Fachhochschule Dortmund
University of Applied Sciences and Arts

BOSCH

Design & Development of an Inter-Company Software Collaboration Platform Using S.O.A and RestAPIs

FROM MONOLITHIC ARCHITECTURE TO FLEXIBLE MICRO-SERVICES

By:

AHMAD HASSAN MIRZA

Matrikel-Nr: 7104716

First Supervisor: Prof. Dr.-Ing. Jörg Thiem

Second Supervisor: Dipl.-Ing. Martin Kisser

This dissertation is submitted to the Department of Computer Sciences, Dortmund University of Applied Sciences and Arts (Fachbereich Informatik, Fachhochschule Dortmund), In partial fulfilment of the requirements for the degree of Master of Engineering in Embedded Systems for Mechatronics.

Declaration of Authorship

I hereby declare that I have authored this thesis independently, all direct or indirect sources that have been used are acknowledged as references, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources. I am aware that the thesis in digital form can be examined for comparison of my work with existing sources. I also agree to the storage of the thesis in the institutional repository to enable comparison with future theses submitted. Further rights of reproduction and usage, however, are not granted here. The content or parts of this thesis have not been presented to any other examination authority and have not been published.

Author: Ahmad Hassan Mirza

Signature:  **Date:** 20.08.2020 **Place:** Stuttgart, Germany

Abstract

Software development and the way in which a project progresses from the early stages to its delivery to customers is an ever-evolving field. Faster time to delivery, processes which enable easier updates and integrations of new components and collaboration across teams and community are the most important points with any software development project. The current trends in IOT and cloud computing have allowed for revolutionizing the way software development and distribution is handled. The traditional means of developing and delivering huge software systems, appropriately termed as monolithic systems, is being replaced with providing specialized components or services accessible over the internet. This allows a software vendor to easily maintain and update the software with minimal interruptions to the client's operations. This approach also has a huge impact on the roll out times associated with the first delivery as well as any future updates.

This thesis aims to study the practicality of Service Oriented Architecture (SOA) and a well-designed Application Programming Interface (API) and how they can be successfully used at a large company for software sharing and collaboration. The literature review in the earlier sections of the thesis aim to study what software development strategies and processes can be used together to achieve this goal in the most efficient way. Followed by a review of some of the most common design patterns used in Service Oriented Architecture. A good and well-designed application programming interface (API) acts as the life force of any software component and even more so for cloud native applications and web services. A review of various techniques and architectures for developing efficient and user-friendly APIs is also done. With migrating to network-based solutions, the topic of application's safety and security, against malicious and unauthorized use, also takes center stage. For this reason, possible attacks, vulnerabilities and security schemes for cloud applications is also explored in the thesis.

To show the practicality of the various techniques reviewed in the thesis, a software platform for publishing software and infrastructure as service is developed using the various techniques, with several use cases showcasing different design and architectural styles. A legacy monolithic system has also been migrated to a web service based model

to show how adapting this new architecture style and proposed software development strategies can increase a company's throughput and efficiency, by enabling easier and simpler collaboration between teams across the globe, easier processes for continuous integration and delivery of software components and software sharing with in as well as outside the company boundaries.

Acknowledgements

I would like to express my sincerest gratitude to my supervisor at Fachhochschule Dortmund Prof. Dr.-Ing. Jörg Thiem for being a great mentor and for providing guidance and support throughout my time working with him.

I also wish to express my sincere appreciation to Mr. Dipl. -Ing. Martin Kisser for his guidance, ideas and all the technical discussions that helped me bring my thesis work to completion. I would also like to thank Robert Bosch GmbH for the support and for giving me the chance to take on this thesis topic.

Last but not least, I would like to thank my mother for her input and moral support throughout my studies without which none of this would have been possible.

Table of Contents

Table of Figures

List of Abbreviations

SaaS	Software-as-a-Service
SOA	Service Oriented Architecture
OOP	Object Oriented Programming
API	Application Programming Interface
REST	Representational state transfer
HTTP(s)	Hypertext Transfer Protocol (Secure)
FTP	File Transfer Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
GPU	Graphics Processing Unit
ECU	Electronic Control Unit
ML	Machine Learning
CI/CD	Continuous Integration / Continuous Deployment (or Delivery)
AWS	Amazon Web Services
CRUD	Create, Read, Update, Delete
HA	Hypermedia Architecture
HATEOAS	Hypermedia-as-the-Engine-of-Application-State
EDA	Event Driven Architecture
RPC	Remote Procedure Call
SOAP	Simple Object Access Protocol
DDoS	Distributed Denial-of-Service
WASC	Web Applications Security Consortium
XSS	Cross Site Scripting
TLS	Transport Layer Security
WAF	Web-Application-Firewall
OEM	Original Equipment Manufacturer
RB	Robert Bosch Gmbh
FDK	Feature Development Kit

Chapter 1. Introduction

Back in circa 2580-2560 BC, in the great Egyptian monarchy, the Pharaoh named Khufu commissioned the construction of a pyramid that would serve as a tomb to the great Pharaoh. This construction is believed to have lasted for over 20 years resulting in a product that stood to be the tallest man-made structure for over 3800 years and one of the oldest of the seven wonders of the ancient world. Now one might question here as to why I began with describing how great the Great Pyramid of Giza is. The answer is simple, the Egyptians used a large work force divided into a hierarchy, and theories suggest that these hierarchies were built depending on the skill set of the labor force. So, we could say that each group provided a specific service, which may have been chiseling or shaping the huge blocks that make up the pyramids, decorating the inside of the tombs or the very menial work of moving the stone blocks from one point to another.

One can assume that a message must have been passed from say the building manager to the stonemasons that he needs 100,000 blocks of stone of a certain dimension. At this point, the masons would prepare the stone block and once done, send back a delivery of the prepared blocks. If we think about it, we can see here a system that can be classified as service oriented, with different teams cooperating with each other to deliver a final product. This distributed hierarchy and teams providing different parts of the final product, when used effectively, formed up a very efficient system that bore a product that has stood the test of time.

When it comes to software development, we can see since the introduction of high-level languages that programming practices have been moving towards implementing efficient ways to make the software components more abstract and modularized. Going all the way back let’s start from structured programming (from which procedural programming paradigm was derived) allowed for code reuse, employing procedure calls to execute a sequence of code, which needed to be run multiple times in a code block. With the introduction of object-oriented programming, the concept of abstraction, inheritance and message passing between objects was more refined.

Object oriented programming software design allows for a diversity of software design architectures and pattern, which encourage distributed architecture and systems, for example the publisher-subscriber pattern[1]. This also has the added advantage for large multi-disciplinary teams working on solving a problem. As with OOP architecture, a problem can be divided into separate components each party or member of the team depending on the team structure can work on developing a specific component that addresses a specific part of the problem. At the completion of the project, the components developed from each of these teams can be integrated forming the end-product. The integration engineer in this case only needs to concern him-self with the interface of all the components, which in the technical jargon are termed as the Application Programming Interface (API) of the component. Once all these components have been integrated to form the product that addresses the original problem, which was divided into smaller problems, In the case of OOP, the components and the API is limited and local to the software package. This huge software containing all the source-code, or the executable-files from individual components packed into one giant module conforms to what is known as the monolithic architecture.

A monolithic application developed using the principles of OOP, the objects or services still need to call each other and pass messages. This is usually achieved through language level method calls. A class defines the operations that in a micro-service architecture will be performed by a service. The different objects of different classes come together to create the entire logic of the application.

Recent years have seen a dramatic evolution in the practices related to software development. A growing focus on the notions of modularity, distribution, elasticity and scalability has driven the field towards the emergence of micro-services and Service Oriented Architecture (SOA) with many big names in the tech industry adapting the design

[1] A messaging pattern, where the component publishing the message does not dispatch the message directly to a component, instead categorizes the message into topics. The subscribers on the other hand get the messages directly by subscribing to topics without any knowledge of the working of the publishing component.

approach including software vendors like IBM and Microsoft as well as the content providers such as Amazon and Netflix etc [1].

An architecture based on micro-services is a cloud-native architecture at its heart that attempts to effectuate a software system as a package of small services. Each singular service in this package is designed and developed as an independent entity and has the possibility of being deployable on a multitude of technological stacks and platforms. This allows the system to be scalable and removes any underlying dependencies on hardware and software architectures allowing for more flexibility in system design at the client's side. In this approach, each service runs its own processes, while in a bigger picture the services communicate with each other only through their external interface adding a layer of abstraction. In contrast to the earlier monolithic applications where components interacted with each other using language level method calls, the interface of these services can be exposed using a lightweight mechanism like RESTful APIs and HTTP(s).

With this approach, a client has the possibility to create an application based on their own business logic using the functionality provided by the APIs without having to concern themselves with software, platform and hardware dependencies of the individual services or the back-end processes. This allows the developers of these services to continue working and updating their software without effecting the clients. So long as the interface of a service remains constant, the customer need not be concerned about the processes running behind the interface. This in turn allows for smooth and seamless software update process as well.

At Robert Bosch GmbH, several internationally located teams work together to develop components for any project that is currently underway. When requirements from a customer are received and a deliverable is being prepared as per the customer specifications, an engineer responsible for integrating all the different tools has to interact and liaise with all the engineers developing the components to get knowledge about the component's dependencies, its interface and its mechanics, as this knowledge is required for proper integration of all the individual component's binaries into a single deliverable package for the customer project.

This process requires a lot of man-hours as bundling up all the components into the final package is a complicated process and requires the integration engineer to implement thorough error handling and wrappers around the components. The inclusion of binaries for each component in the final package also increases the size of the deliverable, this in addition to having to install the dependencies for each component on the server, leads to increased storage requirements further increasing the costs.

This thesis aims to develop a model for simplifying inter-company software cooperation to allow easier integration of new software components in continuous development systems. The build system for the base software of embedded control units for various powertrain purposes is taken as the basis for the transformation from a monolithic software to a system with service-oriented architecture. The teams are distributed worldwide and collaborate on projects, which adds to the complexity involved in managing and developing software projects. It is expected that the parties involved in such projects will continue to grow and get more distributed around the word as the company continues to grow internationally. This complexity and distributed setup continue to make the job of integrating all the components coming from different teams and customers into a single deliverable harder. The goal is to make the software development process for customers more efficient even though the complexity involved continues to increase making collaboration, continuous integration and delivery easier and faster.

To support an efficient, secure and independent model for intercompany software cooperation novel tools and methods need to be incorporated in the development process. The use cases that are considered in the thesis are multi-party software integration, remote continuous testing and integration & support of artificial intelligence services over the cloud. The proposed solution is to have a platform running on a cloud server, where the developers of a component can directly deploy their solution as cloud native applications and expose the interface (API) of the component via URIs (uniform resource identifiers) that directly map the API. The idea is that in the future when a project wants to incorporate this component into their system, it only needs to know the API of the component and can access the functionality via simple API calls. This abstracts the functionality of the software components and so long as the API is kept constant, it also allows the developers

to update and evolve their software behind the scenes without having to interrupt the work of the clients.

For the Machine Learning applications and other such applications which require a lot of initial setup, as well as require a lot of hardware resources such as high-performance GPUs and CPUs which are expensive and might not be widely available, this solution provides the perfect setup. By abstracting the functionality and segregating it to a server, all the necessary setup is only needed to be done once at deployment time on the server. Also, the hardware requirements are only needed to be met at the server side machines. The clients of the application can be primitive devices such as ECUs or raspberry Pi which do not have enough compute power necessary for running ML algorithms. Only equipped with the knowledge of the API of the software components built for ML applications the remote clients can initialize the required workflow on the server, which after doing all the heavy lifting simply returns the results to the client.

The distributed nature of the proposed setup allows for clients with lower performance hardware to perform tasks that would not have been possible without significant hardware upgrade. It also reduces the costs associated with the initial setup required for complex systems, such as installation of the dependencies and preparing the workspace. The task of integrating software components being developed by distributed teams is much easier and subsequent updates require no work at all on the project team's side, further reducing the hours required for setting up the project.

In the thesis, I aim to develop the best strategies that could be adapted for implementing the proposed system for an efficient software cooperation in an international setting. Starting out with a review of the software development strategies that are available and which strategy best suits the programming paradigm that we try to solve. This is followed by the architecture patterns that are used for developing micro services. This is important because with a system being developed by distributed teams, it is very important that standardized design architectures and practices are followed by the teams to keep a consistent architecture throughout the organization. This helps for easier flow and transfer of knowledge throughout the company.

The design techniques and practices for developing APIs are explored in the next chapter. Robust, clear and concise APIs are very important in a system that uses micro-services. If the APIs are not clear, easy to use and provide the functionality that is expected of them, the simplicity that this system offers becomes redundant. An overview of the security threats is also given, that a system incorporating web and cloud native applications could be exposed to and what measures could be taken to make the system more secure against external threats. Finally, a demo proof of concept application by the name of HyAPI is developed which show cases the proposed system and the use cases discussed therein. The final chapters provide an overview of how the application is designed and implemented along with a user and developer documentation.

Chapter 2. Software Development Strategies

A software development strategy or process aims towards dividing the work into different phases, this facilitates the overall design and development process as well as helps make the project management tasks easier. A life cycle model usually is at the core of any software development process that may be adapted by a team or a company. In the current times, most of the companies adapt Agile methodologies for their software development needs to enable faster time to market and shorter lifecycles. Understanding these processes and how they are implemented is crucial part towards choosing how to approach any software development problem. In this section a brief overview of some core and important software development processes is reviewed.

2.1 Software Development Lifecycle

Geoffrey Elliott in his book summarizes the idea behind SDLC as "to pursue the development of Information systems in a very deliberate, structured and methodical way, requiring each stage of the life cycle- from inception of the idea to the delivery of the final system – to be carried out rigidly and sequentially" [2]. Building on this definition, SDLC consists of very clearly defined stages. The developers and project managers can use these

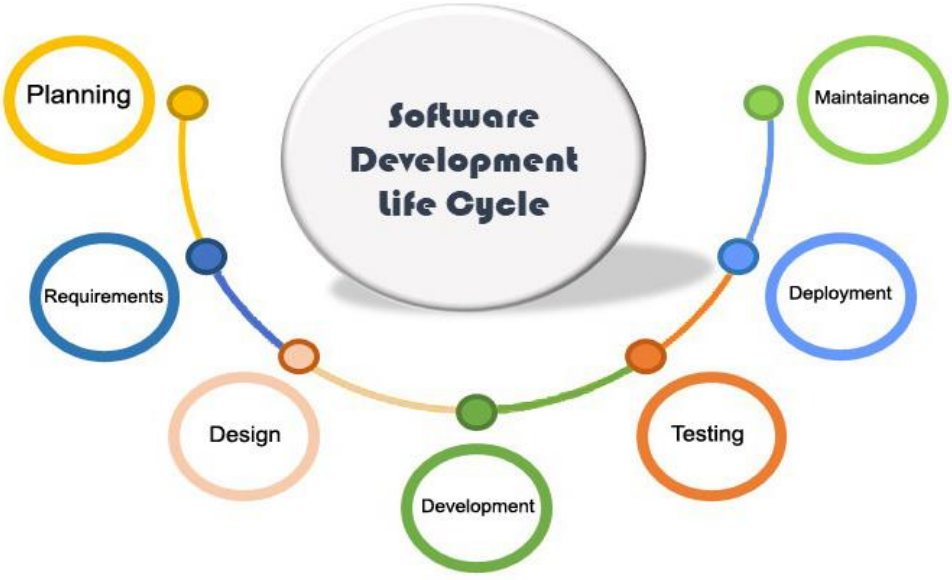

Figure 1 Software Development Lifecycle

to develop plans for rolling out a complete business system. As is the case in goods or hardware manufacturing the product moves down the assembly line with parts added to it on each stage of the manufacturing process with the final aim of delivering a product that meets the requirements and expectations of the customer. SDLC can be seen as an assembly line for software products.

Figure 1 shows the basic software development life cycle. Several models exist that can be applied to SDLC as suitable for the company's needs. The more well-known of these models are "Water-fall Model" and the "V-Model". Although the stages remain the same in both models, the difference is how the perspective is presented. In the waterfall model, the testing activities are carried out after the development is over, but in the V-Model Approach, test activities start with the first stage, albeit these initial testing stages are design and development of test strategies and test cases. In either case the actual testing is done once the development is done. Due to this pre-emptive approach for testing activities in the V-Model, the defects in the end-product after the development phase could be significantly less as compared to a software made using Waterfall model. In short, the V-Model is a simultaneous process while the waterfall model can be classified as a continuous process. The basic SDLC constitutes the stages shown in the figure [3].

2.1.1.1 Planning

The first phase of any software development project is planning, in this part, the concept of the product and the requirements are discussed and acquired from the client. The time, resources and budget allocation can then also be completed in this phase.

2.1.1.2 Requirements Analysis

In the second phase of the SDLC, meeting with all the stake holders are held to gather requirements. Requirements gathering stage is crucial to the further development of the project and all the questions ranging from the very basic e.g. Who is the target user base? To what kind of user inputs should be allowed and What database backend is required etc. should be addressed in this phase. Once the requirements gathering is complete, the requirements are analyzed for their practicality and validity. As a consequent of this stage, a requirements specification sheet is produced which can be used as a reference throughout the project, and this is also used by the testing team for their test planning.

2.1.1.3 Design

In the design phase, using the requirements specification document, system and software design is performed. The phase can usually be divided into two parts for more complex systems, namely, High-Level Design (HLD) and Low-Level Design (LLD) [2]. HLD deals with system architecture, database design, component design and deployment design. This is where UML and other modelling tools may be used to develop flow charts, state diagrams, sequence diagrams etc. In the LLD technical details of the logic and implementation of each individual component defined in HDL is done [4]. A systems design specifications document is generated at the end of this phase. The document serves as the guideline for the next stage as well as for the test team to develop the test cases and testing strategy.

2.1.1.4 Development

Once the system design specifications are finalized and the requirements have been locked down, the development phase can begin. In this phase the work is divided up among teams and they can start writing the actual code for the software using the process and guidelines adapted by the company or each individual team depending on their preferences. This phase usually takes up the longest time in any SDLC.

2.1.1.5 Testing

In this phase the developed software is rigorously tested against the requirements and design specification to infer if the code does what was required from it. This stage itself has a lot of processes like, unit testing, functional testing, integration testing etc.

2.1.1.6 Deployment

Once the testing phase has been successfully completed the product is shipped to the client or deployed as per the requirements. The clients can then perform initial testing at their end before the final deployment e.g. where the software is made available to the users.

[2] Differences between HLD and LLD

2.1.1.7 Maintenance

After the final deployment and the software is made available to the user-base periodic maintenance may be required as problems arise or any bugs that were not caught during the testing phases are discovered. This phase lasts until the end of the software's life cycle.

2.2 Agile

Back in the starting days of software development, programming was mostly done in an impromptu manner, i.e. as and when needed. Not much though went into planning and processes. In the 80's a growth in the complexity of software systems and application domains, prompted development of methods and practices that helped in the design and development of such systems. These complex processes emphasize engineering-based approach, encouraging extensive planning, processes set in stone which was seen by many to be effectively removing the human side of software development from the process [5].

In the late 90's agile methods started to emerge which placed the value on competent human resources, collaboration and flexibility promising high quality results in the end. Agile is basically a set of practices for managing the development process, it places great emphasis on frequent communication, short iterations and software increments which lead to frequent deliverables and a very active customer engagement.

Since its emergence Agile has become the most widespread software development method. According to a survey conducted in 2005, 14% of the companies were using agile methods and 49% were interested in using these methods in the future [6]. In contrast to these numbers from 2005, another study conducted in 2018 which reported 97% of the participating organizations have adapted agile software development practices [5].

There are several frameworks within Agile. SCRUM being the most popular one with a 56% share when used as a standalone framework and 70% when it is used in conjunction with other techniques followed by XP and Kanban at 23% and 5% respectively [5].

2.2.1 SCRUM methodology

SCRUM is a sub-framework of AGILE that further tries to optimize the development process. SCRUM defines roles, artifacts and events that make it unique from other AGILE

processes. G. K. Hanssen et.al. in their book [7] describe the various components of SCRUM as summarized below.

2.2.1.1 Scrum Artifacts:

Product Backlog is a storage for all the tasks that are needed to be done for a product. In simpler words the requirements specifications of the project. The product backlog is usually created before the first sprint and is updated as the project moves further. On the next level comes **Epic** which is a description of what is required by a client without much details, the Epic is further broken down into **User Stories** which provide the details about the required functionality, its priority and cost estimate.

Sprint Backlog contains the stories that are planned to be implemented in the upcoming sprint. These stories are moved from the product backlog to sprint backlog.

2.2.1.2 Scrum Roles:

A **Scrum Master** organizes sprint planning meetings, review meetings and other such events to keep the Scrum process running as smoothly as possible. The **Product Owner** (PO) is basically the customer or the liaison to the customer and represents all the stakeholders in the project. The PO is also responsible for managing the product backlog and the sprint backlog and its contents. Last but not the least is the **Development Team**, which is the group of engineers and other technical roles that are responsible for producing the code and tests.

2.2.1.3 Scrum Events

There are several events associated with scrum that should be carried out periodically. The foremost being a **sprint** which is a short development period usually between 2-4 weeks, where a pre-selected set of stories is implemented incrementing the functionality in the software. A **sprint review** is usually held once the sprint is over and the development team and stakeholders can review the software increment implemented in the previous sprint. In contrast a **sprint planning** meeting is held before a sprint to plan out and discuss the tasks associated with a sprint, and a **daily scrum** meeting is usually a short daily meeting where the developers can discuss the goals for the day and the achievements for the previous day.

2.2.2 Extreme Programming (XP)

XP is another popular agile software development process. The main goal of this approach is to provide a high-quality software according to dynamic requirements within the deadlines, or as the summarized by the experts, deliver what is needed when it is needed [8]. In many software systems the requirements are dynamic for example in situation when the customers do not have the exact idea of the end system and develop it as the project progresses, or in software systems where the requirements can change overtime and the software is expected to evolve every few months or years. The XP framework was designed to address such systems and is believed to handle projects with dynamic requirements better than other methodologies.

XP framework encourages small teams of between 2 to 12 members. The teams are not limited to just software developers and testers, but rather include all the stakeholders involved ranging from project managers to customers. All the members of the team are expected to collaborate not only towards software development but also towards developing functional tests. XP puts a great amount of emphasis on testability of the software system, specifically automated unit and functional tests.

The XP framework because of how it is setup, also address and mitigates the risks involved with delivering dynamic software systems. Whether the software being developed is a new challenge for the team or altogether a new concept, with a tightly knit team of developers, testers, managers and customers working together towards developing a stable software which conforms to the requirements as well as has a high testing throughput lowers the risks involved with rolling out the new software at the time of the release by ensuring a quality product.

2.2.3 Kanban Methodology

Kanban is a management tool that can be used to visualize the stage of a project and the workflow. It can be applied in conjunction with any software development framework like scrum or XP. Kanban does not specify any roles for the team, the products are delivered continuously as the need arises as opposed to pre-planned user stories in both scrum and XP. Kanban framework limits each member of the team to one task enforcing a “pull” method [9]. In contrast to scrum where changes to requirements/user stories once the sprint

has started are not allowed, Kanban allows for changes to be made mid-project which can lead to iterations and improvements being made prior to the completion of a sprint.

2.3 DevOps

DevOps brings the two groups of software development (Dev) and IT operations (Ops) together, to enable fast and flexible software development processes, end-to-end automation in both development and delivery processes is at the core of DevOps. The main aim of adapting DevOps practices is to cut down the software development life cycle; this is achieved by having cross-functional teams working simultaneously to make continuous deliveries as compared to companies using traditional software development processes. With small rollout times for updates and new products the organizations can compete better in the market as well as server their customers more effectively [10].

So far as we have seen in previous sections, the frameworks encourage collaboration between different departments that are involved with the product development e.g. the stakeholders, developers, testers and managers. But the IT or operations department is not included in this mix. They are the ones that handle the deployment and delivery of the developed solutions. When it comes to DevOps, a shift towards collaboration between the development teams, test teams and the operations teams is required.

In adapting DevOps, companies and teams set up continuous development, integration and deployment systems that tend towards smaller sprints and software increments. The teams can still apply agile frameworks within the scope of their work, so DevOps basically forms a blanket around the agile framework. A major challenge when adopting DevOps practices is for safety critical systems and embedded software. As in such system legacy code is usually involved and combining that with CI/CD with short increments can be problematic [11].

To achieve short delivery times, the teams must implement high level of automation in their processes, so choosing and implementing appropriate tooling and technology stacks is imperative to effectively applying DevOps practices. The tools can help engineers deploy their code for new feature or updates and even provision new infrastructure e.g. server,

computing power or remote storage independently, which in traditional setups would require help from 3rd parties and usually increased the time to deployment by a large factor.

There are several practices that come with DevOps the following sections provide a brief overview of these practices.

2.3.1 Continuous-Integration

Continuous Integration or CI for short is a software development practice as well as process. The software developers are expected to merge their code changes into a shared central repository. Automated build process and tests are performed on these changes to verify the functionality of these code changes. As CI is an integral part of DevOps automated build tools are also an integral part of DevOps by association. Automated build tools help reduce manual and time-consuming tasks helping achieve faster iterations. As the code from all the developers is merged and tested periodically, this practice helps in improving overall software quality by allowing bugs to be caught early in the development process. This in turn also helps in faster testing cycle and time to market.

As stated, earlier CI is both a process and a practice, the developers working with CI must learn and make it second nature to commit their code frequently for CI to work. Several tools like version control and automation servers can be used to implement the CI operation e.g. git, Jenkins and AWS Code Pipeline.

CI helps increase a team's productivity, by automating the laborious, time consuming manual tasks. Secondly frequent automated tests on code merges help reduce the bugs that may be present during the testing cycle or even that could have been released with the end-product. Earlier discovery of bugs also reduces the chances of a bug having a butterfly effect and turning into a bigger problem as the code base evolves. Last but not the least CI helps achieve faster roll out times for updates by reducing the development life cycle considerably.

2.3.2 Continuous-Delivery & Deployment

The terms Continuous Delivery and Continuous Deployment can be seen as two subsets of the last stage. Both processes are abbreviated as CD. In Continuous Delivery code changes are automatically prepared for release to a production environment such that when a

developer commits his code (in the CI stage) the changes are integrated into the software, built and tested by automated processes. This stage ensures that as soon as a code change is committed, a production ready build artifact is available after having passed pre-defined standardized tests. The Operations team can then perform more tests if needed and make the decision to deploy this artifact to the production environments.

For a properly implemented CD good automated testing coverage is important, and these automated tests could range from UI, integration, functional and API tests. This process addresses the issue of poor progress visibility and communication between the development team and the business teams [12].

Continuous Deployment takes CD one step further by automating the deploy to production process as well. In this case as soon as the changes are committed by a developer they go through the necessary steps of building and testing, and as soon as the final artifact is ready, it is pushed to production. This automation step aims to decrease the load from the Operations teams.

With CD the benefits of CI are carried even further. With automated build, test and deployment processes, the software delivery is more efficient and faster. With the removal of all manual tasks associated with these processes, the developers can focus more on writing code that could eventually help reduce the number of bugs and as the testing tasks are also automated the test team can instead focus on writing automated tests for better test coverage. With continuous deployment, the load is also taken off from the ops team and they can focus on improving the infrastructure and other operations services. With the CD the production ready version of the software built with the latest code and testing is always available for release, which helps make the software development process more efficient in both time consumption and cost.

2.3.3 Proactive Monitoring and Continuous Testing

The DevOps process relies heavily on effective testing practices and monitoring of the deployed solution which can be automated as well as in the form of customer feedback. This forms a feedback loop, which can be used to plan for future sprints and iterations. Once the solution has been deployed to production after automated testing, the feedback

from it can point to bugs that might have seeped through the testing procedures, as well as new features can be planned according to the feedback received from the customers.

Other than this, monitoring tools can help identify any bottlenecks in the IT infrastructure before they start effecting the customers. Monitoring tools can keep an eye on the system's health e.g. CPU usage, memory allocation and network statistics and administrators or the Ops team can be alerted when some problem is detected so that corrective measures can be taken in due time, resulting in a stable and rugged system.

2.3.4 Micro-Service Architecture

Micro-service architecture breaks down huge monolithic software systems into smaller components with the idea that complex applications can be fragmented, and the required services can be made available on demand. Micro-Service Architecture is discussed more in detail in the later chapters. DevOps provides an effective ecosystem for employing micro-service architecture and benefits can be sowed by using the two in conjunction although DevOps is not dependent on this architecture or vice versa, but micro services bring additional productivity to DevOps, and these two have been popularized and adapted in large companies like Amazon, Google and Netflix among others.

2.4 Lean software development

Lean software development (LSD) and Agile have a lot of common ground. Toyota Motor Corporation invented LSD in the mid-20th century as a method to optimize assembly lines, minimize waste and maximize customer value. Due to the fact it was originally known as Toyota Production System. The idea was to build the parts that are needed instead of manufacturing surplus and investing in maintaining an inventory, instead Toyota would build small batches of components as an when required for assembly and shipment to dealers. The core concept from Lean Production that is also used in LSD was to "pull" the components from the production lines as required instead of "pushing" the components and material from stock or inventories [13].

2.4.1 Principles of Lean

The application of Lean principles in software development was mainly popularized in the book “Lean Software Development” [14] which defines the principles and tools associated with LSD. In the book the authors define the seven principles of LSD as:

- **Eliminate waste:** As stated earlier eliminating waste by for example reducing inventories and using “pull” instead of “push” is a major component of LSD, multiple sources of waste may exist in software development ad are elaborated in a later section.
- **Amplify learning:** Learning and knowledge sharing is a huge part f LSD, this can be achieved by frequent deliveries and getting feedback from the customer on these deliveries to continue to improve the product. This approach reduces the chance of developing features that might be of no value to customers.
- **Decide as late as possible:** LSD encourages to explore all the options for items that may be complicated and expensive to implement and change in a later stage e.g. architecture, programming language etc. The final decision on these topics should be taken at the latest stage possible, this will allow the stakeholders to be able to choose the best available option from multiple options explored in the context of this approach.
- **Deliver as fast as possible:** Like Agile, Lean also encourages very frequent releases with small software increments. Allowing for a faster time to market but also necessitating through automated testing to avoid regression bugs and errors.
- **Empower the team:** LSD encourages teamwork and allowing for each individual to make decisions within their domain.
- **Build integrity in:** Continuous integration of small units of software into a larger system to realize the whole functionality, this process is also known as top-down programming where modules are integrated into the system as they are developed instead of at the end of a project.
- **See the whole:** The whole process of product development should be seen as one instead of treating software development process as a singleton i.e. system design,

software development and deployment among other processes should be seen as one whole part of the project.

The principles are further divided into tools accumulating to a total of 22 tools.

2.4.2 Software development wastes

The wastes associated with software development add a huge overhead to the cost of a project. As described in the preceding section Lean methodology encourages to minimize these wastes. This section gives a brief overview about what may be considered as a waste, so that a better understanding of optimizing processes using lean methodology can be formulated [14].

- **Extra Processes**, sometimes organizations spend too much time on meeting and generating paperwork for a project. This generally just slows down the process and consumes additional resources on top that may have been spent towards something productive. Steps to reduce unnecessary paperwork, meetings and processes should be taken to reduce software development wastes.
- Apart from extra processes, **waiting** for things to move along also adds a huge overhead on the project. This can be caused by delays in starting a project, staffing decisions, excessing documentation at the start of the project etc.
- **Partially Done Work**, Minimizing partially done software development is a risk-reduction as well as a waste-reduction strategy
- **Extra features**, building further on the above point, putting extra features, that may not be required at the time, in to the system is a waste, as resources must be spent in compiling, building, deploying, maintaining and testing of the code. Steps should be taken to minimize spending resources on features that don't add value.
- **Task switching and motion,** can cause an engineer to lose focus on the task at hand and then take significant time to get back in the flow of working and gather their thoughts. The best way to deal with this is to approach each task one at a time and to organize the teams so that the team members are easily accessible to each other without having to move around too much

- **Defects,** that may go undetected for longer periods of time can be a huge source of waste and incur overhead costs on a project, so to rectify this, fast testing, integration and delivery methods should be in place.

Chapter 3. Service-Oriented Architecture

In today's fast paced demographics, the time to market for new products and update rollouts can mean the success or failure of a product. In the world of monolithic software, which consisted of one big chunk of software that added up to the full functionality of the software and individual components could not be executed separately, delivering fast and periodic updates could be a hassle to achieve and maintain in the long run. In a monolith, updating one component could have regressive effect on all the other components, which if not tested in detail could result in show stopping bugs, which could cause a loss in market-share and user base. Doing regression testing on a monolithic system is also very time consuming and expensive and adds to the time to deploying an update or rollout of a new feature.

Service oriented computing builds on the principles of distributed and cloud computing. It breaks down a system into a collection of independent services. A service is an application with a sole responsibility, and can be developed, scaled, tested and deployed independently without having any effect on the functionality of other services. A large software system can then be developed by integrating these services to provide a seamless experience. In such a granular system, these services communicate with each other by passing messages via APIs. As an example, in a RESTful web-service, lightweight http messages can be passed between the services to enable communication. The adaptation of this architecture and development methodology can allow organizations to achieve better time to market by enabling faster and continuous deliveries and updates.

S.O.A. allows services to communicate across platforms and come together as one to form an application. For applications developed in this architectural style, business processes implemented as services are available for access via a pre-define application programming interface (API). For web-services, this API can be exposed via URLs defined as RESTful endpoints. In other implementation model these services are integrated together to form an

application using service orchestration.[3] In this chapter, I will explore and discuss some design patterns that are commonly used for the design and development of systems using service-oriented architecture.

3.1 Micro-Service Design Patterns:

Architectural design patters for micro services are still not standardized and a plethora of such patterns could be found in the literature for both academia and industry. Despite that, a select few of these design patterns are being used frequently in industrial and open-source projects [11] [12]. Most of the design patterns are designed to address one of the following quality attributes:

- Scalability
- Flexibility
- Performance
- Autonomy & Failure isolation
- Testability
- Continuous delivery (DevOps)

In this section, I will focus on the design patterns that take up the limelight and are more frequently used and are relevant to the use-cases implemented in this thesis.

3.1.1 Micro-service Architecture Pattern

To start the discussion about design patterns, it is only apt to discuss the pattern in focus here that is the micro services architecture pattern [15].

3.1.1.1 Problem

The development team gets the requirement to design and develop a video streaming application that can support multiple platforms and clients for example windows, Linux, web-browsers and mobile applications. The application is also required to expose APIs that can be consumed by third party developers and applications. Some requirements for the

[3] Service orchestration is the process involved in designing, creating, and delivering end-to-end services. [50]

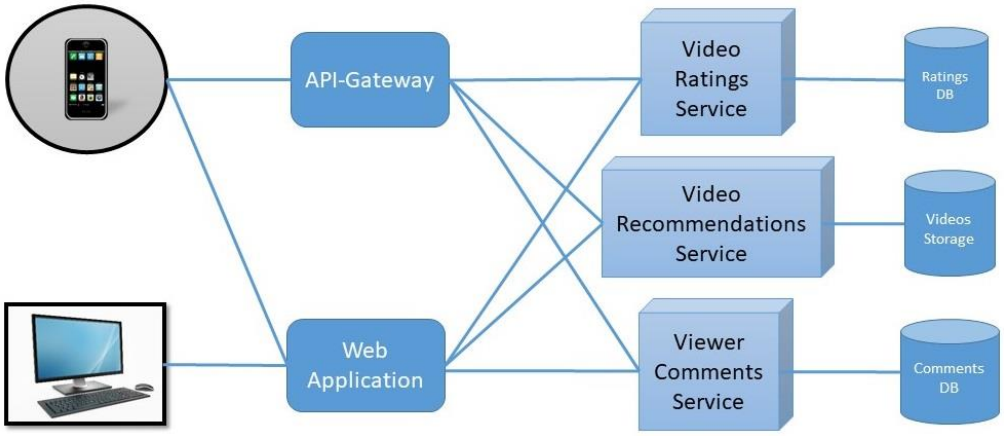

Figure 2 Layout of a typical application built using service-oriented architecture.

application are setup of a Continuous Integration & Continuous delivery (CI/CD) pipeline. Frequent update rollouts are expected to keep the application in sync with new technologies.

3.1.1.2 Solution

Structure the application as a set of loosely coupled services which communicate with each other via message passing using a lightweight protocol such as HTTP and using JSON/XML for returning responses[4]. Different functional/business areas of the application should be divided into different services; this will enable different teams or team members to work independently on a specific service without effecting other services or the changes committed to other services effecting their work [16].

3.1.1.3 Outcome

As the logic and the code for the execution of each service is hidden and only APIs expose the service's interface, this essentially makes the application platform independent. The client can call the API from their platform of choice (desktop app, browser or mobile app) and get the results back via the http response.

As each service in independent of the other, each team can update and deploy their work independently. This allows for faster update rollouts and time to market. Dividing the

[4] Netflix is one of the popular tech companies that user micro-service architecture and several of the patterns described in this section. [46]

application into smaller services also allows for easier maintainability and testing, and fault isolation.

3.1.2 API-Gateway Pattern

An API is a software interface that allows software components to communicate with the outside world. Usually an application contains a huge number of such APIs each exposing a specific functionality. To manage all the APIs additional infrastructure is needed. This is where API-gateway comes into play. An API-Gateway is the programming that is set up in front of an API and acts as a single-entry point for the backend or a gateway as implied in the name itself. The API-gateway routes all the calls from clients to the relevant micro service running on the server. An API-Gateway plays a crucial role in security and scalability of the systems. Typically, several different web protocols might have been employed in the backend, the API gateway also translates client requests to be compatible with the protocols in place at the server side. An API-Gateway also composes multiple client requests into a single request, which in turn reduces the load on the server and allows for faster processing of client requests by reducing the round trips between the client and the server [17].

3.1.2.1 Problem

Implementation of a system using S.O.A. leads to a very granular application. This means that a single task required by a client can require calls to several different APIs. Let us talk about the video streaming service example in the previous section. Say on a client-side mobile application, the details of a video are needed to be displayed to the user. The details include, title of the video, cast, director and user ratings. From the perspective of the client, all these details are one component “Video details” but in the micro service architecture of our system all these details come from different services and that means different APIs. This means that the client side needs to make 4 API calls just to get the video information. This can induce a lot of unnecessary load on the server increasing cost and reducing performance and extra work for the client-side developers.

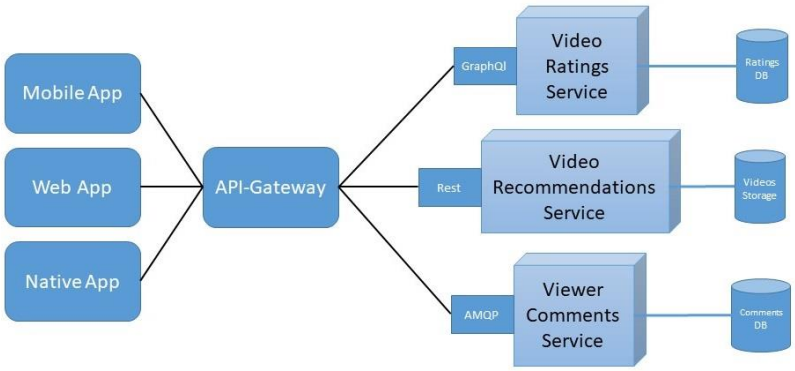

Figure 3 API-Gateway Pattern showing Protocol Translation and support for clients on various platforms.

In a future update this granularity might also change increasing or decreasing the number of API calls for getting the video details. Ideally any changes to the backend service architecture should be hidden from the client side.

Another problem is that different protocols might have been used for different services and the same goes for clients i.e. different data (return) might be needed for different types of clients e.g. web-apps, mobile apps and desktop apps.

3.1.2.2 Solution

An API-Gateway is implemented that acts as the only entry point for every client. As described in the earlier section, API gateway handles incoming request from each type of client, taking the request and sending it to each service in the backend that is needed to fetch the required information and sends it back to the client as an aggregated payload.

3.1.2.3 Outcome

An API-Gateway that is client specific, provides the APIs that are best suited to the type of the client. It removes the overhead that would have been generated because of multiple calls to receive video details which instead can now be done by a single call. All the complexities, architecture, locations and any future changes of the backend are hidden from the client side.

This pattern also has the drawback that the complexity of implementation is increased because of the API-Gateway component. The maintenance of this component adds to the cost of the system.

3.1.3 Health Check Pattern

In a system implementing a micro service architecture, it may happen that a service is running but unable to handle requests, or it may be down due to any number of reasons ranging from undergoing maintenance or server issues. Sometimes it is better to provide a way for the client side to check the status of the service before it is invoked to avoid running into unnecessary errors.

3.1.3.1 Problem

Detecting the status of a service, if a service is down due to any reason and getting an alternative route if possible.

3.1.3.2 Solution

Each service should have an endpoint that runs a number of self-tests, e.g. network connections, database connection, disk space, application unit tests etc. A monitoring service can invoke this endpoint, if the status is not Ok then the endpoint to the problematic service should be replaced with an alternative endpoint, if fault mitigation is implemented, otherwise a well composed error message should be returned to the client as well as an alert should be sent to the developers or maintenance engineers [18].

3.1.3.3 Outcome

This pattern enables periodic testing of a web-service, and avoids clients running into unexpected errors and behaviors

3.1.4 Service Registry Pattern

Number of services and their locations vary with time; each instance of a service usually runs in a Docker container or a virtual machine. IP addresses are allocated to these containers and VMs dynamically and may change over time. Apart from this, each service exposes an API at a specific port. For a client to access the services, they must invoke the specific IP address with the port location.

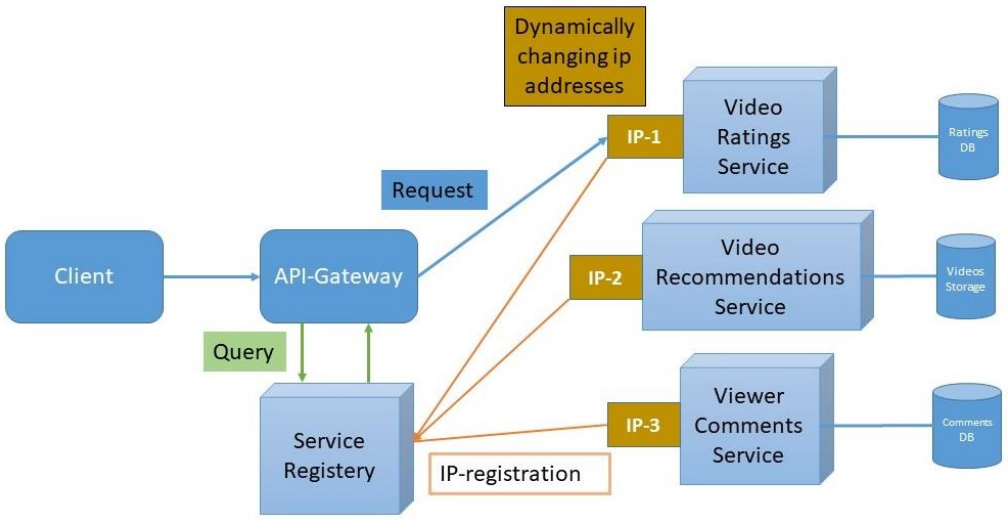

Figure 4 Service Registry Pattern

3.1.4.1 Problem

This changing nature of the locations associated with the instances of the services gives rise to the discovery problem that is, how should a client know which URI to call for a required service. The client in this case could be the API-Gateway itself or an external client application or a browser.

3.1.4.2 Solution

A dedicated database for services, which contains their instances and their locations and is updated at runtime, can be implemented. Such a database is called service registry. Every time an instance of the service is created, it is registered in the service registry, and subsequently de-registered when it is terminated. Any client that wants to access a service first consults the service registry to get the information about the required service. The service registry pattern can also extends the health check pattern, such that each time a client queries the service registry for a specific service, the service registry can first invoke the health-check endpoint of that service to infer if the service is operational or not.

For registration of the services with service registry, there are two possibilities[5]. One where the service instance registers itself on startup and de-registers on shutdown. The other

[5] Self-registration pattern and 3rd party registration pattern are two separate patterns but for the sake of simplicity included in the section with service registry pattern.

possibility is that a 3rd party application acts as registrar and handles this operation for the services.

3.1.4.3 Outcome

This simple solution enables the clients to discover the URIs of the required services dynamically.

3.1.5 Service Discovery Patterns

The service registry pattern addresses the issue of record keeping of the dynamically allocated URIs to the several instances of running micro-services. This pattern gives rise to two further patterns that address the logistics of querying the service registry and subsequently calling the required micro-service. The architecture of the following services remains the same as that of service registry pattern (fig. 4). The only difference is the component making the call to the service registry would be the client or the server instead of the API Gateway.

3.1.5.1 Server-side Service Discovery Pattern

With this pattern, the client always makes the request to a server that runs at a known and static location. The server makes the query to the service registry and forwards the clients request to the appropriate address. In this pattern, the server acts as an interlocutor and load-balancer between the client and the micro-services. The pattern allows for a simpler client-side code but adds an addition component in the network path that must be maintained and would also add to the server call's latency [19].

3.1.5.2 Client-side Service Discovery Pattern

This pattern makes the client obtain the address of the required service by communicating with the service registry directly. This pattern removes the intermediate server but may increase the complexity of the client-side code and also establishes the client's dependency on the service registry [20].

3.1.6 Database/Service Pattern

Most applications whether monolithic or service oriented need to save data in a database. For service-oriented architecture, multiple services will need to save the data relevant to them. The idea behind Service Oriented Architecture is to have micro services that are as

independent of each other as possible. Yet there is a possibility that one service could need to access, or update data owned by another service. Sharing a single database is counterproductive to this previous statement.

3.1.6.1 Problem

Services should be able to save their relevant data in a database and still be as loosely coupled as possible without any dependency on a shared database.

3.1.6.2 Solution

Each service should have their own independent database that conforms to the data storage requirements of that service. The database of one service should not be accessible directly from any other service, but the owner service should provide a special API for database operations from external elements [21].

3.1.6.3 Outcome

This implementation ensures that the services are as loosely coupled as possible from the data-storage perspective, and database can be adapted or implemented as per the requirements of each single service without having any impact on any other service. However, this pattern also increases the complexity of implementation, as data handling across multiple databases could be a difficult coding task.

3.1.7 Access Verification Pattern

The interface of an application built on micro services architecture exposes the interface of the services via numerous APIs. The access to these APIs must be protected such that only authorized parties can access the services. If the API-Gateway pattern has been implemented, then the API-Gateway is responsible for authorizing access. Yet the services themselves also need to ascertain the identity of the requester.

3.1.7.1 Problem

The system should be able to ascertain the identity of the originator of the request and authorize or decline the service accordingly.

3.1.7.2 Solution

When it comes to API the best and most often used way for authentication is using access tokens e.g. API keys[6]. The API key can be included in the request itself using parameters e.g. *?accessToken = "value"* or in the http header [22] [23].

3.1.8 Circuit Breaker Pattern

Faults in a system are unavoidable. However, the severity of the faults can vary. In a distributed system like one implemented using the service oriented architecture, there could be minor faults that are fixed automatically with time like, slow network connection, unavailability of a resource etc., On the other hand more severe faults could also appear which may render a service completely unusable for a longer period of time. A major problem could occur if this fault is propagated through the system by for example blocking several incoming requests, hogging resources or blocking threads.

3.1.8.1 Problem

A failsafe should be available, that prevents faults from cascading to other services and systems.

3.1.8.2 Solution

In the circuit breaker pattern, a proxy monitors the services for the number of recent failures. This information is used to decide whether to allow an incoming request to a faulty service proceed and wait till it times out or goes through or return an exception immediately.

The circuit breaker also allows a client to check if a previous fault has been fixed so that the client can try to access the service again. This way a client is prevented from wasting resources on trying to invoke a non-responsive service [24].

3.1.8.3 Outcome

This pattern prevents the faults of one service cascading to other services. But the complexity of client services might increase as the exception handling for faulty service

[6] API token can be generated using uuid, secrets or PyJWT libraries

has to be implemented at the client side as well i.e. the requesting service should be able adapt its behavior if the required service is not available

3.1.9 Gatekeeper pattern

Cloud native applications usually expose endpoints or URIs to which clients connect to via requests. These applications then handle the requests and process them by invoking other services as well as managing data-storage operations. If the security is compromised and a hacker gains access to the hosting environment of any of the services, they could have access to all the data that is accessible to the service which could include user's private information and credentials. A way to isolate the sensitive data and/or the hosting environment from the outside world is needed to prevent such malicious access.

3.1.9.1 Problem

Isolate the hosting environment of the application, i.e. the trusted applications that have access to data storage and privileged operations should be segregated from the interface that is exposed to the public.

3.1.9.2 Solution

To isolate the services that access data storage or perform operations based on client requests from the publicly accessible endpoints by implemented a gatekeeper that is decoupled form the trusted host and services. The gatekeeper is decoupled from all the sensitive operations, such that it does not have access to the data storage and neither does it perform any operations based on the client requests. Instead, the gatekeeper validates all the incoming requests, if the requests pass the validation criteria, then the request is forwarded to the trusted host otherwise it is rejected [25].

3.1.9.3 Outcome

The gatekeeper acts as a firewall, validating the requests and deciding whether to pass it on or not. The pattern could be used to protect services that handle sensitive data or perform critical operations. The gatekeeper should be implemented in such a way that it is a single point of failure. This pattern could also add to the server latency because of an added layer

3.1.10 Distributed Tracing Pattern

It is inevitable that at some point in time errors and problems will occur with the running services, for this reason, to allow easy troubleshooting and debugging of problems keeping detailed logs of the operation is very important. Usually a logging module or service is employed which stores detailed logs with the associated level of the log message (debug, info, error etc.) in a place. There should be a way to pinpoint the source of the error and what caused the behavior in the first place.

3.1.10.1 Problem

Going through extensive logs could be very hectic job, and there should exist a way to pinpoint the exact origin of the problem and which request caused the behavior

3.1.10.2 Solution

Each incoming request should be assigned a unique ID, which is then propagated to all the services it interacts with. This unique request ID should be logged with all the log messages to allow for easy debugging and record keeping [26].

3.1.10.3 Outcome

Logging of request ids allows for easier trace back and observing the actual behavior of the program associated with the original call that caused the problem.

Chapter 4. API Architecture Design

Previous chapters explore how latest software development strategies combined with micro-services architecture and its associated design patterns can help increase the throughput of an organization by decreasing the software development lifecycle. With adapting the micro-services architecture, a crucial design question arises of how the several independent services are going to communicate with clients in an effective and secure manner. In the case of a micro service "clients" could be other services with in the same eco-system as well as external clients who may want to use the functionalities provided by the services, for example a mobile application trying to get information about the objects present in an image from an "object identification" service of a machine learning/AI platform or a "recommendation" service within a video-streaming application that tries to get the user-ratings of videos to calculate a good list of recommended videos.

For any interactive software, interfaces to the external world whether it be users or software trying to access the functionality of the program are a crucial part and effect the overall quality of the product. *Web Application Programming Interfaces (Web-APIs)* provide a very practical, network-based access to functionalities and data. With the current trend towards cloud computing and Internet of Things, the application of web-APIs has seen a huge boom as well. With cloud-based infrastructure, remote devices can be used to perform various functions and collect data via sensors (offering platforms and infrastructure as services). But to enable interoperability among these cloud devices, a secure and dependable way to communicate and exchange data between devices and remote clients is crucial. This can be achieved via web-APIs that can be open to public or for internal use as required.

A poorly designed or poorly documented API could have an adverse effect on the quality of the software. As described in [27] a good API can lead to long term customers and a bad one can have a very adverse effect on an organization's image , as a basic guiding principle an API should be very easy to use and hard to misuse, and thorough documentation should be available for complex APIs.

As explained in chapter 3, different APIs can have different communication protocols depending on the requirements and the design architecture. This chapter aims to cover the best practices for designing consumer friendly API and provide a brief overview of API design protocols.

4.1 Web-API Design Styles

Web-API is a broad term that applies to any software interface that can be accessed over the internet using for example, Hyper Text Transfer Protocol (HTTP) without any design constraints. There are several types of web APIs, which can be chosen from, depending on the application requirements. Once the category of an API has been chosen there are several design styles which can be applied to the API depending on the requirements and architecture of the software. There are several standards that define the design styles like IETF, W3C and OASIS. Web APIs can be classified as belonging to one of the following design style categories.

4.1.1 Uniform Resource Identifier (URI) Design:

As defined in RFC 3986 [28], A URI is an identifier consisting of a sequence of characters matching the syntax rules. It enables uniform identification of resources via a separately defined extensible set of naming schemes. The APIs designed using this style use **C**reate, **R**ead, **U**pdate, **D**elete (CRUD) pattern for interacting with outside world. This means that most of the basic functionality can be directly mapped to HTTP methods. URI design pattern is directly related to the concepts of RESTful API design. The simplicity and ease of use related to this design make it one of the most commonly used patterns. The same simplicity associated with this pattern could result in very complex APIs if employed to larger systems, which could lead to a steep learning curve for the developers using the APIs. Moreover, the URI design pattern gives no standard for naming the URIs which means that each API could be unique. When it comes to systems that are going to be used internally in an organization a specific

Table 1 Mapping of HTTP to CRUD methods

HTTP Method	**CRUD Method**
POST	Create
GET	Read
PUT	Update (replace)
PATCH	Update (modify)
DELETE	Delete

design guideline can be used to keep uniformity for all the APIs, but if this pattern is applied to systems which are exposed to public or clients, this could be counterproductive.

4.1.2 Hypermedia Architecture (HA) Design:

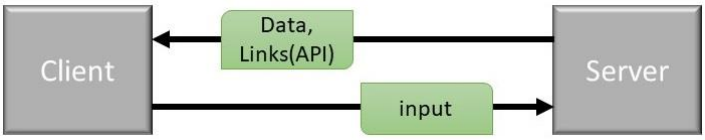

Figure 5 Hypermedia Architecture Diagram

Bedrock of hypermedia architecture is what tasks are to be performed and which route can be taken to complete those tasks, such that an API built on HA pattern helps the client navigate through the workflow by providing the links along with the results of the operation which signify what step the client can take moving further, just like a map can help a user navigate between different points [29] [30]. HA pattern works with dynamic URIs instead of static addresses as used in URI pattern. The messages passed in HA pattern are self-descriptive messages, each message represents the current state of the application and the links included in the message show which state the application could take next, the choice of the next state depends on the client/user. Hypermedia as the Engine of Application State (HATEOAS) is a component of the REST architecture that attempts to standardize Hypermedia Architecture Design.

4.1.3 Tunneling Style:

Tunneling is considered as one of the earlier API design styles and as such it is not particular to any communication protocol (transport agnostic) e.g. HTTP instead of remote procedure calls (RPC) are organized in a message e.g. in xml or json format. The server interacts with RPCs as if it is local to the client. Some famous systems that use tunneling style are SOAP and gRPC among others.

4.1.4 Event-Driven Architecture Design:

While other API design styles require either the server or the client to listen to triggers and react accordingly, in EDA, both client and server must listen and respond to events. This style embodies a Sender/Receiver model. WebSocket protocol is an example of an event driven system, several push notification systems in social media websites implement web-

sockets and hence the EDA design as it entails and allows for the development of more responsive applications.

4.2 Architectural styles

There are several software architectural styles that further enrich the API design styles defined in the preceding sections, the following sections give a brief overview of these architectural styles.

4.2.1 Representational State Transfer:

Representations **S**tate **T**ransfer (REST) is an architectural style that was introduced by Roy Thomas Fielding in his PhD dissertation [31]. REST architecture is gaining more and more popularity for developing webservices, and the webservices that employ REST architecture are termed as RESTful webservices.

REST architecture defines some constraints that must be met in order for a webservice to be considered RESTful. The constraints required for a RESTful architecture, are defined by Fielding [31, 32] and are summarized below:

1. **Client-Server:** This defines that the client application and the server side should be decoupled and able to evolve separately (Separation of concerns). The REST API should not be dependent on any particular communication protocol, instead the server resources should be available if a client is aware of the resource URI.
2. **Stateless:** All the interaction between the client and server must be stateless i.e. the server never stores any information about any request (caching, history or session information). If a state is necessary to be maintained for an application, then the client must handle this implementation. If for example authorization details are required for a particular request, then the authentication and authorization details must be included in every request.
3. **Cacheable:** A Client-cache-stateless-server implementation is required for a real RESTful system architecture. The constraint requires that the server categories each response as cacheable or non-cacheable. depending on the category of the response the client can reuse the response data for later requests of the same kind. This eventually helps improve the network efficiency by reducing the number of server

calls, although it can also decrease the reliability of the system because of redundant or old data stored in the cache.

4. **Layered System:** A RESTful architecture requires a layered architecture such that each component only sees the information only available for the layer that it is interacting with. This architecture is useful for web services in a system which also has legacy components, encapsulating the legacy services and clients from new services.
5. **Uniform Interface:** The main feature of REST architecture is the importance it places on a uniform interface between components. This is to optimize the interface for web applications and hypermedia data transfer. Four interface constraints define the REST architecture i.e. Resource identifiers, manipulating the resources, self-descriptive messages and hypermedia as the engine of app state. A uniform interface simplifies the system architecture by disassociating the implementation of a service from the interface it exposes to the outside world which in turn allows for the application to evolve independently of the other system components.
6. **Code on Demand:** This is an optional constraint in the REST architecture. The constraint implies that client functionality can be extended even after deployment by downloading and executing additional code. This feature enhances the extensibility of a system but also reduces the visibility of the functions.

REST architecture can be seen as a combination of the URI and hypermedia design styles.

4.2.2 GraphQL

GraphQL [33] developed by Facebook, is a data query language for APIs, it supports reading, writing, updating and subscribing to changes to data (in real-time commonly using webhooks). For an API developed using GraphQL, a client makes a request to the server explicitly defining the data that it requires. The server fills in the data fields requested by the client and returns the response. As opposed to a RESTful API, GraphQL APIs are organized using types and fields instead of endpoints, an example of type definition is shown in the adjacent figure.

The idea behind GraphQL is that the client asks exactly for what it needs as opposed to in a URI call, where the response may contain extra data that was not required by the client, in this architecture only the requested field from the specific type is returned to the client. Furthermore, references between different resources allow for a dynamic response i.e. data fields from multiple resources can be added to a single response. This allows for one request being sufficient for fetching all the required data reducing the overhead on the server by eliminating recurring calls for fetching different resources from the server. GraphQL also allows for dynamically adding new fields and types to the API without impacting the existing queries. This removes the need for versioning APIs and new functionality and resources can be provided in the same API without having any effect on the client-side functionality (with the updates the same API remains backward compatible). GraphQL is currently being used in Facebook, GitHub and Coursera among other commercial application.

4.2.3 gRPC

A remote procedure call (RPC) is a technique used in distributed computing where a running program initiates a procedure in a different address space most commonly on another computer on the network treating it like a local code/script. No additional coding on part of the programmer is required to specify the details of the remote interaction. gRPC is an open-source version of RPC developed by google with HTTP/2 as transport protocol, Protocol Buffers[7] as the interface description language and TLS and token-based authentication methods. This is one of the biggest differences between REST and gRPC architecture, mainly REST allows using any payload format but JSON is the most commonly used, while for gRPC Protobufs are used as payload format. the table below summarizes more differences between the two architectures.

```
type Video {
        Name: String
        Rating: Int
        Tags: String}
```

Figure 6 GraphQL Structure; type definition

A gRPC client typically exposes a local function that implements a specific operation, and behind the scenes that function invokes another function implemented on a remote server.

[7] Developed by Google, protocol buffers (like json and xml) are language & platform-neutral mechanism for serializing structured data.

For webservices and cloud-native applications, a plethora of programming languages and technology stacks are used, making inter-process communication a complicated affair. gRPC abstracts these complications, the developers are free to develop in the platform of their choice and all the communication is handled by gRPC [34].

Table 2 REST vs. gRPC

Domain	**REST**	**gRPC**
Payload format	JSON (most commonly used)	Protocol Buffers
Transfer protocol	HTTP (typically 1.1)	HTTP/2
Interface definitions	Uses HTTP verbs	Structured messages/ interfaces
Communication	Request-Response model	Allows bi-directional data streaming

4.2.4 Webhooks

Webhooks are user defined HTTP callbacks which are triggered by some events e.g. a commit to a repository, such that they initiate a resultant process in another web-service or webpage. Webhooks are usually user-configurable, and as they run on HTTP, its rather easy to integrate them into an existing infrastructure. Webhooks are inherently different as compared to web APIs. Generally, APIs support at least the CRUD methods, and need specific instructions as to which method shall they execute. On the other hand, web-hooks are called through HTTP POST methods from external sources as a result of some specific action or data manipulation [35]. Webhooks also allow for a near real-time information flow with less overhead as compared to implementing a real-time information flow using API calls.

Webhooks are registered by specifying a URL/API which should be called/notified once a certain event occurs. This API holds the code or process that needs to be executed when that event occurs. This adds a layer of abstraction such that the client does not need any knowledge of the process that would be carried out as a result of the event, but rather it

only needs to keep track of the API endpoints that need to be notified. The figure above shows two scenarios that can be implemented using webhooks. Client A only needs to inform the server or the webhook receiver that Event-A has occurred, it expects no response from the server. On the other hand, Client-B informs the server about an Event-B for which it expects some processing to happen at the server side and expects a response. In both cases the server should be able to handle incoming HTTP POST requests.

Webhooks are not much of an API design architecture but rather a framework for handling how systems interact with each other and at the base of it all a RestAPI, GraphQL or any other architecture could be implemented.

4.2.5 SOAP

Simple Object Access Protocol is another messaging protocol for exchanging structured and typed information between various SOAP nodes in a decentralized, distributed environment [36]. The data exchange format is XML. Like other specifications this enables cross platform processes to run and interact with each other seamlessly. A SOAP message is fundamentally a one-way transmission between SOAP nodes, from a SOAP sender to a SOAP receiver, but SOAP messages are expected to be combined by applications to implement more complex interaction patterns ranging from request/response to multiple, back-and-forth "conversational" exchanges. A SOAP message is in an xml data format comprising of 3 main parts, a SOAP Envelop, SOAP Header (optional) and a SOAP Body. The contents of the message are not specified by SOAP but rather defined by the application programmer.

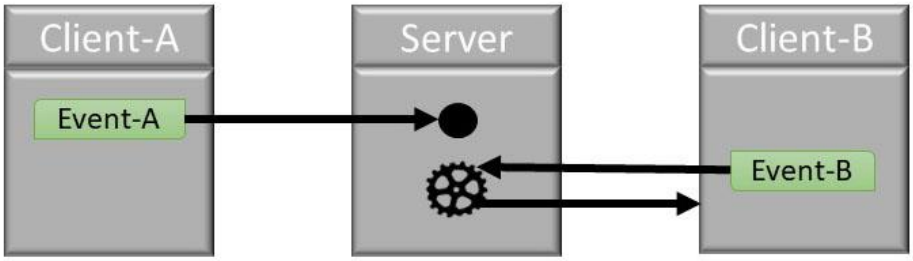

Figure 7 WebHooks architecture

SOAP as compared to REST is a standardized protocol and supports several WS protocols for security, addressing, and reliable messaging, among others. One of the features that especially distinguishes it from REST is the built-in error handling, such that the response always contains error information if something goes wrong with the client's request. SOAP also allows any protocol to be used and is not limited to HTTP as REST is, in some cases SMTP is also used in conjunction with SOAP.

4.2.6 Comparison

Table below provides a summary of the important differences between the different architectural styles described in this chapter. Depending on the requirements of a specific web-service any of

Table 3 Comparison of API architectural styles (== commonly used but not required)*

Parameter	REST	GraphQL	gRPC	WebHooks	SOAP
Summary	Data transfer between nodes, stateless	API Query language, user defined data in the response	Lightweight system built using remote procedure calls.	Automated serving of data and state updates.	Resource sharing in a systematic & standardized way.
Transport Protocol	HTTP	*HTTP	HTTP/2	HTTP	*HTTP & SMTP
Payload Format	* JSON, XML etc.	GraphQL query (JSON)	Protobufs	HTTP POST	XML

the above techniques can be used. From the table, the basic features of each style can be seen and consequently applied to the requirements at hand. In the software implemented in the course of this thesis, REST architecture is used for simplicity, easily available documentation and the application requirements mostly involved simple transfer of data between client and server nodes such that all use-cases could easily have been implemented with the REST architecture.

4.3 API Development Guidelines:

When it comes to developing REST APIs there are several best practices and design guidelines that exist, apart from the common best practices, each organization may define its own set of rules for the naming conventions of the APIs as well as how different aspects of an API operate. This section maps out the best practices that should be followed for defining the APIs as well as the development guidelines that are outlined for API development in Robert Bosch GmbH.

4.3.1 API URI Design Guidelines

As described in the earlier sections of this chapter, REST APIs use URIs to refer to the resources or functions that a RESTful service provides. Although no standardized document exists that controls how these URIs should be designed exactly, Internet standard RFC 3986 [37] defines the syntax of a URI as:

scheme "://" authority "/" path(empty) ["?" query] ["#" fragment] 1

an example of a valid URI as per the defined syntax could be with each part corresponding to a segment defined above by the RFC3986:

http://localhost: 5000/students/engineering? name = john#grades 2

Overtime the RESTful community has also developed a set of best practices and design guidelines for this purpose, which are summarized in this section.

- Every character in the URI contributes towards the resource it refers to. The URI should be concise and clear about what resource it refers so any trailing slashes at the end of the URI should be avoided for example in the URI given in "2", in order to access the students, section the URI should be:

  `http://localhost:5000/students`

  instead of

  `http://localhost:5000/students/`

- A hierarchical relationship between resources should be indicated using "/", the "path" place holder in "1" should clearly show this hierarchical relationship to avoid any confusions, as shown in "2" the URI is to access the students resources,

sub-category engineering students, and then the name of the student is passed as a query parameter.

- If a resource name is to have spaces or separators in between, for better readability, hyphens should be used instead of underscores as they tend to be unreadable when a hyperlink is underlined automatically.
- For naming convention all lowercase letters should be used for a URI. A URI using a mix of lower and uppercase letters would be different than the one using all lower- or upper-case letters even if the same names are used.
  Table below shows this more clearly.

Table 4 URI naming conventions

Uniform Resource Identifier	Status
http://localhost/students/engineering/enrollment-status	Same (best practice)
http://localhost/STUDENTS/engineering/enrollment-status	Same
http://localhost/students/engineering/Enrollment-Status	Different resource

- As a rule, a REST API should use the HTTP content-type header to identify the type of resource included with the payload. This means file extensions should not be a part of the URI or API call.

4.3.2 Development Guidelines

Development guidelines for RESTful architecture from Bosch in-house sources and standards provide a good structure towards developing secure and robust applications. The guidelines as stated earlier are not part of the REST architecture per say but depends on the organization and their personal preferences. The development guidelines by Bosch inhouse sources categorize the sub-sections in one of the following categories [38]:

- Mandatory
- Exceptions only with approval
- Recommended

An overview of these guidelines is given in this section.

4.3.2.1 Secure Web Development

To avoid malicious use of data and information, security is the most important feature for any web service. For developing secure webservice and APIs, several standards and protocol documents exist e.g. Bosch Enterprise IT Security Architecture (Bosch EISA) and OWASP Application Security Verification Standard (ASVS) can be used as Industry standard security protocols. Apart from this all web applications are expected to support HTTPS and SSL[8] encryption protocols [38]. The topic of securing a web application is quite important and so more details on this topic are given in chapter 5.

4.3.2.2 Logging

To allow easy debugging of problems, logging is an essential tool and categorized as mandatory part of the guidelines. Logging of the event stream to a centralized logging tool is the recommended approach, with a retention time between 14 and 30 days. The granularity of logging is usually an adjustable parameter allowing the client to use between various logging levels such as debug, information and error. A good log entry is expected to be through, and containing a message including relevant information and parameters and should address the "When, Where Who and What" questions. Logging of personal information without the client's consent is illegal and logging of any sensitive information should be done as a hash or encrypted value.

4.3.2.3 RESTful Design Guidelines

The URI naming guidelines from section 4.2.1 apply here as well as the standard REST architecture. The guidelines constitute standard REST design guidelines as well as recommendations from the CI department of Robert Bosch GmbH [39]. The recommended naming conventions for resources is plural nouns with the HTTP verb for example GET/students POST/students. Unique IDs that are not reproducible e.g. uuid, should be used to refer to a specific resource and should be a part of the API call. The use of HTTP

[8] SSL (Secure Sockets Layer) and TLS (Transport Layer Security), are protocols for establishing secure links between networked computers. [49]

methods in the correct context is an important part of the RESTful architecture. The table below shows the http methods and their characteristics.

Valid HTTP return codes should be included along with the server response corresponding to the execution status of the webservice. A complete list of HTTP codes is available online but as a rule of thumb, codes in 200 series correspond to a successful execution, 400 series correspond to an invalid request form the client, and 500 series correspond to server-side errors.

Table 5 HTTP methods and their characteristics [38]

Method	**Action**	**Safe**	**Success Return Code**
POST	Create a resource	No	201 – Created
GET	Get a resource	Yes	200 – Ok
PUT	Update a resource	No	204 – No Content
PATCH	Update parts of a resource	No	204 – No Content
DELETE	Delete a resource	No	204 – No Content

Error handling is very important part of API development and the API should be able to return the error code along with a message giving details about the nature of the error, as per the guidelines and the standard RFC7808 a JSON including the HTTP status code, the title and the message can be returned as a response to an unsuccessful request.

Versioning semantics and clear indication of deprecated APIs is also important. The versioning is recommended to be done as per three levels corresponding to the amount of change done i.e. Major, minor and patch updates, such that three decimals indicate the change like “v. 1.2.3”. A major change includes any change that need an update of the client side implementation and could cause the older implementation to break, a minor update entails structural changes to the API which do not require any action to the client side code and a patch could be changes to the API documentation etc. For indicating a deprecated API an *X-API-DEPRECATED* header should be included in the response from the said API.

For a good API performance and scalability are very important factors. For this API caching as described in an earlier section could be implemented, other than that support for compression could be provided in the API. This allows for a reduced size of the data packet, algorithms like *gzip* can be used for implementing payload compression. For compression support, *Accept-Encoding* and *Content-Encoding* headers should be included in the request and response respectively so that the server and the client know which algorithm has been used to compress the payload. Rate limiting is also another good technique for performance improvement. It ensures that bursts of heavy traffic are smoothed out over time. This in turn also protects the system against Denial of Service (DDoS) and brute force attacks. This can be implemented using the existing API Gateway and Manager.

API Security is an essential point to keep in mind when building a webservice. Apart from standard security practices like protection against SQL and NoSQL injection, Transport Layer Security is required, this means that the webservice must implement HTTPS protocol. For authorization and authentication techniques like OAuth 2.0, API-keys or HTTP basic authentication could be used. Data validation to enforce the input format and content type validation are also very important towards implementing a secure API. This prevents any malicious users from injecting data or scripts that could be harmful for the system.

Chapter 5. Safety & Security for Cloud Native Applications

Everything on the internet including but not limited to websites and web applications running REST APIs, ftp or other tech stacks are always at a risk of being exploited by malicious users, data breaches and hacks. Ensuring that an application is robust and stands firm against such attacks and that any data on the server remains secure is one of the most important part of developing any application. There are best practices as well as techniques that could be adopted while developing the web services that help make the application more secure, some of these have been mentioned in section 4.2.2.3.

5.1 Common attacks against web and cloud applications

A web service can be exposed to a variety of attacks from malicious user that try to interrupt the service or gain access to the data. This section provides a brief overview of some common types of attacks.

5.1.1 Denial of Service Attack

A form of attack in which the hacker intends to make a network resource unavailable by flooding the resource directly or the infrastructure surrounding it with a flood of bogus requests preventing real requests from going through and getting served. The common form of DOS attach is distributed denial-of-service attack (DDoS attack), where the concept is the same except the incoming bogus requests originate from different sources so that it makes it harder to block the attack.

Recently Amazon Web Services has been hit by a DDoS attack in February 2020, where its servers were hit by a peak traffic of 2.3 Tbps, in the past GitHub has been hit by 1.35Tbps and NetScout Arbor with a 1.7 Tbps. Although the companies have been able to fend off these attacks successfully [40].

5.1.2 SQL Injection

SQL injection is a form of attack that exploits security vulnerabilities in a software by injecting database queries disguised as user input. This can let the attacker gain access to

data that would otherwise not be available to them like, data belonging to other users, sensitive data like usernames, passwords and credit card information, and the attacker can also use this to gain access to the server infrastructure to cause denial of service.

An application becomes vulnerable to this kind of attack when proper input validation is not implemented, for example a user can pass escape characters or code snippets (SQL queries) via the input fields of a form, and if they are included in an SQL query without validating the input, the system could be compromised.

5.1.3 XML Injection

XML inject also works on the same principle as SQL injection but the queries in this case modify an XML database if proper input validation and sanitization is not implemented in the web application. The Web Applications Security Consortium (WASC) document defines XML injection as:

XML Injection is an attack technique used to manipulate or compromise the logic of an XML application or service. The injection of unintended XML content and/or structures into an XML message can alter the intend logic of the application. Further, XML injection can cause the insertion of malicious content into the resulting message/document [41].

5.1.4 XPath Injection

This is another form of attack for XML databases. XML data can be queries using XPath which is also a query language similar in concept to SQL, which can be used to find specific elements in an XML tree. Unlike SQL, XPath does not provide any access restrictions and can be used to access any part of an XML document. The WASC define XPath Injection attack as "XPath Injection is an attack technique used to exploit applications that construct XPath (XML Path Language) queries from user-supplied input to query or navigate XML documents. It can be used directly by an application to query an XML document, as part of a larger operation such as applying an XSLT[9] transformation to an XML document, or applying an XQuery to an XML document." [42].

[9] Extensible Stylesheet Language Transformations (XSLT) is a language for transforming XML documents into other XML documents.

Other forms of attacks include Cross Site Scripting (XSS) attack, OS Command Injection attack and brute force attacks. In an XSS attack, malicious users are able to inject scripts into a website, which when a client access can grant session access to the hackers giving them access to the user's data. In OS command injection the attack is done on the server side by injecting OS commands which also give untethered server access to the hackers. Brute force attacks are done against cracking password and access keys by trying out multiple combinations of characters until a combination works. This can be avoided by having strong and long passwords and keys, for example by using secrets module in python (see footnote 6 - page 26).

5.2 Critical Security Risks for web applications

Open Web Application Security Project (OWASP) is a nonprofit foundation that works to improve the security of software. The organization has compiled a list of top security risks that web applications face, and the developers and the IT departments need to address when developing and deploying web applications. Having an idea of what plagues the world of cloud and web application development is the first step towards developing secure and robust applications. This section lists and provides a brief overview of the risks as evaluated by OWASP [43].

- The attacks which are most common, and the applications are at the risk of facing from malicious users are injection attacks. There are several type of injection attacks e.g. SQL, XML OS etc. more details of common injection attacks are given in a later section.
- The second on the list is broken authentication. Incorrect implementation of session management and authentication modules allows hackers to gain access to otherwise restricted data and modules. There are various forms of attack which hacker can use to bypass authentication like brute force attacks, or automated attacks like dictionary attacks. To make a system more robust session management at server side should generate random session ID with high entropy, and application should implement test that do not allow users to set weak passwords. Multi-factor authentication should also be used when possible to mitigate brute force attacks.

- Improper protection of sensitive data e.g. identity, financial or healthcare data can allow hackers to use such data for criminal and malicious purposes. An application that deal with sensitive data that is handled or stored as clear text (e.g. operating on HTTP, FTP) is potentially vulnerable to this issue. All sensitive data, personal information passwords and ID should be stored as strong hashes, and encryption protocols like TLS/HTTPS should be used for transmitting the data over the network.
- Processing of external entity references in XML documents if poorly implemented could expose the application to remote code execution, DDOS and various other vulnerabilities. SOAP framework prior to version 1.2 are susceptible to this sort of attacks if XML documents or entries are passed to the SOAP framework. To avoid this situation, SOAP versions greater than or equal to 1.2 should be used. Also, it is recommended to use JSON instead of XML when possible and avoiding serialization of sensitive data.
- Improper implementation of access control and user groups could allow hackers to access parts of the application that should otherwise have been inaccessible to clients e.g. admin functionalities and access to private databases.
- Security misconfiguration, improperly configured HTTP headers and other such human errors are also a common security risk. Improper error handling that exposes stack traces to clients is also a huge security risk and should be avoided at all costs. It is recommended to disable any unnecessary features, ports, and default accounts on network applications.
- Cross site scripting (XSS) attacks enable hackers to execute client-side scripts in web pages viewed by other users, bypassing same origin policy.
- Insecure Deserialization of objects and payloads from untrusted resources could expose the API to unauthorized control of the system logic, where the hackers are able to change the system implementation and perform remote code execution.
- Using vulnerable components with known issues could expose a web application to attacks. To avoid this any unnecessary dependencies should be removed and monitor and update all components for security updates and patches.

- Insufficient logging and monitoring are a major point exploited by hackers. It is recommended to monitor and log all user event including but not limited to login, failed login attempts, and some processes should be in place that raise an alert when suspicious activity is detected by such events.

5.3 Securing cloud native and web applications

The first step towards developing a secure network application is input sanitization and validation. Every parameter that comes from an external source should always be validated first to verify that it is in fact the data type that is expected and includes no escape characters or scripts. All sensitive data including but not limited to client information should never be

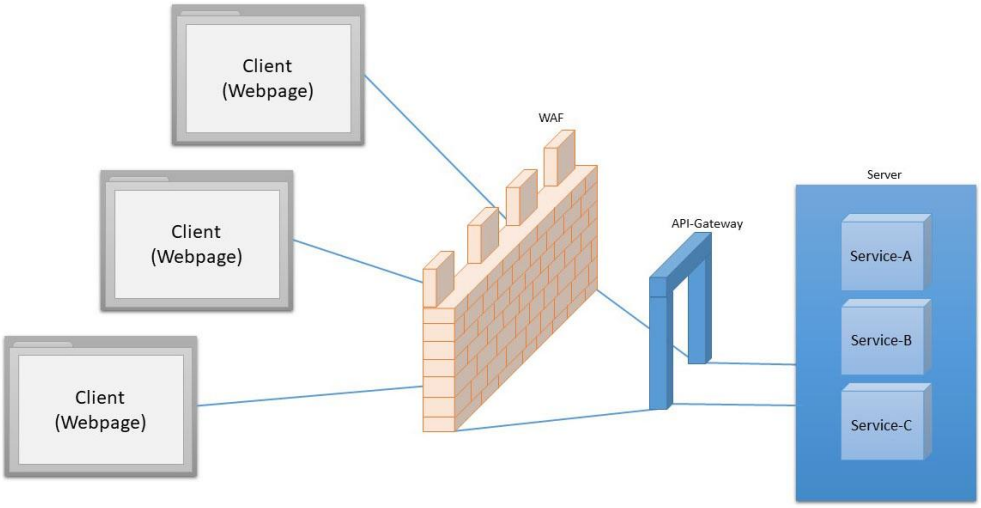

Figure 8 WAF + API Gateway architecture for web services' security

passed as query parameters and such information should be stored in an encrypted format. Some form of TLS for transport layer security should be used for encrypting data transfer and avoiding "man in the middle" attacks. Rate limiting can be used to avoid abnormal amount of incoming traffic. This also helps with avoiding brute force and DDOS attacks.

A Web-Application-Firewall (WAF) layer between the application and the internet is also a very valuable tool for mitigating attacks against web services. It operates on the application layer as per the OSI model (figure.9 shows the OSI model), and protects the application against XSS, injection and DDOS attacks among others, by monitoring and filtering the incoming traffic. An API gateway can also add an additional layer of protection

as explained in the previous sections, as API gateway provides its own set of security services like rate limiting, message filtering and authentication and authorization. For added security a WAF can be added in front of the API gateway as shown in figure 8. A WAF runs according to pre-defined policies from the IT department which are configurable usually called a web access control list. It can operate on a negative security model such that traffic that does not meet required standards is blocked or blacklisted, and it can also run on a positive security model, where a white list of pre-approved traffic is given and all other traffic is disallowed, both of these security schemes can also be used in conjunction in a hybrid model which is mostly how WAFs are implemented [44].

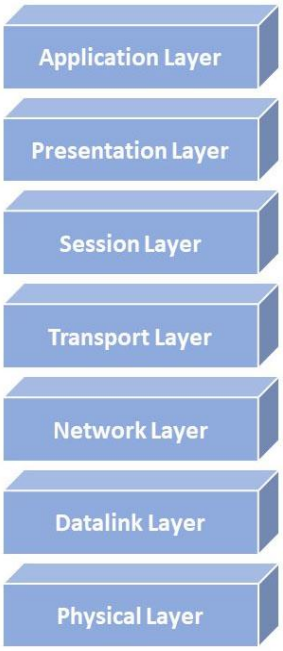

Figure 9 OSI Model

Chapter 6. HyAPI – Design & Architecture

HyAPI platform is developed as prototype to illustrate the usage of the principles of Service Oriented Architecture, software development strategies and API design patterns. The platform shows a proof of concept by implementing a monolithic software system (Feature Development Kit) which is used for software sharing between OEMs and Bosch and ECU software build process using web-services communicating via RESTful APIs instead of being bundled up as a single package.

The platform is designed in a way which allows developers to publish new tools and software as webservices and expose their interface for integration into new packages as RESTful APIs. This architecture abstracts all the functionality, dependencies and environment setup to the server, instead of having to go through all the setup for each package that requires those tools.

This chapter illustrates the requirements for the platform development, the architecture design as per the requirements and how the sequence of execution of the HyAPI.

6.1 Requirements

6.1.1 General Requirements

- **R-001:** A platform (HyAPI) which makes software available as web services.
- **R-001.1:** Web-service interfaces should be in the form of URIs
- **R-001.2:** Each functionality should have its own endpoint.
- **R-001.3:** Interfaces should employ the respective API style (REST, GraphQL etc.) as per the implementation.
- **R-001.4:** Clients should be able to call the URIs using HTTP requests.
- **R-001.5:** Sever should return JSON response with appropriate information as per the client's request.

6.1.2 Use-Cases

- **R-002.1:** User should be able to upload a file to the server.
- **R-002.2:** User should be able to download selective files from the server.

- **R-002.3:** Web-service to generate UML diagrams from a text description of the diagram.
- **R-002.4:** Web-services for tools used in FDK
    - **R-002.4-*a:*** Computation method generation.
    - **R-002.4-*b:*** OS Scheduling generation
    - **R-002.4-*c:*** CAN adapters generation
    - **R-002.4-*d:*** DCM configuration generation
    - **R-002.4-*e:*** Automated testing Services
        - CAN Testing
        - DSM Testing
        - Computation Methods Testing
        - Inputs Testing
- **R-002.5:** Web-service to demo data-streaming capabilities, for example data generated from running simulations.
- **R-002.6:** Web-service to demo Machine Learning processes as remote services.
    - **R-002.4-*a:*** Users should be able to perform object detection using a pretrained model available on the server.
    - **R-002.4-*b:*** Users should be able to train a TensorFlow model on the server using their own datasets.
    - **R-002.4-*c:*** Users should be able to perform object detection using a custom trained model.
    - **R-002.4-*d:*** Users should be able to download a custom trained model for their local use.
    - **R-002.5-e:** No setup (hardware or software) should be required at the client side for this implementation.
    - **R-002.4-f:** The service should be accessible from both command line interface as well as a graphical user interface in form of a web application.
    - **R-002.4-g:** The client-side web application should be platform independent.
- **R-002.7:** Status messages and images should be available for direct integration in HTML pages.

- **R-002.8:** User should be able to check the status of a service via a health check API.

6.1.3 Client CLI

- **R-003.1:** Clients should be able to access the web services via CLI regardless of the platform (Windows, Linux, Raspberry-pi)
- **R-003.2:** Client CLI should show that no tool dependencies are present (all tool dependencies should be abstracted to the server)
- **R-003.3:** The API should be accessible via CLI and GUI integration.

6.1.4 Security

- **R-004.1:** Transport Layer Security (TLS) should be used for all client server communication.
- **R-004.2:** API-keys should be used for client access authorization.
- **R-004.3:** Encoding-keys should be used for verifying uploaded documents.
- **R-004.4:** Input strings should be verified and sanitized before usage.
- **R-004.5:** File names should be mapped/translated to hashed keys (File-ID) and these file-ids should be used for all communications.

6.1.5 Client Server Communication

- **R-005.1:** JSON should be used as the standard format for requests and responses.
- **R-005.2:** All requests should be logged in history for trouble shooting and debugging.
- **R-005.3:** All incoming requests should be assigned a unique ID
- **R-005.4:** All server responses should include an appropriate HTTP code and a corresponding status message.

- **R-005.5:** Where possible, server should send the possible routes that the client can use after the current request.

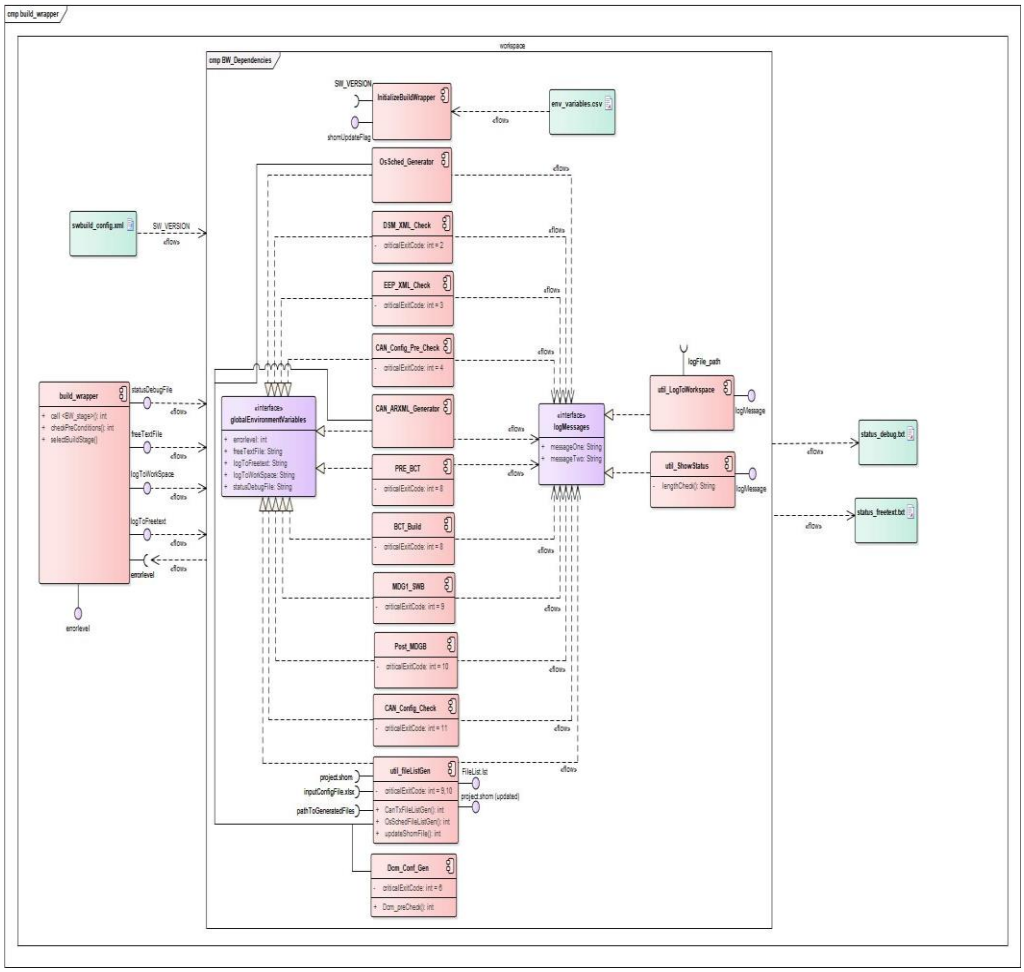

Figure 10 Monolithic Architecture of ECU build process

6.2 HyAPI Architecture

Figure 10 shows the monolithic architecture of the ECU build software, the model showcases the advantages of service oriented architecture and the associated software development practices, and how they can transform the software development processes to be more efficient as this monolithic system is migrated to a service oriented architecture. For developing the prototype Python is chosen as the development language. Flask python is used as the underlying framework for developing web services and exposing the endpoints of the webservice as RESTful APIs. As per the requirements JSON is used as the format for communications between the server and the clients. The design specs of the HyAPI platform are as follows:

- **Architecture:** Micro-service Architecture Pattern.
- **API Design Style:** Representational State Transfer (REST)
- **Payload Format:** JSON
- **Project Management:** DevOPs

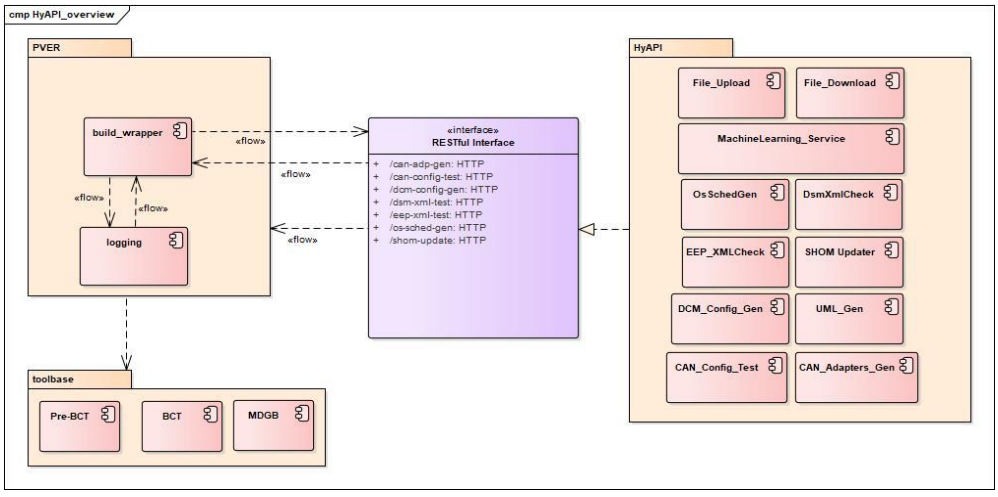

Figure 11 Components of ECU build software in Service Oriented Architecture

6.2.1 High level Architecture

As can be seen from the component diagrams of figure 10 and 11. The change in complexity of the two model is clearly visible. All the components that were previously included within the package and the communication between them was hardwired, have now been moved to the HyAPI package with no preset communication lines between them i.e. each component is independent of each other and exposes its interface for the client (in this case the PVER) to use as it requires. The client is free to choose how it routes the output or response it receives from a component internally. With the removal of all the extra components to an external package the size along with the complexity of the software package can also be reduced considerably. This allows for easily and independently updating both packages without any dependency on each other. It is also designed so that as new software components are developed they are expected to be deployed directly to

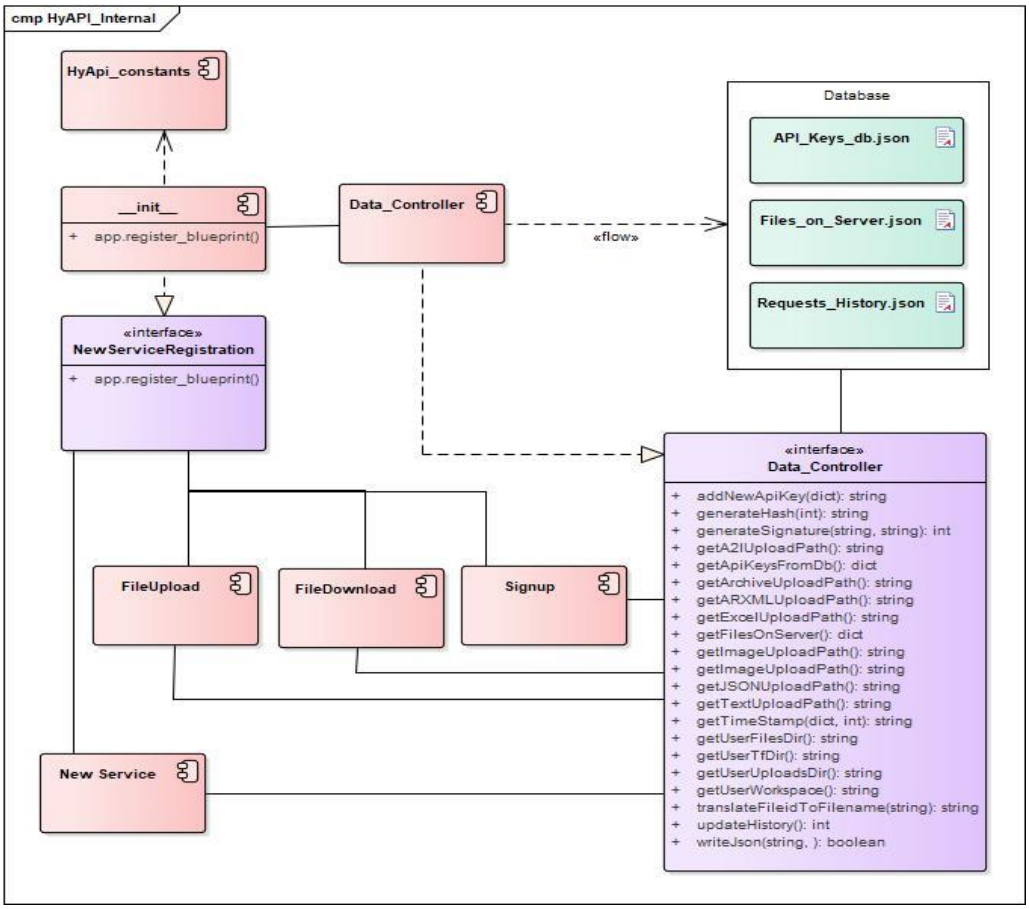

Figure 12 Component Diagram showing internal architecture of HyAPI.

the HyAPI package so they are immediately available for use by customer PVERs via API calls without having to wait for the delivery of the binary and the associated overheads in onboarding sessions.

Figure 12 shows the internal architecture of HyAPI, showing the base components whose interface is to be adapted by any New Services that are added to the platform down the line. The component *__init__* is the entry point into the application. All new services that are to be added to the platform must be registered with the *__init__* component to be recognized by the application and to make its API available to the outside world.

Data_Controller component is the interface to the internal databases of the application i.e. API keys, the record of all the user files on the server uploaded using the Upload service,

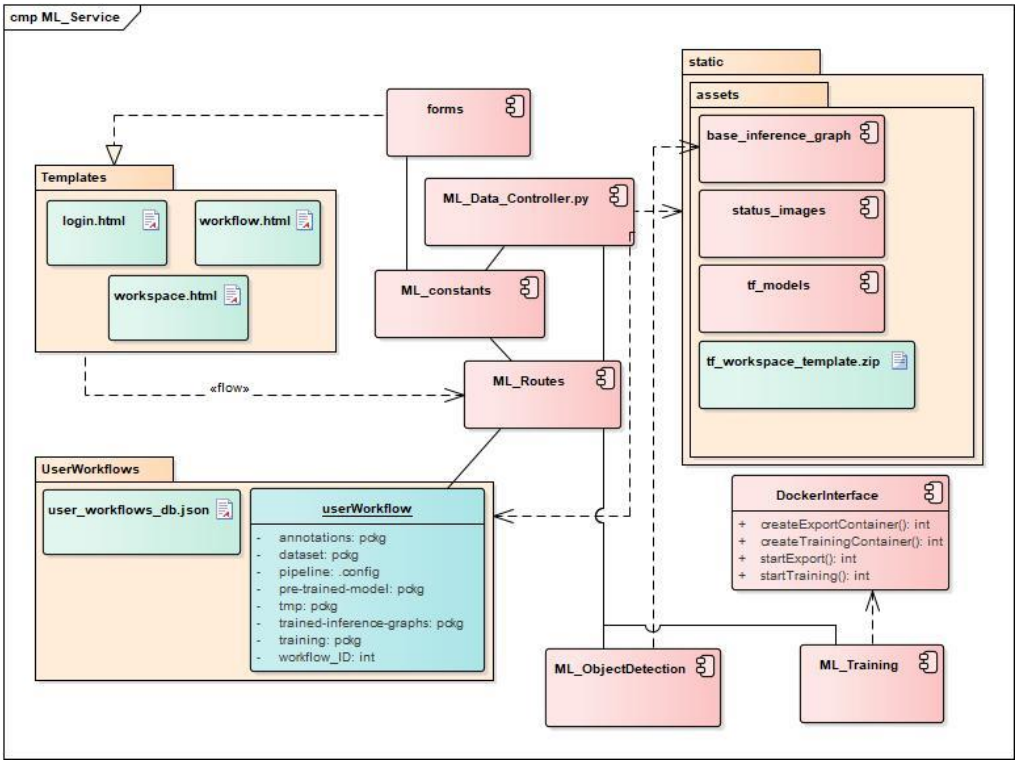

Figure 13 Architecture of HyAPI Machine Learning service

and the requests history log. *FileUpload*, *FileDownload* and *SignUp* services are the default services of the HyAPI package. These services allow clients to initiate meaningful communications with the platform, any file transfer between the clients and server happens only using the two services. This allows for safety and security by limiting the points of

data transfer between the server and external entities. All services need to tap into the interface provided by the *Data_Controller* in order to access the internal databases

6.2.2 Overview of Selective Implemented Services

As demo services several web-services have been implemented as shown in Figure 11 in the HyAPI package. Some services share the architecture and design between them so architectural overview of selective services is given in this chapter, which provide a general overview of the whole application.

6.2.2.1 Machine Learning Service

Figure 13 shows the architecture for the Machine Learning Service. The service is divided into two parts, one that supports CLI based operation for integration into automated workflows, and the other that is based on a web application GUI. The service architecture is as follows:

- **Requirements:** Satisfies **R-001,4,5(all sub-requirements), R-002.6, R-002.7, R-002.8**
- **API Design:** Representational State Transfer (REST) + Hypermedia Architecture (HA)
- **Payload Format:** JSON
- **Dependencies:** TensorFlow, Docker, Object Detection API

A breakdown of the components of the service is given below:

- Static – *Package*:

  This package is recommended to be included in all services. The object of this package is to include all the assets associated with the service e.g. icons, image assets templates etc. For the machine learning service, the pre-trained models; used for object detection, status images, as well as the template of the user workflows is included in the static package.

- Templates – *Package*:

  The templates package is recommended for services which include a GUI (Graphical User Interface) in addition to a CLI (Command Line Interface).

In case of the machine learning services, the html front end pages are included in the templates package.

- UserWorkflows – *Package*:

  This package is specific to Machine Learning Service. This service needs to store information for individual user workflows, which are associated with different instances of training jobs that a client may start. At the current time on implementation there is no limit to how many training jobs a client can start, but this will be limited as per the hardware and network capabilities.

- Routes – *Software component*:

  Routes script must be included in all web-services for HyAPI package. This component defines the Flask blueprint for the service as well as the public interface of the service using endpoint-URIs and their functionalities. For example, in the MachineLearning service following routes are defined in this component:

  - /process-image
  - /create-new-workflow
  - /train-user-model/<workflowID>/<inputFileID>
  - /freeze-model/<modelID>
  - /proc-status-graphical/<containerID>
  - /proc-status/<containerID>

- Constants – *Software component*:

  This software component is to accommodate all the constants and abstract them in a single location. This allows for easy access to constants and updating them for any future updates.

- Data_Controller – *Software component*:

  To segregate all the data manipulation tasks that are specific to a service this component is used. For ML service, this component interacts with the assets in the static package, use workflows as well as handling the

communication with the *Data_Controller* component of the HyAPI package for accessing platform databases.

Following software components are specific to the Machine Learning service:

- Forms – *Software component*:

  This component implements the web forms used in the GUI of the ML service.

- Object Detection – *Software component*:

  The functionality for performing object detection using the base model as well as custom models is included in this component. The routes component delegates the tasks to this component.

- Training – *Software component*:

  The functionality for training and exporting TensorFlow models on custom datasets is implemented in this component.

- DockerInterface – *Software component*:

  The training and exporting tasks for custom TensorFlow models is implemented using docker containers to make it possible to run them across platforms. When a training job is initiated this component creates a Docker container from the pre-built images and starts the training process in the container and once the process in finished the container is purged.

6.2.2.2 Data Streaming Service

Data streaming service show cases how some data can be streamed using the flask framework over a RESTful API based architecture. The service makes use of flask's *stream_template* and *generate* functions by sending chunks of data over the network.

- **Requirements:** Satisfies **R-001,4,5(all sub-requirements), R-002.5**
- **API Design:** Representational State Transfer (REST)
- **Payload Format:** Media dependent (images, text etc.)
- **Dependencies:** Flask

The idea behind this service is in order to monitor the progress of a running process like a simulation is a local process can poll a file that reflects the current state of the simulation

and that information can then be sent to the client in chunks providing real time status of the process/ simulation under observation.

6.2.2.3 CAN-Test Service

As the name suggest the service has functionality for testing the CAN adapters generated via other processes in the FDK (details about FDK are given in section 6.5). The CAN test verifies the generated weather the CAN frames generated from the user specifications are valid or not. as input the tool takes in a CAN specification file which contains the customer's requirements for the CAN configuration and an a2l file which is generated via automated processes in the FDK.

- **Requirements:** Satisfies **R-001,4,5(all sub-requirements), R-002.4-e**
- **API Design:** Representational State Transfer (REST), URI Design.
- **Payload Format:** JSON
- **Dependencies:** XLRD, CAN_Config_Validation_Tool

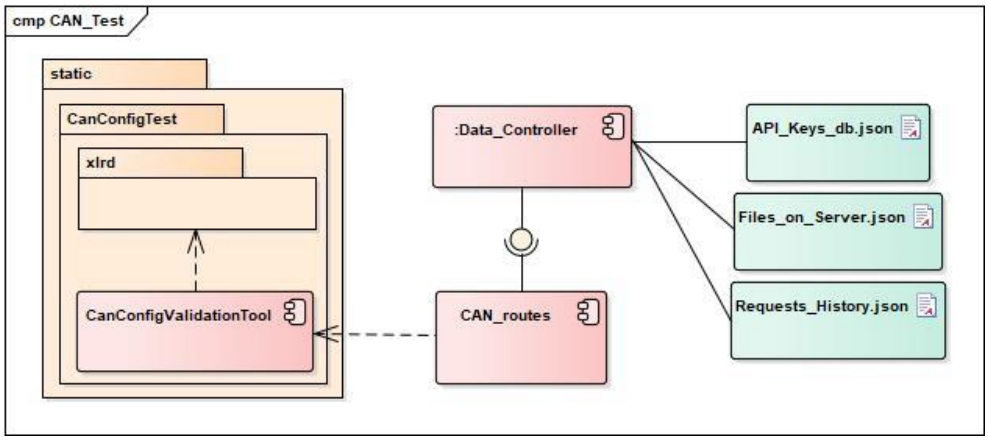

Figure 14 CAN-configuration testing service

Figure 14 shows the how the components of the service interact with each other as well as the parent components of HyAPI. The static package in CAN testing service includes the xlrd library which is a dependency for the tool at the base of this service which is also included in the static package. *CAN_routes* component implements the blueprint for the service which is then registered with the application as well as the information about the endpoint URIs. The service utilizes the API provided by the *Data_Controller* component of the platform for performing read/write operations on the database.

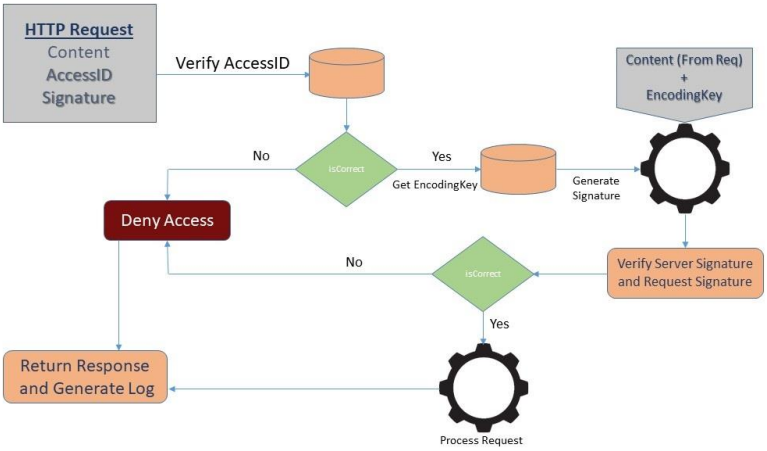

Figure 15 2-Tier architecture for secure file transfers

6.3 Security Schemes

In order to make the application secure and safe from unauthorized use several techniques have been used. All user inputs are type verified and no input parameters are directly used for executing any commands. Instead an ID model is used such that the client must provide the ID of which ever resource they want to refer to. if the provided ID is not found to be of the expected format an exception is raised and a "Bad request" code is sent back to the client. This combined with the fact that JSON is used as the backend database format instead of SQL, the exposure of the application to injection attacks is very low.

To implement encrypted communication the application run on HTTPS, the support for HTTP can be turned on or off depending on the requirements of the developer. Apart from this to ensure only authorized clients can access the functionalities, access token schemed are used. For each client a unique API key is generated and with a given API key (accessID) a client can access the functionalities for which the API key has access.

A 2-tier verification scheme is used for verifying the integrity of any files transferred to the server and to avoid any unauthorized user to upload a file to the server. Each client is assigned a unique *EncodingKey* in addition to the *AccessID (api key).* This secret Encoding key is used to generate a *signature* for files using hash-based message authentication code.

When a client wants to upload a file to the server, they must generate the signature of the file by using the secret *EncodingKey* with the HMAC algorithm and provide the generated signature with the request. On the server side, once the AccessID has been verified, the server gets the *EncodingKey* stored in the database corresponding to the AccessID and generates the signature for the attached file, if the signature generated at the server matches the one provided by the client then the file transfer is authorized otherwise the request is declined. Figure 15 shows the flow for 2-Tier security scheme.

For further security names of all files, whether transferred by the client or generated by the server are sanitized to remove all escape and special characters. After every operation where a new file is generated on server (file upload from client or file generated by a service) a unique UUID is generated which serves as the file's ID. For any further reference to the file this FileID must be used instead of the file's name which is not used on the server for referencing.

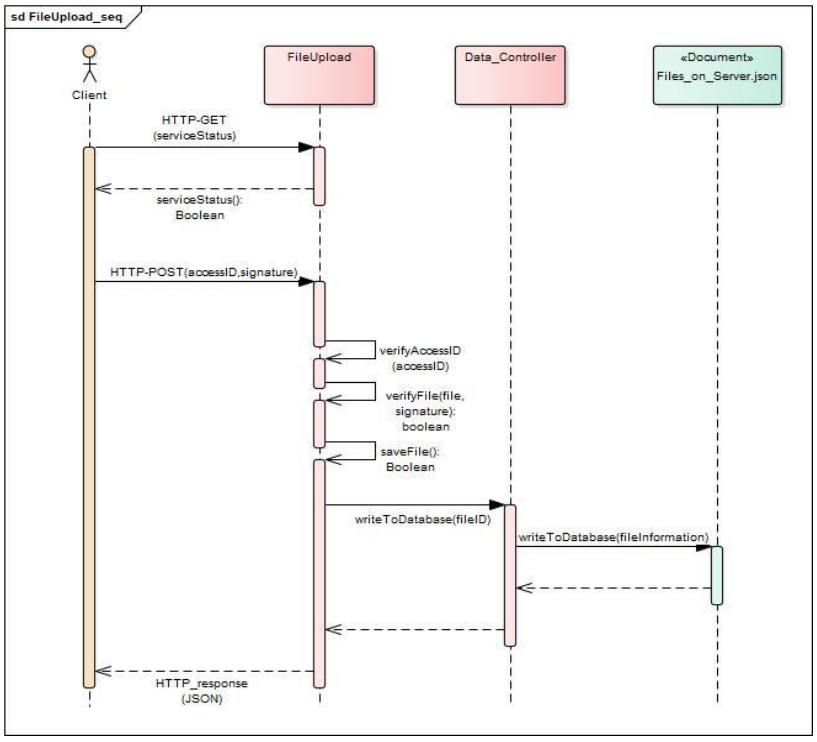

Figure 16 Sequence diagram for the file upload service

6.4 Application Flow

The flow of the application is illustrated in this section using some basic services. The sequence diagram in figure 15 shows the sequence of steps for uploading a file to the server. The process is initiated by the client via CLI by calling the health-check API for the *FileUpload* service. This API returns the status of the service, whether it is available or not. Depending on the response from the service, the client can continue with the process by making an HTTP-POST request with the file to be uploaded in the payload of the request. Detailed sample commands for making the HTTP-post requests are given in Chapter 7. When the server receives the client's request, the access-ID from the request is extracted and verified against the information in the database. After a successful

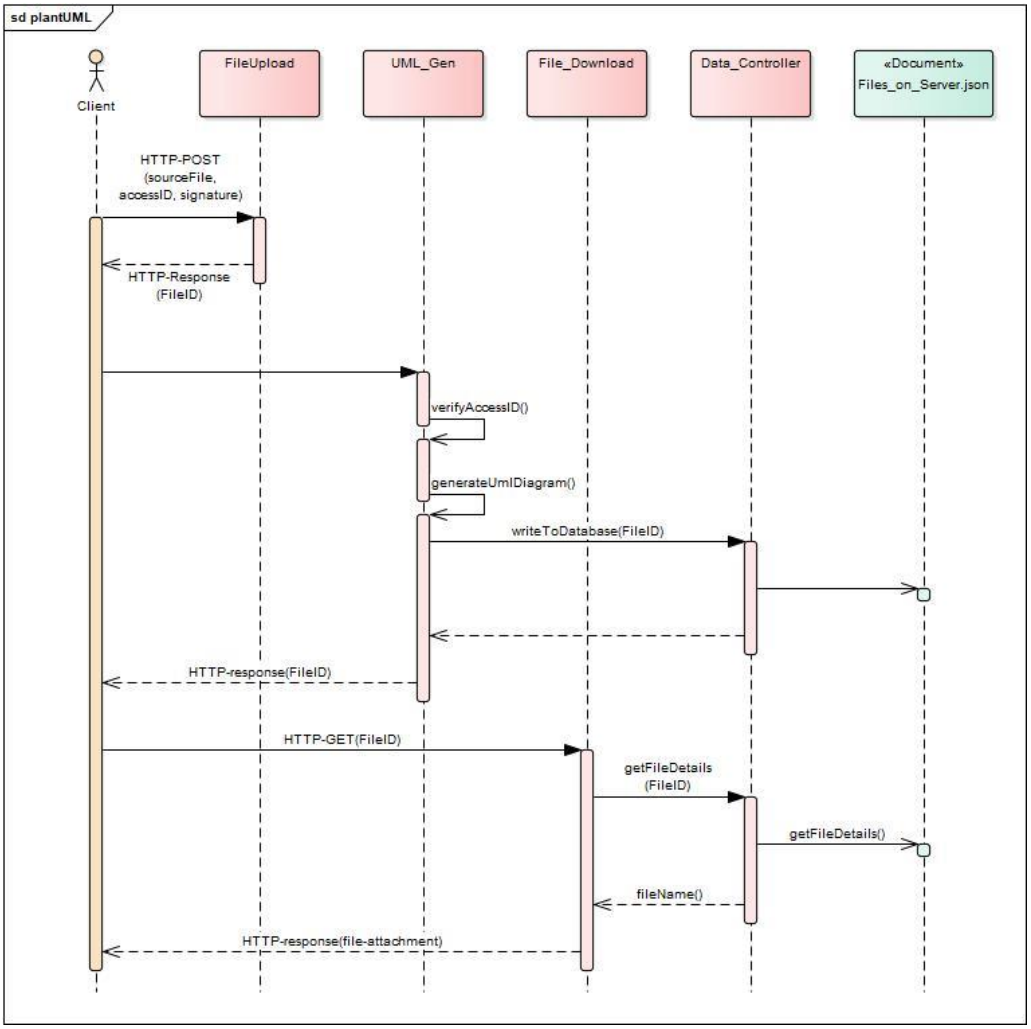

Figure 17 Execution sequence of UML-generation service

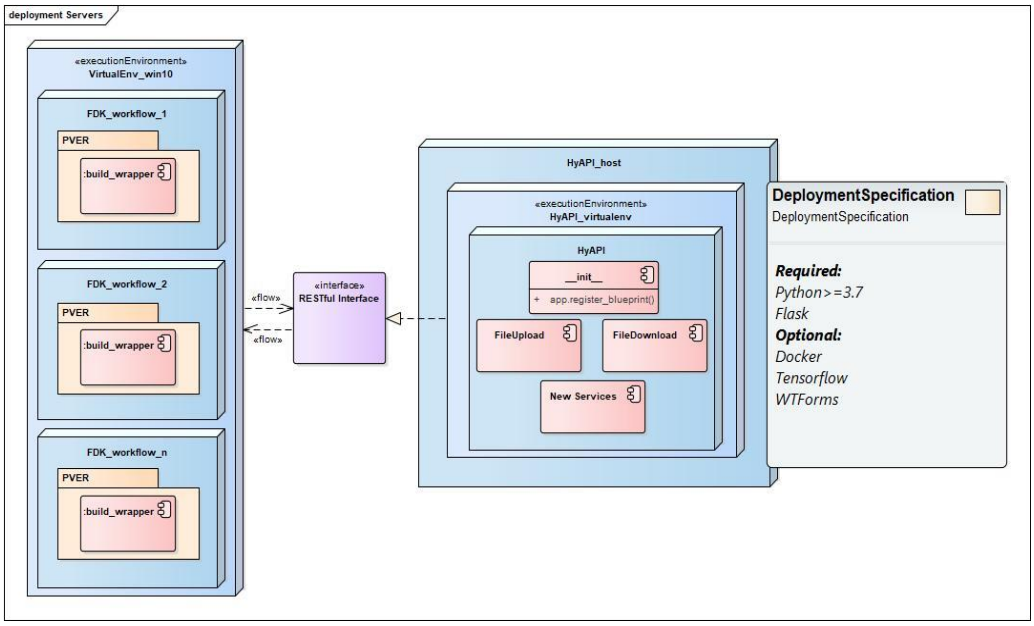

Figure 18 Deployment of HyAPI

verification, the signature of the file is verified, if this step also passes, the file is extracted from the payload and saved on the server in user files database. A unique file ID is generated for the uploaded file and this data is also saved on the "*Files_on_server*" database, so that the file's information can be retrieved for use later. The response in JSON format containing the fileID, a status message as well as the associated http-code is returned to the client.

Figure 16 shows the execution sequence of the UML generation service which makes use of a user uploaded file to generate a UML diagram from the source code. At the base of the service is the PlantUML [45] program. To initiate the process the service needs an input source file which must be provided via a fileID received from the upload service. The client then makes a POST request with the fileID of the uploaded source file. After the verification of the AccessID, the service calls the PlantUML process with the source file's path as a parameter and the generated file is moved to "User Files" database, and its information is written to the "*Files_on_server*" database. The client can download the generated UML file for further use by passing the FileID of the newly generated file to the download service.

6.5 Deployment

As a proof of concept, the FDK build process has been transformed into service-oriented architecture by moving certain components that previously used to be part of the package to web services in the HyAPI platform. The deployment of the nodes of the FDK system remain as it was before i.e. for each new workflow, a virtual machine is created on the server which hosts the execution environment for FDK. When a process is started by a client the server initiates the build process on the virtual machine. This is scalable as for each workflow a new virtual machine is initiated. Figure 18 shows this model, several FDK workflows from *1* to *n* can be initiated as required by the client. As shown in the deployment model, the services are now moved to a second server, name as *HyAPI_host* in the diagram, which has the execution environment required for running the platform. All the webservices should be deployed on this platform and their interfaces are exposed by the RESTful interface of HyAPI. To accommodate for scalability Docker containers are used, such that for each process a docker container is initiated so that multiple clients can make calls to the HyAPI at the same time. How scalable the model is will be limited by the hardware and network capabilities of the server.

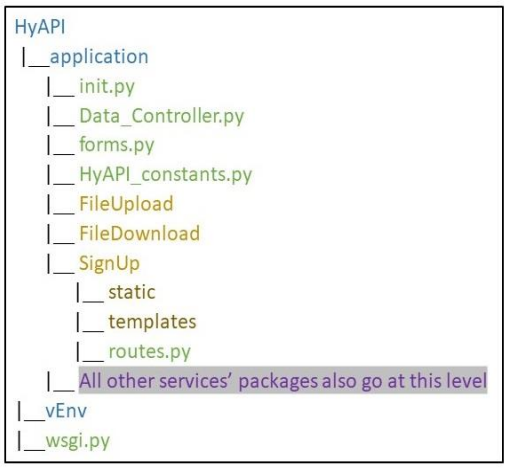

Figure 19 HyAPI blueprint and where to deploy new services

Internally HyAPI is build using Flask blueprints. This allows for a very modularized design and flexibility & ease in adding new services to the platform by giving a central means for extensions and services to register their operations with the main application. Figure 19 shows the overview of how the structure of HyAPI looks like. The main package which contains all the functionality is application which is at the first level of hierarchy. All the default services like upload and download services are available in this package as well as any new services should also be place in this package and registered with the init.py script. The init.py script can be used to turn a service on or off for use by the clients. vEnv is the virtual environment for the HyAPI execution, this package contains all the library

dependencies required by the HyAPI and any included services to run. Wsgi.py script start a development server to test the functionality of the platform.

Chapter 7. Results and Conclusion

In recent years, the entire business model of software industry is shifting from delivering software as a product to providing Software-as-a-Service (SaaS) in the cloud. The research presented here aimed to identify best approaches and architectures for API design as well as development of these cloud-based applications which aim towards providing software, tools and infrastructure as services. The final objective was to apply these findings to design a proof of concept of a cloud-based system that simplifies intercompany software cooperation by exposing services accessible over the network in a safe and secure manner. Although object oriented programming (OOP) has been around for decades, which allows local APIs to be used within the software package to enable modular programming, but at the day's end it produces a monolithic software system which incorporates all the individual components required for the system in one large package. This increases the complexity associated with maintaining, updating and delivering the software among other things, which in turn raises the cost of the project.

A trending new software architecture, named Service Oriented Architecture, to replace the monolithic architecture and the complexities related to it is studied and analyzed in the earlier sections of the thesis. SOA takes the modularized approach of OOP one step further by segregating individual components and making them completely independent of the software package. Instead each component, now behaving as SaaS, works in its own eco system as an independent service that can be accessed over the network using lightweight HTTP calls to its API. This approach removes the step required for integrating each component into the software's package individually, instead the services can be hosted over the cloud and can be easily accessed by the software/clients. This independence between components and a layer of abstraction provided by SOA makes it easier for multiple teams in a large organization to cooperate on developing software, as well as collaboration across company boundaries.

There are several design patterns that address the Service Oriented Architecture design paradigm, some literature reviews of these design patters report more than 170 existing design patterns. There is no one best pattern that can be chosen from, instead for each use

case the best match can be decided or in some cases multiple design patterns could be applied to a single service to provide the functionality in the best and standardized way. In this thesis some common design patterns have been discussed that apply to the use cases that were to be addressed in the implementation phase.

The most important part when using SOA for any use case whether it be for software sharing, artificial intelligence or machine learning services or any other functionality is how the different services should communicate with each other to provide a seamless experience to the clients. The communication protocols need to be safe and secure as well as lightweight so as not to put a large overhead on the processing and increase the response time. As the API now must manage calls from remote clients instead of the application package, the remote procedure calls or method calls as used in conventional architectures cannot be used anymore. Several architectural styles exist, and more are being developed to address different needs associated with SaaS and SOA. SOAP being one of the older styles and REST being one of the most popular styles addressing this use case. Several other styles have also been developed with each style addressing the needs of the developers as well as some specific use cases. Again, saying that a certain style is the best is not possible as it can differ from use case to use case. For this thesis, a combination of URI design, Hypermedia Architecture and RESTful architecture style have been chosen. The RESTful architecture provides simplicity, very vast collection of documentation and resources owing to a large and active community, a lightweight communication mechanism

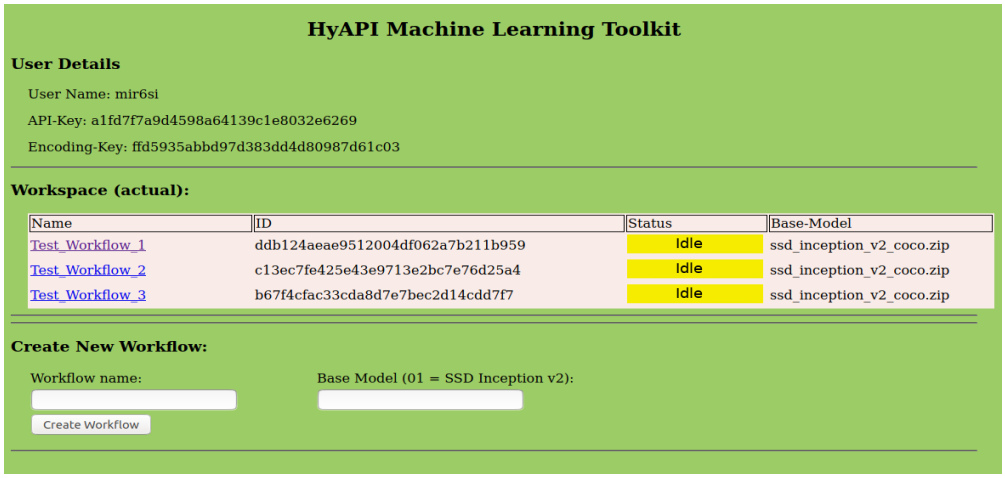

Figure 20 GUI for user workspace of the HyAPI ML toolkit web application

over HTTP and unless a specific use case arise which cannot be specifically addressed with RESTful API design, this proved to be the best choice for the applications and requirements of this thesis.

Serving the software on the cloud makes it susceptible to security breaches, unauthorized access and other forms of online attacks. The topic of security for such services and securing the data that such web services might store or interact with is of crucial importance. To avoid such scenarios, several strategies are explored in the thesis and implemented with the proof of concept software. Input sanitization and TLS in the form of HTTPs are used, which ensure protection & safety of data in transmission and against any malicious users that attempt to run unauthorized scripts through injection attacks. Input sanitization is also applied to any file uploads (file names) and any user defined names are replaced with a secure name generated by the server for added security and abstraction.

As data and file transfer is an integral part of the implemented services, a 2-Tier authentication approach is designed and implemented. This authentication feature adds another layer of security on top of the API key, by generating a signature of the byte content of a file using a secret key on both client and server side. By comparing the results, it is ensured that the file being uploaded has not been tampered with for example using a man-in-the-middle attack and also further establishing the identity of the client making the request.

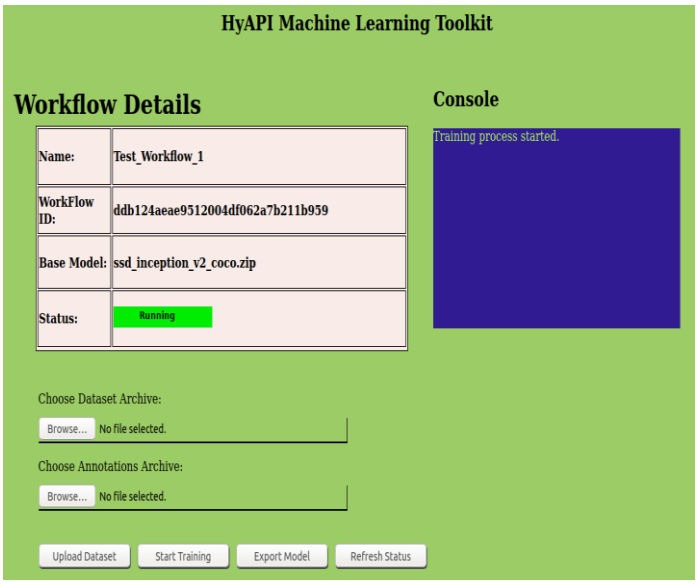

Figure 21 GUI of individual user workflow for HyAPI ML Toolkit

To better showcase the implications of the discussion and literature presented here, the application titled HyAPI has been developed which addresses several different use cases as per the requirements of the organization's current projects that deal with software

sharing. The application in its current state of art does not address continuous integration, continuous deployment, application monitoring and the usability of the application in production environment where the issues of scalability and the behavior of the application behind an API gateway can be studied. Further study is needed to determine what architecture and built-in APIs can be used to implement CI & CD and how the issue of scalability of cloud native applications and micro services can be efficiently addressed.

Figures 20 & 21 show the GUI of one of the components of HyAPI namely the Machine Learning Toolkit. This component implements and serves as a showcase for all the features implemented in the individual services developed in HyAPI. All the data in the tables is populated via API calls embedded in the web application, the information is fetched and displayed dynamically using only the user's credentials that must be entered to login to the webservice. All the buttons also invoke separate services on the HyAPI backend using API calls linked to the button press event. These services in addition to being available as a web-application with GUI can also be accessed individually from a command line terminal if they must be integrated in an automated workflow. Figure 22 shows how running the service for TensorFlow model training which is also implemented in the web application behaves when initiated from a CLI.

One reason for implementing SaaS is to enable running resource hungry software such as machine learning algorithms e.g. object detection, model training, image processing etc. on high performance server side machines so that such services can also be accessed from low-end devices like a mobile device or an electronic control unit with network capabilities. Typically, machine learning algorithms require very high-end GPUs for

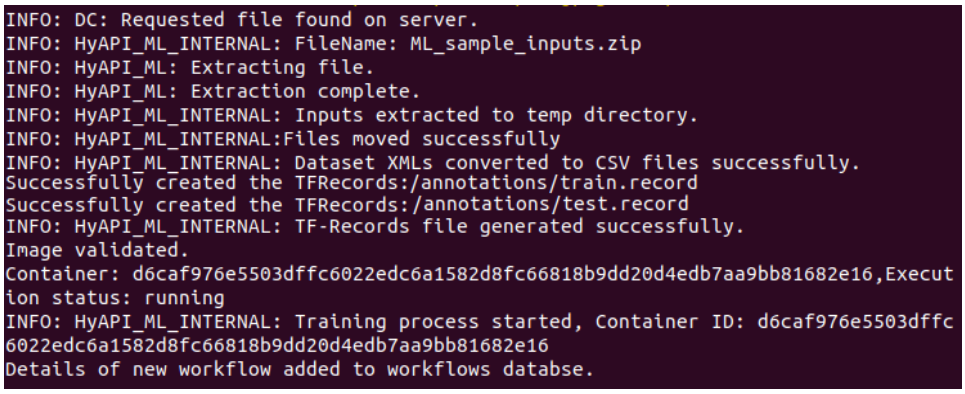

Figure 22 Sample terminal run (in image: Linux terminal, cli based application is also windows compatible)

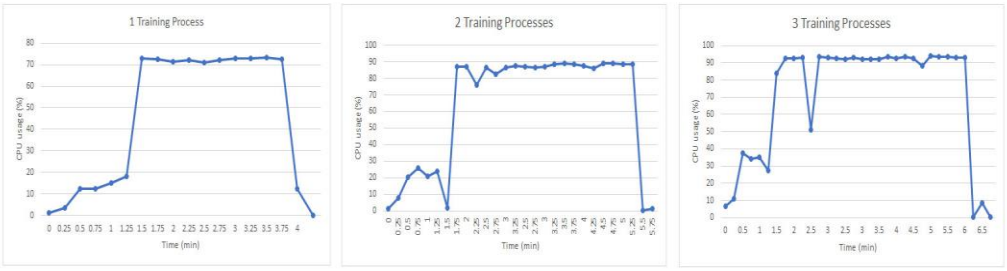

Figure 23 CPU usage for HyAPI running TensorFlow's model training jobs (L-R: 1,2 & 3 jobs simultaneously respectively)

efficient processing hence are not viable for mobile devices. To show case this capability, Machine Learning toolkit was implemented as part of the HyAPI, its design and architecture are described in section 6.2.2.1. The performance analysis was done on a high-end laptop running Linux with Nvidia GeForce GTX1010i GPU and Intel core i7 with 2.8GHz and 8 CPUs, and the client side running on Samsung Galaxy Tab A with android OS. This setup also shows the cross-platform operability offered by the HyAPI platform.

Figure 20 shows how CPU usage varies on the computer while running up to three training jobs simultaneously. This also shows that the platform is capable of handling multiple client requests at the same time and can be scaled up as per the requirements of the system. It can be seen from the statistics that the CPU usage for one training job increases from 70% to almost 95% for 3 training jobs and the training time also increased from 4 minutes to 6. Figure 21 shows the CPU usage, memory and energy usage statistics on the android device which runs the client side web application, The figure shows the point in time where the training process is started from the client side and a very small increase in the CPU usage (about 10%) can be seen and a memory usage of less than 32Mb attributed to running the browser app and very light energy usage is seen during this time frame, and it goes back to idle state once the request has been sent to the HyAPI server side.

These readings assert that this architecture is very viable option for running processor hungry software components on low-end devices by exploiting the distributed cloud-based nature which is possible by using SOA with RESTful APIs. Additionally, using this architecture also allows for simpler software cooperation workflows making the process of software development and collaboration between remotely located teams more manageable and allowing for faster roll out of new software components.

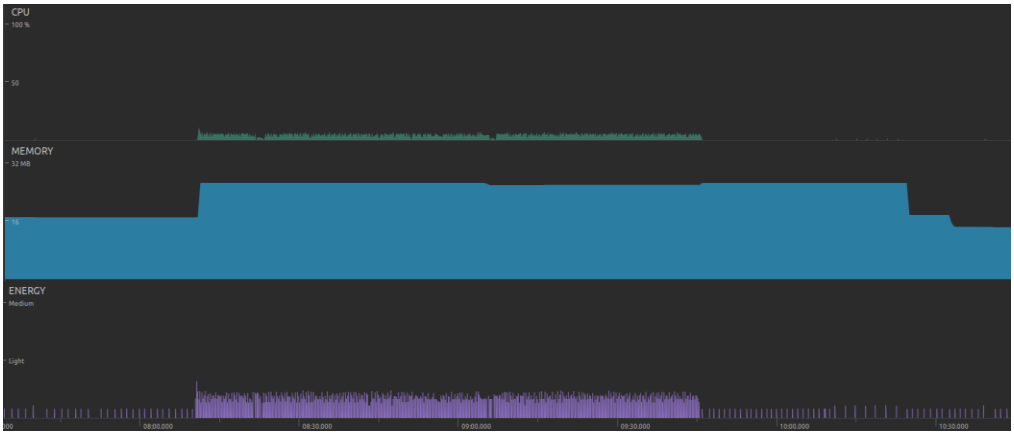

Figure 24 CPU usage on the android device running HyAPI client-side web application.

Further research is needed to assert what patterns can be used to for developing APIs with allow for configuring easy CI/CD pipelines. Apart from this, edge computing is taking the spotlight as the successor to cloud computing. Edge computing brings the computation that is usually performed on the server side as well as the data storage infrastructure, closer to the location where it is needed further improving the response times. Using edge computing and compatible API design practices, in place of cloud computing for applications and services that require faster response times that are close to real time such as automotive applications could make this application usable for real time applications.

As the world moves towards more cloud-based solutions and remote server farms providing virtually unlimited computing and data storage resources, providing software-as-a-service will eventually become inevitable[10]. A good and efficient API is the most crucial part of having a software as a service in the cloud as without efficient and secure communication from the outside world to the remote software components, the system would not have much value. The discussion above provides an insight into how a good cloud-based software service can be developed and what design practices can be used to develop a good and functional API for such services.

[10] Perhaps like a digital version of the construction model adapted by the ancient Egyptians for the great pyramid, as I compared the analogy in the first chapter.

Appendix A HyAPI – Documentation

Setup and Installation

The code of the developed application platform HyAPI is maintained on bitbucket repository, and for further development can be cloned or forked from this link. After a successful cloning of the repo to your local PC you should have a working copy of the source code of HyAPI and the associated client CLI, but some further steps are needed before the application will be able to run properly on a new PC.

The following steps must be run with an account with admin rights activated. It is also recommended to use Anaconda prompt instead of the regular windows command prompt to perform the steps in this section, this avoids any confusions caused because of multiple installations of python and improper Python Path setup in the environment on the PC. The PC in use must have python version 3.7 or later and the virtual environment package installed if the virtual environment package is not available following command can be used to install it.

```
pip install –user virtualenv
```

After a successful installation, open a terminal and navigate to the HyAPI directory, and use the following command to create a virtual environment:

```
virtualenv vEnv
```

Activate the virtual environment by running the following script in the newly created vEnv directory:

```
..\HyAPI\vEnv\Scripts\activate
```

The HyAPI package contains a text file named requirements.txt. This file contains a list of all the libraries required for the HyAPI package, and these dependencies must be installed in the newly created virtual environment. The following command allows pip to read the library names from the file and install them in the environment:

```
pip install -r requirements.txt
```

Once the installation is complete you can run the server by navigating to the HyAPI directory and running the following command:

```
python wsgi.py
```

Please note that if you are not using Anaconda prompt you might have to give the full path to the python executable in the command.

Now a development server should start in the prompt, depending on the network setting now you should be able to use the webservices included in the HyAPI package by making http requests to the available APIs

Client-Side Documentation

Demo command line scripts have been developed to show how a client will interact with the HyAPI server. To demo cross platform operation the commands can be run on both Windows' command prompt and Linux's terminal. Each script contains information about the input parameters it expects, the format of the command line call to the script as well as the output format it generates in the header of the script file. This section is written with "UseCase_CompuMethod.cmd" script as reference.

The *ClientSide-CLI* component contains *example_inputs*, *example_outputs*, *config* and *helper_scripts* subdirectories. The example_inputs folder contains demo input files, whose paths can be given to the client scripts as path parameters, user can choose the files of their choice here as long as they are of the right format. the example_outputs folder is a placeholder where the scripts are coded to generate their output files. The scripts are written so that the outputs are always genearated in the directory:

<current_script_path>\example_outputs\...

The config folder contains the configuration details that are used by the scripts. The scripts are developed this way to remove hardcoding in the script files. The client must edit and update the contents of these files according to their local environment. The folder contains the following files:

- **base_url.cfg:** This file should contain the URL of the HyAPI server, for demo purposes this is set to *http://localhost:5000/hyapi* in production or a server running over the network this should be changed to the URL or the IP of the host computer respectively.
- **cred_access_id.cfg**: The API key specific to the client should be put in this file.
- **cred_encoding_key.cfg:** The encoding key specific to the client should be put in this file.
- **output_directory.cfg: P**lace holder directory for now, may be used in future implementations or if the output directory is to be changed from script path to some other path on the client PC.
- **python_path.cfg:** This file should contain the path to the python installation on the client's PC where the python.exe binary is located.

Adding New Functionalities

When a new service is to be added, a directory for the service should be created under the application folder as shown in Figure 17. This directory for the new service should include all the service specific logic and assets including the database for the new service, as there is no central database for the platform instead each service is expected to have its own database and database handling. This is meant to keep things secure, segregated and independent from each other.

Each new service must have a routes module which includes the declaration of the service's blueprint as well as the routes associated with the API of the service, sample of blueprint declaration is given below:

```
service_bp = Blueprint('service_bp',__name__,
             template_folder = "templates",
            static_fodler = "static",
             url_prefix="/hyapi/new-service")
```

The blueprint of the new service must then be registered with the application in order for the service to be available on the network. For registering the new service, following lines

must be added to the __init.py__ module. In the code lines below the {} show the placeholders which must be replaced with the actual names of the modules and objects as per each service.

```
from {.service_directory} import {routes_module_name}

app.register_blueprint({routes_module_name}.{blueprint_name})
```

If at a later stage a service need to be turned off, the two lines for registration of the service in with the app in the __init.py__ module can be commented out and the service and its functionality will not be available any more.

HyAPI Application Programming Interface

Endpoint	HTTP method	Return codes
Default Services	**Base URL:** /hyapi/	
/upload	POST	201, 400, 403, 500
/download/<fileID>	GET	200, 403, 404, 500
/filestatus/<fileID>	GET	200, 403.404
/signup/cli	POST	201, 400, 500
General Demo Services	**Base URL:** /hyapi/service	
/generate-uml	POST	201, 403, 404, 500
/data-stream-gui	-	-
/data-stream-cli	-	-
/data-stream-cli-2	-	-
/can-config-test	POST	201, 403, 404, 500
/validate-compu-methods	POST	201, 403, 404, 500
ML/process-image	POST	201, 403, 404, 500
/ML/create-new-workflow	POST	201, 403, 500
/ML/get-user-workspace/<accessID>	-	-
/ML/proc-status-page/	-	500
/ML/proc-status/<containerID>	-	200, 500
/ML/proc-status-graphical/<containerID>	GET, POST	200, 500
/ML/train-user-model/<workflowID>/<inputFileID>	POST	201, 404, 500
/ML/freeze-model	POST	201, 403, 500

Machine Learning Toolkit (GUI)	**Base URL:** /hyapi/AI/toolkit	
/login	GET, POST	404
/workspace/<username>/<AccessID>/<accessToken>	-	-
//../create-new-workflow	POST	404
//../<workflowID>	-	-
//../<workflowID>/start-training/<inputFileID>	POST	-

A list of the implemented endpoints is given in this section. The simplest way to interact with the endpoints is to use the cURL command on a terminal with the appropriate parameters and the API endpoints given in this section. As a prototype is developed as a part of this thesis the development WSGI server with the flask is used to demo the application. Because of this the base url of the server is *http://localhost:5000/*. The appropriate base URL must be appended to the start of the endpoints listed in the preceding table.

The API shown above corresponds to the services implemented in HyAPI. The following lists shows the implemented services and a brief description of each service.

- **Signup:** This service handles onboarding of new users. It generates and assigned unique AccessID and encoding keys for the new users and also initializes the workspace and database services for the new users.
- **Database:** This is not an open service, rather provides APIs for internal and developer usage. It provides functionality for interacting with the database, maintaining the user's workspace and individual workflows. The reason for it not being available to clients is to protect access to private assets and user information.
- **File Upload Service:** This service provides the clients with the API to upload files to the service in a secure manner, it generates and assigns each file with a FileID which must be used to access the file in any future transaction. This service also employees file verification procedures, input sanitization as well as the 2-Tier verification to ensure no unauthorized file (type or unallowed size) is uploaded to the server.

- **File Download Service:** This file provides the API to clients to download the files generated by the several services in the HyAPI e.g. the output of the CANTestService or ML toolkit. The generated files can only be accessed via this service and by providing a valid fileID and AccessID.
- **PlantUML Service:** This service takes in a text-based description of the UML file and generates the UML model based on the uploaded text file. This service returns a fileID associated with the generated model file to the client. The input text file must be uploaded using the upload service, and the generated model can be downloaded using the downloaded service.
- **CANTestService:** This service performs a test on the CAN frames generated during the build process. The service shows how testing can be performed in a cloud-based environment. The service again depends on the upload and download service for transferring data from and to the client.
- **ML_ClassificationService:** Machine Learning service, provides object detection and Machine learning model training capabilities (using tensorflow in the backend). The service is accessible via CLI calls as well as a partner web application.
- **ML_App:** Implements the web application for the ML_ClassificationService.
- **DataStreamingService:** The streaming service shows how live data can be streamed over a web API. This service is especially useful when providing a simulation as a service or infrastructure as a service. When a special software which must be run on a remote LabCar or other infrastructure which is not physically available to the client, then the data being generated on that hardware or the running simulation can be streamed in real time to the client via simple API call.

References

[1] A. Balalaie and A. Heydarnoori, "Microservices Architecture Enables DevOps - Migration to a Cloud-Native Architecture," in *IEEE Software*, 2016.

[2] G. Elliott, Global Business Information Technology: an integrated systems approach., Pearson Education, 2004.

[3] Elysium Academy Private Limited, "What are the Software Development Life Cycle (SDLC) phases?," August 2017. [Online]. Available: https://www.linkedin.com/pulse/what-software-development-life-cycle-sdlc-phases-private-limited/.

[4] R. Lekh and M. P. , "Exhaustive study of SDLC phases and their best praxctices to create CDP model for process improvement," in *International Conference on Advances in Computer Engineering and Applications*, Ghaziabad, India, 2015.

[5] R. Hoda, N. Salleh and J. Grundy, " The rise and evolution of agile software development," *IEEE Software*, vol. 35, pp. 58-63, 2018.

[6] T. a. D. T. Dybå, "What Do We Know about Agile Software Development?," *Software, IEEE*, vol. 26, pp. 6-9, 11 2009.

[7] G. K. Hanssen, T. Stålhane and T. Myklebust, SafeScrum® – Agile Development of Safety-Critical Software, Springer, 2018.

[8] D. Wells, "When should Extreme Programming be Used?," 1999. [Online]. Available: http://www.extremeprogramming.org/. [Accessed 06 2020].

[9] Y. K. H.K. Raji, "Kanban Pull and Flow - A transparent workflow for improved quality and productivity in software development," *Fifth International Conference on Advances in Recent Technologies in Communication and Computing*, 2013.

[10] Amazon Web Services, "What is DevOps?," [Online]. Available: https://aws.amazon.com/devops/what-is-devops/. [Accessed 05 2020].

[11] G. G. J. H. N. S. Christof Ebert, "DevOps," *IEEE Software,* 2016.

[12] Red Hat, "What is CI/CD," Red Hat, [Online]. Available: https://www.redhat.com/en/topics/devops/what-is-ci-cd?sc_cid=7013a000002DFrZAAW. [Accessed 05 2020].

[13] M. A. C. Mary Poppendieck, "Lean Software Development: A Tutorial," *IEEE software,* 2012.

[14] T. m. K. S. J. H. Mary Poppendieck, Lean Software Development: An Agile Toolkit, Addison-Wesley-Professional, 2003.

[15] M. Richards, "Microservices Architecture Pattern," in *Software Architecture Patterns*, 2015.

[16] C. Richardson, "Pattern: Microservice Architecture," [Online]. Available: https://microservices.io/patterns/microservices.html.

[17] nginx, "API Gateway," NGINX, [Online]. Available: https://www.nginx.com/learn/api-gateway/.

[18] Microsoft, ""Health Endpoint Monitoring pattern"," Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/patterns/health-endpoint-monitoring. [Accessed 2020].

[19] ""Pattern: Server-side service discovery"," microservices.io, [Online]. Available: https://microservices.io/patterns/server-side-discovery.html. [Accessed 2020].

[20] ""Pattern: Client-side service discovery"," microservices.io, [Online]. Available: https://microservices.io/patterns/client-side-discovery.html. [Accessed 2020].

[21] ""Pattern: Database per service"," microservices.io, [Online]. Available: https://microservices.io/patterns/data/database-per-service.html. [Accessed 2020].

[22] ""Valet Key pattern"," Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/patterns/valet-key. [Accessed 2020].

[23] ""Access Token"," microservices.io, [Online]. Available: https://microservices.io/patterns/security/access-token.html. [Accessed 2020].

[24] ""Circuit Breaker pattern"," Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker. [Accessed 2020].

[25] ""Gatekeeper pattern"," Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/patterns/gatekeeper. [Accessed 2020].

[26] ""Pattern: Distributed tracing"," microservices.io, [Online]. Available: https://microservices.io/patterns/observability/distributed-tracing.html. [Accessed 2020].

[27] J. Bloch, "How to Design a Good API and Why it Matters," *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications,* pp. 506-507, 2006.

[28] B.-L. T., R. Fielding, W. and e. al.., " Uniform Resource Identifier (URI): Generic Syntax," IETF, internet standards, 2005.

[29] R. N. T. ROY T. FIELDING, "Principled Design of the Modern," *ACM Transactions on Internet Technology,*, Vols. Vol. 2, No. 2, p. Pages 118–120, May 2002, .

[30] M. Amundsen, "Understanding Hypermedia," in *Building Hypermedia APIs with HTML5 and Node*, November 2011.

[31] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures".

[32] L. Li and W. Chou, "Desing and Descibe REST API without Violating REST: A Petri Net Based Approach," *IEEE International Conference on Web Services*, 2011.

[33] The GraphQL Foundation, "GraphQL," 2018 June 2020. [Online]. Available: http://spec.graphql.org/June2018/#sec-Overview. [Accessed June 2020].

[34] Microsoft, "Architecting Cloud Native .NET Applications for Azure," 2020. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/grpc. [Accessed 06 2020].

[35] K. Sandoval, "When to Use What: REST, GraphQL, Webhooks, & gRPC," 2018. [Online]. Available: https://nordicapis.com/when-to-use-what-rest-graphql-webhooks-grpc/. [Accessed 2020].

[36] W3C, "SOAP Version 1.2 Part 0: Primer," 04 2007. [Online]. Available: https://www.w3.org/TR/2007/REC-soap12-part0-20070427/. [Accessed 06 2020].

[37] T. F. R. a. L. M. Berners-Lee, "Uniform Resource Identifier (URI): Generic Syntax, RFC 3986," January 2005. [Online]. Available: https://tools.ietf.org/html/rfc3986.

[38] Robert Bosch GmbH, CI/DA_REST, "Web and Mobile Development Framework, Development Guidelines - RESTful APIs," [Online]. [Accessed March 2020].

[39] K. W. M. S. J. S. Paul Krajewski, "Development Guidelines- RESTful APIs," CI/DA_REST, Rober Bosch GmbH.

[40] C. Risi, "Amazon reveals how it repelled the biggest ever DDoS attack in February," June 2020. [Online]. Available: https://www.criticalhit.net/technology/amazon-reveals-how-it-repelled-the-biggest-ever-ddos-attack-in-february/.

[41] WASC Threat Classification, "XML Injection, WASC-23," [Online]. Available: http://projects.webappsec.org/w/page/13247004/XML%20Injection#:~:text=XML%20Injection%20is%20an%20attack,intend%20logic%20of%20the%20application..

[42] WASC Threat Classification, "XPath Injection, WASC-39," [Online]. Available: http://projects.webappsec.org/w/page/13247005/XPath%20Injection#:~:text=XPath%20Injection%20is%20an%20attack,query%20or%20navigate%20XML%20documents..

[43] The OWASP® Foundation, "OWASP API Security Project," 2019. [Online]. Available: https://github.com/OWASP/API-Security/raw/master/2019/en/dist/owasp-api-security-top-10.pdf. [Accessed May 2020].

[44] Cloudflare, Inc, "What is a WAF? | Web Application Firewall explained," [Online]. Available: https://www.cloudflare.com/learning/ddos/glossary/web-application-firewall-waf/. [Accessed June 2020].

[45] PlantUML, [Online]. Available: https://plantuml.com/.

[46] "Microservices at Netflix," NGINX, [Online]. Available: https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/.

[47] K. Sandoval, "A Tale of Four API Designs: Dissecting Common API Architectures," 11 June 2015. [Online]. Available: https://nordicapis.com/a-tale-of-four-api-designs-dissecting-common-api-architectures/. [Accessed 06 2020].

[48] R. Mitra, "Five API Styles," Jan 2018. [Online]. Available: https://www.slideshare.net/ronniemitra/five-api-styles.

[49] SSL Support Team, "What is SSL?," October 2019. [Online]. Available: https://www.ssl.com/faqs/faq-what-is-ssl/.

[50] CIENA, "What Is Service Orchestration?," Ciena Corporation, [Online]. Available: https://www.ciena.com/insights/what-is/what-is-service-orchestration.html.

[51] Microsoft, ""Cloud Design Patterns"," Microsoft, 2018. [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/patterns. [Accessed 2020].

[52] Rober Bosch GmbH, CI/DA_REST, "Development Guidelines - RESTful APIs," Robert Bosch GmbH