Hybrid Cloudification of Legacy Analysis Tools

in Aeroderivative Gas Turbine Design

FOZAIL AHMAD

Department of Electrical and Computer Engineering

McGill University, Montreal

August, 2021

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of

Master of Science

Abstract

The design of aero-derivative gas turbines (AGT) at Siemens Energy is a complex process that requires the use of various software tools for mechanical design, validation and testing. Due to the evolutionary nature of AGT development, changes are incremental and built upon previous models. Therefore, it is preferred to utilize the same tools that were initially developed and adopted decades ago in order to preserve the models and their real world offsets.

Executing finite element analysis (FEA) at Siemens Energy is computationally intensive while its legacy tool is designed to be manually deployed onto local desktops and servers in a sequential manner. Consequently, the analysis of multiple AGT designs is a laborious process requiring users to invoke custom tool scripts on a limited pool of local computing resources. Users are responsible for maintaining each computing environment and have to checkpoint each analysis task manually.

To address these challenges, I propose to cloudify the legacy FEA software tool by designing and developing an Analysis Tool as a Service (ATaaS) with a distributed microservice architecture that provides automation, scalability and distribution ofthe FEA software in a hybrid cloud deployment. The thesis contributions include (1) a distributed software architecture for ATaaS, (2) a prototype implementation for ATaaS integrating a legacy FEA tool used at Siemens and (3) various performance estimation models for executing FEA over ATaaS. (4) To validate the proposed architecture, models and prototype implementation, I carry out an experimental evaluation of performance and the functional testing of key components.

Abrégé

La conception de turbines à gaz aérodérivatives (AGT) chez Siemens Energy est un processus complexe qui nécessite l'utilisation de divers outils logiciels pour la conception et validation mécanique. En raison de la nature évolutive du développement de l'AGT, les changements sont incrémentiels et s'appuient sur les modèles précédents. Par conséquent, il est préférable d'utiliser les mêmes outils qui ont été initialement développés et adoptés il y a des décennies afin de préserver les modèles et leurs décalages dans le monde réel.

L'outil d'analyse par éléments finis (FEA) de Siemens Energy est exigeant en termes de calcul et est développé pour être déployé manuellement sur des ordinateurs de bureau et des serveurs locaux de manière séquentielle. Par conséquent, l'analyse de plusieurs conceptions d'AGT est un processus laborieux qui exige que les utilisateurs invoquent des scripts d'outils sur un groupe limité de ressources informatiques locaux. Les utilisateurs sont responsables de la maintenance de chaque environnement informatique et doivent vérifier manuellement chaque tâche d'analyse.

Pour répondre à ces défis, je propose de cloudifier l'outil logiciel FEA existant en développant un outil d'analyse en tant que service (ATaaS) avec une architecture distribuée de microservices qui fournit l'automatisation, la scalabilité et la distribution du logiciel FEA dans un déploiement de cloud hybride. Les contributions de la thèse comprennent (1) une architecture logicielle distribuée pour ATaaS, (2) une implémentation prototype pour ATaaS et (3) divers modèles d'estimation des performances pour exécuter de FEA sur ATaaS. (4) Pour valider l'architecture, les modèles et l'implémentation proposés, je réalise une évaluation expérimentale des performances et des tests fonctionnels.

Acknowledgements

First and foremost, I would like to thank my supervisor Prof. Daniel Varro, for his diligent guidance and funding throughout my Master's study and research. Through his kindness, patience, enthusiasm, and, immense knowledge he kept me motivated throughout the duration of my studies.

I would like to express my gratitude to Mr. Martin Staniszewski of Siemens Energy, who has been immensely supportive through out and provided me with all resources necessary to conduct my research. I would like to extend my appreciation to all other Siemens engineers, especially, the PGDA team who supported me through this research.

I am thankful to all the other members of the research group at McGill University, in particular, Maruthi Rangappa who provided me valuable support and suggestions, which helped me carry out my research.

I am grateful to Dr. Sébastien Mosser for his valuable review of my thesis and helpful guidance of my research.

Finally, I am extremely grateful to my family but especially Faiq Khalid for his continuous support and encouragement through all kinds of situations during my studies.

This work was partially supported by the Digital Multidisciplinary Analysis and Design Optimization Platform for Aeroderivative Gas Turbines project (SiemensCa CRDPJ 513922-17 X-247371 and NSERC CRDPJ 513922-17 X-247323 funds).

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation: Design of Aeroderivative Gas Turbines

Aeroderivative Gas Turbines Aeroderivative Gas Turbines (AGTs) are gas turbines made for electrical power generation by adapting aircraft turbine engines. Aircraft turbine engines are designed to ramp up and slow down very quickly for effective flight control which enables AGTs to dynamically change their electrical output based on load changes in the grid. The ability of AGTs to rapidly adapt to grid conditions makes them well suited to efficiently fill demand peaks in the electrical grid as traditional power generation technologies such as hydroelectric dams cannot be brought online or offline as quickly. Furthermore, AGTs are comparatively lighter when compared to traditional turbines making them useful in applications with weight restrictions such as off shore marine installations.

Siemens Energy has significant business interest in AGT, offering numerous AGT models with a wide range of electrical power generation capacity. AGTs are an important tool in offering operational and deployment flexibility to operators given the diverse gas turbine market which includes industrial and heavy duty gas turbines. As the shift to renewable energy progresses worldwide, AGTs are poised to be a part of this important process due to their ability to use alternative fuels such as hydrogen. Hydrogen is an energy dense renewable fuel source who's combustion produces no harmful exhausts.

Investments made into improving the performance of AGTs will provide cleaner energy today and for decades to come.

Finite Element Analysis Improving the performance of AGTs is an iterative process spread over many years, which involves designing new AGTs models and then analysing these models to determine if the performance has improved. Instead of using physical prototypes to develop AGTs, computer software is used to build AGT engine models and then to analyze AGT engine models for their simulated mechanical performance. As a critical step in the design process of AGTs, Finite Element Analysis (FEA) is performed on thermomechanical models of AGTs to analyze and test new designs in order to validate their performance against improvement objectives.

At Siemens Energy, legacy software tools are used for FEA. Legacy FEA tools are preferred as engineers gain an understanding of the digitization process and errors of the software and can then reliably estimate the offsets required within the software to achieve the desired real world results. However, utilizing legacy software tools that are no longer maintained introduces its own drawbacks, mainly a lack modern software functionality such as automation, scalability and portability.

Problem statement The key challenge faced by Siemens Energy when using the legacy FEA software tool is the manual and time consuming process for running the simulations. It must be invoked locally and relies on local files to perform its analysis. If multiple analysis have to be performed simultaneously, then custom scripts must be manually invoked. Furthermore, the software is primarily designed to be deployed on-premise and there is no built-in mechanism for distributing the analysis across a dynamic pool of hosts. Any potential solution to this challenge must address the following challenges:

C1 Integrate with the Windows-based legacy FEA software tool for analysis.

C2 Enable distributed execution across heterogeneous computer hosts.

C3 Provide a mechanism for automated task submission.

C4 Support horizontal scaling for parallel computing.

1.2 Objectives

In order to address these challenges I propose to cloudify the legacy FEA software tool by designing and developing an Analysis Tool as a Service (ATaaS) with a distributed microservice architecture in order to provide automation, scalability and distribution of the FEA software in a hybrid cloud deployment with Windows support.

Contributions In the particular, this thesis presents the following contributions:

- I propose a *distributed software architecture for ATaaS* which is deployable over a hybrid computing platform using container technology.
- I provide a *prototype implementation* for the ATaaS architecture where a legacy FEA tool widely used at Siemens is seamlessly integrated with Amazon Web Services.
- I provide *various performance estimation models of FEA over the ATaaS* containing estimates for task time, task analysis time, task batch service time and task wait time.
- To validate the proposed architecture, models and prototype implementation, I carry out an *experimental performance evaluation* and the *functional testing of key components*.

An overview of the proposed ATaaS solution is provided in Figure 1.1 with the blue highlighted boxes represent the novel contributions. After identifying the required thermomechanical engine models and obtaining an existing execution script used by Siemens engineers for the legacy FEA tool in a task preparation step, the cloudification of FEA involves four key steps:

1. **Task initialization** involves creating a task and configuring it accordingly (ie. specifying the engine models required).

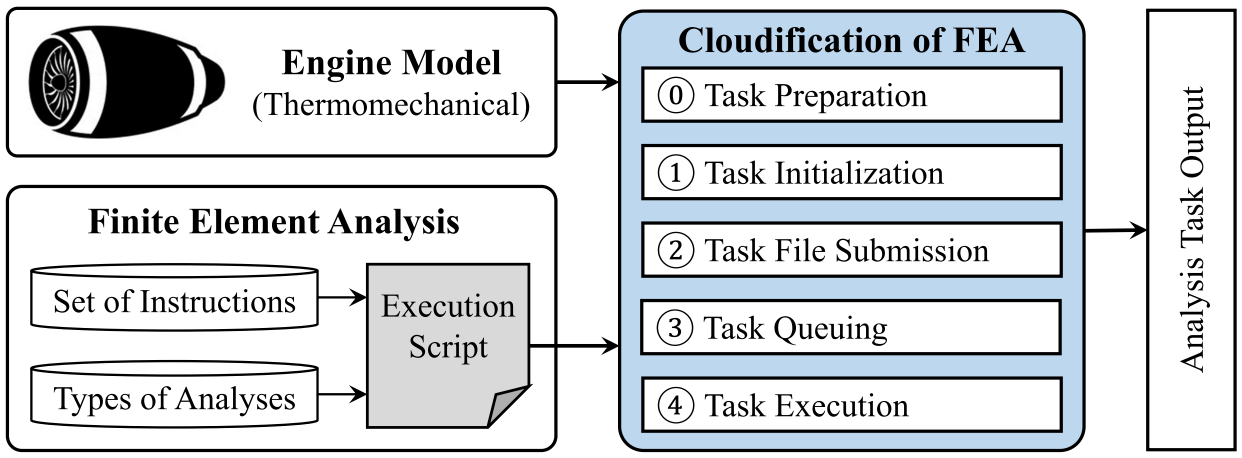


Figure 1.1: An overview of the proposed cloudification of legacy FEA software

2. **Task file submission** involves uploading the execution script and any associated files for the task.

3. **Task queuing** involves queuing the task you just created and submitted files for to the execution queue.

4. **Task execution** involves executing the task by invoking the legacy FEA tool and passing to it the task execution script, any associated files and the engine models.

Highlights of experimental results An initial experimental evaluation of the proposed framework and its prototype implementation has been carried out in (1) a simulated setting used for validating the various performance estimation models. Moreover, the performance of ATaaS has also been assessed when executing (2) a real engineering task performed by Siemens engineers.

Our initial evaluation shows that in a simulated setting the ATaaS is capable of reliably reducing the total service time for a batch of FEA tasks with respect to the concurrent execution capacity of the ATaaS when compared to the theoretical predictions of the performance estimation models. Furthermore, the ATaaS hybrid prototype deployment is able to successfully execute real engineering tasks and provide scalable and automated FEA task execution parallelization.

Benefits for Siemens The proposed ATaaS framework brings substantial benefits for Siemens engineers in two aspects. The increased level of *automation* helps engineers to easily initiate and execute FEA tasks without locally invoking (and installing) the FEA software. The increased scalability and distribution provided by ATaaS offers significant *speedup* in the overall AGT design process by parallelizing the individual FEA tasks, which no longer requires manual interaction.

1.3 Structure of the Thesis

The rest of the thesis is structured as follows:

- **Chapter 2** introduces the core concepts and technologies used for the proposed framework and its prototype implementation.
- **Chapter 3** showcases the ATaaS framework to cloudify the legacy FEA tool.
- **Chapter 4** discusses the ATaaS prototype implementation and hybrid deployment.
- **Chapter 5** demonstrates the correctness testing of all the ATaaS prototype components and experimental validation of the proposed cloudification solution.
- **Chapter 6** discusses the related work done in the research areas of cloudification, containerization, and ATaaS.
- **Chapter 7** summarizes the key accomplishments of the thesis and identifies the future work areas.

Chapter 2

Background

2.1 Finite Element Analysis

Finite element analysis (FEA) is the mathematical modelling of physical objects using the numerical technique of finite element method (FEM). FEM is process in which an object is broken down into many finite element parts and then a system of algebraic equations are used to predict how each finite element and the entire system will behave. FEA allows engineers to model physical objects and predict how it will react to certain forces, temperatures, fluids, and other physical phenomenon. This enables engineers to determine whether a mechanical model will meet its design requirements using relatively inexpensive FEA computer software compared to making use of real physical prototypes [48].

Use Cases for Finite Element Analysis

AGT Design Aero derivative gas turbines (AGT's) are inherently unstable machines which require a careful balance of multiple mechanical domains. The ability to evaluate different AGT designs under various operating conditions in order to determine the optimal AGT design and operating conditions is critical for the successful development and progress of AGT's at Siemens Energy. Therefore, FEA software is used by companies to design AGT's and evaluate whether they meet design objectives without having to build costly prototypes. FEA is primarily used to evaluate the mechanical stress, mechanical vibration, material fatigue, mechanical motion, thermo-mechanical behaviour, fluid flow within an AGT design under different test operating conditions. After many iterations of designing and testing AGT's using FEA on computers, physical prototypes which are hopefully as close as possible to the production version can be manufactured and tested.

FEA Limitations FEA software is ultimately ran on digital computers which are discrete by definition. On the other hand physical objects and phenomenon are continuous which implies that any FEA done on computers will contain some errors compared to the real world. Understating the offsets required when designing models and performing FEA such that the final physical objects produced from these designs are not affected by these errors takes a significant amount of time and research. This results in companies like Siemens Energy maintaining and using the same software for decades in order to not loose the hard earned insight on how to offset designs and when to expect computational errors regardless of the numerous improvements made in modern FEA software.

Finite Element Analysis at Siemens

FEA Software The FEA software used by Siemens Energy is a legacy tool designed to be deployed in private clouds with static number of Windows hosts. It must be invoked manually using a CLI and expects a bare metal computing environment. The FEA software can be used with a GUI or through *execution scripts* that would be passed during invocation. The *execution script* is where all the references to any resources are required, such as the *engine models*. The resources required by an *execution script* are generally expected to be available on network drives mounted onto the host as populating large collection of resource files on every host is unfeasible. Despite that the *execution script* must be written in such a way that everything is referenced to local hard coded locations. Lastly, the FEA software can only be deployed in certain countries as the technology along with AGT engine models must respect Canadian Export Control laws.

License Management Service Furthermore, the legacy FEA software used by Siemens Energy requires to authenticate with a *License Management Service* for every invocation. The *License Management Service* is an IP service that must be made available to any host that will run the FEA software. When installing the FEA software on the host you need to specify the port and IP address of the *License Management Service*. The *License Management Service* is a very basic service that authorizes any FEA software authentication request it receives. That means no extensive checks are performed on the authentication request to ensure that the FEA software is being used in the right country or by an authorized user. Due to this limitation, the *License Management Service* is generally only made available to secure internal networks where all traffic is considered safe and authorized.

2.2 Distributed Computing

Distributed computing is a process in which the computation for a single system is spread across separate computers [43]. Any system making use of distributed computing has to be built using a distributed architecture to take advantage of the hardware setup. The primary advantage of distributed computing is the scalability and redundancy provided by using distributed architectures:

- **Scalability**: A distributed computing system can be scaled horizontally by adding more computers to increase capacity instead of having to scale individual computers vertically which is arguably more difficult and limited by the processing power offered by a single computer.
- **Redundancy**: Distributed computing can allow for redundancy in a system, allowing the architecture to treat individual computers as dispensable to the system in case they fail as many instances of each system component would be deployed at any given time.

Every distributed computing system generally requires two main components to function correctly; a non distributed system view and a message passing function [42]:

- **Non Distributed System View**: For any distributed system is it important to abstract the entire system such that is appears as a single centralized system to end users. By its very nature distributed computing is spread across many computers, making it difficult to deal with the entire system simultaneously as a user. Therefore, a unified system endpoint provides users a simple and efficient way of interacting with the system despite the actual system begin distributed across many computers.
- **Message Passing**: Most distributed systems rely on some form of message passing for different components to communicate and coordinate. This is essential as although each component is ultimately independent they need to work together to provide the functionality of the system. Message passing can rely on internet protocols, message queues, databases. etc.

2.2.1 Master-Worker Paradigm

The master-worker paradigm from distributed computing is a widely used pattern in distributed systems [13, 43]. Tasks are units of work which are the main object in such systems, the master handles the tasks while the workers execute the tasks. This mainly consists of tasks being broken down and stored in a bag by the master and the worker being sent tasks from the manager to execute. The master-worker paradigm can be simplified into three core concepts for most distributed systems; *Manager*, *Worker* and *Executor*.

Manager

The *Manager* in distributed computing is the component which is responsible for the entire system and has a global viewpoint. It serves as the centralized system endpoint from where the users can interact with the distributed system. The *Manager* interacts with users of the system to create and manage tasks in a *Task Repository*. A *Manager* can be responsible for *Workers* by actively scaling the number of *Workers* and managing their operation. However, this is not something that a *Manager* is required to implement, it can be handled by a custom-built system. A *Manager* also needs to send tasks to *Workers* and this can be accomplished through synchronous or asynchronous communication using the message passing functionality of the system.

Worker

The *Worker* in distributed computing is essentially a node in the system serving as a small player in a bigger system. *Workers* can either receive or pull tasks from the *Manager* and then are responsible for being able to execute those tasks. Tasks can be directly executed by a *Worker* however tasks can also be launched as separate processes which can then be called *Executors*. *Workers* are responsible for their tasks and can coordinate with the *Manager* to ensure that tasks are successfully completed.

Executor

The *Executor* in distributed computing is the process on a *Worker* node that performs the execution for a task. An *Executor* is generally isolated from the distributed computing functionality of the *Worker*, allowing for the task be to executed uninterrupted. Depending on the implementation, an *Executors* can be a long running process which will execute many tasks or it can be an ephemeral process that will execute only a single task. *Executors* require the bulk of the computation power in any distributed system since they actually execute tasks.

2.3 Containers

Container technology is a form of operating system (OS) virtualization or abstraction, allowing software to be packaged into standard units and executed virtually anywhere.

Container images contain all the code, executables, libraries, system files and settings and can be executed as containers on any host with an OS that supports containers. When executing, each container receives its own isolated space within the host OS where all the storage, networking and computing is separate and protected. There are three key advantages that containers offer to software deployment:

- **Lightweight**: Containers are fast as they do not package any guest OS in their images and rely on the host OS kernel to execute in isolation. This also enables high container density per host due to their small footprint.
- **Portable**: Containers can be executed on any supported OS and does not have any other dependencies on the host as everything required for execution is packaged into the image, make them useful for hybrid systems with heterogeneous OSs.
- **Scalable**: As every container is an isolated process, containers can be easily scaled vertically within the same host due to container isolation. Moreover, containers can be deployed in clusters making them ideal for horizontal scaling such as in distributed systems.

Due to these key advantages containers are preferred over virtual machines (VMs). Virtual machines are the virtualization or abstraction of physical hardware which means they must package a guest OS on top of everything else already in a container, making them large heavy objects. Moreover, virtualizing hardware is computationally expensive making VMs slower when compared to container which simply virtualize the OS. The key differences between a container and virtual machines can be summarized in Figure 2.1.

2.3.1 Docker Containers

Docker is a popular open source container technology that is used by many companies. Docker container technology is preferred due to its wide support on many different operating systems including Windows and the open source community built around it. Various plugins are available for Docker, enabling extensive third party components support.

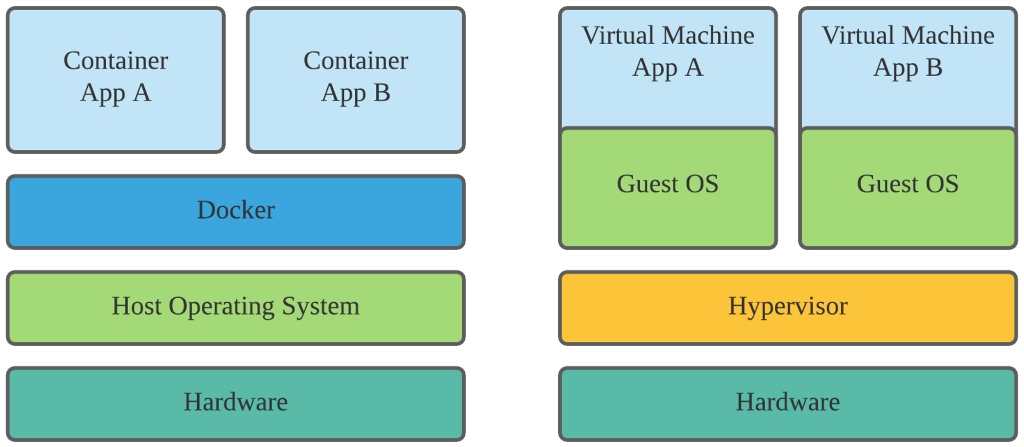


Figure 2.1: Containers vs Virtual Machines

Due to all these aforementioned advantages and mature enterprise support Docker is the container technology of choice chosen by our industrial partner Siemens Energy. When using Docker there are four main Docker components that users interact with; Docker Images, Docker Engine, Docker Client and the Docker API.

Docker Images

Docker containers are spawned from images that are available locally on the host or in an container image registry. A Docker image can be built by anyone using a Dockerfile which contains the following elements:

- **Base Image**: Every Docker image must be built from a base image, base images are essentially a snapshot of a bare bones OS such as Ubuntu or Windows.
- **Commands**: On top of the base image, the image creator can run commands that will setup the container as they wish. This can include copying files, installing additional libraries and so forth.
- **Entry Point** The entry point of an image is the process that is internally started when the image is launched as a container, this is basically the container start instruction such as running a web server.

Docker Engine

The Docker engine is the core technology of Docker enabling containers to run on a host OS. The Docker engine must be installed on any host that need to run Docker containers. There are thee main sub function within the Docker Engine:

- **Container Runtime**: Within the Docker engine there is the container runtime which is responsible for the OS virtualization; executing containers in isolation while providing them with storage, networking and compute power.
- **Image Handling**: The Docker engine is used to build Docker images from Dockerfiles, and is also needed to store and retrieve Docker images from the local host or a repository.
- **API**: Furthermore, the Docker engine provides the daemon that serves the Docker API. The Docker engine also interfaces with other components offered by docker such as the Docker CLI, or external plugins through the Docker API that it implements and exposes.

Docker Client

The Docker client are the tools and wrappers used to access the Docker engine. The Docker API can be directly accessed using a HTTP client however this can be cumbersome and inefficient in many scenarios. Therefore Docker provides two types of clients to users in order to interact with the Docker engine:

- **CLI**: Commands for the host's local command line interface (CLI) which can be invoked from the terminal to work with the Docker engine. For example, the CLI can be used to create an image from a local Dockerfile and then run the newly created image as a container.
- **SDK**: Docker provides software development kits (SDKs) which can be used within programs to access the Docker API to work with the Docker Engine. For example,

if you want a python program to access the Docker engine it does not have to make an HTTP request to the Docker API but can simply use the SDK which will make the appropriate calls to the Docker API.

Docker API

The Docker API is a service endpoint provided by the Docker engine enabling users to interact with the engine. The Docker engine is capable of many different functions as explained earlier however its through the API that users can make use of the Docker engine. The API is made available as a RESTful API on the host's local network, which means it can be accessed using HTTP requests. As the Docker API is web-based, it can be accessed through any IP network as long as there is access to the local host's Docker API service endpoint. The Docker API being available as a network web-service is particularly useful if the Docker engine on the host will be a part of a larger cluster.

2.3.2 Container Orchestration

Container orchestration technology builds on top of container technology to enable the deployment and management of a large number of containers across many different hosts. Running a single container on a single host is a relatively simple process which can be performed manually. However, when running a large complex application requiring multiple containers working in tandem, a manual process is infeasible and the need for container orchestration emerges. Container orchestration enables the automated provisioning, deployment, networking, scaling, availability, and lifecycle management of containers on a cluster of hosts. Container orchestrators generally have a master controller which is responsible for all the nodes in the cluster and node agents which connect the nodes to the cluster master. Container orchestrators essentially enable distributed systems and their architecture observes the master-worker computing paradigm.

There are many container orchestrators available for container technology however the two most popular ones are Kubernetes developed by Google [40] and Swarm developed by Docker [19]. These container orchestrators are highly customizable and are utilized by many companies. However, they have still have room for improvement to allow for widespread adoption, such as hybrid cloud deployment support is still limited. Within the scope of this thesis there was no feasible container orchestrator which could be used simply because the legacy FEA software is Windows based and there is no container orchestrator available that supports Windows containers natively. Most container orchestrators are designed for Linux containers and with Linux based master controllers, and the ability to run Windows and Linux containers side by side is still in experimental stages with no stable production solution.

2.4 Service Technology

2.4.1 Microservices

Microservices is an architectural approach by which a larger complex application is built using loosely coupled smaller services communicating using lightweight mechanisms [18]. Each microservice is supposed to serve a single function to facilitate independent and simple deployment [73]. As each microservice is supposed to be independent, each of them has its own technology stack and can be developed and tested without depending on other components in the application. Microservices make it easy to scale applications horizontally and to take advantage of distributed computing by placing each microservice on different hosts. A key factor in ensuring that microservices are successful is that communication between each of them is simple, most microservices use RESTful API's or message passing services to communicate.

There are no standardized rules or definitions which determine what qualifies as a microservice architecture; however, it is important to identify what microservices are not. Microservices are the opposite of monolith architectures which make large applications by using a single tightly coupled component containing all the logic required. In general, monolith architectures are difficult to maintain and are not recommended [31].

Domain Driven Design Microservices should generally follow the principles of Domain Driven Design (DDD). DDD is a style of software development where software is built using domain model which has a detailed and complete understanding of the processes and rules of the domain in question [75]. To successfully build any domain model it is recommended that a Ubiquitous Language be used to enable developers and users to have a clear understanding [20]. Particularly the concept of Bounded Context from DDD is central to designing and defining the boundaries of any microservice [53]. The Bounded Context concept is part of of DDD's strategic design section where a large models are divided into different Bounded Contexts with explicit interrelationships [20].

2.4.2 RESTful API

REST stands for REpresentational State Transfer and is an flexible architectural style to design and develop web applications [21]. In practice this means that the REpresentational State of resources in these web applications is exchanged between the client and the server. The application programming interface (API) of any application is a set of rules and methods that allow external entities to connect and communicate with the application. A RESTful API is simply an API that subscribes to the REST architectural style of exchanging the state of resources. RESTful APIs are built on top of the HTTP protocol and methods, allowing for CRUD operations on resources and providing request parameters and headers. RESTful APIs provide a flexible and lightweight way for applications to communicate and are commonly used in microservice application architectures where loosely coupled components need to share resources [32].

The REST architectural style for APIs requires that the following six architectural constrains be respected:

1. **Uniform Interface**: All the API requests for the same resource should have the same logical URI regardless of the requesting client.

2. **Client-Server Decoupling**: The API client and the server must be completely independent from each other. The client should only interact with the URIs of resources and not directly with server applications and vice versa.

3. **Stateless**: All the API requests should be stateless such that they contain all the required information in order for the server to process it without needed any further context.

4. **Cacheability**: Resources should be cachable on the client and server side when possible in order to improve performance and stability.

5. **Layered System Architecture**: The API client should not be able to determine whether it dealing directly with the end server or an intermediary system.

6. **Code on Demand (optional)**: API request can return executable code in certain cases to customize client side behaviour on demand.

Flask Flask is an open source web framework module for Python. Flask is considered a lightweight microframework as it does not have any dependencies to external tools or libraries. Due to this, Flask does not include any full service web framework features such as database abstractors. However, any required functionality can generally be made available via plugins. Due to most projects already using Python at Siemens Energy and Python offering cross OS compatibility, Flask was chosen as the web framework of choice for the ATaaS. Flask can be used to develop RESTful APIs for microservices, which will be required for the development of the ATaaS prototype.

2.5 Cloud Technology

Cloud computing is a model for enabling on-demand ubiquitous and convenient access to shared pools of configurable computing resources over the network, that can be rapidly and easily provisioned and released [52]. The National Institute of Standards and Technology (NIST) defined this model of having three service models, and four deployment models which will be further explained below. A popular public cloud provider that offers public cloud computing as defiend by NIST and chosen by Siemens Energy due to its enterprise support and popularity is Amazon Web Services (AWS). The AWS resources utilized within this thesis will be briefly described.

2.5.1 Service Models

According to NIST there are three cloud service models: *Infrastructure as a Service* (IaaS), *Platform as a Service* (PaaS) and *Software as a Service* (SaaS). Each service model covers a larger portion of the technology stack as shown in Figure 2.2. Here are the NIST definitions for the capability provided to the cloud consumer for each service model [52]:

- **IaaS**: *"The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components (e.g., host firewalls)."*
- **PaaS**: *"The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment."*
- **SaaS**: *"The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure . The applications are accessible from various client devices through either a thin client interface, such as a web browser (e.g., web-based email), or a*

program interface. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings."

The ATaaS proposed and developed within this thesis is closest to a SaaS service model where a software is provided as a service, in this case a legacy FEA tool. Engineers need to perform FEA for their jobs in AGT development and would prefer to not invest time in learning and managing their tool's technology. A SaaS service model provides all features of the software as service in such a way that users do not need to do maintenance of the software. Instead, users only configure the software for a particular feature and provide the required data [77]. Therefore, providing the FEA tool as a SaaS (we define it as ATaaS) enables engineers to efficiently and effectively perform FEA.

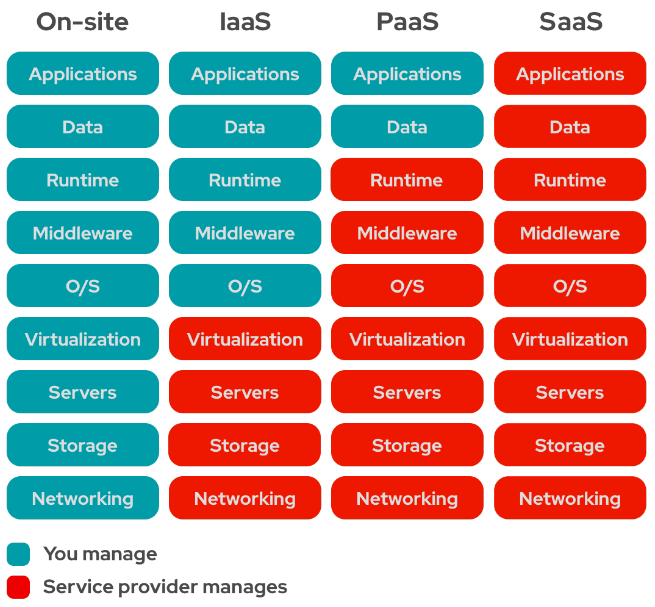


Figure 2.2: Cloud service models [61]

2.5.2 Deployment Models

According to NIST there are four cloud deployment models: Private Cloud, Community Cloud, Public Cloud and Hybrid Cloud. Each cloud deployment models defines who operates the provisioned IT infrastructure for a cloud application. Here are the NIST definitions for each cloud deployment model [52]:

- **Private**: *"The cloud infrastructure is provisioned for exclusive use by a single organization comprising multiple consumers (e.g., business units). It may be owned, managed, and operated by the organization, a third party, or some combination of them, and it may exist on or off premises."*

- **Community**: *"The cloud infrastructure is provisioned for exclusive use by a specific community of consumers from organizations that have shared concerns (e.g., mission, security requirements, policy, and compliance considerations). It may be owned, managed, and operated by one or more of the organizations in the community, a third party, or some combination of them, and it may exist on or off premises."*

- **Public**: *"The cloud infrastructure is provisioned for open use by the general public. It may be owned, managed, and operated by a business, academic, or government organization, or some combination of them. It exists on the premises of the cloud provider."*

- **Hybrid**: *"The cloud infrastructure is a composition of two or more distinct cloud infrastructures (private, community, or public) that remain unique entities, but are bound together by standardized or proprietary technology that enables data and application portability (e.g., cloud bursting for load balancing between clouds)."*

The ATaaS is designed using a distributed microservice architecture for deployment on the hybrid cloud using both the private and public clouds. The hybrid cloud deployment model provides the benefits of each while minimizing the risks associated with each. The private cloud can be used for infrastructure that needs to be deployed privately due to business and regulatory requirements while the public cloud can be used to scale services when demand spikes beyond the private clouds fixed capacity (i.e. cloud bursting). The hybrid deployment model also enables greater cost control as the cheaper option between the public or private cloud can be utilized when appropriate.

2.5.3 Amazon Web Services

Amazon Web Services (AWS) is a public cloud provider offering a large selection of services spanning all three cloud service models. AWS provides extensive enterprise support and is trusted by the cloud community at large to be a market leader. AWS complements their product offering a wide range of SDKs for every possible deployment scenario and provides developer tools such as a CLI. Siemens Energy chose to utilize AWS as their public cloud provider for these aforementioned reasons. Although AWS offers many services, within the scope of the ATaaS the following services were explored: *DynamoDB*, *Simple Storage Service*, *Elastic Compute Cloud* and *Identity Access and Management*.

DynamoDB DynamoDB [4] is a fully managed NoSQL key value database service (PaaS model). Database tables are a collection of items with attributes, which are essentially JSON documents with key value pairs within. Each item needs to have a primary attribute which uniquely identifies it in the database table. The data within an item does not need to be structured and can contain various types of attribute values, such as strings, integers and arrays. The entire database is eventually consistent however some database operations such as read operations can be requested to be strongly consistent.

Simple Storage Service Simple Storage Service (S3) [7] is a fully managed object storage service (PaaS model). An object in S3 is essentially a file of any type which is stored in a bucket. Buckets serve as a collections of objects for the same purpose such as an application or service. Inside a bucket there is no concept of actual folders, instead objects are stored at different paths. Each element in the path from the bucket root until the object name can be considered as a sort of pseudo folder. However, such folders must contain an object eventually. For example, if a pseudo folder contains only one object which then gets deleted, the folder simply disappears along with the path of the object.

Elastic Compute Cloud Elastic Compute Cloud (EC2) [5] is a computing infrastructure service (IaaS model). EC2 offers instances with various OS with different computational capabilities (CPU, memory, storage and networking). The number of EC2 instances can be easily scaled while individual instances can also be scaled vertically. EC2 provides a Virtual Private Cloud (VPC) for all instances and it can be configured with the specific networking functionalities such as VPN tunnels to private corporate clouds. EC2 instances are priced by the hour on demand however for non critical tasks excess EC2 instances can be purchased at significant discounts (spot pricing).

Identity Access and Management Identity Access and Management (IAM) [6] is proprietary secure control access service for AWS services (PaaS model). IAM is used to create users and roles for an AWS account. Permission sets are then assigned to users and roles in order to give them fine tuned access to AWS services such as S3 or DynamoDB. Programmatic access credentials can be generated and assigned to a user, which then allows anyone using those credentials to authenticate requests to AWS services with the same rights as the user. IAM can also be used to assign permissions sets to AWS resources, such as instance profiles which can be attached to an EC2 and would automatically authenticate any AWS request originating from it.

Chapter 3

Architecture

The Analysis Tool as a Service (ATaaS) is a service that provides mechanical engineers the ability to perform finite element analysis (FEA) without needing to install or run any FEA software on their computer. The service implements a standardized way for users to structure, package and perform FEA as tasks. It is built using a distributed microservice architecture that allows the service to be deployed flexibly across different computers and networks. The ATaaS utilizes pools of resources to concurrently execute finite element analyses in order to speed up the computation time for users requesting multiple analyses simultaneously. In the remaining parts of this chapter, Section 3.1 gives an overview of the service architecture, Section 3.2 describes the platform architecture of the prototype ATaaS developed in this thesis, Section 3.3 introduces the taxonomy of a task configuration and Section 3.4 showcases the high level behaviour of a task.

3.1 Service Architecture

The ATaaS is built using three core components; the *Manager*, the *Worker* and the *Executor*. The *Manager* is responsible for serving the ATaaS to users and essentially operates as the primary endpoint for the service. The *Worker* is a node agent in the ATaaS that is responsible for pulling tasks from the ATaaS task queue and launching *Executors*. Consequently, the *Executor* is an ephemeral container which will actually perform the FEA for a task.

There are also external third party services in the ATaaS that the core components depend upon to function correctly, such as the *Task Repository* (as a service), the *File Storage Service* and the *License Management Service*. Each component is built and packaged such that it can be deployed independently of each other and across different environments, enabling the service to be fully distributed in a hybrid deployment. The overall service architecture is represented in Figure 3.1.

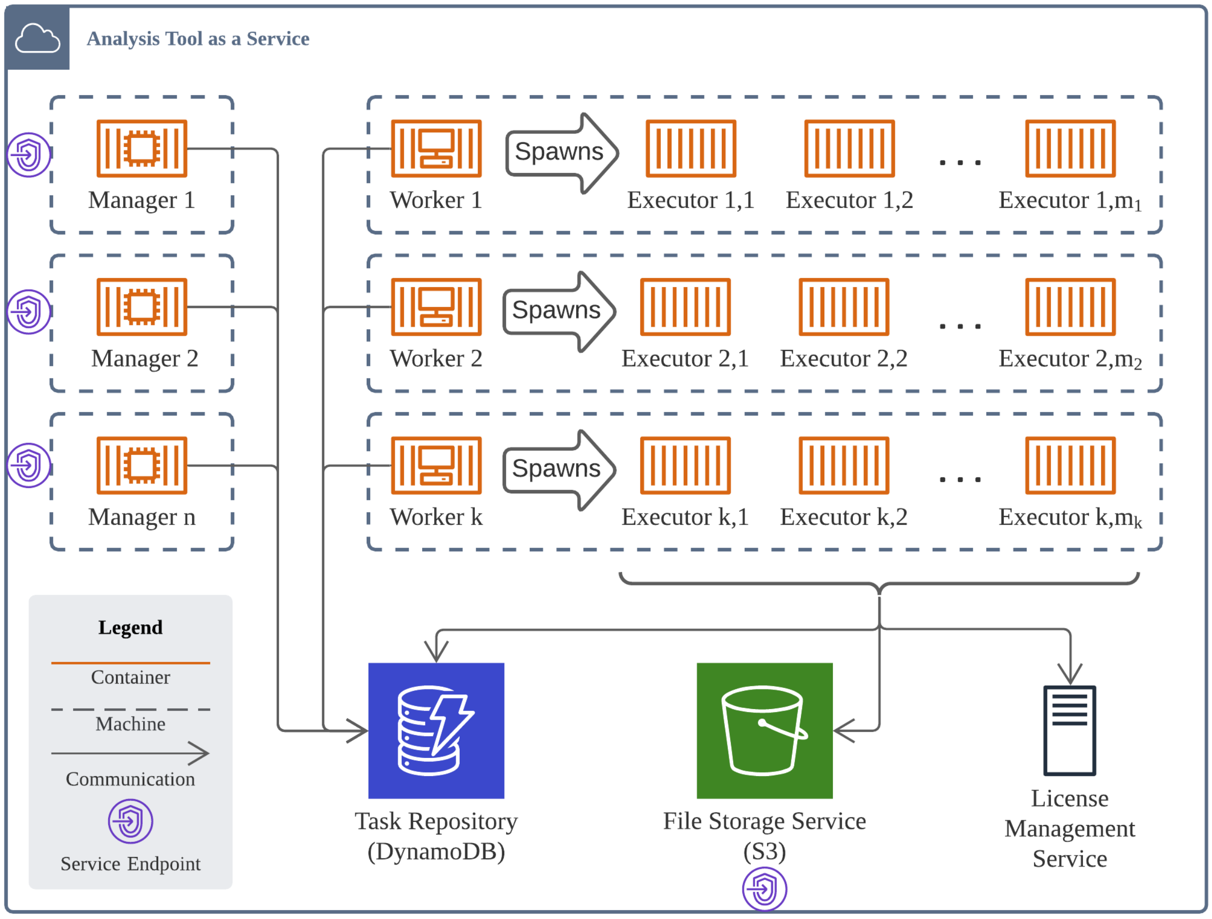


Figure 3.1: General architecture of Analysis Tool as a Service (ATaaS)

3.1.1 Manager

The *Manager* is the primary client facing service endpoint of the ATaaS, allowing engineers to create tasks, queue tasks and get task information. The *Manager* serves an important bounded context in the overall architecture as it focuses only on handling the tasks without actually executing them. The *Manager* exposes an API which the users can then utilize to perform the actions mentioned earlier and explained below:

- **Task Creation**: When a *Manager* receives a task creation request, it generates a globally unique *task identifier (ID)* for the task and then submits the task along with any task configuration parameters passed in by the user to the *Task Repository*. Once the task is successfully saved by the Manager, the *task ID* is returned to the user.

- **Task Queuing**: Once the *Manager* receives a queue request for a task, for which the user should have already uploaded the required task files to the *task folder*, the *Manager* updates the task in the task repository to a queued state, making it eligible for execution.

- **Getting Task**: At any time the *Manager* can respond to task query requests from users. At that point the *Manager* will retrieve the task from the task repository if it exists and return the state, configuration parameters and metadata for the task.

As the primary function of the *Manager* is to serve the ATaaS to users, the *Manager* does not need to directly interact with the other core components of the ATaaS. The *Manager* primarily interacts with the *Task Repository* to create, queue and retrieve tasks. Therefore, the *Manager* is built such that multiple instances of it can be deployed without any single instance interfering in the operation of other instances (horizontal scaling).

3.1.2 Worker

The *Worker* is an agent that runs on a node in the ATaaS, and is responsible for pulling queued tasks from the *Task Repository* and launching and managing *Executor*'s on the node to perform the task analysis. The *Worker* acts within a bounded context as a local task manager, ensuring that tasks are being processed accordingly on one single machine. The *Worker* can be deployed on any host to make it a node in the ATaaS. When a *Worker* is deployed on a node, the number of *Executors* allowed to be launched concurrently on that node can be configured.

Task Checkpointing The *Worker* is responsible for managing any *Executor*'s it launches for tasks it has secured, which entails the following functions:

- **Task Timeout**: The *Worker* must periodically check point each *Executor*. If an *Executor* run time exceeds a maximum threshold, the *Executor* is stopped and the the task it was executing is requeued in the *Task Repository* if it has not reached the requeue limit, otherwise the task is failed.

- **Executor Cleanup**: Once an *Executor* terminates or is terminated for its task, the log files generated by it are uploaded to its respective *task folder* in the *File Storage Service* by the *Worker*. Furthermore the *Worker* updates the end time of the task in the *Task Repository* and confirms that the task state in the *Task Repository* is either completed or failed, if not it marks the task as failed.

Task Launching Launching *Executors* for tasks is the primary function of the *Worker* once it has been deployed deployed, which consists of the following actions:

- **Task Securing**: The *Worker* periodically polls the *Task Repository* for tasks that have been queued and waiting to be executed. The *Worker* then tries to secure a task based on a designated order. Since there is no limit on the number of *Workers* which can be deployed to allow for distributed execution of tasks, the task securing process is designed such that no more than one *Worker* can secure a given queued task from the *Task Repository*.

- **Executor Launching**: Once a *Worker* has secured a task it launches an *Executor* for the secured task, configured with all the configuration parameters from the task.

It also updates the task in the *Task Repository* with a start time and increments the attempt counter.

3.1.3 Executor

The *Executor* is an ephemeral container in the ATaaS which contains the *execution environment* that performs the analysis for a single task. The *Executor* serves the bounded context of task execution in an isolated manner, its solely responsible for ensuring that single task executes successfully. An *Executor* is always launched by a *Worker* for every task that it secures. In order for the *Executor* to function as desired, it is passed the *task ID* and all the configuration parameters for the task it is spawned for upon creation. The *Executor* has three distinct phases in its lifecycle: task setup, task execution and task cleanup.

Task Setup The following actions are performed to setup the *Executor* for a task prior to launching the FEA software for task execution:

1. **Parameter Verification**: The *Executor* performs some checks to ensure that the task parameters were set properly. This entails verifying:

    - The validity of the *task ID*, such that it was passed correctly and that it exists in the *Task Repository*.
    - The existence of an *execution script* at the location specified by the user in the *execution script* configuration parameter or through automatic detection if not specified.
    - The proper *tool version* configuration parameter specification. The *Executor* is packaged with different versions of the FEA software to perform the task analysis. The *Executor* can only use the user specified *tool version* for versions available within it. Moreover, if no *tool version* is specified by the user, a default version is used.

If the *task ID* or the configuration parameters are misconfigured, the *Executor* updates the task in the *Task Repository* to a failed state along with a debug message, and then terminates itself.

2. **Execution Environment Setup**: The *Executor* downloads its *task folder* and user specified *engine models* from the *File Storage Service* into its *execution environment*. All *engine models*, *input folders* and other standard FEA software components are all placed in standardized locations within the *execution environment* for the *execution script* to call upon.

Task Execution Task execution follows the successful completion of the task setup. The *Executor* launches the FEA software to execute the task analysis and the following occurs within the FEA software:

1. **Software Verification**: The FEA software communicates with the *License Management Service* to authenticate itself before executing the task *execution script*.

2. **FEA Analysis**: The FEA software executes the instructions line by line within the *execution script* in order to perform the task analysis. This is the main capability being provided by the ATaaS to its users.

Task Cleanup The *Executor* must cleanup for a task once the FEA software terminates and the task execution is completed, this consists of the following actions:

1. **File Upload**: The *Executor* synchronizes all the *task folder* files from the *execution environment*, or a subset depending on the *output folders* configuration parameter, back to the *task folder* in the *File Storage Service*.

2. **Task Update**: The *Executor* updates the task status to completed in the *Task Repository* along with an end time.

Once all the phases are complete, the *Executor* exits. At this point the *Worker*'s will eventually perform the *Executor* cleanup function.

3.1.4 External Services

Each ATaaS component is designed to operate independently of each other but they all require access to certain external third party services to operate cohesively and communicate. Each of these services were not developed in the scope of this thesis and thus can be considered as black boxes. Moreover, each service can be considered a distributed microservice, in the sense that their function is singular and that they are deployed widely. The three external services within the ATaaS are the *Task Repository*, *File Storage Service* and *License Management Service*, which are elaborated below.

Task Repository The *Task Repository* is the database that contains all the ATaaS tasks. The core ATaaS components can create, update and view tasks in the *Task Repository* as needed. It serves not only to store tasks but also functions as a message passing communication service between each core component in the ATaaS, enabling distributed computing. All the task information along with the task state is stored in the task repository, allowing each component to ensure its properly manipulating each task. However, ATaaS users cannot directly manipulate the *Task Repository* to ensure that each component functions correctly and that tasks are not accidentally corrupted.

File Storage Service The *File Storage Service* is essentially a file server where the ATaaS can upload, download and list engine models and task files within *task folder*'s. The *File Storage service* is responsible for handling and storing files for the ATaaS. Firstly, the ATaaS users can directly interact with the *File Storage Service* as it is the secondary service endpoint of the ATaaS, giving users the ability to upload, download and view their task files within *task folders*. Secondly, the *File Storage Service* is internally used by the *Executor* to populate the *execution environment* and then to return task result files.

License Management Service The *License Management Service* is a service required for the authentication of the FEA software that runs inside every *Executor*. It is built to be deployed in a distributed manner such that each instance of the *License Management Service* is a standalone service that can manage FEA software authentication requests. The *License Management Service* approves any authentication request that can reach the service endpoint from the FEA software as it does not perform any further checks. Ensuring that the FEA software is permitted to operate in a region or whether the appropriate individuals are using the FEA software is not within the scope of the *License Management Service*. To ensure the latter is being respected, the *License Management Service* must only be deployed to approved regions within networks where all the users are permitted to use the FEA software.

3.2 Platform Architecture

The ATaaS is built and deployed using specific technologies to enable a distributed microservice architecture. The three core ATaaS components; the *Manager*, the *Worker* and the *Executor* depend on container technology to operate correctly. The *Task Repository* and *File Storage Service* in the ATaaS are microservices provided by Amazon Web Services (AWS) to handle information and file storage respectively. The *License Management Service* in the ATaaS is a proprietary service provided by Siemens for the FEA software authentication. Finally, all the components in the ATaaS require a specialized networking setup to function together properly and provide functionality.

3.2.1 Container Technology

The *Manager*, *Worker* and *Executor* ATaaS components are designed to be distributed across many computer hosts without interfering with each other and the hosts themselves. Moreover, having the ability to deploy multiple instances of each component is critical to allow the distributed availability of the ATaaS service to users and parallel execution of tasks. To accomplish these requirement, each of the aforementioned components are packaged into containers, enabling their deployment on virtually any available host with a container runtime enabled. With each component packaged into a container, its deployment and operation will not interfere with or modify the host machine in any way other than using the host's resources. Moreover, as each of those component containers do not communicate directly with each other, the ATaaS can be deployed using a hybrid cloud setup. This means that both computer hosts in the public cloud and private cloud can be used in a hybrid setting in the ATaaS.

Manager Container The purpose of each *Manager* is to serve the primary ATaaS endpoint to users, therefore at least a single container deployment of the *Manager* is required for the ATaaS service to be considered available and operational. Furthermore, the *Manager* can be deployed multiple times on different hosts to make the service available in more than one location. This is important as not all users of the ATaaS will be able to access each of the deployed *Manager*s of the service due to network restrictions. Once a *Manager* container is deployed, it has to be exposed to the host's network as required so that ATaaS users within the host's network can reach the ATaaS.

Worker Container The *Worker* is responsible for pulling queued tasks from the task repository and launching *Executors* to perform the task analysis. Consequently, any host on which you deploy a *Worker* container becomes a node in the ATaaS for task execution. At least one *Worker* is required to always be operational for tasks in the ATaaS to be executed. The *Worker* can only spawn *Executor*'s for task execution on the same host it is running on. Therefore, the *Worker* needs to have access to the host container runtime control plane to launch *Executor*'s. It is important to note that the *Executor* containers run alongside the *Worker* on the host container runtime and are not running within the *Worker* container, also known as *container in container*. This access is also used by the *Worker* to manage the number of concurrent *Executor*'s on the host, to timeout long running *Executor*'s and clean up exited *Executor*'s.

Executor Container The *Executor* is an ephemeral container which does the analysis for a single given task and then exits. An *Executor* is always launched by a *Worker* however once the container has been launched it does not require the *Worker* to function as it operates independently. The separation of the *Worker* and the *Executor* allows for fault tolerance, such that even if the *Worker* encounters any issues the *Executor* can complete execution for its given task. The *Executor* is the most computationally demanding container as it performs the actual FEA and therefore must only be deployed on hosts with enough computing power. Multiple *Executor*'s can be launched on the same host without any interference as long as the host machine has enough resources to manage the load.

3.2.2 Amazon Web Services

The *Task Repository* and *File Storage Service* in the ATaaS are external services that are provided by Amazon Web Services (AWS). These external services enable the ATaaS to function without having to implement complex capabilities that are not core to the capability of the ATaaS. Each of these AWS services are accessible via public or private AWS service endpoints using AWS credentials. Furthermore, AWS services are available through numerous software development kits (SDK's) and command line interfaces (CLI's), allowing them to be utilized by the three core ATaaS components and users.

Task Repository AWS DynamoDB is the service utilized for the *Task Repository*. DynamoDB is a distributed microservice offering low latency responses, making it suitable for serving large amounts of requests from many clients. As a NoSQL database is allows the ATaaS to store all the configuration parameters, metadata and state for a task in an unstructured manner while still offering the ability to perform granular operations on the task values. All three core ATaaS components utilize the task repository to operate. This makes DynamoDB the primary channel for indirect communication between the three core ATaaS components, as it allows each component to perform its task functions without having to directly communicate with other ATaaS components.

File Storage Service AWS Simple Storage Service (S3) is the service utilized for the *File Storage Service*. S3 is a highly available global high bandwidth microservice, allowing for the storage of virtually a limitless number of files of any size. This capability makes it ideal to store the *task folder*'s and *engine models* for the ATaaS. S3 is used as the primary system for file storage and transfer in the ATaaS, as it accepts and returns files from users, and provides the stored files to the *Executor* as needed. ATaaS users can directly interact with S3 to manage their *task folder*'s while the *Executor* can rapidly pull *task folder*'s from S3 and push back updated *task folder* files to S3.

3.2.3 License Management Service

The *License Management Service* is a service offered by Siemens Energy to authenticate the proprietary FEA software used by the ATaaS. As the *Executor* is the only ATaaS component that utilizes the FEA software, it becomes critical that the *License Management Service* is accessible to all the *Executor*'s that are launched. The *License Management Service* is not a containerized service which means it must be deployed directly on a host. Regardless, it can be deployed on any host and exposed to any network as long as security and Canadian Export Control requirements are respected. As multiple instances of the *License Management Service* can be deployed, the service can be offered to many different isolated networks where it is required.

3.2.4 Networking

The overall networking architecture required between all the components in the ATaaS to function correctly is shown in Figure 3.2.

The *Manager* can be deployed on any suitable host and network, as long as it is made accessible to the local area network (LAN) of the host so that users within that network can access the ATaaS. The *Worker* can also be deployed on any host and network, provided that it has access to the host's loopback network to launch *Executor*'s on it. Furthermore,

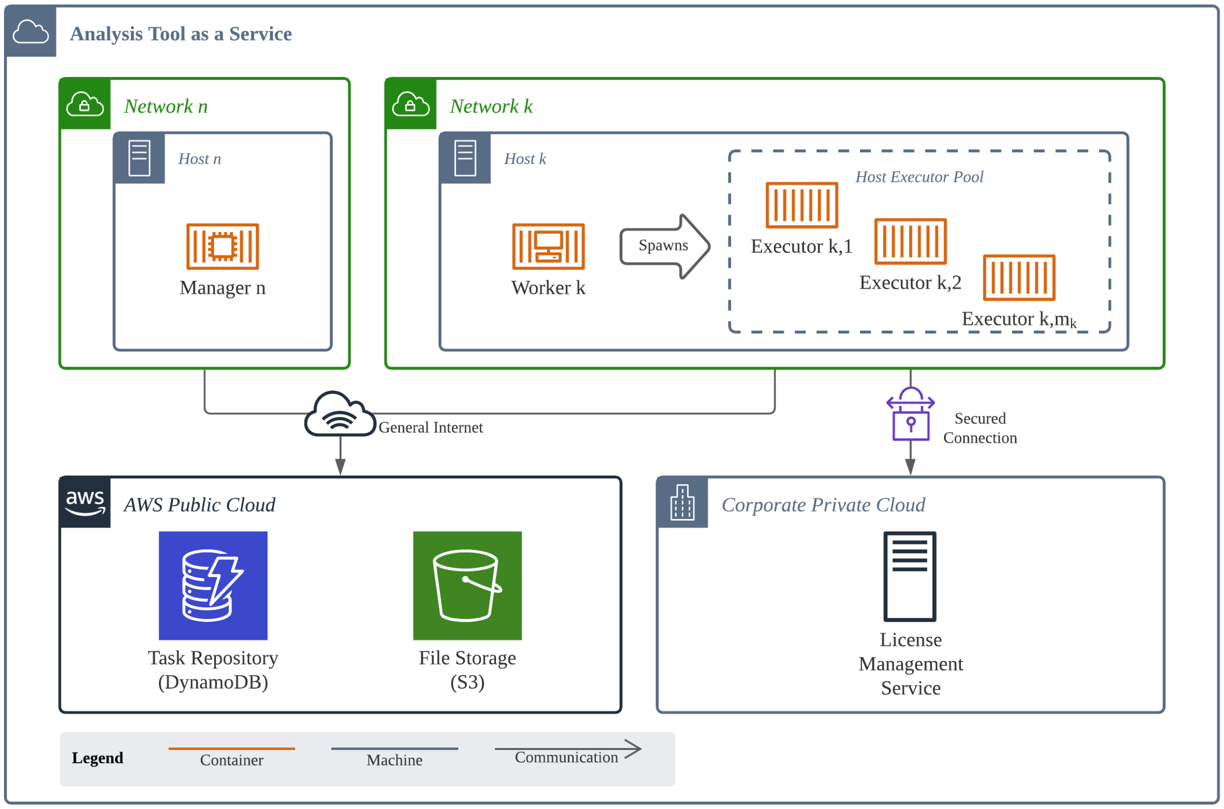


Figure 3.2: Network architecture for Analysis Tool as a Service (ATaaS)

as the *Executor* is always launched by a *Worker* on the same host, the *Executor* itself does not require access to the host loopback network or the LAN of the host.

Since both the *Manager* and *Worker* do not directly communicate, the hosts and consequently their networks used by both can be completely isolated from each other without any issues. However, any host in any network that contains an ATaaS core component requires general internet access to be able to communicate with the public endpoints of AWS in order to access the *Task Repository* and *File Storage Service* external services. Moreover, any host and its network that contains an *Executor* must also have access to the *License Management Service* to authenticate the FEA software that runs within. Due to the deployment and access restrictions of the *License Management Service* mentioned earlier, a secure encrypted connection between the network of any *Executor* and the network where the *License Management Service* is deployed is required.

3.3 Taxonomy

A task configuration includes all the relevant details that need to be specified in order to set up and execute a task in the hybrid cloud based ATaaS environment. The domain model for task configuration is defined by the UML class diagram shown in Figure 3.3.

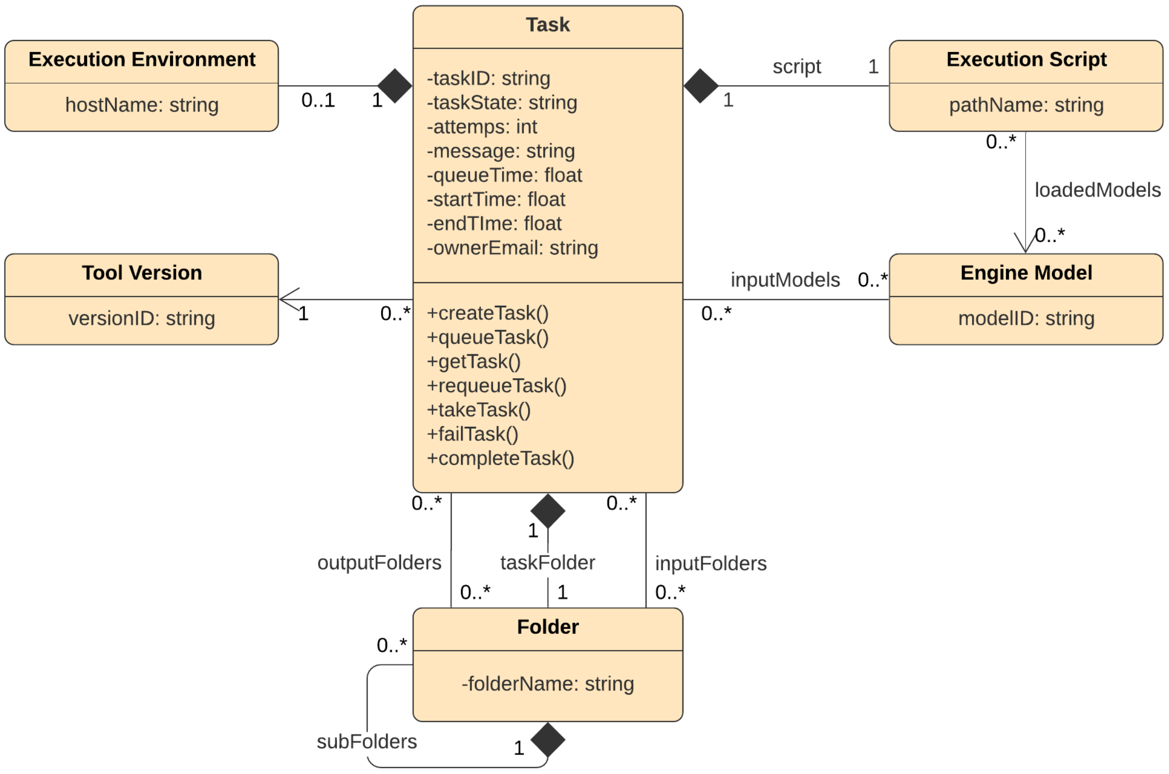


Figure 3.3: Domain model for task configuration

Task In a task configuration, a task is the main resource which users interact with. A task can be defined as a single self contained finite element analysis. Every task must have a *task ID*, which is an ATaaS unique identifier for a task and enables the users and the service to monitor and manipulate a task from creation to completion. Moreover, a task contains service metadata and various configuration parameters which can be set by the user to customize the analysis behaviour. Lastly, a task always has a state which will be further explained in Section 3.4.1.

A task has multiple configuration parameters shown in Figure 3.3 such as *tool version*, *execution script*, *engine model*s and input/output *folder*s, which are discussed subsequently. As further relevant metadata, we also detail the *task folder* and the *execution environment*.

Tool Version The *tool version* configuration parameter for a task controls which version of the FEA software would be used by the *Executor* to perform the finite element analysis. This capability is critical as different versions of the FEA software exist, with each offering a unique set of capabilities which don't completely overlap.

Execution Script The *execution script* configuration parameter for a task is the name and relative location of the execution script within the *task folder*. The *execution script* is essentially a required file that the user creates for the FEA software. During execution, the ATaaS passes the *execution script* to the FEA software which then interprets and executes the commands within to perform the desired FEA for the task. The *execution script* is the main input to the FEA software from which any secondary files present in the *task folder* may be may be loaded into the FEA software, such as *engine models*. The *execution script* has to be prepared with the *execution environment* in mind, such that all paths within should be relative to the *task folder* or point to standard predefined ATaaS *execution environment* resource locations.

Engine Models The *engine models* configuration parameter is an optional parameter and it specifies to the ATaaS which engine models to make available within the *task folder* of an *execution environment*. An engine model represents the mechanical geometry and thermo-mechanical properties of an engine. It can be shared and reused amongst different FEA tasks and generally requires large amounts of storage space. The engine models specified by the *engine models* configuration parameter are always found in a standard location in the *task folder* of an *execution environment* and can then be referenced by the task *execution script* to load them into the FEA software during analysis. Alternatively, users are free to upload engine models to *task folder*'s directly at the *File Storage Service* and subsequently reference them in their *execution script*. However, it is preferred to utilize the *engine models* configuration parameter as it reduces the data transfer and storage overhead.

Input Folders The *input folders* is an optional configuration parameter that specifies to the ATaaS which empty sub folders to create inside the *task folder* of an *execution environment*. This is a critical parameter as empty folders cannot be created inside a *task folder* at the *File Storage Service* and the FEA software cannot write to non existent folders inside the *task folder* of an *execution environment*. Therefore, any *execution script* for a task that would create a new file in a new folder inside the *task folder* of an *execution environment* would necessitate the usage of this configuration parameter.

Output Folders The *output folders* configuration parameter is an optional parameter that specifies to the ATaaS which sub folders from the *task folder* of the *execution environment* to synchronize back to the *File Storage Service* once the analysis has completed. During the analysis of a task, large amounts of data can be generated and not all of it would be relevant to save. By default all new data generated in the *task folder* of an *execution environment* is synchronized back to the *task folder* at the*File Storage Service*. Therefore by specifying this configuration parameter, the user could reduce the data storage and transfer overhead for any large data intensive analysis task.

Task Folder The *task folder* for a task is the folder which contains all the files required and generated by a task, it is an isolated file space for every task. The *task folder* inherits the same name as the task ID of a task. A single *task folder* is available for every task that is created in the ATaaS at the *File Storage Service*. The execution script for a task should always be located in the *task folder* according to the path specified by the *execution script* task configuration parameter. When an *Executor* is spawned for a task, a copy of the *task folder* from the *File Storage Service* is made in the *execution environment*. The *task folder* of an *execution environment* is where the task *input folders* are created and where the task *output folders* should be located as well. Moreover, the same folder inside the *task folder* of an *execution environment* can be treated as an input folder and an output folder. Ultimately when an *Executor* terminates, the *task folder* in its *execution environment* is synchronized back to the *File Storage Service* according to whatever was specified for *output folders*.

Execution Environment The *execution environment* for a task is the container environment of the *Executor* which performs the analysis for a task. A task can only be associated to one *execution environment* at a time as one and only one *Executor* can be spawned for a task simultaneously. The *execution environment* also has an association to the host which runs the *Executor* container. The *execution environment* of a task ultimately contains the actual *input folders*, *output folders*, *execution script*, *engine models*, *tool version* and *task folder*, albeit only while the task is executing as the *Executor* is ephemeral.

3.4 Task Behavior

3.4.1 Task States

The task state for any task is defined and governed by the state machine as shown in Figure 3.4.

A task has a total of 5 possible states which each representing a different point in the lifecycle of a task as detailed below:

- **Created**: The created state indicates that the task has been created within the ATaaS *Task Repository* along with all its configuration parameters, and is ready to accept task files in its *task folder* at the *File Storage Service*.
- **Queued**: The queued state indicates that the task is ready to be be taken by *Worker* for execution, and it expects that all the task files have been uploaded to its *task folder* at the *File Storage Service*.
- **Executing**: The executing state indicates that the task has been secured by a *Worker* and is being executed by an *Executor*.

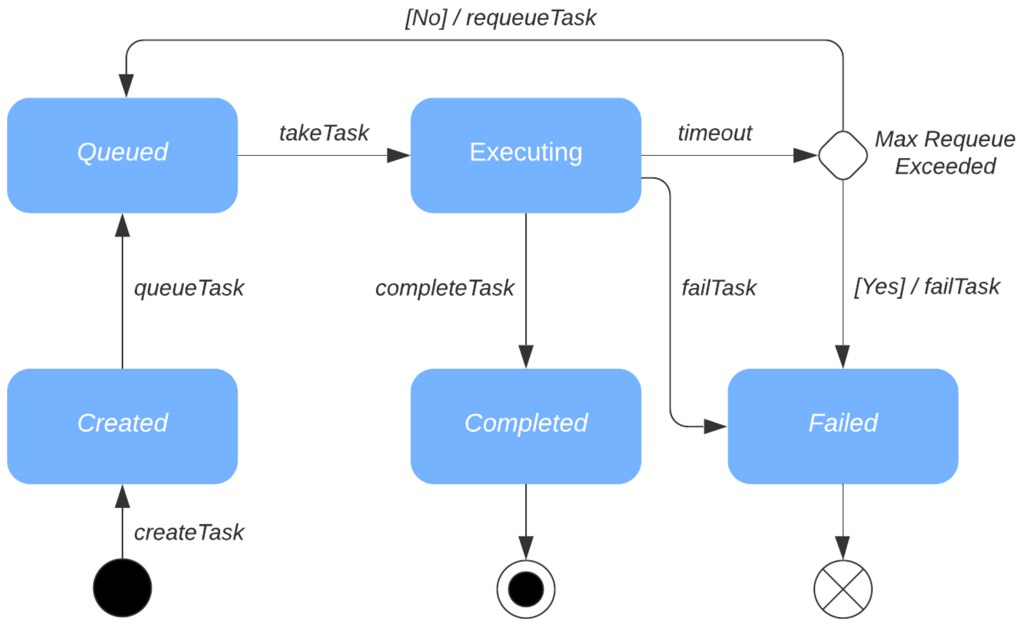


Figure 3.4: UML State Diagram for a Task

- **Completed**: The completed state indicates that the task *execution script* successfully exited, and that the resulting files are available in the *task folder* at the *File Storage Service*.
- **Failed**: The failed state indicates that the task for some reason was not able to complete, the reason for failure is always available.

3.4.2 Task Lifecycle

The manner in which each ATaaS component interacts with a task from task creation to completion is shown by the the following UML communication diagram in Figure 3.5.

A task changes state during its lifecycle due to the user commands and the behaviour of the ATaaS core components, this process is detailed below:

1. A task is created by the *Manager* with the required configuration parameters upon user request and specification.

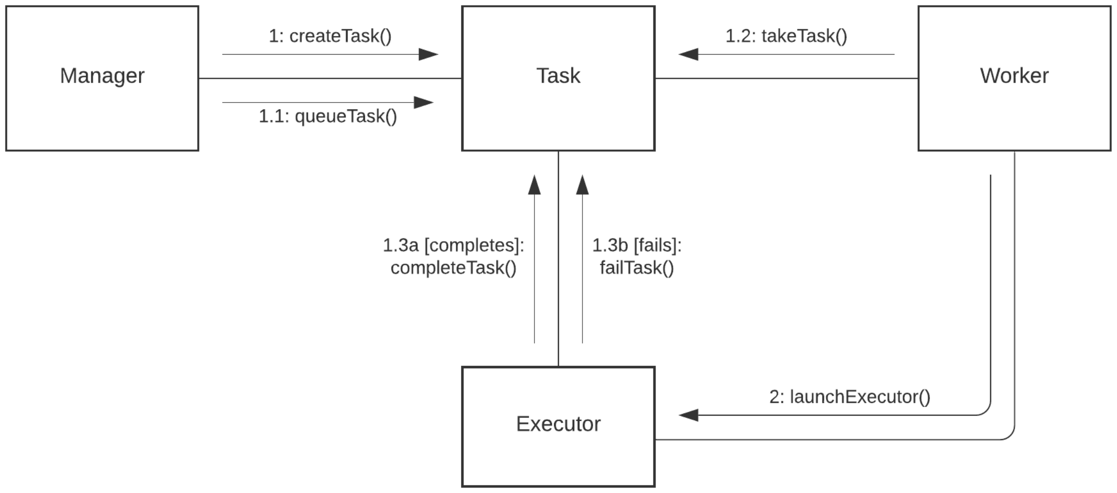


Figure 3.5: UML Communication Diagram for a Task

- When all the necessary task files have been successfully uploaded by the user to the *task folder* at the *File Storage Service*, the task is ready to be queued.

2. A task is queued by the *Manager* upon user request.

   - The *Manager* can only successfully queue a task that is in the created state.
   - Once a task is submitted to the queue it becomes available to the ATaaS to execute, meaning a *Worker* can secure it.
   - User should have uploaded all their task files to the *task folder* at the *File Storage Service* prior to queuing a task.

3. A task is taken by the *Worker* when it has the availability to launch an *Executor* for the task.

   - A task can be requeued by a *Worker* if it reaches a timeout.
   - A task can be failed by a *Worker* if it reaches a timeout and max requeue limit or if an *Executor* unexpectedly terminates without failing or completing the task.

4. A task is completed by the *Executor* when the task analysis completes successfully.

- Users can pull from the *task folder* at the *File Storage Service* all the new files that were generated in the *task folder* of the *execution environment* when the analysis completed.

- Users can pull from the *task folder* at the *File Storage Service* the *Executor* log file that performed the analysis for the task.

5. A task is failed by the *Executor* when the task analysis could not be completed successfully.

   - Task failure occurs when the configuration parameters were not set properly, such as incorrect *tool version* or *execution script* being specified

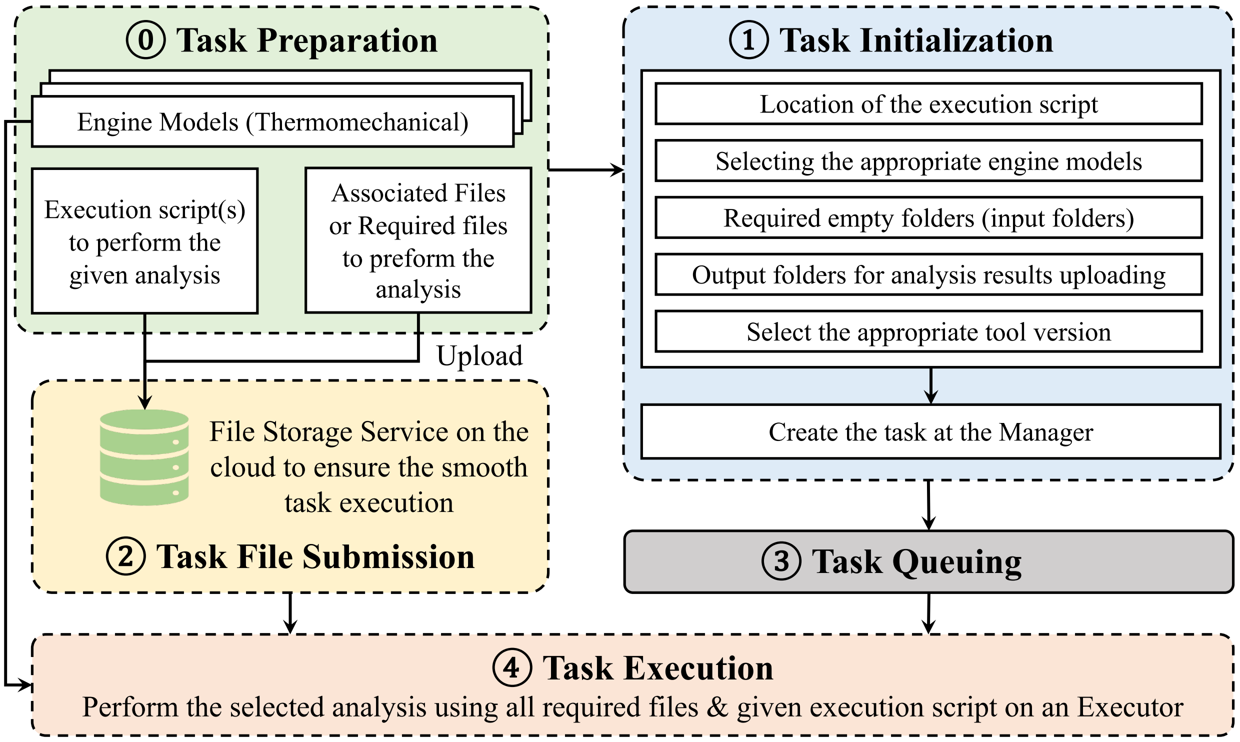
3.4.3 Task Cloudification

The contribution of the ATaaS is to cloudify the legacy FEA tool in five steps; (0) task preparation, (1) task initialization, (2) task file submission, (3) task queuing, (4) task execution with the final step of retrieving the analysis task outputs. This entire process is detailed in relation to the proposed ATaaS architecture in Figure 3.6.

Each step of the cloudification process in Figure 3.6 are detailed below:

0. **Task Preparation**: The user is responsible for preparing all the files for single finite element analysis, most importantly the *execution script*, prior to accessing the ATaaS to create a task. The user also has to identify all the engine models that would be required by the *execution script*.

1. **Task Initialization**: A task is created at the *Manager* and a *task ID* is returned. Various configuration parameters can be set in a task creation request which allow the ATaaS to customize the task execution and analysis per the requirements of the user. For example the engine models required by the execution script should be specified in the task creation request.



⓪ Task Preparation
Engine Models (Thermomechanical)
Execution script(s) to perform the given analysis
Associated Files or Required files to preform the analysis
Upload
① Task Initialization
Location of the execution script
Selecting the appropriate engine models
Required empty folders (input folders)
Output folders for analysis results uploading
Select the appropriate tool version
Create the task at the Manager
File Storage Service on the cloud to ensure the smooth task execution
② Task File Submission
③ Task Queuing
④ Task Execution
Perform the selected analysis using all required files & given execution script on an Executor

Figure 3.6: Overview of cloudification of the proposed ATaaS architecture

2. **Task File Submission**: Once a *task ID* is created and returned to the user in the previous step, they can then upload all the task files they prepared in the task preparation step to its respective *task folder* at the *File Storage Service*.

3. **Task Queuing**: Once all the required task files have been uploaded for a task in the previous step, the task can be queued by the user. At this point the task enters the execution queue and waits for a *Worker* to secure it and launch an *Executor* for it.

4. **Task Execution**: Once a *Worker* has secured the task and launched an *Executor* for it, the FEA can be performed using the task files submitted in the task file submission step and engine models requested in the task creation step.

Task Status Through this entire process users can view the task state and additional metadata throughout the task lifecycle. Task metadata includes the number of attempts made to execute the analysis, the node which secured the task for execution, the execution start and end time for the task and any debug messages that were generated by the ATaaS during the task life cycle. Furthermore, users can also view their *task folder* and files within at any point after a task has been created. Lastly, users need to check the status of their tasks themselves as no task callback functionality is provided by the ATaaS once a task finishes execution.

Task Output Retrieval Once the task analysis has been completed, the task state gets updated from executing to completed or failed. Subsequently, users can then download the task results files from the *task folder* at the *File Storage Service*, including the log file of the *Executor* that performed the task analysis. At this point the task is considered done by the ATaaS and the user has full access to the FEA output of their task.

Chapter 4

Implementation

The Analysis Tool as a Service (ATaaS) is built using a distributed microservice architecture that allows the service to be deployed flexibly across different computers and networks. Users can access the ATaaS through service endpoints to perform finite element analysis (FEA) and are given programmatic tools to make use of the ATaaS more effectively. The core ATaaS components are entirely developed within the scope of this thesis, and primarily makes use of open source technologies to enable code and technology reuse. The external third party services used within the ATaaS are proprietary off the shelf services. The entire ATaaS is designed and deployed in a hybrid cloud environment, making use of the public and the private cloud to leverage the benefits offered by both. In the remaining parts of this chapter, Section 4.1 gives an overview of the ATaaS service endpoints, Section 4.2 describes the implementation of each of the ATaaS components, and Section 4.3 presents the deployment of the prototype ATaaS developed in this thesis.

4.1 Service Endpoints

Users interact through two service endpoints to perform FEA with the ATaaS. The *Manager* is the primary service endpoint which enables the users to create, queue and get

ATaaS tasks. The *File Storage Service* is the secondary service endpoint which allows the users to upload their task files, download their task result files and view their *task folder*'s.

Both service endpoints are required to successfully perform FEA through the ATaaS and provide a RESTful API. In order to effectively make use of the ATaaS service endpoints offered by the *Manager* and the *File Storage Service*, a service script is offered to users that allows them to easily manage ATaaS tasks from creation to completion programmatically on their local computer.

4.1.1 Manager

The *Manager* is the primary service endpoint in the ATaaS, and it provides a web based RESTful API which allows users to create, queue and get tasks within the ATaaS. The REST API implemented by the *Manager* provides three service URI's with specific query and header requirements for each HTTP request. Each successful HTTP request that is returned provides a detailed JSON response body from the *Manager* with all the relevant data. The three service URI's for the REST API are further detailed in Table 4.1.

HTTP Method	**URI**	**Description**
POST	`/create-task`	Create an analysis task
POST	`/queue-task`	Queue an analysis task
GET	`/get-task`	View an analysis task

Table 4.1: ATaaS Manager REST API

It is important to note that the REST API implemented by the Manager is a preliminary design based off the existing workflows at Siemens Energy and hence does not follow some of the best practices of defining resources in REFTful API's. In the future the API should define the task as a resource and then use HTTP methods to interact with the task. For example, creating a new task would be defined as POSTing a task document describing its details to the task resource or getting a task could be GETting the task resource based on its identifier as query parameter.

Create Task The create task API call allows users to create a task in the ATaaS and specify any task configuration parameters they wish to set. Every HTTP request for this call must include a header key *user* with the value being the the email of the requester. The optional task configuration parameters are specified as query parameters in the HTTP request as detailed in Table 4.2.

Key	**Value**	**Configuration Parameter**
`tool`	FEA Software Version	*Tool Version*
`exec`	Execution Script Path & Name	*Execution Script*
`model`	Engine Model Name	*Engine Model*
`input`	Input Folder Name	*Input Folder*
`output`	Output Folder Name	*Output Folder*

Table 4.2: Create Task API call parameters

The response for the create task API call will indicate whether the task was successfully created and if so it will return a *task ID*. If a task could not be created by the *Manager*, the following reasons could be the possible cause:

- In the HTTP request header, the user did not specify the email properly.
- In the HTTP request query, the user did not specify a valid *tool version*.
- The *Manager* encountered a backend error, the user should try again.

Queue Task The queue task API call allows users to queue a task in the ATaaS. This call should only be made once the user has uploaded all their task files to the *task folder* at the *File Storage Service*. Every HTTP request for this call must include a query parameter with the key being *task_id* and the value being the *task ID* of the task the users wishes to queue.

The response for the queue task API call will indicate whether the task was successfully queued or not. If a task could not be queued by the *Manager*, the following reasons could be the possible cause:

- In the HTTP request query, the user did not specify a *task ID*.

- The *task ID* specified does not exist in the ATaaS.
- The task has already been queued and is waiting execution.
- The task is in a state which does not permit queuing; executing, completed, failed.
- The *Manager* encountered a backend error, the user should try again.

Get Task The get task API call allows users to view all the task information for a particular task excluding the task folder and any files within. This call can be made at any time as long as the user has a valid *task ID*. Therefore, every HTTP request for this call must include a query parameter with the key being *task_id* and the value being the *task ID* of the task the user wishes to view.

The response for the get task API call will return all the task information for the requested task. If the *Manager* finds the requested task in the *Task Repository*, the task information returned by this call is detailed in Table 4.3.

If the *Manager* cannot find the requested task in the *Task Repository*, the following reasons could be the possible cause:

- In the HTTP request query, the user did not specify a *task ID*.
- The *task ID* specified does not exist in the ATaaS.
- The *Manager* encountered a backend error, the user should try again.

4.1.2 File Storage Service

The *File Storage Service* is the secondary service endpoint in the ATaaS which allows the users to (1) upload their task files, (2) download their task result files and (3) view their *task folder*'s. The *File Storage Service* is an external third party ATaaS component that relies on the AWS Simple Storage Service (S3) for all of its capabilities. Therefore, to access the *File Storage Service*, ATaaS users have to use AWS authentication and tools.

Key	Value	Explanation
`state`	Current task state	One of the possible task states
`owner`	User email	The task owner's email
`attempt`	Number of execution attempts	The number of task execution attempts so far
`tool`	FEA Software Version	*Execution script* configuration parameter if specified
`exec`	Execution Script Path & Name	*Execution script* configuration parameter if available
`model`	Engine Models List	*Engine Models* configuration parameter if specified
`input`	Input folders list	*Input Folders* configuration parameter if specified
`output`	Output folders list	*Output Folders* configuration parameter if specified
`task_message`	Message	Task status message if available
`host`	Hostname	The most recent task *Executor* host if applicable
`queue_time`	Queue Time	The most recent task queue time if applicable
`start_time`	Start time	The most recent *Executor* start time if applicable
`end_time`	End Time	The most recent *Executor* end time if applicable

Table 4.3: Get Task API call response

Service Use Cases

Every use case of the *File Storage Service* detailed below is always in relation to a task. For every task that is created in the ATaaS, a *task folder* in the *File Storage Service* with the same name as the *task ID* is available. Therefore to perform any *File Storage Service* operation a *task ID* is required in the request in order to manipulate the correct *task folder*.

Upload Task Files Users upload task files to the *task folder* in order to have those files available to the *Executor* that performs the task analysis. Task files can be uploaded to the *task folder* at any point. However, all the task files should be uploaded to the *task folder* before the task is queued in order to ensure that all the required task files are available to the task *Executor* when it launches. In order for a task to succeed it is required that the user upload at least an *execution script* to the *task folder*, otherwise the task will fail.

Task files can be uploaded to the *task folder* individually by name. However, it is recommended that the user creates a local task folder on their computer and then upload the entire folder's contents to the *task folder* using the copy folder command offered by the *File Storage Service*. The copy folder command takes all the files available within a local folder and uploads them to the specified *task folder*. If used more than once, the copy folder command will overwrite existing files in the *task folder* with the same name.

Download Task Result Files Users download task result files (or any task file) from the *task folder* in order to obtain the results from their task execution. Task results files become available in the *task folder* only once they have been uploaded by the *Executor* to the *task folder* upon the successful completion of the task analysis. Therefore, users should only expect to download their task result files when the task state becomes completed. In addition to the task result files, the log file of the *Executor* that performed the task analysis is also available in the *task folder* of any completed task.

Any task result file in a *task folder* can be individually downloaded from the *task folder* by file name. However, it is recommended to use the folder sync command offered by the *File Storage Service*. This command will synchronize the contents of a specified *task folder* to any local folder, preferably the local task folder that was prepared earlier for the task file uploading. There are two primary advantages to using the folder sync command:

- The user does not need specify the task result files by name. Otherwise the user would have to view the contents of the *task folder* and then download the desired task result files individually.
- All the files from the *task folder* that do not exist in the local target folder or that are an older version will be downloaded. This ensures that all the task result files are downloaded without incurring the overhead of transferring files that already exist locally from when the task files were initially uploaded.

The copy folder command can also be used to download task result files from the *task folder* to any local folder. This action will download all the task files from the *task folder* into a target local folder, overwriting any existing files with the same name in the target local folder even if they are current.

View Task Folder Users can view the contents of any task folder and sub folders within in order to verify that task files have uploaded successfully and to inspect which task files exist within the *task folder*, including any task result files. The *task folder* and its sub folders can be viewed during any stage of the task lifecycle at the *File Storage Service*. The list folder command should be used to view the contents of a *task folder* and its sub folders. This command provides users a visual representation of their *task folder* files and sub folders, including the size, and upload date and time of every object within.

Service Access

The *File Storage Service* is accessible via the public internet as an AWS service, and therefore requires that all requests it receives are authenticated using AWS credentials. ATaaS users are manually assigned and given AWS credentials upon request to use the *File Storage Service*. These credentials would specifically allow them programmatic access to task folders in the *File Storage Service*, allowing them to upload, download and view task folders and files. However, it is important to note that AWS credentials with programmatic access do not allow users to access the public AWS web page.

The *File Storage Service* can be accessed through virtually every platform and system because of the numerous SDK's offered by AWS for their services such as S3. Due to the programmatic credentials given to ATaaS users, they cannot access the *File Storage Service* through the AWS web page and must do so programmatically using an SDK. There is a REST API available for S3 but users can take advantage of the AWS command line interface (CLI). The AWS CLI allows users to authenticate once and then use the CLI on their local computer terminal to access AWS services such as S3 for the *File Storage Service*.

4.1.3 Endpoint Usage

Direct Access

The RESTful API provided by the *Manager* and the *File Storage Service* is accessible to anyone who can reach the service endpoints and make valid HTTP requests according to the API requirements. Moreover, there is no web page available for the *Manager* service endpoint, and the service endpoint web page for the *File Storage Service* is not accessible using programmatic credentials. Therefore, users would have to pick and use a tool of their choice to manually make the HTTP requests to the service endpoints in order to perform FEA analysis with the ATaaS API.

Service Script

In order to facilitate ATaaS usage, we developed an ATaaS service script that would allow users to create and launch multiple ATaaS tasks from their local machine without having to manually access the ATaaS service endpoints for each task and make the required API calls. The service script has the following local machine prerequisites in order to be used:

- The script must be launched on a Windows machine as it is built using the Powershell framework.
- The AWS CLI must be installed on the machine and already authenticated using AWS programmatic credentials that allow access to the *File Storage Service*.

Script Setup In order to use the ATaaS service script, a user has to locally prepare a batch analysis folder that will contain one sub folder for each task the user wants to launch. Inside each task sub folder, the user will place all the task files required for the analysis of the task. Furthermore, a standardized JSON task configuration file must be created within each task sub folder. Each JSON task configuration file can contain the following data representing the task configuration parameters previously explained in Section 3.3:

Key	Value	Format
`tool`	FEA Software Version	String
`exec`	Execution Script Path and Name	String
`model`	Engine Model Names	String Array
`input`	Input Folder Names	String Array
`output`	Output Folder Names	String Array

Table 4.4: Local task folder JSON configuration file

Script Usage Once the local batch analysis folder has been prepared according to the script setup requirement detailed earlier, the user can launch the service script on their local machine PowerShell terminal. The service script should be launched from the local batch analysis folder and must be passed the user's email, and optionally the name of the task configuration file and the *Manager* endpoint IP address.

Script Behaviour Upon launching, the service script performs the following actions for each task sub folder found in the local batch analysis folder in order to create, prepare and queue ATaaS tasks:

1. **Task Creation**: The task configuration file that should be found in the task folder is parsed. Subsequently a task creation HTTP request is sent to the *Manager* endpoint with the appropriate task configuration parameters set as HTTP request query parameters depending on the parsed contents of the task configuration file. The *task ID* is outputted to the user if the task creation request is successful. If no task configuration file is found in the task sub folder or the task creation request fails then the task sub folder is ignored and skipped.

2. **Task File Uploading**: With the *task ID* returned in the previous step, all the folders and files within the task sub folder are uploaded to the *task folder* in the *File Storage Service*. The success and failure of each task file is outputted to the user.

3. **Task Queuing**: With all the task folders and files uploaded, the task is queued. The success or failure of this operation is outputted to the user.

Once the service script completes executing it is up to the user to periodically poll the *Manager* endpoint with the *task ID*'s outputted by the script to see whether the newly created and queued tasks have started executing or completed or failed.

4.2 Service Components

The Analysis Tool as a Service is built using three core components; the *Manager*, the *Worker* and the *Executor*. Each core component is entirely developed within the scope of this thesis and is primarily built using open source technologies. The *Manager* is a web server that handles user requests, the *Worker* is a web server that serves as a node agent and the *Executor* is a script that handles the analysis of one task.

The *Task Repository*, the *File Storage Service* and the *License Management Service* are the external services within the ATaaS. Each is implemented by proprietary off the shelf services; the *Task repository* is provided by AWS DynamoDB, the *File Storage Service* is provided by AWS S3 and *License Management Service* is provided by Siemens Energy.

Each core component works together with the *Task Repository* to ensure that any task state transition respects the task state machine model. This entails that no component will modify the state of any task in the *Task Repository* without verifying if that action will violate the task state machine model. Furthermore each core component utilizes UTC whenever dealing with time based operations. Lastly, the *File Storage Service* essentially provides the file storage and delivery functionality to the ATaaS. It allows the users and core components to efficiently move files from task creation to completion.

4.2.1 Manager

The *Manager* is the primary service endpoint of the ATaaS and therefore implements a Python Flask web server to serve the RESTful API of the ATaaS. The *Manager* does not implement any security for the API and therefore anyone who can reach the service endpoint has full access to the API. Whenever a properly formatted API call it made to the

Manager a 200 HTTP code response is always given with a JSON payload, irrespective of if the API call was actually successful or not.

Task Creation When the *Manager* receives a task creation request the following actions are performed as part of the task creation routine:

1. It is verified if an owner email was specified in the HTTP request header.
2. It is verified that if a tool version was specified as a HTTP query paramater, whether it is a valid tool version available within *Executor*'s.
3. All the configuration parameters are parsed from the HTTP query parameters.
4. A new *task ID* is created by joining the current date and time with a new UUID.
5. The *Task Repository* is queried with the new *task ID* to ensure it does not exist already, even though this is highly unlikely.
6. A task is created in the *Task Repository* with the new *task ID* as its identifier, the configuration parameters and owner specified in the HTTP request are included as well, and finally the new task is assigned the created state.
7. The new *task ID* is returned to the user after the successful completion of the above steps, otherwise an error message is returned.

Task Queuing When the *Manager* receives a task queuing request the following actions are performed as part of the task queuing routine:

1. It is verified if a *task ID* was specified in the HTTP request as a request query parameter.
2. The *Task Repository* is queried with the requested *task ID* to ensure that the task requested to be queued actually exists.

3. With the task query results from the previous step it is verified if the task is in the created state, if not an error message is returned.

4. The task is updated to the queued state along with a queue time in the *Task Repository* with the condition that the task exists and was in the created state.

5. A success message is returned to the user after the successful completion of the above steps, otherwise an error message is returned.

Getting Task When the *Manager* receives a get task request the following actions are performed as part of the getting task routine:

1. It is verified if a *task ID* was specified in the HTTP request as a request query parameter.

2. The *Task Repository* is queried with the requested *task ID*.

3. With the task query result from the previous step it is verified if the task exists, if not an error message is returned.

4. If the task query result returns a task then a success message is returned to the user along with all the task information; including the task configuration parameters, the task state and additional metadata.

4.2.2 Worker

The *Worker* implements a Python Flask web server and operates as a node agent for the ATaaS. A node in the ATaaS is any host on which the *Worker* is deployed as an agent so that it can launch *Executor*'s which perform analysis for tasks. The *Worker* needs access to the host container engine API in order to launch and manage *Executors*. On deployment of the *Worker* the maximum number of concurrent *Executor*'s (*Concurrency*) allowed on the host can be configured. *Concurrency* should set depending on the computational power of the node as deploying too many *Executor*'s on a single node will slow down the execution time for every *Executor*. The *Worker* has a periodic update routine which first checkpoints all the *Executor*'s and then launches new *Executor* as necessary. At the beginning of every update routine the concurrent *Executor* counter is reset.

Task Checkpointing

The task checkpointing process discovers all the *Executor*'s available on the host. For each *Executor* found, the *Worker* extracts the *task ID* it is executing for. The *Worker* subsequently queries the *Task Repository* for that particular *task ID* and caches the task information returned. If the *Executor* is still running, the *Worker* runs the task timeout subroutine. Otherwise, if the *Executor* has exited the *Worker* runs the *Executor* cleanup subroutine.

Task Timeout The *Worker* determines how long an *Executor* has been running for by comparing the start time of the *Executor* with the current time. If the *Executor*'s runtime is less than the maximum runtime allowed for tasks in the AtaaS, the *Executor* is allowed to continue running and the concurrent *Executor* counter is incremented. Otherwise the *Executor* is stopped, the end time is recorded and the following task timeout cleanup actions are performed:

1. The *Executor* logs are uploaded to the *task folder* of the task it is executing for at the *File Storage Service*.
2. The number of execution attempts mode on the task is extracted from the task query results cached earlier.
3. The task is updated in the *Task Repository* as following:
    - If the number of task attempts exceeds the maximum number of attempts allowed for a task, the task is updated in the *Task Repository* to a failed state along with a task message and the end time.

- If the number of task attempts is below the maximum number of attempts allowed for a task, the task is updated in the *Task Repository* to a queued state along with a task message and the end time.
- A task is updated in the *Task Repository* to the failed or completed state only if its current state was running in the *Task Repository*.
- In the unlikely event that the task state is not running in the *Task Repository*, the task is failed along with a special state violation task message.

4. The *Executor* is removed from the container engine of the machine.

Executor Cleanup The *Worker* needs to clean up the exited *Executor* from the host in order allow for new *Executor*'s to be launched. The *Executor* cleanup routine consists of performing the following actions:

1. The *Executor* logs are uploaded to the *task folder* of the task it is executing for at the *File Storage Service*.
2. The task state is extracted from the task query results cached earlier.
3. The task state is verified to ensure that the *Executor* updated the task in the *Task Repository* to a failed or completed state before exiting.
4. In the unlikely event that the verification in the above step fails, the following steps are taken:
    - The current time is recorded to serve as the end time for the task.
    - The task is failed in the *Task Repository* along with a special *Executor* failure task message and end time.
5. The *Executor* is removed from the container engine of the machine.

Task Launching

The task launching process consecutively secures a task in the queued state from the *Task Repository* and then launches an *Executor* for the secured task until the number of concurrent *Executor* on the node reaches the maximum value set during the deployment of *Worker* on the host. If there are no available tasks in the queued state in the *Task Repository* before the number of concurrent *Executors* reaches the maximum value, the task launching process exits until the next periodic invocation. The tasks securing subroutine consists of securing a task for execution from the *Task Repository* and the subsequent *Executor* launching subroutine consists of launching an *Executor* for the secured task.

Task Securing During the task securing subroutine the *Worker* scans the *Task Repository* for all tasks that are in the queued state. The query result list is then organized from oldest to newest queue time. The *Worker* then tries to secure the oldest queued task in order to follow a first in first out principle for task execution. The following actions are performed repeatedly until a task has been secured:

1. In the *Task Repository* the state of the oldest queued task from the query result list is updated to running, the task host is set to the host name where the *Worker* is deployed, the task start time is set to the current time and the task attempt number is incremented.

2. A strongly consistent query is performed on the *Task Repository* with the *task ID* of the oldest queued task from the query result list to ensure that the eventually consistent update call was successful, the task query results are cached.

3. The oldest queued task is removed from the query result list.

4. If the task query result returned different values from the update task values, another *Worker* has secured the task.

5. If the task query result returned the same values as the update task values, the task has been successfully secured by the *Worker*.

Executor Launching During the *Executor* launching subroutine the *Worker* launches an *Executor* for the task it has secured. The *Executor* launching subroutine consists of the following actions:

1. The task configuration parameters are extracted from the task query result cached earlier.
2. Using the container engine API, the *Executor* is launched for the task and is passed the *task ID* and the task configuration parameters.
3. The concurrent *Executor* counter is incremented.

4.2.3 Executor

The *Executor* is essentially a script which runs inside a container and performs the analysis for a single task. When the script inside the *Executor* exits, the *Executor* terminates as well. The *Executor* is designed to be ephemeral and resilient to failure, such that the task it is executing for can be recovered and rerun by the AtaaS in case the *Executor* terminates unexpectedly. On launch the *Executor* takes a *task ID* as the primary required input and optionally the following configuration parameters; *tool version*, *execution script*, *engine models*, *input folders* and *output folders*. The *Executor* assumes that the task it is being launched for is already in the running state and only makes task fail or complete calls on that condition being verified. The following actions are performed by the script in the *Executor* after launching for a task analysis:

1. It is verified if the *task ID* was inputted, otherwise the script exits prematurely.
2. Downloads the *task folder* of the executing task from the *File Storage Service* into the *execution environment*.

3. If the *engine models* configuration parameter was specified, the requested *engine models* are downloaded into a sub folder in the *task folder* of the *execution environment*.

4. If the *input folders* configuration parameter was specified, the requested *input folders* are created as sub folders in the *task folder* of the *execution environment*.

5. The *execution script* is identified:

   - If the *execution script* configuration parameter was specified, the specified path and name of it is verified in the *task folder* of the *execution environment*.
   - If the *execution script* configuration parameter was not specified, it is automatically detected in the *task folder* of the *execution environment*.
   - If the *execution script* cannot be identified in the *task folder* of the *execution environment*, the *Executor*'s task is failed in the *Task Repository* along with a task message and the end time, and subsequently the script exits.

6. The FEA *tool version* to use for analysis is identified:

   - If the *tool version* configuration parameter was specified, it is verified if the requested *tool version* is available within the *Executor* for analysis.
   - If the *tool version* configuration parameter was not specified a default *tool version* is set to be used for analysis.
   - If the *tool version* configuration parameter that was specified is not available within the *Executor*, the *Executor*'s task is failed in the *Task Repository* along with a task message and the end time, and subsequently the script exits.

7. The FEA software of the *tool version* identified earlier is launched and is passed the *execution script* as input.

8. The FEA software log file and select files from the *task folder* in the *execution environment* are uploaded to the *task folder* at the *File Storage Service* once the FEA software finishes executing:

- If the *output folders* configuration parameter was specified, only the requested *output folders* within the *task folder* of the *execution environment* are uploaded.
- If the *output folders* configuration parameter was not specified, all the new and modified files within the *task folder* of the *execution environment* are uploaded.

9. The *Executor*'s task is updated in the *Task Repository* to a completed state along with the end time and the script exits successfully.

Once an *Executor* terminates it remains in an exited state on the host it ran on until the host *Worker* is able to clean up the *Executor* in order to free up resources for new *Executor*'s.

4.2.4 External Services

The *Task Repository*, the *File Storage Service* and the *License Management* are the external third party services used to provide critical functionality to the ATaaS. The first two services are provided by AWS DynamoDB and AWS S3 respectively while the last service is provided by Siemens. Each external service is used in a particular manner to enable it to function as desired within the ATaaS.

Task Repository

The *Task Repository* is provided by AWS DynamoDB, where each ATaaS task is stored into the *Task Repository* as an item and is identified by its *task ID* as the primary attribute of the item. Additionally, all the metadata, the configuration parameters and the state for each task is stored into the task item as attributes in the *Task Repository*. The *Task Repository* is accessed by all the core ATaaS components, effectively serving as the primary communication channel between all of them. There are four primary functions that the *Task Repository* provides for ATaaS tasks; create task item, update task item, query task item and scan task items.

Create Task Item The create task item function creates an item for a task in the *Task Repository*. The task item must be assigned a *task ID* which will serve as the primary attribute of the item. All the task data available at creation such as the task configuration parameters can be included as attributes. Moreover the task state, which is an attribute that is always required to exist in a task item, is set to created. If the create task item function is called for a task with a *task ID* that already exists in the *Task Repository*, the existing task item is overwritten by this call.

Update Task Item The update task item function updates the data within a task item in the *Task Repository*. In the function call it can be specified which task item attributes besides the primary attribute to update and how to do it. The task item attributes are updated as following:

- Updating the value for a key that did not previously exist consists of adding the key value pair to the item, such as when a *Worker* secures a task and assigns a start time value to it.

- Updating the value for a key that did previously exist consists of simply updating the value for the key with what was specified in the function call, such as when the task state is updated.

- Update the value for a key value which is a integer can consist of perform increments on the previous value, such as when the number of attempt for a task is updated by the *Worker* which secures it for execution.

Finally the update task item function call can contain a condition which determines whether to perform the update operation. The condition can be based on existing attributes in the task item and uses common logical operands such as *and*, *or*, *equal to*, etc. Update task conditions are primarily used within the ATaaS to ensure that update task item calls are not violating the task state machine model when updating the task state.

Query Task Item The query task item function retrieves an entire task item from the *Task Repository* for the *task ID* requested. It can be requested that the task query result be strongly consistent. Therefore the query task item function can be called right after using the update task item function to verify if the update call was successful. This functionality is important as it is used by the *Worker* in the task securing process detailed earlier.

Scan Task Items The scan task items function retrieves all the task items from the *Task Repository*. It can be requested that the results be strongly consistent and be filtered according to attribute values. A filter is essentially a condition that each task in the scan must satisfy before being included in the results sent back to the requester. The filters are very useful to find particular types of task items in the *Task Repository*, such as when the *Worker* scans for only the queued tasks during the task securing process.

File Storage Service

The *File Storage Service* is provided by AWS S3, where all the ATaaS task files are stored in the *File Storage Service* as objects at different paths in the same bucket. The path for each task file starts at the bucket root, second comes the *task ID* which essentially becomes the *task folder*, then any sub folders if needed and finally the file name. All the *engine models* are also stored in the *File Storage Service* in a single folder of one bucket. The *File Storage Service* is accessible and used by both the ATaaS users and components as detailed earlier and serves to enable file storage and delivery.

There are three primary functions that the *File Storage Service* provides to the ATaaS; upload files to a folder, downloading files from a folder and viewing files at a folder. There are various different AWS API calls and CLI commands that S3 offers to accomplish the three different functions provided by the *File Storage Service*. However in practice only three AWS S3 CLI commands are used; copy, sync and list:

- **Copy** The copy command copies files from a local source to a s3 path or vice versa. It can be configured to target a single file, all the files or specific types of files in a given folder. For example, this command is used by the *Executor* to download all the task files from a *task folder* in S3 into its *execution environment* at the beginning of a task analysis.

- **Sync** The sync command synchronizes all the files from a local source folder to a s3 path or vice versa and can be configured to target specific types of files. For example, this command can be used by the *Executor* to upload any new or modified files from the *task folder* in the *execution environment* to the *task folder* in S3 at the end of a task analysis.

- **List** The list command displays all the folders and files along with their size and creation time at a particular path. For example, this command can be used by an ATaaS user to view the contents of their *task folder*.

S3 can be accessed through a diverse set of API calls and CLI commands which are highly customizable. Therefore the above is not an exhaustive list of all the S3 calls which can be used in order to accomplish the functions of the *File Storage Service* within the ATaaS, rather the ones which were used in our implementation of the ATaaS.

License Management Service

The *License Management Service* is provided by Siemens, which is a fully managed software activation service. The legacy FEA software used by the *Executor* to perform task analysis is a proprietary software that must be activated with a license from the *License Management Service* on every invocation. The *License Management Service* communicates with the FEA software using IP protocols and two different ports. The first port is used to establish a connection from the FEA software and the second port is used to transmit a license back. The *License Management Service* treats any license request that it receives as valid and does not perform any additional checks such as verifying if the software is being used in the correct region or if an authorized users is invoking it.

4.3 Deployment Platform

The ATaaS is designed using a distributed microservice architecture and deployed in a hybrid cloud environment with heterogeneous computers, networks and services. This entails making use of both the public and the private cloud to leverage the benefits offered by both. The first step to enabling a hybrid deployment was containerizing the core ATaaS components so that they could be deployed on any host. The second step was making use of the public cloud Amazon Web Services to offer fully managed specialized services to the ATaaS. The third step was strategically deploying the *License Management Service* to fulfill the requirements of the ATaaS and the FEA software. The last and final step was implementing a robust networking strategy that enabled all the ATaaS components to function cohesively while ensuring operational security and resilience for the service.

4.3.1 Containerization

The core ATaaS components are all built and packaged into Docker containers as using the Docker container engine allowed for the ATaaS to depend on a well developed and supported container technology. This is evidenced by the Docker having extensive Windows container support which is critical as the FEA software used by the *Executor* is only available on Windows. Utilizing containers for the core components allows for their deployment to be flexible, enabling any computer to host a core component of the service.

The base ATaaS container image for every core component of the ATaaS begins from a base windows image, which results in the ATaaS core components only being deployable on Windows hosts. Subsequently, every ATaaS container image has the AWS CLI installed on it, and the IAM credentials for accessing AWS services are embedded into the container as environment variables. Furthermore, each core component further builds on top of the base ATaaS container image and utilizes the container technology in a tailored way to effectively deliver its functionality.

Manager

The *Manager* is a Python Flask web server packaged into a container, which is accomplished through the following steps:

1. Installing Python and Flask into the base ATaaS container image.
2. Copying the web server source code into the container image.
3. Loading the python libraries required by the web server into the container image.
4. Exposing the container image on the web server port.
5. Designate launching the web server as the entry point of the container image.

When launching the *Manager*, the web server container port must be mapped to a host port. As long as the *Manager* container is active the web server within will run and serve as a primary endpoint of the ATaaS. Multiple *Manager* containers can be launched without interfering with each other. Therefore each *Manager* instance serves an independent primary endpoint of the ATaaS, this allows the ATaaS primary endpoint to be deployed and available on many different hosts ad networks.

Worker

The *Worker* is a Python Flask web server packaged into a container, which is accomplished through the following steps:

1. Installing Python and Flask into the base ATaaS container image.
2. Copying the web server source code into the container image.
3. Loading the python libraries required by the web server into the container image.
4. Designate launching the web server as the entry point of the container image.

Unlike the *Manager*, the web server port does not need to be exposed and mapped to the host. This is because the web server inside the *Worker* periodically and internally calls the update routine and does not serve any requests. As long as the *Worker* container is active, the web server within will launch and manage task *Executor*'s on the host. Subsequently, in order for the *Worker* to launch and manage *Executor*'s, the *Worker* need to have access to the host's Docker API. This must be done by configuring the host's docker engine to expose the API endpoint to the isolated network where the containers operate.

When launching the *Worker* container, the host's Docker API endpoint and the number of *Executor*'s the *Worker* is allowed to launch concurrently can be specified as container parameters which would be passed through to the web server. If the host's Docker API endpoint is not specified the *Worker* will assume it is available at the default gateway of the host container network. Lastly, multiple *Worker*'s can be launched overall however only one *Worker* per host should be deployed as the *Worker* operates as a node agent.

Executor

The *Executor* is a PowerShell script which performs the task analysis with FEA software inside a container, this is accomplished through the following steps:

1. Copying every version of the FEA software and resources into the base ATaaS container image.

2. Installing every version of the FEA software inside the container image.

3. Setting the *License Management Service* endpoint for the FEA software as an environment variable.

4. Copying the PowerShell script into the container image.

5. Designating the PowerShell script as the entry point of the container image.

When launching the *Executor* container, the *task ID* and the task configuration parameters can be specified as parameters which would be passed through to the PowerShell script. The *Executor* containers are ephemeral and are designed to terminate once the execution for the task has been completed.

4.3.2 Amazon Web Services

AWS DynamoDB, S3, EC2 and IAM services are used by the ATaaS to deliver its service without having to implement every functionality required. As explained earlier the *Task Repository* was provided by DynamoDB and the *File Storage Service* was provided by S3. Moreover, EC2 was used to provide computer hosts that would serve as nodes in the ATaaS and IAM was used to generate access credentials for the AWS services. The deployment and configuration of each of these services within the context of the ATaaS is further detailed below.

DynamoDB AWS NoSQL database DynamoDB was used to provide the *Task Repository* as a service to the ATaaS. It was configured to use *task ID*'s as the primary attribute of any item within. The service was provisioned be on demand such that it would scale automatically to service any number of read or write requests. In order to respect the geographic restrictions of IT infrastructure deployments within Siemens, it was not replicated across different regions in the world. The database was made to be accessible through the general internet as long as the requests were authenticated with the proper IAM credentials.

S3 AWS S3 was used to provide the *File Storage Service* to the ATaaS. Two buckets were created for the ATaaS, one for all the task folders and another for all the engine models. The first bucket served to store the task files within their respective task folders and the second bucket to store engine models, both using the object path scheme introduced in Section 4.2.4. The buckets were configured such that their data would only exist in one region to ensure that AGT data did not cross international borders as prohibited by export control laws. Each bucket was secured such that they could only be accessed with IAM credentials and not through any public endpoint.

EC2 AWS EC2 was used to provide nodes on which the *Worker* could be deployed so that *Executors* could be launched to perform analysis for tasks. The EC2 machines were running Widows as their OS and were configured to have at minimum the CPU, memory and networking capacity to reliably run at least one *Executor*. The machines also had IAM credentials attached to them allowing access to the DynamoDB table and S3 buckets provisioned for the ATaaS, allowing any process within the EC2 to access the *Task Repository* and *File Storage Service* automatically. Lastly, the EC2 Machines were all in the same EC2 Virtual Private Cloud (VPC) and hence were in the same local area network and shared the settings associated to the VPC such as networking security.

IAM AWS IAM was used to provide credentials to the ATaaS core components and users in order to access AWS services. IAM credentials are used in the ATaaS to authenticate requests made to S3 for the *File Storage Service* and DynamoDB for the *Task Repository*. Following the operational security principles of giving access of least privilege, two sets of credentials were created; one for the core components and the other for end users.

- **Core Credentials**: The IAM credentials for the core components granted full access to the S3 buckets used in the *File Storage Service* and the DynamoDB table in the *Task Repository*. These IAM credentials were either embedded within the container for each core component or the EC2 machine that was running any core component.
- **User Credentials**: The IAM credentials for the users granted write, read and list access to the objects of the S3 bucket used for task folders. These credentials would allow the user to manipulate any possible task folder and file.

4.3.3 Hybrid Cloud Deployment

The ATaaS is designed using a distributed microservice architecture which means that it can take advantage of a hybrid cloud deployment. The core components of the ATaaS can be deployed anywhere as long as they have access to the external third party services utilized in the architecture. A hybrid cloud deployment of the ATaaS enables the system to take advantage of underutilized on-premise hosts to act worker nodes in the ATaaS. The most important aspect in ensuring that hybrid deployment of the ATaaS is functional is the correct networking between all the components as explained in Section 3.2.4. Figure 4.1 showcases the ATaaS component placement and the general networking configuration for a hybrid deployment.

License Management Service The *License Management Service* is deployed on private Siemens Energy infrastructure and managed as a service by Siemens Energy IT. The deployment of the *License Management Service* must be in a private corporate network where all network users can be implicitly trusted due to the lack of any security mechanism within the service itself. In a hybrid deployment the *License Management Service* must be exposed to the public cloud *Worker* nodes using a secure private connection such as a VPN tunnel instead of traversing the public internet. This means that any *Worker* node using a public cloud instance without a secure network path to the *License Management Service* is guaranteed to timeout all the *Executor*'s it would launch as the FEA software would freeze during the license authentication request.

AWS Services The *Task Repository* and the *File Storage Service* within the ATaaS are AWS services which can be accessed using the general internet as all requests are encrypted and authenticated using IAM credentials. This introduces the requirement that any non EC2 host with a *Manager*, *Worker* or *Executor* must maintain general internet access to the AWS service endpoints to function correctly. It is important to note that the *Task Repository* serves as the message passing service of the system, therefore general internet access becomes critical for the proper operation of the ATaaS. The *File Storage Service* is primarily used by the *Executor*, without access to it the *Executor* will never be able to perform any actual FEA as all task files would be unavailable.

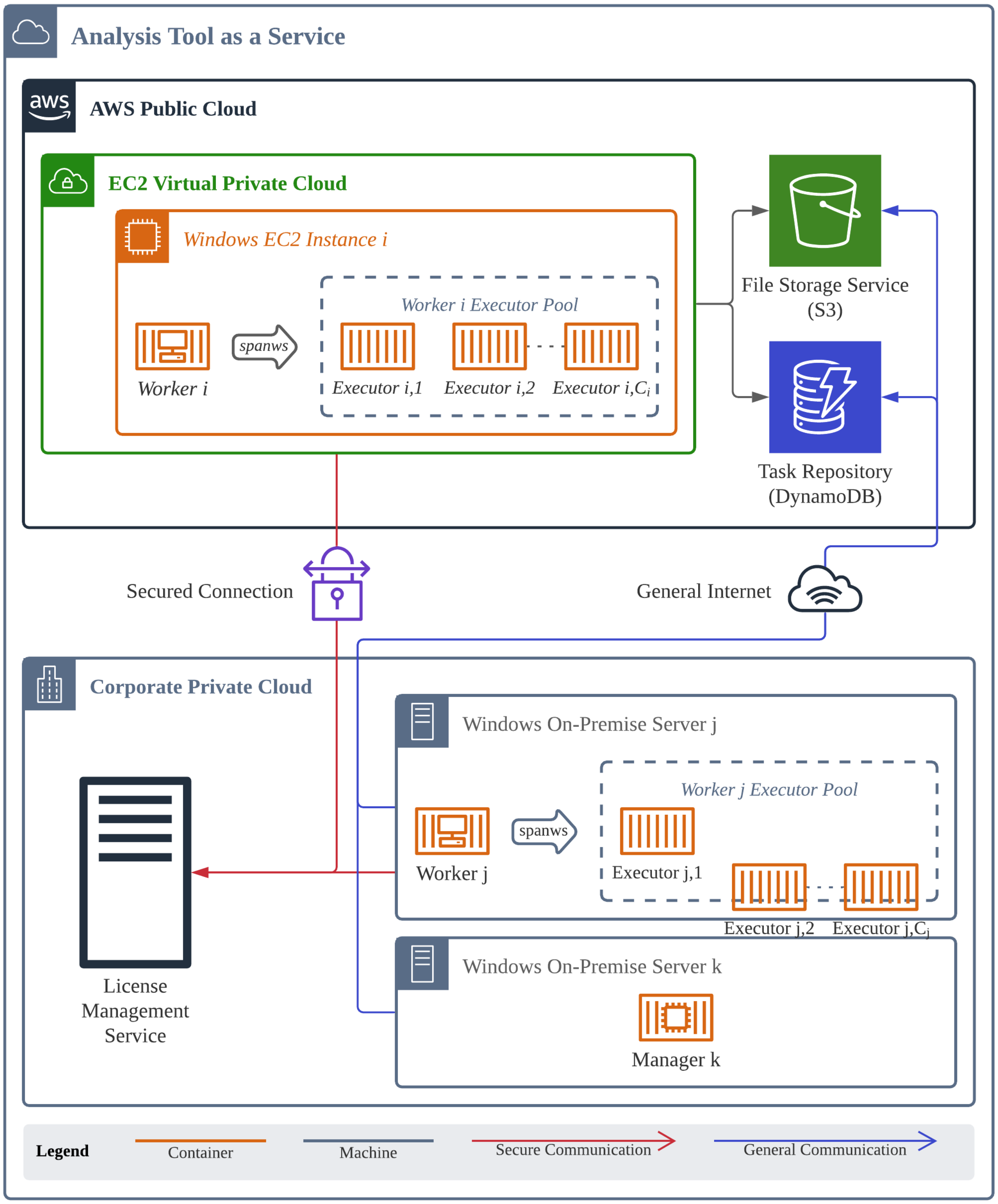


Figure 4.1: Hybrid cloud deployment and networking for ATaaS

Manager The ATaaS primary service endpoint served by the *Manager* implements no security or authentication mechanism. Any entity which can reach the network address and port of the *Manager* would have access to the ATaaS primary service endpoint. In order to secure the ATaaS primary service endpoint the *Manager* should only be deployed on hosts in networks where all inbound traffic can be implicitly trusted to use the ATaaS. Hence the *Manager* is preferably deployed in a closed network such as the network of the corporate private cloud or the EC2 VPC.

Chapter 5

Validation

The ATaaS was designed, built and then deployed for testing and validation of the service. The core components and the entire system were manually tested to ensure that they operated as envisioned. Subsequently, the ATaaS was validated experimentally to demonstrate that the performance and behaviour of the system were as expected and fulfilled our two research questions.

5.1 Testing

The correctness of the ATaaS is validated by testing the individual components and the system as a whole to ensure that they operate as required. First, each core ATaaS component must be tested individually in isolation to verify that it correctly performs its designated functionality. Since the ATaaS uses a distributed microservice architecture, the *Manager*, the *Worker* and the *Worker* are the key components that had to be tested individually to ensure that designated behaviour can be observed upon certain inputs.

Once all the core components have been individually tested, the entire system must be tested to ensure that the ATaaS functions as envisioned. Black box testing is used for the system as such that it is verified for a set of inputs, the outputs are as expected.

5.1.1 Component Testing: Manager

The *Manager* was tested to ensure it provides a functional and correct service to ATaaS users as the primary endpoint. The *Manager* endpoint provides a web based RESTful API which allows users to create, queue and get tasks. Each API URI was tested with multiple different requests in order to verify the ability of the *Manager* to respond to valid and invalid requests. Testing each API URI individually and independently is essential in ensuring that the system can handle production workloads without malfunctioning.

Create Task

As *main success scenario* (**CT1**), the create task API call was tested with a valid task creation request in which all the configuration parameters and the owner were specified. The result of this test was successful as verified by the response given by the *Manager* returning a valid *task ID*. As further validation the *task ID* was manually queried in the *Task Repository* and the task item returned contained all the task configuration parameters as specified in the request, the created task state as required and all the metadata was sound.

Subsequently the create task API was tested with two invalid task creations requests. In the first invalid request (**CT2**) the owner was not specified, which resulted in no *task ID* being returned and a debug message indicating the issue. In the second invalid request (**CT3**) a non existent *tool version* was specified as a configuration parameter, which resulted in no *task ID* being returned and a appropriate debug message. These tests demonstrated that the API call could protect the *Task Repository* from ill-formatted task creation requests while providing feedback to the user.

Queue Task

As *main success scenario* (**QT1**), the queue task API call was tested with a valid task queuing request in which a *task ID* for a task in the created state was specified. The result of this test was successful as verified by the response given indicating that the task was successfully queued. As further validation, the *task ID* was manually queried in the *Task Repository* and the item returned indicated a queued task state and the correct queue time.

Subsequently the queue task API was tested with two invalid task queuing requests. In the first invalid request (**QT2**) a non existent *task ID* was specified, which resulted in the response simply containing a debug message identifying the issue correctly. In the second invalid request (**QT3**) a *task ID* was specified for a task that was not in the created state, which resulted in the response containing a debug message correctly indicating that the task cannot be queued due to its state. This test demonstrated that the API call would not violate the task state machine model.

Get Task

As *main success scenario* (**GT1**), the get task API call was tested five times with a valid get task request in which a *task ID* for a task in a particular state was specified. The five states from the task state machine model was tested; created, queued, running, completed or failed. For every request the results of the test were successful as the task configuration parameters, metadata and state returned in the response were as expected and the manual query of the *Task Repository* with the *task ID* confirmed the latter.

Subsequently, the get task API call was tested with two invalid get task requests. In the first invalid request (**GT2**) a non existent *task ID* was specified, which resulted in the response simply containing a debug message informing that the *task ID* requested did not exist in the ATaaS. In the second invalid request (**GT3**) no *task ID* was specified in the request, which again resulted in the response containing a debug message identifying the request issue correctly. These tests demonstrated the ability of the *Manager* to clearly provide feedback to the user when something failed.

5.1.2 Component Testing: Worker

The *Manager* was tested to ensure that it can secure tasks and launch and manage the *Executor*'s for its tasks correctly. The *Manager* is a web server with a update routine that it periodically and internally invokes in order to perform task timeouts, task cleanups and task launching. Each of these main functionalities was tested by creating special tasks in the *Task Repository* and setting up the host of the *Worker* in a particular manner. The following actions can be considered as inputs and then the behaviour of the *Worker* was considered as the output.

Task Timeout

As *main success scenario* (**TT1**), an *Executor* was manually launched for a special task in the *Task Repository* with a set runtime that would exceed the maximum runtime allowed by the ATaaS and an attempt count below the maximum value. Once the *Executor* exceeded the maximum allowable runtime, the *Worker* stopped and cleaned up the *Executor* as expected and then requeued the task.

Subsequently, we then let the *Worker* relaunch *Executor*'s for this special task (**TT2**) until it was attempted the maximum number of times allowable by the ATaaS, at which point the *Worker* stopped and cleaned up the *Executor* as expected and then failed the task in the *Task Repository*. The successful behaviour of the *Worker* demonstrated its ability to correctly timeout tasks which seem to be running excessively long and subsequently communicate this information to the user as important feedback.

Task Cleanup

As *main success scenario* (**TC1**), an *Executor* was manually launched for a valid task onto a node and allowed to execute to completion. Once the *Executor* terminated it was successfully verified that the *Worker*'s update routine cleaned up the task correctly by uploading the container logs of the *Executor* to the appropriate *task folder* in the *File Storage Service*, removing the *Executor* from the node and then identifying that the *Executor* successfully updated the task state to completed in the *Task Repository*.

Subsequently, the same test above was performed (**TC2**) however the *Executor* was manually terminated before it could complete. In this case it was successfully verified that the *Worker*'s update routine correctly cleaned up the task by identifying that the *Executor* did not update the task state to completed or failed in the *Task Repository* and subsequently failing the task in the *Task Repository* along with an appropriate debug message and end time. This showcased the ability of the ATaaS to recover from the sudden termination of an *Executor* in case of a unknown error.

Task Launching

As *main success scenario* (**TL1**), multiple valid tasks in the queued state were created in the *Task Repository*. It was then successfully verified that the *Worker*'s update routine scanned for queued tasks in the *Task Repository*, organized all the queued task results by queue time, secured the oldest tasks and launched an *Executor* for each until the maximum number of concurrent *Executor* was reached. It was further verified that each *Executor* launched by the *Worker* was properly passed all the task's configuration parameters.

Subsequently, once all the queued tasks were taken from the *Task Repository* (**TL2)**, it was successfully verified that the *Worker*'s update routine identified the lack of queued tasks in the *Task Repository* and exited without launching any new *Executors*. Lastly, for a *Worker* that had the maximum number of *Executors* running (**TL3**), it was successfully verified that the the *Worker*'s update routine did not attempt to launch more *Executors*.

5.1.3 Component Testing: Executor

The *Executor* was tested to ensure that it is able to perform FEA for tasks correctly. The *Executor* is a script within a container which passes a *execution script* to a particular version of the FEA software within to perform the FEA for a given task. There were two tests performed on the *Executor*, one with a valid task and another with a invalid task. For each test, a special task in the running state was created in the *Task Repository* for both tests as inputs and the behaviour of the *Executor* was considered as the output.

Valid Task (ET1)

A valid task was manually created in the *Task Repository* in which every configuration parameter was set. Moreover, all the required task files for this task, including the execution script, were uploaded to the *task folder* in the *File Storage Service*. A *Executor* for this task was then launched and passed the *task ID* and all the configuration parameters. It was then successfully verified that the *Executor* performed correctly:

1. All the task files from the *task folder* in the *Fie Storage Service* were downloaded into the *task folder* of the *execution environment*.

2. All the engine models specified were downloaded from the the *Fie Storage Service* into the *task folder* of the *execution environment*.

3. All the input folders specified were created as sub folders in the *task folder* of the *execution environment*.

4. The *execution script* is verified to exist at the location specified in the *task folder* of the *execution environment*.

5. The correct version of the FEA software is launched as specified by the *tool version*, it is passed the *execution script* as input.

6. Upon completion of the analysis, the FEA software log file and the *output folders* specified are uploaded from the *task folder* in the *execution environment* to the *task folder* in the *File Storage Service*.

7. The task state is updated to completed in the *Task Repository* along with the end time.

Invalid Task (ET2)

An invalid task was manually created in the *Task Repository* in which every configuration parameter was set. However, all the required task files for this task, including the required execution script, were not uploaded to the *task folder* in the *File Storage Service*. A

Executor for this task was then launched and passed the *task ID* and all the configuration parameters. It was then successfully verified that the *Executor* performed correctly:

1. No task files from the *task folder* in the *Fie Storage Service* were downloaded into the *task folder* of the *execution environment* as they were never uploaded.
2. The execution script is verified to not exist at the location specified in the task folder of the execution environment.
3. The task state is updated to failed in the *Task Repository* along with the end time and a debug message.

5.1.4 System Testing

The entire ATaaS system must be tested to ensure that users can perform FEA with it from task creation to completion. As *main success scenario* (**ST1**) we test the ATaaS by creating a computationally demanding FEA task and then using the ATaaS to execute the task. The results of this analysis were then verified to ensure that the ATaaS functioned correctly. The test was successfully performed manually as outlined below along with a detailed UML sequence diagram of the internal system calls in figure 5.1.

1. **Task Preparation**: A local task folder was prepared on the machine of a user, inside which the *execution script* and any additional task files were placed.
2. **Task Initialization**: A create task request was made to a *Manager* service endpoint by the user and a *task ID* was returned. In the create task request, the user's email was specified and the following configuration parameters were set as determined by the objectives of the task and the *execution script*:
    - Tool Version: The required FEA software version to invoke for the analysis.
    - Execution Script: The path and name of the *execution script* that was prepared inside the local *task folder* and will ultimately be in the *execution environment*.

- Engine Model: The *engine model* to download into the *task folder* of the *execution environment* so the *execution script* can use it during analysis.
- Input Folder: The empty folder to create inside the *task folder* of the *execution environment* so the *execution script* can write data to it during analysis.
- Output Folder: The sub folder in the *task folder* of the *execution environment* that will contain the desired results from the FEA, which will need to be uploaded back to the *task folder* in the *File Storage Service* for the user to retrieve.

3. **Task File Submission**: The AWS CLI was used to upload the entire local task folder to the *task folder* in the *File Storage Service* according to the *task ID* returned earlier.
4. **Task Queuing**: The queue task request with the *task ID* was made to a *Manager* service endpoint by the user and a success message was returned.
5. **Task Getting**: The get task request with the *task ID* was made to *Manager* service endpoint by the user to verify the queued status the task and ensure all the task configuration parameters set set correctly.
6. **Task Launching & Executing**: The above get task request was periodically made until the task state changed to running and and subsequently completed.
7. **Task Output**: The AWS CLI was used to download all the task results from the *task folder* in the *File Storage Service* into the local task folder.
8. **Task Verification**: It was successfully verified that the task result files were correct and as expected for the FEA requested. Additional log files from the FEA software were also present and valid.

The above test was similarly performed on a batch of tasks using the service script (**ST2**) in order to validate its automation capability and benefits. In this test a local batch folder with individual task folders was prepared and the service script was used to automatically create the tasks in the ATaaS, upload the local task folders accordingly and

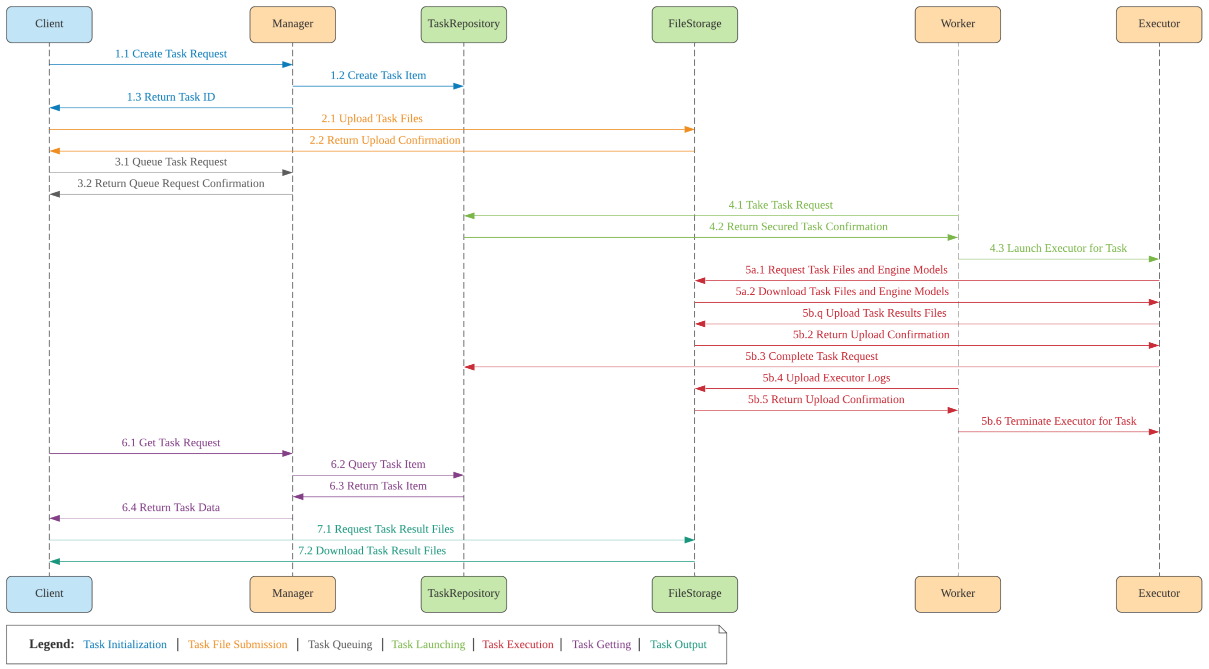

Client
Manager
TaskRepository
FileStorage
Worker
Executor
1.1 Create Task Request
1.2 Create Task Item
1.3 Return Task ID
2.1 Upload Task Files
2.2 Return Upload Confirmation
3.1 Queue Task Request
3.2 Return Queue Request Confirmation
4.1 Take Task Request
4.2 Return Secured Task Confirmation
4.3 Launch Executor for Task
5a.1 Request Task Files and Engine Models
5a.2 Download Task Files and Engine Models
5b.q Upload Task Results Files
5b.2 Return Upload Confirmation
5b.3 Complete Task Request
5b.4 Upload Executor Logs
5b.5 Return Upload Confirmation
5b.6 Terminate Executor for Task
6.1 Get Task Request
6.2 Query Task Item
6.3 Return Task Item
6.4 Return Task Data
7.1 Request Task Result Files
7.2 Download Task Result Files
Client
Manager
TaskRepository
FileStorage
Worker
Executor
Legend: Task Initialization | Task File Submission | Task Queuing | Task Launching | Task Execution | Task Getting | Task Output

Figure 5.1: UML Sequence Diagram of internal system calls for scenario **ST1**

then queue the tasks. The results from the completed tasks were successfully verified to be correct. This test demonstrated that the tools we developed for users to more easily and effectively use ATaaS worked as intended.

5.2 Experimental Validation

The deployment platform of the ATaaS has multiple parameters which have an impact on the performance of the overall system. Understating how each variable affects the system in order to optimize the execution time of the analysis tasks while minimizing the resource wastage is essential when determining the deployment strategy of the ATaaS. Furthermore, ensuring that the ATaaS offers better execution time performance than the existing solutions helps justify the business use case of the system. Finally, validating the ATaaS on real engineering analysis tasks proves that the system can handle productions workloads while delivering the promised benefits of automation, scalability and increased execution time performance.

As an experimental validation of the proposed ATaaS architecture, we carried out measurements to address two main research questions:

- **RQ1**: How do various platform parameters influence the performance of the proposed architecture?
- **RQ2**: How does the proposed architecture perform for real engineering analysis tasks?

5.2.1 Performance Estimation Models

Mathematical models for any software system may enable to analyze or predict key characteristics of the system without prior to execution. For that purpose, we first define two models to estimate the end-to-end task time and the task analysis time. We then use values from these two models to build a high-level *performance estimation model*. Such a *performance estimation model* for the proposed ATaaS architecture reveals how the various platform parameters would affect the total service time for a batch of analysis tasks. Subsequently, we derive a maximum task wait time model from the three previous models.

Task Time Model

The end-to-end execution time of a single task from when it was queued to when it will complete can be modelled as following:

$$T_{\text{task}} = T_{\text{wait}} + T_{\text{launch}} + T_{\text{analysis}}$$

The formula uses the following notation:

- T_{task}: The total response time to execute an analysis task from queuing to completion
- T_{wait}: The amount of time a task waits in the queued state while the nodes in the ATaaS are executing other analysis tasks, this should only occur if the number of tasks exceeds the concurrent task execution capability of the entire ATaaS

- T_{launch}: The amount of time it takes for a *Worker* to invoke its update routine, checkpoint all of its existing tasks and secure the task and launch an *Executor* for it

- T_{analysis}: The amount of time it takes for an *Executor* to run for an analysis task from creation to completion.

Analysis Time Model

We can further break down T_{analysis} for a single task into a model as following:

$$T_{\text{analysis}} = T_{\text{setup}} + T_{\text{execution}} + T_{\text{cleanup}}$$

Where the following notation is used:

- T_{setup} : The amount of time the *Executor* takes to setup the *execution environment* from container creation until invocation of the FEA software for analysis

- $T_{\text{execution}}$: The amount of time the FEA software within the *Executor* takes to perform the FEA for a given task that it was invoked for

- T_{cleanup}: The amount of time the *Executor* takes to cleanup the task until the container exits

Service Time Model

Assuming that the ATaaS is idle, the total execution time for a batch of similar analysis tasks queued together can be estimated using the following *performance estimation model*:

$$T_{\text{service}} = \left\lceil \frac{M}{\sum_{i=1}^{N} C_{\text{i}}} \right\rceil \times (T_{\text{analysis}} + T_{\text{launch}})$$

The *performance estimation model* uses the following notation:

- N : The number of nodes

- C_i : *Concurrency*, i.e. the maximum number of concurrent *Executors* on a node
- M : The number of analysis tasks in the batch
- T_{analysis} : The approximate runtime of an *Executor* for a single task in the batch
- T_{launch} : The amount of time it takes for a *Worker* to invoke its update routine, checkpoint all of its existing tasks and secure the task and launch an *Executor* for it

We call the sum of all node concurrencies the *Concurrent Executor Capacity* of the ATaaS.

Maximum Wait Time Model

Figure 5.2 illustrates how individual tasks (in a batch of tasks) can be assigned to different nodes. One can observe that T_{wait} depends on the scheduling of the individual tasks. For example, $T_{\text{wait-1}} = 0$ and $T_{\text{wait-2}} = 0$ while $T_{\text{wait-3}}$ and $T_{\text{wait-4}}$ equal to $T_{\text{task-1}}$ and $T_{\text{task-2}}$, respectively. Henceforth, from that we can model that T_{wait} for any given task in a batch of tasks can be expressed as following:

$$T_{\text{wait}} \leq T_{\text{service}} - (T_{\text{analysis}} + T_{\text{launch}})$$

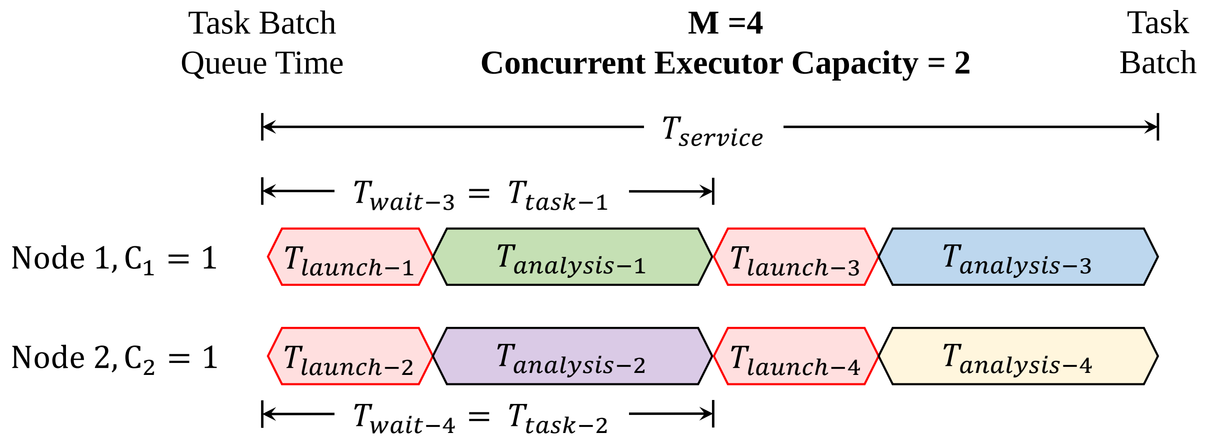


Figure 5.2: T_{wait} for tasks in a batch (with $M = 4$ and *Concurrent Executor Capacity* $= 2$)

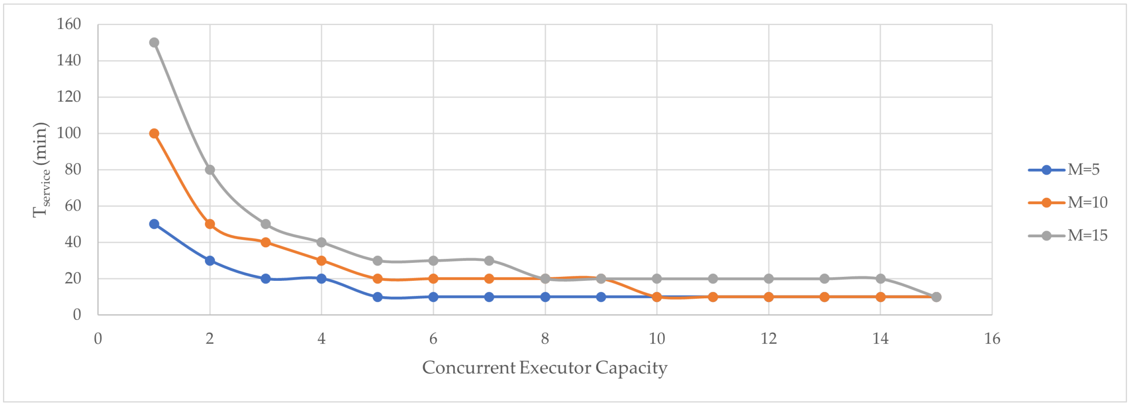
5.2.2 Experiments for RQ1

Theoretical Estimate

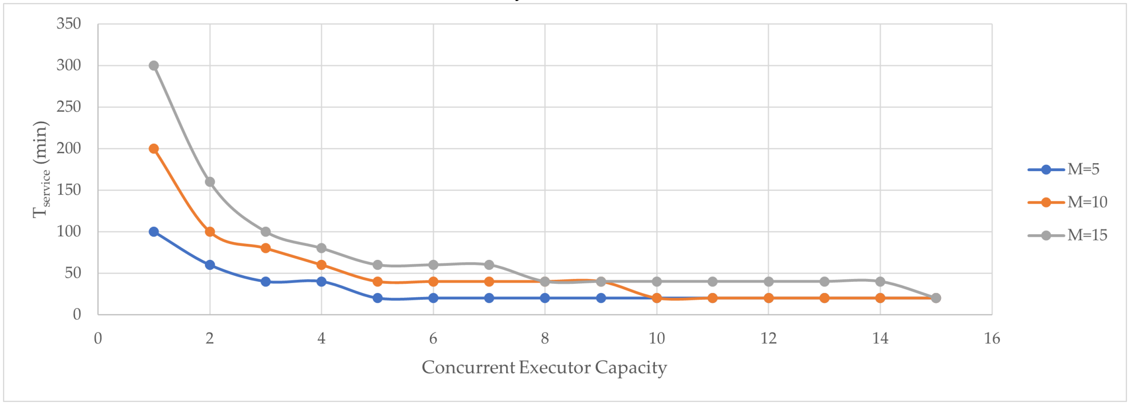
From analysing the performance estimation model we can observe that the two main variables that effect the total service time for a batch of tasks is the number of tasks in the batch (M) and the *Concurrent Executor Capacity* of the deployment. If we input different values for the variables in the performance estimation model we can observe how T_{service} is affected. T_{launch} can be considered as a constant as it is related to the frequency of the update routine in the *Worker* which is five minutes for our test deployment, therefore we set $T_{\text{launch}} = 5$ minutes. If we input various different values of $M = 5, 10, 15$ and *Concurrent Executor Capacity* between $[1 : 15]$ for $T_{\text{analysis}} = 5, 10, 15$ minutes in the performance model, we obtain the following graphs for T_{service} in Figure 5.3.

From analyzing these graphs, we can extrapolate that if *Concurrent Executor Capacity* is equal to one then the ATaaS essentially becomes a sequential analysis system where T_{service} is at its maximum and the only benefit that the ATaaS provides is automation. Furthermore, we observe that once the *Concurrent Executor Capacity* is equal to M, T_{service} reaches its minimum at which point $T_{\text{service}} = T_{\text{analysis}} + T_{\text{launch}}$. Increasing the value of *Concurrent Executor Capacity* beyond M will offer no performance gain.

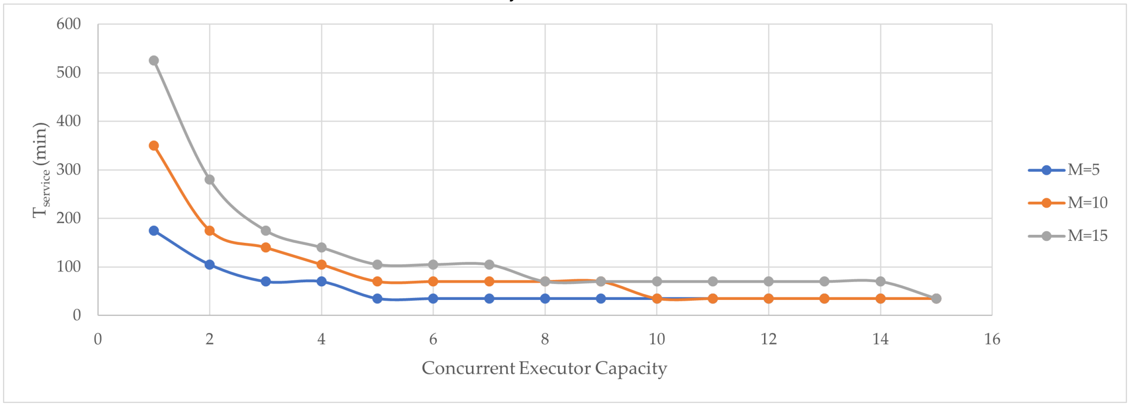
We also observe that while T_{analysis} affects the T_{service} maximum and minimum, it has no theoretical impact on the rate of change of the T_{service} for task batches of the same size. Therefore it is important to consider that regardless of T_{analysis}, the *Concurrent Executor Capacity* in a deployment provides diminishing returns in terms of T_{service} for every extra node. For a deployment with a static number of nodes, setting the value of *Concurrent Executor Capacity* should be done such that T_{service} is small enough to meet business deadlines without over provisioning resources that will bring relatively little gain.



(a) $T_{\text{analysis}} = 5$ minutes



(b) $T_{\text{analysis}} = 15$ minutes



(c) $T_{\text{analysis}} = 30$ minutes

Figure 5.3: Estimation of T_{service} w.r.t. to Concurrent Executor Capacity and the number of tasks in the batch (M)

Simulated Estimate

Simulation Setup The performance estimation model was simulated on an ATaaS deployment in an public cloud setup with two identical EC2 machines for nodes. Each *Worker* on the nodes was configured to have a different *Executor Concurrency* depending on the *Concurrent Executor Capacity* experiment that was being performed. The *Worker*'s all had an update routine frequency of five minutes. There was a *Manager* deployed on the local computer from where the tests were being performed in order for the tester to interact with the ATaaS. The simulated analysis tasks being used for the tests were the same as real engineering analysis tasks except for the following difference:

1. During the task execution phase of the *Executor* the FEA software was not invoked and instead a sleep command with a set time was invoked
2. Since no actual task result files would be generated during the task execution phase, a sleep command was used during the task cleanup phase to simulate uploading task result files.

The sleep command allowed us to simulate the performance estimation model while ensuring that T_{analysis} was relatively homogeneous between the different tasks in a batch, the task setup phase of the *Executor* was real. Furthermore, using the sleep command enabled us to test the performance estimation model for different values of the *Concurrent Executor Capacity* as we could increase the *Concurrency* for each of the two EC2 nodes without reaching their computational limit.

The performance estimate model assumes that M tasks in a batch would be available instantaneously in the queued state for execution. However, in our simulation setup, the individual tasks in a batch could only be queued sequentially one after another. Hence in our simulation tests the task batch does not have a universal queue time which makes it impossible to calculate the simulated service time for a batch of tasks. In order to address this issue we recorded the individual queue time for each task in a batch and then we considered the median queue time and the last queue time as replacement for the task batch queue time. This allowed us to have a meta task batch queue time that could serve in our calculations and validate our simulation results against theoretical estimates.

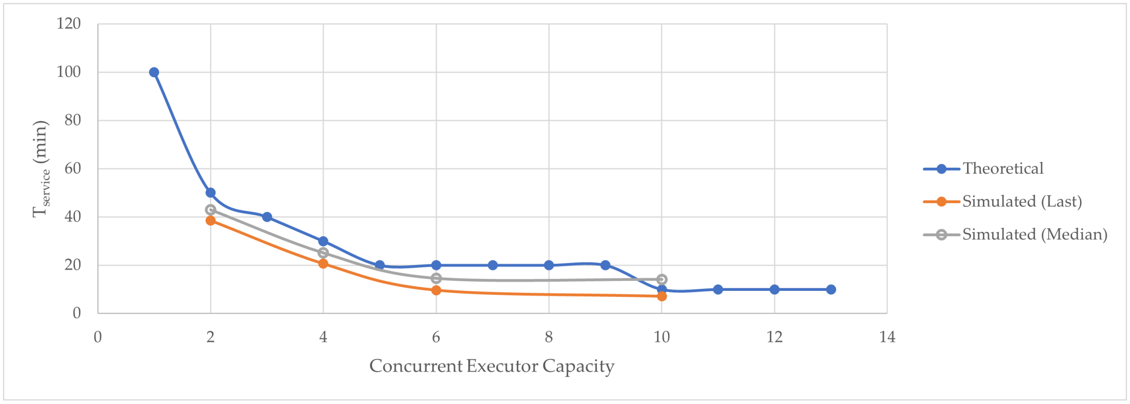
Simulation Results We picked certain input values from the above theoretical estimates and tested them onto the simulation setup detailed earlier in order to generate simulated performance estimates for the ATaaS. We tested the model setup for task batches of size $M = 10$ with *Concurrent Executor Capacity* $= 2, 4, 6, 10$ for $T_{\text{analysis}} = 5, 10, 15$ minutes, and we obtained the following graphs for T_{service} in Figure 5.4.

We observe from the simulation results in Figure 5.4 that T_{service} for any data point is generally below the performance estimate from the model. The only scenarios in which the performance estimate does not hold true is for when *Concurrent Executor Capacity* $= 10$ and we use the median queue time for calculating T_{service}. This can be explained by the fact that when the 10 tasks were sequentially queued over a period of time longer than the frequency of the *Worker* update routine, it resulted in the *Concurrent Executor Capacity* not being fully utilized for all 10 tasks by the median queue time.

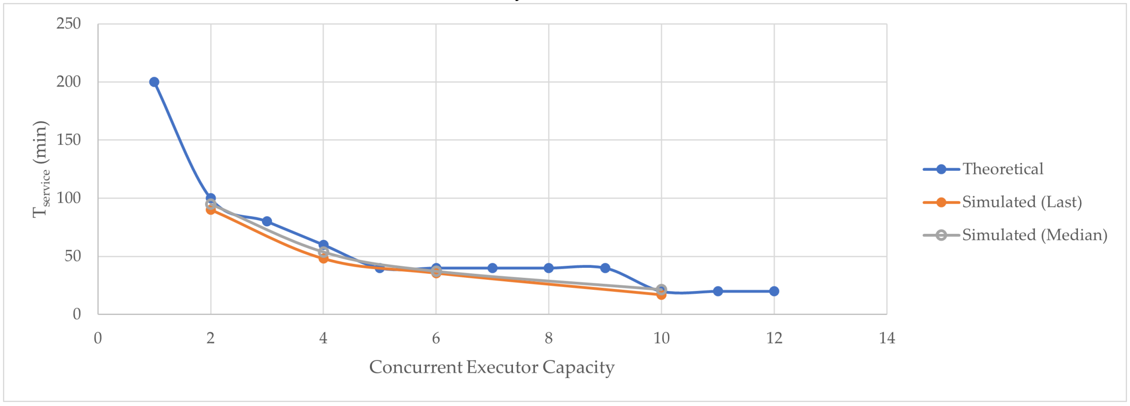
Concurrent Executor Capacity	$T_{\textbf{analysis}} = 5$	$T_{\textbf{analysis}} = 15$	$T_{\textbf{analysis}} = 30$
2	25.99%	10.23%	2.82%
4	37.15%	21.96%	9.31%
6	69.81%	11.56%	4.50%
10	34.06%	16.51%	0.20%

Table 5.1: T_{service} percent difference between performance model estimate and simulations results using last t_{queue} for calculations

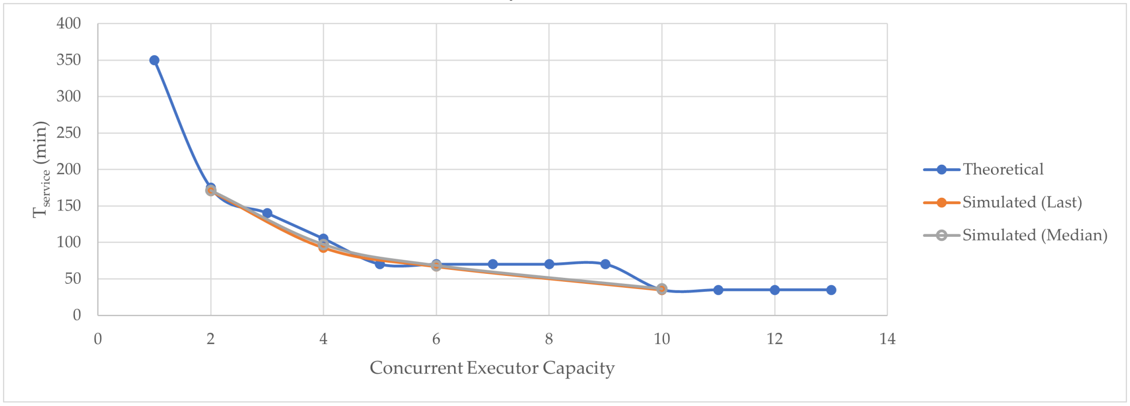
If we observe Table 5.1 for the percent difference between the simulation results and the theoretical estimate from the performance model we can deduce that as T_{analysis} becomes smaller the estimates from the performance models becomes more inaccurate. Furthermore, looking at the simulation results empirically we can assume that the that performance estimate model is valid and can be used as reliable indicator for T_{service} for any ATaaS deployment.



(a) $T_{\text{analysis}} = 5$ minutes



(b) $T_{\text{analysis}} = 15$ minutes



(c) $T_{\text{analysis}} = 30$ minutes

Figure 5.4: Comparison of estimated and simulated T_{service} w.r.t. to Concurrent Executor Capacity for $M = 10$, $T_{\text{launch}} = 5$

Threats to Validity There are certain threats to validity of the simulation setup and results which are used to validate the performance estimate model. Although empirically the performance estimate model is a good indicator of T_{service} as verified by the simulation results, we outline the following threats to the validity of that statement:

- The simulation results were derived from a single run of the simulation input data points and hence we cannot make any statistical statement about the validity of our simulation results.
- The task batch queue time did not truly exist and a meta value was used (last and median queue times) as the task batch could not be instantaneously be queued. It would have been preferred to develop a mechanism to queue tasks in parallel in order to have a true task batch queue time.
- The analysis tasks were simulated using a special *Executor* container in which the task execution time and task cleanup times were set. It would have been ideal to use a probability models and have these values be stochastic in order to increase the variance and validate the performance estimate model.

RQ1: Summary of results

The ATaaS performs as expected depending on the configuration of the deployment. The following platform parameters influence the performance of the proposed architecture:

- The number of analysis tasks in conjunction with the *Concurrent Executor Capacity* determines how many analysis tasks can be performed in parallel on the proposed architecture. As the ability to parallelize increases more tasks can be processed in the same unit of time.
- The execution time for any given analysis task determines the minimum amount of time required to process the task. Increasing *Concurrent Executor Capacity* will not speedup individual task times.

5.2.3 Experiments for RQ2

The ATaaS has to be tested and validated on real engineering tasks in order to ensure that performs as designed. In order to accomplish this real engineering tasks would be tested on a hybrid cloud deployment of the ATaaS as well on a local server with traditional scripts. Through these tests we will be able to gauge and compare the performance of the proposed architecture.

Hybrid Test Setting

Hybrid Setup In order to observe the performance of the ATaaS with real engineering tasks, we decided to deploy a fully operational system. As the ATaaS is built using a distributed microservice architecture to be deployed on a hybrid cloud, we deployed the ATaaS across different machines and networks spanning the public and private cloud as modeled in Figure 4.1. For our particular setup of the ATaaS we used two AWS public cloud EC2 instances and one Siemens private cloud server as worker nodes each having a *Concurrency* $= 1$ for a *Concurrent Executor Capacity* $= 3$. We tested $M = 10$ real engineering task batch with each task ideally having $T_{\text{analysis}} = 150minutes$. The *worker* update routine runs every 5 minutes so we can expect to observe $T_{\text{launch}} \leq 5minutes$.

Hybrid Results The real engineering tasks T_{service} performance on the hybrid deployment of the ATaaS are outlined as following in Table 5.2. T_{service} is calculated twice and uses a meta task batch queue time as outlined earlier, one being the last queue time for a task in the batch and the other being the median queue time for a task in the batch. T_{task}, T_{analysis} and T_{service} are all calculated in minutes as following:

- $T_{\text{task}} = t_{\text{end}} - t_{\text{queue}}$
- $T_{\text{analysis}} = t_{\text{end}} = t_{\text{start}}$
- $T_{\text{service-last}} = t_{\text{end-10}} - t_{\text{queue-10}}$

- $T_{\text{service-median}} = t_{\text{end-10}} - t_{\text{queue-5/6}}$

Task #	$t_{\textbf{queue}}$	$t_{\textbf{start}}$	$t_{\textbf{end}}$	$T_{\textbf{task}}$	$T_{\textbf{analysis}}$
Task 1	07:04:43	07:05:27	09:22:28	137.75	137.02
Task 2	07:05:43	07:06:38	10:15:55	189.28	189.28
Task 3	07:07:23	07:09:20	10:17:27	190.07	188.12
Task 4	07:08:57	09:29:41	12:05:59	297.03	156.30
Task 5	07:09:58	10:20:32	13:28:47	378.82	188.25
Task 6	07:10:57	10:21:20	13:28:47	377.83	186.95
Task 7	07:11:58	12:16:50	14:54:49	462.85	157.98
Task 8	07:12:58	13:31:22	16:38:18	565.33	186.93
Task 9	07:13:58	13:34:34	16:43:22	569.40	188.80
Task 10	07:14:59	15:00:50	17:26:39	611.67	145.82
$T_{\textbf{service-last}}$		611.67			
$T_{\textbf{service-median}}$		616.20			

Table 5.2: T_{task} for all the tasks along with T_{service} for a batch of tasks

If we were to use the performance estimation model to estimate what T$_{\text{service}}$ should be for an ATaaS with aforementioned settings, T_{service} would be $620 minutes$. In our real engineering test where tasks are not perfectly identical and the computational capability of each node is heterogeneous, we notice that T_{analysis} varies between 145 and 190 minutes. The variation can be explained by the fact that the EC2 nodes have less powerful CPU's then the Siemens server. Despite this variation, the model we built is still able to provided a reasonable estimate with less than 0.61% difference than the theoretical estimate. Furthermore, the task result files for each task in the batch were empirically verified by engineers to ensure if the FEA was performed as required, to which they were satisfied.

Local Test Setting

Local Setup In order to compare the performance of the ATaaS with the existing process of launching the FEA software locally using a custom script and local task folders, we deployed a local standalone machine with four cores and executed on it the same task batch used for the hybrid test. In a local setup the FEA software was invoked ten times simultaneously by the script and passed the *execution scripts* for each task in the batch from the local task folder. Therefore, we can only measure $T_{\text{execution}}$ as there is no T_{wait}, T_{launch}, T_{setup} and T_{cleanup} for tasks in a local setting as these concepts relate to the automation provided by the ATaaS. T_{service} for the batch is calculated as the difference between the end time of the last task and the invocation time of the custom script.

Local Results The $T_{\text{execution}}$ for each task and the T_{service} in minutes for the task batch (same one from the hybrid test) in the local setup are outlined in Table 5.3.

Task #	$T_{\textbf{execution}}$
Task 1	387.85
Task 2	445.47
Task 3	448.45
Task 4	451.12
Task 5	451.25
Task 6	452.95
Task 7	454.98
Task 8	456.37
Task 9	456.42
Task 10	460.77
$T_{\textbf{service}}$	460.80

Table 5.3: $T_{\text{execution}}$ for all the tasks along with T_{service} for a batch of tasks

We observe that T_{service} for the batch of tasks is lower than the hybrid setup however when you factor that this was performed on a four core machine, the ability to parallelize the tasks was greater than the ATaaS (*Concurrent Executor Capacity* $= 3$). However, we also observe that since 10 tasks were simultaneously launched on the local machine, the individual $T_{\text{execution}}$ was significantly longer than T_{analysis} ($387.85 \gg 189.28$), even though T_{analysis} additionally includes T_{setup} and T_{cleanup}. This showcases that the CPU scheduling on local machine slows down all the tasks in the batch in order to perform the task execution in parallel rather than trying to finish each task as quickly as possible before moving on to another task like the ATaaS. Due to ability to set the *Concurrency* per node appropriately (C_{i} should generally be set to less than the number of cores on a node), the individual T_{analysis} will generally be lower than $T_{\text{execution}}$ when launching task batches all at once on a local machine. We can also postulate that with a larger *Concurrent Executor Capacity*, the ATaaS would have had a lower T_{service} time compared to a local machine.

RQ2: Summary of results

The hybrid test demonstrated the ability of the ATaaS to perform FEA for real engineering analysis tasks and produce valid results while also meeting the expected performance time based on the deployment setup. When compared to local test the ATaaS completed each task significantly quicker than a local machine and given enough compute resources it would reduce the overall execution time of real engineering analysis tasks significantly. Therefore, we can conclude that the the proposed architecture performs adequately for real engineering analysis tasks.

Chapter 6

Related Work

Cloud-based distributed computing is an emerging computing paradigm that is widely used to implement various configurable and reliable models of computing resources like servers, networks, services, database/file storage, and applications [27, 36, 77]. Some of the key benefits that lead to refactoring of several applications, softwares, and analysis tools are; reducing the cost of ownership, reducing time-to-market a product (software/tool), increasing the application resilience, enabling scalability, and improving the security of an application [2]. Furthermore, due to fast-emerging solutions in speeding up and automating applications, several references to the related work are available in the literature. Therefore, in the following sections, we present and discuss some of the most important related works about cloudification, job submission systems in grid computing, containerization in distributed computing, and ATaaS because these works directly align with the scope of this thesis.

6.1 Cloudification

Although several applications are being rearchitected for cloud platforms (referred to as cloudification), this process of rearchitecting is always associated with several challenges like dependencies, the complex nature of the application, and many more. Researchers have developed different methodologies to cloudify several applications, platforms, software, and infrastructures [3, 35]. However, in this section, we mainly focus on the cloudification of legacy tools because it directly aligns with the scope of this thesis. One of the solutions for cloudification of the legacy tool is Service-oriented Architecture, in which the functions are defined as services, and these services are connected with well-defined interfaces [14, 33]. This enables it to reconfigure tools in a loosely coupled set of services to prepare them for cloudification. Another solution is to enable the multi-tenant feature using application profiling and categorization, which migrates the legacy application, including infrastructure, on the cloud [8, 76]. Other solutions include reverse engineering and implementing the code in another programming language, wrapping the legacy code [60,70], and tool-supported model-driven technology [63]. Although these solutions provide cloudification of legacy tools, the research in this direction is in the early stage, and several open research challenges require further investigation [33], for example; the unknown internal structure of the application, lack of knowledge for infrastructure environment, multi-tenancy influence, variability in configuration based on user preferences, and different resource requirements.

In summary, before cloudification of legacy analysis tools, the following key parameters should be considered; (1) architecture implications of legacy system migration to the cloud, (2) quality attributes of the transformation of the architectural elements, and (3) the most affected architectural elements.

6.2 Grid Computing

Grid computing is a distributed computing paradigm that pools heterogeneous computer, storage, network and software resources from various sources to provide computing and associated services to a target community through a unified interface [11]. The management software for any compute grid always provides a job submission system such that users can utilize the grid effectively, along with workflow management for complex tasks [34]. There are many grid management toolkits available to use on commodity hardware and software each with their own set of capabilities and availability.

Globus is a fully developed grid computing toolkit for UNIX based machines that enabled global deployments of grids and was widely used in the academic community to share computing resources [24, 25]. However, as of 2018 the *Globus* project was retired due to a lack of funding [23]. Another widely used toolkit for grid management and job scheduling system for Linux based machines is *SLURM* [81]. Moreover, *SLURM* is an active open source project with ongoing research and development [78]. It is important to note that both of these grid computing solutions are not built for Windows which means any task or jobs requiring the Windows OS are out of luck.

Conversely, there has been some effort put into grid computing toolkits with support for Windows, such as *GridBus* and *Condor*. *GridBus* is a service oriented grid and utility computing project that enabled a wide array of hardware and operating systems including Windows to become a feature complete compute grid [12]. However, a cursory search and a visit to the projects website indicates that the project has retired as it has no recent activity. The *Condor* open source project is a long running and extensively developed job scheduling system for grid computing. Originally developed for idling Linux computers to be used as part of a compute grid for job processing with minimal performance impact to the computer owner, it can now be used to manage fully distributed computer pools and schedule jobs with novel matching algorithm's [46,72]. Moreover, the project is moving towards fully supporting Windows natively however there are still some significant missing functionalities such as job check-pointing in Windows [17].

Grid computing and the related job submission systems domains that have been well defined and developed over the decades with very powerful open source tools available to the community. However, when trying to build a container based distributed service which has to natively support a Windows application, the practical usage of existing solutions is complex and unpractical, necessitating the research and development of the novel distributed ATaaS proposed in this thesis.

6.3 Containerization

In cloud-based distributed computing, containerization is a methodology to package the application to make it deployable in the cloud with very little overhead [58]. However, it is not always straightforward to determine the design and deployment of these containers because container orchestration deals with the entire lifecycle of the containers while also migrating them among different servers. In container orchestration, users can also handle the coordination between different containers of the same application. There exist several projects to orchestrate the containers depending upon the technology stack and management services. One of the commonly used containerization technology is Docker, which introduces an underlying container engine [54]. For example, Docker is being used in many resource-constrained applications because of its small overhead, i.e., Capillary Network [57] (containers without hardware virtualization support) and Glasgow Raspberry Pi cloud [74] (implemented using the built-in Linux containers). Apart from these solutions, many Docker-based containerizations are being deployed in cloud platforms, e.g., SciServer [51], Sciserver-Compute [51], data duplication based on Map-reduction [37], Virtual-Hadoop [16], ISLET [65] and many more [41,56,79]. Docker-based containerization makes the management of applications feasible, but several orchestration features need to be considered before deploying them, i.e., resource limit control, scheduling, load balancing, health check, fault tolerance, and auto-scaling.

6.4 Analysis Tool as a Service (ATaaS)

Cloud-based distributed computing provides Software as Service (SaaS) [44,55,80], Platform as a Service (PaaS) [1,9,69], and Infrastructure as a Service (IaaS) [64,68]. Although these services show significant performance improvement in several applications, like the telecom industry [38,62], automobile industry [15,49], and several other safety-critical applications [10,22,28], these services do not cover analysis and verification tools which are critical in designing any software, hardware, or hybrid system. All the existing analysis and verification tools are computationally intensive and required long periods of execution time. It has been reported in many applications that analysis and verification are the most time and resource consuming phases of the design cycle [39,50,66]. Therefore, there is a dire need to speed up and automate analysis and verification tools.

Towards this goal, recently, researchers have started to develop cloud-based verification and analysis tools, which can provide an analysis model as a service (MaaS) and/or the ATaaS [59]. In MaaS, cloud-based web services are used to build a distributed system model that can be used to improve the performance of several analyses, e.g., FEA, Computational Fluid Dynamics [26, 30, 45, 47, 71]. Although MaaS reduces the analysis time and runs several jobs, distributing the model for asynchronous analysis is not always feasible or practical. Thus, ATaaS has emerged as the most appropriate alternative solution along with its own underlying challenges. For example, the existing ATaaSs, e.g., CloudMEMS [67] and [59], mainly focus on providing the interface between the cloud platform and the user but do not entirely address the automation and speed-up issues in ATaaS.

While the ATaaS developed in this thesis builds on [59], it addresses the aforementioned challenges and issues by providing an asynchronous interface between the analysis tools (in this thesis, it is the FEA Tool) and the analysis systems. Moreover, it also supports deployment over a hybrid cloud-based platform while [59] was restricted to on-premise execution.

Chapter 7

Conclusion and Future Work

7.1 Conclusions

In this thesis, I proposed a *distributed microservice architecture* for an Analysis Tool as a Service (ATaaS) to enable the scalable (**C4**) and distributed execution (**C2**) of a legacy finite element analysis software used at Siemens Energy over a hybrid cloud platform. I also developed a *prototype implementation* (**C3**) which (1) uploads relevant task files and execution scripts to the AWS file storage service, (2) persists the task state in an AWS NoSQL database, and (3) executes FEA tasks on arbitrary Windows-based hosts (EC2 or on-premise) by using the legacy FEA software packaged into Docker containers (**C1**).

Since FEA may run for several hours, it is also important to predict the expected execution time of an FEA task. For that purpose, I developed various *performance predication models* to estimate various phases of task execution time by incorporating the number of *Workers* nodes and the number of tasks available within the ATaaS as input parameters.

In order to validate the proposed architecture and prototype software, I developed and executed *functional tests* (both on component and system level), and carried out various experiments. The faithfulness of the performance prediction models were investigated in a *simulated setting*, while the overall performance of the ATaaS framework was assessed in the context of a *real FEA engineering task* provided by Siemens engineers.

Limitations Our first prototype of the ATaaS framework has several limitations such as:

- Due to the technological context of the legacy FEA tool, the ATaaS framework is currently Windows-native, i.e. it cannot make use of Linux instances and tools (such as container orchestrators).
- Currently, the scaling of *Workers* needs to be carried out manually by the engineers, which is unsustainable for very large-scale analysis tasks with hundreds of *Workers* appearing or disappearing in a short amount of time.
- The endpoints for file-based operations and task-based operations are currently separate. Ideally, the entire service could have a unified access point for the end users.
- The endpoints are RESTful APIs which must be accessed using a HTTP client. Ideally the service endpoints would also be available as graphical web pages to facilitate users.
- While a failed *Executor* can be recovered, a failed *Worker* results in a deadlock due to the lack of a recovery mechanism for *Worker*.
- The *Licence Management Service* is a legacy program, which is a single point of failure for the entire system. A failure in licence management would bring down the entire ATaaS service.
- The *File Storage Service* does not restrict users from accessing all task folders.
- While the proposed solution is assumed to be deployed in secure (corporate) network, the ATaaS framework itself does not provide extra layers of security (such as user authentication).
- Certain parameters in the performance estimation models could benefit from a stochastic treatment (instead of using constants as in our simulations).

7.2 Future Work

The future work for this project in order of importance to the industrial partner Siemens Energy is discussed below; security, auto-scaling, performability model.

Security The primary service endpoint implemented by the *Manager* offers to security layers, whereas any requests it receives are implicitly trusted and processed. As future work, the *Manager* can implement various security functions into the RESTful API; such as user authentication and code injection protection. Users of the ATaaS can be authenticated so that they can queue and get information only on tasks they created. Furthermore, API requests can possibly be targeted with code injection attacks which the *Manager* can scan for and protect against.

Auto-Scaling The *Manager* developed in Section 3.1.1 serves as only the primary service endpoint for the ATaaS. As future work, the *Manager* can be further developed to enable cloud bursting whereas public cloud instances could be automatically provisioned and configured as ATaaS *Worker* nodes whenever the task queue becomes unreasonably long. This auto scaling functionality can further be used to perform health checks on *Workers* and recover failed nodes and their tasks.

Performability model The performance model developed in Section 5.2.1 predicts expected execution time by assuming that no faults (e.g. timeouts, node failures, communication failures) are present in the system. As future work, the performance model could be further developed into a performability model to investigate the effects of faults on performance. For that purpose, one could take the state machine of a task (Figure 3.4) as a baseline, and assign frequencies to events triggering the transitions in order to derive queuing networks or stochastic reward nets as formal performability models. The steady-state analysis of such Markovian models could then provide reasonable estimates for software services as shown in [29].

Bibliography

[1] ALBUQUERQUE JR, L. F., FERRAZ, F. S., OLIVEIRA, R., AND GALDINO, S. Function-as-a-service x platform-as-a-service: Towards a comparative study on FaaS and PaaS. In *ICSEA* (2017), pp. 206–212.

[2] ARCHITECH CORP. 5 reasons why you need to re-architect applications for the cloud. https://www.architech.ca/5-reasons-why-you-need-to-re-architect-applications-for-the-cloud/, Nov 2018.

[3] ARONSSON, A., AND NIE, L. Cloudifying a legacy batch-processing: A design science study.

[4] AWS. Amazon DynamoDB. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html, 2021.

[5] AWS. Amazon EC2. https://docs.aws.amazon.com/AWSEC2/latest/WindowsGuide/concepts.html, 2021.

[6] AWS. Amazon IAM. https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html, 2021.

[7] AWS. Amazon S3. https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html, 2021.

[8] AZEEZ, A., PERERA, S., GAMAGE, D., LINTON, R., SIRIWARDANA, P., LEELARATNE, D., WEERAWARANA, S., AND FREMANTLE, P. Multi-tenant SOA mid-

dleware for cloud computing. In *2010 IEEE 3rd International Conference on Cloud Computing* (2010), IEEE, pp. 458–465.

[9] BEIMBORN, D., MILETZKI, T., AND WENZEL, S. Platform as a service (PaaS). *Business & Information Systems Engineering 3*, 6 (2011), 381–384.

[10] BOURASSA, S. C., HOESLI, M., MERLIN, L., AND RENNE, J. Big data, accessibility and urban house prices. *Urban Studies* (2021), 0042098020982508.

[11] BUYYA, R., AND VENUGOPAL, S. A gentle introduction to grid computing and technologies. *CSI Communications 29* (11 2004).

[12] BUYYA, R., AND VENUGOPAL, S. The gridbus toolkit for service oriented grid and utility computing: an overview and status report. In *1st IEEE International Workshop on Grid Economics and Business Models, 2004. GECON 2004.* (2004), pp. 19–66.

[13] CAPPELLO, F., FEDAK, G., KONDO, D., MALÉCOT, P., AND REZMERITA, A. *Chapter 3: Desktop Grids: From Volunteer Distributed Computing to High Throughput Computing Production Platforms*, vol. 1. 07 2009, pp. 31–61.

[14] CHANNABASAVAIAH, K., HOLLEY, K., AND TUGGLE, E. Migrating to a service-oriented architecture. *IBM DeveloperWorks 16* (2003), 727–728.

[15] CHEN, C., LIU, Y., SUN, X., DI CAIRANO-GILFEDDER, C., AND TITMUS, S. Automobile maintenance prediction using deep learning with GIS data. *Procedia CIRP 81* (2019), 447–452.

[16] CHEN, Y.-W., HUNG, S.-H., TU, C.-H., AND YEH, C. W. Virtual Hadoop: Mapreduce over Docker containers with an auto-scaling mechanism for heterogeneous environments. In *Proceedings of the International Conference on Research in Adaptive and Convergent Systems* (2016), pp. 201–206.

[17] COATSWORTH, M. Condor for microsoft windows platform, November 2019.

[18] DI FRANCESCO, P., LAGO, P., AND MALAVOLTA, I. Architecting with microservices: A systematic mapping study. *Journal of Systems and Software 150* (2019), 77–97.

[19] DOCKER. Swarm mode overview. `https://docs.docker.com/engine/swarm/`, 2021.

[20] EVANS, E., AND FOWLER, M. *Domain-driven design tackling complexity in the heart of software*. Addison-Wesley, 2019.

[21] FIELDING, R. T., AND TAYLOR, R. N. *Architectural Styles and the Design of Network-Based Software Architectures*. PhD thesis, 2000. AAI9980887.

[22] FIKRI, N., RIDA, M., ABGHOUR, N., MOUSSAID, K., AND EL OMRI, A. Bigdata and regulation in high frequency trading. In *Proceedings of the 2017 International Conference on Cloud and Big Data Computing* (2017), pp. 45–49.

[23] FOSTER, I. Support for open source globus toolkit will end as of january 2018, May 2017.

[24] FOSTER, I., AND KESSELMAN, C. Globus: a metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing 11*, 2 (1997), 115–128.

[25] FOSTER, I., AND KESSELMAN, C. The globus project: a status report. In *Proceedings Seventh Heterogeneous Computing Workshop (HCW'98)* (1998), pp. 4–18.

[26] FRAZIER, N. J., WILLS, Z., AND CUI, S. National water model as a service: Towards a new frontier of community engagement. In *AGU Fall Meeting Abstracts* (2019), vol. 2019, pp. H52D–02.

[27] GILL, S. S., AND BUYYA, R. A taxonomy and future directions for sustainable cloud computing: 360 degree view. *ACM Computing Surveys (CSUR) 51*, 5 (2018), 1–33.

[28] Goldstein, I., Spatt, C. S., and Ye, M. Big data in finance. *The Review of Financial Studies 34*, 7 (2021), 3213–3225.

[29] Gönczy, L., Déri, Z., and Varró, D. Model transformations for performability analysis of service configurations. In *Models in Software Engineering, Workshops and Symposia at MODELS 2008* (2008), M. R. V. Chaudron, Ed., vol. 5421 of *LNCS*, Springer, pp. 153–166.

[30] Guo, K., Ren, S., Bhuiyan, M. Z. A., Li, T., Liu, D., Liang, Z., and Chen, X. MDMaaS: Medical-assisted diagnosis model as a service with artificial intelligence and trust. *IEEE Transactions on Industrial Informatics 16*, 3 (2019), 2102–2114.

[31] IBM. Microservices. `https://www.ibm.com/cloud/learn/microservices`, Mar 2021.

[32] IBM. REST APIs. `https://www.ibm.com/cloud/learn/rest-apis`, Apr 2021.

[33] Ionita, A. D. *Migrating Legacy Applications: Challenges in Service Oriented Architecture and Cloud Computing Environments: Challenges in Service Oriented Architecture and Cloud Computing Environments*. IGI Global, 2012.

[34] Iosup, A., and Epema, D. Grid computing workloads. *IEEE Internet Computing 15*, 2 (2011), 19–26.

[35] Jamshidi, P., Ahmad, A., and Pahl, C. Cloud migration research: a systematic review. *IEEE transactions on cloud computing 1*, 2 (2013), 142–157.

[36] Josep, A. D., Katz, R., Konwinski, A., Gunho, L., Patterson, D., and Rabkin, A. A view of cloud computing. *Communications of the ACM 53*, 4 (2010), 50–58.

[37] JULIAN, S., SHUEY, M., AND COOK, S. Containers in research: initial experiences with lightweight infrastructure. In *Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale* (2016), pp. 1–6.

[38] KESHAVARZ, H., MAHDZIR, A. M., TALEBIAN, H., JALALIYOON, N., AND OHSHIMA, N. The value of big data analytics pillars in telecommunication industry. *Sustainability 13*, 13 (2021), 7160.

[39] KNUUTINEN, A. *Formal connectivity verification of clock and reset signals in ultra-low-power SoC designs*. PhD thesis, University of Oulu, 2021.

[40] KUBERNETES. Kubernetes. `https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/`, Jul 2021.

[41] KYONG, J., JEON, J., AND LIM, S.-S. Improving scalability of apache spark-based scale-up server through docker container-based partitioning. In *Proceedings of the 6th International Conference on Software and Computer Applications* (2017), pp. 176–180.

[42] LAMPORT, L., AND LYNCH, N. Chapter 18 - distributed computing: Models and methods. In *Formal Models and Semantics*, J. VAN LEEUWEN, Ed., Handbook of Theoretical Computer Science. Elsevier, Amsterdam, 1990, pp. 1157–1199.

[43] LI, K.-C., HSU, C.-H., YANG, L. T., DONGARRA, J., AND ZIMA, H. *Handbook of Research on Scalable Computing Technologies 2-Volumes*. Information Science Reference - Imprint of: IGI Publishing, Hershey, PA, 2009.

[44] LI, L., ZHANG, Y., AND DING, Y. MT-DIPS: a new data duplication integrity protection scheme for multi-tenants sharing storage in SaaS. *International Journal of Grid and Utility Computing 9*, 1 (2018), 26–36.

[45] LI, Z., YANG, C., HUANG, Q., LIU, K., SUN, M., AND XIA, J. Building model as a service to support geosciences. *Computers, Environment and Urban Systems 61* (2017), 141–152.

[46] LITZKOW, M., LIVNY, M., AND MUTKA, M. Condor-a hunter of idle workstations. In *[1988] Proceedings. The 8th International Conference on Distributed* (1988), pp. 104–111.

[47] LIU, H., GAO, Q., LI, J., LIAO, X., XIONG, H., CHEN, G., WANG, W., YANG, G., ZHA, Z., DONG, D., ET AL. JIZHI: A fast and cost-effective model-as-a-service system for web-scale online inference at Baidu. *arXiv preprint arXiv:2106.01674* (2021).

[48] LOGAN, D. L. *A First Course in the Finite Element Method Using Algor*, 2nd ed. Brooks/Cole Publishing Co., USA, 2000.

[49] LV, S. Design of the automobile marketing system based on the big data. In *International Conference on Big Data Analytics for Cyber-physical Systems* (2019), Springer, pp. 1713–1719.

[50] MAMMO, B., FURIA, M., BERTACCO, V., MAHLKE, S., AND KHUDIA, D. S. BugMD: Automatic mismatch diagnosis for bug triaging. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2016), IEEE, pp. 1–7.

[51] MEDVEDEV, D., LEMSON, G., AND RIPPIN, M. Sciserver compute: Bringing analysis close to the data. In *Proceedings of the 28th International Conference on Scientific and statistical database management* (2016), pp. 1–4.

[52] MELL, P., GRANCE, T., ET AL. The NIST definition of cloud computing.

[53] MICROSOFT CORPORATION. Identifying microservice boundaries, Nov 2021.

[54] MORABITO, R. A performance evaluation of container technologies on internet of things devices. In *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)* (2016), IEEE, pp. 999–1000.

[55] NEDBAL, D., AND STIENINGER, M. Success factor analysis for cloud services: a comparative study on software as a service. *International Journal of Grid and Utility Computing 11*, 3 (2020), 315–329.

[56] Nguyen, D.-T., Yong, C. H., Pham, X.-Q., Nguyen, H.-Q., Loan, T. T. K., and Huh, E.-N. An index scheme for similarity search on cloud computing using mapreduce over docker container. In *Proceedings of the 10th International Conference on ubiquitous information management and communication* (2016), pp. 1–6.

[57] Novo, O., Beijar, N., Ocak, M., Kjällman, J., Komu, M., and Kauppinen, T. Capillary networks-bridging the cellular and IoT worlds. In *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)* (2015), IEEE, pp. 571–578.

[58] Pahl, C., Brogi, A., Soldani, J., and Jamshidi, P. Cloud container technologies: a state-of-the-art review. *IEEE Transactions on Cloud Computing 7*, 3 (2017), 677–692.

[59] Rangappa, M. *Analysis Tool as a Service: A Cloud-based Microservices Architecture for the Design and Analysis of Aero-derivative Gas Turbines*. McGill University (Canada), 2020.

[60] Razavian, M., and Lago, P. A systematic literature review on SOA migration. *Journal of Software: Evolution and Process 27*, 5 (2015), 337–372.

[61] Red Hat. IaaS vs PaaS vs SaaS. `https://www.redhat.com/en/topics/cloud-computing/iaas-vs-paas-vs-saas`, Apr 2020.

[62] Rehman, S., and Al-Raqom, D. Using big data in telecommunication companies: A case study. *African Journal of Business Management 14*, 7 (2020), 209–216.

[63] Sadovykh, A., Hein, C., Morin, B., Mohagheghi, P., and Berre, A. J. REMICS-reuse and migration of legacy applications to interoperable cloud services. In *European Conference on a Service-Based Internet* (2011), Springer, pp. 315–316.

[64] Sahu, I. K., and Nene, M. J. Model for IaaS security model: MISP framework. In *2021 International Conference on Intelligent Technologies (CONIT)* (2021), IEEE, pp. 1–6.

[65] Schipp, J., Dopheide, J., and Slagell, A. Islet: an isolated, scalable, & lightweight environment for training. In *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure* (2015), pp. 1–6.

[66] Segev, E., Goldshlager, S., Miller, H., Shua, O., Sher, O., and Greenberg, S. Evaluating and comparing simulation verification vs. formal verification approach on block level design. In *Proceedings of the 2004 11th IEEE International Conference on Electronics, Circuits and Systems, 2004. ICECS 2004.* (2004), IEEE, pp. 515–518.

[67] Sehgal, A. *CloudMEMS Platform for Design and Simulation of MEMS: Architecture, Coding, and Deployment*. PhD thesis, University of Toledo, 2018.

[68] Serrano, N., Gallardo, G., and Hernantes, J. Infrastructure as a service and cloud technologies. *IEEE Software 32*, 2 (2015), 30–36.

[69] Shamseddine, M., Nusayr, A., and Itani, W. Platform-as-a-service sticky policies for privacy classification in the cloud. *International Journal of Computer and Information Engineering 15*, 6 (2021), 410–413.

[70] Sneed, H. M. Integrating legacy software into a service oriented architecture. In *Conference on Software Maintenance and Reengineering (CSMR'06)* (2006), IEEE, pp. 11–pp.

[71] Suram, S., MacCarty, N. A., and Bryden, K. M. Engineering design analysis utilizing a cloud platform. *Advances in Engineering Software 115* (2018), 374–385.

[72] Thain, D., Tannenbaum, T., and Livny, M. Distributed computing in practice: the condor experience. *Concurrency and Computation: Practice and Experience 17*, 2-4 (2005), 323–356.

[73] Thönes, J. Microservices. *IEEE Software 32*, 1 (2015), 116–116.

[74] TSO, F. P., WHITE, D. R., JOUET, S., SINGER, J., AND PEZAROS, D. P. The glasgow raspberry pi cloud: A scale model for cloud computing infrastructures. In *2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops* (2013), IEEE, pp. 108–112.

[75] VERNON, V. *Implementing domain-driven design*. Addison-Wesley, 2013.

[76] WAN, Z., DUAN, L., AND WANG, P. Cloud migration: layer partition and integration. In *2017 IEEE International Conference on Edge Computing (EDGE)* (2017), IEEE, pp. 150–157.

[77] WANG, J., YANG, Y., WANG, T., SHERRATT, R. S., AND ZHANG, J. Big data service architecture: a survey. *Journal of Internet Technology 21*, 2 (2020), 393–405.

[78] WANG, K., ZHOU, X., QIAO, K., LANG, M., MCCLELLAND, B., AND RAICU, I. Towards scalable distributed workload manager with monitoring-based weakly consistent resource stealing. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing* (New York, NY, USA, 2015), HPDC '15, Association for Computing Machinery, p. 219–222.

[79] WU, Y., RAO, R., HONG, P., AND MA, J. Fas: A flow aware scaling mechanism for stream processing platform service based on lms. In *Proceedings of the 2017 International Conference on Management Engineering, Software Engineering and Service Sciences* (2017), pp. 280–284.

[80] YEN, I.-L., BASTANI, F., HUANG, Y., ZHANG, Y., AND YAO, X. SaaS for automated job performance appraisals using service technologies and big data analytics. In *2017 IEEE International Conference on Web Services (ICWS)* (2017), IEEE, pp. 412–419.

[81] YOO, A. B., JETTE, M. A., AND GRONDONA, M. Slurm: Simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing* (Berlin, Heidelberg, 2003), D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds., Springer Berlin Heidelberg, pp. 44–60.