 3 made obsolete by 4. Moreover, my surprise in the replacement of architecture 3 was grounded in expecting the final security objectives to require more then just the EDH's.

5.2 Core Processes

In this section I discuss the core processes developed for these architectures.

5.2.1 FarmOS Connector

Known Issue Causing Four Times the Expected Runtime A solution to this exact problem was not found. However, future efforts should use a new approach to the transfer of data over the connector. A better technique would be for the provider_app to deploy all data collectively in a single package rather then relying on requests to be passed back and fourth through the connector for files. All the data to be used would be known prior to the deployment of the network anyway, so all data could be sent together. This would alleviate the issue of individual files failing.

IDSCP2 Protocol Policies The "Connector-Restricted" policy was not an option because the data must be transferred outside of the connector for use in the Batch Processing Connector. Moreover, the data was stored on disk outside of the connector after being pulled, so the "Use Data and Delete it After" policy was impossible to enforce.

The "Local Logging" and "Remote Notification" data usage policies were enough for the purpose of these networks. It would also have been possible to assign the "Duration-Restricted" or "Interval-Restricted" policies. However, both ends of the connector are on instances owned by the Provider, so duration and time of usage are not a security concern. The "Interval-Restricted" policy could have a use in architectures 3 and 4 given that the Provider must available to approve the results. However, the FarmOS Connector is only used during the first stage in the network. It is no longer needed after all the raw data is pulled and stored locally.

The "Restricted Number of Uses" policy could have been applied with two separate restriction counts. A restriction of $n_(logs) = 1$ for the log object because it only needs to be parsed a single time and $n_{files} = TotalFilesToBePulled$ for the file requests. However, it was deemed unnecessary and at no benefit to security. The data endpoint was only accessibly by a single process (FarmOS Data Requester) in all of the networks which has its own security features. The FarmOS Data Requester's security features were enough to where limiting the number of data uses and pulls in the connector would needlessly increase complexity without increasing security.

5.2.2 IDH

All cryptographic processing the IDH employs is handled by the CPU. However, this could be improved by moving the encryption calculations to the GPU. The GPU handles parallel processing more efficiently then the CPU and has been shown to improve AES and SHA algorithm (among others) operations by up to 2.5x [31] [32].

The GPU was not employed in the IDH for cryptography initially because I did not expect the GPU usage to decrease by such a large amount. Prior to the IDH incorporation, GPU use 100% for the entire training duration. I anticipated a slight decrease of roughly 3-6% which would not have supported the computation without slowing image processing for training.

The determined decrease in GPU use of 70% means utilizing it for the IDH encryption would have actually improved training times. If these algorithms were processed 2.5x faster by the GPU, the total training time may have decreased, in suit, by over half.

Additionally, other algorithms could be explored besides AES and SHA. These are two of the most commonly used encryption techniques for this application, but does not necessarily mean they are the best or most optimal.

5.2.3 EDH

FarmStack Trusted Connectors could have been deployed in place of the PDH and CDH and their respective clients for secure data transfer and attestation. Specifically, two bi-directional trusted connectors as each EDH transfers data to and from their client.

The PDH and CDH would both require complex connectors because their endpoint on the shared instance would need to broadcast a server for other internal processes to communicate with them. They would likely be very similar to the FosC which broadcasts for the Requester and transfers data bi-bidirectionally with both the Provider's instance and Requester. The PDH would need to broadcast only for communication with the IDH. The CDH would be slightly more intricate as it would need to broadcast for both the IDH and Trainer.

Using Connectors here in place of the EDH would improve security as Trusted Connectors are hardened with greater security benefits then the two EDH processes. Moreover, the PDH and CDH were developed quickly to perform only what was necessary to meet security objectives. FarmStack Trusted Connectors, on the other hand, have been optimized for the secure peer-to-peer transfer of data. Further, they use the IDSCP2 protocol which was designed with security in mind and enable policies to further improve security guarantees.

The replacement of the EDH's with connectors was also discussed in section 3.12.1 in the approach chapter for the theoretical architecture 5. The PVC would match the exact requirements of what I discuss here as a replacement for the PDH and CDH. The PDCo, however, would differ slightly in architecture 4 from its architecture 5 requirements in that it would still need to facilitate the handshake with the Trainer.

5.3 Hardware Utilization

The IDH drastically increased architecture time and decreased process throughput. However, the IDH also had the unexpected outcome of decreasing hardware utilization sufficiently to enable multiple consumers to train on the same shared instance simultaneously.

The decrease in GPU consumption is a result of the additional time required to pull images from the IDH compared to reading them from the disk. The processes spend more time waiting for the IDH to decrypt and transfer files then they spend processing data. Hence, the GPU is not being used to its fullest potential. The decrease in Memory consumption is a result of the same data transfer time requirements of the IDH.

The increase in CPU usage comes from multi-threading the IDH and the extensive cryptographic calculations performed on the CPU. As discussed above, the architecture could utilize the GPU for this processing therefor lowering the load on the CPU and increasing the GPU and Memory loads.

5.4 Security

5.4.1 Checking the Payload for Secure Data

Ensuring the training results payload does not include the Provider's data is a difficult problem. This issue is where the tipping point between better security for the Provider versus better for the Consumer exists. The best solution would be an automated process to compare payload contents with all secure image data. However, this requires research of its own and is beyond the scope of this paper. However, it is an active research topic with similarities seen in verifying false images [33] and duplicate data [34]. See section 5.5.1 below for more information.

Instead, I was forced to use the method of manual checking by the Provider as the means for validation. This is non-ideal, but, nevertheless, provides that final layer of security.

5.4.2 The Architectures that Ensure the Security Objectives

Architecture 4 solves all risks to each security objective. It is the most secure architecture and should be used if absolute security is required.

Security for Provider's Data in Transit Architectures 1, 2, 3 and 4 address this objective as they all use the Fosc.

All Training Related Data on Shared Instance Secure The IDH is used in both architecture 3 and 4, but architecture 4 is the only architecture that addresses the complete risk. Architecture 3 is at risk as it writes the training payload directly to disk. What ensured the security of architecture 4 was the additions of the PDH and CDH and their respective clients. Architecture 1 and 2 write all their data to disk unencrypted so clearly these are severely at risk.

Only Training Results may leave the Instance Architecture 4 is the only architecture with EDH's and, therefor, is the only architecture to address this objective.

Consumer's Code is Secure All four architectures may use containers with obfuscated code, so they all partially solve this objective. However, security for the payload while it is exported to remote instances is accomplished by the EDH's and meaning only in Architecture 4. Moreover, only architecture 4 stops the Provider from using the Consumer's outside of the network.

No Undefined Communication between Containers The only architecture lacking this security objective is architecture 1. No custom docker networks are used in architecture 1, so containers can communicate freely. Instead of custom docker networks, the containers each share the host network which also allows them to communicate freely with external sources.

Security from Consumer's Code & Containers cannot be Modified after Starting These two objectives are listed together here because a single solution solves the risks of both.

Architectures 3 and 4 both uphold these security objectives. Both architectures use the IDH which is given write privilege to a shared volume. However, the IDH is deployed by the Provider, therefor, posing no threat. Architectures 1 and 2 do not address these objectives as the Trainer and Generator are given shared volume read and write privilege.

5.5 The Future of this Research

5.5.1 Automated Payload Validation

A substantial improvement to the training network would be a tool for efficiently examining the training results payload for the Providers data. This would replace the requirement for the Provider to manually validate the results. This would improve security for the Consumer because the Provider would never need to see the results. It would also render the entire architecture as an black box with end-to-end training automation and complete security. Related research in this area includes image copy move forgery detection [33] and duplicate data elimination [34].

While these topics are not ideal for the intended purpose of this research, they are similar and their proposed techniques may potentially be expanded for this use case.

5.5.2 Architecture Changes

Merged FosC and Requester Merging the Requester in the FosC would reduce network complexity and throughput. The consumer_core application in the FosC is a broadcasting server so that the Requester can pull the Provider's data from the FarmOS database. The Requester's purpose was to pull all the Provider's data and store it. The consumer_core application could be replaced by the Requester to submit data requests directly over the connector. Merging the Requester in to the FosC would decrease the requests per data file by 2.

Merging these two processes isn't essential as the lengthy Requester runtime is largely due to the high quantity of data being transferred remotely and less due to the number of internal requests. Nevertheless, this may be a worthwhile improvement in future efforts.

Generator before Raw Data Storage This architecture alteration involves moving the Generator so it receives the raw data directly from the Requester. The raw data would never be stored on disk. Instead, the Generator would generate the training and label data then send only the generated training data to storage. This change would increase the throughput of the Generator and decrease overall network runtime. The Generator would no longer need to pull data from the IDH and wait for it to be decrypted. However, it would dramatically elevate the number of requests going through the connector if multiple Consumers were training either simultaneously or consecutively because the raw data would not be stored on disk.

Alternatively, if multiple Consumers were to use the architecture, the Requester could forward the raw data to both the Generator and storage in parallel. This would ultimately improve the Generator throughput and lead to no additional requests over the connector. However, only the first Generator to receive the data would see improvements and all others would resort back to pulling the data from the IDH as is currently done.

An additional caveat is the Generator must be designed to generate training data as it receives raw data from the requester. Moreover, the Generator must be able to generate data without knowing the order or the Provider would need to supply the order in which the data will be received over the connector. This is the reason it was not implemented in this way in test for this paper. The Generator was not developed to accept and generate training data in this order and I wanted to keep the Trainer and Generator as close how I received them as possible.

The runtime impact of the Generator in my test scenario was minute in comparison to the Trainer and roughly half of the Requester when using the IDH. Given this and the other mentioned issues, this change should not be considered until others potential improvements have been addressed.

5.5.3 Architecture 5

I have mentioned this architecture throughout this paper as purely theoretical because it is not possible to use enclaves, this architectures core requirement, in model training that requires a GPU. See section 3.12, the final section in the Approach chapter, for extensive insight into the architecture. Enclaves are not currently equipped to interact with GPU's. For this reason, it is considered a future work effort.

If enclaves eventually gain GPU capabilities, I believe this would be the most secure of all the architectures for both the Provider and Consumer. These security improvements would benefit both the Provider and Consumer. The Consumer's security is improved because the enclaves are made from the Generator and Trainer. Therefor, the Consumer could package their applications into Enclaves before delivering them to the Provider rendering them completely inaccessible to the Provider. As long as the original Docker Image the enclave is created from is private, the Provider cannot reverse engineer the processes the enclave contains. The shared instance and all other processes deployed by the Provider would be void of harm from the Consumers applications. Enclaves can only interact with the machine through a single socket and can only process data provided directly over that socket. Enclaves are completely isolated from their host instances apart from this meaning they cannot pull data from the host at all except for what is provided.

Architecture 5's throughput, I imagine, would land somewhere around or slightly greater then architectures 3 and 4. I believe the throughput would be near the IDH architectures because the same amount of encryption and decryption would occur inside the shared instance. The difference being that the data would be encrypted inside the enclaves. The Trainer and Generator become the enclaves, so they would have to encrypt the data. However, slightly less encryption would occur inside the architecture because the data would come from the Provider encrypted rather then the IDH encrypting it.

Finally, architecture 5 is the only architecture capable of securing data from the Honest-but-Curious data provider (AWS in this case). This is only possible because the data in architecture

5 is never outside of an enclave unencrypted on the shared instance.

6 Conclusion

In this paper I illustrated four network architectures for secure model training and the fleet of technologies and techniques, both new and old, utilized in developing them. I also exposed a fifth architecture which has strong potential but is currently an impossible feet due to technological inadequacies [28].

The primary achievements within all that I have researched, developed and presented are the following. I developed a network to assure complete security of agricultural data for use in ML Model Training. Moreover, the network secured the model's code with the only caveat being the data owner must see the results. I paired the architectures with software built for the agricultural domain to promote a true-to-life use. Specifically, these software's include FarmOS and FarmStack Trusted Connectors. I was able to exemplify a strong use case for the FarmStack Trusted Connector data transfer utility which is currently in early development. I, further, found a use for FarmOS beyond just their web interface, database and API's. I utilized their data structure schema as an early approach to resolving the issue of disorganized agricultural data that is inconsistent between farmers. My final noteworthy contribution was the Internal Data Handler. A cryptographic data storage server designed specifically to secure and manage data internal to a compute instance.

The majority of specific processes I developed were not necessarily new, but my collective use of them in answering this specific question is unique. Further, my approach to solving the issue of training a secure model on secure data as self-contained between peers is diverse from other attempts like functional encryption, perturbation [18] and block-chain [4, 5] among others [16, 17, 19].

The future of machine learning on secure agricultural data surely has a way to go, but the research I have discussed in this paper is a step in the right direction. The fourth and final network architecture I exhibited may still require additional work to fill holes in security I may have missed. Further, there are extensive areas, some of which I defined in chapter 5, that require enhancements to improve architecture throughput. This is especially true for the IDH. Nevertheless, Architecture 4 is a viable option for an environment capable of training a Consumer's secure model on a Provider's secure agricultural data without revealing the secure information to either party.

Acronyms

BPA Batch Processing Application.

CDC Consumer Data Client.

CDH Consumer Data Handler.

CFH Cryptographic File Handler.

EDH External Data Handler.

FosC FarmOS Connector.

FSS File Storeage Server.

IDH Internal Data Handler.

PDC Provider Data Client.

PDCo Payload Delivery Connector.

PDH Provider Data Handler.

PVC Payload Validation Connector.

Terminology

Consumer The owner of the model that is to be trained on the Provider's data in a model training network..

Consumer Instance The remote instance owned by the Consumer where the final training results payload will be sent to..

Core-Process A collection of one or more sub-processes in a single container..

External-CFH The CFH that only handles the external data payload..

FosC-consumer Consumer end of the FosC. Contains the consumer_core and consumer_app.

FosC-provider Provider end of the FosC. Contains the provider_core and provider_app.

Generator The training image and label generator application..

Internal-CFH The CFH that only handles internal data..

Payload *(* Results Payload, Training Results Payload) - File and Directory archive containing encrypted copies of training results and the key to decrypt them..

PDCo-consumer Consumer end of the PDCo. Contains the consumer_core and consumer_app.

PDCo-provider Provider end of the PDCo. Contains the provider_core and provider_app.

Process Either a core-process or sub-process depending on context..

Provider The owner of the secure data set that will be passed through a model training network..

Provider Instance The remote instance owned by the Provider where the secure data will be sent from..

PVC-consumer Consumer end of the PVC. Contains the consumer_core and consumer_app.

PVC-provider Provider end of the PVC. Contains the provider_core and provider_app.

Requester The FarmOS Data Requester. A sub-process in the BPA in architecture 1, but a standlone core-process in all other architectures..

Service Synonym for process..

Sub-Process A single program with a specific purpose..

Trainer The model training application..

References

[1] A. Tzachor, M. Devare, B. King, S. Avin, and S. O. hEigeartaigh, “Responsible artificial intelligence in agriculture requires systemic understanding of risks and externalities,” *Nature Machine Intelligence*, vol. 4, pp. 104–109, 2 2022. [Online]. Available: https://www.nature.com/articles/s42256-022-00440-4

[2] L. Wiseman, J. Sanderson, A. Zhang, and E. Jakku, “Farmers and their data: An examination of farmers’ reluctance to share their data through the lens of the laws impacting smart farming,” *NJAS - Wageningen Journal of Life Sciences*, vol. 90-91, 12 2019.

[3] M. B. Cole, M. A. Augustin, M. J. Robertson, and J. M. Manners, “The science of food security,” *npj Science of Food*, vol. 2, 12 2018.

[4] J. Liang, S. Li, B. Cao, W. Jiang, and C. He, “Omnilytics: A blockchain-based secure data market for decentralized machine learning,” 7 2021. [Online]. Available: http://arxiv.org/abs/2107.05252

[5] V. Koutsos, D. Papadopoulos, D. Chatzopoulos, S. Tarkoma, and P. Hui, “Agora: A privacy-aware data marketplace,” vol. 2020-November. Institute of Electrical and Electronics Engineers Inc., 11 2020, pp. 1211–1212.

[6] S. Volos, K. Vaswani, and R. Bruno, “Graviton: Trusted execution environments on GPUs,” in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 681–696. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/volos

[7] N. Peng, R. Doshi, S. Hiu, M. Ball, A. Edwards, A. Smith, and B. Murray, “Global cloud services market q1 2021,” https://www.canalys.com/newsroom/global-cloud-market-Q121., april 2020, accessed: 2022-5-10.

[8] K. Kambatla, G. Kollias, V. Kumar, and A. Grama, “Trends in big data analytics,” *Journal of Parallel and Distributed Computing*, vol. 74, pp. 2561–2573, 2014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731514000057

[9] S. M. Idrees, M. A. Alam, and P. Agarwal, “A study of big data and its challenges,” *International Journal of Information Technology (Singapore)*, vol. 11, pp. 841–846, 12 2019. [Online]. Available: https://link.springer.com/article/10.1007/s41870-018-0185-1

[10] P. Taylor, “Big data - it is not the silver bullet that some people think it is?” *The Open Access Journal of Science and Technology*, vol. S, August 2020. [Online]. Available: https://www.agialpress.com/articles/big-data–it-is-not-the-silver-bullet-that-some-people-think-it-is.pdf

[11] D. Jaeger, H. Graupner, A. Sapegin, F. Cheng, and C. Meinel, “Gathering and analyzing identity leaks for security awareness,” in *Technology and Practice of Passwords*, S. F. Mjølsnes, Ed. Cham: Springer International Publishing, 2015, pp. 102–115.

[12] Sam Cook Data journalist, privacy advocate and C.-C. Expert, “US schools leaked 28.6 million records in 1,851 data breaches since 2005,” https://www.comparitech.com/blog/vpn-privacy/us-schools-data-breaches/., Jul. 2020, accessed: 2022-5-3.

[13] J. Bunge and T. C. Dreibus, “Monsanto confirms security breach.” https://www.wsj.com/articles/monsanto-confirms-security-breach-1401403777, May 2014, accessed: 2022-5-3.

[14] F. J. Pierce and P. Nowak, “Aspects of precision agriculture,” ser. Advances in Agronomy, D. L. Sparks, Ed. Academic Press, 1999, vol. 67, pp. 1–85. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0065211308605131

[15] Y. Huang, Z. xin CHEN, T. YU, X. zhi HUANG, and X. fa GU, “Agricultural remote sensing big data: Management and applications,” *Journal of Integrative Agriculture*, vol. 17, no. 9, pp. 1915–1931, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2095311917618598

[16] S. Sagar and C. Keke, “Confidential machine learning on untrusted platforms: a survey,” *Cybersecurity*, vol. 4, 12 2021.

[17] “Confidential computing: Hardware-based trusted execution for applications and data,” 1 2021.

[18] W. jie Lu, S. Kawasaki, and J. Sakuma, “Using fully homomorphic encryption for statistical analysis of categorical, ordinal and numerical data.” Internet Society, 5 2017. [Online]. Available: https://doi.org/10.14722/ndss.2017.23119

[19] N. Lisin and S. Zapechnikov, “Methods and approaches for privacy-preserving machine learning,” vol. 80. Springer Science and Business Media B.V., 2020, pp. 141–148. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-33491-8_17

[20] A. Paullada, I. D. Raji, E. M. Bender, E. Denton, and A. Hanna, “Data and its (dis)contents: A survey of dataset development and use in machine learning research,” *Patterns*, vol. 2, no. 11, p. 100336, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666389921001847

[21] S. J. Cunningham, “Dataset cataloging metadata for machine learning applications research,” in *Proceedings of the Sixth International Workshop on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, D. Madigan and P. Smyth, Eds., vol. R1. PMLR, 04–07 Jan 1997, pp. 139–146, reissued by PMLR on 30 March 2021. [Online]. Available: https://proceedings.mlr.press/r1/cunningham97a.html

[22] C. Anderson, “Docker [software engineering],” *IEEE Software*, vol. 32, no. 3, pp. 102–c3, 2015.

[23] B. B. Rad, H. J. Bhatti, and M. Ahmadi, “An introduction to docker and analysis of its performance,” *International Journal of Computer Science and Network Security (IJCSNS)*, vol. 17, no. 3, p. 228, 2017.

[24] D. Merkel, “Docker: lightweight linux containers for consistent development and deployment,” *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[25] “Fraunhofer connectors,” 1 2022. [Online]. Available: https://www.dataspaces.fraunhofer.de/en/software/connector.html

[26] L. Beckmann, C. Banse, M. Lux, and G. Brost, 9 2021. [Online]. Available: https://github.com/industrial-data-space/idscp2-jvm/wiki/IDSCP2-Core

[27] M. Pradhan, G. Sagar, M. Gautam, V. Singh, and R. K. M., 5 2022. [Online]. Available: https://github.com/digitalgreenorg/farmstack-open

[28] "Aws nitro enclaves," https://aws.amazon.com/ec2/nitro/nitro-enclaves/., accessed: 2022-5-10.

[29] Kamangir, Guevara, and Earles, "Industrial-scale vineyard yield forecasting via spatiotemporal deep learning using satellite and management data fusion," (in-review).

[30] M. Panda, "Performance analysis of encryption algorithms for security," in *2016 International Conference on Signal Processing, Communication, Power and Embedded System (SCOPES)*, 2016, pp. 278–284.

[31] C. Tezcan, "Optimization of advanced encryption standard on graphics processing units," *IEEE Access*, vol. 9, pp. 67 315–67 326, 2021.

[32] S. Neves and F. Araujo, "On the performance of gpu public-key cryptography," in *ASAP 2011 - 22nd IEEE International Conference on Application-specific Systems, Architectures and Processors*, 2011, pp. 133–140.

[33] M. Abdel-Basset, G. Manogaran, A. E. Fakhry, and I. El-Henawy, "2-levels of clustering strategy to detect and locate copy-move forgery in digital images," *Multimedia Tools and Applications*, vol. 79, no. 7, pp. 5419–5437, 2020. [Online]. Available: https://doi.org/10.1007/s11042-018-6266-0

[34] B. Hong, D. Plantenberg, D. Long, and M. Sivan-Zimet, "Duplicate data elimination in a san file system." 01 2004, pp. 301–314.