LAB University of Applied Sciences

Leveraging Serverless Computing with Azure Functions for High-Performance Backend Services in Precast Wall Designer

LAB University of Applied Sciences
Bachelor of Engineering, Industrial Information Technology
2025
Kha Tieu

Abstract

Author(s)	Publication type	Completion year
Kha Tieu	Thesis, UAS	2025
	Number of pages	
	31	

Title of the thesis

Leveraging Serverless Computing with Azure Functions for High-Performance Backend Services in Precast Wall Designer

Degree, Field of Study

Bachelor's Degree Programme in Industrial Information Technology

Name, title and organisation of the client

Peikko Group Corporation

Abstract

This thesis explores the integration of serverless computing into a microservices backend architecture to enhance scalability, maintainability, and performance in cloud-based applications using Microsoft technologies. Conducted in collaboration with Peikko, the research focuses on the backend of the Precast Wall Designer web application, a long-term project that follows microservices architecture. As the thesis was carried out in the initial phase of the project, Peikko company required the backend to support increasing business logic and intensive computational workloads. To meet this requirement, the system adopted a partial architecture that used serverless Azure Functions alongside the backend. This approach allows the backend, developed with ASP.NET Core framework and Clean Architecture principles, to offload intensive computational tasks to serverless functions while focusing on handling the core application logic. Using the Design Science Research methodology, the thesis develops an architectural artifact and evaluates its effectiveness through expert feedback and load testing. Results confirm that the computation using this architecture performs well under increasing user loads, improves maintainability, and supports scalability. This thesis can be used as a practical design pattern for organizations that do not intend to fully commit to either microservices or serverless architectures in their backend systems.

Keywords

Backend, Serverless computing, Azure Functions, Azure Web App, Azure Service Bus, Clean architecture, Computational Offloading

Contents

1 Introduction 1
2 Background 3
2.1 Software architectures patterns 3
2.1.1 Monolithic architecture 3
2.1.2 Microservices 5
2.2 Virtualization 7
2.3 Cloud computing 8
2.3.1 Cloud service models 9
2.3.2 Serverless computing 10
3 Implementation 12
3.1 Project introduction 12
3.2 Design requirements and goals 12
3.3 System architecture overview 13
3.3.1 System design overview 13
3.3.2 Architectural decision justification 16
3.4 Backend architecture 18
3.5 Integrating serverless functions 21
3.6 Deployment and CI/CD 22
4 Evaluation 24
5 Summary 27
References 28

Symbols and abbreviations

Abbreviations

API	Application Programming Interface
AWS	Amazon Web Services
CI/CD	Continuous Integration and Continuous Delivery
CNCF	Cloud Native Computing Foundation
DB	Database
DSR	Design Science Research
HTTP	Hypertext Transfer Protocol
FaaS	Function as a Service
IaaS	Infrastructure as a Service
IBM	International Business Machines Corporation
ID	Identifier
PaaS	Platform as a Service
REST	Representational State Transfer
SQL	Structured Query Language
YAML	Yet Another Markup Language

1 Introduction

The rapid development of cloud computing has transformed the design and deployment of modern software systems. One of the most popular concepts emerging with this transition is the microservices architecture. Microservices split an application into a set of independently running services. By creating loosely coupled services, microservices have successfully shown significant scalability and performance gains for large enterprises such as LinkedIn (Mauro 2015) and Netflix (Ihde & Parikh 2015). However, microservices also introduce complexities in service orchestration and deployment. Therefore, developers must rigorously assess the benefits and trade-offs of microservices in an application.

Initiated by Peikko company as a long-term project, the Precast Wall Designer web application follows a microservices-based approach. The frontend and backend of the application are managed and deployed independently by two separate teams of students. This distributed architecture ensures better scalability and loose dependencies for future updates or development. However, as the codebase and the business logic scale, the microservices become overly complicated for students to develop further or maintain. Because of that, the system design must adhere to the microservices architecture while sustaining the ease of development and the efficiency in heavy computational processing.

Serverless computing has emerged as a practical solution to address this challenge. In particular, Function-as-a-Service (FaaS) like Azure Functions allows developers to offload intensive computing tasks from a server. As a result, instead of splitting into more services, a backend within a microservices architecture can delegate computing logic to serverless functions. This approach helps keep the backend focused on handling core application logic while executing computations within scalable serverless functions.

While existing literature on serverless computing primarily focuses on fully serverless architectures or migration to microservices, there is very limited research on partial architecture using serverless computing. Collaborating with Peikko company, this study addresses this gap by presenting a practical approach to integrating serverless Azure Functions into a backend within a microservices ecosystem of the Peikko Precast Wall Designer web application. Based on the Design Science (DSR) methodology, the research evaluates the effectiveness of the proposed partially serverless backend architecture.

The Design Science Research (DSR) methodology was selected due to its alignment with the practical and solution-oriented focus of this thesis. DSR is particularly effective for research aimed at creating and evaluating artifacts that address specific problems within information science. In this case, the research focuses on designing and implementing a partial architecture that uses backend and serverless functions to enhance the performance,

scalability, and maintainability of the overall microservices architecture. Apart from building the solution, it is crucial to evaluate its effectiveness through feedback from industry experts and performance testing after deployment. By following the DSR iterative structure for development and validation, the thesis can ensure that the proposed solution is both practically and theoretically relevant. Ultimately, the contribution of this thesis is an architectural artifact that includes a backend service as an entry point for requests and serverless services to efficiently handle the actual computational tasks within the overall microservices structure of the application.

2 Background

2.1 Software architectures patterns

Understanding software architecture is essential to comprehend how to integrate serverless functions because it defines the structure and principles of an application. According to David Garlan (2000, 91–101), a well-designed architecture can guarantee that a system meets requirements in performance, reliability, portability, scalability, and interoperability. Monolithic architecture and microservices architecture are the two most popular architectural design patterns in web development. Figure 1 shows the difference between monolithic and microservice architectures. While both have distinct advantages and trade-offs, the architectural choice heavily relies on the project requirements and team maturity. This section introduces and evaluates the pros and cons of two architectures.

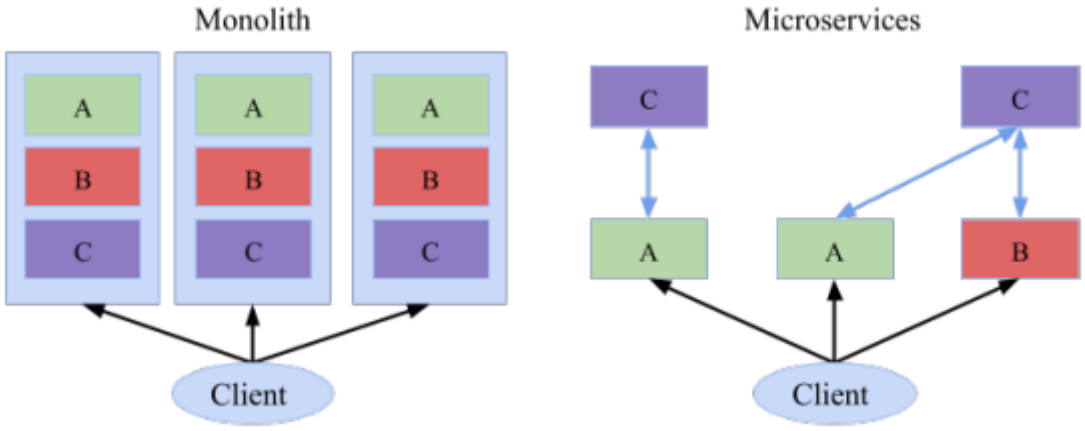

Figure 1. Monolithic and microservice architectures (Paakkunainen 2019)

2.1.1 Monolithic architecture

Monolithic architecture, also known as layered architecture or n-tier architecture, is the most well-known architecture pattern. It is a software model with a single codebase that performs several business tasks (Powell & Smalley 2019). This single codebase has many layers with smaller components. Moreover, each layer serves a particular task within the application. In the n-tier architecture, the number of layers can vary. More complex business applications may introduce additional layers to split the application into even smaller units of concern.

A three-tier architecture, a form of n-tier architecture, organizes the application into 3 layers: presentation, business, and data layers, as shown in Figure 2. Every layer serves a specific purpose and depends on the one below it to support the clear separation of concerns and prevent circular dependencies. At the top, the presentation layer forwards user requests to

the business layer and shows processed responses through a graphical user interface (GUI). As the core of the entire application, the business layer orchestrates the request flows between the presentation layer and the data layer. This layer rules how data is processed and how services coordinate. Some of the components of this layer are services, controllers, and domain models. Lastly, at the bottom, the data layer is responsible for storing and accessing data. This layer ensures data integrity and provides an interface for the business layer to perform CRUD (Create, Read, Update, Delete) operations in the database.

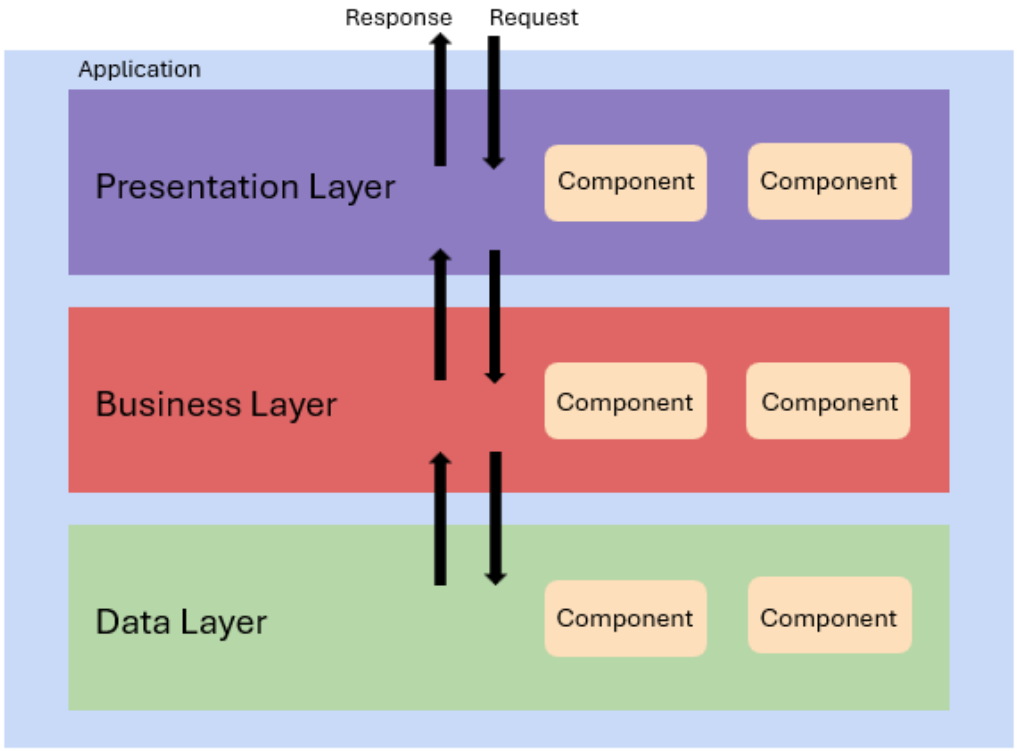

Figure 2. Three-tier architecture pattern

Simplicity is the most notable advantage of monolithic architecture. This architecture simplifies development, testing, deploying, and scaling processes when the codebase size remains relatively small (Richardson 2017). A common practice that reinforces this simplicity is the monolithic repository (monorepo), where the entire application's codebase sits in a single repository. A survey conducted among Google engineers has shown a preference for monolithic repositories because of codebase visibility and simplified dependency management (Jaspan et al. 2018, 225-234). In terms of visibility, developers who have access to the repository can view, search, and modify the code. Also, the availability of source code within a monolithic repository assists maintainers with testing and evaluating the performance of changes before committing (Potvin & Levenberg 2016, 78-87).

However, monolithic architecture and monolithic repositories are distinct concepts. While the development of monolithic applications uses monolithic repositories, it is not a requirement. Based on the project needs, a monolithic architecture can also adopt multiple repositories, which is referred to as polyrepo. The use of a monolithic repository is more favored because it aligns with the centralized nature of monolithic applications, making it easy to develop and maintain.

The drawback of monolithic architecture only starts appearing in later development and maintenance cycles. As the system evolves and expands, the codebase grows huge and becomes more challenging to change or maintain (Thones et al. 2015). This problem is usually related to tightly coupled dependencies where different parts of the application form a chain of references. Changes in one place can result in conflicts in other places because there are a lot of references to it. Over time, it becomes increasingly difficult to introduce new features or refactor existing ones without causing disruptions. Moreover, as the monolithic application encapsulates several services, new features or modifications to a service can lead to the redeployment of the entire application (Villamizar et al. 2015, 583–590).

2.1.2 Microservices

Microservices are architecture designs that organizes an application as several small, independent services executing in its process and communicating through protocols such as HTTP APIs (Lewis & Fowler 2014). In contrast to monolithic architectures, microservices decompose complex applications into smaller services focusing on specific business capabilities.

Figure 3 exemplifies a system that uses microservices. The architecture consists of a frontend service that interacts with multiple backend services, each responsible for distinct business logic. These backend services exchange information through APIs and store data in either SQL or NoSQL databases. This separation of concerns ensures independence and flexibility in selecting the most suitable technologies and databases for different service needs.

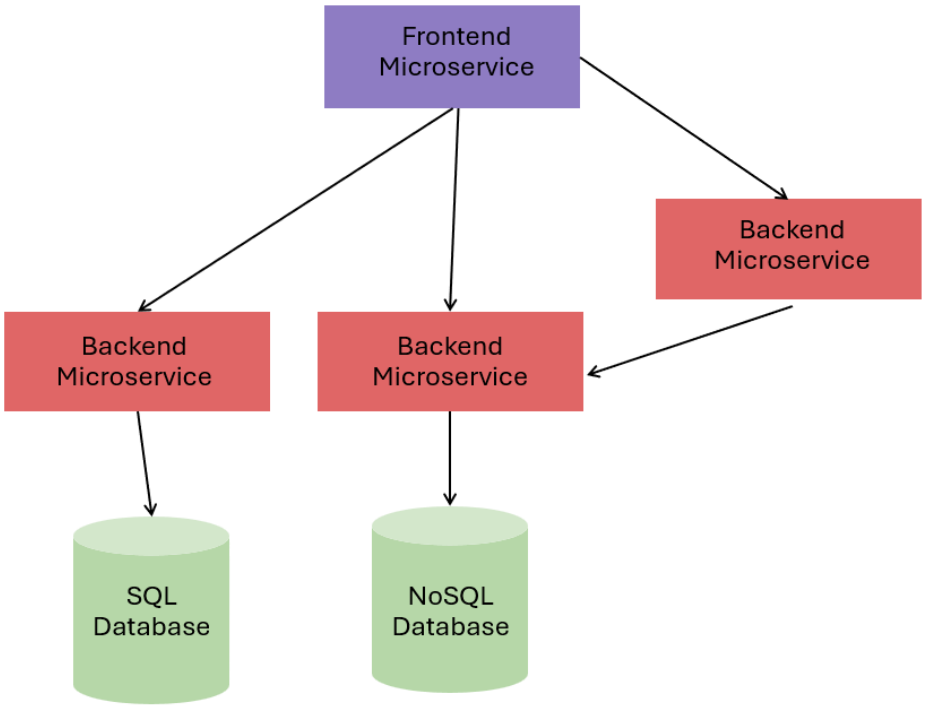

Figure 3. Microservices architecture pattern

One of the most persuasive reasons for adopting microservices is the ease of development. Due to the nature of independent services in microservices, modifications to one service are unlikely to impact others (Richards 2015, 34-35). This independence also enhances technology flexibility and heterogeneity. For instance, microservices allow backend services to use C# instead of making an application rely solely on JavaScript frameworks and libraries.

Additionally, microservices reduce the deployment risks. When implemented correctly, failures in one service exert minimal impact on the operation of others. This feature solves a major problem with monolithic systems. In monoliths, a modification necessitates the redeployment of the entire application, complicating development, testing, and deployment (Dragoni et al. 2017).

Microservices also offer greater scalability. Horizontal scaling in microservices only replicates the necessary services across multiple instances. While a monolithic system imposes the scaling of the entire system, microservices allow a particular service to scale independently (Newman 2021). Consequently, microservices can avoid disruption and resource deficiency when scaling less-demanding services.

Despite its advantages, the microservices approach is not a panacea. The independence in microservices presents complexity in effectively designing and managing the system. Defining service boundaries is a key challenge in microservices design, as the separation of business capabilities is often ambiguous (Soldani et al. 2018, 215–232). Therefore, a

poorly designed system can negatively affect the basic principles of low coupling and service cohesion, as defined by Yourdon and Constantine (1979). In regard to management, efficiently managing a microservices-based system mandates considerable expertise in distributed system development and DevOps (Baškarada et al. 20-18, 428–436).

Another issue with microservices is the communication mechanism. Inter-process communication in microservices introduces significant overhead compared to in-process communication within monolithic architectures (Newman 2021). This overhead happens because services within microservices architecture communicate over the network, leading to additional steps like data serialization, transmission, and deserialization.

2.2 Virtualization

It is crucial to understand virtualization as it is a core technology in cloud computing. Virtualization implies the abstraction of some physical components into a logical entity (Portnoy, 2012). In other words, virtualization abstracts and splits physical hardware components like processors, memory, and storage in a single computer into multiple virtual machines (IBM, 2023). Hypervisor plays an integral role in this capability.

As shown in Figure 4, a single operating system controls all hardware resources in a traditional architecture. However, there is a special software layer known as a hypervisor or virtual machine monitor (VMM) in a virtual architecture. The hypervisor makes it possible to execute several guest operating systems and software applications on a physical computer (Susnjara & Smalley 2024a). There are two types of hypervisors. Type 1 or bare-metal hypervisors run directly on the physical hardware, while Type 2 or hosted hypervisors run on top of a host operating system. Type 1 hypervisors generally offer better performance because there is no underlying operating system.

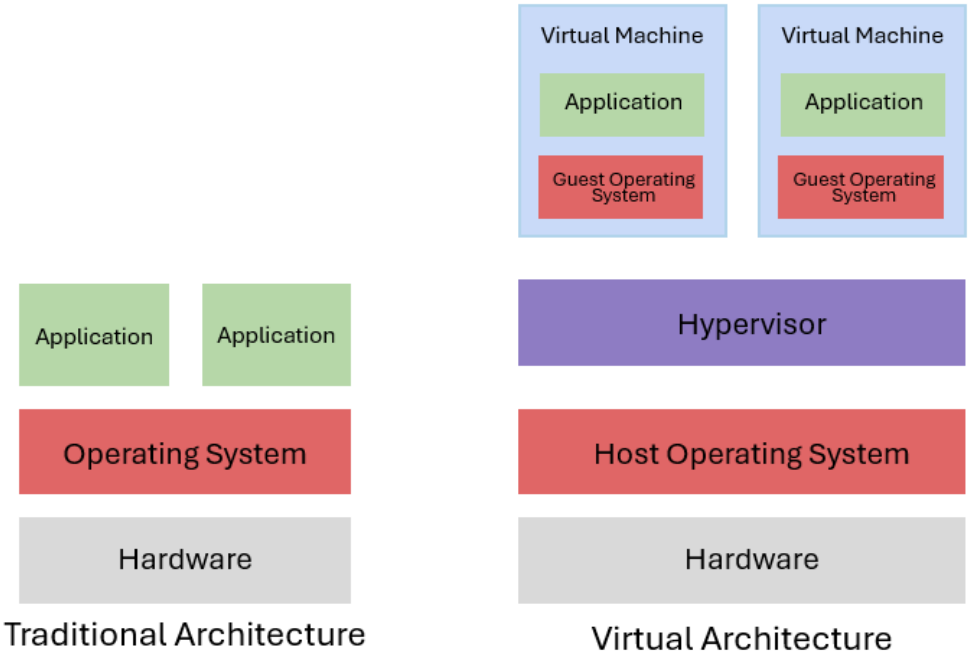

Figure 4. Traditional architecture and virtual architecture

Virtualization ensures workload isolation since all applications are contained in their own virtual machines. This isolation leads to better reliability because a software failure in one virtual machine does not affect other running virtual machines. Moreover, virtualization can lower the overall costs of data centers by combining the workloads of unutilized servers onto a single physical server. (Uhlig et al. 2005, 48-56.)

2.3 Cloud computing

According to The National Institute of Standards and Technology (NIST), cloud computing is a model that allows access to a shared collection of configurable computing resources, such as networks, servers, storage, applications, and services. Users can set up and release these resources quickly with little management or interaction with the service provider. (Mell & Grance 2011.)

The most significant advantage of cloud computing is its cost-effectiveness. The usage-based pricing model in cloud computing enables enterprises to reduce capital expenditure. This model allows businesses to pay only for consumption and scale their resources or services to fit the demand. Consequently, businesses can minimize the cost of operation and do not need to worry about unexpected spikes in demanded services. (Grossman 2009, 23-27.)

2.3.1 Cloud service models

There are three primary cloud computing service models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). Due to the emergence of serverless computing, Functions as a Service (FaaS) is available as another service model option. FaaS focuses on executing individual, event-driven functions without managing infrastructure. This serverless model will be discussed further in the Serverless computing section. Figure 5 illustrates how infrastructure management responsibilities are distributed between customers and cloud providers across these models. Each model offers varying levels of control, flexibility, and management.

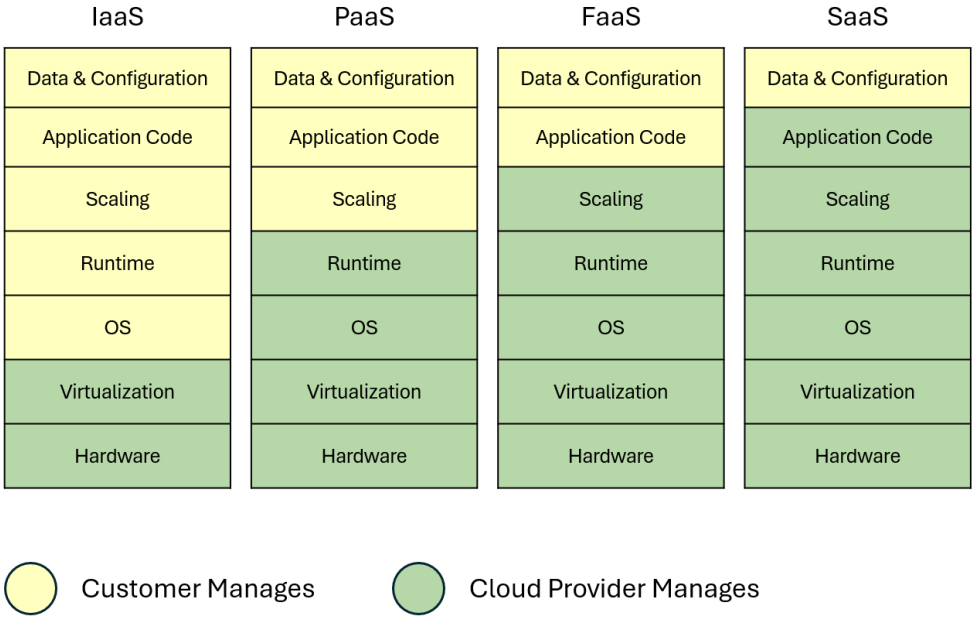

Figure 5. Traditional architecture and virtual architecture

Infrastructure as a service (IaaS) is a type of cloud computing model that supplies on-demand IT infrastructure services such as servers, virtual machines (VMs), computation, network, and storage to users on a pay-as-you-go over the internet (Susnjara & Smalley 2024b). This model gives users greater control over the infrastructure compared to the other previously introduced cloud models. With more flexibility and control, the need to maintain the infrastructure also increases. While cloud providers maintain the physical infrastructure, users are accountable for configuring, updating, deploying, and scaling the applications. Major providers of IaaS include Amazon Elastic Compute Cloud, Microsoft Azure Virtual Machines, and Google Compute Engine.

Platform as a service (PaaS) models provide a computing platform and solution stacks that enable users to deploy applications without the cost and complexity of managing the underlying hardware and software infrastructure (Chang et al. 2010, 55-56). This model allows

developers to focus on coding and deployment since it abstracts maintenance and administrative tasks. Google App Engine, Microsoft Azure App Services, and AWS Elastic Beanstalk are some of the most notable PaaS choices.

Software as a Service (SaaS) delivers software applications over the internet through a subscription-based approach. Having the highest level of abstraction, SaaS entitles users to access the application without managing the underlying infrastructure. The multi-tenant architecture in this model lets users share the same hosted environment while maintaining data security. SaaS offers a host of advantages, including cost efficiency, scalability, and accessibility from any location with the internet. Common SaaS applications consist of Gmail, Salesforce, and Microsoft 365. (Goode 2021.)

2.3.2 Serverless computing

A decade after Armbrust et al. (2009) defined and anticipated the evolution of cloud computing, Jonas et al. (2019) extended this perspective in A Berkeley View on Serverless Computing, predicting serverless computing as the dominant cloud computing paradigm. Their paper describes serverless computing as a cloud model where developers focus on writing code and delegate all server provisioning and administrative responsibilities to the cloud provider. According to their definition, a service is considered serverless if it automatically scales without explicit provisioning and is invoiced according to consumption. Serverless computing includes Function as a Service (FaaS) and Backend as a Service (BaaS):

- FaaS delivers the computing resources required to execute application logic in reaction to requests. As a core component of serverless computing, FaaS enables developers to write lightweight, independent, event-driven, and stateless functions. These functions are typically executed in response to predefined events, such as HTTP requests. FaaS functions are usually short-lived and can be written in any programming language supported by the platform. However, they usually suffer from cold start issues, causing a significant impact on latency. (Shahrad et al. 2019; Google.)

- BaaS implies external backend services that offer an API to replace specific functionalities within an application. These services can scale automatically and execute transparently, making them appear to be serverless to users. Authentication services, database administration solutions, and push notifications are examples of BaaS. (CNCF 2018; Google.)

Serverless computing brings a wide range of benefits. Firstly, it emphasizes functional-level programming. In serverless computing, developers shift their focus towards developing and

deploying functions to serverless platforms without complicated environment configurations. Cloud providers save those functions and their runtime, prepare a running environment called a sandbox, and set up an application-specific environment so that the functions can execute efficiently when triggered. Secondly, serverless computing enables automatic scaling in accordance with demand. For example, AWS can automatically scale up its infrastructure to handle a high volume of Lambda function requests by running as many concurrent execution environment instances as needed (Brisals & Hedger 2024). Thirdly, the consumption-based pricing plan in serverless computing transitions computing costs from capital expenditures to operational expenditures. This model eliminates the cost of idle services and the need to rent and pay for memory resources on standby continuously. (Li et al. 2022; Li et al. 2023, 1522–1539; Wen et al. 2023.)

Startup latency, especially cold start, is a serious challenge in serverless computing. It denotes the delay between a function's invocation and its execution, caused by the need to initialize an isolated environment or sandbox. Cold starts occur when no preinitialized environment is available, leading to longer delays, whereas warm starts reuse existing environments for faster execution. Figure 6 displays many steps contributing to the latency of function invocation. (Li et al. 2023, 1522–1539)

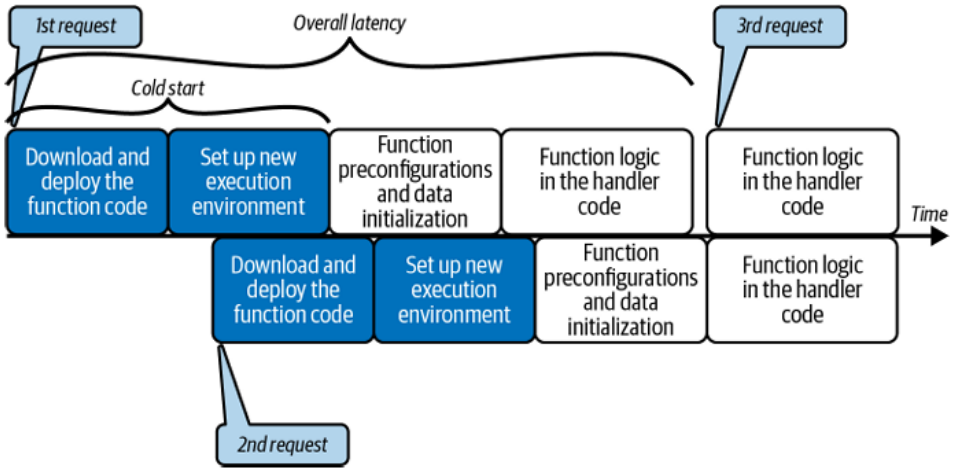

Figure 6. Function invocation latency for cold and warm starts— the first and second requests have cold starts, while the third request incurs a warm start (Brisals & Hedger 2024)

3 Implementation

3.1 Project introduction

The Precast Wall Designer application project was developed for Peikko Group Corporation. Peikko is a global supplier of innovative structural solutions for the precast and cast-in-place concrete industry. Founded in 1965, Peikko is a Finnish family-owned company operating in over 30 countries with approximately 2000 professionals. The company provides innovative, high-quality, and sustainable solutions that improve construction processes. The company also develops several tools for structural designers, including technical documents, connection design software, components for building information management, and modelling of structures.

The entire development project is being carried out by two student groups from LAB University of Applied Sciences and Salpaus Vocational School under the guidance of Peikko specialists and academic professors. The objective of the project is to replace the legacy desktop application called Peikko Ties with a web-based application called Precast Wall Designer. The new application enables structural designers to design and configure the connectors and layout of precast walls using Peikko construction products via the web interface. The shift to a web application includes designing a new system, updating formulas, and improving the user interface. This project will take around two to three years, and this thesis addresses the first phase. At this stage, the architectural decisions are critical because they provide the foundation for the future development, maintainability, and scalability of the application.

3.2 Design requirements and goals

Setting a balance between scientific goals and actual deliverability was the first requirement for the Precast Wall Designer project. This criterion was important for the long-term vision and educational context of the project because it ensured the system was approachable for students. Because the company put practicality first, they required the design of the project to be simple and extensible so that students could easily grasp the underlying concepts and contribute to further development. Secondly, the system design was required to be robust to facilitate future expansion and maintenance. Due to the long-term plan, early architectural decisions must establish clear responsibilities and organize the codebase and services effectively. Ultimately, the system must utilize Microsoft technologies, comprising C#, ASP.NET Core, Azure Cloud services, and Azure DevOps to guarantee interoperability with an internal ecosystem of Peikko. Upon completion, the application will be delegated to Peikko's internal development teams to integrate into their infrastructure for integration into their overall service infrastructure.

Based on these requirements, the following design goals were defined for the implementation. Firstly, the entire system architecture adopted microservices that originally consisted of frontend and backend services. This separation allowed LAB University and Salpaus Vocational School to develop and deploy independently. In this setup, the backend acts as a microservice within the overall microservices architecture. Instead of further decomposing the backend into smaller services, the Clean Architecture principles were implemented to establish clear responsibilities and maintain well-organized dependencies. Additionally, serverless Azure Functions were implemented to handle computational tasks. Integrating Azure Functions into the backend simplifies the backend logic and improves maintainability because the formulas are expected to change throughout the lifespan of the project. This integration needs other Azure services to operate reliably. Those services include Cosmos DB for storing computing results and Azure Service Bus for reliable messaging between the backend and the serverless functions. As the system adopted serverless functions, the microservices eventually encompassed frontend, backend, and a computing service based on Azure Functions.

3.3 System architecture overview

3.3.1 System design overview

The system architecture of the Precast Wall Designer application adheres to the REST architectural style, which was introduced by Roy Thomas Fielding (2000) in his PhD dissertation. REST (Representational State Transfer) is a widely adopted architectural standard for designing scalable, maintainable, and interoperable web applications. It is based on six key principles: client-server separation, statelessness, cacheability, uniform interface, layered system, and optional code-on-demand. The current implementation of the system adheres to most of these constraints. While cacheability is not yet implemented, it is planned for the next phase of the project, where the frontend will store frequently accessed construction product information to reduce backend requests. The code-on-demand constraint is excluded because it is irrelevant to the nature of the system.

The backend exposes its functionalities through RESTful API endpoints with the “/api” prefix to separate it from the frontend. The API design uses plural nouns instead of verbs to emphasize resources rather than actions. Moreover, standard HTTP methods are used to express the intended operations on the resources. These methods include GET for retrieving data, POST for submitting new computation requests, and PATCH for partially updating the status of a computation. Currently, the backend provides three API endpoints related to the computation of the precast wall loads:

- The `POST /api/computations/loads` endpoint allows the frontend to submit information about the layers in the precast wall for computing. The request body contains essential information regarding the internal, insulated, and external layers, formatted as a JSON object. After receiving the request, the backend validates the provided information, creates a computation record in the NoSQL database, and sends a computing request message to the Azure Service Bus. Finally, it returns the ID of the created computation record in the response.
- The `PATCH /api/computations/loads` endpoint is reserved for server-to-server communication between the backend and an external Azure Function responsible for executing the computations. After completing the calculation, the Azure Function calls this endpoint to update the computation value and status in the database. Since it partially updates an existing resource, the PATH method is used instead of PUT, which implies full replacement.
- The `GET /api/computations/loads/{id}` endpoint enables the frontend to periodically poll this endpoint using the previously returned ID of the computation. This endpoint returns the payload containing the result value and the status of the computation with the given ID.

Figure 7 summarizes the communication process between the frontend and backend. First of all, the frontend sends a POST request that includes information about the wall layers to the computation endpoint of the backend. After receiving the POST request with a JSON payload, the backend processes the information and prepares the computation before returning its ID to the frontend. However, the backend does not perform the actual computation. The details of how the backend incorporates with other Azure services to handle the computation will be discussed later in this section. Finally, the frontend periodically sends GET requests containing the computation ID to retrieve the result and status of the computation.

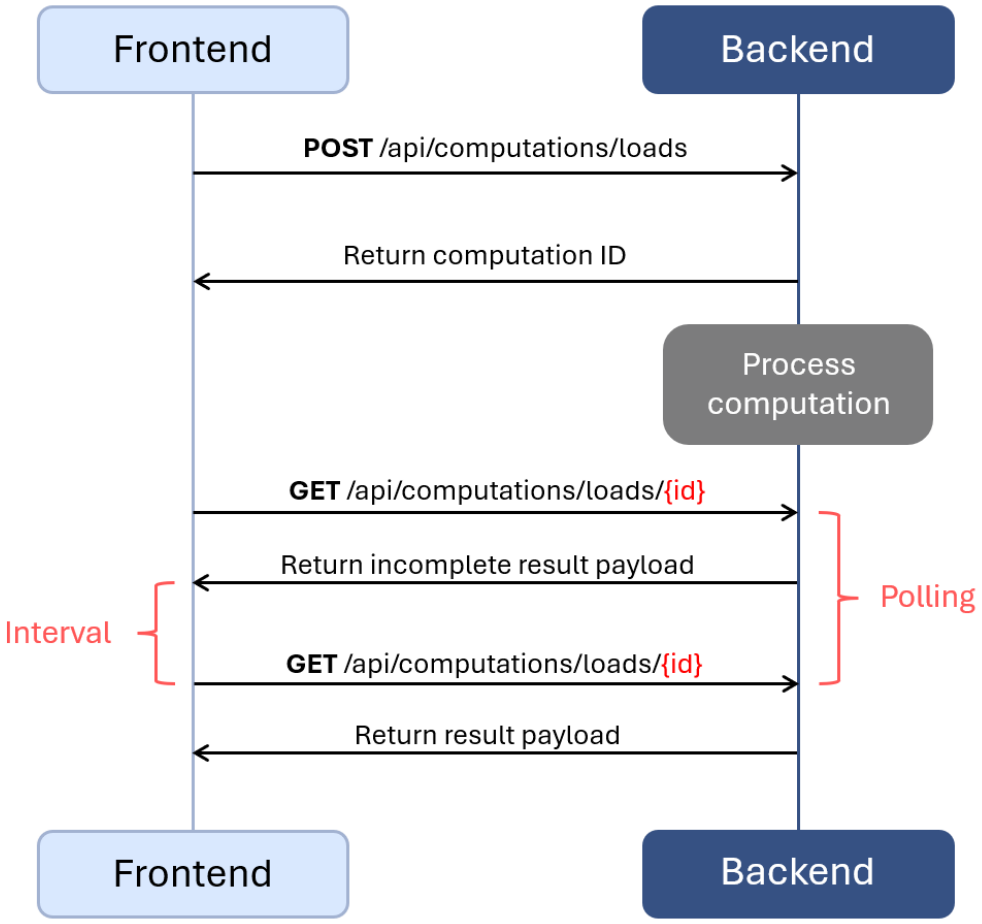

Figure 7. Overview of frontend and backend communication via RESTful API

After clarifying the communication between the frontend and backend, it is important to explain how the backend interacts with Azure services to perform computations. Figure 8 explains the entire workflow of handling a computation that begins when the frontend sends a POST request to the `/api/computations/loads/` endpoint. After receiving the POST request, the backend validates the JSON object containing the input data. Upon successful validation, the backend creates a new computation record in the Cosmos DB with a unique ID, empty value, and a status of `processing`. Consecutively, the backend sends a message to the Azure Service Bus queue, including the computation ID and the serialized JSON input data as a string. When a new message arrives in the queue, it triggers the Azure Functions to start up. After the initialization, the Azure Function instance retrieves the message, performs the load computation based on the wall layer specifications in the message, and prepares the result. Lastly, the Azure Function instance sends a PATCH request to the `/api/computations/loads` endpoint of the backend to update the computation record with the calculated value and mark its status as `finished`.

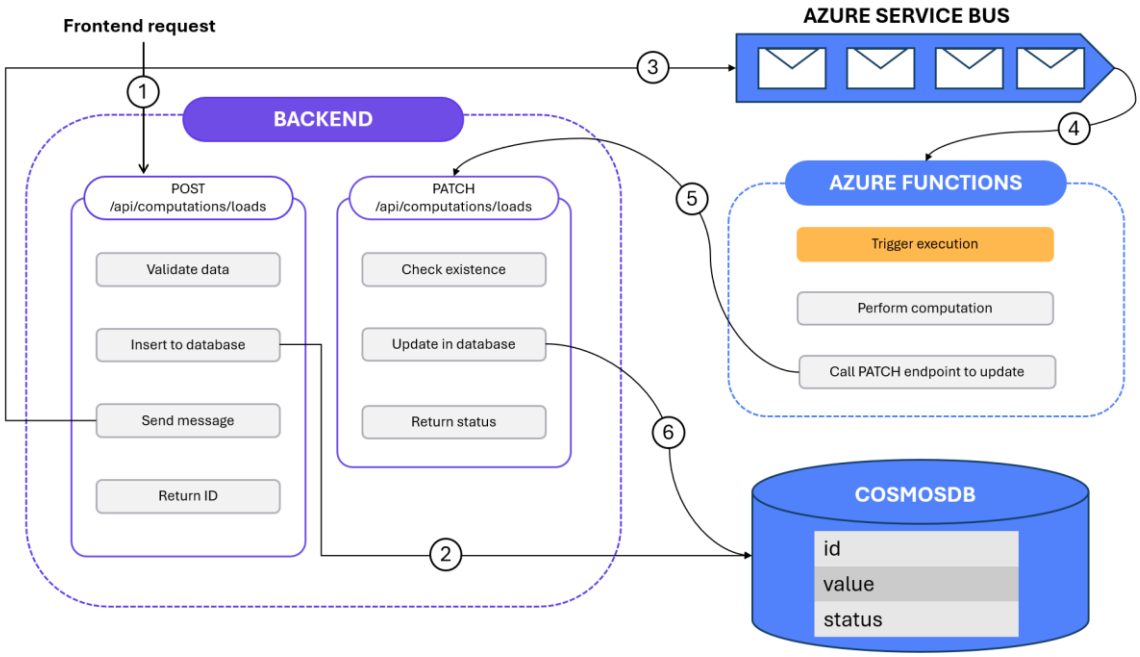

Figure 8. Interaction between the backend and Azure services to handle computations

Another important design consideration is the input validation before sending the message to the queue. If a POST request fails validation, the system immediately returns an error response without instantiating the Azure Function. Therefore, input validation in the first place can help avoid unnecessary execution costs from Azure Function. Consequently, the validation process must rigorously validate that the computation input data is valid and feasible for calculation. However, this approach introduces a duplication issue in the business logic between the backend and the Azure Functions. Some changes to the computation rules may require updates in both components. Additionally, the `POST /api/computations/loads` endpoint implements a rate limiter to protect the backend from potential overload during high traffic.

3.3.2 Architectural decision justification

To ensure reliable and scalable computation handling, the backend and the serverless functions follow the Competing Consumers pattern recommended by Microsoft, which utilizes Azure Service Bus and Azure Functions. Instead of invoking the Azure Function directly, the backend sends a computation request as a message to the Azure Service Bus queue. Azure Functions then listen for new messages, create new instances, and process them asynchronously. To the architecture of the entire system, this approach decouples the backend and creates a new serverless computing microservice. According to Microsoft, the Competing Consumers pattern is effective when the volume of work is variable, when tasks can be processed independently and in parallel, and when the system must ensure high availability and resilience to task failure.

Figure 9 illustrates the Competing Consumers approach. It is important to note that the Competing Consumers pattern is different from the Publish/Subscribe (Pub/Sub) pattern. In the Competing Consumers model, each message is processed by exactly one consumer, which is an instance of the Azure Function in this case. In contrast, the Pub/Sub model would deliver the same message to all subscribers. By applying the Competing Consumers pattern, each queue is reserved for a specific domain of computation, ensuring that each task is processed once by the appropriate service (Microsoft.)

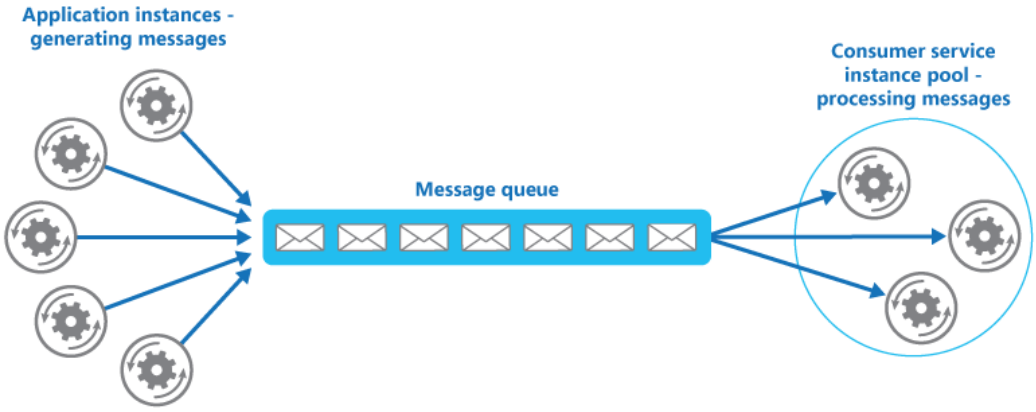

Figure 9. Competing Consumers pattern (Microsoft)

The Competing Consumers pattern offers several benefits. Firstly, the message queue acts as a buffer between the backend and the computing service, allowing the system to handle sudden spikes in workload without overwhelming the consumer services. This buffering minimizes the impact on the availability and responsiveness of both the backend and Azure Functions. Secondly, reliability is improved because messages are not lost if a consumer fails. Any working service instance can process a failed message because the message does not go to a specific instance. Instead, any healthy instance can process the message. Finally, this approach offers scalability. The automatic scaling configuration in Azure Functions allows dynamic scaling of instances based on the volume of messages. (Microsoft.)

After completing the computation, the Azure Function instance does not directly update a record inside the database. Instead, it sends a PATCH request to the `/api/computations/loads` endpoint exposed by the backend. This approach ensures that all interactions with the database are centralized through the backend service to preserve data integrity. By enforcing a single access point for database access, the backend can apply validation, logging, and security rules before committing changes to the database.

In the Peikko Precast Wall Designer application, computation results are generated on-demand using serverless functions. During the computation process, the frontend continuously polls for the results. When the frontend retrieves the completed result, that particular result is rarely accessed again. This data usage pattern requires a data storage solution

with flexible schema support to accommodate changing result formats, as well as low latency and high throughput for read operations during polling. Additionally, it requires automatic scaling to minimize provisioning and high availability to ensure uninterrupted service. Azure Cosmos DB was selected because it met these requirements better than the previously planned SQL Server. Another reason for selecting Cosmos DB is the ease of setup, which is ideal for the early development of the Peikko Precast Wall Designer web application. By reducing the infrastructure setup effort, developers can focus on developing the core business logic, which is the main objective at this phase.

Lastly, the backend and the Azure Functions use Azure Key Vault to securely store the application secrets, configurations, and connection strings to other Azure services. In development environments, these values are often stored in local files such as `.env` or `appsettings.json`. Although these files are convenient for local development and testing, they can lead to security risks if used in a production environment. Consequently, Azure Key Vault can help hold sensitive data to guarantee that connection strings and secrets are neither hardcoded nor revealed in the source code.

3.4 Backend architecture

To achieve a robust, maintainable, and testable backend, the Precast Wall Designer service adopts a tailored variant of Clean Architecture, originally proposed by Robert C. Martin. The Clean Architecture builds on earlier architectures, such as Hexagonal Architecture by Alistair Cockburn (2005) and Onion Architecture by Jeffrey Palermo (2008). Those architectures share the same objective of separation of concerns achieved by dividing the application into layers. Clean Architecture organizes these layers into concentric circles, as depicted in Figure 10. According to Robert C. Martin (2012), the Dependency Rule is the overriding rule in the Clean Architecture where all source code dependencies must point inward, and no code in an inner layer may depend on code in an outer layer.

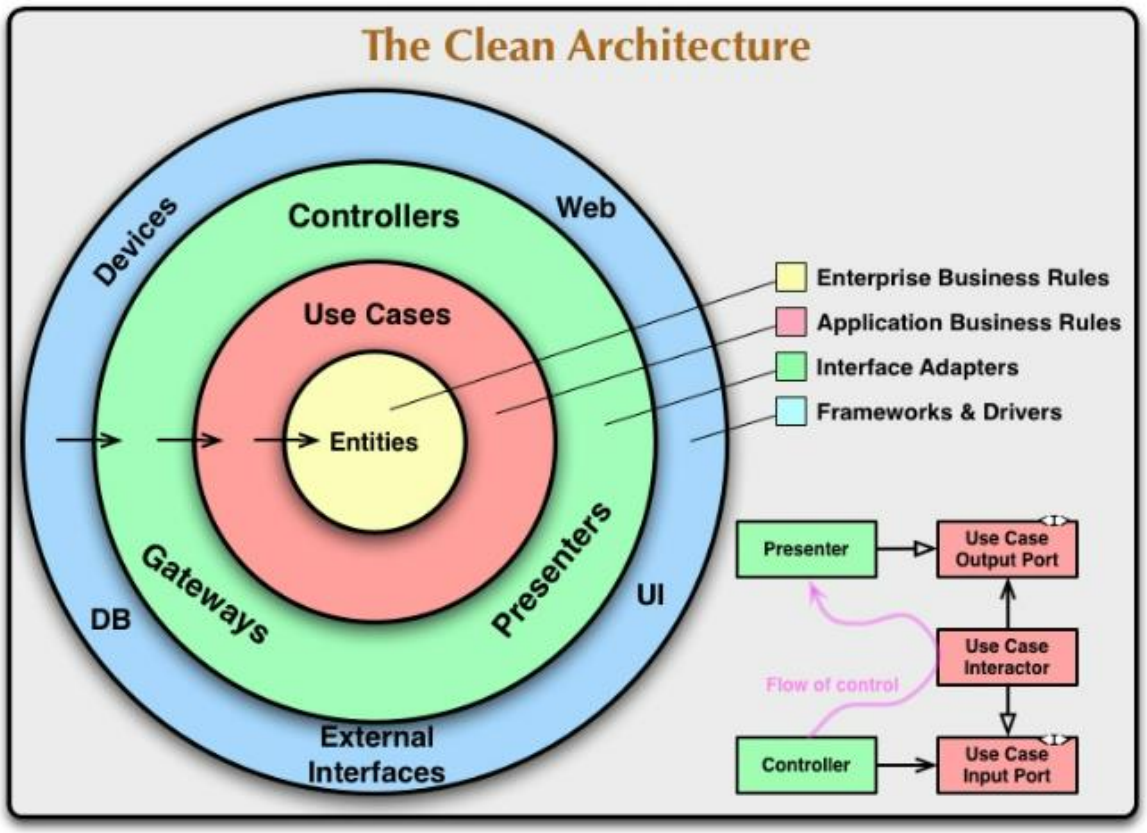

Figure 10. The original Clean Architecture (Robert C. Martin 2012)

As depicted in Figure 11, the Peikko Precast Wall application has 4 layers, including Domain, Application, Infrastructure, and API layers. The Application layer replaces the original Use Cases layer, which mediates the flow of data to and from the entities in the Domain layer. Moreover, the API layer is the entry point for incoming requests, tackling HTTP concerns like routing, controllers, rate limiters, middleware, and the entire application configuration. This customization preserves the core intent of Clean Architecture while tailoring the architecture to fit the ASP.NET Core development environment. To make the proper reference chain illustrated as arrows in Figure 11, the backend architecture uses Dependency Injection configuration with these layers via functions like `AddDomain`, `AddApplication`, and `AddInfrastructure`. For example, the API layer must call the `AddApplication` and `AddInfrastructure` functions to register the business orchestrating services from the Application layer, as well as the external and persistence services from the Infrastructure layer.

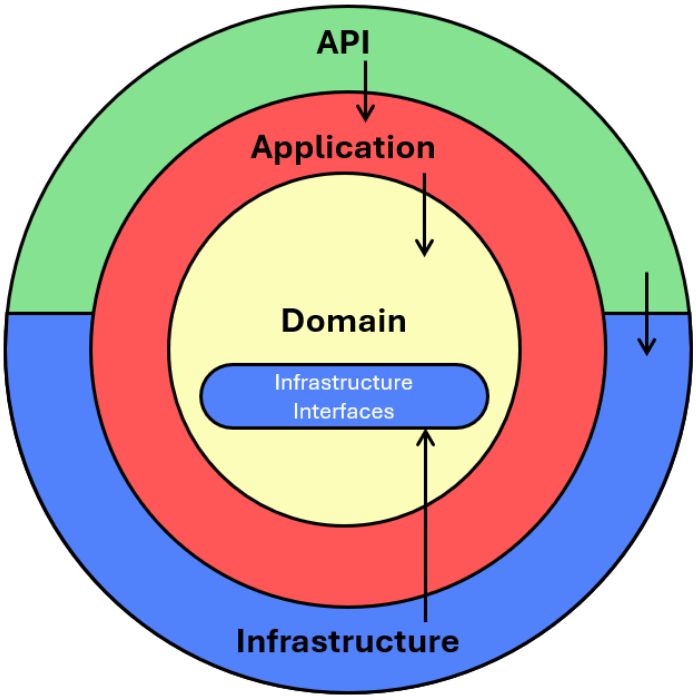

Figure 11. Backend architecture of Peikko Precast Wall Designer application using Clean Architecture. The arrows represent the references from one layer to another

The innermost Domain layer contains pure domain entities and core business logic, including classes, enumerations, interfaces, and domain functions. This layer should be designed to operate without any dependencies on external layers or frameworks. For that reason, changes in this layer must be driven by business rules rather than technical concerns. The Domain layer exposes a function for dependency injection called `AddDomain` for the Application layer. Additionally, it defines interfaces that represent Infrastructure services, such as repositories for data access or message brokers for message queues. These interfaces are implemented in the Infrastructure layer and injected into the Application layer. This practice ensures that higher layers can perform external operations without knowing about the implementation.

The Application layer contains business use cases, including services, commands, and queries. It defines the operations that can be performed and implemented using the domain entities from the Domain layer and external services from the Infrastructure layer. This layer solely depends on the Domain layer and uses Infrastructure interfaces defined in the Domain layer. Moreover, the Application layer is also responsible for mapping the payload received from the API layer to domain objects or entities. For dependency injection, it provides a function named `AddApplication` for the API layer.

The Infrastructure layer provides implementations for the interfaces defined in the Domain layer. This layer includes services such as databases, logging, message brokers, or third-

party services. To securely retrieve sensitive configuring information such as connection strings, the API layer must call the `AddKeyVault` function from the Infrastructure layer. This function establishes a connection to the remote Azure Key Vault using the provided credentials and allows the backend to read the secrets stored in the vault. For dependency injection, the Infrastructure layer provides a function named `AddInfrastructure` that is invoked by the API layer. It is crucial that the API layer calls the `AddKeyVault` function before the `AddInfrastructure` function because the latter depends on information retrieved from the Azure Key Vault to initialize its services, like databases and message brokers.

Finally, the outermost API layer serves as the main entry point for incoming HTTP requests. In terms of web development using C#, this layer or project is the only web project using the ASP.NET Core framework. It uses the controller-based approach to handle routing and organize the endpoints. This layer delegates tasks to the Application layer and returns the appropriate response as well as the status code to clients. Additionally, it includes the `appsettings.json` file for environment configurations and the `Program.cs` file, which acts as a startup file for the entire backend project. The API layer also implements a rate limiter to protect the system from excessive requesting attempts.

3.5 Integrating serverless functions

In the implementation, serverless functions are organized as a standalone .NET Core project and maintained in a separate repository. This separation is advantageous due to easy development and maintenance, particularly when updating formulas or adding new computation logic. This means new developers or maintainers can focus on the function level as each function performs a particular task with minimal complexity. Furthermore, changes can be made directly to this serverless function project without introducing complexity or risk to the backend.

Figure 12 shows a function that uses a queue trigger to listen to messages from an Azure Service Bus queue in the Peikko Precast Wall application. The `RunLayerLoadComputing` function computes the loads of the precast wall. The first parameter of the function is the queue trigger named `ServiceBusTrigger`, which includes the other 2 parameters. The first parameter is the `QueueName`, which refers to the specific Azure Service Bus queue the function listens to and involves if there is an incoming message. The second `AzureServiceBusConnection` parameter is a connection string for accessing the Service Bus. Both the `QueueName` and `AzureServiceBusConnection` are securely stored in Azure Key Vault and retrieved at runtime to adhere to the security practices used in the backend.

```
1
2  [Function("FunctionName")]
3  public async Task RunLayerLoadComputing(
4      [ServiceBusTrigger("QueueName",Connection = "AzureServiceBusConnection")]
5      ServiceBusReceivedMessage message)
```

Figure 12. Azure Function prototype using Service Bus queue trigger

One challenge with this separation is the duplication of business logic in the Azure Function project and the backend. For instance, there is a Layers class in the domain layer of the backend, which includes Layer and Hole classes to represent three layers and one hole in the middle of the precast wall. These classes exist in both the computation in the Azure Functions and the backend validation. As a result, any modifications made to these classes require developers to update both code bases to ensure synchronization.

3.6 Deployment and CI/CD

The deployment of both the backend and Azure Functions is automated using Continuous Integration and Continuous Deployment (CI/CD) pipelines configured in YAML files. These YAML configurations are used in Azure DevOps to set up pipelines that build, test, and deploy the code when changes are committed to the main branch. The system uses Azure App Service to host the backend and Azure Functions for serverless operations.

Figure 13 demonstrates how to configure Azure Key Vault secrets for deployment from the Azure Portal. Both Azure Web App and Azure Functions share a similar Azure Portal interface for setting up the connection strings for Azure Key Vault. To begin, developers need to navigate to the Environment variables tab from the left panel and then click on the Connection strings section. Click the Add button to open a side panel where developers can fill in the Name, Value, and Type fields.

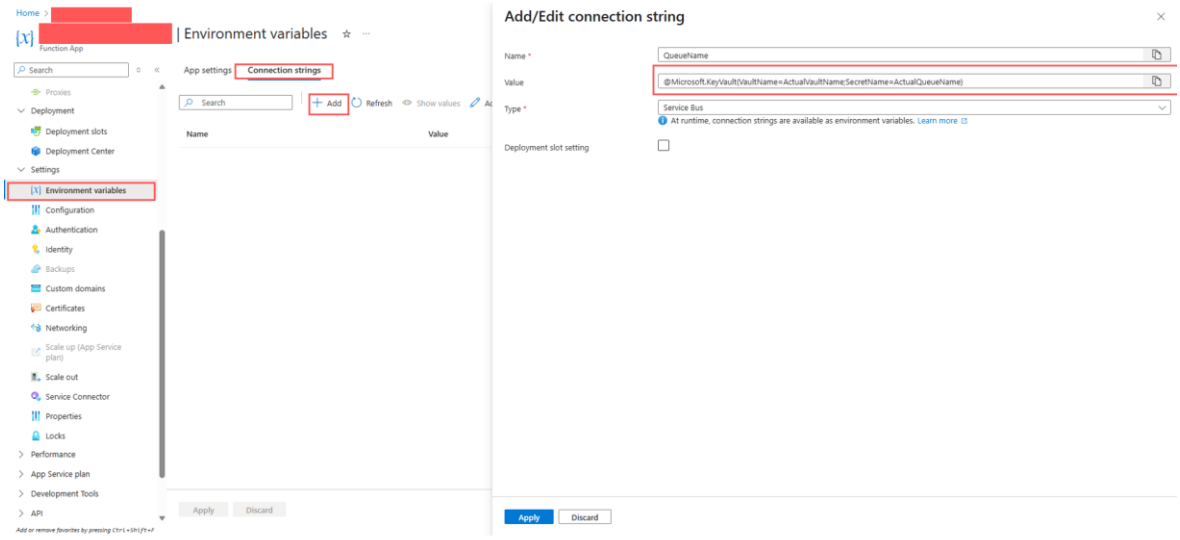

Figure 13. Configuring Azure Key Vault Connection Strings via the Azure Portal interface

In order for a configured secret from Azure Key Vault to work properly, the Value field within the connection string configuration must adhere to a syntax defined by Azure. The correct structure for retrieving the name of the queue is formatted as follows:

```
@Microsoft.KeyVault(VaultName=KeyVaultName;SecretName=QueueName)
```

In this syntax, VaultName refers to the actual name of the Azure Key Vault that contains the secret, while SecretName represents the name of the secret to be retrieved. Without this step, the deployed source code may face issues, as it will not be able to resolve the secrets from Azure Key Vault, resulting in a null value.

4 Evaluation

The backend system architecture and its integration with Azure Functions were reviewed and approved by specialists from Peikko company. The architectural decisions, which included the use of clean architecture, overall system design, and serverless computing, received positive feedback. The specialists confirmed that the implemented design met both technical and business requirements. Additionally, the company appreciated the frequent inquiries made during the implementation process to ensure that the system design and technical choices aligned with their requirements and were feasible for implementation within three months.

As part of the thesis work, an independent evaluation of the performance of the designed computing workflow was conducted using the k6 tool with Grafana Cloud for metric visualization. Figure 14 illustrates the configuration for load testing. These tools were independently evaluated during the project to assess their suitability to the system, capabilities, and potential value for the project in the future.

Figure 14 illustrates the configuration used during the performance evaluation. The test included four stages, simulating a gradual increase in user load: five virtual users for one minute, then ten, fifteen, and twenty users for one minute each. Additionally, the Stockholm region was defined as the nearest server to the deployed application services for testing. Several custom performance thresholds were defined to ensure the system met quality standards. These thresholds, specified in Figure 14, included the 95th percentile latency for POST and GET requests, making up to the overall response time of less than 3 seconds. In this setting, the 95th percentile refers to the latency value below which 95% of requests fall. Moreover, the test set a success rate of more than ninety-nine percent, representing the number of successful requests.

```
export let options = {
    cloud: {
        distribution: {
        loadTestEU: { loadZone: 'amazon:se:stockholm', percent: 100 },
        },
    },
    stages: [
        { duration: '1m', target: 5 },
        { duration: '1m', target: 10 },
        { duration: '1m', target: 15 },
        { duration: '1m', target: 20 },
    ],
    thresholds: {
        'post_duration': ['p(95)<1000'],
        'get_duration': ['p(95)<2000'],
        'response_time': ['p(95)<3000'],
        'success_rate': ['rate>0.99'],
    },
};
```

Figure 14. Load testing configuration

Figure 15 displays the results of the performance testing, where each predefined number of virtual users (VUs) repeatedly executed an iterative cycle of sending a POST request followed by polling with GET requests. A five-second delay between iterations caused fluctuations in the request rate line. At the beginning of the test, a noticeable spike in response time occurred due to the cold starts of Azure Functions. These cold starts happened because the Azure Functions are using the Flex Consumption plan with Always On mode turned off. Apart from this initial latency, the response time remained efficient and stable despite the gradually increasing user load.

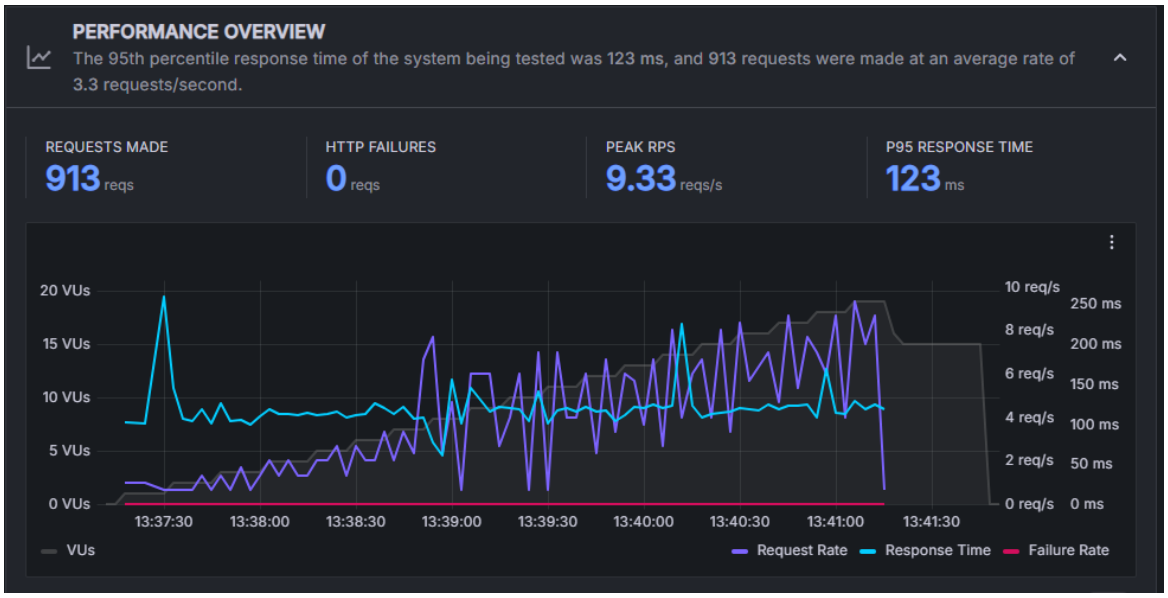

Figure 15. Results of load testing for the computational workflow, including a POST request and polling GET requests

Despite the decent implementation and validation, the project still has some limitations. The first challenge is the initial learning curve for student developers in adopting C# programming language, ASP.NET Core framework, and understanding the Clean Architecture pattern. While the learning curve of these technologies was manageable, it got significantly harder in the implementation. As the Peikko Precast Wall application emphasizes computational logic rather than database querying, some common design principles used in Clean Architecture, like Command Query Responsibility Segregation (CQRS) and adapters, were intentionally omitted to avoid overengineering practice. Furthermore, the separation between the backend and the serverless computing layer introduced some duplication of business logic. This duplication can increase maintenance complexity over time. These tradeoffs were made to meet project deadlines and deliver a working product to the company. However, they also highlight opportunities for future improvement and optimization.

5 Summary

This thesis examined the design and implementation of a cloud-based backend system for the Precast Wall Designer application from Peikko company. The primary goal was to build a scalable, efficient, and maintainable backend architecture using Microsoft technologies, focusing on the use of serverless Azure Functions to offload computational tasks.

The backend and serverless functions operate as a microservice within the microservices architecture of the application. They were developed using the ASP.NET Core framework, following the Clean Architecture principles. To ensure reliability and scalability, the system implemented the Competing Consumers pattern using the backend, the Azure Functions, and a message queue. In this pattern, the backend acts as a publisher of computational requests to Azure Service, while the Azure Functions serve as the consumers of these requests. While the backend acts as the entry point for computation requests, Azure Functions performs the actual computations. This system design enhances clear separation of concerns, maintainability, and stability under high user load. Moreover, it is easier for incoming developers or maintainers to the project to update or add formulas because the backend and the Azure Functions code are located in two different repositories.

In addition to the feedback received from the Peikko company, this thesis carried out an independent evaluation of the system's performance. Commonly used tools, including k6 for setting up the tests and Grafana Cloud for visualizing metrics, were explored to assess their suitability and potential for use in the future. These tools excelled in visualizing and analyzing how the system performed under increasing user load. The tests simulated multiple stages of gradually increasing virtual user traffic. Each user repeated a computation cycle involving a POST request to submit the input data and repeatedly called GET requests to retrieve results until the result was ready or the status was aborted. The results indicated that the backend handled increased load well with a high success rate and acceptable response times.

Despite these promising outcomes, there are several limitations and opportunities for future work to consider. The use of serverless Azure Functions is currently causing cold start latency, affecting the response time for the first few requests. Turning the Always On mode in the Azure Functions could be explored to avoid this issue. Furthermore, the backend needs more testing and enhanced security policies because this phase of the project focuses on development. Lastly, it is crucial to integrate the frontend with the existing backend system and to evaluate an effective deployment scheme after the integration.

References

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R. H., Konwinski, A., Lee, G., Patterson, D. A., Rabkin, A., & Zaharia, M. 2009. Above the clouds: A Berkeley view of cloud computing. University of California, Berkeley.

Baškarada, S., Nguyen, V., & Koronios, A. 2018. Architecting Microservices: Practical Opportunities and Challenges. Journal of Computer Information Systems, Vol. 60(5), 428–436. Retrieve on 16 March 2025. Available at DOI https://doi.org/10.1080/08874417.2018.1520056

Brisals, S. & Hedger, L. 2024. Serverless Development on AWS: Building Enterprise-Scale Serverless Solutions. O'Reilly Media, Inc. Retrieved on 30 March 2025.

Chang, W. Y., Abu-Amara, H., & Sanford, J. F. 2010. Transforming Enterprise Cloud Services. Springer Dordrecht, 55-56. Retrieve on 1 April 2025. Available at https://doi.org/10.1007/978-90-481-9846-7

CNCF Serverless Working Group. 2018. CNCF WG-Serverless Whitepaper v1.0. Retrieved on 6 April 2025. Available at https://github.com/cncf/wg-serverless/blob/master/whitepapers/serverless-overview/cncf_serverless_whitepaper_v1.0.pdf

Cockburn, A. 2005. The Hexagonal (Ports & Adapters) Architecture. Alistair Cockburn website. Retrieved on 28 March 2025. Available at https://alistair.cockburn.us/hexagonal-architecture

Dragoni, N., Giallorenzo, S., Lluch Lafuente, A., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. 2017. Microservices: Yesterday today and tomorrow. Mazzara, M., Meyer, B. (Eds.). Present and Ulterior Software Engineering. Springer, Cham. Retrieve on 10 March 2025. Available at DOI 10.1007/978-3-319-67425-4_12

Fielding, R. T. 2000. Architectural Styles and the Design of Network-based Software Architectures. University of California, Irvine. Retrieved on 24 March 2025. Available at https://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

Garlan, D. 2000. Software architecture: a roadmap. In Proceedings of the Conference on The Future of Software Engineering (ICSE '00). Association for Computing Machinery, New York, NY, USA, 91–101. Retrieve on 18 February 2025. Available at DOI 10.1145/336512.336537

Goode, J. 2021. Software as a Service (SaaS) Explained in 5 mins. IBM Technology. Youtube videos. Retrieve on 2 April 2025. Available at https://www.youtube.com/watch?v=20QUNgFIrK0

Google. What is serverless computing? Website. Retrieved on 6 April 2025. Available at: https://cloud.google.com/discover/what-is-serverless-computing?hl=en

Grossman, R. L. 2009. The case for cloud computing. IT Professional, Vol. 11(2), 23–27. Retrieve on 24 March 2025. Available at https://doi.org/10.1109/MITP.2009.40

IBM. 2023. What is virtualization? Website. Retrieve on 19 March 2025. Available at https://www.ibm.com/think/topics/virtualization

Ihde, S, & Parikh, K . 2015. From a Monotlith to Microservices + REST: the Evolution of LinkedIn's Service Architecture. InfoQ. Website. Retrieve on 18 February 2025. Available at https://www.infoq.com/presentations/linkedin-microservices-urn

Jaspan, C., Jorde, M., Knight, A., Sadowski, C., Smith, E. K., & Winter, C. 2018. Advantages and Disadvantages of a Monolithic Repository: A Case Study at Google. ACM/IEEE 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 225-234. Retrieve on 3 March 2025. Available at DOI 10.1145/3183519.3183550

Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.-C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N., Gonzalez, J. E., Popa, R. A., Stoica, I., & Patterson, D. A. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. University of California, Berkeley.

Lewis, J., & Fowler, M. 2014. Microservices: A definition of this new architectural term. Martin Fowler. Website. Retrieve on 8 March 2025. Available at https://martinfowler.com/articles/microservices.html

Li, Y., Lin, Y., Wang, Y., Ye, K. & Xu, C. 2023. Serverless Computing: State-of-the-Art, Challenges and Opportunities. IEEE Transactions on Services Computing, Vol. 16 (2), 1522–1539. Retrieved on 7 April 2025. Available at DOI 10.1109/TSC.2022.3166553

Li, Z., Guo, L., Cheng, J., Chen, Q., He, B. & Guo, M. 2022. The Serverless Computing Survey: A Technical Primer for Design Architecture. ACM Computing Surveys Vol. 54 (10). Retrieved on 30 March 2025. Available at DOI 10.1145/3508360

Martin, R. C. 2012. The Clean Architecture. The Clean Code Blog. Retrieved on 28 March 2025. Available at https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

Mauro, T. 2015. Adopting Microservices at Neflix: Lessons for Architectural Design. F5. Blog. Retrieve on 18 February 2025. Available at https://www.f5.com/company/blog/nginx/microservices-at-netflix-architectural-best-practices

Mell, P., & Grance, T. 2011. The NIST definition of cloud computing. National Institute of Standards and Technology. Retrieve on 19 March 2025. Available at https://doi.org/10.6028/NIST.SP.800-145

Microsoft. Competing consumers pattern. Microsoft Learn. Retrieved on 27 March 2025. Available at https://learn.microsoft.com/en-us/azure/architecture/patterns/competing-consumers

Newman, S. 2021. Building microservices: designing fine-grained systems. O'Reilly Media. Retrieve on 16 March 2025.

Paakkunainen, O. 2019. Serverless computing and FaaS platform as a web application backend. Aalto University. Thesis (Master's degree). Retrieve on 18 February 2025. Available at https://urn.fi/URN:NBN:fi:aalto-201906264219

Palermo, J. 2008. Onion Architecture. Jeffrey Palermo website. Retrieved on 28 April 2025. Available at https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/

Portnoy, M. 2012. Virtualization essentials. John Wiley & Sons. Retrieve on 19 March 2025

Potvin, R., & Levenberg, J. 2016. Why Google Stores Billions of Lines of Code in a Single Repository. Communications of the ACM, Vol. 59 (7), 78-87. Retrieve on 3 March 2025. Available at DOI 10.1145/2854146

Powell, P., & Smalley, I. 2019. What is monolithic architecture? IBM. Website. Retrieve on 28 February 2025. Available at https://www.ibm.com/think/topics/monolithic-architecture

Richards, M. 2015. Software Architecture Patterns. O'Reilly Media, 34-35. Retrieve on 10 March 2025.

Richardson, C. 2017. Pattern: Monolithic Architecture. Microservices.io. Retrieve on 28 February 2025. Available at http://microservices.io/patterns/monolithic.html

Shahrad, M., Balkind, J. & Wentzlaff, D., 2019. Architectural implications of function-as-a-service computing. Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52). Association for Computing Machinery, New York, NY, USA, 1063–1075. Available at: https://doi.org/10.1145/3352460.3358296

Sill, A. 2016. The Design and Architecture of Microservices. IEEE Cloud Computing, Vol. 3 (5), 76–80. Retrieve on 13 March 2025. Available at DOI 10.1109/MCC.2016.111

Soldani, J., Tamburri, D. A., & Van Den Heuvel, W.-J. 2018. The pains and gains of microservices: A Systematic grey literature review. Journal of Systems and Software, Vol.146, 215–232. Retrieve on 15 March 2025. Available at https://doi.org/10.1016/j.jss.2018.09.082

Susnjara, S., & Smalley, I. 2024a. What are hypervisors? IBM. Website. Retrieve on 20 March 2025. Available at https://www.ibm.com/think/topics/hypervisors

Susnjara, S., & Smalley, I. 2024b. What is infrastructure as a service (IaaS)? IBM. Website. Retrieve on 31 March 2025. Available at https://www.ibm.com/think/topics/iaas

Thönes, J. 2015. Microservices. IEEE Software, Vol. 32 (1), pp. 116-116. Retrieve on 3 March 2025. Available at DOI 10.1109/MS.2015.11

Uhlig, R., Neiger, G., Rodgers, D., Santoni, A. L., Martins, F. C. M., Anderson, A. V., Bennett, S. M., Kagi, A., Leung, F. H., & Smith, L. 2005. Intel virtualization technology. Computer, 38(5), 48–56. Retrieve on 19 March 2025. Available at https://doi.org/10.1109/MC.2005.163

Vergadia, P. PaaS vs. IaaS vs. SaaS vs. CaaS: How are they different? Google Cloud. Website. Retrieve on 27 March 2025. Available at https://cloud.google.com/learn/paas-vs-iaas-vs-saas?hl=en

Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., & Casallas, R. 2015. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. 2015 10th Computing Colombian Conference (10CCC), Bogota, Colombia, 583–590. Retrieve on 3 March 2025. Available at DOI https://doi.org/10.1109/ColumbianCC.2015.7333476

Wen, J., Chen, Z., Jin, X. & Liu, X. 2023. Rise of the Planet of Serverless Computing: A Systematic Review. ACM Transactions on Software Engineering and Methodology, Vol. 32 (5). Retrieved on 30 March 2025. Available at DOI 10.1145/3579643

Yourdon, E., & Constantine, L. L. 1979. Structured design: Fundamentals of a discipline of computer program and systems design. Englewood Cliffs, N.J. : Prentice Hall. Retrieve on 16 March 2025.