Analysis Tool as a Service: A Cloud-based Microservices Architecture for the Design and Analysis of Aero-derivative Gas Turbines

Maruthi Rangappa

Department of Electrical and Computer Engineering

McGill University, Montréal

Canada

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

Master of Science (MSc.)

Abstract

The design of complex cyber-physical systems such as aero-derivative gas turbines (AGT) at Siemens Canada, requires the execution of multidisciplinary workflows in which computationally intensive analysis tools are run along with various design workflows. A workflow-based tool integration framework facilitates the chaining of tools in the workflow, minimizes data transfer and conversion, and allows for automatic tool invocation using tool connectors.

The existing in-house tool integration framework at Siemens Canada necessitates the local execution of the analysis tools, which results in limited performance for complex analysis tasks due to limited computing resources. When an analysis tool has to be run for multiple configurations originating from design parameters, a sequential execution method is currently used, where the analysis of the next configuration only starts once the analysis of the previous configuration is completed. The main objective of the current thesis is to propose a framework that supports the distributed parallel execution of analysis tools to reduce the overall execution time of the analysis.

I aim to address these issues with cloud-based microservices architectures for workflow-based tool integration framework and distributed parallel execution of analysis tools using replicated instances of web services providing the tool functionalities and accessible via RESTful APIs. As a proof-of-concept, I implemented web services for wrapping two analysis tools - a Secondary Air System analysis tool and a Finite Element Analysis tool - used in the design of AGTs as web services, and service orchestration module to deploy and manage these services. The evaluation performed using these services prove that cloud-based microservices contribute to reducing the execution times by running multiple analyses in parallel on distributed cloud infrastructure despite the overhead of data transfer and service orchestration.

Abrégé

La conception de systèmes cyber-physiques complexes tels que les turbines à gaz aérodérivées (AGT) chez Siemens Canada, nécessite l'exécution de flux de travaux multidisciplinaires dans lesquels des outils d'analyse intensifs en calcul sont exécutés. Un cadre d'intégration d'outils basé sur le flux de travail facilite le chaînage des outils dans le flux de travail, minimise le transfert et la conversion de données et permet l'appel automatique d'outils à l'aide de connecteurs d'outils. Le cadre d'intégration d'outils existant à Siemens Canada nécessite l'exécution locale des outils d'analyse, ce qui se traduit par des performances limitées pour des analyses complexes en raison des ressources informatiques limitées. Lorsqu'une analyse doit être exécutée pour plusieurs configurations provenant de paramètres de conception, une méthode d'exécution séquentielle est utilisée, où l'analyse de la configuration suivante ne démarre qu'une fois l'analyse de la configuration précédente terminée. L'objectif principal de cette thèse est de proposer un cadre qui supporte l'exécution parallèle distribuée d'outils d'analyse pour réduire le temps d'exécution global de l'analyse. Dans ce mémoire, je vise à résoudre ces problèmes avec des architectures de microservices basées sur le nuage dans le cadre d'intégration d'outils basés sur le flux de travail et l'exécution parallèle et distribuée d'outils d'analyse à l'aide d'instances répliquées de services Web fournissant les fonctionnalités de l'outil et accessibles via des API RESTful. En guise de preuve de concept, j'ai implémenté des services Web pour encapsuler deux outils d'analyse - un outil d'analyse du Système d'Air Secondaire et un outil d'analyse par éléments finis - utilisés dans la conception des AGT et un module d'orchestration des services pour déployer et gérer ces services. L'évaluation réalisée à l'aide de ces services prouve que les microservices basés sur le nuage contribuent à réduire les temps d'exécution en exécutant plusieurs analyses en parallèle sur

une infrastructure cloud distribuée malgré la surcharge du transfert de données et de l'orchestration des services.

Acknowledgements

Foremost, I would like to thank my supervisor Prof. Dániel Varró, for his continuous 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 program.

I would like to express my gratitude to Mr. Martin Staniszewski of Siemens Canada, 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 who supported me through this research.

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

Finally, I am extremely grateful to my family for their continuous support and encouragement through all kinds of situations during my study.

Contents

Abstract i

Abrégé ii

Acknowledgements iv

List of Figures xi

1 Introduction **1**

1.1 Context and Motivation 1

1.2 Objectives and Contributions 3

1.2.1 Objectives 4

1.2.2 Contributions 4

1.3 Thesis Outline 5

2 Background: Tool Integration Workflows at Siemens Canada **6**

2.1 Introduction to Gas Turbines 7

2.2 Gas Turbine Components 9

2.3 Design of Power Generation Gas Turbines 11

2.3.1 Gas Turbine Design Process 12

2.3.2 Design Workflows at Siemens 13

2.4 Workflow Based Tool Integration Frameworks 15

2.4.1 Tool Integration Aspects 16

2.4.2 Existing Tool Integration Framework at Siemens 17

2.4.3 Architecture of Existing Tool Integration Framework 20

2.4.4 Discussion of Challenges 20

2.5 Summary 23

3 Background: Concepts and Technologies **24**

3.1 Cloud Computing Service Models . . . 24

3.2 Software as a Service (SaaS) . . . 26

3.2.1 Introduction . . . 26

3.2.2 Architecture . . . 27

3.2.3 Benefits . . . 27

3.3 Microservices . . . 29

3.3.1 Introduction . . . 29

3.3.2 Architecture . . . 30

3.3.3 REST APIs . . . 32

3.3.4 Benefits . . . 33

3.4 RESTful Microservices Development Frameworks . . . 35

3.4.1 Flask-RESTful Web Service Framework . . . 35

3.4.2 Task Queue Framework . . . 35

3.5 Docker Containers . . . 37

3.5.1 Containers . . . 37

3.5.2 Docker Ecosystem . . . 38

3.6 Docker Swarm Cluster Manager . . . 38

3.6.1 Cluster Management and Service Orchestration . . . 39

3.6.2 Docker Swarm . . . 40

3.7 Continuous Integration and Delivery . . . 40

3.8 Summary . . . 42

4 Service Architecture for Tool Integration Framework **44**

4.1 Service Architecture . . . 44

4.1.1 Architectural Components . . . 45

4.1.2 Service Cluster . . . 47

4.1.3 Workflow Execution as a Service . . . 48

4.1.4 Analysis Tool as a Service . . . 49

4.2 Concepts of Tool Services . . . 50
4.3 Software Architecture of the ATaaS Prototype . . . 52
4.3.1 Tool Services . . . 53
4.3.2 Execution Manager . . . 56
4.3.3 Service Provisioning . . . 56
4.3.4 Load Balancer Configuration Management . . . 57
4.4 Summary . . . 58

5 Development of ATaaS Prototype **59**
5.1 Tool Service APIs . . . 59
5.1.1 Create an Analysis Task . . . 60
5.1.2 Get Task Information . . . 61
5.1.3 Upload Input Files . . . 61
5.1.4 Starting a Task . . . 62
5.1.5 Get Task Status . . . 62
5.1.6 Download Analysis Results . . . 63
5.2 Development of Tool Services . . . 63
5.2.1 Flask Web Service Instance Creation and Initialization . . . 64
5.2.2 Celery Task Queue Instance Creation and Initialization . . . 66
5.2.3 Analysis Tool Executors . . . 66
5.2.4 Service Methods . . . 69
5.3 Deployment of Tool Services . . . 73
5.3.1 Docker Images for Tool Services . . . 73
5.3.2 Cluster Management . . . 76
5.3.3 Service Management . . . 78
5.3.4 LB Config Management . . . 78
5.4 Development of Execution Manager . . . 79
5.5 Summary . . . 81

6 Performance Evaluation **83**

6.1 Research Questions . . . 83

6.2 Benchmarking Setup and Execution Methodology . . . 84

6.2.1 Benchmarking Setup . . . 84

6.2.2 Hardware and Software . . . 86

6.2.3 Execution Methodology . . . 87

6.3 Analysis of Results . . . 89

6.3.1 RQ1: Effect of ATaaS execution on single local computer . . . 89

6.3.2 RQ2: Effect of parallel distributed ATaaS . . . 90

6.3.3 RQ3: Effect of Increasing Service Replicas . . . 91

6.4 Summary . . . 92

7 Related Work **94**

7.1 Tool Integration for Design of Cyber-Physical Systems . . . 94

7.2 Workflow Execution as a Service . . . 96

7.3 Analysis Tool as a Service . . . 98

8 Conclusions and Future Work **100**

8.1 Conclusions . . . 100

8.1.1 Conclusions from architecture for tool integration framework . . . 100

8.1.2 Conclusions of architecture for analysis tool services . . . 101

8.1.3 Conclusions by performance evaluation . . . 102

8.2 Future Work . . . 103

Bibliography **104**

List of Figures

2.1 Sections in an Aero-derivative Gas Turbine. Image source [49] 9
2.2 High-Level GT design workflow at Siemens AGT 15
2.3 Structure of a sub-workflow 16
2.4 Aspects of tool integration 17
2.5 Domain model of the main concepts of Siemens tool integration framework 19
2.6 AGT workflow structure 19
2.7 Architecture of the existing framework 21

3.1 IaaS vs PaaS vs SaaS. Image Source [25] 25
3.2 SaaS access and deployment architecture 28
3.3 RESTful Microservices and their interactions 31
3.4 REST URI example 33
3.5 Celery task queue [69] 36
3.6 Docker container ecosystem. Image Source [8] 39
3.7 Docker swarm architecture. Image Source [9] 41
3.8 CI/CD with Docker ecosystem. Image Source [26] 42

4.1 Service Architecture for WEaaS and ATaaS at Siemens AGT 45
4.2 Architecture for workflow execution as a service (WEaaS) 48
4.3 ATaaS architecture 50
4.4 Concepts of tool services 51
4.5 Software architecture 52

5.1 REST APIs of SAS and FEA tool services 60
5.2 API for creating an analysis task 60

5.3 API response for creating an analysis task . . . 61
5.4 API response for get task information . . . 61
5.5 API for uploading input files . . . 62
5.6 API for getting task status . . . 63
5.7 Download analysis results . . . 63
5.8 Modules of tool service . . . 65
5.9 Tool service creation and initialization . . . 66
5.10 SAS analysis flowchart . . . 68
5.11 FEA analysis flowchart . . . 69
5.12 Service method for creating task . . . 70
5.13 Service methods for receiving input files . . . 70
5.14 Service methods for starting the task . . . 71
5.15 service method for getting task status . . . 72
5.16 Download analysis results . . . 72
5.17 Software modules facilitating deployment of tool services . . . 74
5.18 Dockerfile statement for installing tool dependencies . . . 75
5.19 Setting up virtual environment . . . 75
5.20 Cluster management flowchart . . . 77
5.21 Cluster configuration file . . . 78
5.22 Method of creating a service . . . 79
5.23 Load balancer configuration management . . . 80
5.24 Execution manager module . . . 81

6.1 Benchmarking setup . . . 85
6.2 Hardware configuration of the benchmarking setup . . . 86
6.3 Software tools used for benchmarking . . . 87
6.4 Mean execution times for local vs one instance of ATaaS . . . 90
6.5 Execution times for multiple analysis tasks: local vs ATaaS . . . 91
6.6 Execution times: Local vs 4 replicas vs 8 replicas of ATaaS . . . 92

7.1 Architecture of workflow-based tool integration frameworks. Source [3] . . 95

Chapter 1

Introduction

1.1 Context and Motivation

The designing of complex cyber-physical systems (CPS) such as aero-derivative gas turbines (AGT) at Siemens Canada, involves experts from multiple disciplines working collaboratively to achieve a common design goal. Experts in each discipline perform various design and analysis activities focusing deeply on a specific aspect of the gas turbine and share models and design data across the disciplines. Many commercial and proprietary engineering tools are used by each discipline to perform their design activities that involve design analysis, validation, and testing.

Design Workflow: In order to streamline the design activities, Siemens Canada has established a gas turbine design workflow involving all disciplines with in the organization. The workflow is hierarchical and involves many intra and inter-disciplinary workflows having workflow steps that require execution of computationally intensive analysis tools. In the design workflow, it is often required that the results obtained from one design step need to be used as input in another design step and involves data conversion and requires unambiguous labelling of data.

In-house workflow based tool integration framework: An in-house workflow-based tool integration framework is used at Siemens Canada to execute these workflows. The framework facilitates the chaining of tools in the design workflow and minimizes manual data transfer, conversion and labelling, and allows for automatic tool invocation using

tool connectors. The framework supports collaborative workflow execution and data exchange between disciplines by using a centralized server for storing the workflows and the associated metadata.

Practical Limitations of In-house Tool Integration Framework

Besides all the advanced features offered by the in-house tool integration framework, it has certain practical limitations that affect the overall performance of the framework.

Limitations due to fixed computing resources: The tool integration framework is installed either on the engineer's laptop or a shared network location and runs locally on the engineer's computer. It requires a local installation of the analysis tools and results in limited performance due to the use of limited computing resources available for the execution of computationally intensive analysis tools.

Limitations due to sequential execution: When computationally intensive analysis tools need to be run for multiple configurations, a sequential iterative execution method is used, which takes a long time to give results for all configurations and prolongs the overall design process time as results from the analysis of multiple configurations is often required to validate a design decision.

Possible Solutions

Use of powerful computing resources: One possible solution to overcome resource limitations is to run the tool integration framework on computationally more powerful workstation computers. Although this method gives a better performance to a certain extent, it leads to inefficient use of resources as the workstations could be used by only one user at a time, and if not used by anyone, it stays idle. Moreover, the performance gain is very minimal when analysis for a large number of configurations need to be run.

Multiple instance execution: Another option to reduce the execution time is to run the multiple instances of the tool integration framework on dedicated computers to invoke multiple instances of the analysis tools with different configurations. This solution would require dividing the execution of configurations across many workflows and manual management of multiple instances of tool integration framework during execution. Later, results from different executions need to be manually consolidated to get complete results.

Multi-threading or Multi-processing: The third possible solution to boost the performance could be to implement multi-threading or multi-processing in tool connectors to execute analysis tools in parallel for many configurations to reduce the overall execution time for multiple configurations. This method leads to the maximum utilization of the computing resources for the tool integration framework and the analysis tools, and starvation of other applications on the same workstation, which is not an efficient use of resources in all cases.

Cloud-based web services: The forth possible solution is to provide services of the tool-integration framework and analysis tools through web services. Web services can be hosted on private or public cloud-based computing resources and scaled dynamically to meet the cost-time requirements of executing a multiple-configuration analysis.

1.2 Objectives and Contributions

In this thesis, we explore the possibility of adopting the architecture of the cloud-based web services (forth solution from the above) to address the limitations as the cloud-based web services are known for their scalability, performance, and multiplicity.

1.2.1 Objectives

We identified the following objectives to address the limitations described above.

1. **Architecture for workflow-based tool integration framework:** This thesis aims to propose an architecture for a workflow-based tool integration framework that
   - Addresses the infrastructure limitations of in-house tool integration framework by defining a flexible, scalable execution scheme
   - Integrates with and reuses the in-house tool integration framework as much as possible
2. **Architecture for parallel execution of analysis for multiple configurations:** This thesis aims to propose a scalable architecture for parallel execution of analysis for many configurations to reduce the overall execution time.
3. **Prototype implementation using the proposed architecture:** This thesis aims to develop a prototype to demonstrate the feasibility of the proposed architecture.

1.2.2 Contributions

This section summarizes the contributions of this thesis

1. **Cloud-based microservice architecture for tool integration framework:** In collaboration with my supervisor and Siemens engineers, I designed a microservices architecture for the workflow-based tool integration framework. The architecture uses atomic and independent microservices to host functionalities of the tool integration framework and analysis tools as web services that are accessible via RESTful APIs.
2. **Cloud-based microservices architecture for parallel distributed execution of analysis tools:** In collaboration with my supervisor and Siemens project managers and engineers, I designed a microservices architecture that has analysis tools running in

a containerized environment on cloud infrastructure to achieve parallel distributed execution of analysis for multiple configurations.

3. **Development of analysis tool services and service orchestration module:** I developed web services for two analysis tools by adopting the proposed architecture for parallel distributed execution. A service orchestration module was developed to manage the deployment, scalability and availability of the analysis services.

4. **Performance evaluation of analysis services:** I performed scalability evaluation of the analysis services by using a custom test setup and documented the observations.

1.3 Thesis Outline

In general, this thesis is written in an impersonal style (dominantly in third person singular or passive sentences). When I wish to emphasize my own contribution, first person singular sentence is used. When I wish to emphasize the joint contributions or decisions made together with my supervisor and project managers first person plural is used.

Chapter 2 provides an overview of the multidisciplinary design workflows executed for the design of complex cyber-physical systems and workflow-based tool integration frameworks. **Chapter 3** provides an overview of the concepts of the cloud-based web services applicable in the context of this thesis and the relevant tools and technologies used in this thesis. **Chapter 4** describes the cloud-based microservices architecture for the workflows-based tool integration framework and analysis tool services. **Chapter 5** describes the software architecture of analysis tools services and important modules of the analysis tool services. **Chapter 6** reports observations of the performance evaluation experiments conducted using the analysis tool services. **Chapter 7** outlines the related work and **Chapter 8** presents the conclusion and future work.

Chapter 2

Background: Tool Integration Workflows at Siemens Canada

Gas Turbines (GT) are widely used in various power generation applications primarily due to their flexibility and their wide output power range from 3MW to 600MW. Since their introduction into the power generation segment, gas turbines have evolved into very complex cyber-physical systems (CPS). The design of such complex gas turbines systems is a very challenging task and involves experts from many disciplines working together to complete complex design workflows and need a wide range of tools and powerful computing resources. This is the exact situation in one of the Siemens Energy (formerly, Siemens Power Generation (PG)) divisions, which produces Aero-derivative Gas Turbines (AGT) gas turbines, that have an output power ranging up to 65MW [50]. Recent advancements in software and computer engineering can be leveraged to increase the efficiency of this design process by increasing collaboration and by integrating tools to create efficient and automated workflows.

This chapter provides the background information necessary to understand the challenges faced by Siemens AGT division in the gas turbine design process and how the tool integration practices and frameworks can help to over come many of these challenges.

2.1 Introduction to Gas Turbines

Gas turbines are internal combustion engines that convert the chemical energy of the fuel into either mechanical energy that can turn shafts to produce electric current or to kinetic energy that can generate thrust to propel aircrafts [53]. The gas turbines were originally developed during the 1930s and 1940s as an aviation engine. Later in the early 1980s, they entered the power generation market as standby power generators and for peak power support on power grids.

Gas Turbine in Aviation

All modern civil and military aircrafts, helicopters and business jets use gas turbines because of their superior thrust to weight ratio. The use of gas turbines in aviation applications has helped the aviation industry to increase the fuel efficiency per passenger seat by 70 percent over the last decades[42]. The gas turbines are also playing a very critical role in reducing the level of CO_2 emissions of the aviation industry to meet the requirements of the regulatory bodies such as the Advisory Council for Aviation Research and Innovation in Europe (ACARE)[42].

Gas Turbine in Power Generation

Gas turbines are widely used in various power generation applications firstly due to their operational responsiveness to support peak power demands and standby power generation needs as gas turbines can be quickly brought in to and out of operation. Secondly, gas turbines are easy to set up compared to other conventional power generation methods such as thermal power plants as gas turbines do not need pre-processing stages such as coal burners or nuclear reactors to generate heat required to produce steam. Finally, gas turbines are available for small to large power generation scenarios and cover a wide spectrum of industries ranging from oil and gas to large power generation plants as gas turbines have a wide range of output power capacity (3M to 400MW). For these same

reasons, gas turbines play a very critical role in the recent widespread transition towards renewable energy sources as renewable energy sources such as wind and tidal power generation can not provide stable power due to the very nature of the input sources used for power generation, for example, not enough wind is blowing on a given day. In all cases, when renewable energy sources fail to give stable power, gas turbines are used to generate auxiliary power needed to provide stable output power. In addition to aircraft propulsion and domestic power generation gas turbines are also used in many other industries such as[50]

- The oil and gas industry to produce the electricity required for driving the onshore and offshore oil production. GTs are also used to compress the gas into pipelines to transport it over long distances.
- The chemical and fiber, cement, metals, and mining, as well as other manufacturing industries for power generation and cogeneration

Gas Turbine Evolution

Since their initial development as engines that power the aircrafts, the gas turbines have evolved a great amount with the advancements made in the mechanical, electrical, computer and software engineering. Today, they are very complex CPS that operate at high efficiencies, lower investment, and lesser emissions. The gas turbine manufacturers continue putting their efforts to further improve on already achieved performance and emissions goals by adopting the industry 4.0 technologies to bring in more digitalization and smartness into the gas turbines and by automating design and analysis processes through tool integration technologies.

The following sections of this chapter will give an overview of the gas turbine design process, and design and analysis workflows used in designing aero-derivative gas turbines at Siemens power generation division, which is the industry partner for this re-

search work. A later section of this chapter provides an overview of the workflow based tool integration technologies.

2.2 Gas Turbine Components

It is essential to have an overview of the main components of a gas turbine to understand the gas turbine design steps comprehensively. A gas turbine has 3 main components: 1) Compressor, 2) Combustor and 3) Turbine

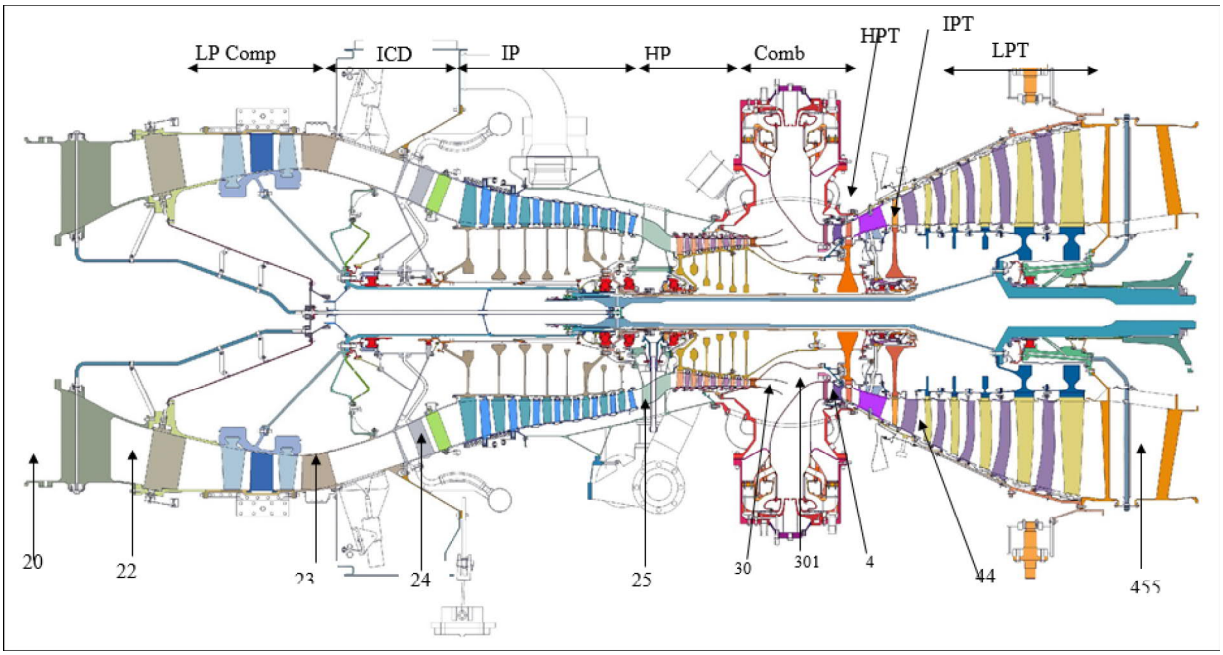


Figure 2.1: Sections in an Aero-derivative Gas Turbine. Image source [49]

Compressor

The compressor, which is located near the inlet of the gas turbine, is used to increase the pressure of the air. The compression happens because the speed of the air is increased as it passes through the stator and rotor blades of the compressor, and more and more incoming air is pressed into the tiny gaps between the rotors and stators. This high-pressure air is necessary for efficient combustion of the fuel in the combustor of the gas

turbine. A compressor in a turbine may have up to 3 sections as shown in Figure 2.1, namely: Low-Pressure Compressor (LP Comp), Intermediate Pressure (IP) compressor, and High-Pressure (HP) compressor. These multiple sections of the compressors increase the pressure of air gradually to a very high point and are mixed with the fuel before entering into the combustion chamber. Each section of the compressor will typically have many stages (One set of rotor and stator blades in the compressors are referred to as a stage). The number of stages needed in the turbine is decided based on the size of the compressor and the air pressure and temperature needed at the output of the compressor.

Combustor

The mixture of highly compressed air and fuel is burnt inside the combustor to produce a jet of hot air at high velocity to move the turbine. The type of combustor used in a gas turbine depends on the type of fuel (natural gas, diesel, or kerosene) that the gas turbines burn. The most common types of combustor used in the gas turbine industry are annular combustor, Dry Low Emission (DLE), and Wet Low Emission (WLE) combustor. The efficiency of the gas turbine depends mostly on how well the fuel is burnt in the combustor.

Turbine

The high-velocity hot air coming out of the combustor moves through the turbine sections before exiting the gas turbine, acting on the airfoil-shaped turbine blades on its way out. The reaction caused by this movement turns the turbine mounted on a shaft, which is also connected to the shaft of a generator. Like the compressor, the turbine is also constructed in multiple sections. The section of the turbine that is close to the combustor is called High-Pressure Turbine (HPT) and is always subjected to very high temperatures that are above the melting point of the metals used in the making of these turbine blades. These blades are cooled by applying cooling techniques to prevent them from melting. To make the most use of the high velocity available near the combustor, HPT blades densely

populated. A large portion of the gas turbine design efforts is concentrated around the design of the HPT turbine blades due to the complexity involved in designing the cooling passages. The HPT section is followed by an Intermediate Pressure Turbine (IPT) and Low-Pressure Turbines (LPT). Blades on these turbine sections are not cooled as they are subjected to reduced temperatures because the air loses the heat as it passes through the HPT section. It is to be noted that while the compressors are upstream (low to high), turbines are downstream (high to low) and each compressor stage is driven by the corresponding turbine stage, i.e., HPC is driven by HPT, IPC is driven by IPT and LPC is driven by LPT.

2.3 Design of Power Generation Gas Turbines

The early gas turbines were highly similar to the gas turbines used in the aviation industry and used the same or similar components as their siblings. Consequently, they were called aero-derivative gas turbines. As the demand for gas turbines in the power generation industry grew exponentially, more and more gas turbines were designed specifically to meet the needs of the power generation industry and were called industrial gas turbines. These industrial gas turbines were bigger in construction than their aero-derivative counterparts as the size and weight were not a concern for stationary applications such as power generation. Today both aero-derivative gas turbines and industrial gas turbines co-exist in the market and serve different power generation needs.

Siemens in Power Generation

Siemens has been in the power generation market since the beginning and has established itself as a significant player in the market in both industrial gas turbine and aero-derivative gas turbine segments. The company has a dedicated division called Power Generation (PG), which is specialized in the production of both types of turbines. Over the years, PG division has acquired a considerable amount of specialized knowledge that

is possessed only by few in the industry. Today, Siemens PG has established standard design steps that help them to produce efficient and competitive gas turbines in the market. The division has also developed many powerful and efficient design and analysis tools that help teams to execute various design steps to achieve high productivity.

2.3.1 Gas Turbine Design Process

A typical gas turbine design starts with a specification prepared by the market research experts based on the facts, current and future trends that they gathered, and the requirements derived from the customer needs and demands [42]. After some preliminary studies about the required configuration, existing designs, and design feasibility, the specification is passed on for detailed design by specialized disciplines. Based on the nature of the work, tools used, and the domain expertise of the involved engineers, Siemens PG has identified the engagement of nine different disciplines in the AGT design process.

Involved Disciplines

1. **Combustion**: The combustion discipline is involved at the beginning of the design cycle in helping the performance discipline to create an initial performance model. Further, it is involved in the design and development of the combustion chamber for the gas turbine

2. **Design**: The design discipline is a group of mechanical designers who design gas turbine components and are involved throughout the gas turbine design process

3. **Performance**: The performance discipline plays a vital role in designing the performance model at the beginning of the design process. The performance model is a thermo-mechanical model providing pressures, temperatures, and mass flow at every section of the engine as well as shaft rotational speeds.

4. **AeroThermal**: Experts in this discipline provide aerodynamic and cooling design and analysis of compressors and turbines using state of the art tools and methods

5. **Secondary Air System (SAS)**: The SAS engineers perform analyses that determine whether the SAS will be able to provide the required amount of cooling air for blades and vanes and sealing to the turbine components at a given operating point. Besides, SAS engineers are responsible for bearing load calculations to assess the integrity of bearings.

6. **Thermo-Mechanical**: This discipline performs thermal analysis on the entire engine and produces useful information for mechanical design and lifing disciplines to perform their analysis.

7. **Mechanical Modelling**: Mechanical modeling team performs the Finite Element Analysis (FEA) on the static structural parts such as casings, bearing housings, and mounts to understand the critical engine operating speeds and loads.

8. **Stress Analysis**: The engineers of this discipline perform stress analysis on the various turbine components particularly turbine blades

9. **Lifing**: The experts of this team perform different analyses to determine the life of gas turbine components for the given performance attributes

2.3.2 Design Workflows at Siemens

In the detailed design phase, disciplines perform design activities concentrated on a specific aspect of the gas turbine. While disciplines such as combustion, aero-thermal, and SAS perform design activities focused on their respective components, disciplines such as performance and thermo-mechanical perform design activities focused on the entire gas turbine. Thus the detailed design process is very segregated but requires interaction between disciplines to communicate the design decisions and often involves sharing lot of design data.

A formal design workflow is established within Siemens AGT to facilitate the collaboration between disciplines and to streamline the gas turbine design process with in the

organization. The workflow is hierarchical and iterative in nature with a high-level workflow facilitating collaboration while low-level workflows are focused on iterative design analysis tasks. This section gives an overview of these workflows.

High-level Workflow

Figure 2.2 shows a simplified end-to-end gas turbine design workflow executed at Siemens AGT. The workflow is simplified by excluding the manufacturing and real engine testing because the design iterations triggered by these two design steps are minimum. The workflow focuses more on the design steps - that are iterative and central to the detailed design - ranging from performance analysis to assessment of the life of critical components (lifing).

The disciplines in the workflow are arranged in the order of their occurrence in the design process. The *combustion* discipline helps the *performance* team to design the performance model at the beginning of the workflow. Since the combustion chamber is a complex GT module, it doesn't get modified frequently. Since it takes a long time to design a new combustion chamber, it uncommon to see new combustion chambers for every version of GT. Consequently, The performance discipline is generally placed at the beginning of the workflow.

The major section of the workflow has a modified waterfall structure. The output of one step is used as the input to the next step, and feedbacks are provided to the previous steps. The *mechanical modeling* discipline has involvement only with the *stress* discipline, and the *design* discipline is present throughout the workflow and supports all disciplines by providing mechanical designs.

Low-level Workflows

The end-to-end high-level workflow is subdivided into many low-level workflows called low-level workflows. The general structure of a low-level workflow is shown in Figure 2.3. Executing a low-level workflow always involves running a design analysis tool -

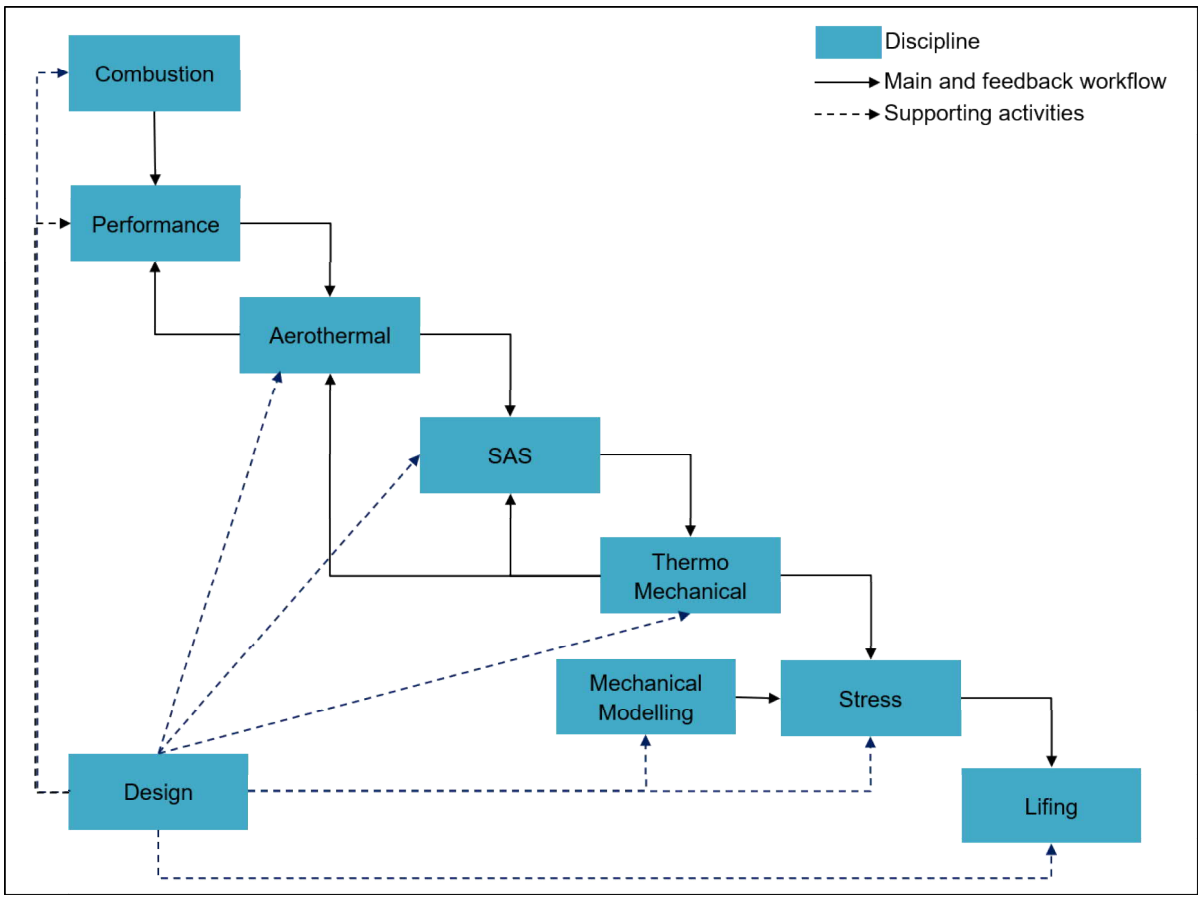


Figure 2.2: High-Level GT design workflow at Siemens AGT

represented as *Process* - on a given set of *inputs* to produce a set of *outputs*. The low-level workflows capture tasks executed by disciplines in greater detail and can be executed independently of other low-level workflows provided all required inputs are available.

A low-level workflow of the discipline, which is located at a higher step, can execute a new analysis when another dependent low-level workflow located at a lower step is working on the previous versions of the data. At Siemens AGT, low-level workflows are executed on a proprietary workflow-based tool integration framework.

2.4 Workflow Based Tool Integration Frameworks

At Siemens AGT, experts of each discipline use different tools along the workflow steps to perform their design activities. A design activity generally involves the use of the output

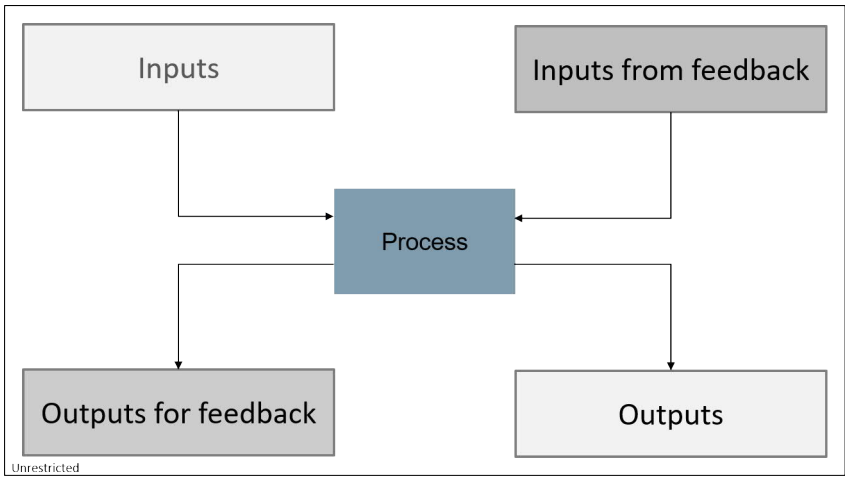


Figure 2.3: Structure of a sub-workflow

produced by a tool at one workflow step as the input of another tool in the next workflow step. One could manually run the first tool and take the produced results, convert them to the format needed by the second tool and manually run the second tool with the updated input. In the light of how computers and programming have changed the way engineering activities are performed in industries, the natural impulse is to automate such activities and could be referred to as tool integration [66].

2.4.1 Tool Integration Aspects

When many workflow based tool integration use cases are considered, one could see that there are many aspects common to all tool integration needs. Figure 2.4 shows few of those aspects which are most relevant in the context of the gas turbine design at Siemens AGT.

Data Exchange in the context of the workflow based tool integration covers the needs of chaining the output of one tool to the input of another tool. When the there is a difference in the format (file type, data format etc.) of the output produced by the first tool and the input needed by the second tool, a *Data Conversion* step is involved in tool integration. In an iterative design process, input used by an analysis tool may change due to the change in design decisions or by incorporating feedbacks originating other analysis results. It is

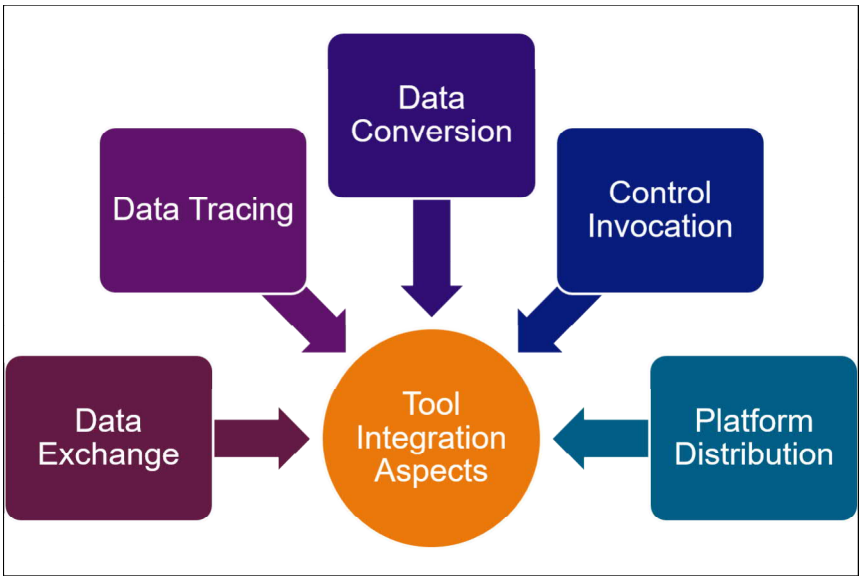


Figure 2.4: Aspects of tool integration

necessary to keep track of input revisions and corresponding output and feedback data by labelling the data appropriately. The *Data Tracing* aspect of the tool integration addresses this requirement. The *Control Invocation* aspect of the tool integration refers to the automatic execution of a tool in the workflow when all required input conditions are met. The *Platform Distribution* aspect of the tool integration refers to the desired benefit of executing tools on heterogeneous computing platforms (different OS and hardware infrastructure).

2.4.2 Existing Tool Integration Framework at Siemens

Integration of two or more tools could be done by writing scripts such as MS Dos batch scripts or Unix shell scripts or by writing specific applications in high-level programming languages such as C++ or Python to address a specific tool integration requirement. Such need based tool integration practices results in ad-hoc tool integration solutions which may be error prone and unstable.

Realizing these issues, and to bring in the benefits resulting from the tool integration aspects Siemens has developed an in-house tool integration framework. An in-house framework offers greater flexibility in workflow design and execution as it can be modi-

fied to meet the specific needs of the group. This section gives an overview of workflow design and execution activities involved in the context of the existing tool integration framework.

Concepts of Siemens Tool Integration Framework

The Figure 2.5 shows a domain model of the main concepts of the Siemens tool integration framework. The *Server* is the central database in which workflow projects, workflow revisions, and data revisions are stored. The *Project* represents a group of workflows related to one AGT engine. An engine has many components such as compressor, turbine and is represented as the *Component* in the domain model. An engineer of *Discipline* working on the component creates analysis *Tasks* composed of many *Designs* which have many *Revisions*. Every design has a *Workflow* and is globally defined in the framework using a *WorkflowDefinition* file. A workflow consists of many configurable *WorkflowSteps*. Each workflow step has one or more *Input*, *Output*, and a *Tool*, and is defined using a *WorkflowStepDefinition* file which contains parameters to specify the nature of execution and the target tool.

Workflow Design in Siemens Tool Integration Framework

In workflow design stage, a Siemens engineer composes a workflow using a workflow definition file. An example workflow composed using this framework is shown in Figure 2.6. A workflow consists of an *Inputs* block, a *Workflow Step (Performance Tool)*, and an *Outputs* block. The workflow step has configuration parameters to specify the analysis tool to be run, the mode of execution (batch or interactive) etc. At this stage, the engineer uses many design objects provided by the framework and performs the following:

1. Configures one or many tools to take designated inputs to produce desired outputs.

2. Develops modules necessary to run the tool.

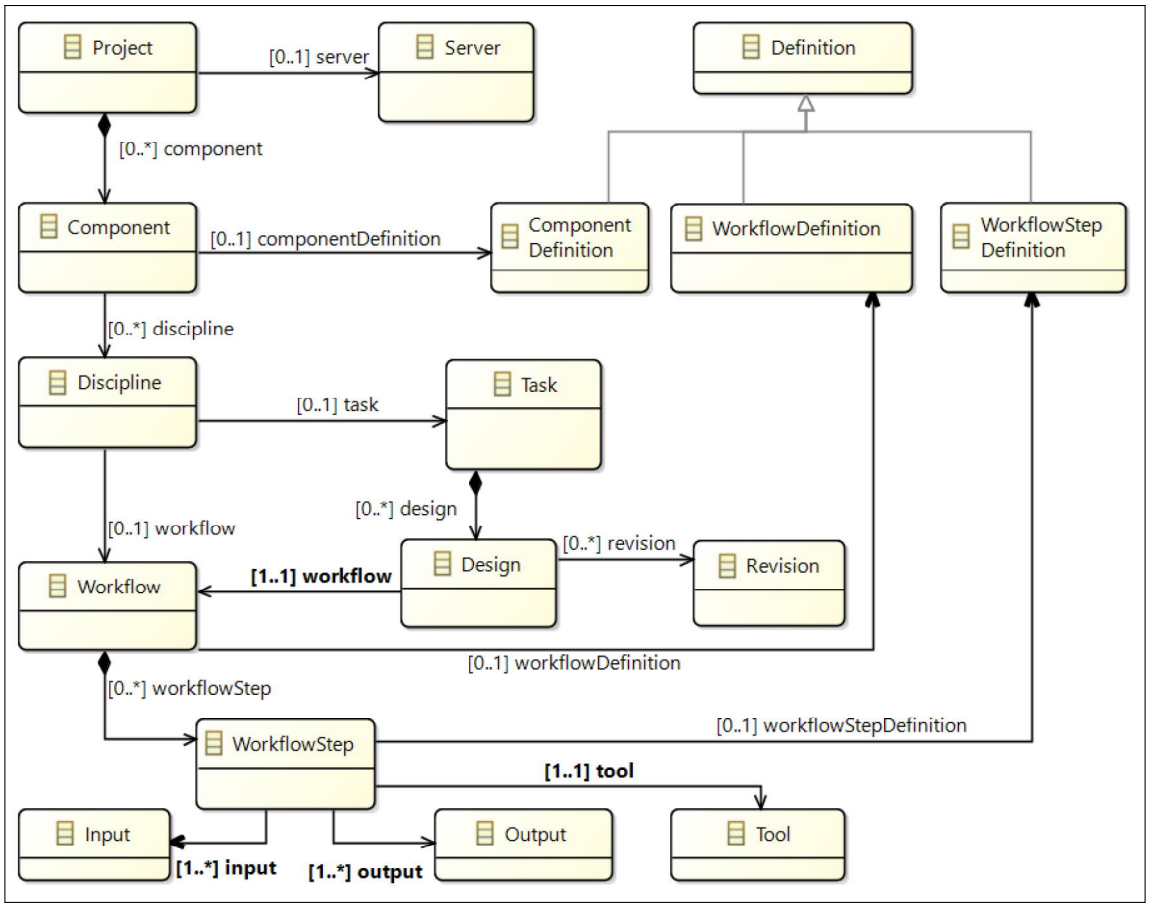


Figure 2.5: Domain model of the main concepts of Siemens tool integration framework

3. Assigns the workflow to a project which typically represents the gas turbine version that the discipline is working on.

4. Uploads the workflow to the workflow repository.

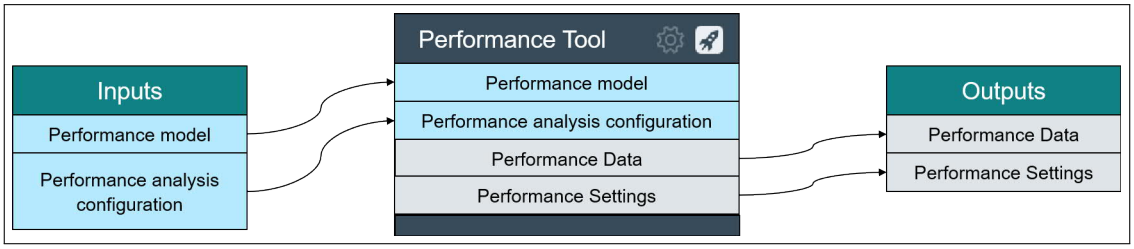


Figure 2.6: AGT workflow structure

Workflow Execution in Siemens Tool Integration Framework

In this stage, an engineer

1. Creates designs using the workflows.
2. Adds them to a personalized task list.
3. Executes these design workflows and examines the output produced.
4. When the satisfactory results are produced, he/she saves the outputs by creating a design revision. These design revisions can be retrieved anytime to review the results or shared with other users of the group for their review.

2.4.3 Architecture of Existing Tool Integration Framework

The architecture of the existing tool integration framework used at Siemens AGT is shown in Figure 2.7. The framework is a monolithic application in which the functional components are tightly coupled. The *workflow execution front-end* is responsible for rendering the workflow definitions on the UI and facilitate the execution of the workflows. The front-end component captures the user actions on the UI and calls methods of the *workflow execution logic* to perform corresponding actions. To execute the analysis tool the workflow execution logic communicates with the *analysis execution logic*. Workflows are stored in a *workflow repository* and are accessed by The workflow execution component or analysis execution components via *DB interface* objects. Similarly, the gas turbine design models are stored in a *model repository* and are accessed by analysis execution logic when necessary.

2.4.4 Discussion of Challenges

The current workflow-based tool integration framework used at Siemens AGT is stable and extensible. It provides Siemens engineers with a feature-rich environment to effectively define and execute workflows. However, Siemens engineers face many challenges

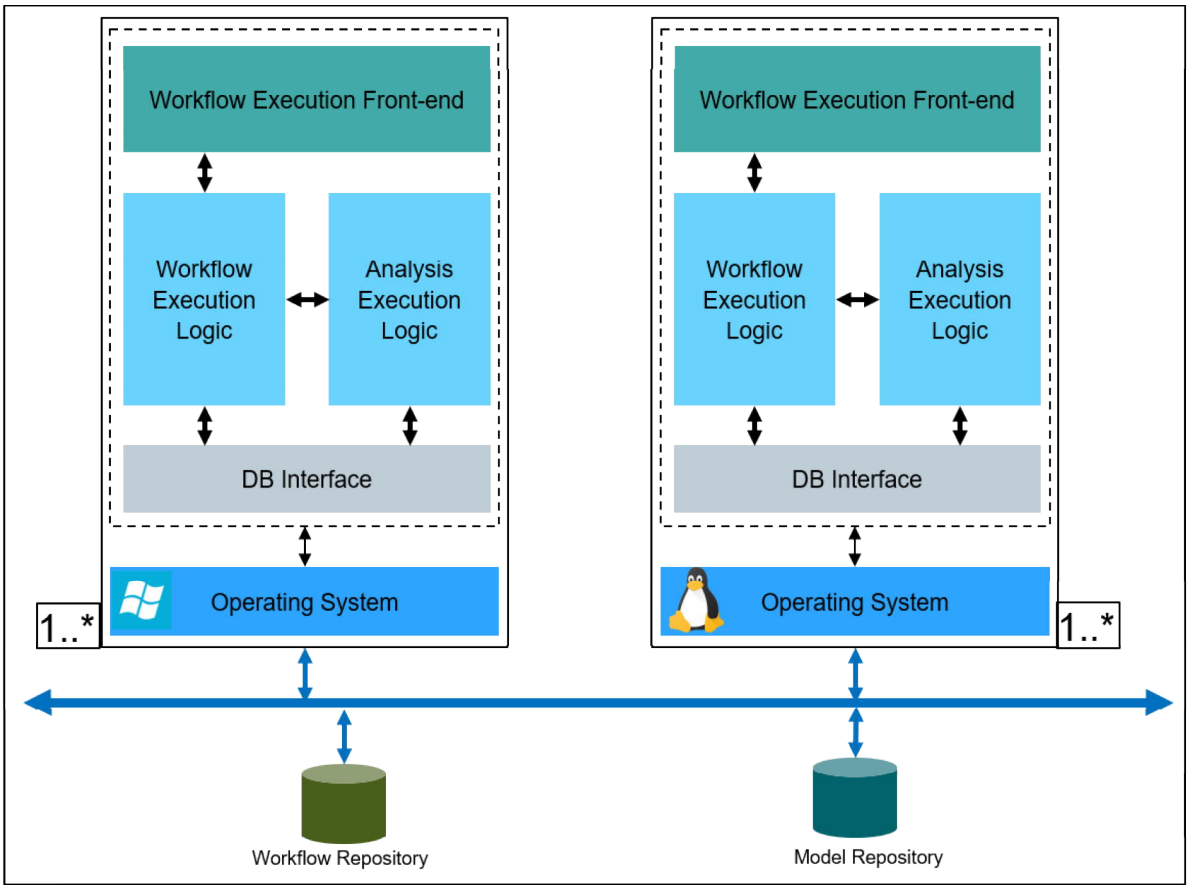


Figure 2.7: Architecture of the existing framework

while using the framework due to the limitations imposed by the architecture and the technologies used. The following list highlights a few of these existing limitations in the framework.

- **Local (desktop) execution:** Engineers run the workflow based tool integration framework and the analysis tools on their own laptop or a workstation. This localized execution results in limited performance. Performance is even more limited when the framework and tool are installed on the network location due to higher application load time. Varying network bandwidths and speed also affect the performance of the workflow execution.

  Results produced with local execution are accessible only on the engineer's laptop. The engineer has to manually upload or publish the results to a central database to

share the results with other engineers. This limits collaboration and may result in sharing of incorrect versions of results.

- **Insufficient compute power:** Many design and analysis tools used in AGT design are computationally intensive and require powerful computational resources. An engineer may have a standard office laptop computer or a high-end configuration tech-laptop which often limits the performance of such tools.

  Teams use workstations with high-end hardware configuration to execute computationally intensive design and analysis steps. These workstations are often dedicated to one type of analysis and are administered by one group which results in waste of computational resources.

- **No on-demand scalability:** The workflow execution frames allows running only instance of workflow or tool at anytime. Single execution instance is a limitation when an engineer needs to run a particular analysis for multiple engine configurations. In such cases, the engineer has to resort to sequential execution of multiple configuration which results in long waiting time for results and delays the overall workflow.

  If a parallel multiple configuration execution is necessary, an engineer must manually setup multiple runtime instances of the framework and run them simultaneously. Such a manual setup is time consuming, uncomfortable and error prone.

- **Platform dependency:** AGT has design and analysis tools are that are OS specific. The requirement to run tools in heterogeneous platforms is not supported in the current framework. The framework is platform-specific and supports execution of tools targeted for host platform. Because of the platform-specific nature of the framework, users need to switch between the Windows and Linux operating systems to run different tools. By doing this, engineers spend a lot of time performing non-value-added tasks.

- **Need for manual deployment:** A Siemens engineer or the IT support team manually installs the workflow based tool integration framework on the engineers laptop computers or on a network location securely accessible to the engineer. Design and analysis tools used in the workflow are also installed either on engineer's laptop computers or the network location. When a new version of the framework or tool is available, this manual deployment is repeated. This decentralized deployment is error prone and requires manual intervention. The deployment becomes even more challenging and error prone when tool integration framework has to be deployed at multiple geographical sites of Siemens.

2.5 Summary

This chapter provided a detailed overview of the background to understand the context in which this thesis is developed. It described the tool integration aspects that are applicable in the context of designing a complex cyber-physical systems such as an aero-derivative gas turbine at Siemens AGT. It also gave an overview of the existing tool integration practices and frameworks used at Siemens AGT. The chapter concluded by discussing the challenges associated with the existing tool integration framework.

Chapter 3

Background: Concepts and Technologies

Delivering AGT design workflow execution and analysis services (Referred to as **AGT services** from this point onwards) as cloud-based microservices requires the use of a wide range of concepts and technologies from the recent developments in the field of software engineering and cloud computing. As part of this thesis, we studied cloud computing service models used in delivering web applications.

3.1 Cloud Computing Service Models

The cloud industry has adapted three main service models namely Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). Figure 3.1 shows a comparison of these three service models.

- **Infrastructure as a Service (IaaS):** IaaS refers to the offering of computing resources such as processors, storage and networking functions as service. An IaaS provider sets up a large pool of computing hardware and offers it in portions to its users on demand and charges them based on usage. The consumer hosts applications which are typically web services on this hardware. Hosting applications on hardware provided by an IaaS provider gives many benefits to the consumer. Firstly, the consumer does not need to invest money to purchase and setup Information Technology (IT) infrastructure. Secondly, the consumer can add more resources easily to scale up their service in case they receive unexpected high demand for their application. Similarly, they can easily scale down when the demand goes down. Finally,

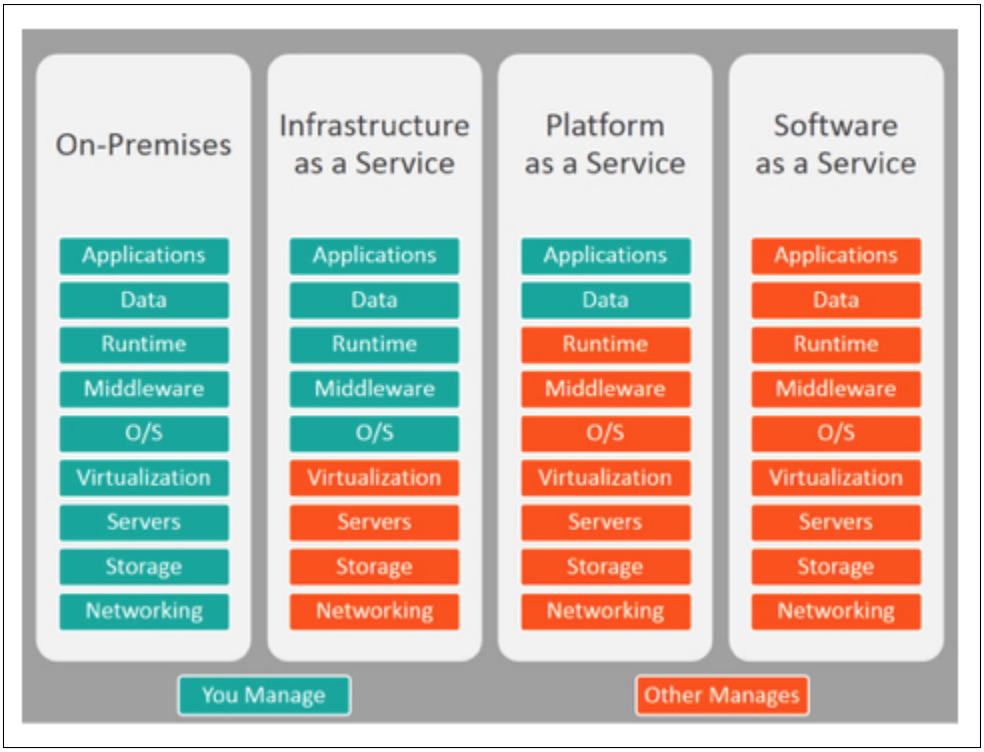


Figure 3.1: IaaS vs PaaS vs SaaS. Image Source [25]

they do not need to maintain the IT infrastructure as the IaaS provider is responsible for all the maintenance.

- **Platform as a Service (PaaS):** PaaS refers to the offering of components such as Operating System (OS), middleware and runtime libraries as service. The consumer uses these components to develop their applications which are offered to end-users as services.

- **Software as a Service (SaaS):** SaaS refers to the offering of software applications as services to the end-users. The service provider hosts the application on a private server or a public IaaS facility and provides Uniform Resource Locators (URLs) to the end-users to access applications using thin client applications such as a web browser. A major portion of the work carried out as part of this thesis is developing software applications to offer AGT design analysis tools as a service that exemplifies

SaaS. The following sections of this chapter give an overview of the concepts and technologies of SaaS that are most relevant to this thesis.

3.2 Software as a Service (SaaS)

The SaaS service model has revolutionized the way software is sold, delivered, maintained, and used. The software provider sells the licenses and gives access to the user to the subscribed software features that can be easily accessed using a thin client application over the internet. The software provider delivers the software to a centralized server and maintains it. The infrastructure and the platform on which the software runs is abstract to the user. This subsection gives an overview of the SaaS service model of cloud computing.

3.2.1 Introduction

SaaS is a capability provided to the consumer to use the service provider's application running on a cloud infrastructure [39]. SaaS symbolizes a transformation in the way in which software is sold and distributed by the application providers and how consumers use it. Historically, companies developing software applications sold licenses and shipped an installer on different physical media such as USB disk or CD-ROMs, or they provided access to a secure location from where consumers could download the software installer over the internet.

The SaaS significantly simplifies the software delivery and maintenance process [34]. The software provider deploys the software on a cloud infrastructure that is privately owned or rented from a third-party public cloud provider. In the case of private infrastructure, the software provider will manage the infrastructure. The public cloud infrastructures such as Amazon Web Services (AWS) [7], Microsoft Azure [32], Google Cloud [18] are provided by third party vendors in the market and the software provider pays to the cloud provider on a usage basis. The Consumer can access the application by using a standard thin client application such as a web browser or by using an application-

specific client developed by the service provider using a mobile device or a Personal Computer (PC).

The cloud infrastructure including network, servers, operating system, storage, and even the individual capabilities of the application is abstract to the consumer. The consumer does not know, manage or control these resources. In a few use cases, the consumer may be able to configure a few parameters of the application through application settings. Few well-known applications offered in the SaaS model are Microsoft Office 365, Google applications, and SalesForce.

3.2.2 Architecture

Figure 3.2 shows the architecture for SaaS which follows the traditional client-server architectural style. The software product or the application being offered - AGT Services in this case - runs on the server component. The *browser*, *mobile applications*, and *desktop applications* are typical client applications which communicate with the server hosting the SaaS application over a secure internet channel using a standard protocol such as HyperText Transfer Protocol Secure (HTTPS). Over the years, the industry has adopted many communication schemes over HTTPS for information exchange between the client and the server. The REpresentational State Transfer (REST) [14] Application Programming Interfaces (API) style has gained popularity in recent years and is described in detail in Section 3.3.3 of this chapter. The SaaS application uses many *Platform Services* which may either be provided by the cloud service provider as PaaS or open-source products such as Docker. These platform services run on cloud infrastructure which is either provided by a cloud service provider as IaaS or could be in-house infrastructure.

3.2.3 Benefits

SaaS applications enable the customer to access applications over the network. Features of the applications are available from a server that is centrally set up and managed by

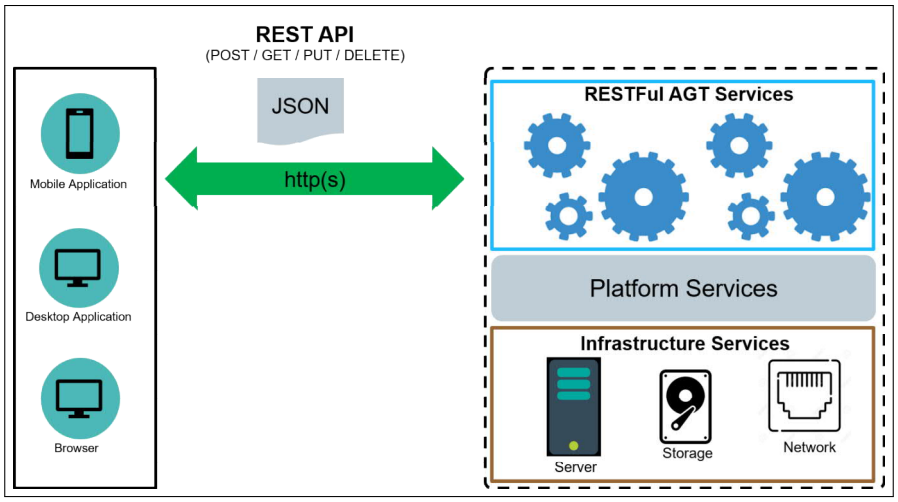


Figure 3.2: SaaS access and deployment architecture

developers. As the application setup is centralized, all updates are immediately available to the customer without needing to download or install any patches. These features of SaaS provide many benefits few of which are most relevant to this thesis are listed below.

1. **Easy to incorporate:** The customer can be up and running instantly as SaaS application is delivered via the internet and does not require any installation or setup at the customer end.

2. **Easy to manage:** SaaS application is hosted on a server hence it is centrally managed by the development team with the help of Continuous Integration and Continuous Deployment (CI/CD) process, which helps in the seamless testing and delivery of application to the servers.

3. **Increased accessibility:** The ability to access the service by using simple clients increases user accessibility. SaaS application model saves data on the securely accessible servers. Making it available for collaborative access from inside or outside the organization through secure channels. This eliminates geographical limitations to access application features. Geographically distributed teams will function better with collaboration.

4. **Increased productivity:** A user can run multiple instances of computation-intensive applications and obtain results faster. When combined with increased accessibility, the SaaS model results in a considerable increase in productivity.

5. **Platform independence:** The SaaS model gives platform independence at both client side and server side. On the client side, an SaaS application that is hosted centrally on a server can be accessed by the client running on any platform (Windows, Linux, Mac). On the sever side, the SaaS service can be hosted on heterogenous platforms.

6. **Reduced client-side resources:** Since the application is centrally hosted, all high computation tasks are run on the server. The client-side require only basic resource to run a browser or a client application.

7. **Improved scalability** SaaS applications are scalable on demand by creating more instances of the application when the demand increases and vice versa.

3.3 Microservices

In order to design and deliver a tool or application in the SaaS model, we need to embrace a reasonable software architecture. This section briefly discusses different software architectures used in developing server-side applications and then gives an overview of microservices architecture as this architecture style is most relevant in the context of this thesis.

3.3.1 Introduction

For a long time, server-side software applications have been dominantly using a monolithic software architecture. A monolithic application embeds all business logic in a single executable file and many Dynamic Link Libraries (DLLs). It is easy to deploy and it can be easily scaled up by running replicas of the application on multiple servers behind a

load balancer. A monolithic application like Enterprise Java Beans is highly coupled and the complexity increases as the application gets bigger.

Service Oriented Architecture (SOA) [13] is another architecture style used in the server-side software engineering community. SOA divides the application into a few coarse-grained services that communicate over the Enterprise Service Bus (ESB) [4]. Each service may still resembles a monolithic application in nature and has the same complexity issues of a monolithic application.

Since early 2011, a new architectural style called RESTful Microservices has emerged in the area of server-side application development and has seen high adaptation [68]. This architecture style is inspired by Service-Oriented Computing (SOC) [19] in which the server-side application is divided into programs called services that offer functionalities to other SOC components via message communication. The following subsections describe this architectural style in more detail.

3.3.2 Architecture

Microservices architecture is widely used in developing cloud-based server-side applications. A microservice is a small, independent application running in its own process and communicating with lightweight mechanisms [15]. Every microservice has a single responsibility and can be independently tested and deployed to provide the capability of software product [63] [48] [56].

RESTful microservices architecture is an adaptation of the REST architectural style for web services initially proposed by Fielding, R. T [14] and microservices architecture discussed above. REST is a set of guiding principles that are specified to have simplicity, scalability and high performance in applications such as web services. A REST-compliant (RESTful) microservice must have client-server architecture and use a stateless communication protocol such as HTTP(S).

An example microservices architecture for web services is shown in Figure 3.3. The *Microservices* are hosted on the distributed cloud servers. Each microservice has a single

responsibility and is accessible through RESTful APIs. When an API request is received from the client, the microservice executes the corresponding business method and sends the response in the predetermined format.

The mobile and desktop applications *clients* that are capable of calling RESTful APIs access the features offered by microservices through an *API gateway*. The API gateway takes one request from the client and constructs multiple requests and sends them to target microservices. It then constructs a consolidated response from individual responses received from different microservices and sends it to the requesting client. The API gateway finds the target microservices through the *service discovery*. The *service orchestration* is responsible for managing the service availability and scalability.

A thin client application like a web browser cannot interact with the microservices through RESTful APIs. The browser requests a webpage from a *web UI service* which is running along with other microservices. The UI that is loaded with the webpage is capable of sending RESTful API requests to access the feature offered by other microservices.

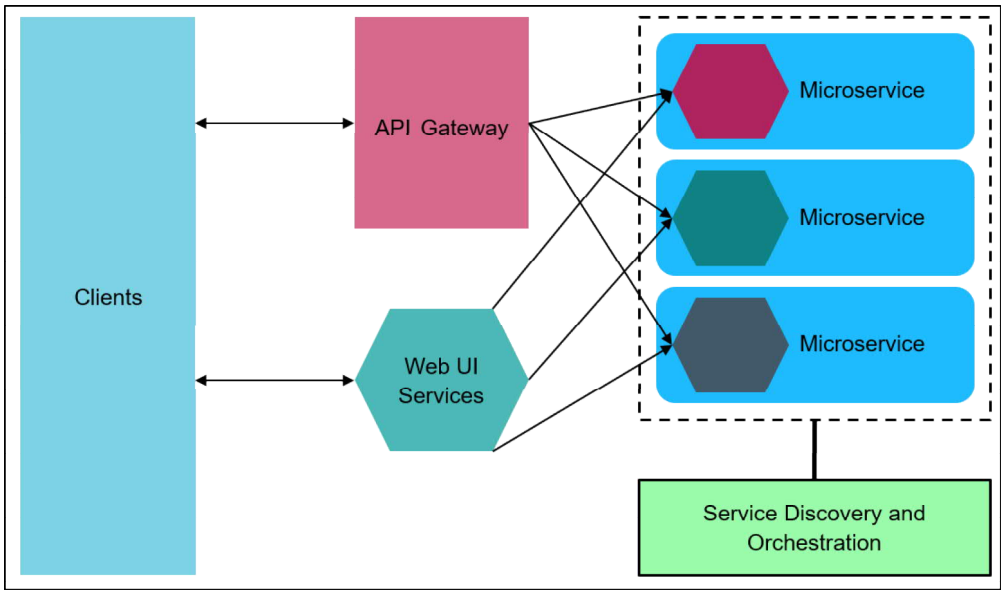


Figure 3.3: RESTful Microservices and their interactions

3.3.3 REST APIs

The fundamental concept behind REpresentational State Transfer (REST) architecture is that the web application has resources whose runtime (REpresentational) state is exchanged between the client and server. The REST architecture states that web applications achieve this by implementing uniform interfaces called REST APIs[14].

Uniform Interfaces

In the typical client-server interaction of web services, the client sends a request to perform some action on a system resource. One of the constraints imposed by the REST architecture guidelines is that the web service shall allow different clients to use the same APIs to operate on resources. This constraint has invariably lead the developers for web services to use Create, Read, Update and Delete (CRUD) operations that are typically found in the database systems. This drive is also because these operations can be easily mapped to GET, PUT, POST and DELETE methods of HTTP(S) protocol embedding the REST APIs.

REST architecture has laid down the following constraints for implementation the of uniform interfaces.

Resource Identification through URI

The resource is the key information abstract in REST. A resource is analogous to an object instance in the object-oriented paradigm. It has a type, associated data, relationships with other resources and a set of methods to operate on. Any aspect of the application domain that can be named is a resource (e.g. person, document etc.). REST mandates that these resources shall be identified through a Unique Resource Identifier (URI) which consists of *base URL* and a *resource* part. An optional *version* information is often used to identify the API version. For example, an URL for getting the information about a *project* with id *12345* using the v1 version of APIS is shown in Figure 3.4

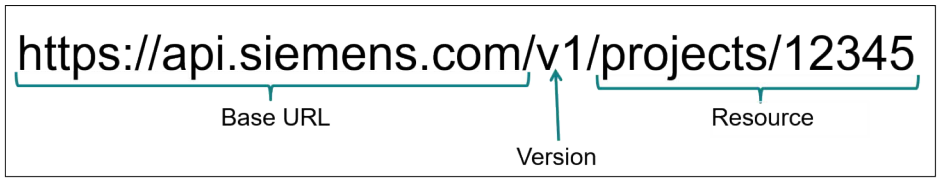


Figure 3.4: REST URI example

Manipulation of resources through representations

The server returns the state of the identified resource in different formats such as HTML, XML, JSON etc. These formats are called resource representations. The client updates or creates a new resource by sending these resource representations. The server may support multiple resource representations and the client can request a particular resource representation by indicating the representation type in the Content-Type header field of the HTTP protocol.

Further the REST architecture constraints state that every message exchanged between the client and server must be independent of the previous or future message and the client shall be able to figure out all possible operations that can be performed on the resource by parsing the response.

3.3.4 Benefits

Microservices designed to meet the REST interface design constraints naturally have the following benefits.

- **Simplified Design:** An Important requirement of RESTful microservices is that all services are stateless. The server does not keep track of the client's state. The client transfers its context whenever necessary to the server and the server provides requested information based on this context. This allows the server-side application to treat each request independently and simplifies the server-side application design.

- **CI/CD Support:** Microservices fit well in the DevOps [40] process as it is easy to deploy a small, independent, and stateless microservice on cloud infrastructure.
- **Easy Scaling:** It is also easy to scale up or scale down. A microservice is scaled up simply by starting a new instance of the service on the server and by registering it with a load-balancer or an API gateway.
- **Decentralized Database:** Dividing the application into single responsibility microservices leads to the separation of data. Data corresponding to one business functionality is stored in one database and is attached to the required service.
- **Distributed Application:** Small, independent services can be hosted on any networked computing resource. This invariably leads to a distributed system and eliminates the risk of single-point failure.
- **Technology Independence:** Each microservice is independent and communicates with other microservices through standard channels. This gives an incredible amount of independence to developers in choosing the technology stack for the service. There is no need to adopt a technology or database because other services are using it.

Open API Specification

As the popularity of REST APIs in the web services community increased, efforts are made to standardize the API design. Since 2016, an open-source community called OpenAPI Initiative [45] has started releasing specifications referenced as OpenAPI Specification (OAS) [44] to help the community design APIs with the following intentions.

- Facilitate designing of programming language-agnostic APIs so that both users and machines comprehend the abilities of APIs without requesting access to source code or extra documentation.

- Promote a design-first approach to invest enough energy on designing APIs instead of diving into usage.
- Use a standard way of recording and communicating APIs by using YAML or JSON to write API specifications.
- Facilitate code generation, test generation and test automation through the usage of standard and uniform specification formats YAML and JSON.

3.4 RESTful Microservices Development Frameworks

3.4.1 Flask-RESTful Web Service Framework

Various frameworks such as Node.js[61] in Java Script, Flask[51] in Python, Spring in Java are used for development of RESTful microservices. For developing AGT services we chose Flask-RESTful framework primarily because it native level integration with the Siemens tool integration framework, which is also written in Python programming language. Using a web service development framework in Python programming language is necessary to reuse the modules of existing tool integration framework in the development of the workflow execution service.

The Flask [31] is one of the widely used framework for building web services in Python programming language and is designed to make getting started easy with ability to scale up do design complex web applications. The Flask-RESTful is a lightweight wrapper above the Flask framework which makes designing the RESTful web services with Flask more easy.

3.4.2 Task Queue Framework

In synchronous request processing web services the server waits until the requested task is complete before sending the reply to the client and is blocking on both client and server

sides. For long running tasks such as AGT analyses, this will result in non-responsive, high-latency workflow execution. A responsive low-latency web service would need to reply to the clients immediately and run the long tasks in the background and be ready to receive new requests. This can be achieved by queuing the incoming requests with the help of a task queue and then executing the tasks in the background in the order they arrived.

We studied many task queue frameworks and found that the Celery [17] framework is the most suitable for distributed microservices as the task queue itself has a distributed architecture and is easy to set up. The Celery task queue framework shown in the Figure 3.5 allows applications to post tasks in a *Task Queue* maintained in a *Message Broker*. Then the celery task manager distributes the tasks one-by-one to one of the registered *Workers* to execute the task. When the execution is complete the task results are stored in the *Message Backend* and can be retrieved by the application on demand.

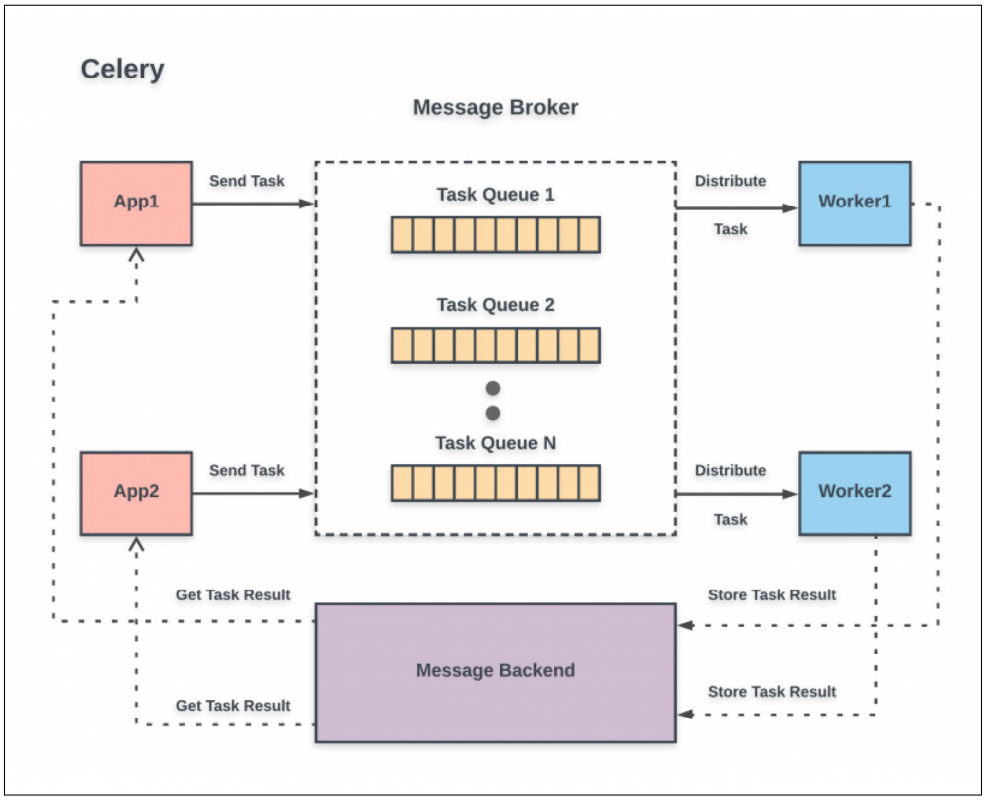


Figure 3.5: Celery task queue [69]

The message broker required for operating the Celery task queue can be set up using frameworks such as RabbitMQ [47] and Redis MQ [33]. We selected Redis MQ as it can also be used as the result backend of the Celery task management framework.

Because of all the benefits listed above, interfaces designed under REST constraints and OAS guidelines are the most relevant for AGT services.

3.5 Docker Containers

Docker containers and Docker ecosystem is the technology used for the parallel distributed execution of analysis in this thesis. This section gives an overview of this technology and its foundational concepts in the context of this thesis.

3.5.1 Containers

Since the beginning, virtualization has been the backbone of the cloud computing. virtualization is the technology that makes the cloud features such as elastic computing and isolation to provide privacy and security in cloud computing possible. Many levels of virtualization exists. In the context of this thesis, OS-level virtualization which has given raise to the virtualization technology called containers is the most relevant. Following sections of this chapter gives an overview of this technology.

In OS-level virtualization the kernel allows multiple isolated user-space instances referred to as containers to run simultaneously. In contrast to equally popular hypervisor based virtualization, containers provide

- **Extremely fast instance creation times:** The daemon application managing the containers can create container instances faster compared to an hypervisor. Containers boot fast and become operational instantly.

- **Small per-instance memory footprint:** Containers are very economical on system resources as lightweight containers take less memory on disk and RAM

- **High instance density on single host:** More instances of containers can be run simultaneously on a host due to their small memory footprint

The industry has seen evolution of many Containerization technologies such as Docker [46], Rkt [41], Linux Containers and Unikernels. In this work, we focus on Docker containers, as they are employed in production environments and are supported by popular cloud platforms such as AWS, Azure and Google and their container orchestration systems

3.5.2 Docker Ecosystem

Figure 3.6 shows components and their interactions of the Docker ecosystem. The Docker container technology is built-in client-server architecture and consists of a Docker *Client*, a *Docker Host*, and a *Docker Registry*. The Docker host is the machine on which the *Docker daemon* and *containers* run. The daemon is the central part of the Docker architecture and is responsible for building Docker *images* and running Docker containers. The Docker client acts as the user interface to the Docker daemon and allows the user to execute commands on the Docker daemon. The Docker daemon works with the Docker registry service which stores and distributes images.

A Docker image is a template that the Docker daemon uses to create a Docker container which is the runnable instance in the Docker ecosystem. Docker has an easy and user-friendly image description language that integrates well with the Docker image build system. Once the image is built, it can be stored in a version-controlled public or private Docker registry. Images can then be downloaded by other hosts to create and run containers or to use as a basis for building new images.

3.6 Docker Swarm Cluster Manager

Creating stable and scalable AGT services involves starting many containers of Docker images built for these services on a distributed cloud infrastructure. This involves managing clusters of computing nodes and orchestration of services on the cluster. This sec-

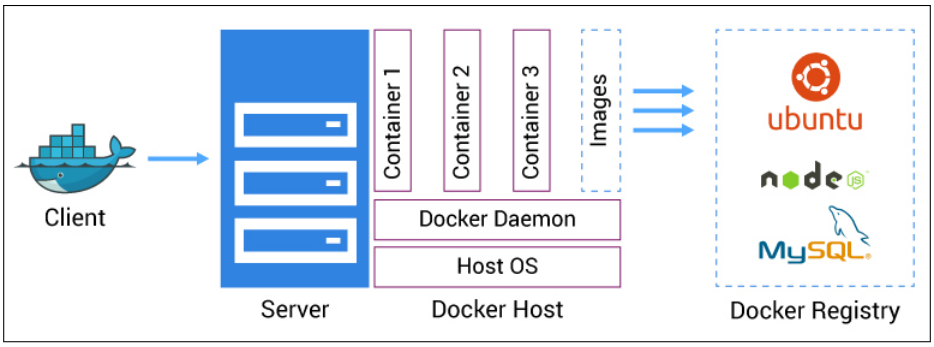


Figure 3.6: Docker container ecosystem. Image Source [8]

tion gives an overview of the cluster management and service orchestration tools used in this work.

3.6.1 Cluster Management and Service Orchestration

In the context of cloud computing, a cluster is a group of nodes coming together to form a computing infrastructure for the application running on the cloud. If the cloud is viewed as a number of connected servers, by clustering these servers, hundreds of CPU cores, thousands of gigabytes of RAM, and hundreds of terabytes of storage becomes available to the run one application. A cluster will typically have few manager nodes and lots of worker nodes. A cluster dynamics keep changing as manager and worker nodes may join and leave the cluster anytime. A stable and powerful management service is necessary to create and manage such a cluster on distributed cloud environments.

To support more users and parallel execution of workflow service or design and analysis for AGT, we need to run many instances of the Docker containers serving the application and distribute the incoming jobs using a load-balancer. We will also need to increase or reduce the number of instances based on demand dynamics. In cloud computing, this type of service creation and scaling is referred to as service orchestration.

To serve such a need, container providers have developed integrated cluster management and orchestration services. Many such services exist. Docker swarm and Kubernetes are two popular open-source cluster management and orchestration services. In

this work, we use Docker Swarm as it comes integrated with the Docker Engine that we have already chosen as container technology for this project.

3.6.2 Docker Swarm

Figure 3.7 shows the architecture of the Docker Swarm. A swarm [60] is a cluster of Docker Engines consisting of one or more nodes. Swarm nodes are classified as a *manager* node or a *worker* node. The manager node is responsible for the cluster management activities such as managing the cluster state, scheduling services and serving API requests. It is recommended to have more than one manager in the swarm to have continued swarm operation in-case a manager fails. Manager nodes save the cluster state on a distributed *state store* and use the persistent state information to recover from failures.

A manager node performs the deployment of the service as well as manages and orchestrates the cluster such that the desired state of the service is maintained. When a service is created, specified instances of the Docker containers are started on the cluster nodes. The manager node dispatches the tasks received from the external clients based on a predefined scheme to the *worker* nodes. Worker nodes execute the given tasks and return the status of the assigned tasks. This feedback allows the efficient scheduling of the jobs to available cluster resources.

In summary, Docker swarm cluster management and service orchestration give all the necessary functions required to create and scale-out service across a dynamic cluster and meets the requirements of this work.

3.7 Continuous Integration and Delivery

Today, more and more software companies are adopting agile methodologies where they aim to deliver new features incrementally in short development cycles [36] [52]. To meet such demanding needs industry has adopted practices such as continuous integration and continuous delivery. Tool and technologies are now available to help organizations

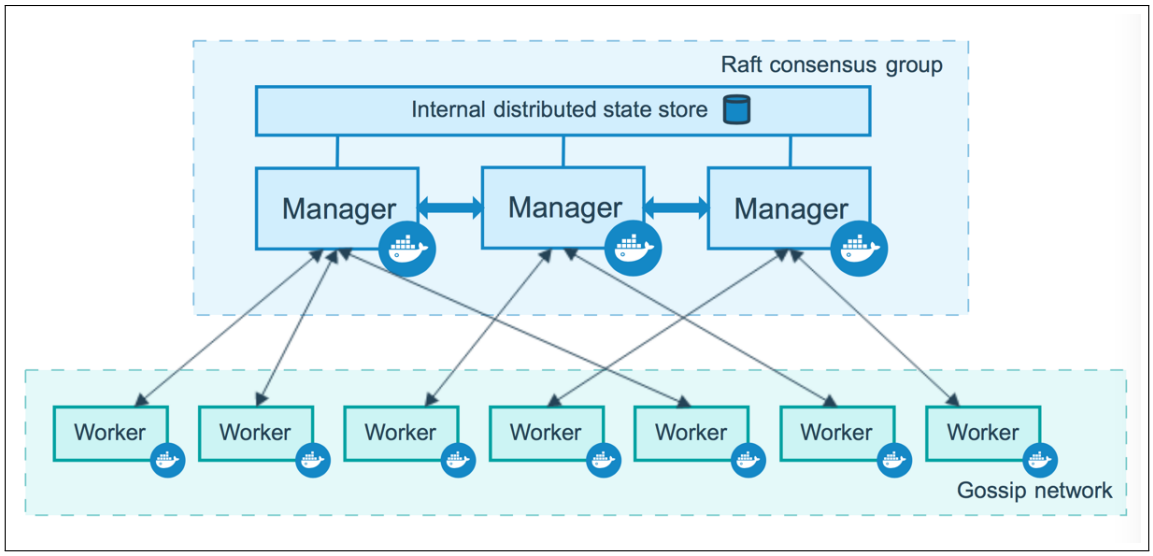


Figure 3.7: Docker swarm architecture. Image Source [9]

in this effort. This section will give an overview of these concepts in the context of our work.

Continuous Integration (CI) is a development practice in which software is integrated continuously [11] during development by automating the build process. When a commit is made to the designated stream of the code repository, the software is automatically built and regression tests are run to verify that the new changes have not broken the software.

Through Continuous Delivery (CD) [6] the software built and verified in the CI stage is delivered to production branch. A release engineer can then release the software to the target group. An automated CD procedure ensures that the latest version of the software is always available for release to the target group. Continuous deployment adaptation of CD takes this process one step further and deploys the software to the locations from where the target users can directly access the new version of the software.

CI/CD in the Context of AGT Services: Many automated build and deployment systems are available to incorporate the CI/CD procedures in any project. The most important ones are Jenkins [27], Travis CI [38] and GitLab [20]. In our work, we use GitLab CI/CD system as GitLab is the repository used at Siemens AGT for software development

projects. Irrespective of the CI/CD system used, resulting CI/CD workflow for software services developed using the Docker ecosystem is as shown in Figure 3.8.

Developers pull the base images needed to build the Docker image containing their application from an image *registry*. The image registry can be private or public. We propose setting up a private image registry on AGT premises and public image registries such as *Docker hub* are not suitable for storing export controlled images of this project. *Developers* push their updated image files to the version control system such as GitLab which has an integrated CI/CD system. The GitLab CI/CD system performs the integration stage and on successful integration, it delivers the image to a pre-configured image registry. Finally, an automated or manual release to the target *RUN* environment makes the updates available to the target users.

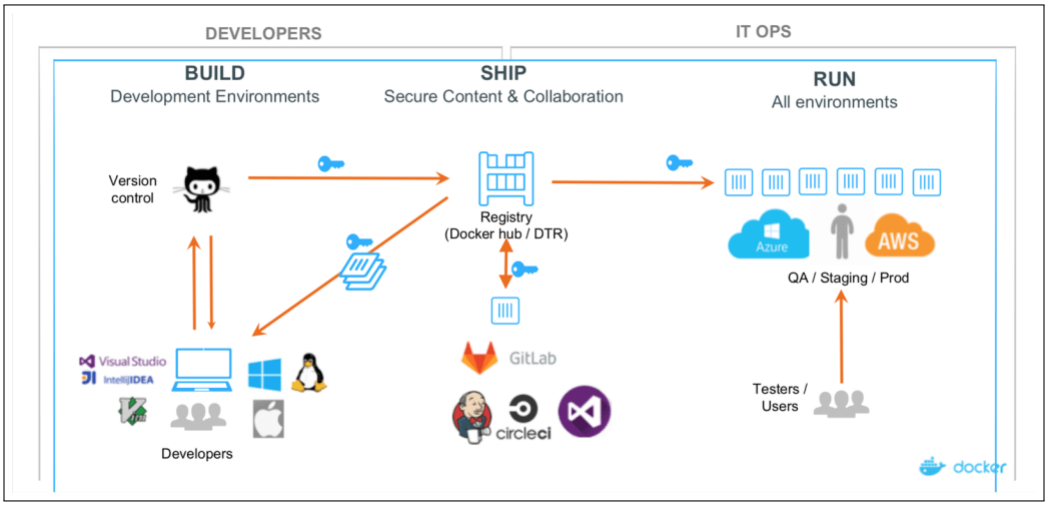


Figure 3.8: CI/CD with Docker ecosystem. Image Source [26]

3.8 Summary

By incorporating a web service like architecture for workflow execution and tools execution, benefits of the SaaS model can be reaped by AGT design workflow that requires running iterative analyses. Adapting such an architecture requires the use of many concepts and technologies from the cloud computing domain. This chapter provided an

overview of these concepts and technologies and their applicability in the context of AGT workflows.

Chapter 4

Service Architecture for Tool Integration Framework

Workflow-based tool integration framework used at Siemens AGT facilitate mapping of AGT design process steps to chaining of tools to execute dependent analysis steps efficiently. Performance bottlenecks are encountered when an engineer need to execute an analysis step having hundreds of iterations - which is typically the case in gas turbine design workflow. Availability of workflow-based tool integration framework and analysis tools as cloud-based web services accessible via REST APIs offer flexibility in both deployment and accessibility. A web service with multiple service instances is able to process multiple simultaneous requests and removes the performance bottlenecks of the single instance tool execution [12]. This chapter gives an overview of the service architecture proposed in this thesis for the tool integration framework and then gives a high-level overview of the software architecture of the prototype developed in this thesis.

4.1 Service Architecture

The fundamental idea of the proposed architecture is to expose the workflow-based tool integration framework and the analysis tools as microservices communicating via RESTful APIs. These microservices can be hosted on any public, private or hybrid cloud infrastructure in consistent with the business and export control requirements of Siemens AGT.

The functional architecture proposed for AGT workflow-based tool integration framework and analysis tools is shown in Figure 4.1. With the proposed architecture a Siemens engineer can first access the *workflow execution (WE) front-end* using a web browser and successively access the workflow execution services through the front-end application. The front-end acts as a User Interface (UI) and allows the user to perform workflow activities such as a) view revisions of workflows b) start execution of a workflow step c) view the current status of the workflow step. The front-end captures these UI actions and sends them to the WEaaS backend as RESTful end-point requests. The backend executes the corresponding service method and returns the results. the front-end presents these results to the user on an interactive UI that emphasizes User Experience (UX).

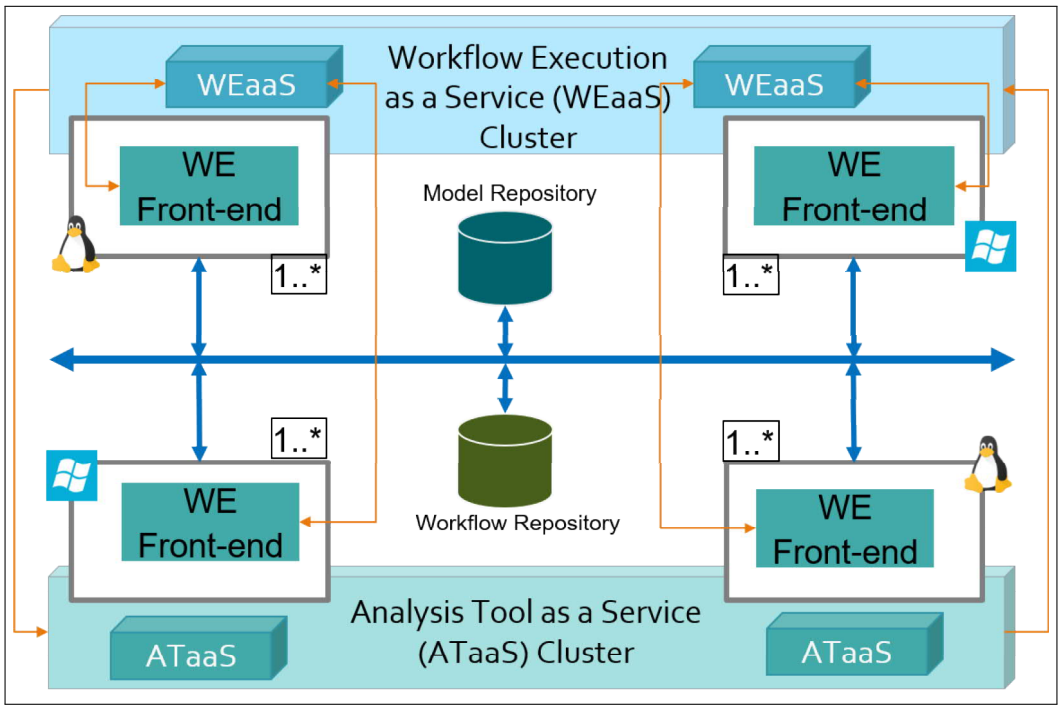


Figure 4.1: Service Architecture for WEaaS and ATaaS at Siemens AGT

4.1.1 Architectural Components

The proposed architecture has the following components:

- **Workflow execution front-end (WE Front-end):** Refers to the client application such as a web browser or a mobile application that is used to access the WEaaS features. The workflow execution front-end is platform-independent and many instances of the client application can be run simultaneously.

- **Workflow Execution as a Service (WEaaS):** Refers to the RESTful microservice that provides the workflow execution service of the tool integration framework. This microservice runs in a Docker container and it is hosted on WEaaS cluster.

- **Analysis Tool as a Service (ATaaS):** Refers to various microservices that provide AGT design analysis tool functionalities as services accessible via RESTful APIs. Like WEaaS, these services also run in containers and they are hosted on a ATaaS cluster.

- **WEaaS and ATaaS interactions:** An AGT analysis workflow consists of running the tool-chains. The WE front-end requests the WEaaS to execute a particular tool for a given configuration. The WEaaS identifies the ATaaS providing this tool service and requests it to run the specified configuration.

- **Service clusters:** A service cluster consists of connected computing resources to create a flexible execution platform for WEaaS and ATaaS services. Computing resources in the cluster are called nodes. A node can be a manager or worker. A cluster will have few manager nodes and many worker nodes. A cluster can be formed on a public, private or hybrid cloud.

- **Integration of repositories:** The proposed service architecture uses two types of repositories. A model repository for AGT engine models used in design and analysis steps. An ATaaS reads and saves models in this repository. A runtime resource repository is used by ATaaS for resources other than models such as results and intermediate artifacts.

WEaaS and ATaaS involves designing microservices by using the concepts and technologies described in Chapter 3. These services are hosted on service clusters created on demand using the Docker swarm. These key architectural components of the proposed service architecture are discussed in more detail in the following sections.

4.1.2 Service Cluster

We propose to host the WEaaS and ATaaS on a service cluster. The service cluster provides all necessary computing resources and middleware frameworks necessary to host WEaaS and ATaaS. A service cluster is created using the Docker swarm engine and it has many *nodes* (Figure 4.3). Any networked computing instance having the Docker engine and Docker swarm engine can be a node in the cluster. A swarm cluster has few manager nodes and many worker nodes. This section describes types of nodes used in hosting WEaaS and ATaaS services and discusses cluster creation and failure handling.

Cluster Nodes

Three types of nodes can be identified in the context of the Siemens project based on the location of nodes.

1. **On-premise nodes:** These nodes are physically located in the Siemens AGT office. In this thesis we used a mixture of laptops of Siemens engineers and few workstation computers as nodes in the service cluster. These nodes run Microsoft Windows operating system and are used to host the analysis tools that run only on Windows operating system. However, these nodes could be used to run linux containers as well.

2. **Remote nodes:** These nodes are physically located in other geographical sites of Siemens and are part of the private computing grids owned by Siemens. These nodes run Linux operating system and are used to run tools that run only on Linux operating system.

3. **AWS nodes:** These nodes are located on the AWS [7] cloud, which is a public cloud operated by Amazon. AWS allows creation of both Windows nodes and linux nodes and hence can be used to run any tool.

4.1.3 Workflow Execution as a Service

The WEaaS has microservices as shown in Figure 4.2.

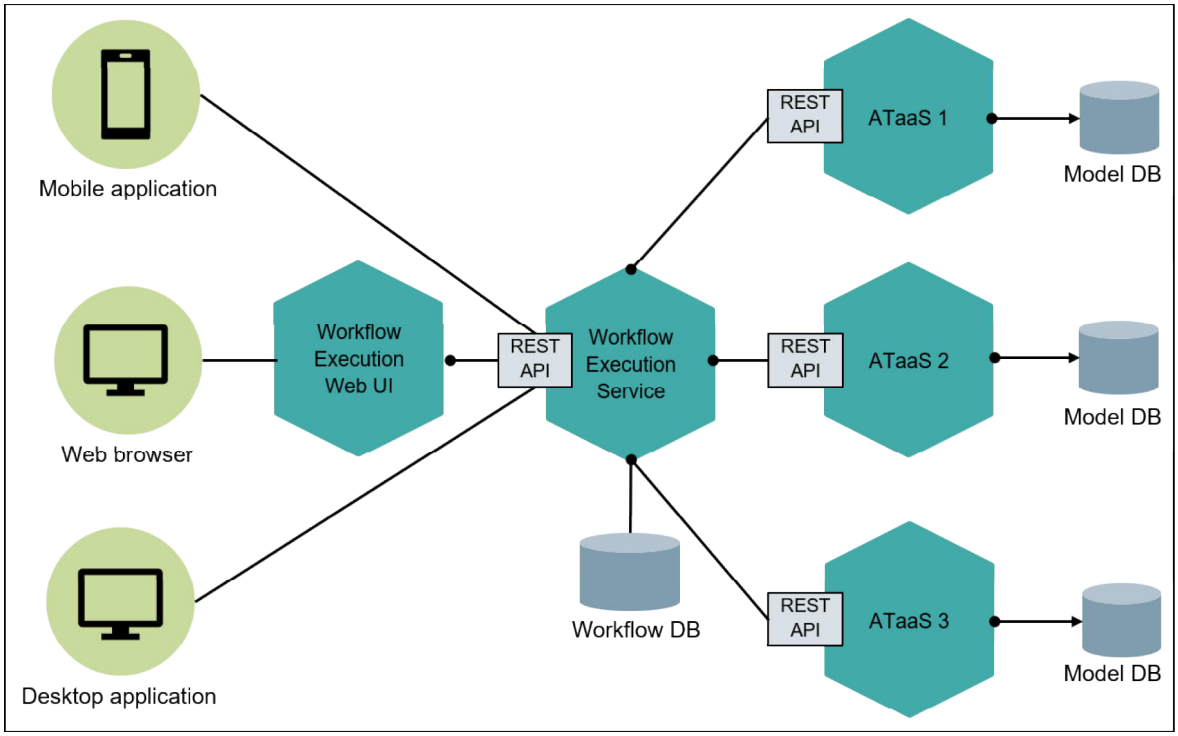


Figure 4.2: Architecture for workflow execution as a service (WEaaS)

Workflow Execution Web UI Service

The *workflow execution web UI* service is responsible for serving the web pages to the browser clients. An engineer can connect to the workflow execution service home page using a given HTTP(S) URL via a web browser. On the workflow execution home page, the user has provisions to

1. Connect to the workflow server and load the project assigned to the user.

2. Explore and create new designs by using the workflows assigned to the project
3. Create a personalized tasks list containing one or many designs.
4. Start execution of a workflow step.
5. View the status of the previously started workflow or currently open workflow.
6. View the results of completed workflows.
7. Download the results for reviewing on personal computer.
8. Upload the workflow revisions and results to the workflow repository

To provide above features, the web browser client accesses the *workflow execution service* using REST APIs. Alternatively, an engineer can access all of the above features by directly accessing the workflow execution service via a desktop application that is specifically designed for this purpose.

Workflow Execution Service

The *workflow execution service* is the microservice in which core business logic of workflow execution state machine is implemented. It has a RESTful interface. Clients can accesses services offered by workflow execution service by sending valid RESTful API requests. On receiving a valid request, the service runs an appropriate service method and returns the response.

4.1.4 Analysis Tool as a Service

On the workflow execution front-end, a user initiates execution of a workflow step. Execution of a workflow step involves running an analysis tool to produce the desired outputs. In the proposed architecture these tool invocations are implemented as RESTful API calls to microservices that provide analysis tool services.

Each tool service (ATaaS) is hosted on a *Service Cluster* (Section 4.1.2) as a microservice on a Docker container as shown in the Figure 4.3. Many instances of ATaaS are up and running at any given point in time and the *Service Provisioning* component of the architecture is responsible for the availability of the ATaaS in the desired configuration (number of instances). ATaaS instances execute requested analysis tasks and use a *Service Metadata* store to keep track of task executions and results. The *Execution Manager* component on the client side is responsible for accessing the tool service via REST APIs to execute many analysis tasks simultaneously by using gas turbine models located in a *Model Repository*. The *Load Balancer (LB)* directs the incoming request from *Clients* to one of the many active ATaaS instances. It also routes the response from the ATaaS back to the client requesting it. A *LB Config Management* module creates and dynamically updates the configuration file using which the load balancer distributes the incoming traffic to ATaaS instances.

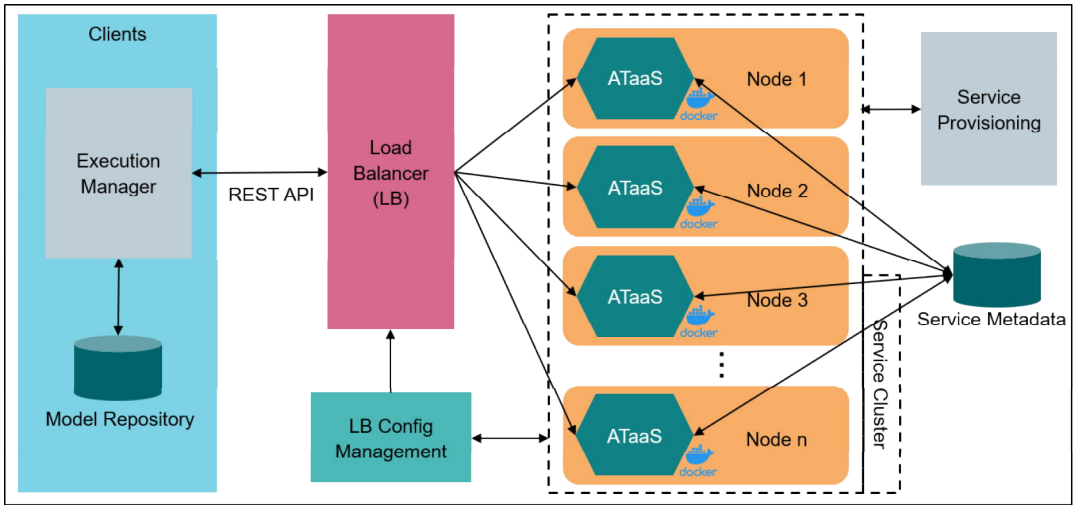


Figure 4.3: ATaaS architecture

4.2 Concepts of Tool Services

Resources are fundamental abstractions in RESTful microservices. Therefore, identifying these resources for the tool that need to be offered as a service is the first step in the process of creating a service offering for an analysis tool. These resources are also the

fundamental concepts of the tool services. Identification of the resources needs a good understanding of the tool use cases and generalization of concepts. We identified that in general, an engineer runs an analysis by executing a tool on specific inputs files to produce output files. Another general requirement is to be able to run multiple analysis tasks simultaneously. Accordingly, we identified tool service concepts as shown in the UML class diagram of Figure 4.4.

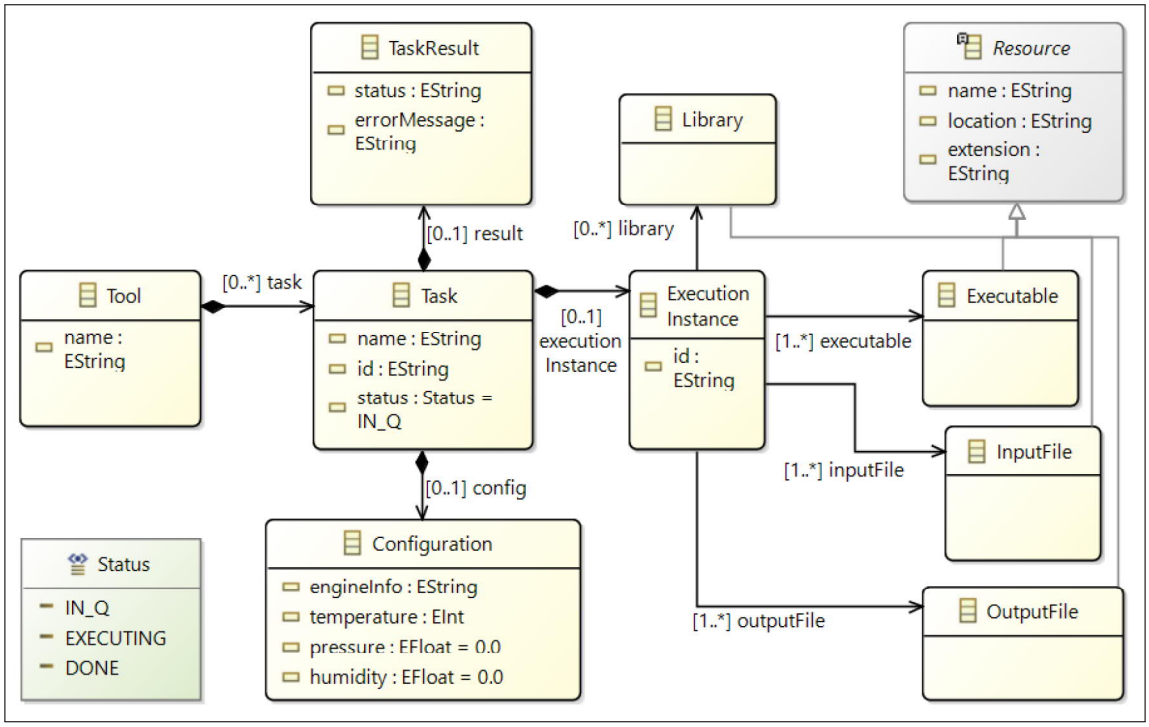


Figure 4.4: Concepts of tool services

A tool service hosts an analysis tool *Tool* and can be referenced using its *name*. A user can create, view and execute analysis *Tasks* by using this tool. When the user creates a task, the service puts the tasks in a task queue and returns an unique task *id* which can be used by the clients to start the execution of that task or view its status. A task may have a *Configuration* that is used during the execution of the analysis tool. When a task is started an *Execution Instance* which keeps track of the background execution of the task is created. The worker processes running on the server pick up the tasks from the queue and runs them. Execution of an analysis task involves running the tool *Executable* by

using dependent *Libraries* to produce *OutputFiles* from *InputFiles*. When the execution is complete, the metadata corresponding to the *TaskResult* is stored in the result database and the client can download the results by using the id of the task.

4.3 Software Architecture of the ATaaS Prototype

Figure 4.5 shows the software architecture used in thesis to implement a prototype of proposed service architecture. The architecture is more focused on the analysis tools services as the development of analysis tools services has been the prime focus of this thesis and omits the architectural details of WEaaS. The *Tool Service (ATaaS)* is implemented using

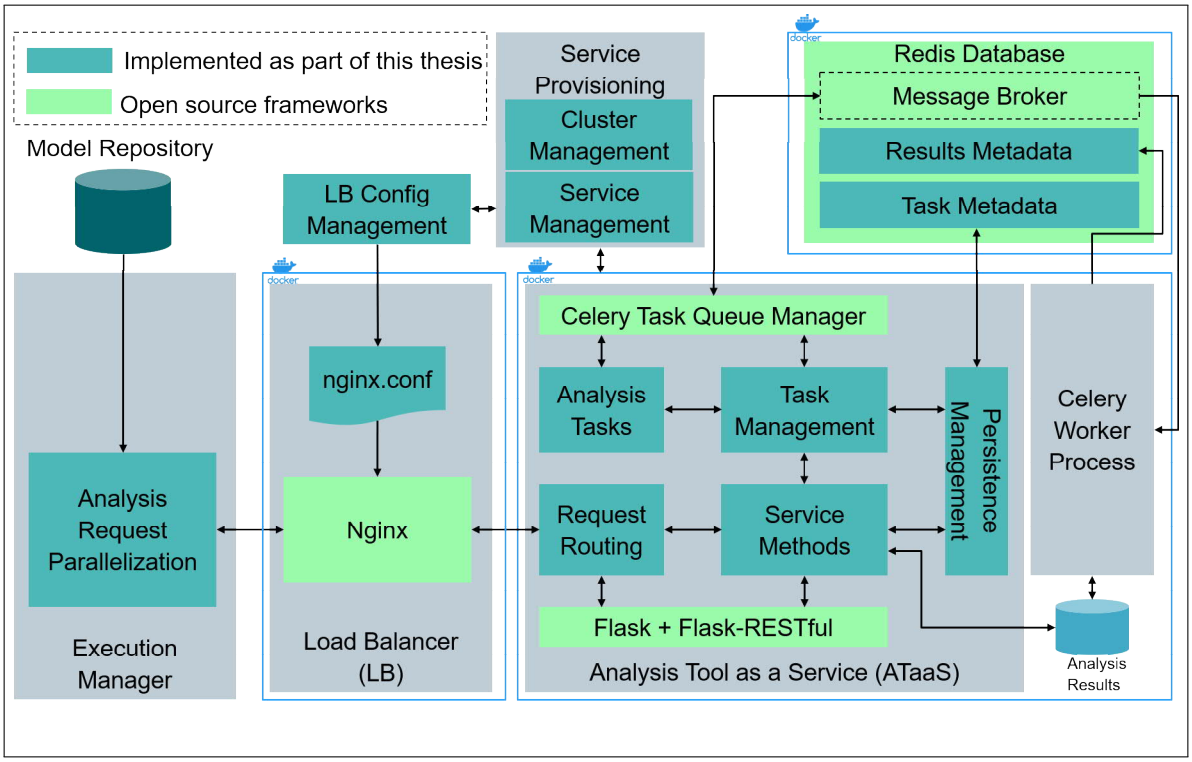


Figure 4.5: Software architecture

Flask and *Flask-RESTful* packages of Python programming language as the web service framework. Flask is one of most popular web application frameworks in Python programming language. It is lightweight and supports incremental development of web services allowing developers to start with a minimal web application and then incrementally

scale up to more complex application. The decision to use the Flask and Flask-RESTful to develop tool services in this thesis is made because the Python programming language is also used in the development of the existing tool integration framework and its tool wrappers. The *Celery* task queue framework is used for implementing asynchronous processing of analysis execution requests and background execution of analysis tasks with the help of *Celery Worker Processes*. The *Request Routing* module is responsible for connecting the API end-points to corresponding *Service Methods*. The operations for processing API requests are implemented in service methods which use methods provided by the *Task Management* and *Persistence Management* modules to process all API requests related to the creation and execution of analysis tasks. The results and task metadata produced by the tool service modules while processing the API requests is stored in a *Redis* database, which is also used as a *Message Broker* by the *Celery Task Queue Manager* for distributing the analysis tasks to Celery worker processes. *Analysis Results* produced after successful completion of analysis tasks are saved in the local file system of the tool service.

The *Analysis Request Parallelization* module of the *Execution Manager* package provides methods required for exercising parallel distributed execution of analysis tasks. The *Cluster Management* and *Service Management* modules provide methods required for provisioning of tool services on service cluster. The *LB Config Management* module creates the *nginx.conf* file used by the *Nginx* load balancer to distribute the incoming traffic to ATaaS instances.

4.3.1 Tool Services

A tool service consists of following modules that collectively handle the requests from clients and perform the specified analysis task.

Request Routing

The *Request Routing* module of the tool service application starts the service instance and manages the client-server interactions. It consists of the following phases:

- **Connection management:** Request routing listens for the incoming connections on the port on which the tool service is serving the requests and establishes connection with clients.
- **Request validation:** Request routing validates an incoming request for correctness as per the given API specification and for completeness of associated metadata or other data items such as files.
- **Invocation of service methods:** Request routing connects a valid incoming request from a client to the corresponding service method that is responsible for processing the request.
- **Response delivery:** Request routing delivers the response generated by a service method after processing a request to requesting the client.

Service Methods

A *Service Method* performs the action required to process a request from the client. It involves the following steps:

- **Validating request parameters:** A REST API request originating from a client has different type request specific parameters [58] associated with it. The service method validates these parameters in the context of the request to detect out of range and incorrect-type parameters which would otherwise result in failures during the request processing.
- **Performing requested action:** The service method performs action specified in the request such as creating or starting an analysis task or getting the information about a specific task.
- **Error handling:** Besides performing the validation of the request parameters, the service method is responsible to handle the configuration and runtime errors such as unavailability of a network resource or failure of an analysis task.

- **Generating response:** The service method builds the response in the format specified in the API and returns it to the request routing module. The response could be summarizing a failure or a successful completion of the action specified in the request. The response includes a HTTP status code [24] as per the OAI specifications Section 3.3.4 for REST APIs.

Task Management

The *Task Management* module of the tool service provides methods for **a)** creating an *Analysis Task* and associated metadata such as task ID, **b)** accessing task information such as current status of the task, and **c)** starting/stopping a task. The metadata associated with the task is stored in a *Redis database* which is a lightweight key-value datastore.

Starting a task involves inserting the task in the task queue which is managed by the *Task Queue Manager*. The queue manager distributes the tasks in the queue to a *Worker* process which is running along the tool service. The queue manager uses a *Message Broker* to communicate with the worker process during task allocation. In this thesis the Redis datastore is set up as a message broker.

The worker process executes the analysis task which runs the requested analysis tool for a given configuration and produces outputs which are saved as *Analysis Results* in the internal file system. The worker process also updates the status and the result metadata of the task in the *Results Store* of the database. Stopping a task requires aborting the currently running analysis if the task is already started and removing the task from the task queue.

Persistence Management

The *Persistence Management* module provides the methods for storing the metadata about tasks in the database and fetch the data whenever a client request is made. The metadata is stored in the database with reference to the ID of the task and is persistent over client-server sessions. The metadata is created when an analysis task is created and updated whenever information related to that task changes.

4.3.2 Execution Manager

In the absence of the WEaaS, the *Execution Manager* module, which is developed as part of this thesis, provides the core functionalities required for execution of analysis tool services using the existing in-house tool integration framework. The execution manager provides methods to the workflow designers to request analysis services.

Parallel Distributed Analysis Execution Requests Exchange

An AGT analysis workflow designer aims to run many analysis in parallel for the same engine model but with different configurations such as different temperatures. The analysis request parallelization module of the execution manager provides a method which can be called by the workflow developer to achieve this with a single call. The request parallelization module converts this one method call to multiple simultaneous API requests to ATaaS application to start multiple parallel analysis tasks.

In addition, it also performs following actions:

- Upload the input files necessary for running an analysis tasks
- Periodically check the status of the analysis tasks and manage task failures
- Download the results of completed analysis tasks

4.3.3 Service Provisioning

Cluster Management

The *Cluster Management* module is responsible for creating and managing the ATaaS service cluster of computing nodes on which the service instances are run. The computing resources available for creating the Docker swarm can be the laptops of Siemens AGT employees participating in the project as voluntary computing [65] resources or desktop workstations or AWS computing instances. The cluster manager initializes a Docker

swarm and then adds a computing resource as a manager or worker if that computing resource is available in the network.

Service Management

The *Service Management* module create a ATaaS service on the service cluster by using Docker service command. Creating an ATaaS service involves starting many Docker container from the Docker image of the ATaaS on nodes of the service cluster. Each container acts as an instance of ATaaS service and runs a *Flask-restful Application* and *Celery Worker Process*. The number of server instances to run on the service cluster is configurable. By default, we create one server instance on every node of the cluster.

The Docker Swarm manager has built-in fail-safe mechanism which balances the service in case a node in the cluster goes down. It creates new instance on one of the other nodes in the cluster and tries to make the service available with the specified number of instances.

4.3.4 Load Balancer Configuration Management

The Nginx load balancer, which is used to generate parallel requests, uses a configuration file (*nginx.conf*) file to route the requests to ATaaS service instances running on the service cluster. The configuration file must have a list of IP addresses of all service instances and the port number on which the Docker container is serving the ATaaS services. The *Load Balancer Config Management* module periodically queries the service and gets the latest service configuration. If any change to the Nginx configuration file is needed, the load balancer config manager updates the config file and loads it on the load balancer.

Load Balancer

The load balancer is responsible for directing the requests received from the request parallelization (Section 4.3.2) module of the execution manager to one of the ATaaS service

instance running on the ATaaS service cluster (Section 4.1.2). We used the Nginx[57] load balancer in our implementation of ATaaS. Nginx is a web server framework that can be used as a reverse proxy, load balancer, mail proxy and HTTP cache. The Nginx load balancer is configurable to handle multiple simultaneous requests and in this thesis it is configured to process 1024 parallel requests.

4.4 Summary

This chapter provided an overview of the proposed service architecture for the tool integration framework by describing the main components of the architecture. It also described the software architecture used in implementing a prototype of the proposed service architecture.

Chapter 5

Development of ATaaS Prototype

In this thesis, we proposed a cloud-based microservices architecture for workflow-based tool integration framework for executing AGT design workflows. Development of the complete framework involves developing applications on both client and server side. On the client-side development of desktop and mobile applications that enable definition, visualization and execution of workflows is needed. On the server-side, development of microservices for web UI, workflow execution service and analysis tool service is needed.

In order to demonstrate the feasibility of the proposed architecture we developed a prototype consisting of two microservices providing tool functionalities of two AGT analysis tools, namely Secondary Air System (SAS) and Finite Element Analysis (FEA), and a manager module that allows integration of tool services with the existing tool integration framework. This chapter first describes the APIs provided by the tool services to run analysis tasks (Section 5.1), and then provides an overview of software modules developed in this thesis to implement processing of these API requests (Section 5.2). It also describes how the tool services are deployed on the service cluster and how they are accessed from the existing tool integration framework using an execution manager (Section 5.4).

5.1 Tool Service APIs

REST APIs used for accessing the tool services developed in this thesis are listed in Figure 5.1. The *toolName* path variable is used for specifying the name of the tool service (SAS or FEA). A more detailed description of the APIs is given in the subsequent sections.

HTTP Method	End Point	Description
GET	/tools/<string:toolName>	Get Information about a specific tool service (SAS or FEA)
POST	/tools/<string:toolName>/tasks	Create an analysis task using a specific tool service
GET	/tools/<string:toolName>/tasks/<string:taskId>	Get information of a specific task
POST	/tools/<string:toolName>/tasks/<string:taskId>/input_files	Upload input files for a specific task
POST	/tools/<string:toolName>/tasks/<string:taskId>/start	Start analysis task specified by taskId
GET	/tools/<string:toolName>/tasks/<string:taskId>/status	Get the status of a specific analysis task
GET	/tools/<string:toolName>/tasks/<string:taskId>/results	Download the results of an analysis task

Figure 5.1: REST APIs of SAS and FEA tool services

5.1.1 Create an Analysis Task

The API for creating an analysis task is *POST: /tools/<string:toolName>/tasks*, where *toolName* is a path parameter specified in the request indicating the name of the analysis (e.g. SAS or FEA). It also takes a *body* parameter in the *JSON format* format as shown in Figure 5.2. This parameter is used to specify a *name* and the *type* of the analysis.

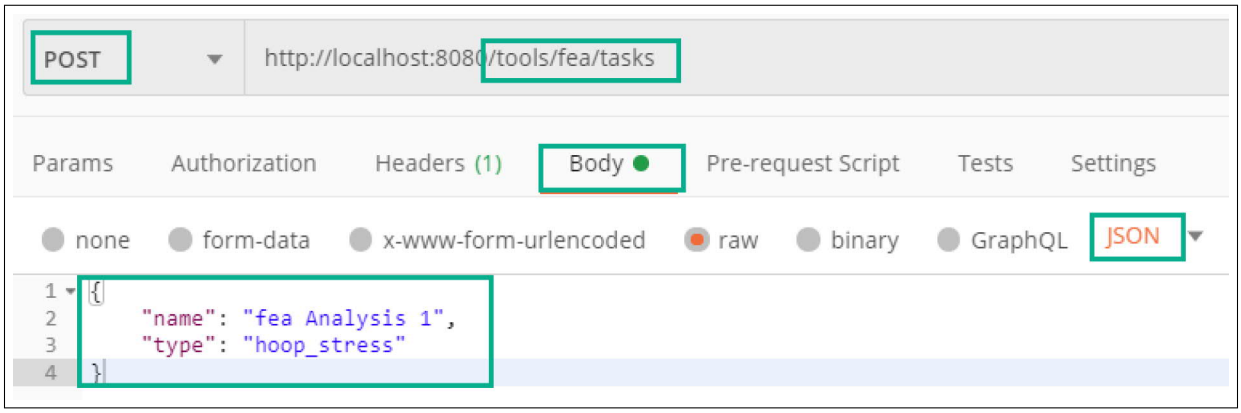


Figure 5.2: API for creating an analysis task

Response: If the tool service creates the task successfully, it returns the HTTP status code 200 along with a response body as shown in Figure 5.3 which is in the JSON format format containing the *task ID* of the newly created analysis task.

Figure 5.3: API response for creating an analysis task

5.1.2 Get Task Information

GET: /tools/<string:toolName>/tasks/<string:taskId> is the API for getting information about a specific task using its *taskId*. Figure 5.4 shows the response for this API request. The response contains a body in the JSON format format and has task *name*, *type* and other fields indicating the current status and runtime information of the task.

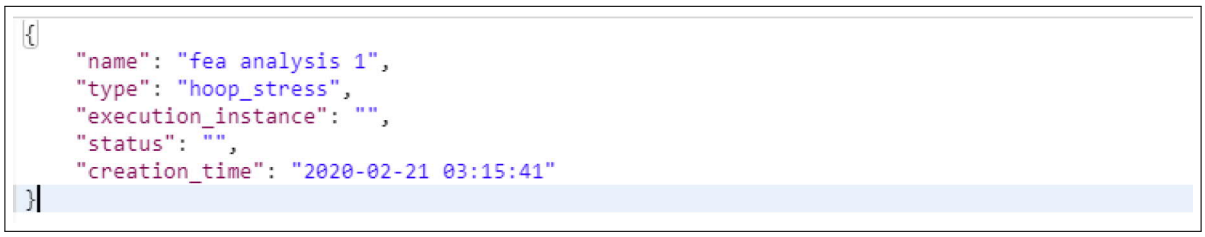


Figure 5.4: API response for get task information

5.1.3 Upload Input Files

POST: /tools/<string:toolName>/tasks/<string:taskId>/input_files API is used by the clients to upload the input files required for executing an analysis task. Figure 5.5 shows an example of this request along with the necessary parameters. The files are sent to the server as part of the request *body* in a *form-data* field. The form-data allows definition of custom fields in the key-value format and includes them in the request body. The *inputFiles* field defined in this API allows clients to send multiple files in single request.

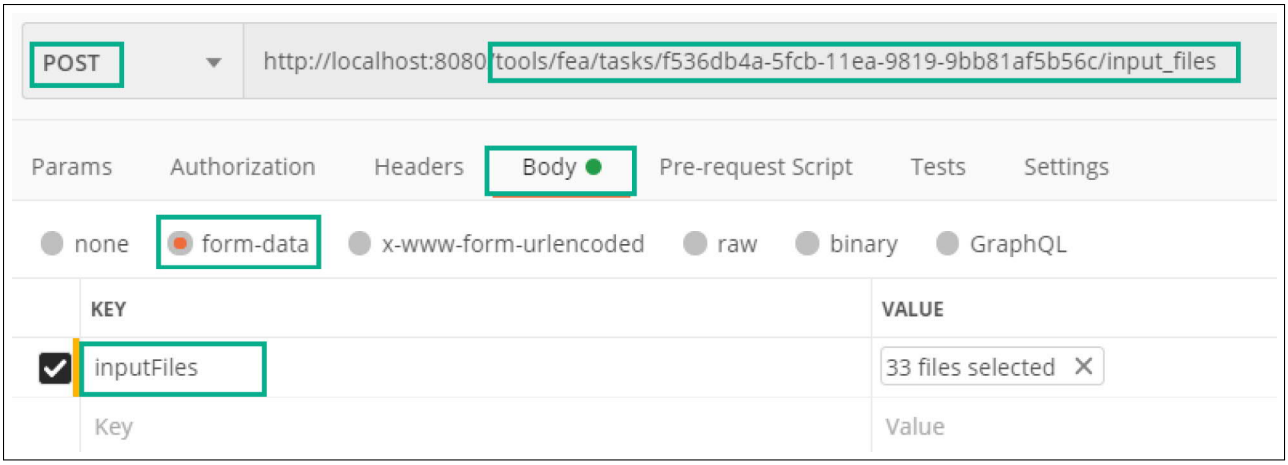


Figure 5.5: API for uploading input files

Response: The response contains the status of the file upload along with the HTTP status code *201* which indicates successful creation of the file elements in the server.

5.1.4 Starting a Task

Once an analysis task is created, and all required input files are uploaded, a client can use the following API to start the task.

POST: /tools/<string:toolName>/tasks/<string:taskId>/start

Response: A response with status code 202 (Figure 5.14) is sent to the client to indicate that the request for starting the task is registered and the task itself will be started in background when the execution resources are available. In case of an error such as invalid task ID an error code 400 is sent as response.

5.1.5 Get Task Status

GET: /tools/<string:toolName>/tasks/<string:taskId>/status API can be used for getting status of an analysis task. An example API request and corresponding response is shown in Figure 5.6. The response is a *JSON object* containing the *execution_instance* ID and the *status* of the task which is one of the *PENDING*, *STARTED* or *DONE* status.

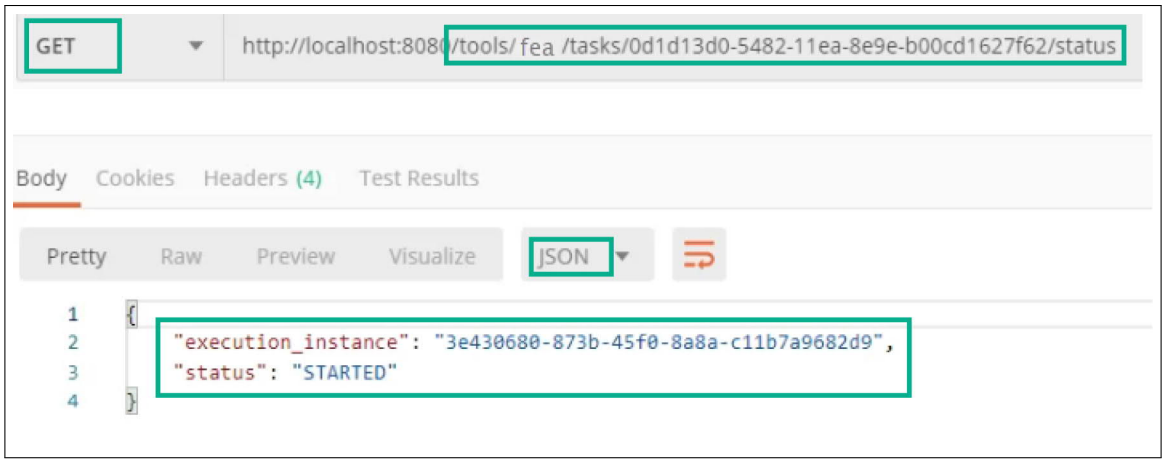


Figure 5.6: API for getting task status

5.1.6 Download Analysis Results

GET: /tools/<string:toolName>/tasks/<string:taskId>/results is the API for downloading the analysis results. Clients shall start this API request in the data streaming mode (*stream=True* in Python using *requests* class) as shown in the code snippet in Figure 5.7 to receive the results file, which is an archive file.

```
with requests.get(url, stream=True) as r:
    if r.status_code == 200:
        with open(local_filename, 'wb') as f:
            copyfileobj(r.raw, f)
        success = True
```

Figure 5.7: Download analysis results

5.2 Development of Tool Services

Processing of the API requests described in the previous section (Section 5.1) is implemented in software modules developed in Python programming language. Operations defined in these modules create and initialize the Flask web service application and Celery task queue, and implement tool execution methods and service methods which are

entry points to API requests. This sections provides an overview of these software modules and describes operations implemented by these software modules.

Software modules: Tool services are developed in Python programming language by using Flask and Flask-RESTful web service frameworks. A tool service consists of many modules that map directly to the concepts of tool services. Figure 5.8 shows most important modules of the tool service that are developed as part of this thesis and their dependencies on the framework classes. The *app* module has instances of the *Flask*, *Flask_restful* and *Celery* classes that are used for configuring the tool service and the task queue. It also maps the REST APIs of the tool service to the corresponding service methods by using the *Api* class of the flask_restful module. Service methods are defined in the user-defined classes (*Tool*, *OutFile*, *InputFile*, *Task*, *ExecutionInstance*, *Persistence*) and they are sub-classes of the *Resource* class which allows the sub-classes to expose a method each for *get*, *post*, *put*, *delete* and *patch* HTTP methods to perform actions corresponding to an end-point request. The *InputFile* class has methods to receive the files uploaded by the clients and uses the *reqparse* class to perform this operation. The *Persistence* class has methods for storing and retrieving the metadata of the tasks and execution instances which are used by the Task and ExecutionInstance classes. It performs the store and retrieve operations by using the corresponding methods provided by the *redis* module. A more detailed description of these modules is given in the following sections by providing flowcharts and code snippets wherever necessary.

5.2.1 Flask Web Service Instance Creation and Initialization

The *flask_app* instance of the Flask class which is from on the Flask package of the Python programming language creates the web server application. This instance is then used as a parameter to the *Celery* class to create the *celery_task_q* object which is used as the task queue for asynchronous processing of analysis task execution requests. The flask_app object is also used for initializing the server application by adding API routes and connect-

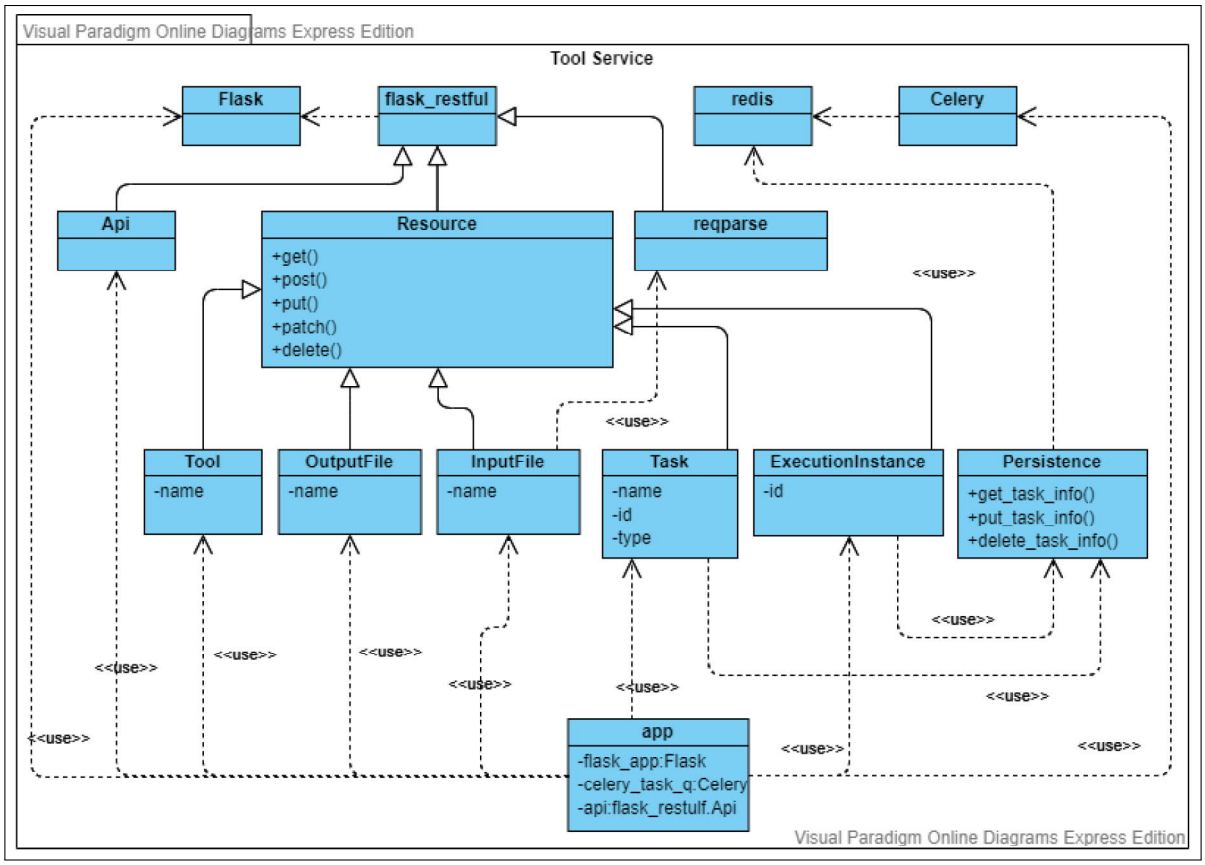


Figure 5.8: Modules of tool service

ing them to corresponding service methods. Finally the tool service is started by calling the *run* method of the flask_app instance as shown in the code snippet in Figure 5.9. By specifying the *host=0.0.0.0* the tool service becomes accessible on the *port=8080* by using the IP address of the Docker container inside which the tool service is running. Specifying the *debug=True*, the tool service is started in the debug mode in which the tool service outputs detailed logs and restarts the application whenever changes are made in the source code. One should start the tool service in the debug mode only during the development phase and strictly remove the debug=True setting in the production mode for security reasons.

5.2.2 Celery Task Queue Instance Creation and Initialization

The *Celery* class from the Celery package initializes task queue and links it with the tool service application. It takes the application *name* (Figure 5.9), and URLs for the message *broker* and results *backend*. The Celery task queue manager uses the message broker URL for connecting to the Redis datastore through which it distributes tasks to the Celery workers. The result backend is used to store the metadata such as task status, and the progress information produced during the execution of the tasks by the workers. The URL should contain the hostname or IP address of the Docker container running the Redis datastore along with the port number on which Redis database is accessible. By setting the *task_track_started* flag to True, the Celery task manager is configured to update the status of the tasks as soon as there is an update, which is not the default behavior.

```
flask_app = Flask(__name__)
flask_app.config.update(
    CELERY_BROKER_URL='redis://tool_service_datastore:6379/0',
    CELERY_RESULT_BACKEND='redis://tool_service_datastore:6379/0'
)
celery_task_q = Celery(flask_app.name, broker=flask_app.config['CELERY_BROKER_URL'], \
                       backend=flask_app.config['CELERY_RESULT_BACKEND'])
celery_task_q.conf.task_track_started = True
flask_app.run(host='0.0.0.0', port=8080, debug=True)
```

Figure 5.9: Tool service creation and initialization

5.2.3 Analysis Tool Executors

The development of a software component that executes an analysis tool for a given configuration requires understanding of the tool capabilities and how to run the tool. In this thesis these components are developed as standard Python methods and qualified as Celery tasks by using a specific decorator provided by the Celery task queue framework. This section describes the methodology we followed to develop SAS and FEA analysis tool executors.

Executor for SAS Analysis Tool

SAS analysis is performed by using a tool called SAS solver which takes an SAS Model, performance data, and aero-thermal data as inputs. Optionally, the SAS solver can take results from engine runs performed on the test rigs and thermo-mechanical analysis as inputs. The execution of the solver is done for a range of ambient temperature typically ranging from -50C to +50C. The analysis for one temperature is done in two stages.

1. **Calculation of boundary conditions:** Using the performance and aero-thermal data boundary conditions required for further analysis are calculated.

2. **Running the solver on the SAS model:** The solver now uses the boundary conditions and calculates the temperatures and pressures required at the source and sinks of the network specified in the SAS Model. This is an iterative step and repeated until the source and sink conditions are converged.

Tool executor method: Figure 5.10 shows the flowchart of the Python method implemented for the SAS analysis task. The method is then qualified as a Celery task by using the *@celery.task* annotation. Only a high-level flowchart of the method is given due to the confidentiality agreement associated with the Siemens AGT resources. To begin with, the method executes *Calculate boundary conditions* step, in which it runs the boundary condition calculator tool by using *Performance data* and *Aero-thermal data* as inputs. If there were no *Errors* in boundary conditions, *Save boundary conditions* is executed to save *Boundary conditions* in to a file. In the *Run solver* step, the method executes the SAS network solver tool by using the *SAS Model* and boundary conditions. If the solver tool runs successfully, *Results* are generated and the *Save results* step is executed. The *Store errors* step stores errors in the Celery result backend.

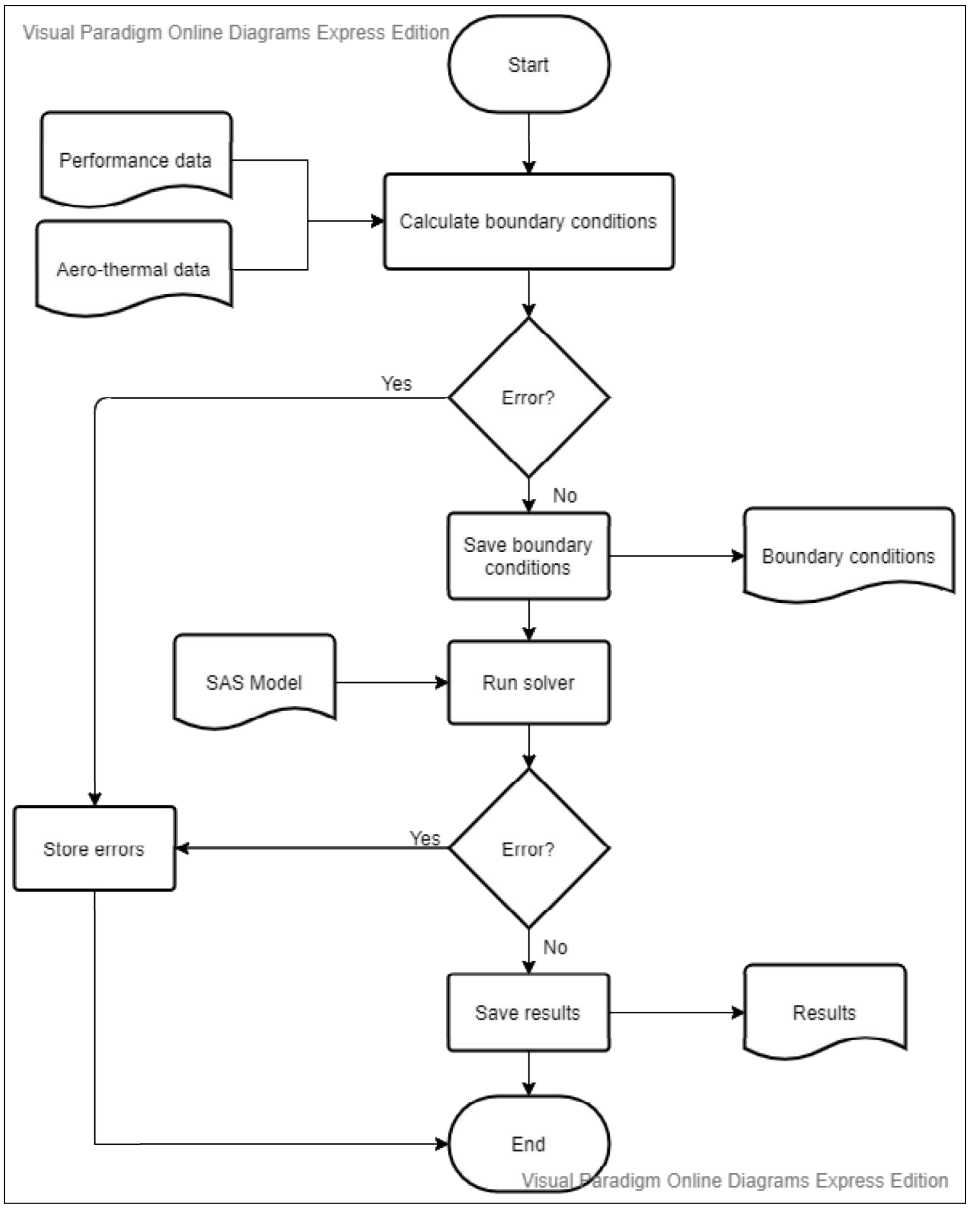


Figure 5.10: SAS analysis flowchart

Executor for FEA Tool

FEA analysis is performed by using the Siemens FEA tool by running a script which is written in the tool specific scripting language. The script consists of statements to load libraries and models, and run different type of finite element analysis such as structural, thermal, or mechanical to produce Finite Element Models (FEM).

Tool executor method: Figure 5.11 shows a snippet of the the Python method in which the fea tool execution operation is implemented. It runs the *script files* corresponding to the given analysis task by running the *fea_tool.exe* in shell mode and examines the *return code* to determine if the tool was run successfully or not. If the tool fails, an error message along with the error code produced by the fea_tool.exe is returned by the worker running the analysis task.

```
for file in script_files:
    output = subprocess.run([ C:/tools/fea_tool.exe', '-b', '-s {}'.format(file)], shell=False)
    if output.returncode != 0:
        os.chdir(working_dir)
        return{
            'message' : "Error while running fea tool",
            'return_code': output.returncode
        }
    else:
        os.chdir(working_dir)
        return {
            'status': 'Done'
        }
```

Figure 5.11: FEA analysis flowchart

5.2.4 Service Methods

Task Creation Method

The service method for processing the task creation API request is implemented in the *Task* (Figure 5.9) module. Figure 5.12 shows the snippet of the code that performs the task creation operation. The method generates the *taskId* which is a UUID in the RFC4122[35] format and is generated using the *uuid1* [64] class of the *uuid* Python module. The uuid1 class generates the UUID from the host ID, sequence number, and the current time. At this stage, the method populates the task information (*taskInfo*) structures using the information available in the *request parameter* and stores it in the *persistence* datastore by using the task ID as the key. In the event of an error in creating the task, a HTTP status code [24] indicating the error will be returned.

```
parameters = request.json
# generate a task_uid
taskId = uuid.uuid1()
# Store it in a datastore
try:
    taskInfo = {'name': parameters['name'], 'execution_instance': "", 'type' : parameters['type']}
except TypeError as e:
    return {'message': e.__str__()}, 400
persistence.put_task_info(str(taskId), taskInfo)

# Return task_uid
return {"task ID" : str(taskId)}, 200
```

Figure 5.12: Service method for creating task

Get Task Information Method

The service method for processing the *Get Task Information* API is implemented in the *Task* (Figure 5.9) module and it retrieves the task information from the persistance datastore by using the specified taskId as the key and returns it with the response code 200.

Input Files Receive Method

The file receive operation is implemented in the post method of *InputFile* (Figure 5.9) module. As shown in the code snipped in Figure 5.13, if the specified task ID is valid, this method creates a directory having the same name as task ID and saves all the files in it. It then sends a response containing the *status* of the file upload along with the HTTP status code *201* which indicates successful creation of the file elements in the server.

```
Path(UPLOAD_FOLDER+'/'+taskId).mkdir(parents=True, exist_ok=True)
for file in data['inputFiles']:
    file.save(os.path.join(UPLOAD_FOLDER+'/'+taskId, file.filename))

return {
    'message':'File uploaded',
    'status':'success'
    }, 201
```

Figure 5.13: Service methods for receiving input files

Task Start Method

The operation for handling the task start request is implemented in the *post* method in the *ExecutionInstance* (Figure 5.9) class and the Figure 5.14 shows the code snippet which inserts the task in the task queue by calling the *apply_async* method on the task.

```
if Path(os.path.join(RUNTIME_FOLDER, taskId)).exists():
    taskInfo = dataStore.get_task_info(taskId)
    if taskInfo['execution_instance'] == '':
        executionInstance = tool_execution_instance.apply_async([taskId])
        taskInfo['execution_instance'] = executionInstance.id
        taskInfo['status'] = 'IN_Q'
        import datetime
        taskInfo['creation_time'] = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        logging.debug('taskInfo in ExecutionInstance Post method {}'.format(taskInfo))
        dataStore.put_task_info(taskId, taskInfo)
        return taskInfo, 202
```

Figure 5.14: Service methods for starting the task

Get Task Status Method

The operation to process the get task status request is implemented in the *get* method of *ExecutionInstance* class and has logic has as shown in Figure 5.15. The *taskInfo* of the task is first retrieved from the *persistence* datastore using its *taskId*. Then the task status is fetched from the celery result backend by calling the *AsyncResult* method of the celery task and returned to the client along with service code 200. An error message is returned in case of an invalid API request having incorrect request parameters.

Download Analysis Results

The result produced from an analysis depends on the type of the analysis.The service method that processes the download results request collects all output files, creates an *archive* in the *zip* format, and sends it to the client via a *send_file* operation as shown in the code snippet in Figure 5.16.

```
if toolName is not '' and taskId is not '':
    taskInfo = persistence.get_task_info(taskId)
    executionInstanceId = taskInfo['execution_instance']
    executionInstance = tool_execution_instance.AsyncResult(executionInstanceId)
    # if executionInstance.status != 'FAILURE':
    return {
        'execution_instance': executionInstance.id,
        'status' : executionInstance.status
    }, 200
else:
    return{
        'message': 'Incorrect API'
    }, 400
```

Figure 5.15: service method for getting task status

```
for file in list_of_files:
    shutil.copyfile(file, temp_result_dir.joinpath(file.stem+file.suffix))
fileName = shutil.make_archive(temp_result_dir, 'zip', temp_result_dir)
return send_file(fileName, as_attachment=True,
                 attachment_filename=Path(fileName).stem+Path(fileName).suffix)
```

Figure 5.16: Download analysis results

Linking APIs to service methods

The *api* instance (Figure 5.8) of the *flask_restful.Api* class is used to map service APIs to corresponding service methods. It has a method named *add_resource* which takes the name of the Python class and one or more API end-points (specified as strings) as arguments and establishes the links between them such that the HTTP POST, GET, PUT, DELETE requests made with these APIs invoke corresponding post, get, put and delete methods defined inside the Python class linked to the APIs. The control flow inside these methods must be implemented to identify the incoming request (incase one class is linked to more than one API request). For example, the following call to the *add_resource* method connects the */tools/<string:toolName>/tasks* and */tools/<string:toolName>/tasks/<string:taskId>* APIs to the post, get, put and delete methods defined in the *Task* class. With these links established, if a `'GET /tools/<string:toolName>/tasks/<string:taskId>'` request is made, the *get* method defined in the Task class is invoked.

api.add_resource(Task, '/tools/<string:toolName>/tasks',
'/tools/<string:toolName>/tasks/<string:taskId>')

5.3 Deployment of Tool Services

Tool services are deployed by launching many Docker containers - created from Docker images of the tool services - of tool service on the distributed service cluster and configuring the load balancer to distribute traffic to the containers. In this thesis, we built the Docker images of the tool services and developed Python modules shown in Figure 5.17 to facilitate deployment of these images.

Cluster Management package has a *Manager* module that contains methods to create and manage a Docker swarm cluster. The *Manager* module in the *Service Management* package is responsible for provisioning the service on this Docker swarm cluster. The *LB Config Management* module has a *Config Manager* module which is responsible for monitoring the services running on the Docker Swarm cluster and generating a configuration file for Nginx load balancer. It uses *DockerSwarmInspect* and *NginxConfigBuilder* modules to accomplish this. An high-level flowchart of these modules is described in the following sections.(Section 5.3.2, Section 5.3.3, Section 5.3.4)

5.3.1 Docker Images for Tool Services

Since the analysis tool services are hosted in Docker containers, Docker images from which containers can be created must be built. A Docker image for a tool service must have the tool (SAS or FEA), all the dependencies needed to run the tool (such as Visual C++ libraries) and the tool service application, and should be configured to start the tool service when a container is created from it.

The Docker container only supports command line environment. Therefore, only the tools that support batch mode execution can be containerized. Creating a Docker image that runs an analysis tool is quite challenging and involves dealing with compatibility

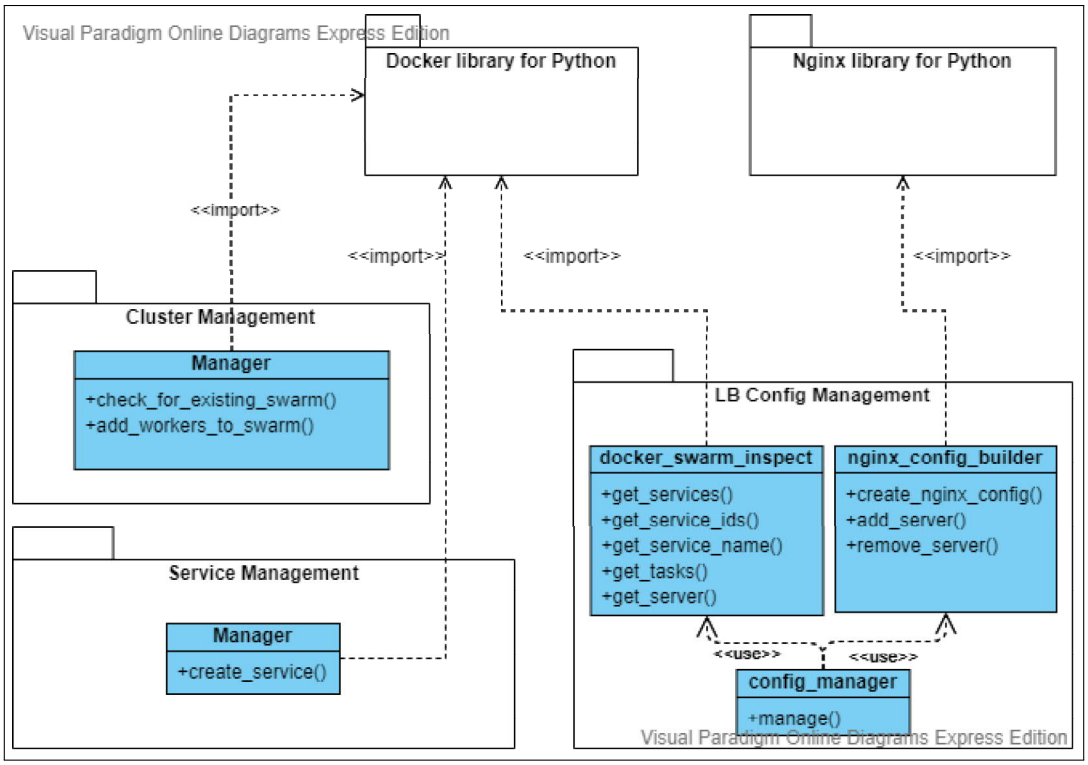


Figure 5.17: Software modules facilitating deployment of tool services

issues. Analysis tools used at Siemens have long history with many tools dating back to early 1990s. Few tools run flawlessly on the latest operating systems while few other will need older version of libraries such as visual C++ libraries on MS Windows OS.

Dockerfile for Building Tool Service Images

In the Docker ecosystem a Docker image is created with the help of Docker build system consisting of base images and Dockerfiles. A base image contains operating system environment needed to install and run the analysis tools and their dependencies. In this thesis, the *mcr.microsoft.com/windows/servercore:1803*[67] is used as the base image because both SAS and FEA tools are windows tools.

Creating a Docker image of the tool service is done by writing a Dockerfile which has statements for performing the following actions:

- **Installing tool dependencies:** All tool dependencies such as visual C++ libraries and Python need to be installed. A Dockerfile statement for installing the Python is shown in Figure 5.18 is and it uses the *RUN* statement to specify a *powershell* command to install the Python from its setup file.

```
RUN powershell.exe -Command Start-Process c:\python-3.6.8-amd64.exe
        -ArgumentList '/quiet InstallAllUsers=1 PrependPath=1' -Wait ;
```

Figure 5.18: Dockerfile statement for installing tool dependencies

- **Setting up the tool:** Both SAS and FEA tools used in this thesis are set up by simply copying them to locations in the container file system and setting up the environment variables required for running these tools.

- **Setting up the tool service:** The SAS and FEA tool services which are developed as Python packages are copied to *designated* location in the container file system.

- **Setting up the virtual environment:** The tool service has dependencies on Flask, Flask-RESTful, Celery packages. A virtual environment is setup using *virtualenv* tool to satisfy these requirements using the Dockerfile statements shown in Figure 5.19. The *requirements.txt* file has a list of the packages that are needed by the tool service package and a *pip install* command installs all packages in it on the virtual environment.

```
COPY ["/requirements.txt", "c:/fea_service/"]
RUN pip install virtualenv
RUN powershell virtualenv.exe c:/fea_service/venv_p368
RUN powershell c:/fea_service/venv_p368/Scripts/Activate.ps1; \
        pip install -r c:/fea_service/requirements.txt
```

Figure 5.19: Setting up virtual environment

- **Exposing ports:** The port 8080 on which tool services on the container are accessible is published by using the *EXPOSE* statement.

- **Setting up the container entry point:** A container entry point is the command or script that should be run when a container instance is created. It is specified by using the *ENTRYPOINT ["powershell", "c:/fea_service/start_server.ps1"]* command for the FEA tool service, where *start_server.ps1* is the powershell script that starts the Flask application which serves the tool service, and the celery worker process which executes analysis tasks in the background.

  The worker process is started by using the *celery.exe* application as follows:

  celery worker -A flask_app.celery_task_queue –pool=solo

  Where *flask_app* is the Flask web service instance defined in the *app* module (Section 5.2) and *celery_task_queue* is the instance of the celery class. The *–pool=solo* option is specified to use one thread per worker to run the analysis task.

5.3.2 Cluster Management

The Cluster Management (Figure 5.24) module creates the ATaaS service cluster on which the service instances are run. It takes a JSON configuration file (Figure 5.21) which has a list of hostnames of computing resources that can be joined as a *manager* or a *worker* node. The computing resources available for creating the Docker swarm can be laptops of Siemens AGT employees participating in the project as voluntary computing resources or desktop workstations or AWS computing instances. The swarm manager initializes a Docker swarm and then adds a computing resource as a manager or worker if that computing resource is available in the network.

The computing resources participating in project in voluntary computing model have a dynamic IP address which changes every time the computing resource leaves the network and reconnects. The IP address of the workstations changes on reboot. The cluster management module keeps track of the IP address of the machines that are part of the swarm and identifies any changes in the IP address. Whenever a change in IP address

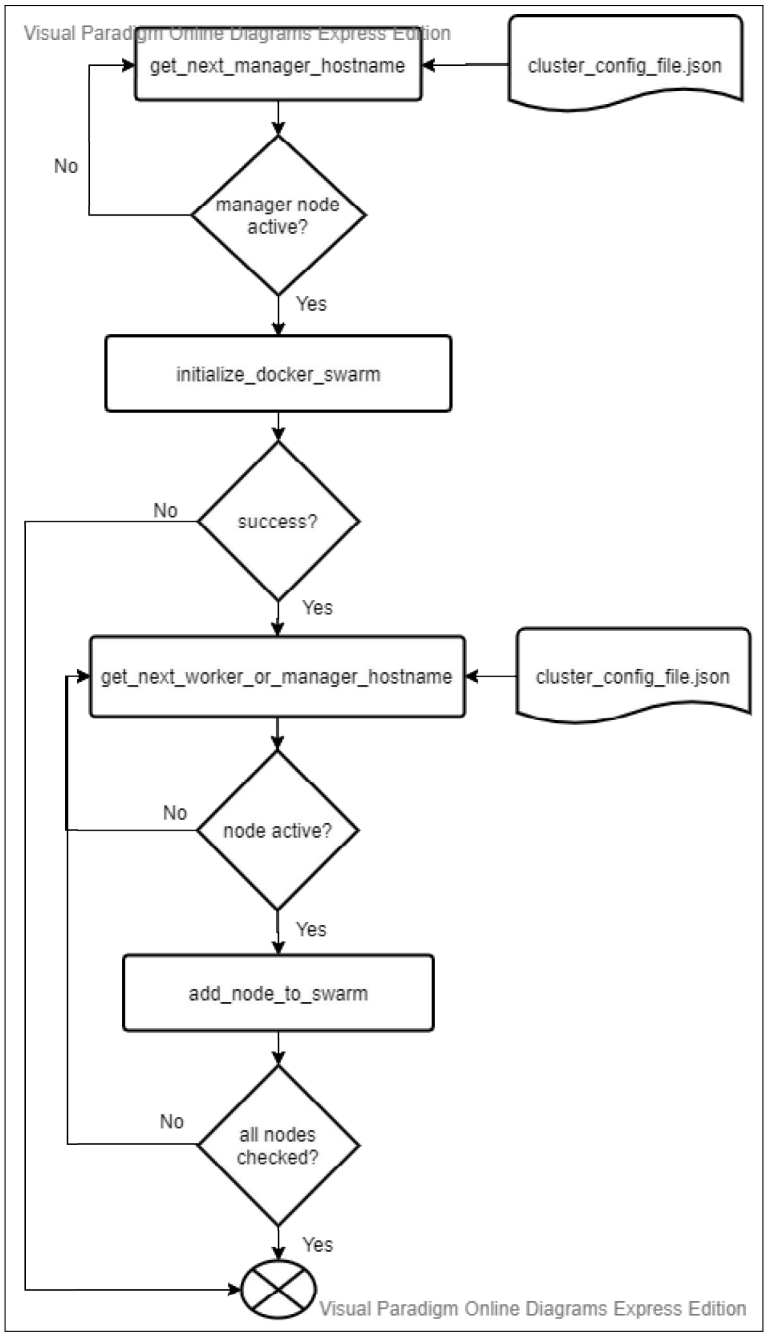


Figure 5.20: Cluster management flowchart

is detected, it forces the node to leave the swarm and rejoin the swarm with the new IP address.

Figure 5.21: Cluster configuration file

5.3.3 Service Management

The Service Management module creates a tool service on the service cluster by calling the *create_service* method, which is defined as shown in Figure 5.22. The create_service method runs on a *managerNode* and it creates the tool service by calling the *services.create* method of the Python docker module by using the Docker *image* of the tool service. The placement of the tool service instances is internally managed by the Docker engine and by default it creates at least one service instance on each node. If a node in the cluster goes down for some reason, the Docker engine will create a new instance in one of the other available nodes in the cluster such that specified number of instances are always running on the cluster.

5.3.4 LB Config Management

The load balancer configuration management module provides a *Manager* class that takes the hostname of the manager node in the ATaaS cluster and generates an Nginx config-

```
def create_service(self, managerNode):
    docker_client = docker.DockerClient(base_url=r'http://{}:2375'.format(managerNode))
    try:
        docker_client.ping()
        from docker.types import EndpointSpec
        docker_client.services.create('images/sas:latest', name='sas_as_service')
    except (ConnectionError, TimeoutError, requests.exceptions.ConnectionError):
            logging.error('Could not connect to {}. Node may be down'.format(managerNode))
```

Figure 5.22: Method of creating a service

uration file. Along with the host name a port number on which the Docker engine on the manager node is accessible can be specified. If not specified, the default port number 2375 will be used.

Figure 5.23 shows the processing done in the ConfigManager class. It uses methods defined in the *DockerSwarmInspect* class (implemented as part of this thesis) to get the IP addresses and port numbers of the Docker containers hosting the analysis tool service and updates a *template configuration file* to generate a new *nginx.conf* file. The template configuration file is edited with the help of methods implemented using the *nginx* open-source Python module.

5.4 Development of Execution Manager

An overview of the high-level functionalities of execution manager is given in Section 4.3.2. This section gives an overview of the implementation aspects. The execution manager module is developed in Python programming language which is also the programming language used in the development of the existing tool integration framework so that the two components seamlessly integrate.

The *Execution Manager* as shown in Figure 5.24 consists of a module that has methods which can be called by the workflows in the existing tool integration framework to start parallel distributed execution of analysis for many configurations.

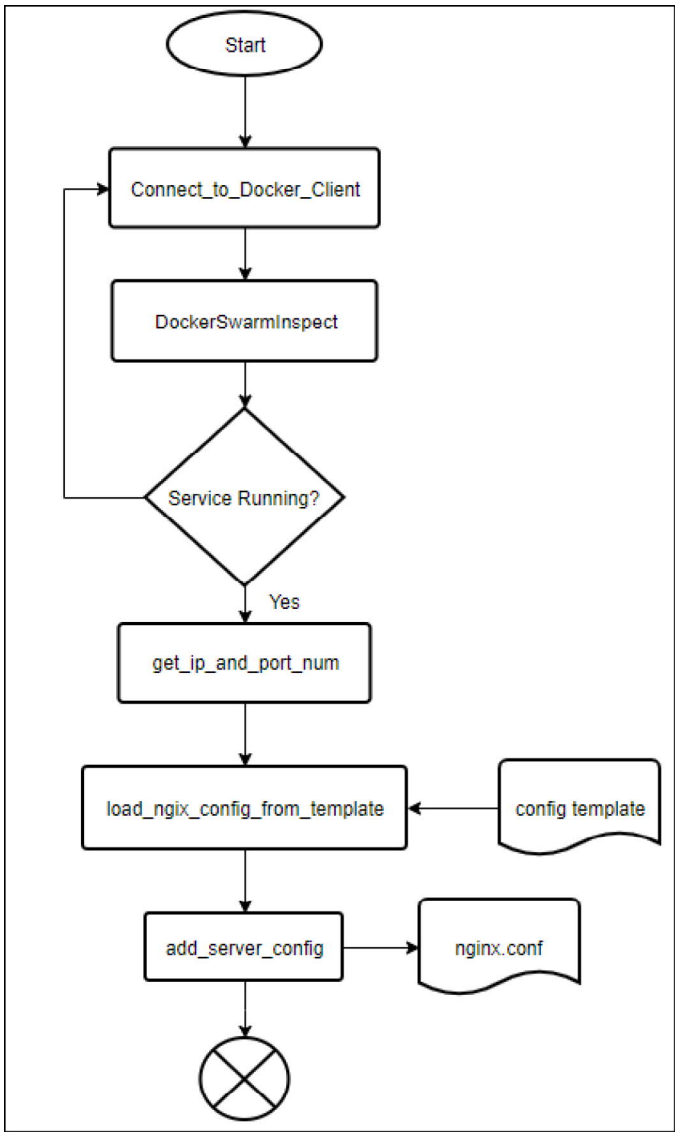


Figure 5.23: Load balancer configuration management

Analysis Request Parallelization: The request parallelization module provides methods for workflow developers to request execution of SAS and FEA analysis for multiple configurations. Section 4.3.2 shows methods implemented in the *Analysis Request Parallelization* Module. A workflow designer can request for a particular analysis by calling the *run_analysis* method by passing a JSON object having information of analysis configurations. The run_analysis method then calls the *run_analysis_for_n_configs* method which is implemented using Python *Asyncio*[21] module. The run_analysis_for_n_configs starts the *run_analysis_for_one_config* method in multiple threads from the thread pool created by us-

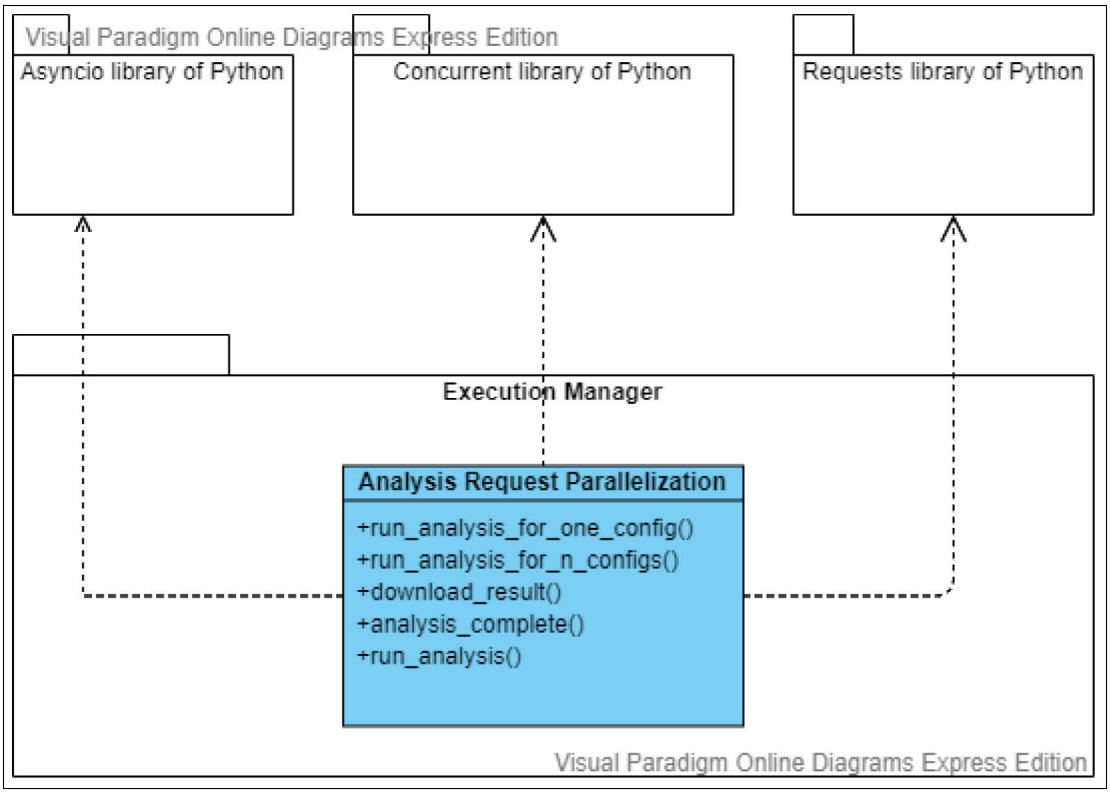


Figure 5.24: Execution manager module

ing the *Concurrent* library. Each run_analysis_for_one_config method sends API requests and performs response handling by using the *requests* library and starts analysis tasks. It then periodically checks the status of the analysis tasks and downloads the analysis results by calling the *download_result* method. Alternatively, the run_analysis_for_one_config can register an *analysis_complete* callback method which would be called automatically when a request processing is complete.

5.5 Summary

This chapter provided an overview of the prototype of the proposed service architecture for Siemens AGT analysis tools. It described the REST APIs exposed by the tool services and the software modules in which processing of these APIs is implemented. The software modules are implemented in Python programming language by using Flask

and Flask-RESTful web service frameworks for developing tool services, and the Celery framework for asynchronous background execution of the analysis tasks. Docker images containing the tool service packages are created to deploy the tool services on the service cluster. An execution manager module which integrates with the existing tool integration framework and provides the methods to run multiple analysis in parallel using asynchronous request processing is also described.

Chapter 6

Performance Evaluation

In this thesis, we proposed the use of cloud-based web services and containerization to develop a prototype of the ATaaS for Siemens AGT analysis tools. The prototype is developed by adopting the service architecture proposed in the thesis for addressing the challenges (Section 2.4.4) in the existing tool integration framework. In this chapter, we present the performance evaluation of the prototype by giving an overview of research questions, environments, and methodologies, and provide an analysis of results obtained for this evaluation.

6.1 Research Questions

We address the following research questions by experimental investigation to measure the effectiveness of the cloud-based web services designed for executing Siemens AGT analysis tools in addressing the challenges presented in the thesis.

- **RQ1::** What is the overhead of running the analysis tasks using ATaaS?
- **RQ2::** What is the effect of parallel and distributed ATaaS for multiple configurations when executed on multiple computers?
- **RQ3::** What is the effect of increasing the number of analysis service instances on a fixed computing infrastructure?

In order to answer these research questions, we set up a benchmarking environment and ran SAS and FEA analysis tasks by creating different analysis configurations. We then analyzed the results obtained from these experiments and derived conclusions.

6.2 Benchmarking Setup and Execution Methodology

This section gives an overview of the test setup, hardware and software used for performance evaluation along with the execution methodology employed for different research questions.

6.2.1 Benchmarking Setup

The setup used for performing benchmarking experiments is shown in Figure 6.1. The setup consisted of computing resources made available by Siemens for this project. The setup used a laptop computer and a workstation computer connected to the Siemens network and sharing network resources with the other computers connected to the network at the same time. The setup was used for local sequential execution and for the parallel distributed execution of the analysis tasks.

Local sequential execution: For local execution of the analysis tasks, only the *engineer's laptop* is used with the following items:

- **Siemens tool integration framework:** The Siemens tool integration framework, which is installed locally on the engineer's machine, provides the workflow required to run the analysis for one or engine configurations. The workflow itself is designed by the domain experts and is made available in the tool integration framework for other users.
- **Analysis tool:** The workflow provided in the tool integration framework invokes the locally installed *analysis* tools with the help of a tool connector.

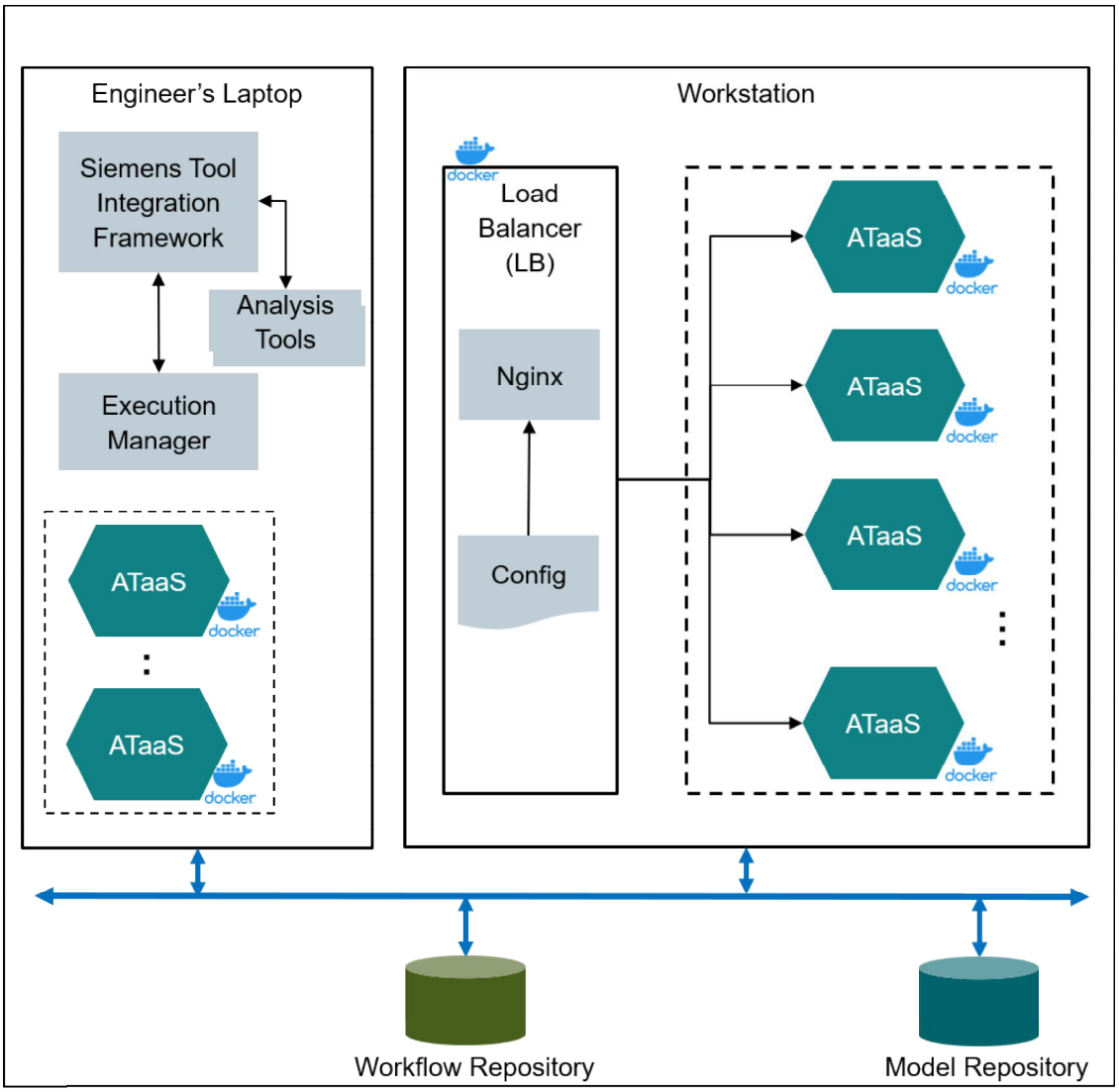


Figure 6.1: Benchmarking setup

- **Model repository:** The analysis tool requires an engine design model as input along with an analysis configuration. The model files are stored in a *model repository*, which is a shared network place.

Parallel distributed execution: The setup used for parallel distributed execution of analysis tasks consisted of many instances of *ATaaS* hosted on a Docker Swarm cluster, which had two nodes, one of which was the engineer's laptop and the other was a *work-*

station computer. The workstation was set up as a manager node in the Docker Swarm, and the engineer's laptop was set up as the worker node.

The tool connector in the Siemens tool integration framework running on the engineer's laptop was set up to call the execution manager's methods to start the distributed parallel execution of analysis by sending RESTful API requests to the service instances running on the cluster. The *Nginx load balancer* running on a Docker container in the remote workstation distributes the incoming requests to one of the many instances of the ATaaS running on the cluster based on the servers list available on the *configuration* file.

6.2.2 Hardware and Software

The hardware configuration of the computing resources used in the benching marking setup is shown in Figure 6.2. The engineer's laptop which was a worker node in the Docker Swarm cluster was a *HP Zbook 15 G5* model laptop computer having *Intel i7* processor along with *32 GB RAM*. The workstation was a *HP Z640* computer with *Intel Xeon* processor and *128 GB of RAM*. While the laptop was connected to the Siemens network through a standard *wi-fi* interface, the workstation was connected through the standard *Gigabit ethernet* interface.

Hardware Configuration		
	Engineer's Laptop	**Workstation**
Model	HP ZBook 15 G5	HP Z640 Workstation
Processor	Intel(R) Core(TM) i7-8850H CPU @ 2.60GHz, 2592 Mhz, 6 Core(s), 12 Logical Processor(s)	Intel(R) Xeon(R) CPU E5-2667 v3 @ 3.20GHz, 3201 Mhz, 8 Core(s), 16 Logical Processor(s)
RAM	32GB	128GB
OS	Microsoft Windows 10 Enterprise (64-bit)	Microsoft Windows 10 Enterprise (64-bit)
Network Access	Through Wi-Fi (802.11n)	Gigabit Ethernet

Figure 6.2: Hardware configuration of the benchmarking setup

Software tools shown in Figure 6.3 were used to set up the analysis services and to execute analyses tasks. The *Siemens tool integration framework* along with the *tool connectors* were used to generate different run configurations which were used as inputs to the *SAS* and *FEA* analysis tools and ATaaS. The Siemens tool integration framework, ATaaS and

the execution manager are built using *Python 3.5.1* and require many packages in order to operate. The ATaaS is built using the *Flask* and *Flask-RESTful* framework and uses *Celery* and *Redis* packages. The ATaaS are hosted using *Docker desktop* platform Windows 10 operating system using the *Docker images* for the *SAS* and *FEA* analysis tools.

Software Tools (versions)	
Siemens Tools	**Other**
Tool integration framework (3.0.1)	Python (3.5.1)
Tool connectors in tool integration framework	Python packages needed for running the tool integration framework and tool wrappers
SAS analysis tool	Flask (1.0.3), and Flask-RESTful (0.3.8) web service framework
FEA analysis tool	Python Redis package (3.2.1)
	Celery task queue framework (4.3.0)
	Docker desktop (2.1.0.3)
	Docker images for SAS tool service
	Docker images for FEA tool service

Figure 6.3: Software tools used for benchmarking

6.2.3 Execution Methodology

In order to address the research questions, we ran analysis tasks using the ATaaS and compare their execution times with local execution. First, we created analysis configurations with different models and gas turbine parameters then we performed execution steps required to answer a specific research question.

Creation of analysis configurations: Analysis configurations were created with the help of workflows available in the Siemens tool integration framework by selecting models and specifying analysis parameters such as temperature, pressure, and humidity. The creation of the analysis configuration using the Siemens tool integration framework consists of the following steps.

1. **Create tasks:** Tasks were created in the Siemens tool integration framework, and SAS and FEA workflows were added into the task. The tool integration framework

has a UI that allows users to select the inputs needed for the task through which required inputs are specified.

2. **Start workflow:** Workflows are stared using the workflow execution UI of the tool integration framework, which in turn starts the tool connector UI specific to the SAS or FEA analysis.

3. **Configure analysis:** Model and different configuration parameters such as temperature, pressure, and humidity were selected by using the UI of the analysis workflow.

Methodology for RQ1: To answer **RQ1**, we used the following execution steps.

1. **Execute and record:** Firstly, we executed analysis for one engine configuration by running the analysis tools directly on the user's computer and recorded the execution time. Then, we executed the same analysis by using the one ATaaS instance running on the same computer.

2. **Repeat:** Step 1 was repeated four to five times each for SAS and FEA analysis.

3. **Compare:** We compared the mean execution times for each analysis type to understand the effect of overhead involved in running the analysis tasks using ATaaS for SAS and FEA analysis

Methodology for RQ2: To answer **RQ2**, we used following execution steps.

1. **Execute and record:** Firstly, we executed SAS analysis for 22 different configurations and FEA analysis for 3 different configurations by running the analysis tools directly on the user's computer. We then executed the same analysis by using the 4 ATaaS instances of SAS and 3 ATaaS instances of FEA running on a distributed computing environment, as described in Section 6.2.

2. **Repeat:** Step 1 was repeated four to five times each for SAS and FEA analysis.

3. **Compare:** We compared the mean execution times for each analysis type to understand the effect of overhead involved in running the analysis tasks using ATaaS for SAS and FEA analysis

Methodology for RQ3: In order to answer **RQ3**, we increased the number of instances of SAS ATaaS to 8 on the same cloud infrastructure as used in experiments for **RQ2** and executed the same SAS analysis tasks to observe the effect on the execution times.

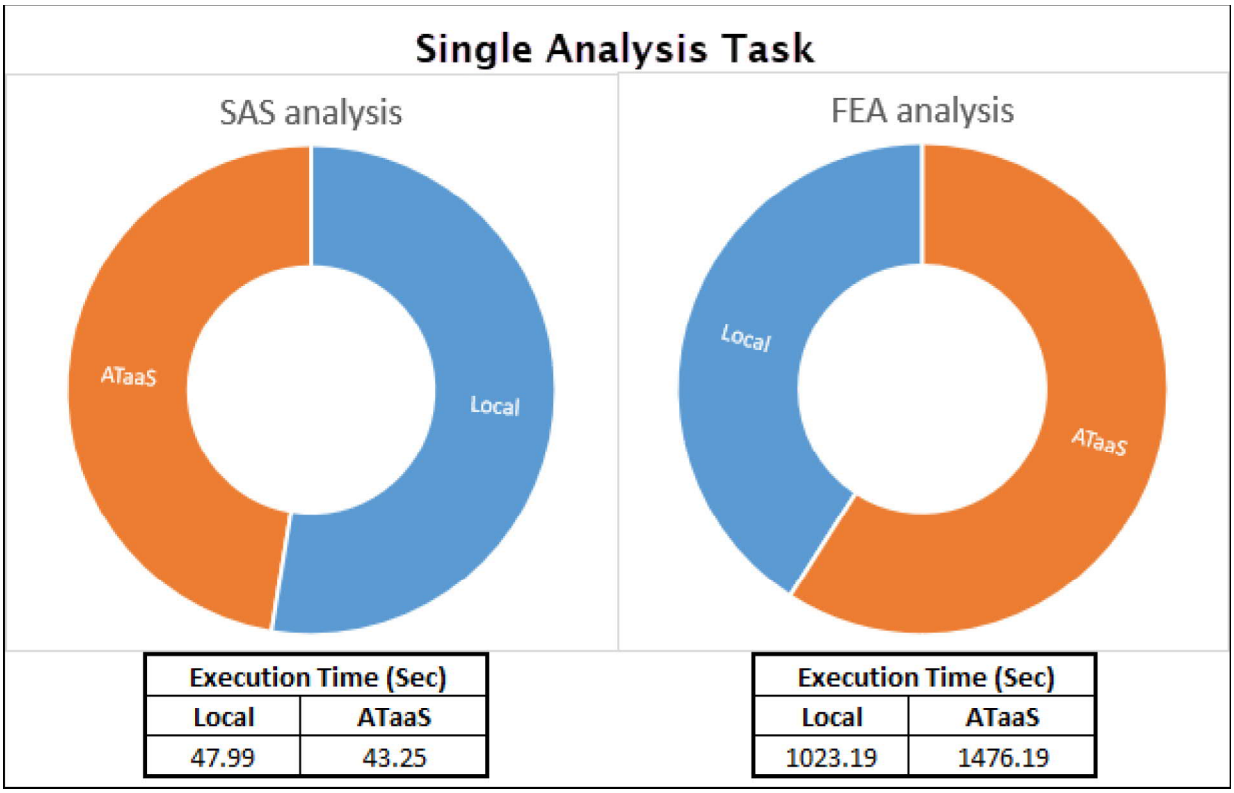
6.3 Analysis of Results

6.3.1 RQ1: Effect of ATaaS execution on single local computer

Figure 6.4 shows a comparison between the execution times for analysis tasks executed by using the tools installed locally on the engineer's computer (baseline) and ATaaS instance running on the same computer. This measurement gives an understanding of the effect of overhead resulting due to the need to upload the input files, executing the analysis on isolated and virtualized container environment, and downloading the analysis results.

In Figure 6.4, it can be seen that the SAS analysis takes slightly less time (9.88%) than the local execution despite the overhead involved in the execution of analysis tasks with ATaaS. This difference can be attributed to ATaaS for SAS analysis having all the tool executable and dependencies inside the Docker container hosting the ATaaS while for local execution analysis tools are loaded from a shared network location.

In contrast, for FEA analysis, it can be seen that the execution of the analysis task using ATaaS takes more time (30.69% on average) than running the analysis using the locally installed tools. This confirms that substantial overhead, which is dominated by the file transfer, is added when the execution of analysis tasks is done on by using tool services hosted on distributed cloud infrastructure. However, if powerful computing resources are used to host the ATaaS and network access is reduced, the effect of the file transfer overhead can be reduced as seen in the case of SAS analysis.



Execution Time (Sec)	
Local	ATaaS
47.99	43.25

Execution Time (Sec)	
Local	ATaaS
1023.19	1476.19

Figure 6.4: Mean execution times for local vs one instance of ATaaS

6.3.2 RQ2: Effect of parallel distributed ATaaS

Figure 6.5 shows that the distributed parallel execution of analysis with cloud-based web services reduces the execution time for SAS analysis by 75.88% (on average) for 22 analysis tasks run by using 4 ATaaS instances running on two distributed computing resources as described in Section 6.2 when compared against the execution time obtained using locally run analysis tools (baseline).

In case of FEA analysis, it can be noted that, despite the 30.69% overhead involved in running the analysis task in ATaaS, a reduction of 7.49% in execution time is seen for 3 analysis tasks run by using 3 ATaaS instances running on two distributed computing resources as described in Section 6.2.2.

These comparisons show that using cloud-based web services providing the analysis tool functionalities to execute multiple analysis tasks in parallel can result in reduced execution times even though some file transfer and execution overhead is involved. Fur-

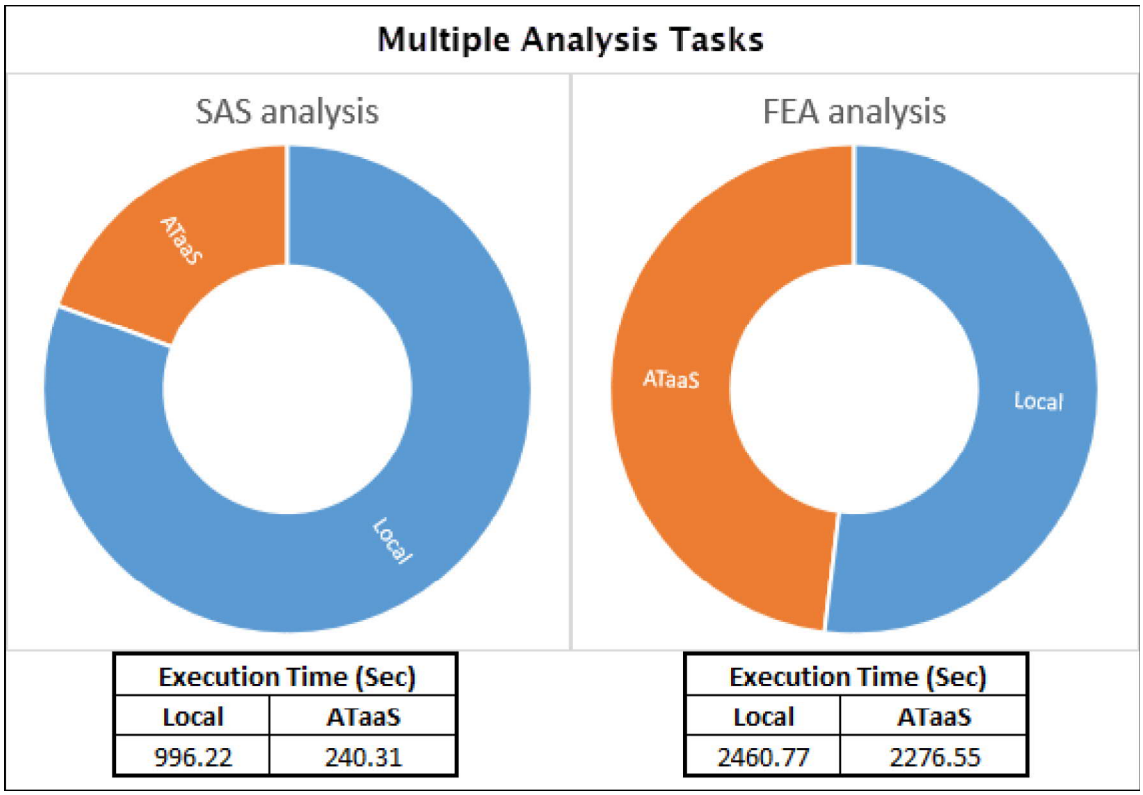


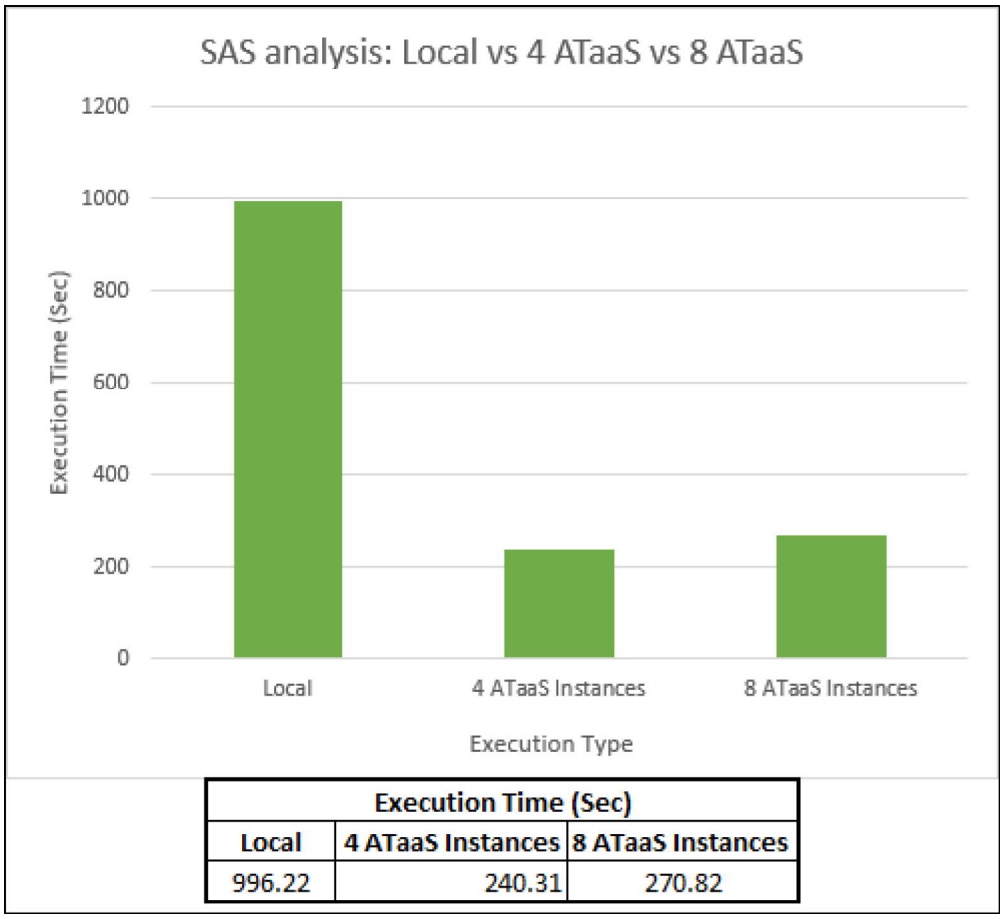
Figure 6.5: Execution times for multiple analysis tasks: local vs ATaaS

ther reduction in the execution time could be achieved by optimizing the network access and resource usage in the virtualized container environment which is out of scope for the current thesis.

6.3.3 RQ3: Effect of Increasing Service Replicas

Figure 6.6 shows a comparison of the execution times of 22 SAS analysis tasks executed using locally run analysis tools, 4 ATaaS instances (baseline), and 8 ATaaS instances. It can be seen that the parallel distributed execution of analysis tasks using 4 ATaaS instances reduced the execution time significantly when compared to local execution. But by doubling the number of ATaaS instances on the same hardware configuration increased the execution time by about 15% when compared to the execution time with 4 ATaaS instances. This shows that simply increasing the number of service instances on the same hardware configuration may not give any improvement in the performance as more ex-

ecution instances sharing the same hardware resources reduces the performance of individual service instances. Adding more computing nodes to run more service instances is required to decrease the execution time further.



Execution Time (Sec)		
Local	4 ATaaS Instances	8 ATaaS Instances
996.22	240.31	270.82

Figure 6.6: Execution times: Local vs 4 replicas vs 8 replicas of ATaaS

6.4 Summary

Our performance evaluation shows that using cloud-based web services to execute the analysis tasks adds file transfer and overhead. However, when multiple analysis tasks are executed simultaneously on the two or instances running on the distributed computing infrastructure, the total execution time is substantially reduced. The time saved is proportional to the number of service instances available for running the analyses. How-

ever, just increasing the number of service instances on the fixed computing infrastructure does not reduce the execution times due to the hardware limitations.

Chapter 7

Related Work

Cloud-based microservice architecture is the architecture of choice for many modern web applications in several domains, and hence numerous references to the related work can be found in the literature. In the following sections, we have identified related work more specific to tool integration, workflow execution as a service and analysis as a service from different domains as this has been the prime focus of this thesis.

7.1 Tool Integration for Design of Cyber-Physical Systems

Workflow-driven tool integration framework: Workflow-driven tool integration framework proposed in[3] uses service-oriented architecture shown in Figure 7.1 and it uses jBoss jBPM [29] workflow execution engine to orchestrate workflows. The framework uses business process workflows to chain two or more tools managed by a *Tool Manager*. Like the Siemens in-house tool integration framework, this framework uses *tool connectors* to invoke the tools corresponding to a business process step. In contrast, the cloud-based microservice architecture for tool integration framework proposed in this thesis, exposes tool functionalities as independent microservices that are accessible via REST APIs. In contrast to the architecture proposed in this thesis, the jBPM workflow-driven tool integration framework has a workflow editor (*Process Editor*), which allows users to define workflows.

jETI: The jETI [37], a framework for remote tool integration, is one of the early works in the area of providing tool integration services over the internet by using Simple Ob-

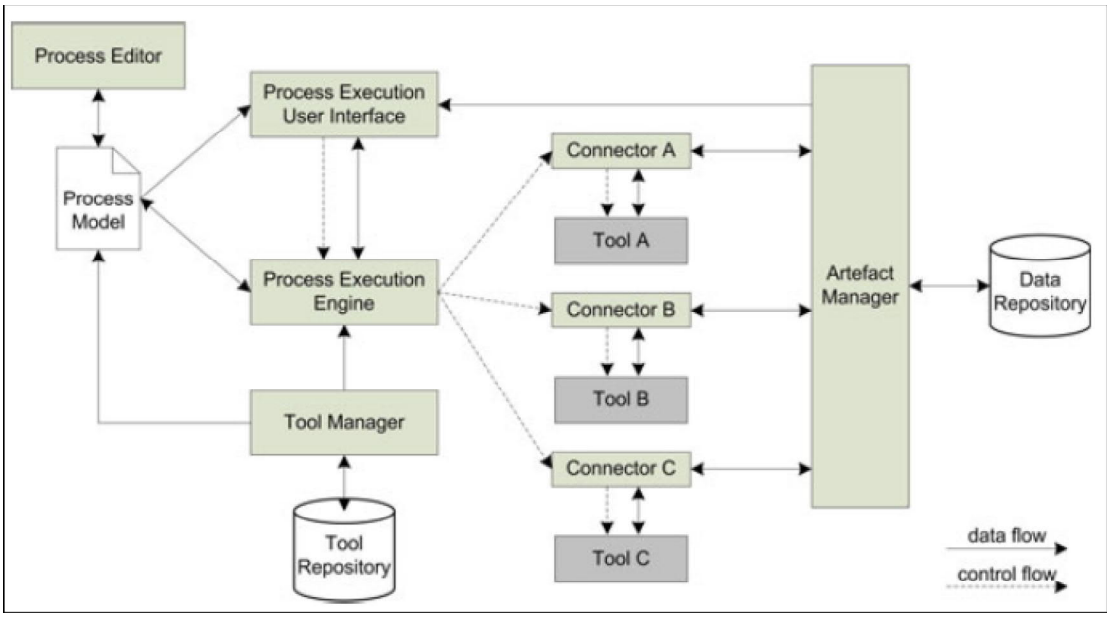


Figure 7.1: Architecture of workflow-based tool integration frameworks. Source [3]

ject Access Protocol (SOAP) [55] based web services. The SOAP-based web services are similar to RESTFul web services used in our work but support only XML data exchange between the client and the server. The jETI has a distributed tool library and uses the Tomcat Servlet container for distributed execution of tools. The cloud-based microservice architecture for tool integration and execution proposed in the thesis uses the most recent technologies and concepts of web services and tool containerization. It provides the benefits of modern web services such as on-demand scalability on distributed cloud infrastructure.

ModelBus: The ModelBus [22], a model-driven tool integration framework having a service-oriented architecture, uses BPEL for orchestration of web services that provide a tool functions as a service for the model-based development process. The ModelBus has a model repository that holds sharable models and unique tool functionalities hosted as web services on a distributed heterogeneous environment. As discussed in Section 3.3, the microservices architecture used in our work, was introduced in the server-

side software engineering domain to address development complexities associated with the monolithic applications and the SOA based web services.

Many research projects are being executed in collaboration between industries and educational institutions to address the tool integration and interoperability challenges faced by industries in designing modern day cyber-physical systems. The CONCERTO [43] is one such project which aims to develop a framework to bring the integration between tools used in various stages (such as development and verification) of the design process for complex cyber-physical systems by means of model-driven engineering. It has a multi-domain user modeling space which uses high-level system models (such as UML [1] models) to combine domain specific execution platforms through model transformations to get a seamless bidirectional integration between tools used at various design steps. OpenMETA [62] is another project which uses model-based approach to provide extensive integration framework consisting of model, tool and execution integration frameworks.

There are also standards such as Functional Mock-up Interface (FMI) [5] developed by Modelica Association to standardize the creation, storage, exchange and reuse of system models to simplify the integration between different tools used in the design of cyber-physical systems and facilitate co-simulation of systems by allowing tools to exchange models of subsystems. In complement to this, projects like openCPS [2] aim to develop frameworks to promote interoperability between model-driven engineering tools aligned with different standards such as Modelica, UML and FMI.

7.2 Workflow Execution as a Service

Business Process Execution Language for Web Services (WS-BPEL) is the de facto standard for the workflow execution by the orchestration of web services and applies well to the workflow execution as a service proposed in this thesis with minimal difference. The focus of the WS-BPEL lies in orchestration and execution of business process steps and

involves a small amount of data transfer while the AGT tool integration workflow steps often involve large amounts of data transfer. In this section, we present few works in the area of BPEL as a service as related work in the area of workflow execution as a service proposed in this thesis.

Orchestration as a Service: The Orchestration as a Service (OaaS) [23] provides a workflow design and management service and workflow execution services similar to the services provided by the workflow execution service proposed in this thesis. It uses a WS-BPEL compliant workflow execution engine to orchestrate the execution of workflow steps on a distributed cloud infrastructure. The OaaS is a web service built in SOA architecture and has the issues associated with a monolithic application, while the microservices architecture proposed for the workflow execution service in this thesis reduces the development complexity of the service. Besides, microservices provide many benefits as listed in the Section 3.3

On-Demand Resource Provisioning for BPEL Workflows Using Amazon's Elastic Compute Cloud: The authors of [10] presents a workflow execution framework based on BPEL to schedule the execution of workflow steps on distributed hybrid cloud infrastructure consisting of dedicated local computing resources and remote computing resources on the Amazon's Elastic Compute Cloud. The architecture presented in this reference is similar to the workflow execution architecture presented in our work but uses SOA type web services running on fixed nodes. As already discussed in the above paragraph, microservice-based web services have less development complexity than monolithic SOA web services and provide many other benefits.

Similar to the [10], [30] presents a cloud-based architecture for distributed execution of workflow steps using the BPEL engine and has a multi-objective scheduling algorithm that considers data dependencies between the workflow steps to optimize the execution time. We consider the scheduling algorithm presented in their work to be novel and wish

to consider such efficient scheduling algorithms in the design of our workflow execution service as future work.

7.3 Analysis Tool as a Service

Model-as-a-Service: Model-as-a-Service [59] proposes a cloud-based RESTful microservices for providing a web service to build system models for Finite Element Analysis (FEA) and Computational Fluid Dynamics (CFD) from component models. It uses a Federation Management System (FMS) to interact with a distributed solver subsystem to run analysis jobs, which is similar to using Docker containers for running solvers in our work. The proposed model-as-a-service runs the analysis jobs asynchronously using a message queue, which is similar to the use of the task queue in our implementation of the analysis tool as a service. The containers based web service increases the elasticity of the application and promote dynamic scaling of the application across distributed cloud (private, public and hybrid).

CloudMEMS: CloudMEMS [54] proposes a three-tier cloud-based web model-view-controller (MVC) architecture for running Finite Element Analysis (FEA) for Micro-Electro-Mechanical Systems (MEMS) using the Comsol Multiphysics [28]. It uses Java REST microservices running on the AWS to provide an interface between the users and Comsol tool. The architecture of CloudMEMS has a similarity with the ATaaS proposed in this thesis in the sense that both provide an asynchronous interface to the analysis tools used in multi-disciplinary design environments and use cloud-based microservices. CloudMEMS architecture is specifically targeted to run on AWS and has limited adaptability to other private and hybrid cloud infrastructure, unlike the architecture proposed in this thesis.

Big data and cloud computing platform for energy internet: The big data and cloud computing platform for energy internet [16] proposes a cloud-based RESTful microser-

vice application for data collection, analysis and visualization of energy consumption monitoring and complementary operation of the multi-form energy system and uses asynchronous background task processing to perform computationally intensive tasks such as big data processing. Though this work does not involve running design analysis such as FEA and CFD, running computationally intensive big data analysis tasks is similar to running the analysis tools discussed in the thesis and qualifies as related work in the context.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

8.1.1 Conclusions from architecture for tool integration framework

In this thesis we proposed a cloud-based microservices architecture for a workflow-based tool integration framework to address certain infrastructural and performance limitations of the Siemens in-house tool integration framework. The proposed architecture consists of independent microservices for the execution of workflows and analysis tools and they communicate through REST APIs. The tool integration framework along with the services providing the analysis tool functionalities can be hosted on service clusters running on private, public or hybrid cloud using Docker containerization platform to support easy deployment and on-demand scalability. Siemens engineers can easily access the workflow execution services provided by the tool integration framework using a lightweight front-end application or thin client applications such as a web browser.

Limitations: The development of the workflow execution services by reusing the components of the existing in-house tool integration framework involves huge efforts in separating tightly coupled monolithic components in to independent microservices communicating through REST APIs.

Since the proposed architecture aims to reuse the components of the existing tool integration framework, it inherits few limitations of the existing framework and they are listed below.

- Currently, the workflow definition is done outside the framework by using text editors and the proposed architecture does not address this limitation.
- Definition and execution of hierarchical (workflow inside workflow) workflows is not supported.
- Only one workflow can be executed at a given point time in the existing tool integration framework. The proposed cloud-based architecture does not explicitly address this limitation, although it should inherently address this limitation at implementation time by asynchronous request processing methods.

8.1.2 Conclusions of architecture for analysis tool services

We also proposed a cloud-based microservices architecture for providing Siemens AGT analysis tool functionalities as services hosted in isolated, replicated container environments accessible via the existing tool integration framework or by using the workflow execution services in the proposed cloud-based architecture for the tool integration framework. Analysis tool services can also be hosted on private, public or hybrid cloud infrastructure and can be accessed through REST APIs. Siemens engineers can run hundreds of analysis tasks in parallel on a distributed cloud computing resources by using hundreds of tool service instances when available to reduce the overall analysis time. Furthermore, computationally intensive analysis tools perform better on cloud computing resources than the local (on engineer's computer) execution of the same tools as optimal computing resources can be allocated to the tools on cloud.

Limitations: The distributed and parallel execution of analysis tasks with hundreds of tool service instances requires lots of computing resources. The service cluster may have low stability when voluntary computing resources are used as worker nodes as proposed in this thesis, because a voluntary node can go down at any time. The fault tolerance in the proposed architecture may be insufficient to manage the disappearance of the worker

nodes in the middle of running hundreds of analysis tasks in parallel. This is particularly more troublesome for the FEA analysis tasks that have long execution time as tasks that were running on the lost node needs to be restarted irrespective of its progress at the time of node loss. The stability of the service cluster can be high when all nodes in the cluster are located in the public cloud (AWS) but this could be very expensive as resources on the public cloud costs more than already available in-house resources.

8.1.3 Conclusions by performance evaluation

Furthermore, we developed tool services for two AGT analysis tools, namely, SAS and FEA analysis by using the proposed architecture. The performance evaluation of these tool services showed that by parallel distributed execution of analysis tasks using multiple instances of the tool services reduces the overall execution time although there is noticeable overhead. The overhead is due the upload and download of files, and execution of analysis tasks inside the constraints of Docker containers. We also observed that the execution overhead could be reduced by using more powerful computing resources. While the reduction in the execution time can be more with higher number of tool service instances, simply increasing the number of instances without adding new computing resources will not reduce the execution time.

Limitations: Local (on engineer's laptop) execution is more beneficial than using remote tool services to execute short, low-complexity analysis for just one or two configurations. Thus, availability of the tool services may not encourage the engineers to completely abandon the local execution setup and permanently change over to using the tool services. The effect of overhead is more severe when engineers are working remotely and it may force the engineers to abandon using the tool services and fallback to using local execution. Furthermore, engineers may be compelled to be physically present in office premises to use the tool services hosted using in-house computing resources as better performance is only guaranteed when directly connected to the Siemens network.

8.2 Future Work

Development of the workflow execution service: The development of the workflow execution service by reusing the components of the existing tool integration framework is a major future work in the development of the proposed cloud-based tool integration framework. The in-house tool integration framework is a monolithic application and has a tightly coupled workflow visualization front-end and workflow execution back-end. Atomic, independent microservices will not only decouple the workflow visualization front-end from the back-end, but also separate different concerns into multiple independent microservices communicating using REST APIs.

Deployment on public and hybrid cloud: In this thesis, we deployed the tool services on distributed computing infrastructure with in Siemens AGT which used only few computing resources. How these tool services perform on public clouds such as AWS and hybrid cloud combining the Siemens computing resources with AWS remains to be investigated in depth.

Integration with workflow definition service: The existing tool integration framework and the proposed architecture for the tool integration framework excludes the workflow definition service as the development of workflow definition service is carried out by Ms. Jasvir Dhaliwal, a fellow researcher in the same collaborative project with Siemens Canada. Integrating the workflow definition service with the proposed cloud-based tool integration framework gives a high performance end-to-end workflow based tool integration framework.

Bibliography

[1] OMG. *OMG® Unified Modeling Language® (OMG UML®)*. 2.5.1. OMG® Unified Modeling Language® Publication. 796 pp.

[2] openCPS. *Interoperability of the standards Modelica-UML-FMI*. 0.3. ITEA3, p. 93.

[3] András Balogh et al. "Workflow-Driven Tool Integration Using Model Transformations". In: *Graph Transformations and Model-Driven Engineering: Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday*. Ed. by Gregor Engels et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 224–248.

[4] Robin Singh Bhadoria et al. *Exploring Enterprise Service Bus in the Service-Oriented Architecture Paradigm*. Advances in Business Information Systems and Analytics (2327-3275). IGI Global, 2017.

[5] Torsten Blochwitz et al. "The Functional Mockup Interface for Tool independent Exchange of Simulation Models". In: Proceedings of the 8th International Modelica Conference, Mar. 2011, pp. 105–114.

[6] Lianping Chen. "Continuous Delivery: Huge Benefits, but Challenges Too". In: *IEEE Software* Vol. 32.No. 2 (Mar. 2015), pp. 50–54.

[7] Jeffrey Cloud. *AWS: The Ultimate Amazon Web Services Guide From Beginners to Advanced*. Independently Published, 2020.

[8] *Docker Container Overview and Docker Compose - XenonStack*. URL: https://www.xenonstack.com/blog/docker-container/ (visited on 01/29/2020).

[9] *Docker Swarm: How nodes work*. Docker Documentation. Mar. 3, 2020. URL: https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/ (visited on 03/04/2020).

[10] Tim Dörnemann, Ernst Juhnke, and Bernd Freisleben. "On-Demand Resource Provisioning for BPEL Workflows Using Amazon's Elastic Compute Cloud". In: *2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*. 2009, pp. 140–147.

[11] Paul .M. Duvall, Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Signature Series. Pearson Education, 2007.

[12] Floris Erich. "DevOps is Simply Interaction Between Development and Operations". In: *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*. Ed. by Jean-Michel Bruel, Manuel Mazzara, and Bertrand Meyer. Cham: Springer International Publishing, 2019, pp. 89–99.

[13] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. The Prentice Hall Service Technology Series from Thomas Erl Series. Pearson Education, Limited, 2016.

[14] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. Vol. 7. University of California, Irvine Doctoral dissertation, 2000.

[15] Paolo Di Francesco, Patricia Lago, and Ivano Malavolta. "Architecting with microservices: A systematic mapping study". In: *Journal of Systems and Software* Vol. 150 (2019), pp. 77 –97.

[16] Rui Fu et al. "Big data and cloud computing platform for energy Internet". In: *2017 China International Electrical and Energy Conference (CIEEC)*. 2017, pp. 681–686.

[17] Daniel Gaspar and Jack Stouffer. *Mastering Flask Web Development: Build enterprise-grade, scalable Python web applications, 2nd Edition*. Packt Publishing, 2018, pp. 182–190.

[18] JJ Geewax. *Google Cloud Platform in Action*. Manning Publications, 2018.

[19] Dimitrios Georgakopoulos and Michael Papazoglou. *Service-oriented Computing*. Cooperative information systems. MIT Press, 2009.

[20] *GitLab Continuous Integration & Delivery*. GitLab. URL: https://about.gitlab.com/product/continuous-integration/ (visited on 01/07/2020).

[21] Caleb Hattingh. *Using Asyncio in Python: Understanding Python's Asynchronous Programming Features*. O'Reilly Media, 2020.

[22] Christian Hein, Tom Ritter, and Michael Wagner. "Model-driven tool integration with modelbus". In: *Workshop Future Trends of Model-Driven Development*. 2009, pp. 50–52.

[23] André Höing et al. "An Orchestration as a Service Infrastructure Using Grid Technologies and WS-BPEL". In: *Service-Oriented Computing*. Ed. by Luciano Baresi, Chi-Hung Chi, and Jun Suzuki. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 301–315.

[24] *HTTP Response Status Codes REST API Tutorial*. URL: https://restfulapi.net/http-status-codes/ (visited on 04/15/2020).

[25] *IaaS vs PaaS vs SaaS: Which Should You Choose?* URL: https://www.datamation.com/cloud-computing/iaas-vs-paas-vs-saas-which-should-you-choose.html (visited on 01/29/2020).

[26] *Integrating CI/CD with Docker Enterprise Edition - Demo Webinar Recap - Docker Blog*. URL: https://www.docker.com/blog/ci-cd-with-docker-ee/ (visited on 01/29/2020).

[27] *Introducing Jenkins X: a CI/CD solution for modern cloud applications on Kubernetes*. URL: https://jenkins.io/blog/2018/03/19/introducing-jenkins-x/index.html (visited on 01/07/2020).

[28] *Introduction to COMSOL Multiphysics*. en. 2019. URL: https://cdn.comsol.com/doc/5.5/IntroductionToCOMSOLMultiphysics.pdf (visited on 03/08/2020).

[29] *jBPM Documentation*. URL: https://docs.jboss.org/jbpm/release/7.35.0.Final/jbpm-docs/html_single/ (visited on 04/15/2020).

[30] Ernst Juhnke et al. "Multi-objective Scheduling of BPEL Workflows in Geographically Distributed Clouds". In: *2011 IEEE 4th International Conference on Cloud Computing*. 2011, pp. 412–419.

[31] Nitin Kumar Karma. *RESTFul Web Services - the Python Flask Way: Build RESTful APIs Using Python and Flask-Restful*. Independently Published, 2018.

[32] Shijimol Ambi Karthikeyan. *Practical Microsoft Azure IaaS: Migrating and Building Scalable and Secure Cloud Solutions*. Apress, 2018.

[33] Jay Kreibich. *Redis: The Definitive Guide: Data modeling, caching, and messaging*. O'Reilly Media, Incorporated, 2013.

[34] Chandra Krintz. "Software-as-a-Service (SaaS)". In: *Encyclopedia of Database Systems*. Ed. by Ling Liu and M. Tamer Özsu. New York, NY: Springer New York, 2018, pp. 3544–3545.

[35] Paul J. Leach, Michael Mealling, and Rich Salz. *A Universally Unique IDentifier (UUID) URN Namespace*. en. Library Catalog: tools.ietf.org.

[36] Mika V. Mäntylä et al. "On rapid releases and software testing: a case study and a semi-systematic literature review". In: *Empirical Software Engineering* Vol. 20.No. 5 (2015), pp. 1384–1425.

[37] Tiziana Margaria, Ralf Nagel, and Bernhard Steffen. "jETI: A Tool for Remote Tool Integration". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Nicolas Halbwachs and Lenore D. Zuck. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 557–562.

[38] *Meet Travis CI: Open Source Continuous Integration*. InfoQ. URL: https://www.infoq.com/news/2013/02/travis-ci/ (visited on 01/07/2020).

[39] Peter Mell, Tim Grance, et al. "The NIST definition of cloud computing". In: *NIST Special Publication 800-145* (2011).

[40] F. Millstein. *DevOps Handbook: What Is DevOps, Why You Need It And How To Transform Your Business With DevOps Practices*. Frank Millstein, 2020.

[41] Rimantas Mocevicius. *CoreOS Essentials*. Packt Publishing, 2015. Chap. 9, pp. 91–94.

[42] A.J.A. Mom. "Introduction to gas turbines". In: *Modern Gas Turbine Systems*. Elsevier, 2013, pp. 3–20.

[43] Leonardo Montecchi, Paolo Lollini, and Andrea Bondavalli. "A Reusable Modular Toolchain for Automated Dependability Evaluation". In: VALUETOOLS 2013 - 7th International Conference on Performance Evaluation Methodologies and Tools, Jan. 2013.

[44] OAI. *OAI/OpenAPI-Specification*. Mar. 3, 2020. URL: https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.3.md.

[45] *OpenAPI Initiative, https://www.openapis.org/about*. OpenAPI Initiative. URL: https://www.openapis.org/about (visited on 03/04/2020).

[46] Rene Peinl, Florian Holzschuher, and Florian Pfitzer. "Docker Cluster Management for the Cloud - Survey Results and Own Solution". In: *Journal of Grid Computing* Vol. 14.No. 2 (June 2016), pp. 265–282.

[47] Francesco Pierfederici. *Distributed Computing with Python*. Packt Publishing, 2016.

[48] Carlos Pinheiro, André Vasconcelos, and Sergio Guerreiro. "Microservice Architecture from Enterprise Architecture Management Perspective". In: *Business Modeling and Software Design*. Ed. by Boris Shishkov. Cham: Springer International Publishing, 2019, pp. 236–245.

[49] Rolls-Royce Plc. *RTrent Third Party Packager - Engine Operation*.

[50] *Power up your business*. siemens.com Global Website. URL: https://new.siemens.com/global/en/products/energy/power-generation/gas-turbines.html (visited on 10/02/2019).

[51] Kunal Relan. *Building REST APIs with Flask: Create Python Web Services with MySQL*. Apress, 2019.

[52] Pilar Rodríguez et al. "Continuous deployment of software intensive products and services: A systematic mapping study". In: *Journal of Systems and Software* Vol. 123 (2017), pp. 263–291.

[53] Meinhard T. Schobeiri. "Introduction, Gas Turbines, Applications, Types". en. In: *Gas Turbine Design, Components and System Design Integration*. Cham: Springer International Publishing, 2018, pp. 1–30.

[54] Anil Sehgal. "CloudMEMS Platform for Design and Simulation of MEMS: Architecture, Coding, and Deployment". MA thesis. University of Toledo, 2018.

[55] James Snell, Doug Tidwell, and Pavel Kulchenko. *Programming Web Services with SOAP: Building Distributed Applications*. O'Reilly Media, 2001.

[56] Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. "The pains and gains of microservices: A Systematic grey literature review". In: *Journal of Systems and Software* Vol. 146 (2018), pp. 215 –232.

[57] Rahul Soni. *Nginx: From Beginner to Pro*. Apress, 2016, pp. 153–171.

[58] *Step 3: Parameters (API reference tutorial) | Documenting APIs*. URL: https://idratherbewriting.com/learnapidoc/docapis_doc_parameters.html (visited on 04/08/2020).

[59] Sunil Suram, Nordica A. MacCarty, and Kenneth M. Bryden. "Engineering design analysis utilizing a cloud platform". In: *Advances in Engineering Software* Vol. 115 (2018), pp. 374 –385.

[60] *Swarm mode key concepts*. Docker Documentation. Jan. 7, 2020. URL: https://docs.docker.com/engine/swarm/key-concepts/ (visited on 01/07/2020).

[61] Basarat Syed. *Beginning Node.js*. Apress, 2014.

[62] Janos Sztipanovits et al. "OpenMETA: A Model- and Component-Based Design Tool Chain for Cyber-Physical Systems". In: *From Programs to Systems. The Systems perspective in Computing*. Ed. by Saddek Bensalem, Yassine Lakhneck, and Axel Legay. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 235–248.

[63] Johannes Thönes. "Microservices". In: *IEEE Software* Vol. 32.No. 1 (2015), pp. 116–116.

[64] *UUID objects according to RFC 4122 Python 3.8.2 documentation*. URL: https://docs.python.org/3/library/uuid.html#uuid.uuid1 (visited on 04/08/2020).

[65] *Volunteer computing*. en. Page Version ID: 919041984. Oct. 2019. URL: https://en.wikipedia.org/w/index.php?title=Volunteer_computing&oldid=919041984 (visited on 04/08/2020).

[66] Anthony I. Wasserman. "Tool integration in software engineering environments". In: *Software Engineering Environments*. Ed. by Fred Long. Berlin, Heidelberg: Springer Berlin Heidelberg, 1990, pp. 137–149.

[67] *Windows Servercore Image*. URL: https://hub.docker.com/_/microsoft-windows-servercore.

[68] E. Wolff. *Microservices: Flexible Software Architecture*. Pearson Education, 2016.

[69] Vijay Yellepeddi. *@Scale - Part I (Task Queues)*. experience@imaginea. URL: https://blog.imaginea.com/scale-part-i-task-queues/ (visited on 02/19/2020).