Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Enabling multi-tenant scalable IoT platforms

Muhammad Ismaiel

Course of Study:	Computer Science
Examiner:	Prof. Dr.-Ing. habil. Bernhard Mitschang
Supervisor:	Dr. rer. nat Pascal Hirmer

Commenced: Sep 20, 2019

Completed: Mar 20, 2020

Acknowledgment

First of all, I would like to thank my thesis advisor Dr. rer. nat Pascal Hirmer of the IPVS at the University of Stuttgart for his helpful remarks and comments during this master's thesis. His supervision benefited me throughout this thesis' research and writing.

I would like to express my thorough gratitude to my father, my mother, my brother, and my sister, for providing me with continuous help and support throughout my years of education. Without them, this achievement would not have been possible.

I extend my heartfelt gratitude to Waseem, Sohaib, Moiz, and Awais. I greatly appreciate their encouragement, inspiration, and affection.

Finally, I would like to thank my wife for her love and unwavering support. Thank you very much for being my best friend.

Abstract

Internet of Things (IoT) platforms have multi-layer architectures that facilitate the provisioning, automation of connected devices, and monitoring. These platforms simplify development because they solve a lot of the problems and complexities in building an IoT application. IoT platforms are typically central components with many heterogeneous users, which leads to the effect that these platforms are essential for IoT scenarios and must not become a single point of failure. Furthermore, since many users access these platforms, scalability and multi-tenancy are crucial. The scalability of IoT platforms makes user applications stable and more comfortable to extend. In contrast, the multi-tenancy trait allows multiple tenants to access user applications at the same time.

In this thesis, we examine every software stack layer of an IoT platform and explain different methods to make each layer scalable. Enabling multi-tenancy at the application layer and attaining scalability of message brokers, databases, the middleware as a whole, network layer, and device layer are the primary tasks for enabling multi-tenant scalable IoT platforms. To provide a solution to these tasks, we go through the research work already done in the area of scalability and multi-tenancy. We address multiple solutions for each task and also provide the best suitable solution for each task at the corresponding layer of an IoT platform.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

List of Abbreviations

COAP Constrained Application Protocol. 17

DBMS Database Management System. 21

IoT Internet of Things. 4

MBP Multi-purpose Binding and Provisioning Platform. 17

MQTT Message Queuing Telemetry Transport. 7

OPC-UA Open Platform Communications Unified Architecture. 17

OS Operating System. 21

RDBMS Relational Database Management System. 21

RL Reinforcement Learning. 53

RP Retention Policy. 40

SaaS Software as a Service. 27

TSD Timeseries Data. 17

TSDB Timeseries Database. 17

VM Virtual Machine. 19

1 Introduction

In the IoT, interconnected devices equipped with sensors and actuators communicate with each other through standardized network protocols to reach common goals [GBF+18]. Famous examples of IoT environments are Smart Homes [RMD+06], Smart Cities [SKP+11], or Smart Factories [LCW08]. Most IoT environments have a central component called an IoT platform, which is responsible for the management of the IoT devices, as well as their equipped sensors and actuators. These IoT platforms usually support different standards, such as MQTT, Constrained Application Protocol (COAP), or Open Platform Communications Unified Architecture (OPC-UA), for data exchange and configuration. IoT platforms are typically central components with many heterogeneous users, i.e., the IoT devices, IoT applications, and human users that configure devices or view the data through dashboards. This leads to the effect that these platforms are essential for IoT scenarios and must not lead to a single point of failure. Furthermore, since many users access these platforms, scalability and multi-tenancy are very important.

Multi-tenancy means using shared services and resources for multiple clients. These resources can include hardware resources (such as CPU, storage) and networking [KCD+17]. In a multi-tenant environment, tenants' data is isolated and invisible to other tenants. Data isolation is key to achieve multi-tenancy at the application level. There are different data isolation strategies [ABG+18], but we suggest a shared database with a shared schema for the platform because data coming from IoT devices are more or less similar. Scalability is the ability of software to adjust itself dynamically according to the workload. Vertical and Horizontal scaling are the two generic ways of achieving scalability [LML12]. In vertical scaling, hardware resources of a machine are upgraded and degraded according to the need while more machines are added or removed in case of horizontal scaling. There are several auto-scaling strategies available that could be used for deciding when to scale out and when to scale in. These strategies can be used for either reactive scaling or proactive scaling [LML14]. Auto-scaling actions are based on some performance indicators. CPU and memory utilization are the major performance factors, along with currently active sessions and response time. For the platform Multi-purpose Binding and Provisioning Platform (MBP) [HBS+16] [HWBM16a], we implement a monitoring system using a static threshold scaling strategy and use CPU and memory utilization, and active sessions as scaling indicators for the scalability of the middleware [MSA14].

Timeseries Data (TSD) is data gathered over time consisting of a series of observations (e.g., sensor data, data from smart grids). Timeseries Databases (TSDBs) are being used to store and handle TSD [BKF17]. In terms of consistency, accessibility, schema-less layout and immutability, Druid and InfluxDB provide mostly comparable characteristics.

InfluxDB, however, supports more programming language interfaces than Druid. InfluxDB also has a Time Structured Merge Tree engine that increases ingestion and compression of data at higher rates [RPHB18]. InfluxData offers a high-availability open-source replication solution while many users use sharding in the open-source version as a workaround for the missing clustering feature.

Messaging is a way of exchanging messages between multiple endpoints. It is used for decoupling of software components. The message sender is called producer in the messaging middleware term, and the message recipient is called a consumer. A Message broker sends messages from one application to another using point-to-point or publish/subscribe messaging models [Mag15]. The publish/subscribe model is best to be used by a message broker because it provides a one-to-many relationship between publishers and subscribers. In the scope of this thesis, we implement a clustering of message brokers for the management of a massive amount of IoT devices [JPV+17]. A gateway is used at the network layer, which provides an interface to connect the devices to other systems. It also provides a mechanism for translating between diverse protocols, payload formats, and communication technologies [GBF+18]. The scalability of this gateway improves communication latency between devices and middleware. The network may become the bottleneck because the network bandwidth constrains the data flow. If an IoT platform is delivering/receiving a higher volume of data than what is recommended by the existing capacity of the network, then a network bottleneck will occur. The device is a hardware component that has its storage and processor. Achieving gateway scalability also satisfies our goal for the scalability of devices.

This thesis is organized as follows. In Chapter 2, we address related work of the scalability of IoT platforms and its importance for this thesis. The main goals of this thesis are discussed and compared in Chapter 3. The best solution for each task at the corresponding layer of an IoT platform is also defined in this Section. The implementation part of this thesis is discussed in Chapter 4 while we present our summary and future directions in Chapter 5.

2 Related Work

In this chapter, we address work related to the scalability of IoT platforms, e.g., regarding middleware components, message brokers, gateways, and databases. We also review the methods for the replication of databases and the multi-tenancy functionality of applications.

Scalability describes how a system copes with an increasing workload by provisioning of resources. Lorido-Botrán et al. [LML12] explain two approaches for achieving scalability which are 1. vertical scaling and 2. horizontal scaling. Vertical scaling is the process of adding more computing power (CPU, memory, disk) to the hosting machine while horizontal scaling is achieved by introducing additional hosting machines (e.g., Virtual Machines (VMs)).

Qu et al. [QCB18] introduce a study of auto-scaling approaches for web applications. In their work, auto-scaling frameworks' actions are based on high or low-level performance factors. CPU and memory utilization are low-level performance indicators monitors at the server layer. While high-level performance factors, for example, response time, active sessions, are monitors at the application layer. Auto-scaling techniques presented by Lorido-Botrán et al. [LML12] derive scaling decisions based on algorithmic analysis of current workload parameters. Auto-scaling procedures are characterized into five classifications about cloud-based applications. Static threshold-based rules, queuing theory, time series analysis, control theory, and reinforcement learning are the primary techniques that have been proposed. A survey of these auto-scaling strategies has been introduced by Lorido-Botrán et al. [LML14].

Chieu et al. [CMKS09] present an architecture to scale web applications dynamically in a virtualized cloud computing environment. Their architecture includes several VMs, an Apache HTTP load balancer, and monitoring and provisioning sub-systems, as shown in Figure 2.1. This work considers application-specific scaling indicators, such as the number of current users and the number of currently active sessions for each instance of an application. The work uses a monitoring agent running in each application instance, which forwards the scaling indicators to the dynamic scaling algorithm. This algorithm arbitrates between monitoring and provisioning sub-systems. Depending on the statistics and moving average of the scaling indicators, the algorithm triggers the provisioning or de-provisioning event in the provisioning sub-systems. However, this work does not consider CPU, memory, and disk usage of the machine hosting the VMs.

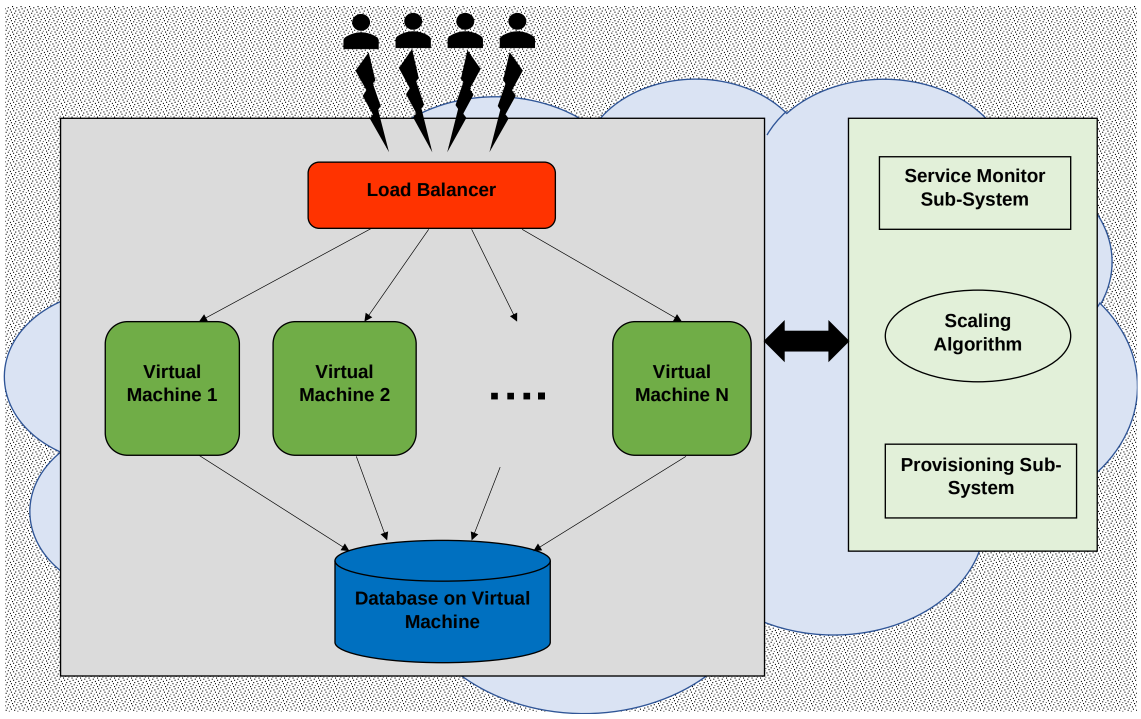

Figure 2.1: Architecture of the auto-scaling approach in the Cloud [CMKS09]

Hung et al. [HHL12] propose a similar kind of cloud computing auto-scaling architecture. Their work contains a cluster monitoring system, a front-end load balancer, and an auto-provisioning system. The purpose of a front-end load balancer is to send and balance incoming requests to different application instances running in a virtual cluster. A cluster monitoring system monitors the resource utilization of each host machine in a cluster. The auto-provisioning system is responsible for the provisioning of VMs, depending on the resource usage or currently active sessions. The authors also present an algorithm for auto-scaling, which is being utilized by the auto-provisioning system.

Murthy et al. [MSA14] describe an architecture for threshold-based auto-scaling of virtual machines. The architecture includes a configuration file, monitoring, and decision-maker components, several virtual machines, and a VM configuration component. The configuration file contains a predefined static upper and lower threshold values. The monitoring component performs the monitoring of resource utilization of the virtual machines. A decision-maker receives data about resource utilization as input from the monitoring component. The decision-maker determines performance metrics using threshold values given in the configuration properties file. The VM configuration component performs provisioning or de-provisioning of VMs according to the input from the decision-maker.

In a multi-tenant architecture, a single instance of the software is deployed on the server functioning on the service provider infrastructure. It provides access to multiple tenants at the same time, as describe by Karataş et al. [KCD+17]. Abdul et al. [ABG+18] describe several multi-tenant data layer models, such as dedicated, isolated, shared, and hybrid. By sharing infrastructure, middleware, or application among tenants, multi-tenancy can be achieved. Figure 2.2 visualizes the multi-tenancy strategies described by Walraven et al. [WLJ14]. However, application-level multi-tenancy with a shared-everything architecture can lead to cost reduction. Chong et al. [CCW06] have outlined several viable database patterns, primarily for Microsoft SQL Server, which supports application-level multi-tenancy. In a multi-tenant situation, they present three major strategies which are: distinct databases, shared databases with distinct schemes, and shared databases with shared schemes.

Bezemer et al. [BZ10] define an architectural strategy that seeks to distinguish multi-tenant configuration and internal execution. The authors implement a three-tiered architecture (authentication, setup, database). In the same way, experiences have been recorded throughout the transition of single-tenant industrial software systems to multi-tenant software by Bezemer et al. [BZP+10]. Also, a reengineering pattern is described by the authors to assure multi-tenancy in software systems. This pattern requires three additional components: a multi-tenant database, tenant-specific authentication, and configuration. However, the configuration just serves the look-and-feel purposes of applications and workflows.

There are typically two sorts of multi-tenancy styles: multiple instances and native multi-tenancy. The various instances style supports every tenant with its committed application over shared hardware, Operating Systems (OSs), or a middleware server in a hosting environment. The latter can assist all tenants through a single shared application instance over numerous hosting resources, as describe by Guo et al. [GSH+07]. The authors also describe a mechanism for the development and management of multi-tenant applications. They believe that the primary challenge of multi-tenancy is tenant isolation. Thus, their framework includes mainly tenant isolation components, such as data, performance, and security isolation.

TSD is data gathered over time consisting of a series of observations (e.g., sensor data, data from smart grids). TSDBs are being used to store and handle TSD, as mentioned by Bader et al. [BKF17]. The authors perform a comprehensive research and comparison that identifies 83 TSDBs. They compare 12 popularly chosen open source TSDBs with 27 criteria. Open source TSDBs have been divided into three groups: TSDBs depending on any existing Database Management Systems (DBMS), TSDBs independent of any DBMS, and Relational Database Management Systems (RDBMS). They infer that not one TSDB suits every case of use but highlighted InfluxDB [Inf], MonetDB [Mona], and Druid [Dru] as the feature richest.

InfluxDB, Druid, and SiriDB [Sir] are investigated by Ravichandran et al. [RPHB18] as an alternative from TSDBs for their research. SiriDB and Druid were excluded for quantitative comparison among these three databases. The reason for this is that SiriDB is a new database that lacks some characteristics, such as supporting programming languages, e.g., C, C++ and Java, community support, and documentation. In terms of consistency, accessibility, schema-less layout and immutability, Druid and InfluxDB provide mostly

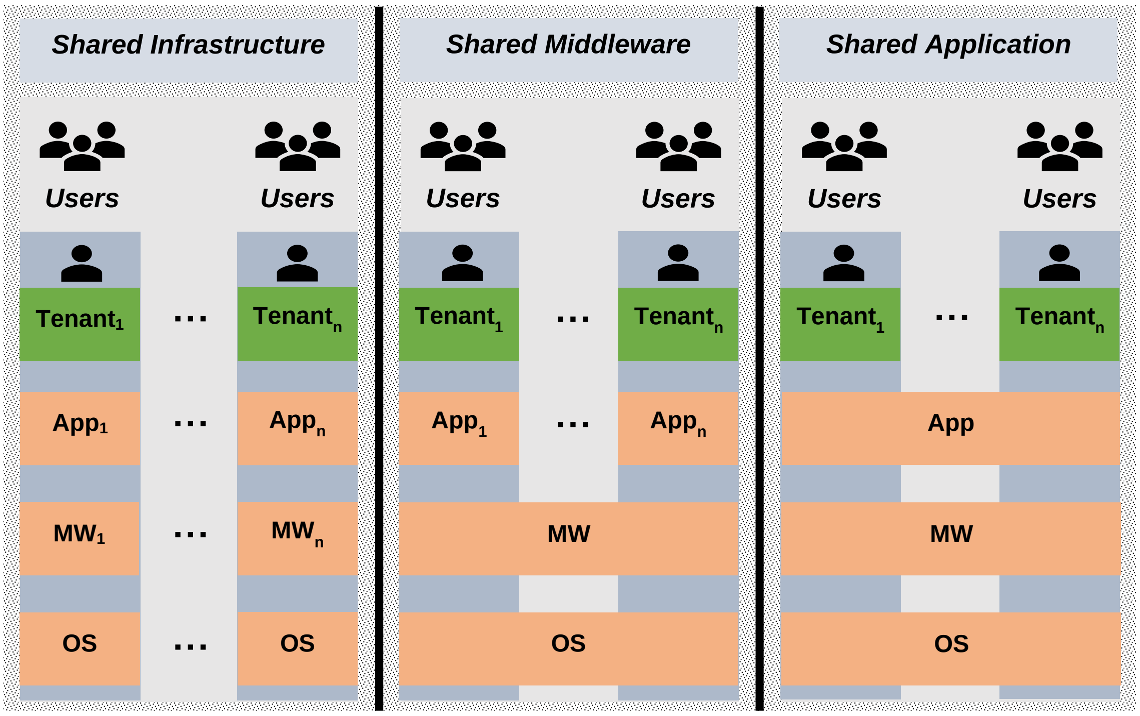

Figure 2.2: Multi-tenancy architectural strategies [WLJ14]

comparable characteristics. InfluxDB, however, supports more programming language interfaces than Druid. InfluxDB also has a Time Structured Merge Tree engine that increases ingestion and compression of data at higher rates.

In the context of IoT, Arnst et al. [AHPW19] conduct a comparative survey on the reading and writing performance of three common DBMS. The authors chose MongoDB [Monb], InfluxDB, and MariaDB [Mar] for their study. They rate InfluxDB higher than others and endorsed it as a suitable DBMS for handling high recurrent IoT sensors data. Sanaboyina [San16] records OpenTSDB [Ope] and InfluxDB performance assessments based on data center energy usage. The study hypothesizes that the database application energy is directly proportional to its use of resources (RAM, Disk, and CPU). They observe InfluxDB be somewhat more energy effective than OpenTSDB in both single-node and distributed database environments.

Mangoni [Mag15] provides an overview of messaging concepts, communication protocols, functionalities, and advanced distributed communications systems. The author compares and analyzes their weaknesses and strengths. This research includes most famous messaging technologies which are: ActiveMQ [Apa], RabbitMQ [Rab], Apache Kafka [Kaf], and ZeroMQ [Zer]. Mangoni rated Apache Kafka as the best technology for data movement among processing systems. John et al. [JL17] and Dobbelaere et al. [DE17] also compare two leading messaging technologies, Kafka and RabbitMQ, in their respective studies. They explore the variety of their characteristics and architectures along with their efficiency under diverse testing workloads. Both studies conclude their research by ranking Kafka

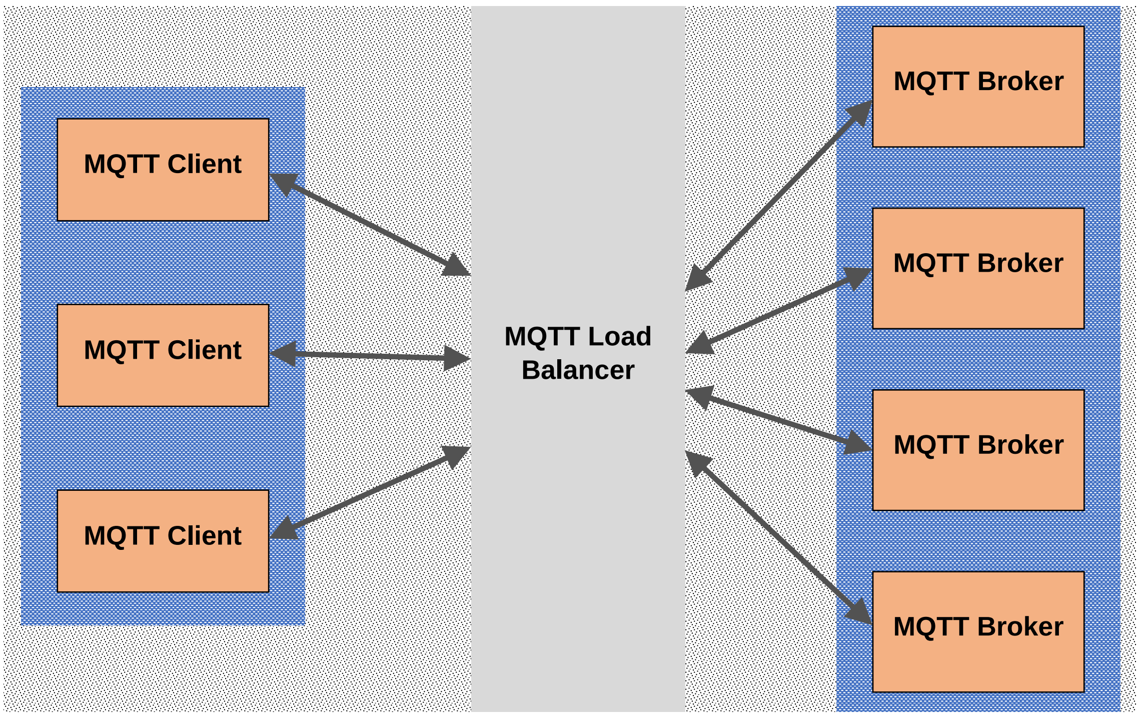

Figure 2.3: Scalable message broker architecture [JPV+17]

ahead of RabbitMQ on a throughput basis and rated RabbitMQ better when it comes to latency. Dobbelaere et al. outline that if the configuration consists of a single node, a single partition, a single channel, and no replication, then RabbitMQ also performs better than Kafka in terms of throughput. Apache Kafka is the most advanced message broker. It has been created to address the need to transfer enormous amounts of data from producers to numerous consumers in an efficient and scalable manner. Kreps et al. [KNR+11] perform an experimental study. It discloses that Kafka could publish messages twice and consumes four times more messages than RabbitMQ and ActiveMQ in a second.

For the management of a massive amount of IoT devices using message brokers, Jutadhamakorn et al. [JPV+17] implement a scalable and low-cost clustering architecture. Figure 2.3 visualizes an architecture that is quite similar to the approach discussed by Sen et al. [SB18] for scalability of message brokers. The architecture includes a MQTT broker cluster at the backend and a load balancer in front of brokers. A load balancer is used for the distribution of messages within the cluster. Docker Swarm [Doc] is used to build the cluster and Mosquitto [Lig+17] on each broker in a cluster.

3 Enabling multi-tenant scalable IoT platforms

In this chapter, we discuss the necessary tasks to enable scalability in each architectural layer of the MBP [HWBM16b]. By doing so, we present different possible solutions for each task. Furthermore, we suggest a solution that is most suitable for the MBP. Figure 3.1 exhibits the different layers of the MBP.

The application layer represents software that seeks sensor data and uses actuators for supervising physical actions. It interacts with the middleware [HWS+16] layer to attain insight and/or exploit the real world. The middleware layer acts as an integral component that is responsible for receiving and processing data from the devices. It also serves the data to the connected applications. A gateway is used at the network layer, which provides an interface to connect the devices to other systems. It also provides a mechanism for translating between diverse protocols, payload formats, and communication technologies. The device is a hardware component that has its storage and processor. It forms a

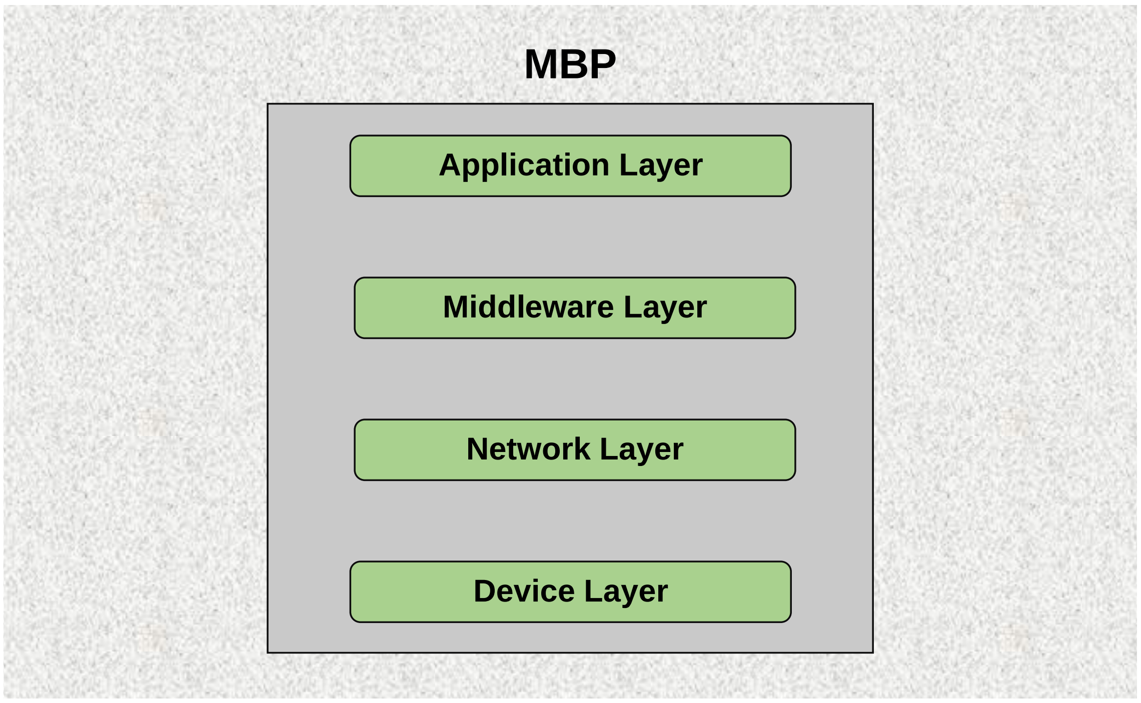

Figure 3.1: MBP Layers

Functionalities of Layers		
#	**Layer**	**Functionalities**
1	Application	Software (Smart Grid App, Intelligent Transport Systems App, etc.)
2	Middleware	Message Broker
		Data Storage
3	Network	Gateway / Communication
4	Device	Hardware Component (connected to sensors, actuators, and middleware)

Table 3.1: Functionalities of MBP layers

connection with the middleware using driver software running on it. Devices are the physical environment's point of entry into the digital world. The sensors and actuators are the hardware components connected to devices. A sensor detects the physical environment data, such as humidity, sound, or light, while the actuator manipulates the physical environment [GBF+18]. Table 3.1 illustrates the functionalities of each individual layer of the MBP.

The major goals of this thesis are to create a generic concept to enable the scalability and multi-tenancy of an IoT platform. Each layer of the platform should be scalable in such a way that a single point of failure is avoided. Enabling applications to be multi-tenant is our goal for the application layer. Multi-tenancy enables an application to serve multiple users at the same time [BZP+10]. At the middleware layer, the scalability of the message broker and database should be achieved. Data replication and scalability of the middleware as a whole are also our significant tasks for this layer. Moreover, the scalability of gateways is the main task for the network layer. This improves communication latency between devices and middleware. Regarding the device layer, we aim at supporting unlimited devices, which is dependent on the gateway scalability. Achieving gateway scalability also satisfies our goal for the device layer. Table 3.2 explains the significant requirements which we need to fulfill for each layer.

In the following, we look into all layers individually and discuss each requirement in detail. We examine and compare different ways of fulfilling each goal of every layer. Furthermore, we propose which approaches are most suitable for the MBP platform.

3.1 Application Layer

The application layer uses the middleware to manipulate the real world (cf. Figure 3.1) by demanding data from the sensors and by using actuators to control physical actions. For example, a software system that controls a building's temperature represents a

Requirements of Layers			
#	**Layer**	**R#**	**Requirements**
1	Application	R1.1	Multi-tenancy (enable application to serve multiple tenants at the same time)
2	Middleware	R2.1	Scalability of Message Broker
		R2.2	Scalability of Database and Replication
		R2.3	Scalability of Middleware as a whole
3	Network	R3.1	Scalability of Gateway to avoid bottleneck issue
4	Device	R4.1	Support of unlimited Devices

Table 3.2: Requirements of MBP layers

middleware-connected application [GBF+18]. Our main goal at the application layer is to attain multi-tenancy enabled software. In Section 3.1.1, we discuss different generic multi-tenancy architectures and choose the best-suited one for the MBP.

3.1.1 Multi-tenancy

Multi-tenancy is one of cloud computing's key features. It is essential to know the complete variety of tenancy alternatives for bringing cloud functionality into any application or platform. Choosing a tenancy architecture for the application or platform depends on the enterprise's needs. A rational explanation of multi-tenancy is that it is a paradigm of software architecture in which various clients concurrently use a single application. However, when an application has a single client, it is prevalent for companies to use a single-tenant architecture [ABG+18]. Figure 3.2 visualizes the distinction between multi-tenant architectures and single-tenant architectures. The multi-tenant architecture allows various clients to access the same application server and database at the same time, while single-tenant applications are structured to provide each client with a dedicated application server and database.

In a multi-tenant environment, a tenant comprises a group of customers of an organization. A tenant uses a set of Software as a Service (SaaS) functionalities to attain organizational objectives. In a cloud paradigm, each tenant is ignorant of other tenants. The benefits of using multi-tenant architecture are 1. simplicity to add a new tenant, 2. maintaining the same application becomes more convenient, 3. maximization of the use of resources, 4. handling of multiple tenants at the same time. The disadvantages of this architecture may include those related to 1. confidentiality of the data, and 2. security of the data. However, one can create and define a multi-tenancy model at the data layer to address these issues. Dedicated Schema, Shared Schema, Isolated Schema, and Partially Isolated are the type of models which can be used at the data layer.

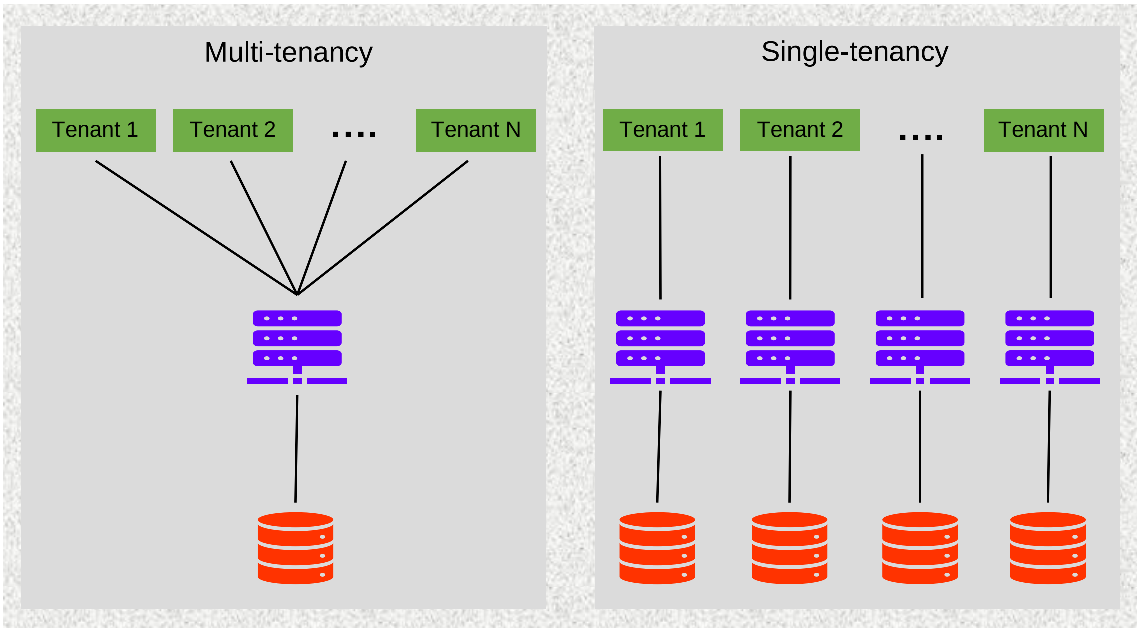

Figure 3.2: Multi-tenancy Vs Single-tenancy [KCD+17]

Multiple instances and native multi-tenancy are two architectural strategies to attain multi-tenancy [GSH+07]. Multi-tenancy can be realized at the Application level, Middleware level, and Infrastructure level of the software stack as listed below.

1. Multiple Instances Multi-tenancy
    - Infrastructure Level Multi-tenancy
    - Middleware Level Multi-tenancy
2. Native Multi-tenancy
    - Application Level Multi-tenancy

Middleware and infrastructure level multi-tenancy provides each tenant with its separate application instance known as *multiple instance multi-tenancy*. Application-level multi-tenancy supports all tenants through a single shared application instance over a hosting infrastructure, called *native multi-tenancy*. Multiple-instance technique is adopted to support dozens of tenants. Native multi-tenancy is used to support a significantly larger number of tenants, usually hundreds or even thousands. It is important to note that as the level of scalability raises, the amount of isolation among tenants declines. Because of various and/or all levels of the software stack which are shared among tenants. In such cases, tenant isolation is performed at the data level.

Recently, the software industry has progressed from a rigorous hardware split between tenants (shared-nothing architecture) to a growing point of sharing between tenants (shared everything architecture). The business requirements are key to determine which multi-tenancy model to use by taking each tenant's performance and efficiency into account. For

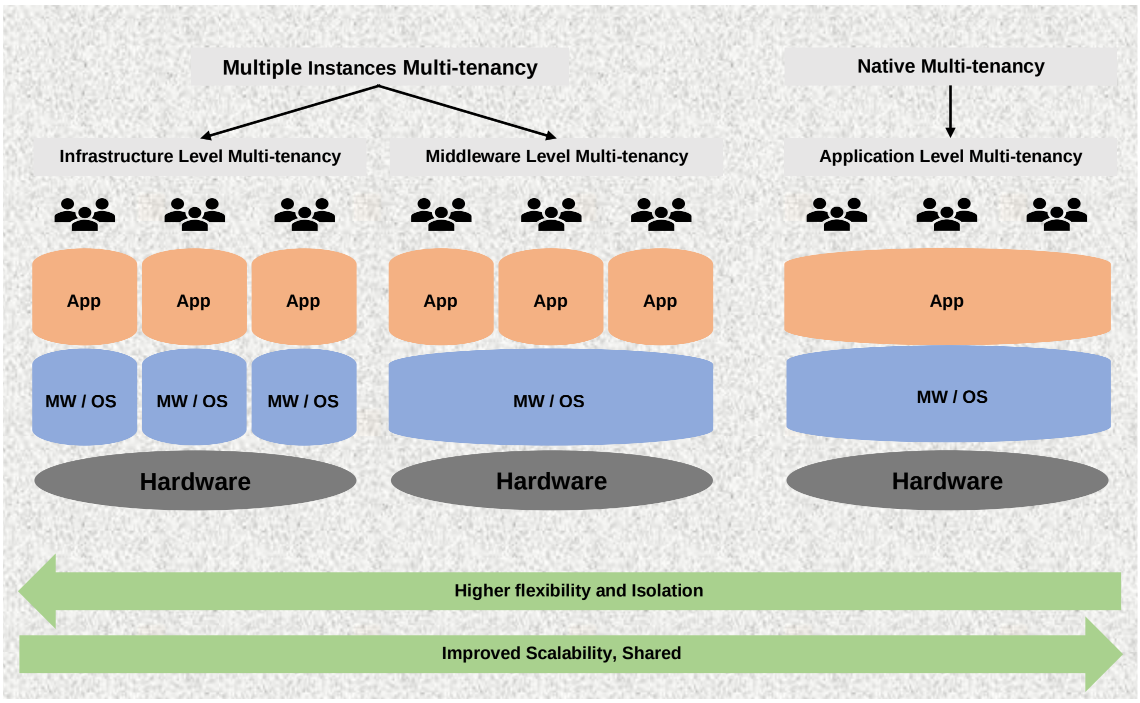
Multiple Instances Multi-tenancy
Native Multi-tenancy
Infrastructure Level Multi-tenancy
Middleware Level Multi-tenancy
Application Level Multi-tenancy
App
MW / OS
Hardware
Higher flexibility and Isolation
Improved Scalability, Shared

Figure 3.3: Multiple Instances and Native Multi-tenancy [GSH+07]

instance, a big corporation may willingly pay a surcharge for various instances to avoid potential resource sharing hazards. At the same time, most small and medium business companies would prefer services with reasonable quality at reduced expenses.

In the following, we briefly look into each of the multi-tenancy approaches mentioned above and decide which approach is suitable for the MBP according to its business needs. Section 3.1.2 explains the multiple instances multi-tenancy patterns while Section 3.1.3 discusses the native multi-tenancy pattern in detail.

3.1.2 Multiple Instances Multi-tenancy

Multiple instances multi-tenancy supports each tenant with its dedicated application instance over shared hardware, and/or a middleware server in a hosting environment. In this multi-tenancy architecture, some or no layers of the software stack are shared among tenants. Multiple instances multi-tenancy can be achieved using this approach by sharing infrastructure (virtualization) or middleware components (application server and/or database server). Multiple-instances architecture utilizes the ability of virtualization. It hosts the application code on different instances to meet the increase in resource demand. It is useful for multi-tenant applications when the number of tenants remains relatively small. Section 3.1.2.1 and Section 3.1.2.2 briefly describe infrastructure and middleware level multi-tenancy patterns, respectively.

Pros and Cons of Infrastructure Level Multi-tenancy	
Pros	**Cons**
Higher flexibility	Higher operational cost
Higher tenant isolation	Higher maintenance cost
Secure	Software licensing expense per tenant
Strong data isolation	Possible for limited clients
Easy to extend/restore tenant database	

Table 3.3: Pros and Cons of Infrastructure Level Multi-tenancy

3.1.2.1 Infrastructure Level Multi-tenancy

The infrastructure level multi-tenancy [WLJ14] technique is used to maximize scalability and resource usage by sharing infrastructure. Infrastructure includes physical/virtual hardware (e.g., Servers, VMware). It provides dedicated VMs to each tenant and all the requests of any tenant goes to its specific VM as shown in Figure 3.3. In this architecture, the tenant applications and databases are isolated at the instance level. Each VM entirely hosts a different tenant application, and no layer of the software stack is shared among tenants, except the hardware. This approach is a coarse-grained way of isolating the tenants. It can be used when each client is viewed as an entity without any business logic linked with it or when the data privacy and regulations are of the greatest interest. Table 3.3 shows the advantages and disadvantages of this architecture.

The advantages of infrastructure level multi-tenancy are that 1. it gives the highest amount of tenant isolation compared to other multi-tenancy patterns 2. it is very secure because nothing is shared among tenants except hardware 3. strong isolation of tenant's data 4. it can easily be used to extend or restore individual tenant database instances without interrupting other tenants. The disadvantages of this pattern are that 1. hosting distinct applications and middleware for each tenant is very expensive, 2. it is virtually impossible to establish for most SaaS providers due to the number of clients that they intend to serve, 3. additional costs, such as software licensing and development expenses per tenant will be significantly higher due to secure data isolation and customization.

3.1.2.2 Middleware Level Multi-tenancy

The middleware lies between the OS and the application in a distributed computing system. It is software that offers services other than those offered by the OS to software applications. Application and database servers are examples of such middleware software. For larger tenant sets, the VM-per-tenant strategy does not scale. The application provider further optimizes it by sharing the application server, and possibly also the database layers of the software stack [WLJ14]. However, the application is still deployed on top of the application

Pros and Cons of Middleware Level Multi-tenancy	
Pros	**Cons**
Moderate operational cost	Low flexibility
Moderate maintenance cost	Low tenant isolation
High data isolation if database is not shared	Low data isolation if database is shared
Support for maximum clients	Separate application for each tenant
Improved scalability	Less secure

Table 3.4: Pros and Cons of Middleware Level Multi-tenancy

server in separate application spaces or application domains to realize tenant separation, as demonstrated in Figure 3.3. This multi-tenancy pattern has several advantages over the infrastructure level multi-tenancy architecture, as shown in Table 3.4.

The advantages of the middleware level multi-tenancy pattern are 1. less operational and maintenance costs than infrastructure level multi-tenancy 2. it can support the maximum number of tenants 3. scalability can be achieved by scaling middleware 4. strong data isolation if the database is not shared among tenants.The disadvantages of this pattern are that 1. it gives a lower amount of tenant isolation and flexibility compared to infrastructure level multi-tenancy 2. hosting of separate application instances for each tenant 3. it is less secure compared to infrastructure level multi-tenancy because everything is shared among tenants except the application 4. low data isolation in case of a shared database among tenants.

3.1.3 Native Multi-tenancy

Native multi-tenancy supports all tenants by a single shared application instance over various hosting resources [GSH+07]. It is used to support more tenants than multiple instances multi-tenancy patterns, as discussed in Section 3.1.2. With this architecture, isolation among tenants decreases, but the scalability increases. Application-level multi-tenancy is the only way of fulfilling native multi-tenancy, as depicted in Figure 3.3. Section 3.1.3.1 explains the application-level tenancy architecture in detail.

3.1.3.1 Application Level Multi-tenancy

In application-level multi-tenancy, a single application serves the needs of multiple tenants at the same time. The application code is distributed among tenants, and a pool of deployed application instances can be shared between tenants to enhance scalability further. Segregation of tenants is tackled exclusively in the application code instead of virtualization or separate application instances for each tenant. Different tenant groups with multiple end-users access the same instance of the application running but getting a completely different experience potentially because of tenant isolation being done in the application

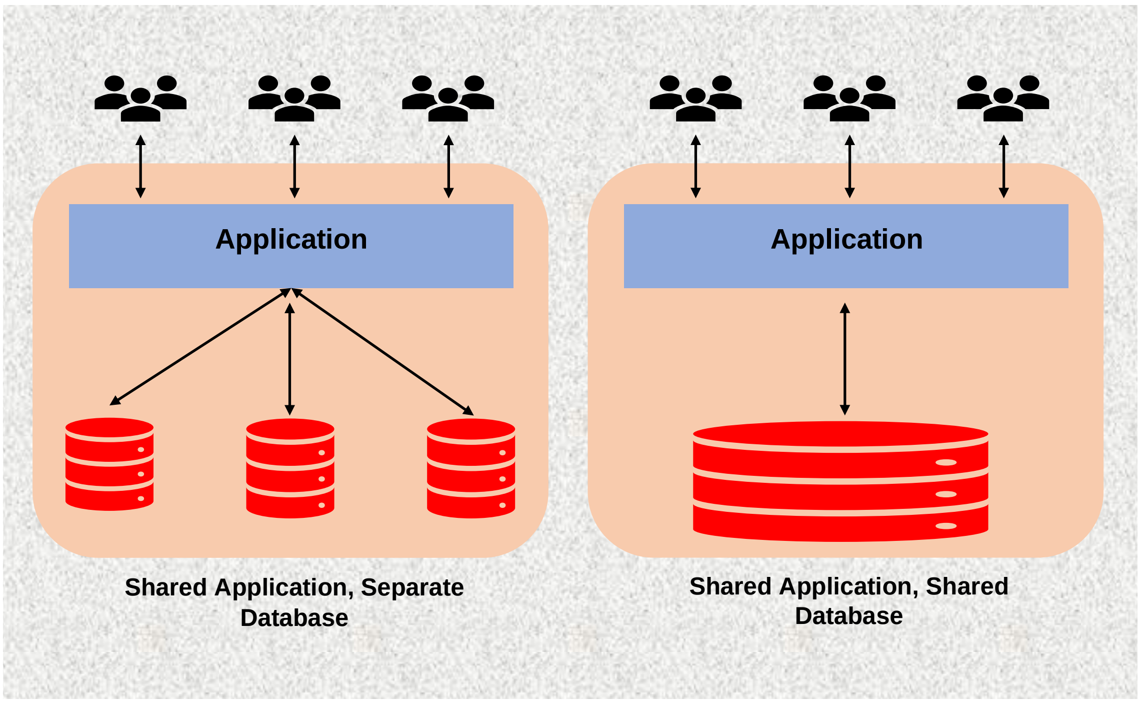
Application
Application
Shared Application, Separate Database
Shared Application, Shared Database

Figure 3.4: Data Storage Strategies [KCD+17]

code. Each tenant gets completely different data that is being isolated by the platform or application code underneath it. This multi-tenancy pattern is a very data-centric intelligent model. Data for all the tenants are being stored in a common database, but it is being served to the tenants very differently.

There are two vital tasks (data isolation and tenant-specific authentication) which we need to achieve for enabling multi-tenancy at the application level. These are discussed in detail in the following.

1. **Data isolation:** There is a high requirement for data isolation in a multi-tenant application. Since all tenants use the same instance of the database, it is necessary to ensure that only their data can be accessed. A separate database, shared database, and hybrid model are the data storage patterns to realize multi-tenancy at the data layer [ABG+18]. Data storage strategies are visualized in Figure 3.4.
   1. *Separate Database:* It is a dedicated model in which each tenant has its own physically separated database but sharing the application with other tenants. Each instance of the database entirely hosts specific tenant data, and nothing at the data level is shared among tenants. Storing tenants' data in separate databases is the simplest approach to data isolation. It is easy to modify the data model of the application to satisfy the individual needs of the tenants. Using this pattern, it is simple to restore a tenant's data from backups if any failure occurs. But it tends to result in higher costs for equipment maintenance and tenant data backup. The price

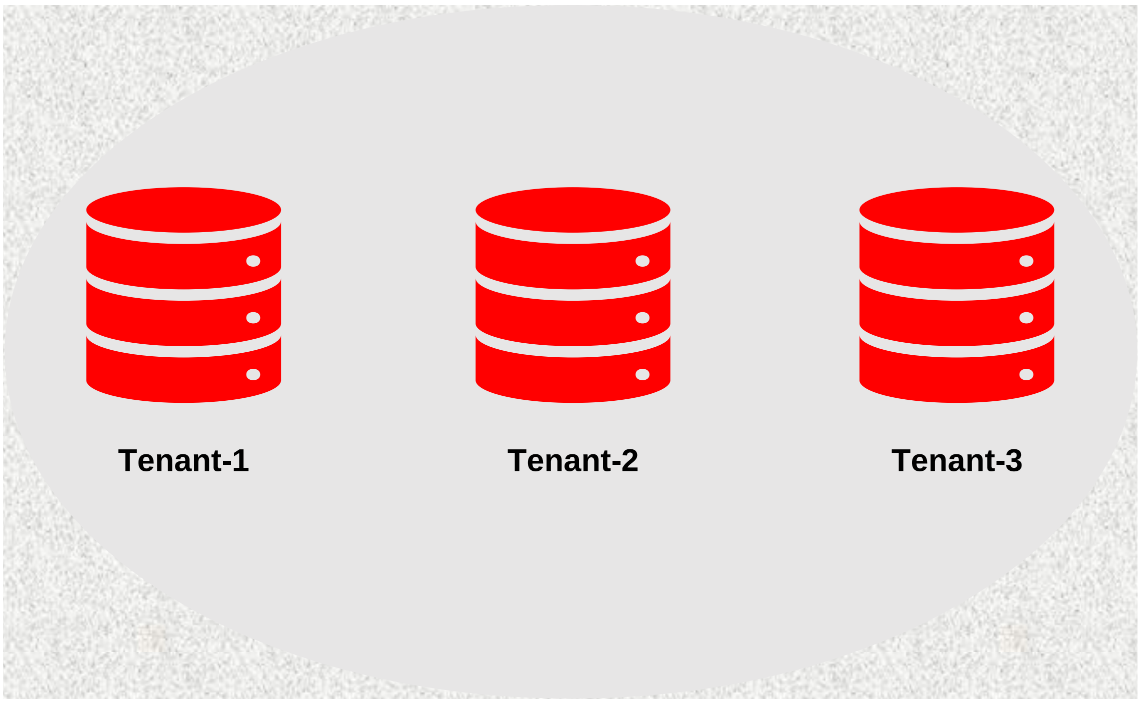

Figure 3.5: Separate Databases [CCW06]

of hardware is also lower than in alternative approaches because a database server can only support the limited total of tenants due to the limited number of databases that the server can support.
Segregation of tenants' data into separate databases is an expensive approach because of high maintenance requirements and hardware costs. It is relevant for the tenants that are happy to pay such a hefty amount for the benefit of customizability and added security. Tenants belonging to banking and health records management fields often have rigorous requirements for data isolation. They may not even rate an application that does not accommodate their data in separate database instances. This approach is shown in Figure 3.5.

2. *Shared Database:* In this strategy, each tenant uses a common database, but it is further divided into two models. These are Separate Schema and Shared Schema, which we discuss in detail in the following.
   - *Separate Schema:* In this approach, multiple tenants' data is kept in the same database, with each tenant having its own schema. A schema is a group of tables that are created specifically for each tenant. Each tenant schema is separated from other tenant schemas, as shown in Figure 3.6. Data is moderately isolated on a schema level rather than using instance-level isolation. A set of tables is generated and organized into a schema, and authorization is granted for each new tenant. The application can then customize and design tables using the schema name and table name. This approach is quite easy to implement, and

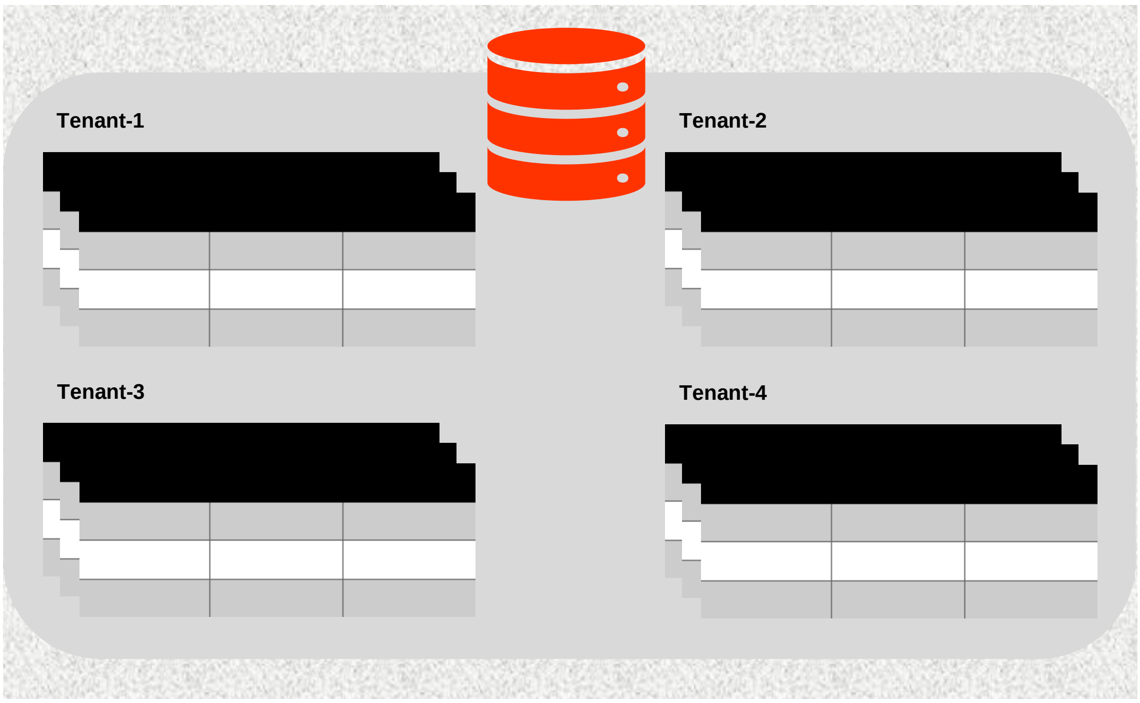

Figure 3.6: Shared Database, Separate Schema [KCD+17]

tenants can broaden the data model as conveniently as with the separate database approach. It is very convenient where the schema of all tenants is different, and usually, schema design is needed to be modified. It lessens the costs of a separate database approach and the security drawback of the shared schema model. It is also relevant for the applications that use a comparably meager number of database tables (100 tables per tenant or less). It can typically entertain more tenants per server than the separate database strategy can. The application cost is low as long as tenants agree with their data being stored in a shared database instance with other tenants. The disadvantages are that a large number of tables must be kept, and it is tough to restore tenants' data in the situation of a breakdown. If each tenant has its database, restoring a single tenant's data means directly recovering the database from the most up-to-date backup. But restoring the entire database will require overwriting each tenant's data.

- *Shared Schema:* In this approach, both the database and schema are shared for hosting every tenants' data. This model yields the most productive use of hardware with the slightest maintenance and usage costs. It is useful when the tenants are willing to share the application components, specifically the functionality at the data layer. Data isolation is achieved at row level within tables, as shown in Figure 3.7. A group of tables is designed to hold all the data of all the applicant tenants. It is also needed to design a security mechanism of the databases to prevent tenants from accessing other tenant's data. Tenant identifiers are used to persist and access the tenant-related information

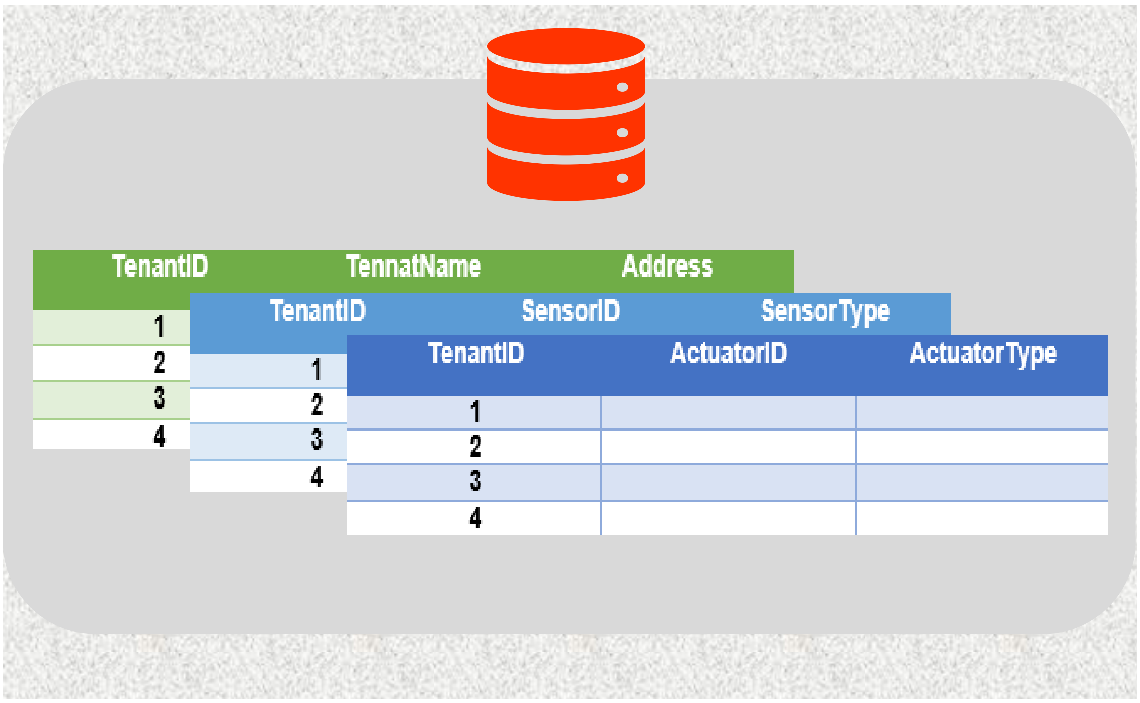

Figure 3.7: Shared Schema [CCW06]

within such tables. It assigns each row to the specific tenant. The shared schema approach is usually applied in applications where the tenants' data are strictly related. Because of the shared schema, the extensibility of data is a major challenge opposed to the separate schema pattern. This approach uses multi-tenant metadata and indexes to provide data extensibility. This involves a complicated schema and the development of dummy columns that break the normalization rules of relational databases. Nevertheless, in terms of hardware, software licenses, maintenance, and backup, it requires the lowest costs because one database instance serves multiple tenants. This model is ideal when the system is designed to support a significant number of tenants who are willing to give up strong isolation for the cheaper cost of use. In case of a failure, restoring tenants' data is done by recovering data from up-to-date backups, like the separate schema approach.

3. *Hybrid Model:* This model is a blend of different schema approaches and a shared schema. The hybrid model shares application/platform segments (e.g., user interface, database) among tenants that have identical features and functionality. It isolates the sections with specific or unrelated features. Common data, such as tenants' identification information, are stored in a single table or shared schema. Tenant-specific data are segregated at the schema level by defining a separate schema for every tenant, as illustrated in Figure 3.8.

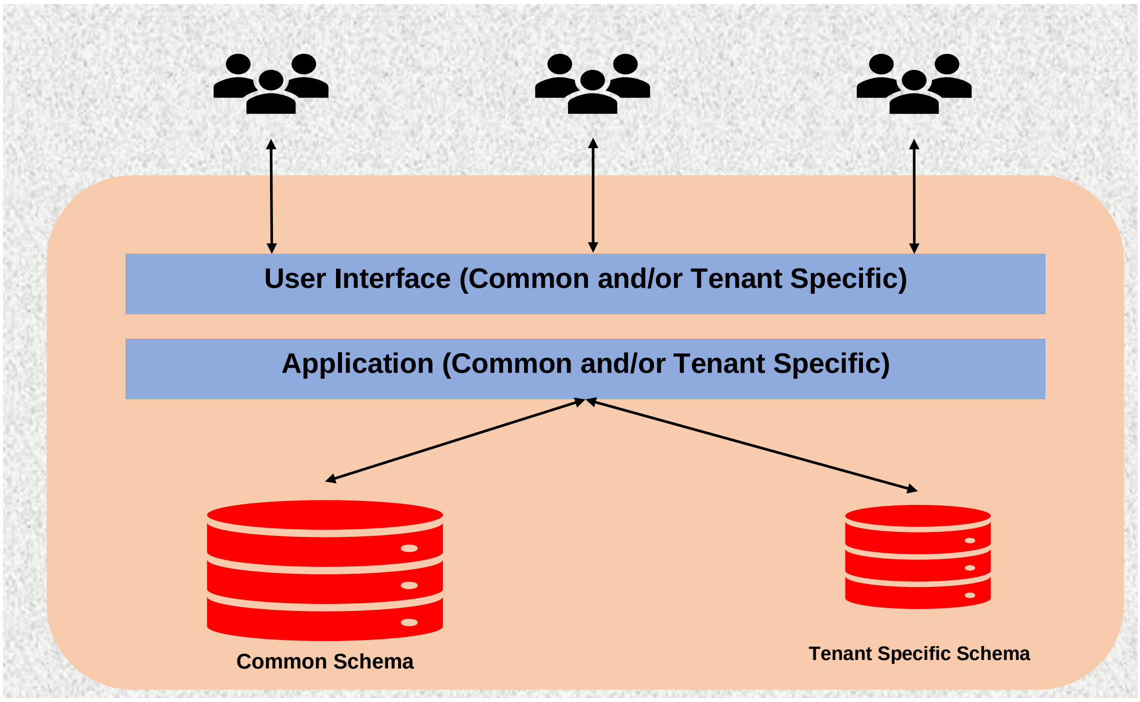
User Interface (Common and/or Tenant Specific)
Application (Common and/or Tenant Specific)
Common Schema
Tenant Specific Schema

Figure 3.8: Hybrid Model [ABG+18]

2. **Tenant-specific authentication:** In a native multi-tenant system, all tenants share the same application instances, infrastructure, and database instances. Therefore, isolation should be carefully considered in architectural design. To be able to offer the environment flexibility and to ensure tenants can only access their data, tenants must be authenticated. Each tenant is assigned a unique tenant id and a sub-tree in the organization directory isolated by tenant id during on-boarding. The tenant administrator can easily map user accounts of its private domain into the organization sub-tree. To authenticate the request, we can use the standard middleware security mechanism (e.g., Java Authentication & Authorization Service). According to the information stored in the corresponding organization sub-tree, a credential is generated. We can restrict a user of one tenant from using this credential to access the private domain of other tenants without authentication since the credential contains the tenant data.

3.1.4 Application Layer Conclusion

Multiple instance multi-tenancy and native multi-tenancy are the two main architectural strategies for achieving multi-tenancy. We discussed these two approaches in Section 3.1.1. Multiple instances multi-tenancy supports each tenant with its dedicated application instance over shared hardware, and/or a middleware server in a hosting environment. It can be applied at the infrastructure level or middleware level of the software stack. The infrastructure level multi-tenancy technique is used to maximize scalability and resource usage by sharing infrastructure. An application server, and possibly also the database layers

Comparison of Data Isolation Approaches			
Approach	**Data Isolation**	**Extensibility**	**Cost**
Separate Database	Database Instance Level	Custom Columns	High
Shared Schema	Row Level	Pre-allocated columns	Low
Separate Schema	Schema Level	Custom Columns	Moderate
Hybrid Model	Row Level Schema Level	Custom Columns Pre-allocated columns	Moderate

Table 3.5: Data Isolation Approaches Comparison

of the software stack are shared in the middleware level multi-tenancy strategy. Native multi-tenancy is attaining at the application level that supports all tenants by a single shared application instance over various hosting resources. Data isolation and authentication are two significant tasks needed to be performed for attaining application-level multi-tenancy. A brief comparison of different data isolation approaches is shown in Table 3.5.

In the MBP, native multi-level tenancy (cf. Section 3.1.3) is a suitable approach for enabling multi-tenancy. It is also known as application-level multi-tenancy, in which a single instance of the application serves all of the tenants. It is a shared everything architecture, which means every component of the software stack is shared among tenants. Tenant authentication is done in software code, and data isolation is achieved at the data layer. From the tenants' data isolation, a shared schema approach with a shared database is convenient for the MBP because the data is more or less similar. The shared schema approach is also less costly than the other alternate options. To accommodate a vast number of tenants, we use a scalability mechanism for the middleware layer, which we discuss in detail in Section 3.2.

3.2 Middleware Layer

The middleware lies between the OS and the application in a distributed computing system. It is software that offers services other than those offered by the OS to software applications. Application and database servers and message brokers are the components of the middleware. IoT middleware is a software that acts as an intermediary between IoT elements, allowing interaction among non-capable elements. Middleware often integrates complicated and already existing programs that were not originally designed to be connected. The nature of the IoT makes it possible to connect anything and exchange data over a network. Middleware is part of the architecture that offers a layer of communication.

In this section, we discuss scalability and replication of databases, scalability of message brokers, and different possible ways to scale the middleware as a whole. We also discuss multiple auto-scaling techniques that can be used to decide when to scale-in or scale-out.

3.2.1 Data Replication and Scalability

TSD is data gathered over time consisting of a series of observations (e.g., sensor data, data from smart grids). TSDBs are being used to store and handle TSD. Bader et al. [BKF17] perform a comprehensive research and comparison that identifies 83 TSDBs. They compare 12 popularly chosen open source TSDBs with 27 criteria. Open source TSDBs have been divided into three groups: TSDBs depending on any existing DBMS, TSDBs independent of any DBMS, and RDBMS. They infer that not one TSDB suits every case of use but highlighted InfluxDB [Inf], MonetDB [Mona], and Druid [Dru] as the feature richest. In terms of consistency, accessibility, and immutability, Druid and InfluxDB provide mostly comparable characteristics. InfluxDB, however, supports more programming language interfaces than Druid. InfluxDB also has a Time Structured Merge Tree engine that increases ingestion and compression of data at higher rates.

In the context of IoT, Arnst et al. [AHPW19] conduct a comparative survey on the reading and writing performance of three common DBMS. The authors chose MongoDB [Monb], InfluxDB, and MariaDB [Mar] for their study. MongoDB is a document-oriented database that allows flexible schemas. Data is stored internally in binary JSON documents, which are organized in collections in turn. InfluxDB has a strict schema architecture as a time-series database. Each data series is composed of points. Each point includes a timestamp, the measurement name, an optional tag, and one or more fields of key values. The timestamp has a nanosecond precision and is indexed. The measurement name should be used to describe the stored data. Therefore, the optional tags are indexed and used to group data. InfluxQL, a SQL-like query language, is used to retrieve data. They rate InfluxDB higher than others and endorsed it as a suitable DBMS for handling high recurrent IoT sensors data. Sanaboyina [San16] records OpenTSDB [Ope] and InfluxDB performance assessments based on data center energy usage. The study hypothesizes that the database application energy is directly proportional to its use of resources (RAM, Disk, and CPU). They observe InfluxDB be somewhat more energy effective than OpenTSDB in both single-node and distributed database environments.

The CAP theorem implies that a common data system cannot promise at the same time, all three of the following characteristics:

- Consistency
- Availability
- Partition tolerance

Consistency ensures that everybody can read the latest version of the data from the database whenever an update activity is complete. A system where users are unable to perceive the new data immediately is not firmly consistent and is referred to as eventually consistent. Availability is usually accomplished by running the database as a node cluster, using replication or partitioning data across several nodes, so if one node fails, the other nodes can still operate, which ensures that the system requires continual operations. Partition tolerance means that although part of it is unavailable (e.g., due to maintenance purpose or network connectivity), the system will continue to operate. This can be done by redirecting writings and reading to still available nodes. Nevertheless, for a single node network, this property is useless, it only works for a cluster. For databases used in banking or accounting data, consistency is essential. However, some systems may favor partition tolerance and availability over consistency. For example, the emphasis on availability and partition tolerance lies on social networks, wikis, forums, and other large-scale websites with high traffic and low latency requirements. It is difficult to achieve the ACID properties for these systems and, therefore, the BASE approach is more likely to be implemented, that is, 1. Basic Availability, 2. Soft-state and 3. Eventual consistency. Basic availability means that in terms of the CAP theorem, the system guarantees availability. Soft-state implies that, while there is no input, the state of the system can change over time. Eventual consistency means that the system will be consistent over time during the state where the input is not received. The system using the BASE method does not need to be exclusively available and consistent all the time but is more tolerant of faults.

The distributed database architecture is usually defined as follows [SWED16]:

- *Single Server:* The simplest option without any distribution is represented by a single-server architecture. In this architecture, a single node manages all requests for reading and writing.
- *Master-Slave:* The data is spread across many nodes in a master-slave distribution. One node serves as a master node that executes all write requests and synchronizes the slave nodes that perform read requests only. It is possible to apply the master-slave distribution to scale databases based on reading workloads. This architecture allows the replication of data from the master database server to one or more slave database servers. The master reports changes that spread to the slaves afterward. The slave issues a message stating that it has successfully received the update, allowing future updates to be sent.
- *Multi-Master:* The distribution of multi-master or peer-to-peer is not based on different types of nodes, i.e., all nodes are similar. Replication and sharding of the data are implemented in a multi-master distribution to distribute write and read requests across all cluster nodes. It is a decentralized architecture, unlike Master-Slave.

InfluxDB is an open-source time-series database developed by InfluxData with optional closed-source components. It is written in the GO programming language and is designed for handling data from time-series. The open-source version, namely the TICK Stack, provides a time-series database platform with a variety of services including the InfluxDB core and can run on a single node in the cloud and on-premises. The closed-source versions, such as InfluxEnterprise and InfluxCloud, offer additional features, such as scalability, high availability, backup and restore, and run on-site or in the cloud. Kapacitor is used to stream data in real-time, and Chronograf is used to monitor data. The InfluxData ecosystem

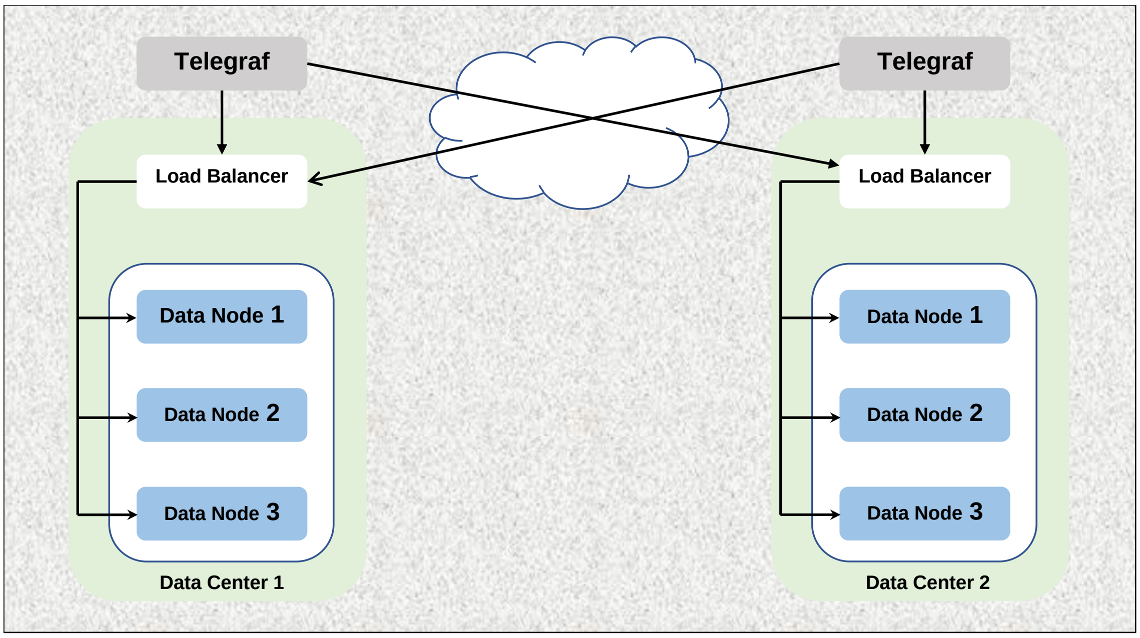

Figure 3.9: Telegraf Replication

includes both Kapacitor and Chronograf. For IoT Monitoring, DevOps Monitoring (Application Monitoring, Infrastructure Monitoring, Cloud Monitoring), and Real-Time Analytics, InfluxDB is typically chosen by clients.

When InfluxDB revealed that the open-source version would not include clustering, it faced a global uproar. Nonetheless, InfluxData offers a high-availability open-source replication solution while many users use sharding in the open-source version as a workaround for the missing clustering feature. This can, for instance, cause InfluxDB to look unattractive for low-budget start-ups. Sharding is the horizontal division of data into a database. Every partition is referred to as a shard. InfluxDB stores data in shard groups ordered by retention policies and stores data inside a specific time period with timestamps. A retention policy specifies how long the data is stored by InfluxDB and how many copies are stored in the cluster. The length of the above interval of time relies on the Retention Policy (RP) duration. The standard shard group intervals are 1 hour for RP less than 2 days, 1 day for RP between 2 days and 6 months and 7 days for RP greater than 6 months. For efficient drop operations, the shard group length is essential, where data is dropped per shard, not per data point. For example, if a RP has a 10 hour length, splitting the data into 5-hour intervals does not make sense. Nevertheless, compression and speed can be affected by short shard group durations for large RP.

There are usually two patterns that could be used to replicate data into the InfluxDB multi-datacenter. The first is to transfer data to the second data center cluster when it is ingested into InfluxDB. The second approach is to replicate data on the backend from one cluster to another.

Listing 3.1 Exemplary Telegraf Config File [Pat]

```
##Cluster1
[[outputs.influxdb]]
urls = ["URL of the cluster load balancer"]
database = "Name of the DB you want to write to"
retention_policy = "RetentionPolicy"
write_consistency = "any"
timeout = "5s"
content_encoding = "gzip"

##Cluster2
[[outputs.influxdb]]
urls = ["URL of the cluster load balancer"]
database = "Name of the DB you want to write to"
retention_policy = "myRetentionPolicy"
write_consistency = "any"
timeout = "5s"
content_encoding = "gzip"
```

Ingest replication is a method used to copy data to InfluxDB when ingested. This is, perhaps, the easiest way to set up for the data replication and the most beneficial for all new data coming into our clusters from external sources. Most of the ingestion patterns are based on Telegraf in some way. Telegraf can be used as a data aggregator, agent, or to assist in setting up pipelines for ingesting data. Replication on ingestion can be done by Telegraf replication or Telegraf replication with Kafka. The first pattern to be discussed is to use Telegraf alone to replicate all data to both clusters when ingested, as shown in Figure 3.9. To set this up, in the Telegraf config file, we define the URL of both clusters.

The configuration, depicted in Listing 3.1, states that we need to write to two different clusters of InfluxDB. One thing to note is that either the URL of a load balancer in front of the cluster or a list of the URLs of each data node in the cluster should be the URL that one specified in the config file. It is the simplest method to set up. Telegraf will try to rewrite the failed writes in the case of a network partition between Telegraf and the clusters. It will also store data to be written in a buffer in the memory. One can customize the size of this buffer in Telegraf.

Telegraf replication with Kafka, shown in Figure 3.10, has Kafka in front of Telegraf instances. Kafka offers a long-lasting write queue for all data as it reaches the cluster. Therefore, this also brings more versatility to what can be achieved with all the input data. For example, for other types of analysis, one may also want to send all the data to long-term storage or send it to other analytical platforms.

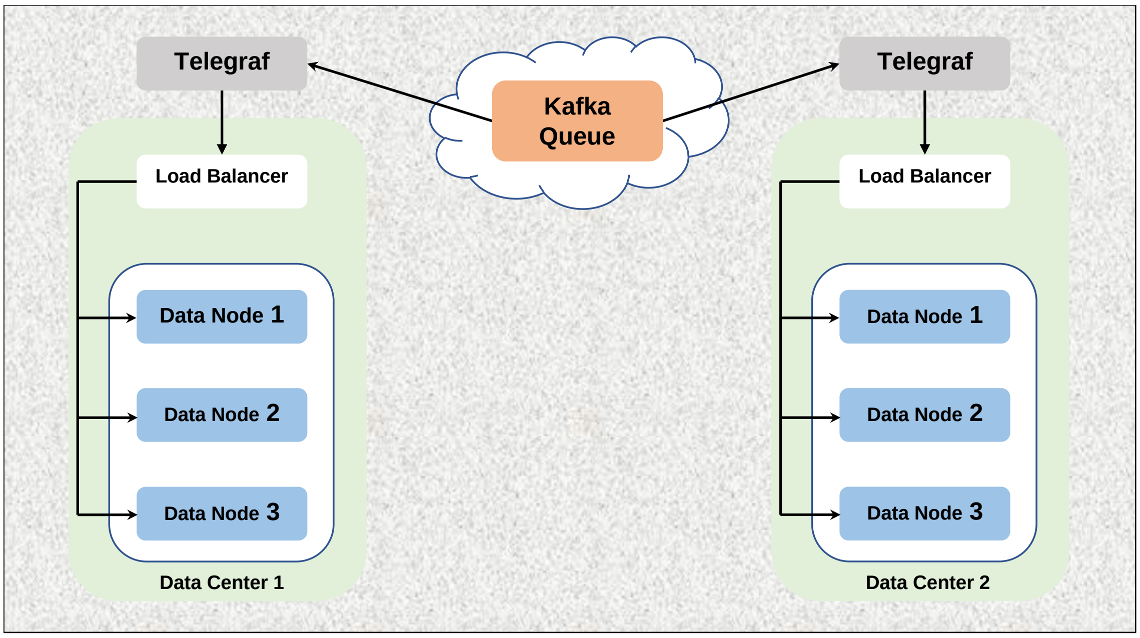

Figure 3.10: Telegraf Replication with Kafka

3.2.2 Scalability of Message Brokers

Messaging is a loosely connected interaction solution that greatly reduces the dependence of producers and consumers. Eliminating these dependencies allows the architecture as a whole to be more versatile and tolerant of change, but it comes with increased complexity. Therefore, over the years, dedicated middleware messaging has been developed to provide communication features without having to deal with the inner complexities. A messaging system, as depicted in Figure 3.11, serves as a layer of indirection between communicating entities. Commonly mentioned as a message broker, it is known for transferring data from one application to the other as messages, so that producers and users can concentrate on what to communicate instead of how to communicate. Message brokers are the messaging system's most popular implementation. A message broker is an intermediate standalone entity offering messaging features through custom or standard protocols. There are many message brokers with various features, protocols, languages for implementation, and support for platforms. Many approaches have been suggested that can usually be split into broker-based solutions (such as Kafka [Kaf], Active MQ [Apa], and RabbitMQ [Rab]) and broker-less solutions, like Java Messaging Service or ZeroMQ [Zer]). Broker-less systems primarily provide an API that abstracts network communication information while broker-based systems use a central system as a communication channel between peers. We analyze and discuss only broker-based solutions.

A flexible and minimal-cost message broker is needed to manage a huge number of messages that communicate between IoT devices in real-time, supporting a massive amount of IoT devices. MQTT is a lightweight topic-based publish/subscribe messaging protocol that is designed for machine-to-machine communications. It is standardized by the Organization

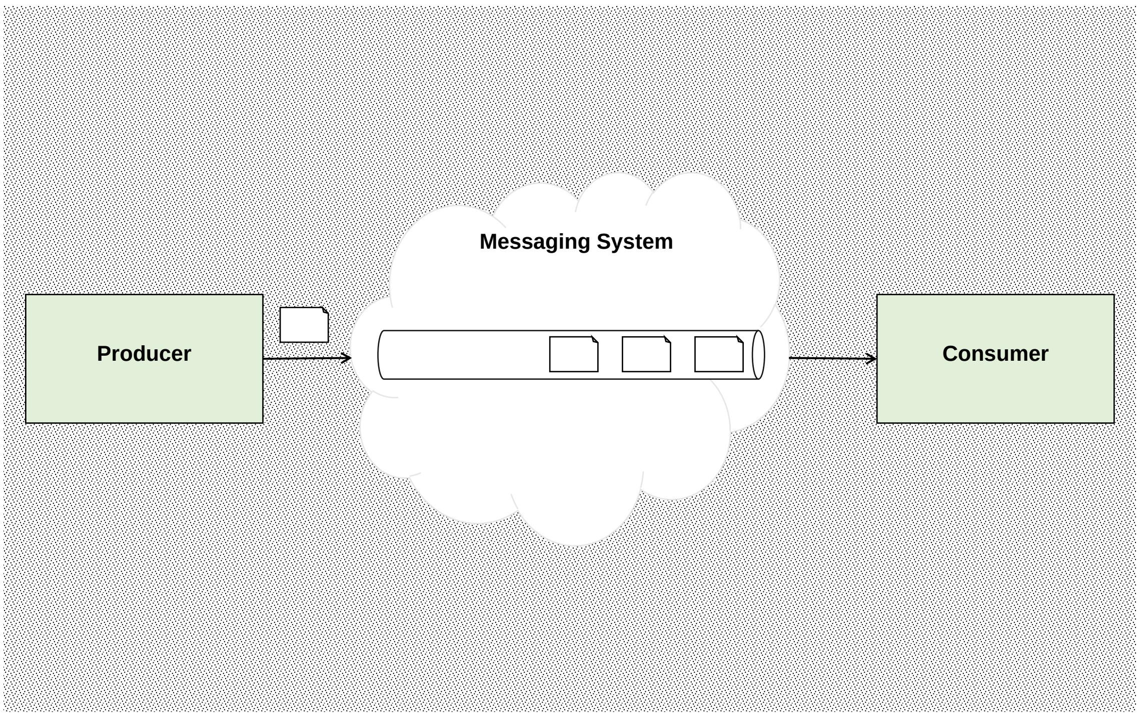

Figure 3.11: Messaging Communication [Mag15]

for the Advancement of Structured Information Standards [MQT]. The benefit of the publish/subscribe design is that it is not important for the author and the subscriber to know each other. The broker acts as a facilitator between publishers and subscribers for the exchanges of messages. The connection is formed by the client forwarding the broker a connect message request. The broker acknowledges the client's request by sending a response with a connection acknowledged message. Once the connection is established to the broker, the user can produce or consume messages. A client's published message must contain a topic name that is a string that the broker uses to send the message to the topic subscribers. The subscriber may subscribe to the MQTT client by sending a subscribe packet containing the specific topic to a broker. There are three general strategies to enhance the scalability of the message broker: 1. Improving a single machine's performance, 2. clustering of messages brokers or 3. Apache Kafka. Upgrading resources of a single machine is not the right way to achieve scalability. We discuss the clustering of brokers and Apache Kafka in detail below.

Clustering of Message Brokers: Figure 2.3 illustrates the system overview consisting of the message broker cluster and the MQTT load balancer. This includes the load balancer at the frontend and the MQTT broker cluster at the backend. To perform the MQTT load balancing, NGINX [Ngi] is running as a frontend server. This is the core part of the client-broker architecture. It is the interface interacting with the clients of MQTT. The messages will then be sent to a broker node in the MQTT cluster backend. For the MQTT

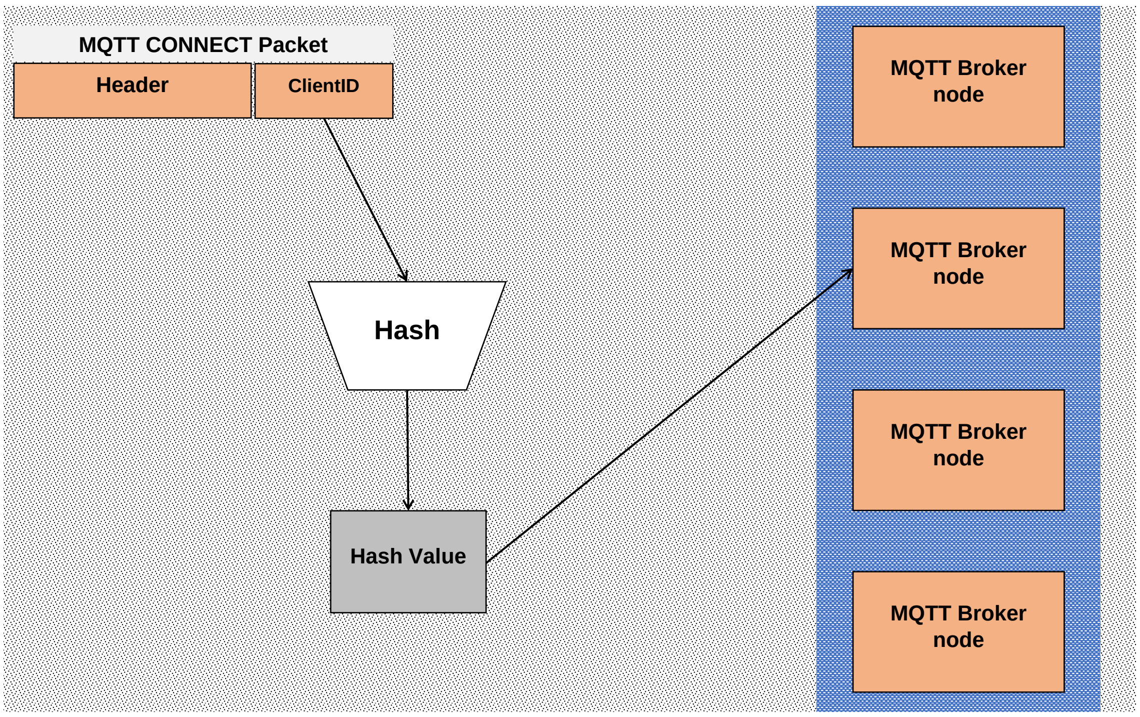

Figure 3.12: Load Balancing MQTT messages based on ClientID [JPV+17]

cluster at the backend, Docker Swarm [Doc] can be used to form a cluster of Raspberry Pi nodes and running Mosquitto on each node in container form. Figure 3.13 illustrates the clustering of message brokers.

The load balancer is used in combination with the MQTT broker cluster to have a single entry point for all MQTT clients. The message producer sends messages to a broker, and by subscribing to the broker, the subscriber will receive the messages. In the case of a lot of devices and messages, more brokers are required to manage the broker's message congestion and network traffic. Therefore, to spread incoming network traffic through several brokers, it requires a load balancer. It also offers high availability and accuracy by sending messages only to online brokers. When communicating with such a system, a MQTT client sends a MQTT connect packet to establish a connection with the broker. This packet contains that MQTT client's header and ClientID as identifier. Thus, we can use the ClientID for hashing to perform the load balancing, as shown in Figure 3.12. By using this process, various ClientIDs will be hashed and linked thoroughly to a different broker for messaging.

If one of the brokers is shut-off, the load balancer will automatically connect to the new broker, which will keep the broker cluster running and available. Also, the load balancer offers the consistency of the connection as it must first send the connect packet before forwarding the request to a message broker from a client. If the acknowledgment message is not received after sending the connect packet, the load balancer will know that the broker is not available so that it will forward the MQTT request to another available broker.

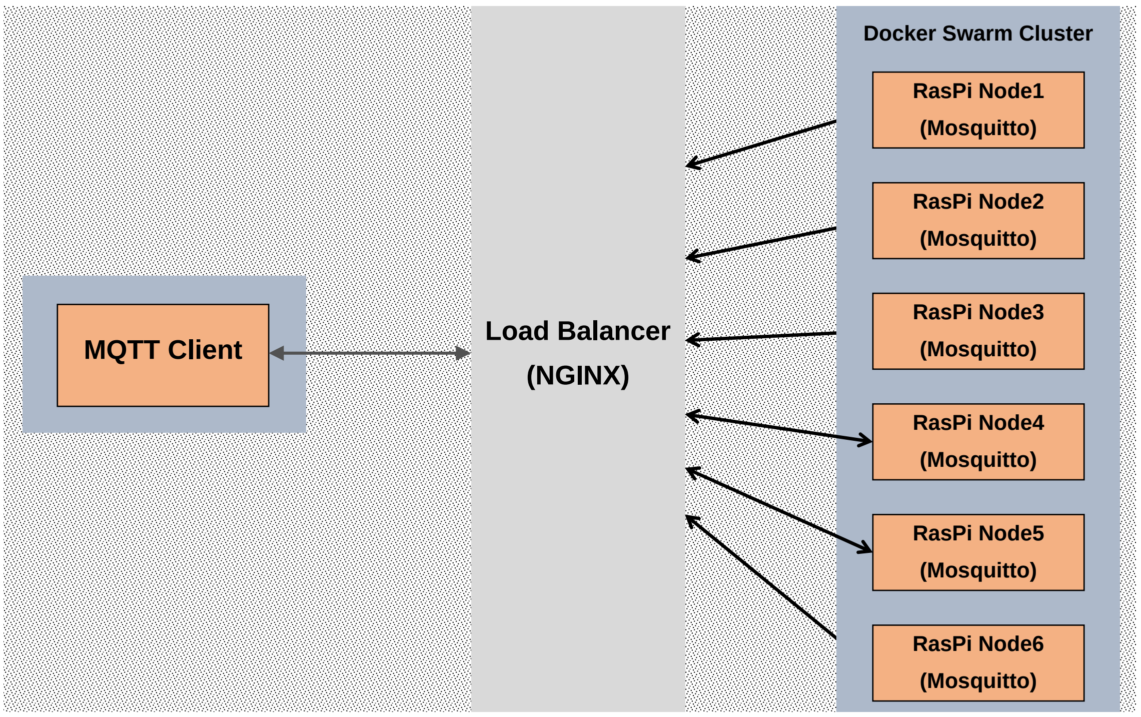

Figure 3.13: MQTT Cluster [JPV+17]

Docker Swarm, which offers a framework for a cluster orchestration management, can be used for the backend message broker clustering. Docker Swarm can be easily installed and configured by simply receiving a token from the manager node to add a new node to the cluster, which makes the cluster scalable.

Jutadhamakorn et al.- [JPV+17] first install Docker on each node for the implementation of the MQTT cluster, and then the nodes join the cluster one by one by running the Docker swarm init command. Next, they launch a MQTT broker's service, such as Mosquitto, an open-source message broker. Under Docker, a MQTT broker runs and shares the cluster's internal storage. A broker will be able to join and leave the cluster easily. By running the token from the manager node, a new node is connected to the Docker Swarm cluster. The swarm join-token manager command needs to be run to get the token and copy the token to the desired node. The new node then joins the cluster.

Apache Kafka: Apache Kafka is built based on a broker-based message-oriented middleware as a distributed streaming platform. The goal of Kafka is to provide a medium for high-performance input data sequence processing, and its major design objectives are scalability, reliable data transfer, fault tolerance, and fast processing. From a general point of view, a distributed system based on Kafka consists of three main components: 1. message producer, 2. message consumer, and 3. Kafka messaging broker. The Message producer generates data streams that one or more consumers read and/or process. The Kafka messaging broker, which is primarily responsible for the storage of all messages until the user receives them. A Kafka message consists of a timestamp, a key/value pair

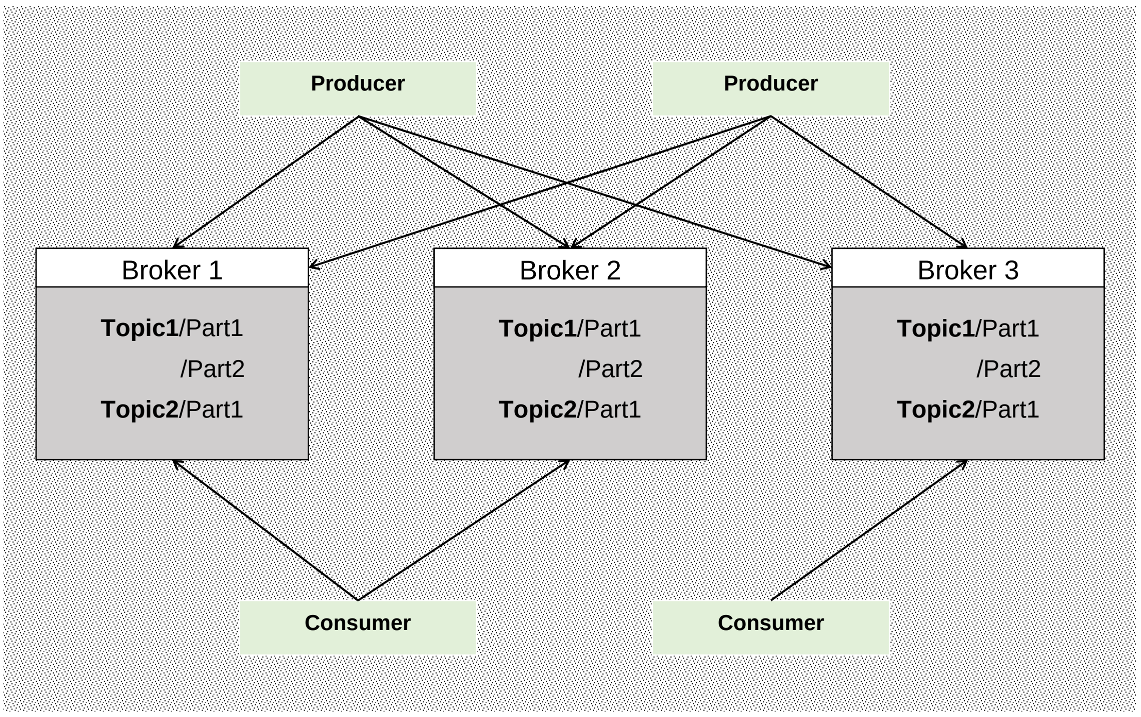

Figure 3.14: Kafka Architecture [KNR+11]

for communication encoding, and an information address provided by the core message topic definition. A topic essentially gives a name for a message group with a certain kind. The producer can publish a message of that kind by using that topic name. On the other hand, customers may subscribe to such messages using the topic name as well. There may be multiple consumers on each topic. The messaging protocol used among Kafka clients and brokers is considerably more powerful than the RabbitMQ and ActiveMQ, which use AMQ. This becomes particularly apparent when looking at a performance analysis that measures latency and CPU workload when processing bundles of messages. Figure 3.14 demonstrates Kafka's overall structure. Because Kafka is distributed in nature, there are usually multiple brokers in a Kafka cluster. To manage load, a topic is split into several partitions, and one or more of those partitions is stored by each broker.

Kafka uses so-called partitions to provide high performance while messages are published or consumed. Partitions can be interpreted within a single topic as message sub-queues where the published messages are processed in a fixed first-in-first-out order. Usually, each partition is managed by a Kafka cluster instance running on a dedicated host. Writing and reading performance of a topic is, thus, substantially increased by allowing multiple consumers and/or producers to write and read in parallel. Published messages are organized into exactly one particular partition, either explicitly specified by the message producer or in a Kafka-controlled round-robin manner. Essentially, a consumer polls a topic for message consumption and retrieves messages from any of the partitions. Also, consumers can be grouped into consumer groups to provide a means of load balancing and parallel storage on the recipient side, leading to direct mapping among consumers in that group

Listing 3.2 Sample producer code [KNR+11]

```
producer = new Producer();
message = new Message(msg.getBytes());
set = new MessageSet(message);
producer.send(topic1, set);
```

Listing 3.3 Sample consumer code [KNR+11]

```
streams[] = Consumer.createMessageStreams(topic1, 1)
for (message : streams[0])
{
bytes = message.payload();
// do something with the bytes
}
```

and certain partitions. Every group consumer gets messages from a specific partition. Parallel reads and writes reduce the increased latency created by general message brokering. Concerning the Kafka communication system's fault tolerance, each partition may have replicas on other Kafka instances that provide a means of redundancy. One of the replicas is called a master in which messages are first stored and then distributed to other instances. The message can only be pulled by subscribed users after the message has been hard-drive-persisted at each replication. This persistence message also provides the opportunity of fault tolerance in the event of a Kafka instance system failure. Kafka offers secure transport layer connections between broker systems and producers/consumers. It also provides access control lists defining read/write access rights for each topic that a consumer or producer has. Listing 3.2 and Listing 3.3 illustrate the message producer and message consumer code simultaneously.

The information on how much each consumer has consumed in Kafka is not ensured by the broker, but rather by the user. Such a model reduces the broker's complexity and overhead. However, it makes it difficult to remove a message because a broker does not know if all subscribers have processed the message. By using a straightforward time-based service level agreement for the retention period, Kafka solves this problem. If a message has been preserved in the broker for longer than a certain timespan, commonly 7 days, a message is deleted immediately. In practice, this solution works well. Most of the consumers end up consuming messages either hourly, weekly, or in real-time, including offline ones. The fact that Kafka's performance with a larger data size does not degrade makes this long persistence possible. A user may purposely return to an older offset and re-use data. It breaches a queue's standard contract, but for many customers, it appears to be an important feature. Kafka ensures that messages are distributed to a customer in order from a single partition. However, there is no guarantee that messages from separate partitions will be ordered. Kafka is an optimized data flow solution, which is often used as a pipe for various processing systems, such as Hadoop and Storm.

3.2.3 Scalability of the Middleware as a whole

The scalability of the middleware as a whole involves mapping performance demands to the offered underlying resources. It can be pretty challenging to adjust resources to an application's on-demand requirements, which is called scaling. Under-provisioning resources will eventually hurt performance, while over-provisioning resources can lead to idle instances, leading to needless costs. The first consideration is to plan the average load capacity or peak load capacity and set up the fixed infrastructure according to the planned capacity. There is less cost incurred when planning for the average load, but performance will be a concern if load spikes occur. Poor performance will discourage customers and affect revenue. On the other hand, most resources will stay idle almost all the time if capacity is anticipated for the maximum possible workload. Thus, a more comprehensive resource allocation strategy seems to be appropriate, which will dynamically scale resources to demand. These are called auto-scaling strategies.

Well-known cloud service providers, such as Amazon EC2, generally offer auto-scaling based on rules. Typically, this straight-forward approach includes two rules to assess when the system should be scaled-up or scaled-down. The user must classify a condition for each rule based on a target variable, e.g., HTTP requests > 80. When the criterion is satisfied, a predetermined scaling-down or scaling-up action is activated; e.g., instantiating a new VM. The rule-based approach could be defined as a reactive algorithm, meaning that it responds to the load of the system but does not predict system load. Predictive or proactive auto-scaling strategies, on the other hand, attempt to predict the system needs in the future and, thus, obtain or release resources ahead of time so that the system is ready in advance to tackle the load when required.

Auto-scaling techniques are grouped into the following categories [LML12]:

- Static, threshold-based policies
- Reinforcement learning
- Queuing theory
- Control theory
- Time-series analysis

It is complicated to work out a proper categorization of auto-scaling strategies. It is possible to split auto-scaling techniques into two main classes: reactive and proactive [LML12]. Threshold-based strategies are part of the reactive classification while analyzing time-series is a solely proactive approach. In comparison, queuing theory, reinforcement learning, and control theory can be used with both reactive and proactive approaches.

Reactive strategies, as the name suggests, refer to the number of methods that respond to the state of the existing system and/or application. Decisions are made based on the previous values acquired from a set of variables that have been monitored. Threshold-based norms or strategies follow a purely reactive method among several classifications listed above.

The lack of awareness of a reactive approach has a strong impact on the performance of auto-scaling. With the massive traffic bursts likely to result from special deals or business campaigns, the system might not be able to scale proportionally. In addition, it may end up taking up to 15 minutes to instantiate a new VM, and the impact of a scaling-up operation may come too late. Hence, proactive or predictive resource-based scaling is needed to cope with the variant pattern of resource usage and to be able to scale ahead of time continuously. Time series research covers a wide range of approaches, which adopt a proactive approach among the auto-scaling categories mentioned above.

Now, we discuss each of the auto-scaling technique and their limitations when applied to the scaling task.

Static, Threshold-Based Policies: Application specifications can change over time, and users may also host various applications (which require different resources) on the VM. Fixed VM capacity in these cases can result in resource waste or degradation of application performance. This issue can be resolved by interactively scaling the VM as per the requirement of the host application. The resource usage of the VM is controlled in threshold-based auto-scaling. In the case of vertical scaling, if they surpass the clearly defined threshold values, the computing resources of the VM will be increased or dynamically decreased as needed without shutting down the VMs, thereby reducing the waste of resources. A new VM is added instead of upgrading resources if horizontal scaling is being followed. Threshold-based policies or rules are very common and popular among cloud service providers like Amazon EC2. These policies' simplicity and straightforward nature make them quite desirable to users. Though setting thresholds is a task per application and requires a thorough understanding of patterns in the workload. In the target application, the number of VMs will depend on a set of rules. Usually, there are two sets of rules: one to scale-up and one to scale-down. These rules may use one or even more performance indicators, such as CPU load, request rate, or average response time. Each rule has several user-defined variables: a lower threshold; an upper threshold; two-time values that specify how long the condition must always be fulfilled to cause a scaling action. The upper and lower thresholds variable for each performance metric must be defined. The scaling behavior differs depending on the kind of scaling. For horizontal scaling, the user should specify a fixed amount of VMs to be allocated or de-allocated, but for vertical scaling, the number of resources that will be added (RAM, CPU, etc.) need to be defined by the user. The resulting rules will have a structure similar to these [LML12]:

$$\begin{aligned} &\text{if } x > upThr \text{ for vUp seconds then} \\ &\qquad n = n + s \text{ and} \\ &\qquad \text{do nothing for inUp seconds} \end{aligned} \tag{3.1}$$

$$\begin{aligned} &\text{if } x < downThr \text{ for vDown seconds then} \\ &\qquad n = n - s \text{ and} \\ &\qquad \text{do nothing for inDown seconds} \end{aligned} \tag{3.2}$$

Threshold-based rules are shown in the following example: “introduce 2 more instances of virtual machines if for more than 5 minutes the average CPU load is above 70%”. The user can identify the highest possible $nMax$ and the lowest possible $nMin$ amount of VMs based on horizontal scaling, both to control the total cost and to ensure a minimum threshold of availability. If the performance measurement x breaches the parameter $upThr$ for vUp seconds at each iteration, s number of VMs are proposed and introduced to the application server pool. Then, for $inUp$ seconds, the auto-scaling system restricts itself from performing any other scaling task. If the performance measurement x becomes less than $downThr$ for a user-defined time of $vDown$ seconds, it will remove s number of VMs and release their resources. And again, the system prohibits itself for $inDown$ seconds.

Threshold-based rules could easily handle the number of resources allocated to an application deployed on a cloud platform and carry out self-scaling actions to evolve the resources required for the input demand. Setting the rules, however, requires additional effort from its users, who are required to choose the appropriate performance parameter or logical possible factors combination, as well as setting multiple parameters. Among these variables, the upper and lower output vector thresholds (e.g., 30% and 70% percent of the CPU load) are the key to the proper functioning of the rules. Rules conditions are typically based on one or at the very most two performance indicators, and the most common has always been the VM’s average CPU workload, input request frequency, or response time. In particular, Dutreilh et al. [BGS+09] point out that thresholds ought to be carefully adjusted to avoid system fluctuations in terms of several VMs or the total number of CPUs allocated. It is, therefore, appropriate to define a cool down or quiet period to avoid this problem, a time in which no scaling decisions need to be made. Many of the researchers and cloud service providers use only two threshold variables per performance metric, but Hasan et al. [HMC+12] take into account the use of four threshold values for each performance metric and two durations. $ThrBU$, slightly below the upper threshold; ThrUP, the upper threshold; ThrOL, slightly above the lower threshold; ThrL, the lower threshold; and a couple of durations applied to control the persistence of the performance metric.

Algorithm 3.1 depicts the web service application auto-scaling algorithm. First, the algorithm calculates the current VMs with bandwidth utilization and currently active sessions that are lower or higher than the defined threshold numbers. If all VMs have bandwidth utilization and effective sessions above the defined upper limit, a new VM will be installed, started, and connected to the front-end load-balancer, accordingly. If there are VMs that have active sessions and network bandwidth below a specified lower threshold and at least one VM has no web traffic or effective sessions, the idle VM will be removed from the front-end load-balancer and is immediately terminated from the system. Physical resources of computing, such as CPU and memory, are extremely important metrics for the evolution of a virtual cluster’s workload for distributed computing tasks.

Algorithm 3.2 shows the auto-scaling algorithm for distributed computing tasks. First, the algorithm resolves the current VMs that are using physical resources (CPU and memory) above or below the given threshold. If all VMs are using resources above the specified

Algorithm 3.1 Threshold Based Auto-Scaling Algorithm For Web Applications

Input:

n: Number of Clusters
VC: A Virtual Cluster consists of VMs that run the same computational system
VM_{ns}: Number of the active sessions in a virtual machine
S_{iMax}: Maximum sessions for a virtual machine of i-th Cluster
S_{upper_bound}: Session upper-threshold
S_{low_bound}: Session low-threshold
E_{below}: A virtual machine set records virtual machines that have active sessions below the session low-threshold

Output:

Front Load-Balancer Set FLB

```
for $i = 1$ to $n$ do
    for all VM $\in VC_i$ do
        if VM$_{ns}/S_{iMax} \geq S_{upper\_bound}$ then
            $e = e + 1$
        end if
        if VM$_{ns}/S_{iMax} \leq S_{low\_bound}$ then
            $b = b + 1$
            Record VM to $E_{below}$
        end if
    end for
    if ($e == |VC_i|$) then
        Provision and start a new VM that runs the same system as $VC_i$
        Add new VM to FLB                          // Front Load-Balancer Set
    end if
    if $b \geq 2$ then
        for all VM $\in E_{below}$ do
            if VM$_{ns} == 0$ then
                Remove VM from FLB                 // Front Load-Balancer Set
                Destroy VM
            end if
        end for
    end if
    Empty $E_{below}$
end for
```

Algorithm 3.2 Threshold Based Auto-Scaling Algorithm For Distributed Computing Tasks

```
Input:
  $n$: Number of Clusters
  $VC$: a Virtual Cluster consists of VMs that run the same computational system
  $VM_R$: The use of resources in a virtual machine
  $R_{upper\_bound}$: The upper-threshold of use of physical resources
  $R_{low\_bound}$: The low-threshold of use of physical resources
  $V_{below}$: A virtual machine set records virtual machines that have the usage of resources
  lower than low-threshold

  for $i = 1$ to $n$ do
      for all VM $\in VC_i$ do
          if $VM_R \geq R_{upper\_bound}$ then
              $e = e + 1$
          end if
          if $VM_R \leq R_{low\_bound}$ then
              $b = b + 1$
              Record VM to $V_{below}$
          end if
      end for
      if $(e == |VC_i|)$ then
          Provision and start a new VM that runs the same computing tasks as $VC_i$
      end if
      if $b \geq 2$ then
          for all VM $\in V_{below}$ do
              if VM is idle then
                  Destroy VM
              end if
          end for
      end if
      Empty $V_{below}$
  end for
```

upper threshold, a new VM will be created, provisioned, started, and then conduct the very same computing tasks in the virtual cluster. If some VM's resource usage is below a specified lower threshold and at least one VM has no computing task, the inactive VM from the virtual cluster will be removed.

Many researchers introduce and implement the auto-scaling strategy, which integrates rules for every performance metric. Initially, Chieu et al. [CMKS09] introduced a set of adaptive rules depending on the number of currently active sessions. If there are active sessions beyond the specified upper threshold in all instances of the application, a new instance is created. If there are running instances with presently active sessions under a given lower

Static Threshold-based Rules		
Reference	**Horizontal/Vertical Scaling**	**Metric**
[DMM+10]	Horizontal Scaling	Response Time
[HMC+12]	Both	CPU load, response time
[HGGG12]	Both	Response Time
[MBS11]	Vertical Scaling	CPU, Memory, Storage

Table 3.6: Summary of Static Threshold-Based Rules [LML12]

threshold and at least one instance without an active session, it will shut down the idle instance. Table 3.6 shows a variety of performance metrics and scaling decisions adopted by few authors using static threshold-based rules as auto-scaling techniques.

In conclusion, rules could be used to efficiently automate a specific application's auto-scaling without that much effort, especially in systems with predictable patterns that are quite frequent. In the case of bursty workloads, however, the user should acquire a more technically advanced auto-scaling method from the remainder of the categories.

Reinforcement Learning: Reinforcement Learning (RL) refers to a collection of general trial-and-error strategies by which an agent can learn how to make sound decisions via a series of interactions in an environment. The basic behavior is to analyze the current state of the environment, choose an appropriate action, and then receive an instant reward. [TJDB06]. It is a form of an automated approach to decision-making that can also be used for auto-scaling. It collects a target application's performance pattern and its strategy without any prior information. It is a mathematical approach for understanding and streamlining purpose-driven learning and decision-making. It relies on learning via an agent's direct interaction with its environment. The decision-making component is known as the agent, which will benefit from the trial-and-error process. Agents always improve their judgment capability from the previous actions which were conducted for each state of the environment. They always attempt to optimize the returned reward. In our problem, the auto-scalar is the agent that communicates with the scalable application environment all the time. Depending on different performance indicators that include current input workload, output, or other collection of variables, auto-scalar will decide whether to add or remove resources. It always attempts to minimize the response time as a reward. Figure 3.15 visualizes the interaction between environment and its agent.

Formally, at every time step t, where t is a series of discrete-time steps, the agent receives a certain representation of the state of the environment and chooses an action on that basis. One time step later, the agent receives a response or reward (e.g., performance improvement) as an outcome of its action and puts itself in a new state. The agent enforces a mapping of states to select the probabilities of each possible action at each step of the time. RL systems can also be termed memory-less, in the context that potential forthcoming system states can only be decided with the existing state, irrespective of the previous record. This is known as the Markov principle, which claims that the likelihood of a transition

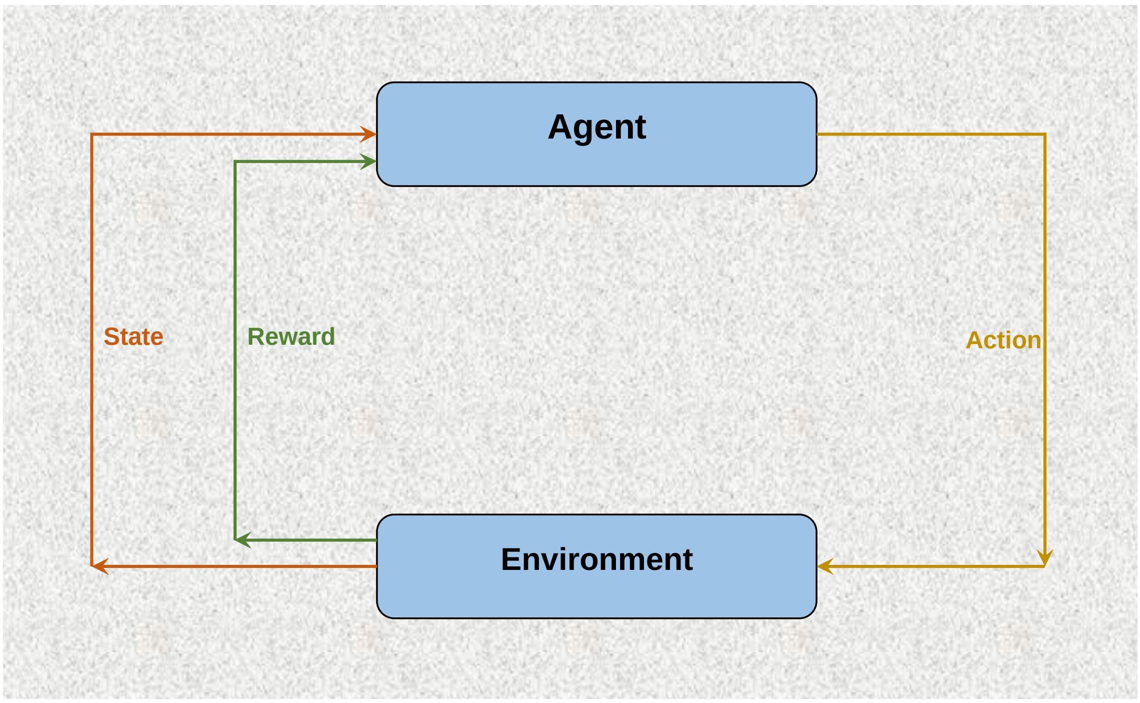

Figure 3.15: The agent-environment interaction in reinforcement learning [LML12]

to a new system state relies only on the present state and the action of the agent. It is provisionally free from all preceding actions and states. The Markov Decision Process offers a computational framework for modeling decision-making problems and is usually used to formulate scenarios for reinforcement learning. RL's key problem is finding an agent or decision-maker policy that maps each state with the best action to maximize future cumulative rewards. In the case of horizontal scaling, we might consider a state specified as (r, v, p), where r is the number of client requests experienced per time frame, v is the amount of VMs assigned to the application, and p is the average response time of requests, bounded by the $pMax$ value chosen from experimental observations.

Q-learning is one of the model-free RL algorithms, which can be used to produce optimal policies. The update rule for Q-learning is specified in Algorithm 3.3, and it is estimated each time a non-terminal state is attained. Approximations of $Q(s, a)$, which are suggestive as to the benefit of taking action a while in state s, are determined after each time interval. Actions are taken based on the policy being pursued. Q-learning can usually require significant experience within a given environment to learn a good policy. To find the optimal policy, the RL algorithm should estimate a utility value for each state s. The utility is the amount of the anticipated discounted rewards. Given the current state, once the agent has learned all the estimated values, it will choose the behavior that leads to the next state with the highest utility value. The policy is based on a value function $Q(s, a)$, usually called the Q-value function. Every $Q(s, a)$ value estimates the future cumulative rewards by executing an action a while in state s.

Algorithm 3.3 Reinforcement Learning Algorithm (Q-Learning)

```
Initialize $Q(s, a)$ arbitrarily.
Observe the current state, $s$.
loop {infinitely}
    Choose an action, $a$, for state $s$ based on one of the action selection policies, such as $\epsilon - greedy$.
    Execute the action, $a$, and observe the reward, $r$, as well as the new state, $s'$.
    Update the Q-value for the state using the observed reward and the maximum reward possible for the next state. The resulting update formula is: $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \cdot \max_{a'} Q(s', a') - Q(s, a)]$
    Set the state $s$ to the new state $s'$. $s \leftarrow s'$
end loop
```

Reinforcement Learning		
Reference	**Horizontal/Vertical Scaling**	**Metric**
[BHD13]	Horizontal Scaling	Number of user requests, and number of VMs
[RBX+09]	Vertical Scaling	CPU and memory usage
[DKM+11]	Horizontal Scaling	Number of user requests, VMs, and response time
[TJDB06]	Horizontal Scaling	Arrival rate, Response time

Table 3.7: Summary of Reinforcement Learning [LML12]

Nearly all RL algorithms are dependent on estimating the value function of states (or state-action pairs) from experience. There are three primary methods for RL: dynamic programming, temporal difference learning, and the methods of Monte Carlo [LML12]. Dynamic programming needs to know both probabilities of transition and rewards, whereas the other two are model-free. Temporal-difference approaches are ideal for step-by-step learning and are best suited for the scenario of auto-scaling. Q-learning is a one-step TD algorithm based on learning the Q action-value function, which approximates the optimal Q directly, regardless of the policy being pursued. Therefore, it is called an off-policy method: the optimum policy can be obtained at any time, taking into account the behavior with the highest Q-function value for each state. Table 3.7 shows the scaling type and performance metrics used by different authors using the RL auto-scaling technique.

It is important to note the capability of the RL algorithms to identify, without a priori information, the best management strategy for a target scenario. The user does not need to specify a specific application model. Instead, if the task workload or system conditions change, it is learned remotely and adapted. RL techniques might be a promising way of solving the specific application auto-scaling problem, but the field is not advanced

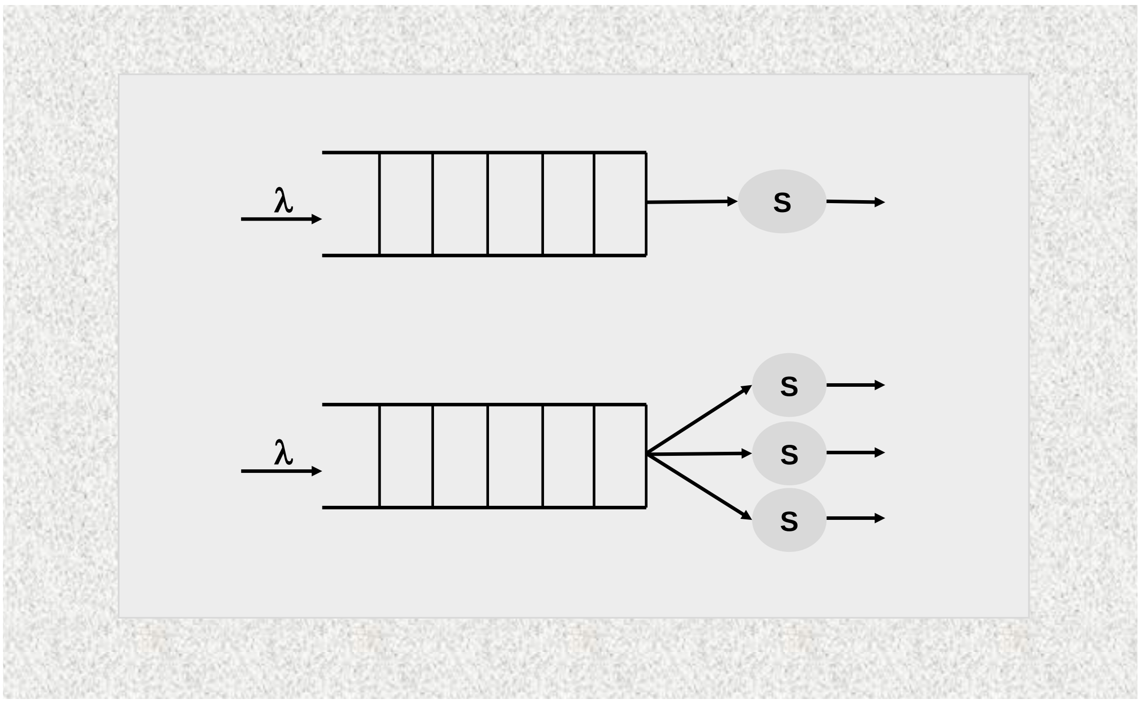

Figure 3.16: A simple queuing model with one server and multiple servers [LML14]

enough to fulfill the requirements of an actual production scenario. In this dynamic field of research, steps should be taken to ensure a proper ability to adapt to occasional bursts in the input workload and to tackle current state spaces and actions [LML14].

Queuing Theory: Queuing theory has been widely used to model Internet applications and typical servers to measure performance metrics, such as the queue length or the average waiting time for requests. The queuing theory concept refers to the mathematical analysis of queues. A model's basic structure is shown in Figure 3.16. Consumer requests arrive at an average arrival frequency of λ to the network and are en-queued until processed. As the figure shows, there may be one or even more servers in the system that will serve requests at an average service rate of μ.

Kendall's notation [Ken] is the generic method for the description and identification of queuing models. A queue is represented by $A/B/C/K/N/D$. The meaning of each element in the notation is as follows:

A: Inter-arrival time distribution.
B: Service time distribution.
C: Number of servers.
K: System capacity or queue length. It applies to the total number of clients in the network, even those in service. More arrivals are discarded when the system is filled.
N: Calling population. The size of the community that the clients come from. If the requests come from an unlimited customer population, the queuing model will be open while a closed model will be centered on a finite customer population.

Queuing Theory		
Reference	**Horizontal/Vertical Scaling**	**Metric**
[USC+08]	Horizontal Scaling	PEAK workload
[VPR07]	Horizontal Scaling	Arrival rate
[ZCS07]	-	Number and type of requests
[TJDB06]	Horizontal Scaling	Arrival rate, Response time, Number of servers

Table 3.8: Summary of Queuing Theory [LML12]

D: Service discipline or priority order. The discipline of operation or priority order in which jobs are done in the queue. The most common is First-In-First-Out / First-Come-First-Served, serving the requests in the order they arrived. Alternatives are available, including Last-in-First-Out / Last-Come-First-Served and Processor Sharing.

Elements K, N and D are optional. The most common values are M, D, and G for both inter-arrival times A and service time B. M is Markovian and corresponds to a Poisson process defined by a λ parameter indicating the number of requests per unit of time. Therefore, an exponential distribution may follow the inter-arrival or operation time. D means times that are deterministic or constant. Another widely used value is G, which corresponds to a known parameter general distribution. A basic queuing model can be used to formulate the elastic application scenario, considering a single queue comprising the load balancer that delivers the requests among n VMs, as shown in Figure 3.16. A queuing network can be used to describe more complicated systems, such as multi-tier applications. For example, with one or n servers, each level can be modeled as a queue.

The theory of queuing is used to evaluate stationary systems characterized by constant rates of arrival and operation. The goal is to extract some performance metrics dependent on the queuing model and some established parameters (e.g., arrival rate λ). The average waiting time in the queue and the mean response time are examples of performance metrics. For situations with changing conditions (i.e., non-constant arrival and service rates), such as our target, scalable applications, the queuing model parameters have to be recalculated regularly, and the parameters have to be readjusted. To obtain these metrics, there are two primary approaches: analytical methods and simulation. Analytical methods are available only for comparatively simple queuing models, with a well-defined arrival and service distributions. A typical formula in this context is the Little's Law. It states that the average number of users or requests $E[N]$ in the system is equal to the average customer arrival rate λ multiplied by the average duration of each customer $E[T]$ [LML14]. The formula is as follows: $E[N] = \lambda \times E[T]$. A similar description for the Little's Law states that it is possible to determine the average queue length as the product of the mean arrival rate and the average queue waiting time.

Information required for a queuing model, such as input workload (number of queries, transactions) or service time, can be collected through online monitoring or calculated using different methods. Approximation based on regression can be used to estimate the demand for CPU on client transactions. Table 3.8 visualizes the scaling type and performance metrics used by different researchers using the queuing theory auto-scaling technique.

Control Theory: Control theory is applied to simplify the management of various information processing systems, such as storage systems, web server systems, clusters of data centers/servers, and other systems. It also demonstrates interesting results on platforms for cloud computing. Control systems are predominantly reactive, and there are some proactive approximations, like Model Predictive Control, or even a hybrid model by combining a control system with a predictive model. A control system could combine the evaluation and scheduling phases of the auto-scaling process for cloud-hosted, elastic applications. A controller's main goal is to automate management, such as a cloud application's scaling tasks. The controller must maintain a controlled variable value of CPU load, y, close to the desired level or set point of $yref$, by modifying the manipulated variable u, such as the number of VMs. The manipulated variable is the input to the target system, whereas a sensor is used to measure the controlled variable and consider the system output.

Three types of control systems are available: open-loop, feedback, and feed-forward. Open-loop controllers, often known as non-feedback, use only the current state and its system model to determine the input to the target system. They do not use feedback to determine if the output of the system has reached the desired objective. Feedback controllers, on the other hand, observe the system output and can correct any deviations from the desired value. Feed-forward controllers attempt to anticipate output errors. They forecast the system's actions based on a model and respond before the error occurs. The prediction may fail, and feed-forward controllers are usually combined for this reason. In the scope of scalable applications, the most effective solutions tend to be both feedback and feed-forward types. Researchers have also used feedback controllers to tackle the problem of auto-scaling. The controller's main objective is to keep the output, such as target system performance, to the desired level (e.g., response time must be less than 2 seconds). This can be done by adjusting the control input, such as resource allocation and the number of VMs. Feedback controllers can be divided into four categories which are described below:

- *Fixed gain controllers:* Because of their simplicity, they are very popular. However, they stay fixed during the time the controller is in operation after selecting the tuning parameters. An example is the Proportional Integral Derivative Controller. It is possible to represent an integral controller as:

$$u_{k+1} = u_k + K_i(y_{ref} - y_k) \tag{3.3}$$

where u_{k+1} is the latest actuator value, such as the new number of VMs. The current actuator value is u_k which is the current number of VMs, K_i is the integral gain parameter, y_k is the current sensor calculation (e.g., CPU load), and y_{ref} is the target value for the variable. The formula for a Proportional-Integral (PI) controller is:

$$u_{k+1} = u_k + (K_p + K_I)E_k - K_p E_{k-1} \tag{3.4}$$

It includes two parameter settings: E_k, which refers to an error (reference minus measured actuator value), and K_p, which is a proportional term that determines the previous cycle's output response ratio to the error. K_I is an integral term for accumulated error throttling in previous cycles.

- *Adaptive control:* Adaptive control by changing the controller tuning parameters online solves some of the drawbacks of fixed gain controllers. A dynamic model of the system is built on each sampling phase. Examples of adaptive controllers are self-tuning proportional integral derivative controllers, gain-scheduling, and self-tuning regulators. They are appropriate for slowly varying conditions of workload, but not for sudden bursts. In this case, the system dynamics may not be captured by the online model estimation process.
- *Re-configuring control:* The parameters are extracted online in adaptive control, but the controller stays the same. The re-configuring controllers eliminate this restriction by making it possible to change the controller at run time depending on the conditions. Some examples are Model-Switching and tuning adaptive control. They can be ideal for handling workload bursts that are both predictable and unpredictable.
- *Model predictive control:* Model predictive control, unlike mere feedback controllers, uses a proactive approach and predicts the system's future behavior. For cloud platforms, this may be an interesting feature.

As explained earlier, the controller must adjust the input variable, such as the number of VMs, to keep the intended value of the output variable (e.g., 90% average CPU load). For this specific reason, a formal relationship must be constructed between the input and the output to evaluate how the output value is affected by a change in the former. This formal relationship is known as state-space function in modern control theory, transfer function in classical control theory, or simply as a performance model. Proportional Integral Derivative controllers consider a simple linear equation, but there are several alternative options that take non-linear approaches into consideration. Furthermore, even several variables of input and output yielding multiple-input multiple-output controllers, rather than a single-input single-output controller.

The suitability of controllers for the role of auto-scaling is highly dependent on the controller model and target system dynamics. The concept of having a controller that streamlines the mechanism of introducing/eliminating resources is very attractive, but the issue is how

Control Theory		
Reference	**Horizontal/Vertical Scaling**	**Metric**
[PH09]	Vertical Scaling	CPU usage and job progress
[LBCP09]	Horizontal Scaling	CPU utilization
[LBC10]	Horizontal Scaling	CPU utilization requests
[PHS+09]	Vertical Scaling	CPU usage and disk I/O
[ATE12]	Horizontal Scaling	Number of requests, service rate

Table 3.9: Summary of Control Theory [LML14]

to develop a reliable model of performance that maps the variables of input and output. Simple reactive controllers can be used for easy-to-predict applications. Nonetheless, it seems appropriate to put more focus on both adaptive and model-predictive controllers that can modify the application model and could be more suitable for producing a general solution for auto-scaling. Table 3.9 shows the scaling type and performance metrics used by different researchers using queuing theory auto-scaling technique.

Time-Series Analysis: Time-series can be used in many fields, including engineering, economics, finance, and bioinformatics, typically to represent a measurement change over time. A time-series is a series of data points, generally measured at uniform time intervals at successive time instants. For example, the number of requests that are received by an application at an interval of one minute. The study of time-series could be used to identify repeating patterns in the workload of the data or to attempt to predict future values. At fixed intervals of every minute, certain performance indicators, like average input workload and CPU load, will be measured periodically. The output of this analysis will be a time-series X, as defined below, of length w containing a series of w occurrences.

$$X = x_t, x_{t-1}, x_{t-2}, ..., x_{t-w+1} \tag{3.5}$$

Time-series methods can be used to predict future metric values, such as expected workload in the future or use of resources. An appropriate auto-scaling behavior can be designed dependent on this predicted value, using, for example, a set of predefined rules, or solving a resource allocation optimization problem. Formally, the time series analysis aims to predict future time-series values based on the last q occurrences denoted as a history window or input window where $q \leq w$. Time-series analytical approaches can be divided into two specific groups: some focus on the direct prediction of future values, while other techniques attempt to identify trends observed in the time-series and then generalize them to forecast values. The averaging method belongs to the group, which focuses on directly predicting future values. To eliminate noise or predict outcomes, averaging methods could be used to optimize a time-series. The predicted value y_{t+1} is determined as the weighted average of the last successive w observations. The general formula is as follows:

$y_{t+r} = a_1 x_t + a_2 x_{t-1}, \ldots..,$ where $a_1, a_2, \ldots\ldots, a_w$ are a collection of weighting factors that must be positive and must have the sum 1. Several methods are defined depending on how these weights are determined:

- *Moving average:* Simple moving average is the mathematical mean of the last input window q or a series of w observations, i.e., all measurements are given equivalent weights of $\frac{1}{w}$.
- *Weighted moving average:* Each observation is given different weights. Usually, the most recent terms in the time-series are given more weight, and the older data are given less weight.
- *Exponential smoothing:* This assigns weights that decrease exponentially over time. A new parameter is introduced, α, a smoothing factor that weakens past data capacity. For single exponential smoothing, the predictor equation is:

$$y_{t+1} = \alpha x_t + (1 - \alpha) y_t \tag{3.6}$$

  where y_{t+1} is the prediction value for the time $t + 1$, x_t is the time t value, and y_t is the prediction made for the time t. Simple exponential smoothing is ideal for time-series with no major pattern shifts, whereas with an established linear trend, double smoothing can be applied to the time-series.

Most analytical methods based on time-series are applied separately for vertical or horizontal scaling. Vertical scaling has a limited range but lowers costs of resource and configuration. While horizontal scaling can enable the application to obtain a much higher performance but at a potentially higher cost. For this reason, the combination of both scaling types, VM upgrading for a small increase in the request frequency and enforcing horizontal scaling for significant changes in the application workload, can be used. In such cases, to predict the estimated number of requests for the next interval, polynomial regression is used. Vertical scaling increases CPU and memory capacity for routine workload adjustments, while horizontal scaling is used to accommodate sudden spikes and bursty load. The proactive decision-making forecasting of time-series can be combined with reactive techniques. Techniques for evaluating time-series are very appealing to incorporate auto-scalers, as they can predict future demands that reach elastic applications. With this information, resources can be given in advance, and the time needed to start new VMs or attach resources to a specific instance can be dealt with. Considering this set of techniques' abilities, their key drawback is based on the accuracy of prediction. It depends heavily on the target application, burstiness and the input workload pattern, the metrics being considered, the duration of prediction, and the history window, as well as the particular technique being used. Table 3.10 shows the scaling type and performance metrics used by different researchers using the queuing theory auto-scaling technique.

Time Series Analysis		
Reference	**Horizontal/Vertical Scaling**	**Metric**
[LBCP09]	Horizontal Scaling	CPU load of all VMs
[GSLI11]	Both	Number of requests
[LZ10]	Horizontal Scaling	CPU load, Number of requests, Number of VMs
[CGS03]	Horizontal Scaling	Request rate and service demand I/O

Table 3.10: Summary of Time Series Analysis [LML14]

3.2.4 Middleware Layer Conclusion

The middleware lies between the OS and the application in a distributed computing system. It is software that offers services other than those offered by the OS to software applications. Application and database servers and message brokers are the components of the middleware.

The scalability of the middleware as a whole involves mapping performance demands to the offered underlying resources. Under-provisioning resources eventually hurt performance, while over-provisioning resources can lead to idle instances, leading to needless costs. There are several auto-scaling strategies that can be used to determine when to scale out/up or scale in/down. For the scalability of message brokers, we can use Apache Kafka or clustering of MQTT based on message brokers. Apache Kafka is built based on a broker-based message-oriented middleware as a distributed streaming platform. The goal of Kafka is to provide a medium for high-performance input data sequence processing, and its major design objectives are scalability, reliable data transfer, fault tolerance, and fast processing. Kafka uses a pull-base technique in which messages can only be pulled by subscribed users after the message has been hard-drive-persisted at each replication, unlike other message brokers. The clustering of message brokers includes the load balancer at the front-end and the MQTT broker cluster at the backend. To perform the MQTT load balancing, Nginx is running as a front-end server.

TSDBs are being used to store and handle TSD and are best to use when it comes to IoT data. InfluxDB [Inf], MonetDB [Mona], and Druid [Dru] are the suitable TSDBs for the TSD. In terms of consistency, accessibility, and immutability, Druid and InfluxDB provide mostly comparable characteristics. InfluxDB, however, supports more programming language interfaces than Druid. InfluxDB also has a Time Structured Merge Tree engine that increases ingestion and compression of data at higher rates. For data scalability, distributed database architecture needs to be adopted instead of a monolithic approach. InfluxDB offers a high-availability open-source replication solution while many users use sharding in the open-source version as a workaround for the missing clustering feature.

For the scalability of MBP middleware as a whole, reactive scaling strategy is suitable because of the nature of the load. A static threshold-based reactive auto-scaling strategy can be used, and its' threshold parameters used for scaling decisions can be optimized over time. The clustering of message brokers is an applicable approach for achieving the scalability of message broker in the MBP. Because the MBP currently uses a push-based approach for data from message brokers, and Kafka is applicable when data needs to be pulled. In the MBP, we are using a monolithic approach that is not suited for data replication and scalability. The distributed approach should be used in the MBP with a cluster of databases for attaining replication and scalability of data.

3.3 Network Layer

The network layer works for the transmission of received data from one device to another or a sub-system located in different networks. It is responsible for connecting network devices, smart things, and servers. It is also used for processing and transmitting sensor data. A gateway is used at the network layer, which provides an interface to connect the devices to other systems. It also provides a mechanism for translating between diverse protocols, payload formats, and communication technologies. The scalability of gateways is the main task for the network layer, which improves communication latency between devices and middleware.

3.3.1 Scalability of Network Layer

The network between devices and an IoT platform has to be able to manage both the traffic created by the IoT devices and the servers of the platform. On the other hand, more traffic is generated by accessing applications. Since all devices and applications connect to one of the system's respective access points, these two points appear to be bottlenecks if the network is unable to accommodate the number of incoming requests. Devices generate data and are queried and controlled over the web. These devices are connected directly to the platform or with the use of a gateway. The primary purpose of a gateway is to provide a bridge between different types of communication technologies, which often differ in terms of connectivity types, interfaces, or protocols. An IoT gateway builds the bridge between sensors and actuators at the one hand and the intranet or internet on the other. It comprises a lot of other capabilities, such as filtering of data that is not necessary at the backend, and security implementation. The network may become the bottleneck because the network bandwidth constrains the data flow. If an IoT platform is delivering/receiving a higher volume of data from/to sensors/actuators than what is recommended by the existing capacity of the network, a network bottleneck will occur.

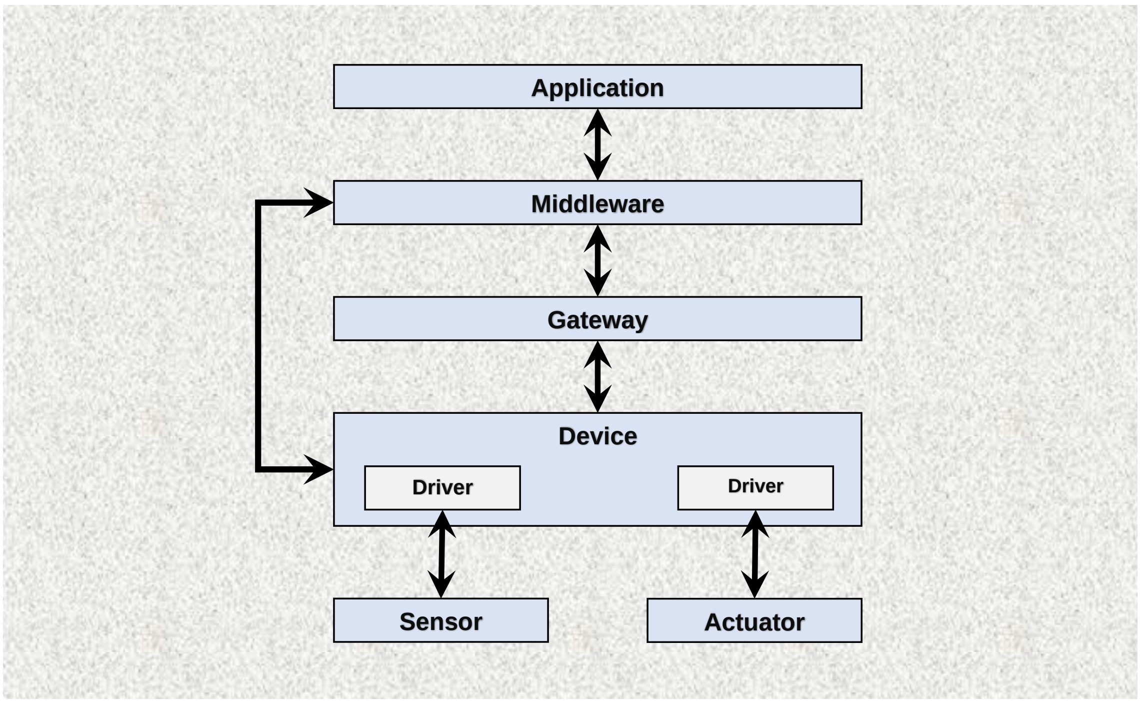

Figure 3.17: IoT Platform Architecture [GBF+18]

3.4 Device Layer

The device layer of an IoT platform encompasses devices, sensors, and actuators. The device is a hardware component that has storage and processing capabilities. It forms a connection with the middleware using driver software running on it. Devices are the physical environment's point of entry into the digital world. The sensors and actuators are the hardware components connected to devices. A sensor detects the physical environment data, such as humidity, sound, or light, while the actuator manipulates the real environment. The main task at the device layer is to support unlimited devices.

3.4.1 Scalability of Device Layer

Physical Devices and Actuators are at the bottom level of the IoT platform architecture. These devices can be connected to the platform with or without the use of a gateway, as depicted in Figure 3.17. At the device layer, we aim at supporting as many devices as possible. In the case of gateway connectivity, the task of supporting unlimited devices becomes dependent on the gateway scalability. In such a scenario, achieving gateway scalability also satisfies our goal for the device layer. Otherwise, one can add as many devices as possible, and the platform will support this because of its scalable middleware.

4 Implementation

In this chapter, we discuss the prototypical implementation of the scalability of middleware as a whole and message brokers. We also discuss the tools and technologies used for the execution of these tasks.

4.1 Scalablity of the Middleware as a whole

For achieving scalability of middleware, we implement a provisioning and monitoring system which uses static threshold based auto-scaling techniques for provisioning and de-provisioning of middleware. The resource usage of the VM is controlled in threshold-based auto-scaling. There are two sets of rules: one to scale-up and one to scale-down for every performance indicator. There are several performance indicators available that can be used for making a provisioning or a de-provisioning decision. We use the following performance indicators for deciding on scaling decisions.

- CPU usage
- Disk usage
- Memory usage
- Requests load

Node exporter is installed on each VM and collects OS metrics. It is a hardware and OS metrics exporter and enables the calculation of various computer resources, such as the use of memory, disk, and CPU. We use the Prometheus database for storing the matrices collected by node exporter. Prometheus is an open-source time-series database, which serves as the monitoring system storage tier. It gets the addresses of the targets, whose data need to be scraped, from the configuration file. It is specifically designed to track and collect metrics. It includes the multi-dimensional data query language called PromQL for data extraction. We provide a reverse proxy server (Nginx), which acts as a load balancer in front of VMs. Nginx distributes incoming user requests to different instances of our IoT platform installed on separate VMs. There are multiple load balancing methods (e.g., round-robin, session persistence, weighted load balancing), which can be configured in Nginx for the distribution of incoming requests.

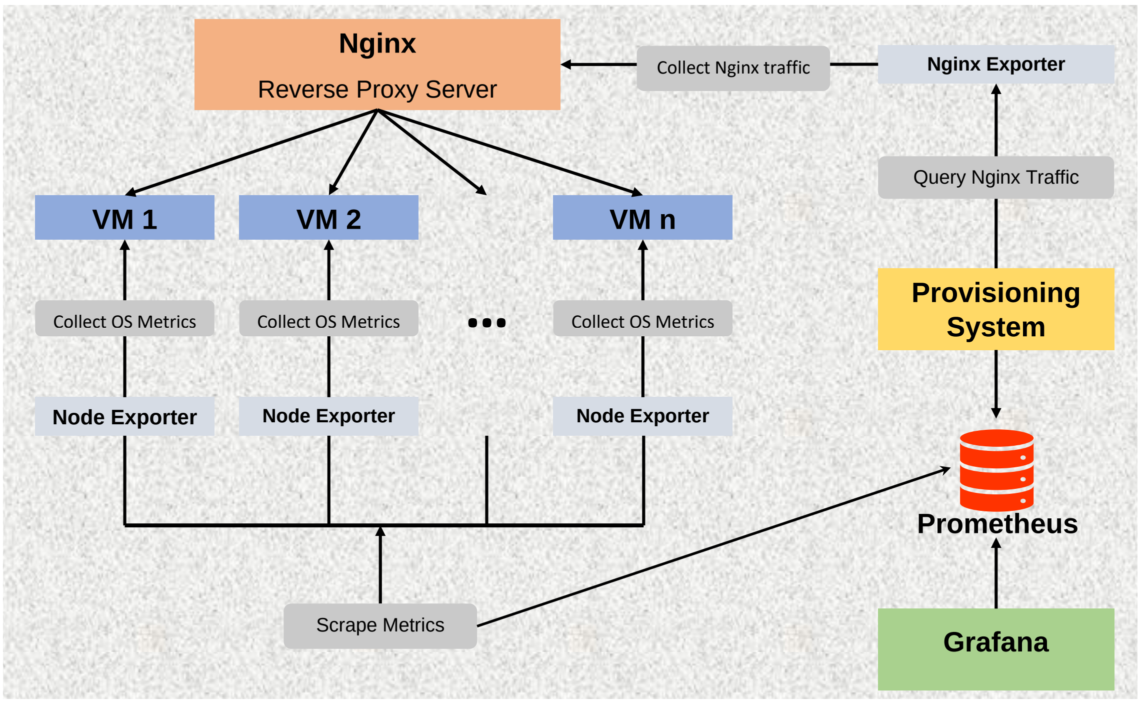

Figure 4.1: Scalability of Middleware

Nginx, Prometheus, Grafana, and Nginx exporter are installed in front of the VMs running the MBP. Nginx exporter is responsible for collecting the data related to Nginx reverse proxy server, such as response time and requests load (incoming traffic) on each MBP instance. Grafana is used for visualization, which queries metrics data from Prometheus and displays it as a dashboard for manual observations.

The provisioning system is implemented in Java in which we query our performance indicators, as discussed above, from Prometheus and also the traffic data of Nginx, as shown in Figure 4.1, from time to time. Our provisioning system reads the static threshold variables and file locations of Nginx and Prometheus config files from a user-specified configuration file before querying Prometheus. Once the system starts collecting matrices, it compares each performance indicator with the user-defined threshold values. If any of the performance indicators violate the upper threshold for a user-specified amount of time, then a scale-out decision will be made. In case of the violation of a lower threshold for a given amount of time, a scale-in decision will be made. In the case of provisioning, a new VM is configured by installing and running the MBP and the node exporter on it. Upon successful configuration, the addresses of newly installed VMs are added to the Prometheus and Nginx configurations, and we also restart Nginx by establishing an ssh connection from the provisioning system. Prometheus automatically starts scraping matrices of newly created VMs also without reload or restart, but Nginx needs to be restarted so that it starts distributing user requests to them as well. In the case of de-provisioning, a VM is removed from the cluster of VMs and the addresses of deleted VMs are removed from the Prometheus and Nginx configurations. We again restart Nginx by establishing an ssh

Listing 4.1 Exemplary Prometheus Targets File

```
[
{
"targets":["IPAddress:9100"],
"labels":{"job":"JobName"}
},
{
"targets":["IPAddress:9100"],
"labels":{"job":"JobName"}
}
]
```

Listing 4.2 Exemplary Nginx Configuration File

```
events{
}
http {
upstream mbp {
server IPAddress:8080;
server IPAddress:8080;
}
server {
access_log /var/log/nginx/access.log upstreamlog;
listen 80;
location / {
proxy_http_version      1.1;
proxy_pass_request_headers on;
proxy_set_header        Host            $host;
proxy_set_header        X-Real-IP       $remote_addr;
proxy_set_header        X-Forwarded-For $proxy_add_x_forwarded_for;
proxy_pass http://mbp;
}
}
}
```

connection from the provisioning system to the VM running Nginx so that it does not consider the deleted VM anymore for the distribution of requests. Prometheus automatically stops scraping the matrices for a deleted VM. After every provisioning or de-provisioning decision, the provisioning system goes into idle mode for the amount of time given by the user. Listing 4.1 shows the Prometheus targets file which contains the information about the VMs. This file is then referred to in the Prometheus configuration file. Listing 4.2 shows the exemplary Nginx configuration file.

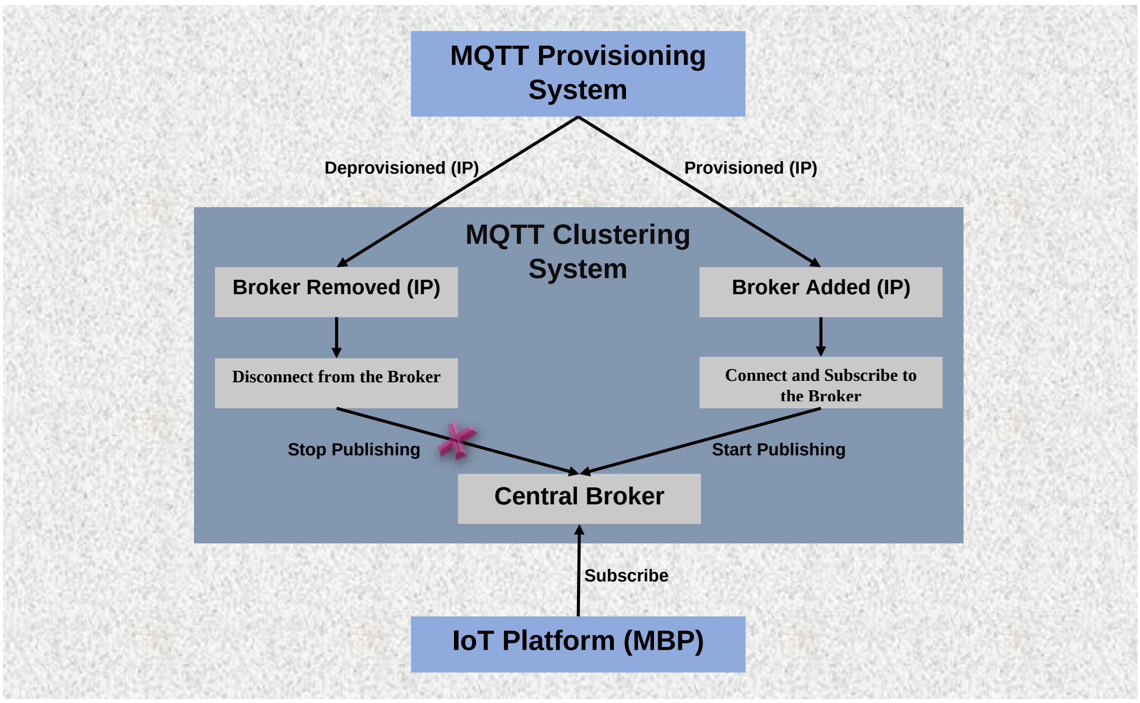

Figure 4.2: MQTT Clustering Connectivity

4.2 Scalability of the Message Brokers

For the scalability of message brokers, we implement clustering of message brokers instead of Apache Kafka because of the architecture of the MBP. The clustering of message brokers includes the load balancer at the front-end and the MQTT broker cluster at the backend. To perform the MQTT load balancing, Nginx is running as a front-end server. For the provisioning and de-provisioning of message brokers, we use the similar approach to middleware scalability as discussed above. However, we need some extra mechanism to connect the platform MBP with all of the message brokers in the cluster. Currently, MBP does not support connectivity to multiple brokers at a time as it does not have functionality for the discovery of message brokers. There are three ways to obtain data from multiple brokers in the platform.

- Getting data from brokers in a cluster and storing it to a central database. MBP is then connected to this database and fetches the data. In this approach, there is an overhead of storing and querying data from a database.
- Implement the search functionality in the MBP to discover message brokers in a cluster and gets connected to each of them.
- A centralized broker getting the data from brokers available in a cluster and the MBP is connected to it. In this approach, it is possible that this centralized broker becomes the bottleneck.

We use a centralized broker technique for the prototypical implementation of connecting the MBP to multiple brokers. For getting data from multiple brokers, we implement a MQTT clustering system which is responsible for getting data from brokers in a cluster and publishing this data to a centralized broker, as depicted in Figure 4.2. This system runs side-by-side with the provisioning and monitoring system and it exposes some endpoints for getting information about addition or removal of message brokers in a cluster. The provisioning system informs the MQTT clustering system over the exposed endpoints about the provisioning and de-provisioning of message brokers in a cluster. We are using OpenStack cloud platform for provisioning and de-provisioning of the resources.

5 Summary and Future Work

In this chapter, we present a summary of the tasks and the best suitable solution for each task of this thesis. We will then provide possible future work on the scalability feature of IoT platforms in general.

5.1 Summary

The goals of this thesis were to examine every software stack layer of an IoT platform and explain different methods to make each layer scalable. Enabling multi-tenancy at the application layer and attaining scalability of message brokers, database, middleware as a whole, network layer, and device layer were the primary tasks for enabling multi-tenant scalable IoT platforms. To provide a solution to these tasks, we went through the research work already done in the area of scalability and multi-tenancy. We addressed multiple solutions for each task and also provided the best suitable solution for each task at the corresponding layer of an IoT platform.

Multiple instance multi-tenancy and native multi-tenancy were the two main architectural strategies that we discussed for achieving multi-tenancy. Multiple instances multi-tenancy supports each tenant with its dedicated application instance over shared hardware, and/or a middleware server in a hosting environment. It can be applied at the infrastructure level or middleware level of the software stack. Native multi-tenancy is attaining at the application level that supports all tenants by a single shared application instance over various hosting resources. Data isolation and authentication were two significant tasks needed to be performed for attaining application-level multi-tenancy. From the tenants' data isolation, a shared schema approach with a shared database was suggested as convenient for the MBP because the data were more or less similar. The shared schema approach was also less costly than the other alternate options.

The scalability of the middleware as a whole involves mapping performance demands to the offered underlying resources. Several auto-scaling strategies were discussed that could be used to determine when to scale out/up or scale in/down. We implemented a provisioning and monitoring system which uses static threshold-based auto-scaling techniques for provisioning and de-provisioning of middleware. For the scalability of message brokers, we could have used Apache Kafka or clustering of MQTT based message brokers. We used the clustering technique for the scalability of message brokers. TSD is data gathered over time consisting of a series of observations (e.g., sensor data, data from smart grids).

TSDBs are being used to store and handle TSD. In terms of consistency, accessibility, and immutability, Druid and InfluxDB provide mostly comparable characteristics. InfluxDB, however, supports more programming language interfaces than Druid. InfluxData offers a high-availability open-source replication solution.

A gateway is used at the network layer, which provides an interface to connect the devices to other systems. The primary purpose of a gateway is to provide a bridge between different types of communication technologies, which often differ in terms of connectivity types, interfaces, or protocols. The network may become the bottleneck because the network bandwidth constrains the data flow. If an IoT platform is delivering/receiving a higher volume of data from/to sensors/actuators than what is recommended by the existing capacity of the network, a network bottleneck will occur. The device layer of an IoT platform encompasses devices, sensors, and actuators. Devices can be connected to the platform with or without the use of a gateway. At the device layer, we aimed at supporting as many devices as possible. In the case of gateway connectivity, the task of supporting unlimited devices became dependent on the gateway scalability. In such a scenario, achieving gateway scalability also satisfies our goal for the device layer.

5.2 Future Work

The research work discussed in this thesis opened various ways of achieving the scalability of an IoT platform. Middleware is the core part of any platform that should be highly scalable. Several auto-scaling strategies have been discussed, and we used a static threshold based scaling strategy for the scalability of middleware. However, for the future, a proactive scaling strategy, such as reinforcement learning techniques, is a better option because RL algorithms identify the best management strategy for a target scenario without any prior information. Another way to make an IoT platforms highly scalable is by breaking down the software code into loosely coupled services, known as microservices. Microservices are a way of breaking an application into standalone, independent applications. These services can be deployed individually and can be scaled when needed. In this way, we achieve the scalability at a service level instead of at the platform instance level.

For the scalability of message brokers, we used the clustering of MQTT based message brokers because it is suited best for our IoT platform MBP. Apache Kafka is an open-source software designed to handle massive amounts of data that can be used in the future to support an unlimited number of devices. Kafka uses a pull-based model, unlike MQTT message brokers, in which consumers pull the data from the Kafka brokers. It is the most advanced message broker and has been created to address the need to transfer enormous amounts of data from producers to numerous consumers in an efficient and scalable manner.

Bibliography

[ABG+18] A. O. Abdul, J. Bass, H. Ghavimi, N. MacRae, P. Adam. “Multi-tenancy Design Patterns in SaaS Applications: A Performance Evaluation Case Study”. In: *International Journal of Digital Society (IJDS)* (2018). DOI: 10.20533/ijds.2040.2570.2018.0168 (cit. on pp. 17, 20, 27, 32, 36).

[AHPW19] D. Arnst, T. Herpich, V. Plenk, A. Wöltche. “Comparative Evaluation of Database Read and Write Performance in an Internet of Things Context”. In: *International Journal on Advances in Internet Technology Volume 12, Number 1 & 2, 2019* (2019). URL: http://www.iariajournals.org/internet_technology/ (cit. on pp. 22, 38).

[Apa] Apache. *Flexible & Powerful Open Source Multi-Protocol Messaging*. URL: https://activemq.apache.org/ (cit. on pp. 22, 42).

[ATE12] A. Ali-Eldin, J. Tordsson, E. Elmroth. “An adaptive hybrid elasticity controller for cloud infrastructures”. In: *2012 IEEE Network Operations and Management Symposium*. IEEE. 2012, pp. 204–212. DOI: 10.1109/NOMS.2012.6211900 (cit. on p. 60).

[BGS+09] P. Bodık, R. Griffith, C. Sutton, A. Fox, M. Jordan, D. Patterson. “Statistical Machine Learning Makes Automatic Control Practical for Internet Datacenters.” In: HotCloud’09 (2009). URL: http://dl.acm.org/citation.cfm?id=1855533.1855545 (cit. on p. 50).

[BHD13] E. Barrett, E. Howley, J. Duggan. “Applying reinforcement learning towards automating resource allocation and application scalability in the cloud”. In: *Concurrency and Computation: Practice and Experience* 25.12 (2013), pp. 1656–1674. DOI: https://doi.org/10.1002/cpe.2864 (cit. on p. 55).

[BKF17] A. Bader, O. Kopp, M. Falkenthal. “Survey and comparison of open source time series databases”. In: *Datenbanksysteme für Business, Technologie und Web (BTW 2017)-Workshopband* (2017) (cit. on pp. 17, 21, 38).

[BZ10] C.-P. Bezemer, A. Zaidman. “Multi-tenant SaaS applications: maintenance dream or nightmare?” In: *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*. ACM. 2010, pp. 88–92. DOI: 10.1145/1862372.1862393 (cit. on p. 21).

[BZP+10] C.-P. Bezemer, A. Zaidman, B. Platzbeecker, T. Hurkmans, A. Hart. “Enabling multi-tenancy: An industrial experience report”. In: *2010 IEEE International Conference on Software Maintenance*. IEEE. 2010, pp. 1–8. DOI: 10.1109/ICSM.2010.5609735 (cit. on pp. 21, 26).

[CCW06] F. Chong, G. Carraro, R. Wolter. “Multi-tenant data architecture”. In: *MSDN Library, Microsoft Corporation* (2006), pp. 14–30 (cit. on pp. 21, 33, 35).

[CGS03] A. Chandra, W. Gong, P. Shenoy. “Dynamic resource allocation for shared data centers using online measurements”. In: *International Workshop on Quality of Service*. Springer. 2003, pp. 381–398. DOI: 10.1145/781027.781067 (cit. on p. 62).

[CMKS09] T. C. Chieu, A. Mohindra, A. A. Karve, A. Segal. “Dynamic scaling of web applications in a virtualized cloud computing environment”. In: *2009 IEEE International Conference on e-Business Engineering*. IEEE. 2009, pp. 281–286. DOI: 10.1109/ICEBE.2009.45 (cit. on pp. 19, 20, 52).

[DE17] P. Dobbelaere, K. S. Esmaili. “Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations: Industry Paper”. In: *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. ACM. 2017, pp. 227–238. DOI: 10.1145/3093742.3093908 (cit. on p. 22).

[DKM+11] X. Dutreilh, S. Kirgizov, O. Melekhova, J. Malenfant, N. Rivierre, I. Truck. “Using reinforcement learning for autonomic resource allocation in clouds: towards a fully automated workflow”. In: *ICAS 2011, The Seventh International Conference on Autonomic and Autonomous Systems*. 2011, pp. 67–74 (cit. on p. 55).

[DMM+10] X. Dutreilh, A. Moreau, J. Malenfant, N. Rivierre, I. Truck. “From data center resource allocation to control theory and back”. In: *2010 IEEE 3rd international conference on cloud computing*. IEEE. 2010, pp. 410–417. DOI: 10.1109/CLOUD.2010.55 (cit. on p. 53).

[Doc] Docker. *Getting started with swarm mode*. URL: https://docs.docker.com/engine/swarm/swarm-tutorial/ (cit. on pp. 23, 44).

[Dru] Druid. *High performance real-time analytics database*. URL: https://druid.apache.org/ (cit. on pp. 21, 38, 62).

[GBF+18] J. Guth, U. Breitenbücher, M. Falkenthal, P. Fremantle, O. Kopp, F. Leymann, L. Reinfurt. “A detailed analysis of IoT platform architectures: concepts, similarities, and differences”. In: *Internet of Everything*. Springer, 2018, pp. 81–101 (cit. on pp. 17, 18, 26, 27, 64).

[GSH+07] C. J. Guo, W. Sun, Y. Huang, Z. H. Wang, B. Gao. “A framework for native multi-tenancy application development and management”. In: *The 9th IEEE International Conference on E-Commerce Technology and The 4th IEEE International Conference on Enterprise Computing, E-Commerce and E-Services (CEC-EEE 2007)*. IEEE. 2007, pp. 551–558. DOI: 10.1109/CEC-EEE.2007.4 (cit. on pp. 21, 28, 29, 31).

[GSLI11] H. Ghanbari, B. Simmons, M. Litoiu, G. Iszlai. “Exploring Alternative Approaches to Implement an Elasticity Policy”. In: *2011 IEEE 4th International Conference on Cloud Computing*. July 2011, pp. 716–723. DOI: 10.1109/CLOUD.2011.101 (cit. on p. 62).

[HBS+16] P. Hirmer, U. Breitenbücher, A. C. F. da Silva, K. Képes, B. Mitschang, M. Wieland. “Automating the Provisioning and Configuration of Devices in the Internet of Things”. Englisch. In: *Complex Systems Informatics and Modeling Quarterly* 9 (Dezember 2016), pp. 28–43. ISSN: 2255 - 9922. DOI: 10.7250/csimq.2016-9.02. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=ART-2016-23&engl=0 (cit. on p. 17).

[HGGG12] R. Han, L. Guo, M. M. Ghanem, Y. Guo. “Lightweight resource scaling for cloud applications”. In: *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. IEEE Computer Society. 2012, pp. 644–651. DOI: 10.1109/CCGrid.2012.52 (cit. on p. 53).

[HHL12] C.-L. Hung, Y.-C. Hu, K.-C. Li. “Auto-scaling model for cloud computing system”. In: *International Journal of Hybrid Information Technology* 5.2 (2012), pp. 181–186 (cit. on p. 20).

[HMC+12] M. Z. Hasan, E. Magana, A. Clemm, L. Tucker, S. L. D. Gudreddi. “Integrated and autonomic cloud resource scaling”. In: *2012 IEEE network operations and management symposium*. IEEE. 2012, pp. 1327–1334. DOI: 10.1109/NOMS.2012.6212070 (cit. on pp. 50, 53).

[HWBM16a] P. Hirmer, M. Wieland, U. Breitenbücher, B. Mitschang. “Automated Sensor Registration, Binding and Sensor Data Provisioning”. Englisch. In: *Proceedings of the CAiSE’16 Forum, at the 28th International Conference on Advanced Information Systems Engineering (CAiSE 2016)*. Vol. 1612. CEUR Workshop Proceedings. Ljubljana, Slovenia: CEUR-WS.org, June 2016, pp. 81–88. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2016-22&engl=0 (cit. on p. 17).

[HWBM16b] P. Hirmer, M. Wieland, U. Breitenbücher, B. Mitschang. “Dynamic Ontology-based Sensor Binding”. Englisch. In: *Advances in Databases and Information Systems. 20th East European Conference, ADBIS 2016, Prague, Czech Republic, August 28-31, 2016, Proceedings*. Vol. 9809. Information Systems and Applications, incl. Internet/Web, and HCI. Prague, Czech Republic: Springer International Publishing, Aug. 2016, pp. 323–337. ISBN: 978-3-319-44038-5. DOI: 10.1007/978-3-319-44039-2. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2016-25&engl=0 (cit. on p. 25).

[HWS+16] P. Hirmer, M. Wieland, H. Schwarz, B. Mitschang, U. Breitenbücher, S. G. Sáez, F. Leymann. “Situation recognition and handling based on executing situation templates and situation-aware workflows”. Englisch. In: *Computing* (Oktober 2016), pp. 1–19. DOI: 10.1007/s00607-016-0522-9. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=ART-2016-12&engl=0 (cit. on p. 25).

[Inf] InfluxData. *Time series database*. URL: https://www.influxdata.com/get-influxdb/ (cit. on pp. 21, 38, 62).

[JL17] V. John, X. Liu. “A survey of distributed message broker queues”. In: *arXiv preprint arXiv:1704.00411* (2017) (cit. on p. 22).

[JPV+17] P. Jutadhamakorn, T. Pillavas, V. Visoottiviseth, R. Takano, J. Haga, D. Kobayashi. “A scalable and low-cost MQTT broker clustering system”. In: *2017 2nd International Conference on Information Technology (INCIT)*. IEEE. 2017, pp. 1–5. DOI: 10.1109/INCIT.2017.8257870 (cit. on pp. 18, 23, 44, 45).

[Kaf] A. Kafka. *Apache Kafka: A distributed streaming platform*. URL: https://kafka.apache.org/ (cit. on pp. 22, 42).

[KCD+17] G. Karataş, F. Can, G. Doğan, C. Konca, A. Akbulut. “Multi-tenant architectures in the cloud: a systematic mapping study”. In: *2017 International Artificial Intelligence and Data Processing Symposium (IDAP)*. IEEE. 2017, pp. 1–4. DOI: 10.1109/IDAP.2017.8090268 (cit. on pp. 17, 20, 28, 32, 34).

[Ken] Kendall. *Introduction to queueing theory*. URL: http://www.mathcs.emory.edu/~cheung/Courses/558/Syllabus/00/queueing/queueing.html (cit. on p. 56).

[KNR+11] J. Kreps, N. Narkhede, J. Rao, et al. “Kafka: A distributed messaging system for log processing”. In: *Proceedings of the NetDB*. 2011, pp. 1–7 (cit. on pp. 23, 46, 47).

[LBC10] H. C. Lim, S. Babu, J. S. Chase. “Automated control for elastic storage”. In: *Proceedings of the 7th international conference on Autonomic computing*. ACM. 2010, pp. 1–10. DOI: 10.1145/1809049.1809051 (cit. on p. 60).

[LBCP09] H. Lim, S. Babu, J. Chase, S. Parekh. “Automated control in cloud computing: Challenges and opportunities”. In: *Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds, ACDC ’09* (June 2009). DOI: 10.1145/1555271.1555275 (cit. on pp. 60, 62).

[LCW08] D. Lucke, C. Constantinescu, E. Westkämper. “Smart factory-a step towards the next generation of manufacturing”. In: *Manufacturing systems and technologies for the new frontier*. Springer, 2008, pp. 115–118 (cit. on p. 17).

[Lig+17] R. A. Light et al. “Mosquitto: server and client implementation of the MQTT protocol.” In: *J. Open Source Software* 2.13 (2017), p. 265. DOI: 10.21105/joss.00265 (cit. on p. 23).

[LML12] T. Lorido-Botrán, J. Miguel-Alonso, J. A. Lozano. “Auto-scaling techniques for elastic applications in cloud environments”. In: *Department of Computer Architecture and Technology, University of Basque Country, Tech. Rep. EHU-KAT-IK-09* 12 (2012), p. 2012 (cit. on pp. 17, 19, 48, 49, 53–55, 57).

[LML14] T. Lorido-Botrán, J. Miguel-Alonso, J. A. Lozano. “A review of auto-scaling techniques for elastic applications in cloud environments”. In: *Journal of grid computing* 12.4 (2014), pp. 559–592. DOI: 10.1007/s10723-014-9314-7 (cit. on pp. 17, 19, 56, 57, 60, 62).

[LZ10] P. Lama, X. Zhou. “Autonomic Provisioning with Self-Adaptive Neural Fuzzy Control for End-to-end Delay Guarantee”. In: *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. Aug. 2010, pp. 151–160. DOI: 10.1109/MASCOTS.2010.24 (cit. on p. 62).

[Mag15] L. Magnoni. “Modern Messaging for Distributed Sytems”. In: *Journal of Physics: Conference Series* 608 (May 2015), p. 012038. DOI: 10.1088/1742-6596/608/1/012038 (cit. on pp. 18, 22, 43).

[Mar] MariaDB. URL: https://mariadb.org/ (cit. on pp. 22, 38).

[MBS11] M. Maurer, I. Brandic, R. Sakellariou. “Enacting SLAs in clouds using rules”. In: *European Conference on Parallel Processing*. Springer. 2011, pp. 455–466 (cit. on p. 53).

[Mona] MonetDB. *An open-source database system*. URL: https://www.monetdb.org/ (cit. on pp. 21, 38, 62).

[Monb] MongoDB. *The database for modern applications*. URL: https://www.mongodb.com/ (cit. on pp. 22, 38).

[MQT] MQTT. *MQTT*. URL: http://mqtt.org/ (cit. on p. 43).

[MSA14] M. M. Murthy, H. Sanjay, J. Anand. “Threshold based auto scaling of virtual machines in cloud environment”. In: *IFIP International Conference on Network and Parallel Computing*. Springer. 2014, pp. 247–256. DOI: 10.1007/978-3-662-44917-2_21 (cit. on pp. 17, 20).

[Ngi] Nginx. *Improve the performance, reliability, and security of your applications*. URL: https://www.nginx.com/ (cit. on p. 43).

[Ope] OpenTSDB. *The Scalable Time Series Database*. URL: http://opentsdb.net/ (cit. on pp. 22, 38).

[Pat] Patton. *Multiple Data Center Replication with InfluxDB*. URL: https://www.influxdata.com/blog/multiple-data-center-replication-influxdb/ (cit. on p. 41).

[PH09] S.-M. Park, M. Humphrey. "Self-Tuning Virtual Machines for Predictable eScience". In: *2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*. May 2009, pp. 356–363. DOI: 10.1109/CCGRID.2009.84 (cit. on p. 60).

[PHS+09] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant. "Automated control of multiple virtualized resources". In: *Proceedings of the 4th ACM European conference on Computer systems*. ACM. 2009, pp. 13–26. DOI: 10.1145/1519065.1519068 (cit. on p. 60).

[QCB18] C. Qu, R. N. Calheiros, R. Buyya. "Auto-scaling web applications in clouds: A taxonomy and survey". In: *ACM Computing Surveys (CSUR)* 51.4 (2018), p. 73. DOI: 10.1145/3148149 (cit. on p. 19).

[Rab] RabbitMQ. *Widely deployed open source message broker*. URL: https://www.rabbitmq.com/ (cit. on pp. 22, 42).

[RBX+09] J. Rao, X. Bu, C.-Z. Xu, L. Wang, G. Yin. "VCONF: A reinforcement learning approach to virtual machines auto-configuration". In: Jan. 2009, pp. 137–146. DOI: 10.1145/1555228.1555263 (cit. on p. 55).

[RMD+06] V. Ricquebourg, D. Menga, D. Durand, B. Marhic, L. Delahoche, C. Loge. "The smart home concept: our immediate future". In: *2006 1st IEEE international conference on e-learning in industrial electronics*. IEEE. 2006, pp. 23–28. DOI: 10.1109/ICELIE.2006.347206 (cit. on p. 17).

[RPHB18] R. Ravichandran, E. Prassler, N. Huebel, S. Blumenthal. "A Workbench for Quantitative Comparison of Databases in Multi-Robot Applications". In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2018, pp. 3744–3750. DOI: 10.1109/IROS.2018.8594241 (cit. on pp. 18, 21).

[San16] T. P. Sanaboyina. *Performance Evaluation of Time series Databases based on Energy Consumption*. 2016. URL: http://urn.kb.se/resolve?urn=urn:nbn:se:bth-13593 (cit. on pp. 22, 38).

[SB18] S. Sen, A. Balasubramanian. "A highly resilient and scalable broker architecture for IoT applications". In: *2018 10th International Conference on Communication Systems & Networks (COMSNETS)*. IEEE. 2018, pp. 336–341. DOI: 10.1109/COMSNETS.2018.8328216 (cit. on p. 23).

[Sir] SiriDB. *Next Gen Time Series Database*. URL: https://siridb.net/ (cit. on p. 21).

[SKP+11] H. Schaffers, N. Komninos, M. Pallot, B. Trousse, M. Nilsson, A. Oliveira. "Smart cities and the future internet: Towards cooperation frameworks for open innovation". In: *The future internet assembly*. Springer. 2011, pp. 431–446. DOI: 10.1007/978-3-642-20898-0_31 (cit. on p. 17).

[SWED16] D. Seybold, N. Wagner, B. Erb, J. Domaschka. “Is elasticity of scalable databases a myth?” In: *2016 IEEE International Conference on Big Data (Big Data)*. IEEE. 2016, pp. 2827–2836. DOI: 10.1109/BigData.2016.7840931 (cit. on p. 39).

[TJDB06] G. Tesauro, N. K. Jong, R. Das, M. N. Bennani. “A hybrid reinforcement learning approach to autonomic resource allocation”. In: *2006 IEEE International Conference on Autonomic Computing*. IEEE. 2006, pp. 65–73. DOI: 10.1109/ICAC.2006.1662383 (cit. on pp. 53, 55, 57).

[USC+08] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, T. Wood. “Agile dynamic provisioning of multi-tier internet applications”. In: *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 3.1 (2008), p. 1. DOI: 10.1145/1342171.1342172 (cit. on p. 57).

[VPR07] D. Villela, P. Pradhan, D. Rubenstein. “Provisioning servers in the application tier for e-commerce systems”. In: *ACM Transactions on Internet Technology (TOIT)* 7.1 (2007), p. 7. DOI: 10.1109/IWQOS.2004.1309357 (cit. on p. 57).

[WLJ14] S. Walraven, B. Lagaisse, W. Joosen. “Application level multi-tenancy: the promise and pitfalls of shared everything architectures”. In: *distri Net Research Group* (2014) (cit. on pp. 21, 22, 30).

[ZCS07] Q. Zhang, L. Cherkasova, E. Smirni. “A regression-based analytic model for dynamic resource provisioning of multi-tier applications”. In: *Fourth International Conference on Autonomic Computing (ICAC’07)*. IEEE. 2007, pp. 27–27. DOI: 10.1109/ICAC.2007.1 (cit. on p. 57).

[Zer] ZeroMQ. *An open-source universal messaging library*. URL: https://zeromq.org/ (cit. on pp. 22, 42).

All links were last followed on February 29, 2020.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature