Ministry of Education and Science of Ukraine
NATIONAL UNIVERSITY OF «KYIV-MOHYLA ACADEMY»
The Department of Informatics, Faculty of Informatics

PRINCIPLES OF SYSTEM ORGANIZATION AND PRACTICAL USE CASES OF NANOSERVICE ARCHITECTURE

Textual Part of the Master's Thesis
in the specialty "Software Engineering" 121

Supervisor of the Master's Thesis
Doctor of Technical Sciences, Associate Professor
Glybovets A.M.

_____
(signature)

“____” _____ 2024

Executed by student
Barsuk O.I.
“____” _____ 2024

Kyiv – 2024

Ministry of Education and Science of Ukraine
NATIONAL UNIVERSITY OF «KYIV-MOHYLA ACADEMY»
The Department of Informatics, Faculty of Informatics

APPROVED
Head of the Department of Informatics
Doctor of Physical and Mathematical Sciences, Associate Professor
Horokhovskyi S.S.

_____
(signature)
"_____" _____2023

INDIVIDUAL ASSIGNMENT
for the Master's Thesis

to the student of 2nd-year of Master's program in the Software Engineering Barsuk Oleksandr Ihorovych
to develop Principles of system organization and practical use cases of nanoservice architecture

Contents of the Textual Part of the Master's Thesis:

Table Of Contents
Abstract
Introduction
Chapter 1: Theoretical foundations for defining nanoservice architecture
Chapter 2: Concepts of system organization based on nanoservice architecture
Chapter 3: Implementation of a use case
Conclusions
Bibliography
Appendices

Date of issue "_____" _____ 2023

Supervisor
A.M. Glybovets, Doctor of Technical Sciences,
Associate Professor _____
(*signature*)

Assignment received
O.I. Barsuk _____
(*signature*)

Topic: Principles of system organization and practical use cases of nanoservice architecture

Calendar Plan of Work:

№	Title of the Stage of the Diploma Project (Thesis)	Deadline for the Stage	Note
1.	Receiving the assignment for the diploma project	20.10.2023	
2.	Review of technical literature on the topic of the thesis	10.11.2023	
3.	Preparing the Chapter 1 of the thesis	31.12.2023	
4.	Preparing the Chapter 2 of the thesis	31.01.2024	
5.	Preparing the Chapter 3 of the thesis	01.03.2024	
6.	Preparing the introduction, conclusion and other parts of the thesis	10.03.2024	
7.	Conducting a comparative analysis of modern solutions	15.03.2024	
8.	Development of the final architecture of the system	20.03.2024	
9.	Implementing the backend part of the system	10.04.2024	
10.	Implementing the frontend part of the system	22.04.2024	
11.	Setup deployment of the system, having the system up and running	30.04.2024	
12.	Creating slides for the presentation and writing the report	06.05.2024	
13.	Analysis of the obtained results with the supervisor, writing the report, and preliminary defense of the Master's thesis	17.05.2024	
14.	Revising the work based on the results of the preliminary defense	20.05.2024	
15.	Final preparation of the explanatory note and slides	31.05.2024	
16.	Defense of the Master's thesis (project)	11.06.2024	

Student Barsuk O.I.
Supervisor Glybovets A.M. “____”_____ 2023

TABLE OF CONTENTS

ABSTRACT

This paper explores the theoretical foundations and practical implementations of nanoservice architecture. It defines software architecture, differentiates nanoservice architecture from other architectural styles, and examines its correlation with the serverless approach. The study addresses the benefits, challenges, and communication patterns of nanoservice architecture, including issues like the double-spending problem and interservice communication. Practical use cases are analyzed to highlight scenarios where nanoservice architecture excels. The paper concludes with an implementation of an air monitoring system using nanoservices, demonstrating the architecture's real-world application.

Key words: nanoservice architecture, serverless computing, software architecture, cloud computing, microservices, service-oriented architecture, event-driven communication, AWS Lambda, autoscaling, distributed systems.

INTRODUCTION

Relevance. The ever-evolving landscape of software development necessitates new architectural paradigms that can effectively address the increasing complexity and scalability demands of modern applications. Nanoservice architecture, a refined and granular evolution of microservice architecture, emerges as a pivotal solution in this context. It aligns with the serverless approach, offering enhanced agility, scalability, and cost efficiency. Understanding the nuances of nanoservice architecture is crucial for software engineers and architects striving to develop robust, scalable, and maintainable systems in today's cloud-centric environment.

Purpose of the Research. The primary purpose of this research is to provide a comprehensive exploration of nanoservice architecture, contrasting it with other architectural styles such as monolithic, service-oriented, and microservices architectures. This study aims to find out the unique benefits and challenges of nanoservices, particularly in the context of their synergy with serverless computing. By examining the theoretical underpinnings and practical applications of nanoservice architecture, the research seeks to offer valuable insights and guidelines for effectively leveraging this architecture in real-world scenarios.

Objectives of the Research. The research aims to achieve the following objectives:

- define and contextualize the concept of software architecture, with a focus on nanoservice architecture;
- differentiate nanoservice architecture from other architectural styles, highlighting its unique characteristics;
- explore the correlation between nanoservice architecture and the serverless approach;
- investigate the benefits and challenges associated with nanoservice architecture;
- identify and analyze communication patterns and models relevant to nanoservice architecture;
- demonstrate practical use cases where nanoservice architecture is most beneficial;

- implement a real-world nanoservice-based system to showcase the practical application of theoretical concepts.

Object of the Research. The object of this research is the architectural style of nanoservices, particularly how it integrates with serverless computing to form highly scalable and efficient software systems.

Subject of the Research. The subject of this research encompasses the design principles, communication patterns, deployment strategies, and practical use cases of nanoservice architecture in modern software development.

Sources of the Research. This research draws upon a diverse range of sources, including:

- academic literature on software architecture and microservices;
- technical whitepapers and standards from IEEE and ISO;
- documentation and case studies from leading cloud service providers, particularly AWS;
- books and articles by renowned software architects and engineers;
- practical implementation guides and tutorials from the software development community.

Scientific Novelty of the Obtained Results. The scientific novelty of this research lies in its detailed examination of nanoservice architecture as a distinct paradigm, separate from microservices, and its deep integration with serverless computing. This study not only consolidates existing knowledge but also introduces new perspectives on the benefits, challenges, and best practices of deploying nanoservices. It offers a novel framework for understanding the interplay between nanoservices and serverless platforms, highlighting innovative solutions to common architectural challenges.

Practical Significance of the Obtained Results. The practical significance of this research is substantial, as it provides actionable insights and guidelines for software engineers and architects. By implementing a real-world use case – the Air Monitoring System – this study demonstrates the practical application of nanoservice architecture. It showcases how nanoservices can be effectively deployed, scaled, and

managed using serverless platforms, providing a blueprint for similar projects. The findings of this research can significantly enhance the development, deployment, and operational efficiency of modern software systems, driving better performance and cost efficiency in dynamic and scalable environments.

CHAPTER 1: Theoretical foundations for defining nanoservice architecture

1.1 The concept of software architecture and approaches to its definition

The word “architecture” in software development is quite overloaded, applied to business, solutions, systems, software itself, and many other areas. Let’s see what software architecture definitions are used and what differences they have.

According to the standard IEEE “Recommended Practice for Architectural Description for Software-Intensive Systems” ***architecture*** is “the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution” [1]. And in its turn, a ***system*** is a collection of components organized to accomplish a specific function or set of functions.

There are also plenty of other definitions of architecture in the literature, such as:

- “Software architecture is a set of architectural (or design) elements that have a particular form” [2];
- “Software architecture is a collection of computational components – or simply components – together with a description of the interactions between these components – the connectors” [3];
- “Software architecture is an abstract system specification consisting primarily of functional components described in terms of their behaviors and interfaces and component-component interconnections. The interconnections provide means by which components interact. Architectures are usually associated with a rationale that documents and justifies constraints on component and interconnections or explains assumptions about the technologies which will be available for implementing applications consistent with the architecture” [4];
- “Software architecture is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them” [5, p. 21].

As mentioned Amit Goel in his work, “a common pattern in these definitions is that software architecture is a description of the system (or software) at a very high level of abstraction which defines the core of the system. The high level abstraction describes the components and their static and dynamic view. Static view describes the interconnection of components to form the structure, and dynamic view describes the interaction of components to form the behavior” [6, p. 31].

Based on these thoughts and documents, there was issued a standard named ISO/IEC 42010 “Systems and software engineering – Architecture description”, which defined ***architecture*** as fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution [7, p. 2].

Simon Brown states that there are many types of architecture in software development, but software architecture comprises of both application architecture and system architecture, where application architecture operates on a single application level (defined with modules and classes, usually with single technology stack), and system architecture describes the whole system of interacting applications (e.g., mobile application + Java web application + MySQL database) [8, pp. 6-7]. He says, that “it’s anything and everything related to the significant elements of a software system; from the structure and foundations of the code through to the successful deployment of that code into a live environment” [8, p. 7].

We agree with Brown’s approach to decomposing software architecture into two components, but his definition of software architecture is too broad and unclear. For example, “anything and everything related to the significant elements of a software system” – it’s arguable which elements are significant to the system, moreover significance of such elements can be changed during system evolution. Also, it’s arguable how deployment can relate to architecture since completely different architectures can be deployed successfully or not, and it can depend not on architecture, but on infrastructure setup.

There is another approach to software architecture, stated in “Software Architecture in Practice” by L. Bass, P. Clements, and R. Kazman. They define the

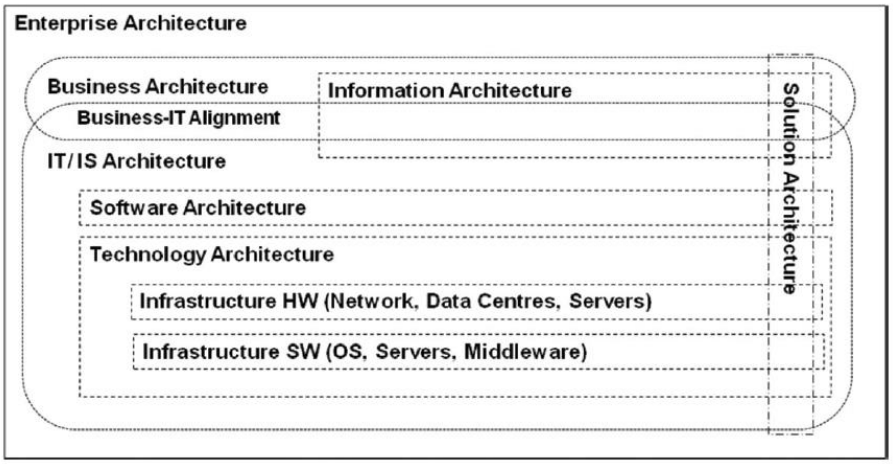
software architecture of a system as "*the set of structures needed to reason about the system, which comprises software elements, relations among them, and properties of both*" [9, p. 4]. We find this definition the most accurate and precise among all aforementioned ones. First of all, this definition is focused on system organization, and organization can be described by its elements (components), relations between them, and their properties. This is about which layers/modules/services/components the software system consists of, and what hierarchy and dependencies on each other they have.

Secondly, this definition does not take environmental factors into account. We agree that software architecture can depend on the environment, sometimes it can even be chosen by the environment, but it does not include the environment, because architecture reflects (or should reflect) the structure and organization inside the system, and the environment is something that is outside of the system.

For all these reasons we will rely on this definition in this paper and use it in our further analysis of other architectures.

At the same time, it's worth mentioning that in the context of information technologies the term "architecture" can be applied to many other different aspects of software engineering besides software itself, like technology architecture, information systems architecture, information architecture, business architecture, solution architecture, enterprise architecture. In Figure 1 we can observe a correlation between these terms [6, p. 32].

Figure 1. "Architecture" terminology in information systems



Here we can see that software architecture can be considered a part of information systems architecture together with the technology architecture.

1.2 The concept of nanoservice architecture

Now that we have a notion of software architecture let's see what a nanoservice architecture is, whether it has its distinctive features and can be defined or not.

E. Wolff defines nanoservices as follows [10, p. 346]:

– Nanoservices compromise in regard to some microservice properties such as isolation and technology freedom. However, nanoservices still have to be independently deployable.

– The compromises enable a larger number of services and therefore for smaller services. Nanoservices can contain just a few lines of code.

– To achieve this, nanoservices use highly efficient runtime environments. These exploit the restrictions of nanoservices in order to enable more and smaller services.

X. Wang, Yu-Ju Huang, T. Yuan, and R. van Renesse define nanoservices as "a type of microservice that is highly specialized for a single type of hardware resource, and that applications be built from such nanoservices, communicating over fast interconnects." [11]. The authors propose the idea of nanoservice, which predominantly uses a single type of resource – for example, a microservice like Ngnix, which requires a set of resources – CPU, memory, and more, can be decomposed into a set of nanoservices, where each one responsive for its own resource.

N. Kratzke describes nanoservices as fine-grained functions, which can be easily deployed and automatically scaled, and provide the potential to reduce infrastructure and operation costs [12]. This description of nanoservices does not specify how nanoservices are differentiated from other approaches or architectures, for instance, microservices can also be easily deployed and automatically scaled with Kubernetes.

Some refer to the term nanoservice as an ultra-lightweight microservice specifically developed for IoT scenarios [13]. The key idea behind using nanoservices is that all capable nodes in the proximity can participate in the service provisioning:

the high-capacity nodes can provide more resources and low-capacity nodes can do with less [13]. In their other work, the authors call nanoservices the microservices which are small enough and could be managed locally [14, p.2]. So, here we can see, that authors don't separate nanoservices into a separate architecture, but rather treat nanoservices as a special type of microservices.

Also, there is an approach to defining a nanoservice as a small software piece created by using a template invoking an application, I/O calls, and a reduced configuration file, which is suitable for edge environments [15].

This approach is quite similar to the previous one, stating that nanoservice is a type of microservice, but just smaller in size. If the service size were the only difference between nanoservice and microservice, there would be no need to differentiate these concepts at all, let alone discuss a distinct nanoservice architecture.

Y. Nasser describes nanoservices as smaller, more specialized, self-contained units of functionality that work together to provide a larger service, which is typically much smaller in size and scope than a microservice [16].

C. Ebi defines a nanoservice as "a piece of code, usually a method or function, that can be deployed to production without the need for scaffolding or boilerplate code… I call it a nanoservice because multiple functions make up the engineer's overall business logic, and this business logic encapsulates a microservice." [17].

J. Kanjilal says, that a nanoservice is "a small, independently deployable, testable, reusable component that essentially splits a microservice into several small pieces. Unlike a microservice, a nanoservice doesn't typically represent an entire business function" [18].

So, as we can see, the term "nanoservice" has a variety of meanings, ranging from an antipattern in microservice architecture to a component of a comprehensive nanoservice architecture. Therefore, from the definitions mentioned, we can identify several common characteristics of nanoservices:

- they are small in size and scope, smaller than a microservice;
- one nanoservice represents a single unit of functionality;

- a group of nanoservices can collectively function as a microservice in terms of size and scope;
- they are easily deployable;
- they are independently deployable.

Let's discuss these characteristics in detail. First of all, how small do nanoservices need to be? As its name implies – nanoservice should be quite small, usually smaller than a microservice. Unlike a microservice, a nanoservice doesn't typically represent an entire business function [18]. Though there are no precise limits defined in the numbers of code lines, it's usually considered that nanoservice is a single method or function (for example, one of the CRUD[1] operations with the resource or one RPC method). And since it comprises a single function, nanoservice usually has one entry point – API endpoint or other triggers (cron events, message queue events, etc).

At the same time, as nanoservice represents a single unit of functionality (a function), to cover the whole business logic related to a particular resource, there is a need in a bunch (group) of nanoservices. For example, if a resource in our system is a movie, then we need to implement some basic CRUD operations – to cover these scenarios:

- get a movie by ID;
- get a movie collection by a filter;
- update a movie by ID;
- delete a movie by ID.

So, each of such operations can be represented by a separate nanoservice, and together they function like a microservice with its bounded context, owned resource, and ubiquitous language. E. Wolff states that "a Bounded Context that might consist of one or a few microservices will now comprise a multitude of nanoservices that each implement a very narrowly defined functionality" [10, p. 347].

This brings us to the conclusion that nanoservices can potentially be less independent than microservices because they can share a common data model,

[1] CRUD – a group of data manipulation operations (Create, Read, Update, Delete)

persistence layer, and even state (execution context). And the more nanoservices share, the less independent (isolated) they are.

When it comes to deployment, nanoservices are considered to be more easily deployed than other architectures. Well, this happens mainly due to these two reasons: a) nanoservice is usually used with FaaS[2] cloud services (no need to set up a platform) and b) instead of deploying one big monolith or service only a small function (nanoservice) can be deployed. M. Roberts states that in a FaaS environment, we upload the code for our function to the FaaS provider, and the provider does everything else necessary for provisioning resources, instantiating VMs, managing processes, etc [19]. Additionally, the Netflix development team claims that frequent developer commits coupled with CI/CD runs resulted in nearly 10x greater deployment velocity in pre-production environments [20].

Having explored the characteristics of nanoservices, let's address the question: What is a nanoservice? A **nanoservice** can be defined as *a type of service that executes a singular function, representing a single unit of functionality*. This function might pertain to business tasks, such as supplying a resource based on its identifier, or to technical aspects, like deleting temporary data or transforming data into a different format. Other attributes of nanoservices, such as their small size and the simplicity of their independent deployment, are more so consequences of this fundamental definition than defining features themselves.

Accordingly, we can define **nanoservice architecture** as *a type of architecture, where a software system is constructed using nanoservices*. As we mentioned in the previous chapter, software architecture of a system is the set of structures needed to reason about the system, which comprises software elements, relations among them, and properties of both. In nanoservice architecture, this implies that the system is composed of nanoservices, which are the fundamental structures of the system. While various relationships can exist among these nanoservices, their interactions are primarily manifested through communication. This aspect of communication between nanoservices will be elaborated upon in the forthcoming chapters.

[2] FaaS – Function as a Service – cloud service providing platform for running functions.

So, as we can see, nanoservice architecture is a separate, standalone type of architecture, which has its distinctive features and differs from other architecture types. In the next chapter we are going to explore, what differences nanoservices have compared to the other architectures.

1.3 Differences between nanoservice architecture and other architectures

a. Monolithic architecture

A monolithic architecture is a traditional software program model built as a self-contained unified unit and independent from other applications [21]. In addition, the application logic and data access are all contained within a single process, reducing the latency and network overhead associated with microservices architecture [22].

Nicola Dragoni and others define monolith as “a software application whose modules cannot be executed independently” [23].

Y. Nasser admits that “monolithic architectures are common in traditional application development. In this model, all components of the application are built, deployed, and maintained as a single unit” [16].

Monolith is an old architectural style, and it is often criticized. Nicola Dragoni and others name such issues of monoliths:

- large-size monoliths are difficult to maintain and evolve due to their complexity;
- monoliths also suffer from the “dependency hell”, in which adding or updating libraries results in inconsistent systems that do not compile/run or, worse, misbehave;
- any change in one module of a monolith requires rebooting the whole application (which entails considerable downtimes);
- deployment of monolithic applications is usually sub-optimal due to conflicting requirements on the constituent models’ resources: some can be memory-intensive, others computational-intensive, and others require ad-hoc components (the developer must compromise with a one-size-fits-all configuration);

- monoliths limit scalability – creating new instances of the same application is inefficient due to the size of the system, where increased traffic stresses only a subset of the modules;
- monoliths also represent a technology lock-in for developers, who are bound to use the same language and frameworks of the original application [23].

Nevertheless, we can assert that monolithic architectures still offer certain advantages. These include resource efficiency, as they operate within a single process and typically involve lower network traffic. Additionally, they facilitate easier management of transactions and the evolution of interfaces/data models. This is because there are no concerns of backward or forward compatibility issues; all system parts can be updated and deployed simultaneously.

On the other hand, in nanoservice architecture, each operation (function) runs in a separate process. This significantly increases resource utilization: the whole system needs many processes to run each operation in a separate process, and communications between nanoservices produce higher network traffic – whereas in monolith communication happens between functions by simply passing arguments/return value, in nanoservice architecture becomes interprocess or even interservice communication. Besides this, nanoservice architecture has complexities of all distributed systems: management of transactions in nanoservice systems becomes much more complex and usually involves additional patterns (saga, two-phase-commit, etc); also, evolution of interfaces or shared data models should be planned considering backward and forward compatibility, as it can affect several nanoservices.

At the same time, nanoservices also offer many notable advantages. First of all, none of the issues listed earlier with monoliths is relevant to nanoservices. Nanoservices are easy to maintain as their code base is limited to a small context – one function/operation. Also, there is no “dependency hell” with nanoservices, as each nanoservice can have its own list of dependencies, without affecting any other service. Same with system updates – changes of the code in nanoservices usually do not lead to any downtime, because firstly nanoservices are deployed independently, and secondly deployment of a nanoservice to FaaS does not require any restart – the cloud

platform just enables a new version of nanoservice and that's all. Nanoservices are also easily scalable and provide the possibility of using the most suitable and efficient technology for each nanoservice (either programming language or data storage). Other nanoservice advantages include the ability for separate teams to develop a set of nanoservices independently.

To sum up, nanoservices are more dynamic and flexible, and easier to develop, maintain, and scale compared with monoliths, but they also introduce additional resource consumption, such as memory for processes and network traffic.

b. Service-oriented architecture (SOA)

Service-oriented architecture (SOA) can be defined as an architectural style which can guide business process definition and support "rapid, low-cost composition of distributed applications" [24, p. 1]. The SOA style was introduced to address architectural complexity, redundant programming and inconsistent interfaces [25; 26]. In SOAs a service is a self-describing unit that "consists of a contract, one or more interfaces and an implementation" [27, p. 57]. SOAs in turn comprise an application frontend, services, a service repository and enterprise service bus.

SOA paradigm presents an approach in which modular, accessible, self-describing, implementation-independent, interoperable, and reusable components are published as services which can be remotely invoked and consumed by other applications or combined with other services [28; 29]. M. Mehta and S. Lee conclude that at the heart of SOA is a collection of services that communicate with each other, and these services are used as building blocks of applications [30, p. 3].

According to M. Mehta and S. Lee, there are three following roles in SOA, which can be performed by each component in a SOA [30, p. 3]:

- service provider – publish the availability of services;
- service broker – register and categorize published services and provide search capabilities;

- service requester – use service broker to find a service and use it to build an application.

As we can observe, transitioning from a monolithic to a service-oriented architecture involves dividing a monolithic application into a collection of services. This process includes defining interfaces or contracts, establishing the roles of each service, and outlining the communication patterns between services.

Though SOA involves breaking down a system into different services, they are not independent. For instance, Chris Richardson claims that "SOA applications typically have a global data model and share databases" [31, p.14]. That means that even though there are cases when SOA services can be developed and deployed independently, the changes that affect the data model or database schema require the update and simultaneous deployment of each dependent service.

Another feature of SOA services that can be considered as a disadvantage is the way services communicate in SOA architecture. Chris Richardson states that "SOA applications typically use heavyweight technologies such as SOAP and other WS* standards. They often use an ESB, a smart pipe that contains business and message-processing logic to integrate the services" [31, p.14]. First of all, in our opinion such "smart pipes" are more difficult to test – they contain logic, which need to be tested, and it can not be covered with unit testing. So, higher-order testing is required – integration, end-to-end, etc. Also, the mentioned protocols are excessive in describing data model, compared to others like JSON, Avro, Protobuf, therefore they produce more network traffic. This might result in slower response times compared to monolithic architectures where components are tightly integrated.

Besides the mentioned drawbacks, authors name complexity as an additional one. Haresh Luthria and others state that "while some benefits were realized in terms of integration, the authors conclude that SOA is difficult to implement in practice because business processes are not included in the service definitions, which tend to be technical service implementations of the business process flow… SOA offers some advantages but the technology implementers have a long way to go before their efforts can result in the benefits outlined by current reports" [32, p.48].

So, we can conclude that SOA architecture inherits most of the monolith disadvantages, tackling only some of them (modularity, reusability of services, better scalability per each service), but introducing instead additional complexity, performance overhead, and data consistency challenges.

That being said, nanoservice architecture provides similar advantages to those it offers over monolithic architecture when compared to SOA, including ease of maintenance, ease of scaling, independent development and deployment, and separation of dependencies and data models.

c. Microservice architecture

Now let's dive into microservices and explore what they are and whether they differ from nanoservices or not.

Microservices [23] is an architectural style originating from Service-Oriented Architectures (SOAs) [33]. The main idea is to structure systems by composing small independent building blocks communicating exclusively via message passing. These components are called microservices. The characteristic differentiating the new style from monolithic architectures and classic Service-Oriented is the emphasis on scalability, independence, and semantic cohesiveness of each unit constituting the system [34].

Nicola Dragoni and others define microservice as "a cohesive, independent process interacting via messages", and a microservice architecture – "a distributed application where all its modules are microservices" [23, p. 2].

Lewis and Fowler define the **microservice architectural style** as *an approach for developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API* [35].

Microservices put a strong emphasis on loose coupling and high cohesion of services, which are expected to result in better scalability, adaptability, and quality of software architectures [36].

It’s worth saying that microservice architecture eliminates or minimizes all of the mentioned disadvantages of monoliths, though introducing some new ones instead. Nicola Dragoni and others suggest such “microservices solutions” to monolith issues:

- difficulty in maintenance: microservices implement a limited amount of functionalities, separated and independent from each other;
- dependency hell: possible gradual transitions to new versions of a microservice (when older versions of microservice coexist with the newer ones);
- whole rebooting on any change: with microservices a change causes re-deployment downtime only in the affected service (and can be eliminated with a “blue-green deployment” technique);
- one-size-fits-all configuration of deployment: as microservices usually use containerization, a configuration of the deployment environment can be selected per each service;
- limited scalability: scaling of microservices is much easier and more effective because it does not need duplication of all modules and components, scaling can be done on a microservice level;
- technology lock-in: microservices offer a much wider range of technologies (for example, each microservice can be written in a different language and using frameworks different from other services), the only constraint is using the same standards for communication (protocols, data formats) [23, p. 3].

As we already mentioned, microservice architecture does not only resolve or minimize the disadvantages of monoliths but also introduces some new ones. The main disadvantages of microservices compared to monoliths can be:

- increase in system complexity;
- increase in resource consumption.

The increase in system complexity can be viewed in several aspects – one is due to the distributive nature of the microservice system, and the other is a necessity to orchestrate microservices. The distributiveness means that the system switches from one executing application where all business operations happen to several (or multiple) applications each responsible for a separate scope of business operations. This

transition means that the CAP theorem applies to the system, meaning that it is impossible for a web service to provide the following three guarantees together: a) consistency, b) availability, and c) partition tolerance [37, p.1].

Another aspect of system complexity, that microservices bring, is the necessity to orchestrate microservices. A monolith in general on the system level can have only two states – running or stopped, whereas a microservice system can have many more states – for example, for a three-service system states can be:

- all services are running;
- the first service is running, and two others are stopped;
- the first service is stopped, and two others are running;
- …
- all services are stopped.

So, even such a small system comprising only three microservices can have many different states, so microservices need to be monitored and managed – checking whether all microservices are running, starting a stopped microservice, checking whether a required number of each service instances are running and run them if needed, preserve order and dependencies when starting or rebooting services. These all are generally covered by orchestration tools and techniques, which need to be considered in microservice architecture.

Let’s also examine a specific characteristic of microservice architecture: its increased resource consumption compared to monolithic systems. This is primarily due to heightened memory usage, as each microservice typically runs in its own process rather than a separate thread, and elevated network traffic, which results from the communication between the microservices. The frequent inter-service communication over the network can introduce latency (by service synchronization or processing asynchronous events), potentially affecting the overall efficiency and responsiveness of the system.

To the aforementioned drawbacks, C. Richardson also adds additional ones [31, p. 17]:

- the challenge of finding the right set of services – incorrect decomposing of a system leads to a distributed monolith – a system consisting of coupled services that must be deployed together;
- difficulty in deciding when to adopt the microservice architecture – at the start of an application development it's often uncommon to have the problems that this architecture solves.

Having delved into the advantages and disadvantages of microservices, let's now examine how nanoservice architecture differs from them in these aspects.

As for microservices advantages – all of them apply to nanoservice architecture. In its essence nanoservices are also a distributed system, having the same advantages and disadvantages as microservices do, plus some additional, specific to nanoservices only.

Similar to microservices, nanoservices are also easy to maintain due to each nanoservice encompassing a limited piece of functionality. For the same reasons, smooth deployment and gradual upgrading become even easier and more seamless in nanoservice architecture. Deployment of new nanoservices usually causes no downtime at all, because cloud providers can switch nanoservice versions in a moment. Also, the environment and resources for each nanoservice can be meticulously selected and configured, thereby optimizing the efficiency of the resources used. And technology stack is only limited to the platforms and programming languages, supported by FaaS services.

But let's look into nanoservices specifics, which differ them from microservices. First of all, their main advantage is the small size of each service which makes them perfect for running in FaaS platforms. Basically, the ability to run a system on a FaaS platform effectively is what differentiates nanoservices from other architectures.

With that being said, we can say, that nanoservices tend to have pros and cons of a serverless approach since they are often used together. Vamsi Krishna Thatikonda names such advantages, as cost efficiency, scalability, improved development speed ("without the need to manage infrastructure, developers can focus solely on writing

and refining code, drastically reducing development cycles"), reduced operational overheads [38, pp. 2-3].

Limitations and disadvantages of the serverless approach include latency due to the cold starts, vendor lock-in, limited customization and control, and security concerns [38, p. 4].

Besides the disadvantages inherent to serverless computing, nanoservices also face challenges specific to their architectural design. Among these disadvantages, we can include:

- increased complexity of communication.
- higher resource consumption.

Let's see why the complexity of communication increases in nanoservice architecture the most. Microservices are typically designed such that a single local transaction creates or updates one aggregate of the domain model within each microservice, thereby minimizing the necessity for distributed transactions as much as possible [31, p. 157]. As for nanoservices, this is harder to accomplish as nanoservices are more granular and one aggregate can be typically created/updated by several different nanoservices, as each service embraces not a whole set of aggregate operations but rather one granular operation. As a result, distributed transactions are more commonly found in nanoservice architectures than in microservices.

Regarding higher resource consumption, it can be observed that an increase in service granularity inevitably leads to an increase in resource consumption. This was true when moving from monolith to service-oriented architecture and further to microservices, and this is the case when moving from microservices to nanoservices either. Again – higher granularity means that each operation (wrapped in a service) runs in a separate process, whereas in microservice usually multiple operations can be run as tasks within one process. Also, more granularity means more communication between services – more network calls and events.

As a result, the nanoservice architecture evolved as an architecture, that suits the most to serverless approach, providing the ability to utilize all benefits of the serverless approach in cloud computing. In comparing nanoservices with microservices, we find

that both architectures aim to break down complex systems into more manageable parts. Microservices focus on decomposing an application into services based on business functionality, offering scalability and easier maintenance but at the cost of increased complexity and potential for higher resource consumption. Nanoservices, being even more granular, further emphasize scalability and maintenance ease, potentially leading to greater efficiency in deployment and upgrading. However, this granularity may result in even higher resource consumption and complexities, especially in communication and distributed transactions.

1.4 The correlation of nanoservice architecture with the serverless approach

As we observe, nanoservices are often closely associated with serverless computing; at times, these terms are even used interchangeably. Let's explore what serverless computing entails, whether it qualifies as an architecture in its own right, and how it relates to nanoservices.

M. Roberts defines serverless architectures as “application designs that incorporate third-party “Backend as a Service” (BaaS) services, and/or that include custom code run in managed, ephemeral containers on a “Functions as a Service” (FaaS) platform” [19]. He says that such architectures remove much of the need for a traditional always-on server component. It’s worth mentioning that not only Roberts but also other authors treat serverless as architecture, for example:

- D. Maden says that “Serverless architecture (another term used in place of “serverless” is “Functions as a Service”) is a design approach that enables you to build and run applications and services without the need to manage the infrastructure” [39];
- T. Dolynyuk refers to serverless also as an architecture [40];
- AWS documentation states that serverless architecture is a way to build and run applications and services without having to manage infrastructure [41];

- Google Cloud documentation defines serverless architecture as a software design approach where developers can build and manage applications without managing the underlying architecture [42].

And this is only part of the full list. So, as we can see, there are many sources (including cloud providers) that treat serverless as an architecture. But is it an architecture?

As we explored in paragraph 1.1, software architecture consists of structures, relations among them, and their properties. And none of the mentioned definitions of serverless complies with these requirements. They rather refer to the way of building and running applications, but not to application structure.

Serverless does not define, what structures (or modules) a system consists of, nor does it describe relations between them. In fact, we can put any type of architecture on top of a serverless platform. It can be a monolith, or services, or microservices, or anything else. What serverless does define – is the way of deploying and running applications. It defines that applications can be run and managed without managing the underlying infrastructure, which means that all infrastructure up to runtime is provided, including physical machine, operating system up and running, network configured, runtime functioning (.NET, Java, NodeJS, etc). T. Dolynyuk notes that the “less” part in “serverless” actually refers to a reduced amount of effort required by developers to publish, maintain, and scale application servers [40].

For all those reasons we are convinced that *serverless can not be architecture as it has no architecture features, but an approach to either managing project resources or a way to deliver product (deployment and running).*

As we already mentioned, any architecture can be deployed as serverless. But which architecture suits the most to serverless – it’s definitely nanoservices. This is due to the existing limitations of FaaS services provided by cloud platforms – there are limitations in a) operation time (for instance, up to 15 minutes in AWS Lambda), b) volume of memory used, and c) volume of incoming (as parameters) and outcoming (as return data) information. And taking into consideration all these restrictions, it’s

hard to imagine a small monolithic system running by call only not more than several minutes, though it is still possible.

So, in theory, any existing architecture can potentially be deployed as serverless, but in practice, only nanoservices comply the most with the current restrictions of FaaS. Of course, cloud providers evolve every day, and these limitations have already become much more appeasable, and proceed to improve, providing more and more available resources to the customers. It is possible that in the future limitations of FaaS services won't differ from a hardware host, allowing to run applications of any size for an unlimited time. And then it will be even more clear that serverless relates to resource management and type of deployment, but not architecture.

Thus, serverless computing is not an architectural style but rather a method for managing resources and deploying products, as it lacks specific architectural features. Although any architecture can theoretically be deployed serverlessly, nanoservices are the most suited due to current cloud platforms limitations like operation time and memory constraints. As cloud services evolve, these limitations are expected to relax, making serverless computing more adaptable for various architectures.

CHAPTER 2: Concepts of system organization based on nanoservice architecture

2.1 Benefits of nanoservice architecture in modern development

In the previous chapter, we have elaborated on the terms “nanoservice” and “nanoservice architecture”, compared nanoservice architecture with other popular architectures (monolith, service-oriented, and microservices), and reviewed differences between these architecture types. Now, let’s see what advantages of nanoservice architecture make software engineers choose this architecture among others when developing modern software systems.

So, why would anyone want to go with nanoservices in the first place?

Cloud computing and FaaS

One of the major benefits nanoservices offer is the ability to utilize serverless computing and in particular deploy the whole or most of the system to a Function-as-a-service platform. This gives the ability to utilize cloud infrastructure effectively, without the need to maintain own physical infrastructure. P. Castro and others say that “cloud providers can allocate servers to run code as needed and can stop servers after functions finish as they run for limited amount of time” [43, p. 47]. So, other architectures like monoliths or microservices can be also deployed to cloud infrastructure, but only nanoservices allow to fully use the benefits of serverless computing, making the whole system serverless with all the benefits serverless offers.

One of the key elements of serverless is Functions as a service (FaaS) – the ability to run arbitrary code without having to provision infrastructure to run it as a service or to pay for the infrastructure [44, p. 9]. As we mentioned in the previous chapter, nanoservices are the best candidates for such an arbitrary code, which can be run in FaaS with regards to all current limitations of FaaS platforms.

Cost efficiency

P. Sbarski and others say that “serverless systems are much more granular with regard to scaling and are cost-effective, especially when peak loads are uneven or unexpected. Because of their utility pay-per-use billing, serverless services can be extremely cost-effective” [44, p. 15]. The crucial here is to understand, that nanoservice architecture in combination with a serverless approach allows one to handle gracefully **unexpected peak loads** – expected peak loads may be not an issue, as one can prepare for them (for example, when the growth of load is expected due to some date – holiday, or onboarding new customers – with other architectures additional capacity can be provisioned, but this strategy can not help for unexpected loads – provision of extra capacity just for the cases which can never occur is cost ineffective, as one pays for resources never utilized). So, nanoservice architecture with a serverless approach gives a possibility to pay only for the resources, that are in use, and release ones that are idling.

Easy scaling

Easy scaling in nanoservice architecture refers to the ability to adjust the capacity and performance of individual services independently, based on demand, without affecting the entire system. This granular scalability is a key advantage of nanoservice architectures, where each nanoservice is designed to perform a very specific, singular function. This design principle allows for more precise resource allocation and scalability, as each nanoservice can be scaled up or down independently based on its specific load or demand, rather than scaling the entire application as one would in a monolithic architecture.

The more granular our architecture is, the more effective we can scale it. While the monolith can be scaled only all at once, in granular architectures (which nanoservices are) only those parts of the system can be scaled, which endure the most load.

Another feature significantly enhancing the scalability of nanoservices is autoscaling, offered by cloud platforms. V. K. Thatikonda admits that with its dynamic

scalability and cost-efficiency, serverless computing marks a transformative shift in cloud technology [38, p. 345]. **Autoscaling** is a mechanism of dynamically acquiring or releasing resources to meet Quality of Service requirements [45]. And if in the case of virtual machines (used for monolith or microservices), autoscaling usually requires a sophisticated algorithm (like Horizontal Pod Autoscaling in Kubernetes), in FaaS autoscaling is a part of the service, provided out of the box [46]. Ethan Cerami says "the real promise of serverless or "functions as a service (FaaS)" platforms is that they automatically scale based on demand" [47]. In his article, he experiments with loading Azure Functions and different settings for the "Scale Out" parameter. As a result, the author managed to decrease processing time from 109.29 seconds to 21.06 seconds for the same load (10 concurrent requests for a total of 1K requests).

Other cloud providers also offer function autoscaling, for example, AWS Lambda provides by default a total concurrency limit of 1,000 concurrent executions across all functions in an AWS Region [48], and GCP Functions allow maximum instances limits from 100 to 3000 depending on a function type (generation) [49]. So, we can conclude that though in monolithic and microservice architectures autoscaling can be set up using additional solutions (like Kubernetes, AWS Auto Scaling, etc), **in nanoservice architecture autoscaling can be utilized as an integral part of FaaS service**, usually even without additional configuring.

This difference in autoscaling of virtual machines (VM) vs functions is important because it leads to unobvious conclusions – autoscaling of VM requires more time to start new nodes compared to functions. P. Sbarski and others describe such an issue in the development of their social network: "We often saw 100x spikes in traffic as thousands of users flooded in all at once to see their favorite influencer's new content. These traffic spikes were usually shortlived, which was problematic for the EC2-based system because EC2 autoscaling couldn't react fast enough. It typically takes EC2 instances a few minutes to spin up. By the time they are ready to serve user requests, it's too late. The traffic spikes have come and gone, and many users would have left after having experienced a laggy response time" [44, p. 59]. As opposed, they say "AWS Lambda autoscales the number of concurrent executions based on load. This

happens instantly and handles those unpredictable spikes we experience effortlessly" [44, p. 60].

Independent development

Another significant advantage of adopting a nanoservice architecture is the ability to develop, test, and deploy nanoservices independently by different teams. And this leads to several subsequent benefits, as follows.

Independent development allows multiple teams to work in parallel on different nanoservices. This parallelism can significantly speed up the overall development process, as teams don't have to wait for others to complete their parts of the application.

Another benefit of nanoservices independence is that teams can specialize in certain domains or technologies, allowing them to leverage their expertise more effectively. For example, one team might specialize in data processing services, while another focuses on user authentication.

Independent development also reduces the risk of one team's changes adversely affecting the entire system. Changes to a single nanoservice can be rolled out and, if necessary, rolled back without impacting other services.

But, we need to remember, that nanoservices are not independent completely, and sometimes can be more dependent on each other than microservices for example. This is due to the fact that they can share resources like domain model, imported modules, persistent data storages. So, the rule of thumb would be to entrust developing of such interdependent nanoservices to the same team, reducing in such way dependencies between teams.

Easy deployment

Due to their small size and scope, nanoservices can be developed and deployed quickly. This supports a more agile development process, enabling rapid iteration, testing, and deployment of new features or updates. "With the immediate execution environment offered by serverless platforms, rapid deployment, and iteration become the norm" [38, p. 343]. P. Sbarski and others claims that "a typical deployment takes

less than a minute and has no downtime because AWS Lambda automatically routes requests to the new code" [44, p. 60].

Automated deployment is easier with nanoservices compared to other architectures – tools and platforms that support continuous integration and continuous deployment (CI/CD) can easily manage frequent updates and rapid deployment cycles that are common with nanoservices. For example, GitLab CI/CD Pipeline and GitHub Actions provide means to the automated deployment of serverless applications [50, 51].

Technology Diversity

The independence of nanoservices allows for the use of different technology stacks or programming languages best suited for each service's functionality. This technological flexibility can optimize performance and developer productivity.

Although, the variety of technologies for nanoservices is usually limited by the FaaS platforms (runtimes), available in cloud providers. Usually, the most popular runtimes are available (AWS supports different versions of Node.js, Python, Java, .NET, Ruby, almost the same in Azure and Google Cloud), though for using less common languages should be considered other options, like using own FaaS platform – OpenFaaS for example, which allows to write functions in any language [52, 53, 54, 55]. Another popular tool for building applications with serverless architectures is Serverless Framework which allows deploying serverless architectures to different cloud platforms [56].

In summary, the benefits of nanoservice architecture in modern development are numerous and significantly impactful. Nanoservices' compatibility with cloud computing and FaaS platforms enables them to leverage serverless computing benefits fully, making them particularly suitable for environments with unpredictable loads. This capability allows for cost-effective scalability and efficient resource utilization, where payment is based only on actual usage. Furthermore, the granular scalability facilitated by autoscaling features inherent in serverless platforms ensures that nanoservices can respond rapidly and effectively to varying demands. The architecture

also supports independent development and deployment, allowing specialized teams to work concurrently on different aspects of the system, thus speeding up development cycles and reducing the risk of widespread system impacts from localized changes. Overall, the agility, resource efficiency, and scalability offered by nanoservice architecture make it a compelling choice for modern software development in dynamic and demanding operational contexts.

While nanoservices offer these advantages, it’s important to balance them with the challenges and overheads such as increased complexity in service management and communication, potential for redundancy, and the overhead of coordinating numerous deployments.

2.2 Issues arising with nanoservice architecture

While nanoservices offer numerous benefits in terms of granularity, flexibility, and scalability, they also come with a set of disadvantages that can impact their overall effectiveness and efficiency. Here are the main disadvantages associated with nanoservices:

- **increased complexity**: managing a large number of highly granular services can significantly increase the complexity of the system architecture. This complexity affects not only the development and deployment processes but also ongoing maintenance, monitoring, and troubleshooting;
- **communication overhead**: nanoservices require frequent communication between services. The overhead of managing these interactions, especially in a highly distributed system, can lead to increased latency and can impact the system's performance;
- **difficulty in managing data consistency**: ensuring data consistency across numerous, independently managed nanoservices can be challenging. The distributed nature of data in such architectures may require sophisticated strategies for transaction management and data synchronization;

- **operational overhead**: deploying, scaling, and monitoring a vast number of nanoservices can introduce significant operational challenges. The overhead associated with managing the lifecycle of each nanoservice can strain resources and complicate operational processes;
- **possible increased infrastructure costs**: although nanoservices can be scaled independently, the overhead of running numerous small services – each potentially requiring its own runtime environment – can lead to higher infrastructure costs compared to more consolidated architectures. Additionally, this nanoservice architecture can lead to the issue known as the "double spending problem", which we will cover in the next sections;
- **increased latency in processing requests from clients**: since our system is broken into small services that need to communicate, each communication involves creating a TCP connection, and waiting for each involved nanoservice to start – all these steps introduce additional latency in the overall request processing;
- **development overhead**: while the idea of small, focused teams developing independent services is appealing, coordinating development across many teams and ensuring consistency and compatibility between nanoservices can be cumbersome and time-consuming;
- **security concerns**: the increased surface area for potential attacks, due to the number of services and communication points in a nanoservice architecture, can raise security concerns. Implementing robust security measures across all services can be more complex and demanding;
- **testing challenges**: comprehensive testing becomes more challenging in nanoservice architectures due to the number of services and their interactions. End-to-end testing strategies must account for the complexity and dynamism of the service interactions, which can be resource-intensive.

Additionally, V. K. Thatikonda names challenges related to serverless computing (which are also applied to nanoservices if they are deployed with serverless approach):

- cold starts: “a cold start occurs when a new instance of a function is initiated, and there’s a latency involved before the function begins executing, primarily because the environment needs to be set up from scratch” [57];
- vendor lock-in: aligning with a specific provider’s toolset, infrastructure, and services, which can lead to vendor lock-in, where migrating to a different platform becomes time-consuming and costly due to incompatible interfaces and configurations [38, p.344];
- limited customization and control: serverless platforms often impose certain restrictions on runtime environments, memory limits, and execution durations, which can limit the scope for deep customization, which might be essential for some niche applications [38, p.344];
- security concerns: “with serverless, while the provider primarily handles infrastructure security, the application's security remains a shared responsibility” [38, p.344]. This model means developers are responsible for securing their application code, configurations, and third-party libraries [58]. Serverless environments may also introduce unique security vulnerabilities. For instance, the dynamic nature of serverless may expose applications to event-data injections or potential unauthorized function invocations [58].

And to this bunch of challenges P. Sbarski and others add some more:

- “You are not comfortable with public cloud-based architectures” [44, p. 15];
- “The services don’t meet the availability, performance, compliance, or scale needs of your customers” [44, p. 15];
- “Your application and business needs require you to stay vendor agnostic” [44, p. 16].

E. Wolff adds that “using nanoservices can limit the choice of programming languages, platforms, and frameworks” [10, p. 345]. As an opposite example he gives microservices, which enable, in principle, a free choice of technology.

Some of these explicit disadvantages lead to other, more implicit challenges. For example, increased infrastructure costs may lead to the “double spending problem”, operational overhead – to the services versioning challenge, and the short-lived nature

of nanoservices – to the challenge of handling sessions, which need to live longer than the service.

While nanoservices can provide significant advantages in specific scenarios, it's crucial to weigh these benefits against the potential disadvantages. For some projects, the overhead and complexity introduced by a nanoservice architecture may not justify the granularity and flexibility it offers. Let's look at some of these challenges closer.

2.2.1 Nanoservices as antipattern in microservice architecture

The term "nanoservices" can be used not only in the context of a particular architecture style but also to describe poor system design in microservice architecture.

For example, when defining microservice architecture M. Fowler and J. Lewis describe the issue of shifting complexity from inside a component to the connections between components. They say that "not just does this just move complexity around, it moves it to a place that's less explicit and harder to control. It's easy to think things are better when you are looking at the inside of a small, simple component, while missing messy connections between services" [35].

C. Richardson defines microservice as "a service capable of being developed by a small team with minimal lead time and with minimal collaboration with other teams" [31, p.43]. He says that "the microservice architecture structures an application as a set of small, loosely coupled services" [31, p.43]. So, for microservice architecture it is important to be loosely coupled, but how nanoservices can be antipattern here?

R. Tighilt and others explore and summarize microservices antipatterns, and in their study they refer to such antipatterns as "mega microservice" and "nano microservice" [59]. With regards to this they say that "the nano microservice antipattern exists when (1) the system has a large number of microservices; (2) microservices exchange a lot of information; or, (3) cyclic dependencies exist" [59]. As a result, this leads to the coupling of services, thus microservices lose their independence.

R. Gall also mentions the nanoservices pattern as an antipattern, saying that “in the case of nanoservices, overheads such as communication and maintenance activities outweigh its utility. Nanoservices should be avoided” [60]. He provides an example of such an antipattern where for each database table a separate service is created and those services expose CRUD operation using events or a REST API.

M. Richards adheres to the point of view that “service components that are too fine-grained will lead to service orchestration requirements, which will quickly turn your lean microservices architecture into a heavyweight service-oriented architecture, complete with all the complexity, confusion, expense, and fluff typically found with SOA-based applications” [61, p. 32].

With that being said, how can we distinguish nanoservices as an antipattern in microservices and nanoservices as an architectural style?

From our perspective, the main difference lies primarily in the initial system design and the communication patterns within it. If the system was initially designed as a microservice-styled with appropriate service isolation and communication patterns, then too fine-grained or small services in such a system can be referred to as the nanoservice antipattern, eliminating microservices benefits. On the contrary, if the system was initially designed as a nanoservice-styled with nanoservice communication patterns, then we can refer to such a system as nanoservices by design.

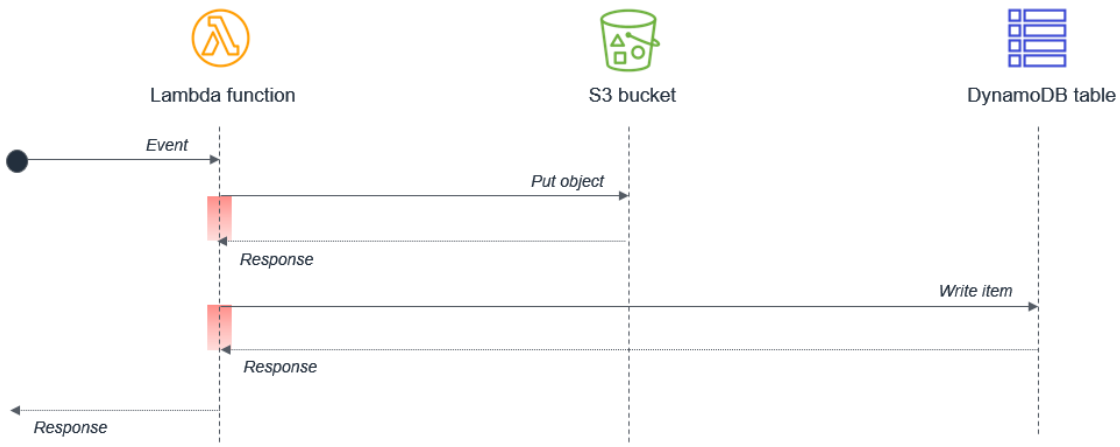
2.2.2 Double spending problem

One of the antipatterns in the serverless approach and therefore nanoservice architecture as well is a synchronous communication between services or inside the system. Let’s delve into this and see why it is an antipattern and what issues it may bring.

AWS documentation recognizes such communication in Lambdas as an antipattern and describes it in detail [62]. According to that, within a single Lambda, an engineer should “ensure that any potentially concurrent activities are not scheduled

synchronously" [62]. Here is the example of such synchronous communication (provided by AWS):

Figure 2. Synchronous communication in AWS Lambdas

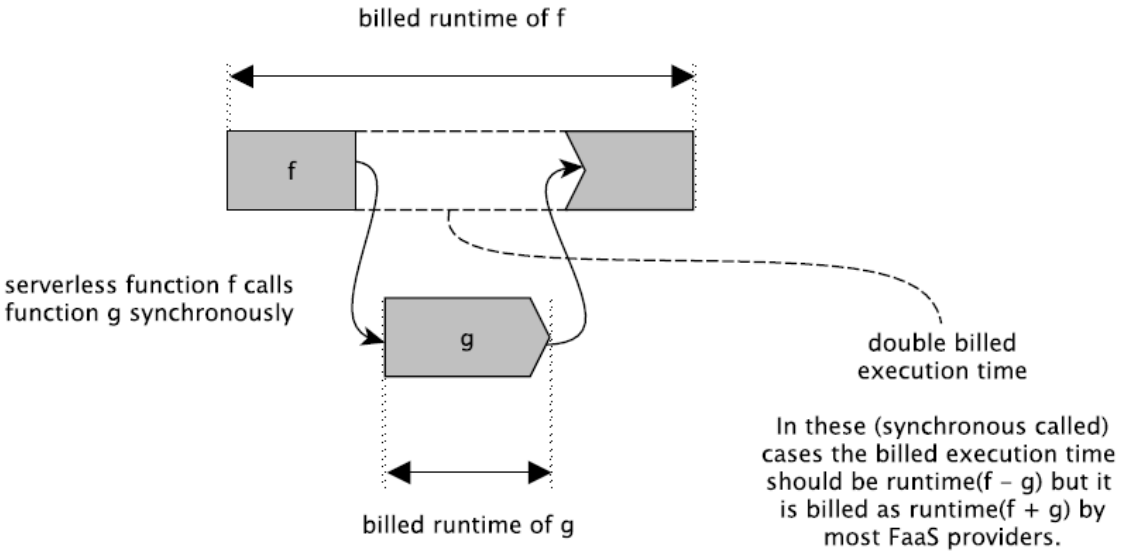


In picture 2 we can see that the lambda has two "wait" states, marked with red when it waits for results from two requests – from the S3 bucket and the DynamoDB table. Why may this be an issue with nanoservices?

Well, first of all, as we mentioned earlier, nanoservices allow to utilize the benefits of a serverless approach and, thus are usually built on top of the FaaS platform. As we know, FaaS generally provides limited operation time per function. So, such waiting periods can lead to failures when lambda operation time together with waiting time takes longer than the time allowed by cloud providers for FaaS.

And secondly, this leads to another problem – the "double spending problem". N. Kratzke and R. Siegfried describe this problem as the problem that "occurs when a serverless function f is calling another serverless function g synchronously. The consumer is billed for the execution of f and g – although only g is consuming resources because f is waiting for the result of g." [63, p.8].

Figure 3. Double spending problem with nanoservices



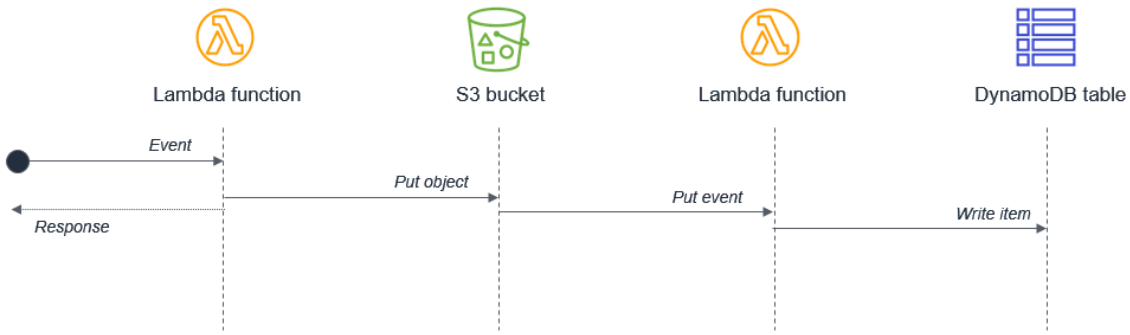
In paragraph 2.1 of this chapter, we mentioned cost efficiency as one of the major benefits of nanoservice architecture. Nanoservices combined with serverless could potentially be much more effective in terms of cost compared to other architectures. But at the same time, this benefit can be completely eliminated by the described double spending problem if the communication in the system is poorly designed, without addressing this issue.

How can this issue be addressed? The answer is straightforward – avoid waiting time in async operations/communication. This does not mean not to use async operations in nanoservices, but rather use them without the need to wait for the result. N. Kratzke and R. Siegfried suggest that "many serverless applications delegate the composition of fine-grained serverless functions into higher order functionality to client applications and edge devices outside the scope of FaaS platforms " [63, p.8]. So, though async operations are still present in the system, it's a client application that waits and combines the results.

AWS offers different solutions to this problem. Firstly, if nanoservice performs several async operations and they do not depend on each other results – they should be run in parallel [62]. In this case, the total wait time will be set by the longest-running task. Another possible solution is breaking down one nanoservice into several, splitting

it by async operation. In such a scenario (example in figure 4) each lambda performs its operation without the need to wait for any async results.

Figure 4. Breaking down nanoservice into two services



To summarize, nanoservice architecture is a useful architectural style, providing ease of scaling and cost-effectiveness to the system based on this architecture. The latter though can be dismissed if the nanoservices are designed poorly, which means introducing waiting periods during asynchronous operations. Avoiding such waits is a key to building a cost-effective nanoservice system.

2.2.3 Interservice communication

Communication in each architecture between its elements is an integral part of the system, shaped and defined by this architecture, and it went a long path of evolvement, together with the architecture itself, so let's see here how architecture affects the fundamentals of communication inside of it.

Since there is no need to communicate between elements in the monolith architecture (as it is a single service in the essence), we can say that Service-Oriented architecture was one of the first where the issue of communication arose and needed to be addressed. There is a standard issued by W3C that defines this architecture (Web Services Architecture), what a Web service is, and how communication with such service should be done [64]. According to this standard, "a Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically

WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.” [64, p.7]

As we can see from this definition, the standard prescribes that communication with Web services should be based on SOAP and WSDL standards. Mayur R. Mehta and others mention that the Web services framework at the minimum consists of a collection of three standards:

- standard to support communication between interacting services,
- standard to describe services, and
- standard to publish and discover services [30, p.4].

And for these purposes serve SOAP, WSDL, and UDDI. SOAP is an XML-based protocol to support communication between a Web service, its clients, and UDDI registry [30, p.4].

Figure 5. Skeleton of SOAP Message

```
<?xml version="1.0"?>

<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

<soap:Header>
...
</soap:Header>

<soap:Body>
...
  <soap:Fault>
  ...
  </soap:Fault>
</soap:Body>

</soap:Envelope>
```

The WSD (Web service description) is a machine-processable specification of the Web service's interface, written in WSDL [64, p.8]. According to the standard, it defines the message formats, datatypes, transport protocols, and transport serialization formats that should be used between the requester agent and the provider agent.

Figure 6. WSDL document example

```
<message name="getTermRequest">
  <part name="term" type="xs:string"/>
</message>

<message name="getTermResponse">
  <part name="value" type="xs:string"/>
</message>

<portType name="glossaryTerms">
  <operation name="getTerm">
    <input message="getTermRequest"/>
    <output message="getTermResponse"/>
  </operation>
</portType>
```

And UDDI standard is an XML-based lookup service for locating Web services in an Internet Topology [65]. It is used to publish, discover, and manage Web services in an UDDI registry.

So, we can agree with Mayur R. Mehta and others that SOA promotes the idea of using loosely coupled services, each of which represent a block of business functionality, that can be assembled together to support a business process [30, p.8].

As we can see, communication in SOA-based architectures is highly standardized, with a few commonly adopted and used standards and formats. However, transitioning to microservices has led to the implementation and adoption of many different standards and approaches in response to new challenges.

With regards to microservice architecture M. Richards and N. Ford admit that architects must decide on **synchronous** or **asynchronous** communication, where synchronous communication requires the caller to wait for a response from the callee [66, p. 254]. According to them, microservices architectures typically utilize protocol-aware heterogeneous interoperability, which means:

- protocol-aware – each service should know how to call other services, thus, architects commonly standardize how particular services call each other;
- heterogeneous – supporting polyglot environments, as each service may be written in a different technology stack;
- interoperability – describes services calling one another [66, p. 255].

M. Richards and N. Ford also compare **choreography** and **orchestration** styles of communication between microservices. In choreography style no central coordinator exists in this architecture, respecting the bounded context philosophy, in orchestration, on the contrary, such coordinator is used (can be called mediator as well), whose sole responsibility is coordinating the call [66, p. 256-257].

Of course, each of the described styles has its trade-offs. M. Richards, for example, recommends avoiding orchestration, because it will "quickly turn your lean microservices architecture into a heavyweight service-oriented architecture, complete with all the complexity, confusion, expense, and fluff typically found with SOA-based applications" [61, p. 32]. He also mentions that orchestration could be the result of poor service design – "if you find you need to perform inter-service communication between service components to process a single request, chances are your service components are either too fine-grained or they are not partitioned correctly from a business functionality standpoint" [61, p. 32].

On the other hand, in a choreographed environment error handling and coordination are more complex [66, p. 258].

But we need to admit that the separation between pure choreography and orchestration styles is rather theoretical, in practice systems usually have the features of both. Though services in general can communicate in a choreographed style, some particular flows can be orchestrated by some services. In such situations there are no explicit mediators, they are implicit.

C. Richardson categorizes service interaction styles by synchronous/asynchronous and one-to-one/one-to-many [31, p. 67]:

Table 1. Inter-service communication styles

	one-to-one	one-to-many
Synchronous	Request/response	—
Asynchronous	Asynchronous request/response One-way notifications	Publish/subscribe Publish/async responses

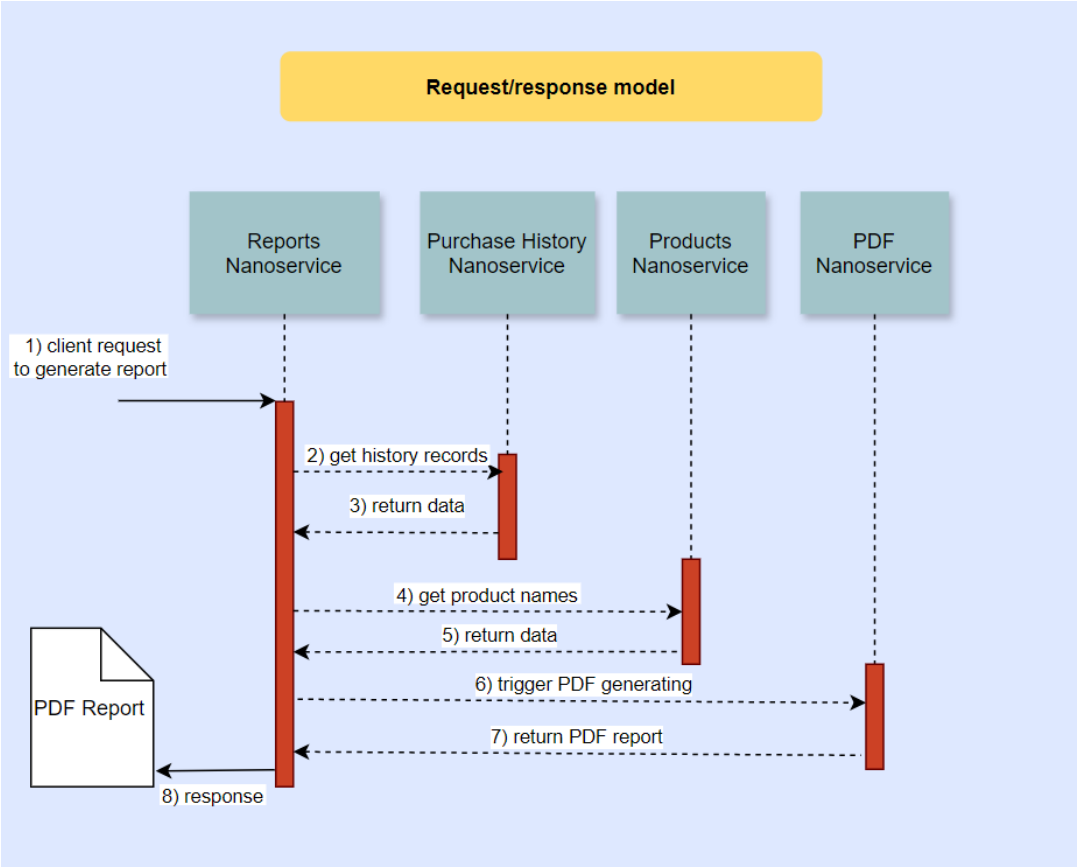
In this table one-to-one means each client request is processed by exactly one service, and one-to-many – each request is processed by multiple services. C. Richardson emphasizes that the choice to use a synchronous request/response interaction style does not depend heavily on the specific inter-process communication technologies employed [31, p. 68]. Services can engage with each other using this interaction style through various methods, such as REST or messaging systems. Additionally, he highlights that even when services use a message broker for communication, the client service may still be blocked while waiting for a response.

Let's see how request/response and publish/subscribe communication models could be used for implementing report-generating functionality. Let's assume that we need to generate a PDF report with aggregated products sold by a shop for some period of time, and we have four nanoservices – Reports (processing requests from clients), Purchase History (providing history data of purchases), Products (providing products info like names) and PDF (generating reports by given data).

So, in Figure 7 we can see a sequential diagram with request flow from the client starting with the request to generate a report and ending with the response with the report. In this diagram, client makes a request and waits for a response, and Reports Nanoservice also needs to wait until the result is ready. This raises several concerns with regard to nanoservice architecture:

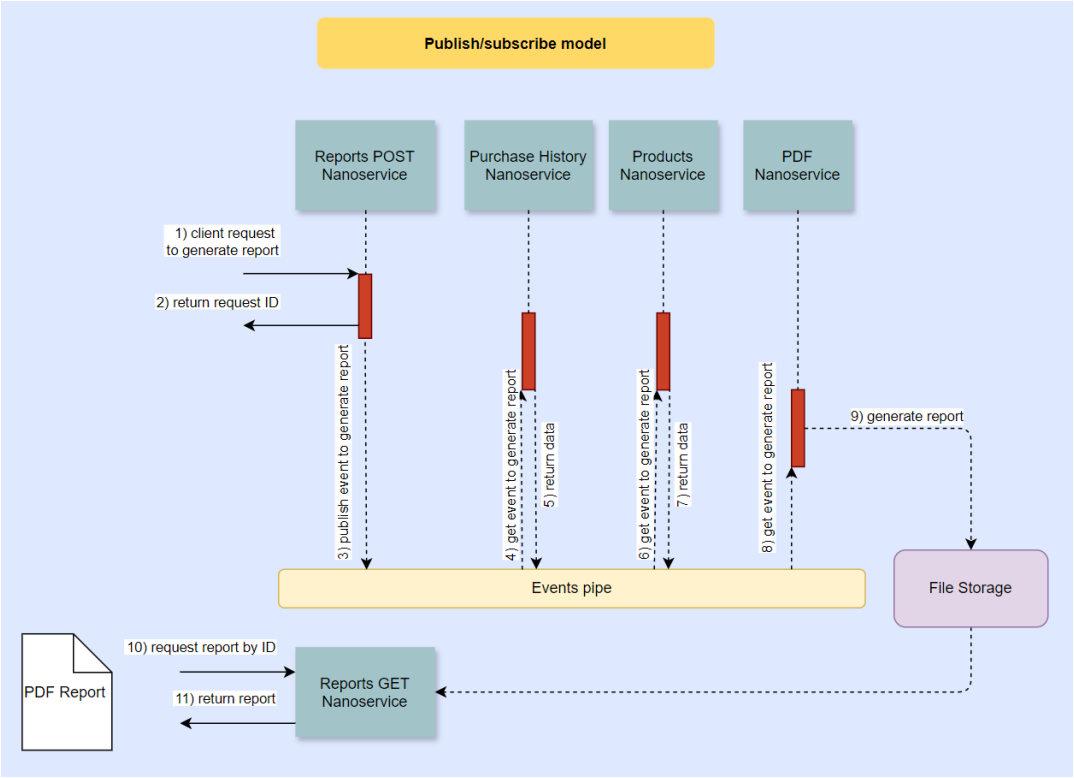
- nanoservices work in a synchronous manner, as the very first nanoservice ("Reports") needs to wait for the results to return to the client – which causes the double spending problem discussed earlier;
- in this model orchestration is present, which creates tight coupling between the services (Reports service need to know about all other services – their names, locations, and to coordinate their work).

Figure 7. Request/response model for report generating



Now let's look at a similar case, but implemented with respect to the publish/subscribe communication model. The diagram in Figure 8 shows that client makes a call to Reports service, which publishes event in the pipe and returns ID of request to the client. At this moment Reports service finishes its work, other services are subscribed to event pipe, receive events and process them. This model is called publish/subscribe because the client here "publishes" event to generate a report and does not wait for the immediate response; instead, he retrieves the ID of the request, which can be used to query the Reports service about report generating state – when the report is ready, it will be returned.

Figure 8. Publish/subscribe model for report generating



With the second approach (pub/sub), we address our previous concerns:

- nanoservices operate asynchronously, meaning that no service waits for the result of any others – which is cost-effective;
- this model employs choreography-based communication, as no service has knowledge of or coordinates/aggregates the work of others.

Message formats

What differentiates microservices and nanoservices architectures from SOA, is that in microservices (and especially nanoservices) the volumes of communication increase significantly. In our opinion, it is accurate to say that the more a system is broken down into fine-grained modules (services), the more communication traffic it generates. For that reason, such excessive message formats like SOAP can be used in service-oriented architectures, but are usually avoided in microservices, and nanoservices as well.

Generally, two types of message formats are used in inter-service communication– text-based and binary.

Text-based message formats (like XML and JSON) are convenient because of being human-readable and self-describing. At the same time, they have many disadvantages, becoming more explicit in nanoservice architecture:

- increased payload size (text-based formats typically result in larger message sizes compared to binary formats);
- processing overhead (parsing text-based messages requires more computational resources than binary formats);
- security concerns (text-based data formats can be more susceptible to injection attacks and other security vulnerabilities if not properly handled and sanitized, also N.Dragoni and others add that particular attention should be paid to providing security of the data being transferred [23, p. 9]);
- verbosity (text-based formats are more verbose and include additional information related to describing the schema of the data).

So, in microservice architecture (and that can be applied to nanoservices as well) over-verbose formats like SOAP are rarely used, instead simple text-based message formats like JSON or binary ones are preferred [67, p. 8].

In all the above-mentioned aspects, binary formats surpass text-based ones. The most popular binary formats include Thrift, Protocol Buffers, and Avro. All of them are efficient and good for messaging in microservice and nanoservice architectures, but in C. Richardson's opinion Protocol Buffers format is better for easiness of API evolution. That is because Protocol Buffers uses tagged fields, whereas an Avro consumer needs to know the schema in order to interpret messages [31, p. 72].

Thus, when deciding upon message formats for interservice communication in nanoservice architecture, we need to consider that this type of architecture produces the most intensive communication, compared to other architectures, therefore the most performant and efficient formats should be preferred – either JSON or binary, like Protobuf or Avro.

To summarize, when designing inter-service communication in nanoservices, it is advantageous to prioritize asynchronous communication over synchronous methods to reduce infrastructure costs and enhance system decoupling and scalability. Embracing

choreography instead of orchestration also helps in decoupling services, allowing them to operate more independently and resiliently. Opting for the publish/subscribe model rather than the traditional request/response pattern further reduces dependencies among services, promoting a more robust and scalable architecture (and eliminating synchronous interaction as well). Finally, employing space-efficient message formats ensures that communication remains fast and efficient, which is vital for maintaining high performance in nanoservice architectures. By adhering to these principles, organizations can build more dynamic, scalable, and maintainable systems.

2.3 Patterns of communication in a nanoservice architecture

Now that we have revealed issues related to nanoservice architecture as well as interservice communication basics, let's see which communication patterns can be applied to the nanoservice architecture, and which are better to avoid. In this paragraph, we are going to discuss such models of communication as request-based and event-based, and such patterns as Request-Reply Pattern, Command Pattern, Event-Driven Communication, The Saga, Two-Phase Commit, and some others.

Request-based and event-based models

M. Richards and N. Ford believe that applications are either request-based or event-based, with most applications following a request-based model [66, p. 179]. In their view, in a request-based model requester makes a request to a request orchestrator, which passes it further to various request processors, which in their turn fulfill the request and provide the requested information. On the contrary, in the event-based model system reacts to a particular event and takes the needed action. An example of such a system could be an online auction, where a bid is the mentioned event; "submitting the bid is not a request made to the system, but rather an event that happens after the current asking price is announced" [66, p. 179].

In the previous paragraph, we discussed request/response and publish/subscribe communication styles, and in our opinion, there is a correlation between these

communication styles and the mentioned application models: request-based model uses request/response communication style, and event-based – publish/subscribe accordingly.

Request-Reply Pattern

The aforementioned request/response is generally treated as synchronous, though it can be also implemented in asynchronous way [68]. Such a pattern is also known as "Request-Reply", and it provides the ability to perform synchronous communication in event-driven architecture (sometimes called "pseudosynchronous communication") [66, p. 204]. The main idea of this approach is communication based on two queues: a request queue and a reply queue. An initial request is sent to the request queue, where it gets taken and processed by the message consumer, after which the response gets put into a reply queue. As this is done using asynchronous events, the main challenge here is for a requester to understand which response refers to which request. This can be done usually in two ways: either using correlation ID in message headers or using temporary reply queues, dedicated to each request [66, p. 204-206].

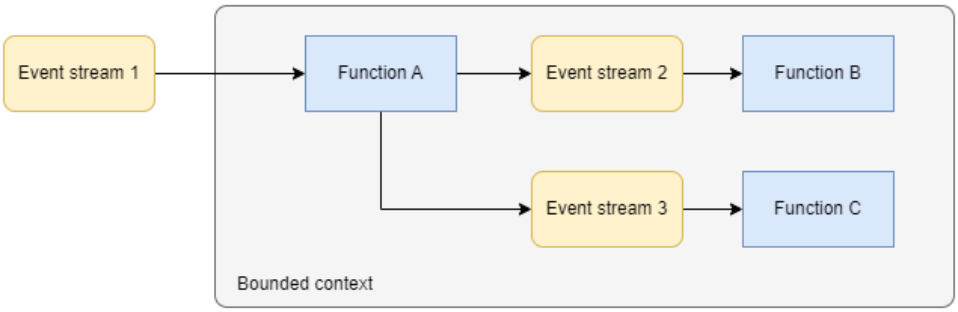
We can say that this communication pattern combines both models – request-based and event-based at the same time, because on one hand it consists of request and response parts of request-based model, and on the other hand it is based on events, and can be implemented in a synchronous or asynchronous way. The latter is the most significant for use in the nanoservice architecture, which is inclined to be event-driven. As we mentioned earlier, event-based communication (publish/subscribe model) should be preferred in the nanoservice architecture for many reasons (scalability, decoupling, cost efficiency), and the Request-Reply pattern allows to set request-response communication over event-driven architecture.

Event-Driven Communication Pattern

This pattern belongs to the event-based communication model. Adam Bellemare describes this pattern by suggesting that the output of one function can be placed into an event stream to be consumed by another function [69, p. 161]. According to him,

each function processes incoming events at its own rate, with events being consumed, work being performed, and outputs produced accordingly.

Figure 10. Event-Driven Communication Pattern schema



This pattern has several benefits, like each function can work with its own consumer offsets or no coordination is needed (meaning that this pattern involves choreography style of communication), but the most important is that functions (nanoservices) are decoupled from each other, not blocking each other – no function waits for the results from others, but rather each function gets triggered when there are unprocessed events in the stream it is subscribed to. This pattern is asynchronous by its nature, letting each service to work at its own pace and avoiding the idling of services.

P. Sbarski and others also describe such a pattern as a “Messaging pattern”, which involves message queues, “decoupling functions and services from direct dependence on one another and allowing storage of events/records/requests in a queue” [44, p. 47]. In our opinion, this is quite similar to the Event-Driven Communication Pattern, without any distinguishable differences, so we will consider this as the same pattern.

Direct-Call Pattern

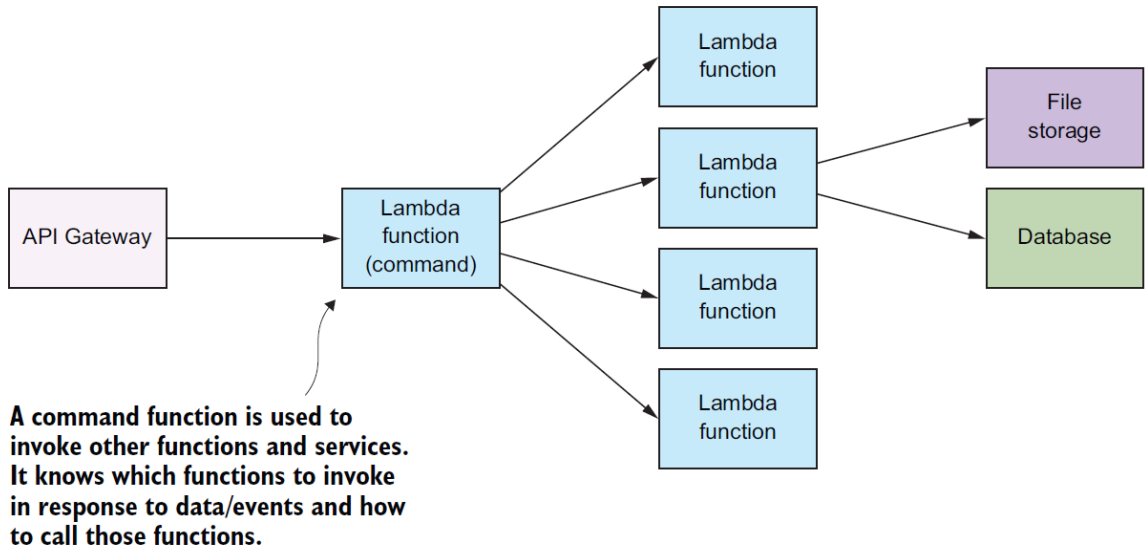
Adam Bellemare identifies it as a distinct pattern when one function directly calls another [69, p. 162]. It can be done in synchronous and asynchronous ways (depending on whether the caller waits for the return value or not).

Though this is the easiest way of interaction between nanoservices, we would suggest avoiding using this pattern, as it significantly increases the coupling between services and reduces isolation between services.

Command Pattern

In this pattern, there is a service, which accepts commands and passes them to other services. P. Sbarski and others believe that this pattern is useful if you want to decouple the caller and the receiver [44, p. 47]. They mention that "having a way to pass arguments as an object and allowing clients to be parametrized with different requests can reduce coupling between components and help make the system more extensible".

Figure 9. Command Pattern schema

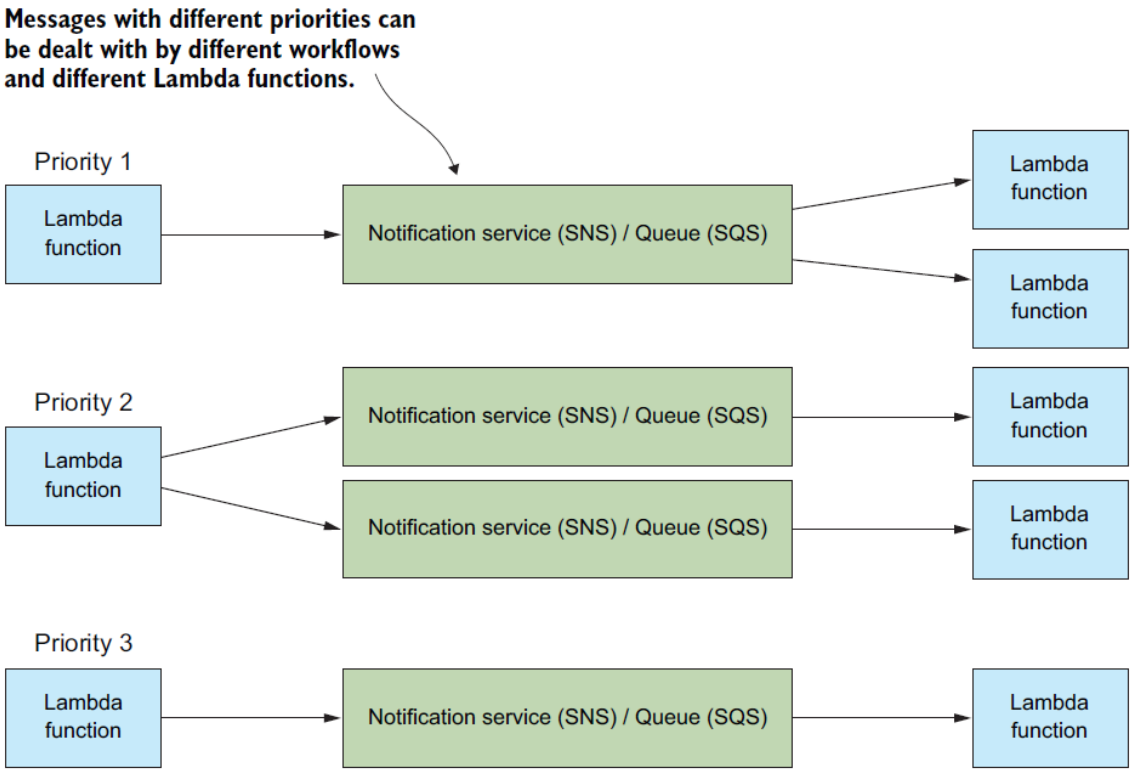


This pattern can be used to decouple the caller from the receiver. This prevents from the caller being "smart" and knowing what function/service should be called for each operation; also, it allows to reorganize/modify receivers without affecting the caller.

Priority Queue Pattern

P. Sbarski and others describe this pattern, where messages with different priorities can have different workflows [44, p. 49]. For instance, messages with higher priority can go through the flow with more resources or capacity for processing.

Figure 10. Priority Queue Pattern schema



This pattern can be used to improve the availability of the most critical part of the nanoservice-based system and guarantee that it's not overloaded with messages of low importance.

Let's now discuss patterns that provide data consistency and are used for handling transactions across multiple distributed services – patterns "Saga" and "Two-Phase Commit", what are the differences between them, and whether either is better for nanoservice architecture.

Pattern "Saga"

Saga is a pattern for maintaining data consistency in the distributed system, which is a sequence of local transactions [70]. Each local transaction updates data within a

single service. C. Richardson assumes that the pattern “Saga” is a mechanism to maintain data consistency in a microservice architecture without having to use distributed transactions, while A. Bellemare claims the opposite – that the saga is the name for distributed transactions in an event-driven world [31, p. 114; 69, p. 144]. A. Bellemare’s opinion is based on understanding of a distributed transaction – a transaction that spans two or more microservices [69, p. 144]. To advocate his approach, C. Richardson provides differences between sagas and ACID transactions:

- sagas lack the isolation property of ACID transactions;
- in sagas changes must be rolled back using compensating transactions, as each local transaction commits its changes [31, p. 114].

At the same time, both authors agree that saga can be implemented in different ways, and generally, there are two types of sagas: choreography-based and orchestration-based. C. Richardson stands for that saga can be built using asynchronous messaging for coordinating local transactions, whereas A. Bellemare contends that sagas may use direct request-response calls in orchestration scenarios [31, p. 114; 69, p. 147].

In choreography-based saga there is no central coordinator to manage workflow participants, each of them subscribes to each other’s events and responds accordingly. This approach has obvious benefits – simplicity of services (which just publish event on each performed business action) and loose coupling (participants have no knowledge about each other). At the same time, C. Richardson considers the following as drawbacks: they are more difficult to understand, there is a possibility of cyclic dependencies between services, and there is a risk of tight coupling because each service needs to subscribe to all events that affect it [31, p. 121]. Because of these drawbacks, he advises preferring more complex orchestration sagas over choreography-based.

From our perspective, cyclic dependencies are particularly undesirable in nanoservice architecture because they can lead to infinite loops of service triggers and, consequently, to resource leaks. Furthermore, AWS official documentation warns to avoid using recursive patterns that cause run-away Lambda functions [71]. It says that

"this anti-pattern has the potential to consume more resources in serverless applications… the loop may cause Lambda to scale to consume all available concurrency" [71].

Therefore, using orchestration sagas is more safe in nanoservice architecture. But, on the other hand, implementing orchestration in nanoservices is hard for several reasons. The most significant issue in our opinion is keeping a state of orchestrator – as we remember, nanoservices as cloud functions have limited time of operation. This means that the current state of the orchestrator during processing of the particular transaction should be stored somewhere persistently while waiting for responses from transaction participants. In its turn, this adds additional complexity and latency to the whole transaction.

So, for nanoservice architecture there is no absolute preference between types of sagas – both choreography and orchestration have their own benefits and disadvantages and can be chosen according to the project needs.

Pattern "Two-Phase Commit"

There are a bunch of definitions for such a pattern in the literature. James Kwon defines a Two-phase commit (or 2PC) as an algorithm for achieving atomic transaction commit across multiple nodes [72]. Chris Richardson's definition is the algorithm "to ensure that all participants in a transaction either commit or rollback" [31, p. 112]. And Martin Fowler's definition is just "update resources on multiple nodes in one atomic operation" [73].

In any case, 2PC according to its name consists of two phases – the prepare phase and the commit phase. In the first phase, a coordinator initiates a transaction, checking whether all of the nodes can commit the transaction (promise to carry out the update), while in the second the nodes are instructed to either commit or abort the transaction. Additionally, in the first phase, all participants need to acquire whatever they need to be able to do the commit in the second phase – like acquiring locks for any resources.

One of the outcomes here is that 2PC is coordinated, meaning that it follows the orchestration model. It can be choreographed, as it needs somebody to coordinate the

phases. Another outcome is that it has a synchronous nature. Chris Richardson says that the problem with distributed transactions is that “they are a form of synchronous IPC, which reduces availability. In order for a distributed transaction to commit, all the participating services must be available” [31, p. 113]. James Kwon also adds that 2PC is often called a “blocking” atomic commit protocol (or “anti-availability” protocol) because all nodes/members must be up for it to work [72].

Having discussed that, we can elaborate on key differences between Saga and 2PC, as both of them have much in common – they are both methods for handling transactions across multiple distributed services. These differences include the following:

– consistency: 2PC ensures immediate consistency (all or nothing), while sagas ensure eventual consistency through compensating transactions. Chris Richardson admits that today, architects prefer to have a system that’s available rather than one that’s consistent [31, p. 114];

– performance and resilience: 2PC can introduce performance bottlenecks and resilience issues due to its synchronous nature, whereas sagas, being asynchronous, typically offer better performance and fault tolerance;

– compensation: 2PC does not require compensation logic as it either commits or rolls back all changes atomically. Sagas, on the other hand, require explicit compensation logic for failed transactions.

These differences make 2PC suitable for environments where immediate consistency is critical, and the performance cost is acceptable. Sagas can be preferred in distributed systems where service independence, scalability, and resilience are more important.

Concerning the nanoservice architecture, we can conclude that nanoservice architectures are designed to be highly scalable and performant, with a focus on small, independent, and rapidly deployable units of functionality. The synchronous and blocking nature of 2PC can introduce significant performance bottlenecks because all involved services must wait during the commit or rollback phase, which can be highly undesired in a distributed environment with many small services. During the first phase

of 2PC, resources are locked until the transaction is either committed or rolled back. This locking can lead to increased latency and decreased availability, both of which are critical factors in nanoservice environments where services are expected to be highly responsive and minimally dependent on each other.

Besides this, 2PC is vulnerable to failures, particularly regarding the coordinator. If the coordinator fails during the transaction, it can leave participating services in an indeterminate state, waiting for a commit or rollback command. This issue is magnified in nanoservices due to the large number of services that might be involved.

While 2PC can be implemented in a nanoservice architecture, the disadvantages typically outweigh the benefits, especially concerning performance, scalability, and system resilience. Adopting more loosely coupled transaction management patterns, like sagas, is generally more aligned with the principles of nanoservice architecture. If transaction management in a nanoservice architecture is needed, using the saga pattern can give more flexibility in implementation, more efficiency in resource utilization and more overall system availability with asynchronous nature of saga.

So, in this paragraph various communication models such as request-based and event-based, along with specific patterns like Request-Reply, Command Pattern, and Event-Driven Communication, were explored. The consensus is that while traditional request/response methods have their place, the nanoservice architecture significantly benefits from event-driven approaches, particularly due to their scalability, decoupling capabilities, and cost efficiency. Patterns such as the Event-Driven Communication Pattern facilitate asynchronous interactions that prevent services from blocking each other, thereby enhancing overall system responsiveness and reliability. Additionally, we have revealed the adaptability of the Command Pattern in reducing dependencies and simplifying system extensions. Our analysis concluded with insights into complex transaction handling in distributed services, advocating for the Saga pattern over the Two-Phase Commit in nanoservice architectures due to its non-blocking nature and better fault tolerance, which are crucial for maintaining the high scalability and performance inherent to nanoservices.

2.4 Use cases for nanoservice architecture

After exploring the pros and cons of nanoservice architecture, as well as communication fundamentals in nanoservices, let's discuss which scenarios are best suited for this type of architecture, and which are not.

When deciding which use cases suit the most nanoservices architecture, we need to keep in mind these peculiarities of nanoservices. As we mentioned in the first paragraph of this chapter, the main advantages of nanoservices include easy scaling (autoscaling), cost efficiency, and independent development. However using this architecture in inappropriate scenarios or in a wrong way can completely eliminate these advantages, leaving only the disadvantages of this architecture. So what scenarios are appropriate?

Ido Vapner describes four main cases when nanoservices are appropriate [74]. Here are they:

- scaling is required;
- processing demands spikes;
- rapid application development;
- handling user events.

As for the first category (scaling is required) he says that “because each function is small and independent, you can scale specific functions without having to scale the entire application” [74]. We discussed earlier that scaling on demand is one of the major powers of FaaS platforms, which handle easily huge traffic spikes, and because nanoservices are quite small they can be bootstrapped much quicker than other types of services (e.g. EC2-based services). The second category (processing demands spikes) is naturally a subsequent of the first one, as easy scaling helps to process these spikes. The third category (rapid application development) – nano services architecture can help to develop and deploy an application quickly as the small, single-purpose functions can be developed and deployed independently, making it easier to test and iterate on specific functions [74]. Finally, nanoservices are suitable for handling user

events, as each nanoservice is a function that can represent a handler for a specific user event.

Peter Sbarski, Yan Cui, and Ajay Nair describe use cases of serverless computing, and we need to admit that they suit nanoservices either [44, p. 41]. They include:

- backend compute – some companies like A Cloud Guru, Comcast, Coinbase, Fender, Nordstrom, and Netflix run serverless backends, based on serverless solutions;
- Internet of Things – serverless application backend (and Amazon's IoT platform) removes a lot of infrastructure management, has granular and predictable billing, and can scale well to meet uneven demands;
- data processing and manipulation – nanoservices and serverless are appealing for building data processing pipelines, like collation and aggregation of data, image resizing, and format conversion;
- real-time analytics – serverless solutions are effective for ingestion of data such as logs, system events, transactions, or user clicks;
- legacy API proxy – serverless can be used to create a new API layer over legacy APIs and services, which makes them easier to use; for example, this approach may be used to add a RESTful interface over legacy service;
- scheduled services – FaaS is effective for repetitive tasks like data backups, imports and exports, reminders, and alerts;
- bots and skills – FaaS is popular in building bots for services such as Slack, Telegram, or other messengers (basically, an app that can process commands coming from messages and respond to them);
- hybrids – using serverless alongside traditional systems. This is the most abstract and yet the most promising way of using nanoservices, because it can combine the benefits of each architecture, building each part of the system utilizing the most suitable architecture for it.

Vamsi Krishna Thatikonda offers four groups of use cases suitable for serverless computing, which in our opinion are fully covered by P. Sbarski and others: web

applications, data processing, chatbots and AI services, and ephemeral tasks (which are scheduled jobs) [38, p. 345].

As we can see, nanoservices can be beneficial where the load is uneven (IOT, real-time analytics, scheduled services), and the processing is simple (all of the above cases). At the same time, let's see scenarios where nanoservices don't fit. Ido Vapner separates such cases [74]:

- large code bases – it may not be practical to break a large code base down into small, single-purpose functions, meaning refactoring the whole system;
- high data requirements – each function is stateless, so data cannot be stored between executions, therefore processing large amounts of data can lead to increased complexity and reduced performance;
- complex interactions – as each function is designed to perform a specific task and does not have access to the state of other functions nanoservices architecture may not be the best fit for such systems.

We agree with these cases, but would also like to add some scenarios where nanoservices would not bring as much value as they could:

- systems with even load, without demands spikes – nanoservices in combination with a serverless approach are good at autoscaling, but if the load on the system is even and predictable it reduces the need for this benefit;
- systems with constant high load – some researches show that nanoservices with the serverless approach are cost-effective only to some specific limit of load [75, 76]; after reaching that limit nanoservices become more expensive compared to other architectures based on permanent virtual machines;
- systems with intensive usage of transactions and high cost of failures (financial sector) – as every distributed system, nanoservices are hard at handling distributed transactions and guaranteeing data consistency. Preference for asynchronous communication in nanoservices leads to event-based patterns, where eventual consistency is a common case. That may be fine for multimedia and information-providing systems (like the "BBC Online"), but not acceptable for processing financial transactions [77].

One of the most interesting and debatable observations here is that systems with a constant high load are better off not using nanoservices, but rather being built with more traditional architectures such as microservices on containers or monoliths. This is the case that faced the tech team of Prime Video, which initially built a distributed serverless system, but then moved to a monolith on virtual machines, reducing costs by 90% [78]. Sr. Product Engineer at Amazon even claims that no company should use serverless for production loads [79]. We find these statements quite debatable and find that they need further investigations to either prove or refute them.

Summary

Nanoservices architecture, characterized by its benefits such as autoscaling, cost efficiency, and the ability to develop independently, is ideally suited for scenarios that demand scalability, handle spikes in processing, foster rapid application development, and manage user events. Use cases extending from IoT and real-time analytics to backend compute and data manipulation demonstrate the flexibility and efficiency of nanoservices, particularly in environments that experience uneven loads or require simple processing tasks. However, nanoservices may not be effective for systems with constant high loads, large codebases, or complex interactions due to challenges in managing distributed transactions and ensuring data consistency. This becomes evident in sectors with high failure costs, such as finance, where traditional architectures might be more cost-effective and reliable. Having revealed the use case scenarios and communication concepts and patterns for nanoservice architecture, in the next chapter we will demonstrate how these concepts and patterns are applied in practice.

CHAPTER 3: Implementation of a use case

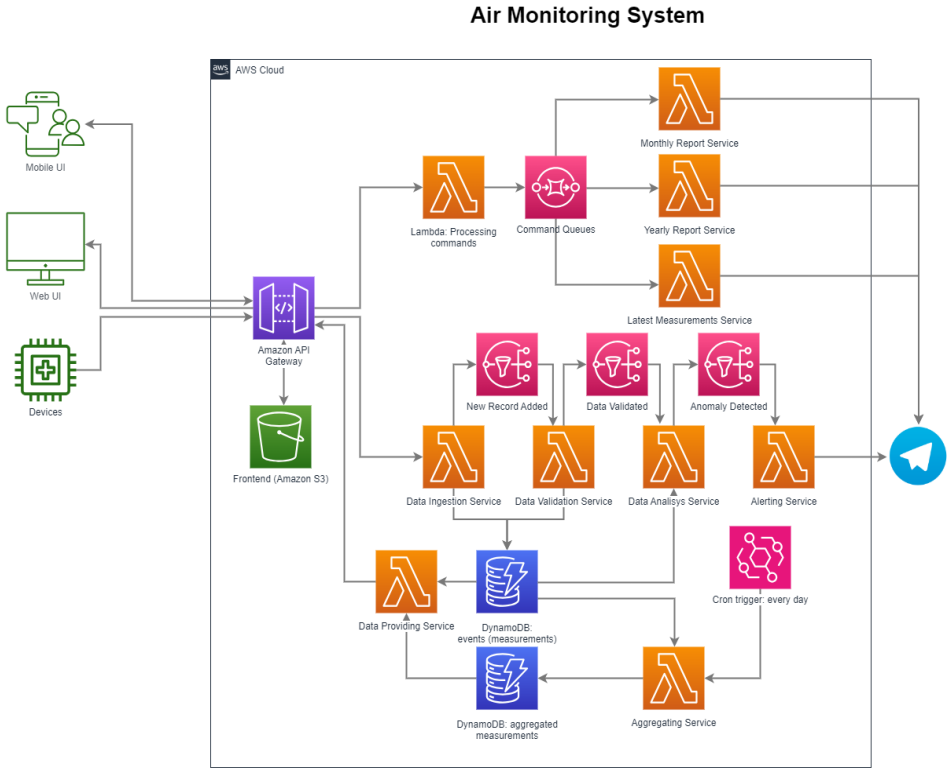
3.1 Designing the architecture of the system based on nanoservices

As we have elaborated in the previous chapters on what a nanoservice architecture is, what benefits and disadvantages it has, what basics of communication it should be based on, and what use cases suit it the best, let's implement a use case demonstrating the usage of nanoservices architecture according to these principles.

The system we are implementing is called the "Air Monitoring System", it is based on nanoservices, and its purpose is to collect air measurements from the hardware sensors, validate and analyze them, notify users in case of anomalies, and also provide aggregated data for different periods in various forms (as diagrams on a webpage and through a telegram chat). So, the goal of the system is to monitor air parameters and adjust them in a semi-automatic way (the user gets notified if something goes wrong – too cold or to hot for example, and can react to such notifications).

The architecture of the system is provided in Figure 11.

Figure 11. Air Monitoring System architecture schema



According to the schema, the system consists of the following services:

- Data Ingestion – service responsible for retrieving data from the hardware sensors and storing it persistently;
- Data Validation – service responsible for verifying the incoming data is valid (satisfies some requirements) and consistent;
- Data Analysis – service responsible for analising data for different anomalies (like reaching absolute limits – minimal/maximal temperature/humidity, or relative limits – abrupt change of the temperature);
- Alerting – service responsible for sending alerts when any anomalies are detected;
- Aggregation – service responsible for aggregation measurements of each day (~140 records per day) into a single record with average values (this is needed for providing data for comparatively big periods of time – more than several days);
- Data Providing – service responsible for serving measurements data, consumed by frontend application to display data charts;
- Command Processing – service responsible for processing commands incoming from telegram bot and resolving command executor;
- Monthly Reporting – service responsible for building a message with monthly report of measurement changes (command executor);
- Yearly Reporting – service responsible for building a message with yearly report of measurement changes (command executor);
- Latest Measurements – service responsible for building a message with latest measurements (command executor).

The system was implemented using AWS cloud provider. AWS is a powerful and flexible platform for developing and deploying a nanoservice-based system, providing multiple serverless solutions (file storages, databases, compute services etc), and according to the latest reports it’s the most popular among other cloud providers (the market share of AWS as of Q1 2023 is 32%, Microsoft Azure – 23%, Google Cloud

Platform – 10%) [80]. Also, it's worth mentioning, that AWS offers a significant amount of free tier products, allowing to build and run systems at no or very low cost [81].

For implementing the system, the following cloud services were used:

- API Gateway – a fully managed service for creating, publishing, maintaining, monitoring, and securing APIs; in the system it's used for building RESTful API for accessing measurements data and posting telegram commands;
- Lambda – a serverless, event-driven compute service that lets run code for backend service without provisioning or managing servers; this is a core and most used service for our system, as each nanoservice is run as a serverless Lambda function;
- DynamoDB – a serverless, NoSQL database service, used for storing gathered measurements data;
- Simple Storage Service (S3) – an object storage service for storing and serving frontend application files;
- EventBridge – a service for building build point-to-point integrations and a serverless scheduler; in our system it's used for triggering Aggregation service by schedule (the cron rule);
- Simple Notification Service (SNS) – a fully managed Pub/Sub service; used for organizing event-driven communication between services in our system;
- Simple Queue Service (SQS) – a fully managed serverless message queuing service, that enables to decouple and scale serverless (and other) applications; in our system we are using SQS queues for queueing commands, which need to be processed by executor services;
- Systems Manager – for our system we are using Parameter Store of Systems Manager – service for securely storing and managing configuration data and secrets; we are using it for storing confidential telegram token, needed to publish messages on behalf of our telegram bot.

In figures 12-15 there are some of the resources used for building our system.

Figure 12. AWS API Gateway RESTful API

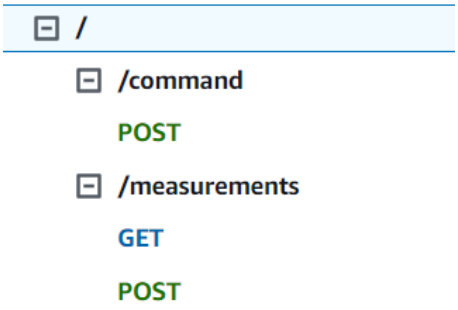


Figure 13. AWS Lambda services

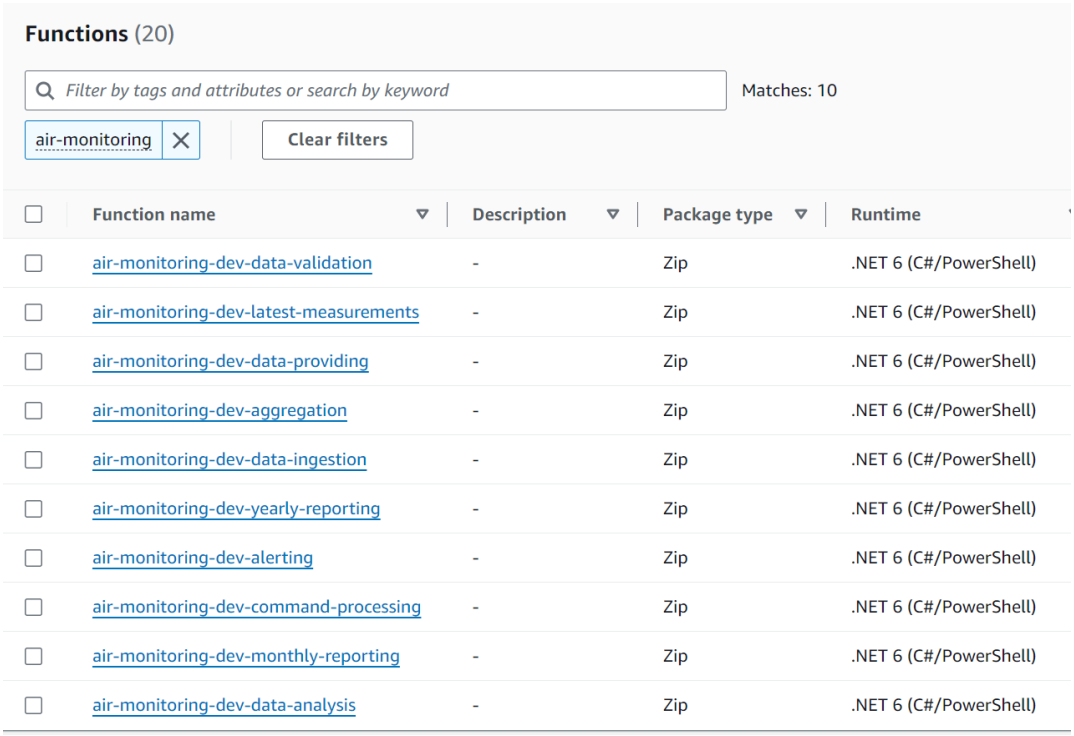


Figure 14. AWS SNS topics

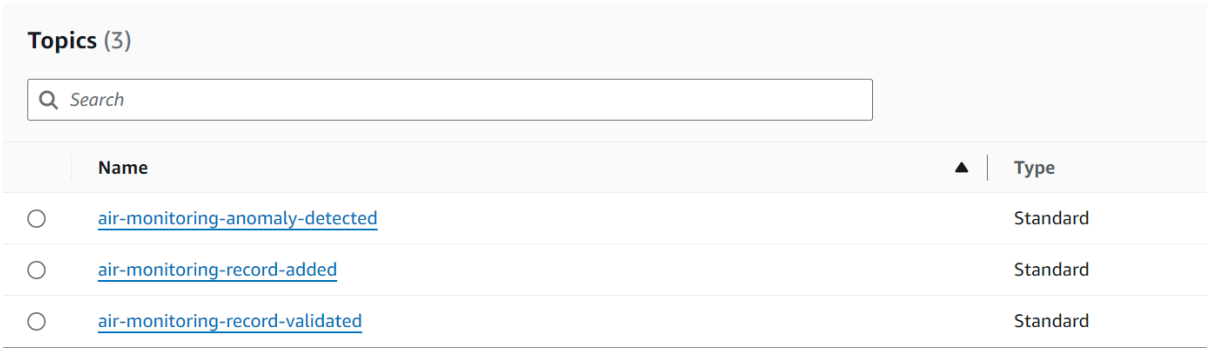
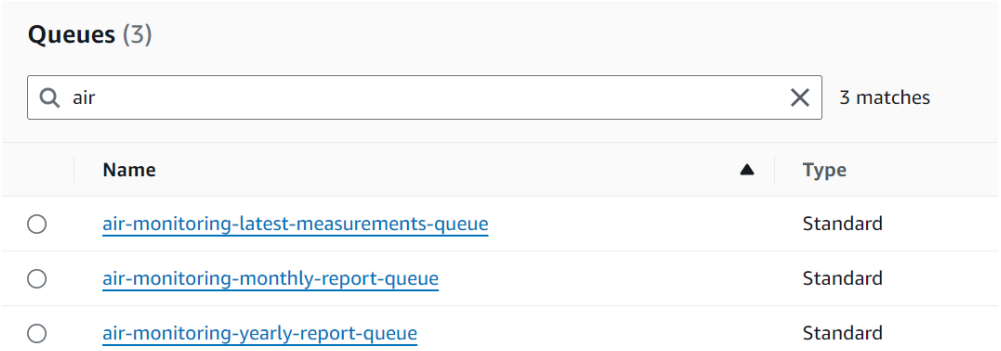


Figure 15. AWS SQS queues



In the previous chapter we have reviewed different patterns of communication, which could be applied to nanoservices architecture, and in our system we have implemented the following:

- Request/Response pattern: frontend application makes HTTP GET calls to /measurements endpoint and waits for a response from the Data Providing service;
- Event-Driven Communication Pattern: all communication between services is done in asynchronous way, where each service does not query others directly, but rather publishes events to SNS topics or SQS queues, and other subscribed services react to these events;
- Command Pattern: in our architecture the Command Processing service retrieves commands from the telegram bot and publishes them to the appropriate queue, after which subscribed service executes the command; three types of commands are supported (monthly report, yearly report and latest measurements), and there is a separate service per each command type.

By leveraging events to trigger actions in different services, the Air Monitoring system exemplifies an event-driven architecture. This approach allows for highly decoupled, scalable, and maintainable systems, where each component reacts to and processes events asynchronously. Additionally, the system is built in choreography style, where there are no central coordinator (orchestrator), each service either consumes or produces events, or both, which makes the system even more decoupled, and each service – more independent and resilient.

3.2 Implementation of the system

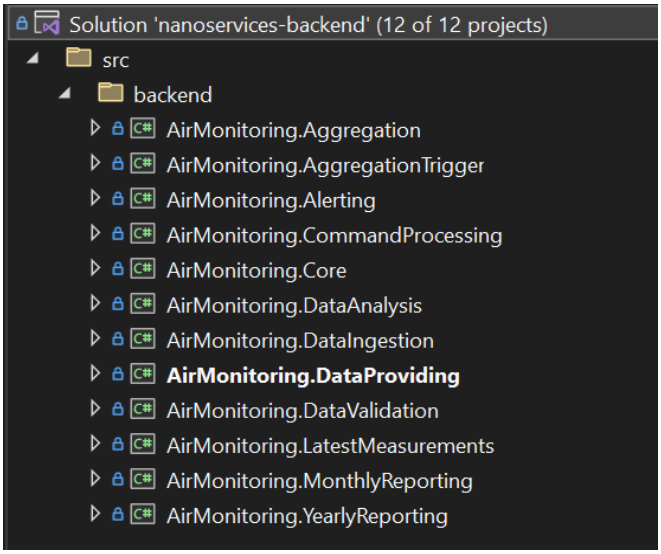
The project repository is available at GitLab by the link – https://gitlab.com/oleksandr.barsuk/nanoservices-diploma-project.git. It has the following structure:

- src:
  - backend – folder containing the code of all of the nanoservices;
  - firmware – folder containing the code of the sensors, that measure air parameters and send them to the system;
  - frontend – folder containing frontend application to present the gathered data;
- .gitignore – rules defining which files need to be excluded from the repository;
- README.md – document, providing general information on how to build and run the system.

The Backend

The backend part of the system is written in C# for .NET platform. The solution structure is shown in the Figure 16.

Figure 16. Backend solution structure



Almost each project here represents a nanoservice (except for the AirMonitoring.Core, which provides some shared classes – model, data repositories, configurations and etc), which performs exact one operation (for example, saving measurements to the database, or validating the data, or analysing the data for anomalies and so on). Also, each service is deployed as AWS Lambda, and for this reason it has specific structure and additional dependencies:

– each project has linked packages “Amazon.Lambda.Core”, “Amazon.Lambda.Annotations”, “Amazon.Lambda.APIGatewayEvents” and some other depending on the service needs – these packages provide support for Amazon Lambda .NET Core;

– each project should have a class with a method, which will be configured to be called when the Lambda is triggered. In our project we use the same structure in each service – it’s a class “Function” with async method “FunctionHandler”. It takes two parameters – trigger event (can be of different types depending which trigger event Lambda is expecting) and Lambda context (object containing environment information, logger for Lambda function). The method should return either void result (for synchronous implementations), Task (for asynchronous implementations) or a typed Task (for example – Task<APIGatewayProxyResponse> in case of returning the response for API Gateway).

System events

The system “Air Monitoring” is event-driven, meaning that each service either produces the event, or consumes an event, or does both. The following events are used in the system to communicate between services:

- New Record Event – is produced by Data Ingestion service after processing data from the sensors. The event is published on the topic “air-monitoring-record-added” in Amazon SNS. The structure of the event is the following:

Figure 17. New Record Event structure

```
public class NewRecordEvent
{
    public string DeviceId { get; set; } = string.Empty;

    public string Date { get; set; }
}
```

- Validation Event – is produced by the Data Validation service after measurement data validation and means that data validation passes successfully and can be processed further; in the other case, the data gets removed from the database. This event is published to the "air-monitoring-record-validated" topic, and it has the same structure as the New Record Event;
- Anomaly Event – is produced by the Data Analysis service when any anomaly is detected and published to the "air-monitoring-anomaly-detected" topic; the event has the structure according to Figure 18:

Figure 18. Anomaly Event structure

```
public class AnomalyEvent
{
    public string DeviceId { get; set; } = string.Empty;

    public string Date { get; set; }

    public AnomalyType Type { get; set; }

    public string Message { get; set; }
}
```

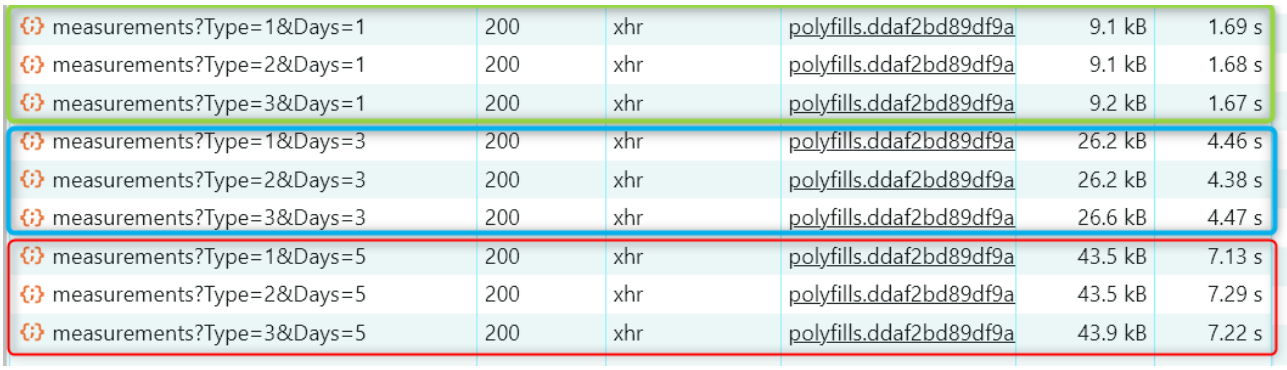
Actually, 5 types of anomalies are supported: Sudden Cooling (temperature change in more than 10°C), Sudden Warming (same), Overcooling (less than 10°C), Overheating (more than 40°C), and Overdrying (less than 20% humidity).

These events are published to Amazon SNS service, allowing to subscribe to each event as many services as needed. It means that each single event can be consumed by multiple services – every service that is subscribed to the topic will receive the event. On the contrary, for processing telegram commands Amazon queues are utilized (SQS service), where each message is processed by strictly one service.

Data aggregation

The implemented and built system produces ~140 records with all measurements per day (one hardware device with sensors sends measurements to the system every 10 minutes, which gives 6 records per hour or 6*24=144 records per day). In the real-world scenario processing of records for 1 day period requires ~1.6-2 seconds, for 3 days – 4.4-6 seconds, and for 5 days – 7-12 seconds (results in Figure 19, where the green group of requests is for 1 day, blue – for 3 days, and red – for 5 days).

Figure 19. Duration of request processing for 1, 3, and 5 days periods



This means that retrieving data over longer periods would result in significantly increased latencies, which is unacceptable from a user experience perspective (e.g., if processing one day takes ~2 seconds, then processing one month would take ~60 seconds). To address this issue, an aggregation mechanism was implemented to allow for processing and loading less data over extended periods.

To achieve this, an Aggregation Service was created. This service is triggered by schedule daily. It collects all records for a one-day period, calculates the average values for each measurement, and generates a single record per day with these aggregated values. And this aggregated record is saved to a separate database (“AirMonitoring.MeasurementsAggregated” table of the DynamoDB). As a result, instead of loading ~51k records for building a one-year diagram, with the new approach only 365 records are needed, which are loaded for ~2-4 seconds (Figure 20).

Figure 20. Duration of request processing for a 1-year period with aggregation

measurements?Type=1&Days=365	200	xhr	polyfills.ddaf2bd89df9a	19.1 kB	2.82 s
measurements?Type=2&Days=365	200	xhr	polyfills.ddaf2bd89df9a	19.1 kB	3.48 s
measurements?Type=3&Days=365	200	xhr	polyfills.ddaf2bd89df9a	19.2 kB	3.50 s

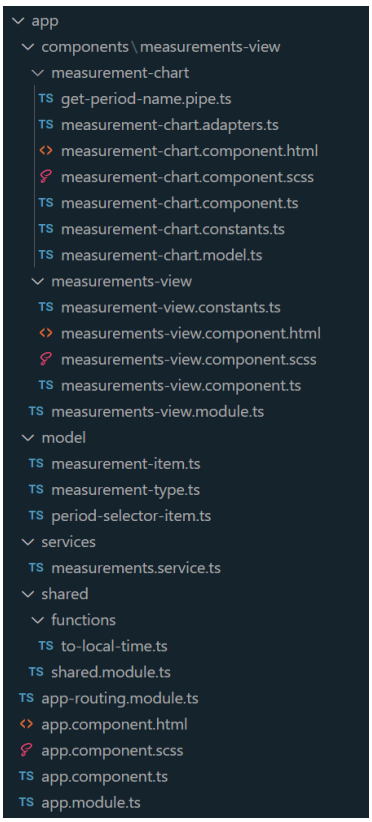
Adding the Aggregation nanoservice to the Air Monitoring system not only increased performance and enhanced user experience by reducing latency in request processing but also potentially decreased expenses on cloud services. In AWS Lambda pricing, the operating time of functions is charged; the less time functions operate, the lower the expenses.

The Frontend

The frontend part of the project is built as a single-page application (SPA), using Angular version 16. Additionally, “Bootstrap” package is used for styling, and “Apexcharts” package is used for drawing the charts.

The frontend project structure is shown in the Figure 17.

Figure 19. Frontend project structure



The main (and only) page of the application is called “Measurements-View”, and it displays inside of it three instances of “Measurement Chart Component” – each per measurement type. For each chart the data is loaded asynchronously and independently from others (appendices A and B show frontend application view during and after loading of the data). Also, periods for displaying charts can selected from 1 day up to 1 year.

The Firmware

As the firmware, the Arduino sketch is used. This Arduino sketch is designed to read temperature, humidity, and pressure data from two sensors (Adafruit BMP085 and SHT31) and send the data to an AWS API endpoint over Wi-Fi using an ESP8266 module.

The sketch includes libraries for BMP085, SHT31, Wi-Fi, HTTP client, and JSON handling. It initializes the BMP085 and SHT31 sensors and connects to the specified Wi-Fi network using SSID and password.

Main Loop:

Data Collection: Reads temperature, humidity, and pressure from the sensors. Prints the data to the serial monitor.

Data Transmission: Sends the collected data to a specified AWS API endpoint using HTTPS. Handles Wi-Fi connection status and restarts Wi-Fi if necessary.

The sketch reads environmental data from sensors and sends it to the API of the Air Monitoring System (POST “/measurements” endpoint) for further processing and storage. It includes handling of sensor initialization, Wi-Fi connectivity, and data transmission using HTTPS.

3.3 Deployment of the system

For the deployment of frontend and backend parts of the Air Monitoring System, the Serverless Framework was chosen. It allows easy deploy serverless architectures based on Lambda functions to the AWS cloud. Also, it supports multiple languages

and can deploy not only functions but some other resources as well (S3 buckets, SNS topics, etc) [82].

Deployment of the frontend and the backend can be performed independently, therefore there are separate serverless configs for each one.

The backend serverless.yml configuration defines a serverless application named “air-monitoring” deployed on AWS using .NET 6. The application includes Lambda functions, each one for a separate nanoservice. Each function is packaged individually and triggered by various events, such as HTTP requests, SNS topics, SQS queues, and scheduled cron jobs. The IAM role statements grant necessary permissions for DynamoDB, SNS, SQS, and SSM Parameter Store operations, ensuring the functions can interact with these AWS services securely and effectively.

The frontend serverless.yml configuration defines a serverless application named “air-monitoring-front” deployed on AWS using Node.js 18.x. The application uses the serverless-s3-sync plugin to synchronize a local directory (dist/air-monitoring/) with an S3 bucket named “air.monitoring”. This setup is used for deploying the frontend application of the Air Monitoring System to AWS S3, enabling efficient hosting and delivery of static content.

Summary

The implementation of the Air Monitoring system demonstrates the practical application of nanoservice architecture within a serverless computing framework. By leveraging AWS services such as API Gateway, Lambda, DynamoDB, SNS, and SQS, the system showcases how nanoservices can be effectively used to create a scalable, resilient, and maintainable solution for real-time data collection, validation, analysis, and user notification.

The architecture’s design emphasizes event-driven communication and decoupled services, ensuring that each component operates independently and asynchronously. This results in a robust system capable of handling high-frequency data inputs while maintaining responsiveness and low latency. The aggregation service, which consolidates daily measurements into single records, significantly enhances

performance, especially for long-term data retrieval, demonstrating a crucial optimization for user experience and cost efficiency.

Overall, the Air Monitoring system illustrates the strengths of nanoservice architecture in building modern cloud-based applications, highlighting the benefits of scalability, modularity, and efficient resource utilization. This implementation serves as a practical example for future projects aiming to leverage nanoservices and serverless technologies.

CONCLUSIONS AND RECOMMENDATIONS FOR THE FURTHER RESEARCH

This thesis has explored the theoretical foundations, practical implementations, and specific use cases of nanoservice architecture. By delving into the definitions, benefits, and drawbacks of nanoservices, the study highlights their unique characteristics and suitability for modern software development, particularly in the context of cloud computing and serverless environments.

The research emphasizes the importance of asynchronous communication and event-driven architecture in nanoservice systems. Key patterns such as Event-Driven Communication, Command Pattern, and the Request-Reply Pattern are identified as effective for achieving decoupled and scalable systems. The study illustrates how these patterns facilitate independent service operation, enhance system responsiveness, and reduce latency, crucial for maintaining performance in highly distributed environments.

Nanoservice architecture is best suited for scenarios requiring high scalability, handling of sporadic spikes in processing demands, and rapid application development. Ideal use cases include Internet of Things (IoT) applications, real-time data analytics, backend compute for web applications, and serverless functions for ephemeral tasks. The architecture's ability to independently scale small, focused services makes it particularly advantageous in these contexts, providing both cost efficiency and improved resource utilization.

Future research should focus on optimizing communication patterns in nanoservice architectures, particularly in minimizing the latency and resource overhead associated with interservice communication. Investigating advanced orchestration and choreography techniques can further enhance the efficiency and reliability of nanoservice systems. Additionally, exploring the integration of nanoservices with emerging technologies such as edge computing and AI-driven automation could unlock new potentials and use cases.

Further studies could also assess the long-term cost implications of nanoservice deployments in various cloud environments, providing more detailed guidelines for selecting appropriate architectural patterns based on specific application requirements and operational contexts.

In summary, nanoservice architecture presents a robust framework for building scalable, maintainable, and cost-efficient software systems. By adopting event-driven communication patterns and focusing on use cases that leverage the strengths of nanoservices, developers can create highly responsive and resilient applications. Continuous research and optimization will ensure that nanoservice architecture remains a viable and powerful approach in the evolving landscape of software development.

BIBLIOGRAPHY

1. IEEE Recommended Practice for Architectural Description for Software-Intensive Systems. [Electronic resource] : IEEE Std 1471-2000. – Available at: https://ieeexplore.ieee.org/document/875998.
2. Perry, D., & Wolf, A. *Foundations for the study of software architecture* // ACM SIGSOFT Software Engineering Notes. – 1992. – Vol. 17, № 4. – P. 40-52.
3. Garlan, D., & Shaw, M. *An introduction to software architecture* // Proceedings of the Advances in software engineering and knowledge engineering. – 1993. – Vol. 1. – P. 1-40.
4. Hayes-Roth, F. *Architecture-based acquisition and development of software: Guidelines and recommendations from the ARPA domain-specific software architecture (DSSA) program*. – Teknowledge Federal Systems, 1994. – Version 1.
5. Bass, Len. Software architecture in practice / Len Bass, Paul Clements, Rick Kazman. – 2rd ed. – MA : Addison-Wesley, 2003. – 528 p. – ISBN 0-321-15495-9.
6. Goel, A. *The Philosophy of Software Architecture*. RMIT University, Australia. [Electronic resource] – 2010 – Available at: https://www.igi-global.com/gateway/article/49564
7. International Standard ISO/IEC/IEEE 42010. Systems and Software Engineering – Architecture Description. Washington, DC: IEEE. [Electronic resource] – 2011 – Available at: https://cwe.ccsds.org/sea/docs/SEA-SA/Draft%20Documents/RASDS%20-%20Systems%20Architecture%20Background%20Materials/ISO-IEC-IEEE-42010-2011.pdf
8. Brown, S. *Software Architecture for Developers*. Leanpub. [Electronic resource] – 2014 – Available at: https://static.codingthearchitecture.com/sddconf2014-software-architecture-for-developers-extract.pdf
9. Bass Len. *Software architecture in practice* / Len Bass, Paul Clements, Rick Kazman. – 3rd ed. – MA : Addison-Wesley, 2015. – 589 p. – ISBN 978-0-321-81573-6.
10. Wolff, E. *Microservices: Flexible Software Architecture*. Addison-Wesley Professional, 2016. – 395 p. – ISBN 978-0-134-60241-7.
11. Wang, Xinwen, Tiancheng Yuan, Yu-Ju Huang and Robbert van Renesse. *Disaggregated Applications Using Nanoservices*. [Electronic resource] – 2021 – Available at: https://www.semanticscholar.org/paper/Disaggregated-

Applications-Using-Nanoservices-Wang/22650f123c88a317b3ec285ca6bf560ea86a95cc

12. Kratzke Nane. *A Brief History of Cloud Application Architectures*. Applied Sciences 8, no. 8: 1368. [Electronic resource] – 2018 – Available at: https://doi.org/10.3390/app8081368
13. Harjula, Erkki & Karhula, Pekka & Islam, Johirul & Leppänen, Teemu & Manzoor, Ahsan & Liyanage, Madhusanka & Jagmohan, Chauhan & Kumar, Tanesh & Ahmad, Ijaz & Ylianttila, Mika. *Decentralized IoT Edge Nanoservice Architecture for Future Gadget-Free Computing*. IEEE Access. PP. 10.1109/ACCESS.2019.2936714. [Electronic resource] – 2019 – Available at: https://www.researchgate.net/publication/334696214_Decentralized_IoT_Edge_Nanoservice_Architecture_for_Future_Gadget-Free_Computing
14. Islam, Johirul & Harjula, Erkki & Kumar, Tanesh & Karhula, Pekka & Ylianttila, Mika. "*Docker Enabled Virtualized Nanoservices for Local IoT Edge Networks*". 10.1109/CSCN.2019.8931321. [Electronic resource] – 2019 – Available at: https://www.researchgate.net/publication/335723922_Docker_Enabled_Virtualized_Nanoservices_for_Local_IoT_Edge_Networks
15. D. D. Sánchez-Gallegos et al. *On the Continuous Processing of Health Data in Edge-Fog-Cloud Computing by Using Micro/Nanoservice Composition*, in IEEE Access, vol. 8, pp. 120255-120281, 2020, doi: 10.1109/ACCESS.2020.3006037. [Electronic resource] – 2020 – Available at: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9129708
16. Nasser Yaman. *From Monolithic to Nanoservice Architecture*. [Electronic resource] – 2022 – Available at: https://medium.com/nerd-for-tech/from-monolithic-to-nanoservice-architecture-d01f35416f18
17. Ebi Chuka. *The rise of nanoservices*. [Electronic resource] – 2020 – Available at: https://increment.com/software-architecture/the-rise-of-nanoservices/
18. Kanjilal Joydip. *Nanoservices: Where they fit – and where they don't*. [Electronic resource] – Available at: https://techbeacon.com/app-dev-testing/nanoservices-where-they-fit-where-they-dont
19. Roberts Mike. *Serverless Architectures*. [Electronic resource] – 2018 – Available at: https://martinfowler.com/articles/serverless.html
20. Netflix Technology Blog. *Developer Experience Lessons Operating a Serverless-like Platform At Netflix*. [Electronic resource] – 2017 – Available at: https://netflixtechblog.com/developer-experience-lessons-operating-a-serverless-like-platform-at-netflix-a8bbd5b899a0

21. Harris Chandler. *Microservices vs. Monolithic Architecture. When monoliths grow too big it may be time to transition to microservices*. Atlassian. [Electronic resource] – 2023 – Available at: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith
22. Al-Debagy Omar and Peter Martinek. *A Comparative Review of Microservices and Monolithic Architectures*. 2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI), pp. 149-154. [Electronic resource] – 2018 – Available at: https://ieeexplore.ieee.org/abstract/document/8928192/authors#authors
23. Dragoni Nicola & Giallorenzo, Saverio & Lluch-Lafuente, Alberto & Mazzara, Manuel & Montesi, Fabrizio & Mustafin, Ruslan & Safina, Larisa. *Microservices: yesterday, today, and tomorrow*. [Electronic resource] – 2017 – Available at: https://www.researchgate.net/publication/319809514_Microservices_yesterday_today_and_tomorrow
24. Papazoglou Mike. *Service-Oriented Computing: Concepts, Characteristics and Directions*. Proceedings of the Fourth International Conference on Web Information Systems Engineering. 10.1109/WISE.2003.1254461. [Electronic resource] – 2003 – Available at: https://www.researchgate.net/publication/262308599_Service_-Oriented_Computing_Concepts_Characteristics_and_Directions
25. Channabasavaiah K., Holley K., Tuggle E. *Migrating to a service-oriented architecture*. IBM DevelopWorks. 16:727-728. [Electronic resource] – 2004 – Available at: https://public.dhe.ibm.com/software/info/openenvironment/G224-7298-00_Final.pdf
26. Bianco P, Kotermanski R, Merson PF. *Evaluating a Service-Oriented Architecture*. Pennsylvania, United States of America: Carnegie Mellon University. CMU/SEI-2007-TR-015. DOI: 10.1184/R1/6573479.v1. [Electronic resource] – 2007 – Available at: https://insights.sei.cmu.edu/library/evaluating-a-service-oriented-architecture/
27. Krafzig D. & Banke, K & Slama, Dirk. Enterprise SOA – Service-Oriented Architecture Best Practices. Prentice Hall PTR. – 382 p. [Electronic resource] – 2005 – Available at: https://books.google.co.uk/books?id=R7oGhITYUuUC
28. Fremantle Paul, Sanjiva Weerawarana, and Rania Khalaf. *Enterprise Services*. Communications of the ACM (CACM), Volume 45, Issue 10, October 2002, pp. 77-82. [Electronic resource] – 2002 – Available at: http://doi.acm.org/10.1145/570907.570935

29. Sleeper B., Robins, B. *The laws of evolution: A pragmatic analysis of the emerging web services market*. San Francisco: The Stencil Group. [Electronic resource] – 2002 – Available at: http://www.stencilgroup.com/ideas_scope_200204evolution.pdf
30. Mehta Mayur & Lee, Sam & Shah, Jaymeen. *Service-Oriented Architecture: Concepts and Implementation*. [Electronic resource] – 2006 – Available at: https://www.researchgate.net/publication/237776180_Service-Oriented_Architecture_Concepts_and_Implementation
31. Richardson Chris. *Microservices Patterns (With examples in Java)*. NY : Manning Publications Co., 2019. – 522 p. – ISBN: 978-161-729-454-9.
32. Luthria Haresh, Rabhi Fethi. *Service Oriented Computing in Practice – An Agenda for Research into the Factors Influencing the Organizational Adoption of Service Oriented Architectures* // Journal of Theoretical and Applied Electronic Commerce Research. – 2009. – Vol. 4, Issue 1, April. – 39-56 pp.
33. MacKenzie M.C. et al. *Reference model for service oriented architecture 1.0*. OASIS Standard, 12. [Electronic resource] – 2006 – Available at: https://angeldeacero.wdfiles.com/local--files/start/oasissoa.pdf
34. Bucchiarone Antonio, Kemal Soysal, Claudio Guidi. *Automatic Migration to Microservice: A Model-Driven Approach*. [Electronic resource] – 2020 – Available at: https://www.conf-micro.services/2020/papers/paper_8.pdf
35. Fowler M., Lewis J. *Microservices: a definition of this new architectural term*. [Electronic resource] – 2014 – Available at: http://martinfowler.com/articles/microservices.html
36. Rademacher F., S. Sachweh and A. Zündorf. *Aspect-Oriented Modeling of Technology Heterogeneity in Microservice Architecture*. 2019 IEEE International Conference on Software Architecture (ICSA), Hamburg, Germany, pp. 21-30, doi: 10.1109/ICSA.2019.00011. [Electronic resource] – 2019 – Available at: https://ieeexplore.ieee.org/document/8703913
37. Gilbert S., Lynch N. *Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services*. [Electronic resource] – 2002 – Available at: https://users.ece.cmu.edu/~adrian/731-sp04/readings/GL-cap.pdf
38. Thatikonda, Vamsi. *Serverless Computing: Advantages, Limitations and Use Cases*. European Journal of Theoretical and Applied Sciences. 1. 341-347. 10.59324/ejtas.2023.1(5).25. [Electronic resource] – 2023 – Available at: https://www.researchgate.net/publication/374097091_Serverless_Computing_Advantages_Limitations_and_Use_Cases
39. Maden Deniz. *Serverless Architecture & AWS Lambda*. [Electronic resource] – 2022 – Available at: https://medium.com/adessoturkey/serverless-architecture-aws-lambda-d964e84365e4

40. Dolynyuk T. *Serverless Architecture: When to Use This Approach and What Benefits It Gives*. [Electronic resource] – 2023 – Available at: https://apiko.com/blog/serverless-architecture-benefits/
41. AWS Documentation. *Building Applications with Serverless Architectures*. [Electronic resource] – Available at: https://aws.amazon.com/lambda/serverless-architectures-learn-more/
42. Google Cloud Documentation. *What is Serverless Architecture?* [Electronic resource] – Available at: https://cloud.google.com/discover/what-is-serverless-architecture
43. Castro, Paul & Isahagian, Vatche & Muthusamy, Vinod & Slominski, Aleksander. *The rise of serverless computing*. Communications of the ACM. 62. 44-54. 10.1145/3368454. [Electronic resource] – 2019 – Available at: https://www.researchgate.net/publication/337429660_The_rise_of_serverless_computing
44. Sbarski P. *Serverless Architectures on AWS*. NY : Manning Publications Co., 2022. – 257 p. – ISBN 9781617295423.
45. Qu, C., Calheiros, R. N., & Buyya, R. Auto-scaling Web Applications in Clouds: A Taxonomy and Survey. ArXiv. /abs/1609.09224. [Electronic resource] – 2016 – Available at: https://arxiv.org/abs/1609.09224
46. Kubernetes Documentation. *Horizontal Pod Autoscaling*. [Electronic resource] – Available at: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
47. Cerami E. *Do Azure Functions Actually Scale?* [Electronic resource] – 2021 – Available at: https://medium.com/fastapi-tutorials/do-azure-functions-actually-scale-2a34648dac8d
48. AWS Documentation. *Lambda function scaling*. [Electronic resource] – Available at: https://docs.aws.amazon.com/lambda/latest/dg/lambda-concurrency.html
49. Google Cloud Documentation. *Configure maximum instances*. [Electronic resource] – Available at: https://cloud.google.com/functions/docs/configuring/max-instances
50. Silveyra J.C. *Automate the deployment of your AWS Lambda functions with GitLab CI/CD pipelines*. [Electronic resource] – 2022 – Available at: https://medium.com/@juca_silveyra/automate-the-deployment-of-your-aws-lambda-code-and-layers-with-gitlab-ci-cd-pipelines-4778219fbdbd
51. Wood J. *Using GitHub Actions to deploy serverless applications*. AWS Compute Blog. [Electronic resource] – 2021 – Available at: https://aws.amazon.com/blogs/compute/using-github-actions-to-deploy-serverless-applications/

52. AWS Documentation. *Lambda runtimes*. [Electronic resource] – Available at: https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html
53. Microsoft Azure Documentation. *Azure Functions runtime versions overview*. [Electronic resource] – 2024 – Available at: https://learn.microsoft.com/en-us/azure/azure-functions/functions-versions?tabs=isolated-process%2Cv4&pivots=programming-language-csharp#languages
54. Google Cloud Documentation. *Cloud Functions. Runtime Support*. [Electronic resource] – Available at: https://cloud.google.com/functions/docs/runtime-support
55. OpenFaaS. [Electronic resource] – Available at: https://www.openfaas.com/
56. Serverless Framework Documentation. [Electronic resource] – Available at: https://www.serverless.com/framework/docs
57. Vahidinia, P., Farahani, B. & Aliee, F.S. *Cold start in Serverless Computing: Current trends and mitigation strategies*. In 2020 International Conference on Omni-layer Intelligent Systems (COINS). [Electronic resource] – 2020 – Available at: https://doi.org/10.1109/coins49042.2020.9191377
58. Marin, E., Perino, D. & Di Pietro, R. *Serverless Computing: A security perspective*. Journal of Cloud Computing, 11(1). [Electronic resource] – 2022 – Available at: https://doi.org/10.1186/s13677-022-00347-w
59. Tighilt Rafik et al. *On the study of microservices antipatterns: A catalog proposal*. In Proceedings of the European Conference on Pattern Languages of Programs 2020, pages 1–13, 2020. [Electronic resource] – 2020 – Available at: https://www.researchgate.net/publication/347707676_On_the_Study_of_Microservices_Antipatterns_a_Catalog_Proposal
60. Gall R. *Microservices, Nanoservices, Teraservices, and Serverless*. [Electronic resource] – 2019 – Available at: https://dzone.com/articles/microservices-nanoservices-teraservices-and-server-1
61. Richards M. *Software Architecture Patterns*. CA : O'Reilly Media, 2015. – 55 p. – ISBN 978-1-491-92424-2.
62. AWS Documentation. *Synchronous waiting within a single Lambda function.* [Electronic resource] – Available at: https://docs.aws.amazon.com/lambda/latest/operatorguide/synchronous-waiting.html
63. Kratzke, Nane & Siegfried, Robert. *Towards cloud-native simulations - lessons learned from the front-line of cloud computing*. The Journal of Defense Modeling & Simulation. 18. 10.1177/1548512919895327. [Electronic resource] – 2020 – Available at: https://www.researchgate.net/publication/338548862_Towards_cloud-

native_simulations_-_lessons_learned_from_the_front-line_of_cloud_computing

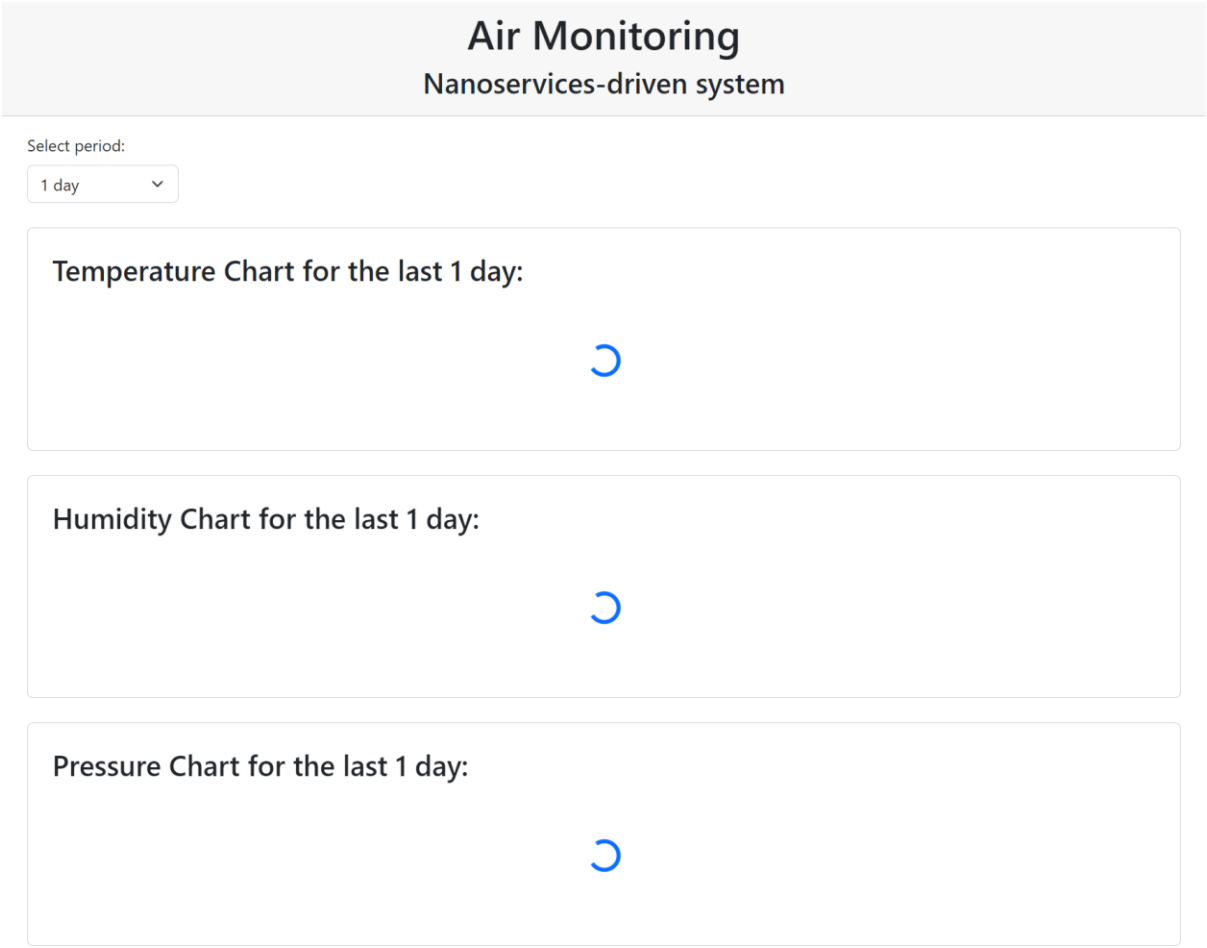
64. W3C. *Web Services Architecture*. W3C Working Group Note 11 February 2004. [Electronic resource] – 2004 – Available at: https://www.w3.org/TR/ws-arch/wsa.pdf
65. Computer Security Resource Center. *Universal Description, Discovery, and Integration (UDDI)*. [Electronic resource] – Available at: https://csrc.nist.gov/glossary/term/universal_description_discovery_and_integration
66. Richards M., Ford N. *Fundamentals Of Software Architecture. An Engineering Approach*. CA : O'Reilly Media, 2021. – 422 p. – ISBN 978-1-492-04345-4.
67. Indrasiri K. *Microservices in Practice – Key Architectural Concepts of an MSA*. WSO2, 07/2019. [Electronic resource] – Available at: https://wso2.com/whitepapers/microservices-in-practice-key-architectural-concepts-of-an-msa/
68. Wikipedia. Request–response. [Electronic resource] – Available at: https://en.wikipedia.org/wiki/Request%E2%80%93response
69. Bellemare A. *Building Event-Driven Microservices: Leveraging Organizational Data at Scale*. CA : O'Reilly Media, 2020. – 324 p. – ISBN 978-1-492-05789-5.
70. Microservice Architecture. *Pattern: Saga*. [Electronic resource] – Available at: https://microservices.io/patterns/data/saga.html
71. AWS Documentation. *Recursive patterns that cause run-away Lambda functions*. [Electronic resource] – Available at: https://docs.aws.amazon.com/lambda/latest/operatorguide/recursive-runaway.html
72. Kwon J. *What is Two Phase Commit in Distributed Transaction?* [Electronic resource] – 2022 – Available at: https://hongilkwon.medium.com/when-to-use-two-phase-commit-in-distributed-transaction-f1296b8c23fd#e66d
73. Two-Phase Commit. [Electronic resource] – Available at: https://martinfowler.com/articles/patterns-of-distributed-systems/two-phase-commit.html
74. Vapner I. *Unlocking the Power of Nano Services: A New Era in Microservices Architecture*. [Electronic resource] – 2023 – Available at: https://medium.com/@ido.vapner/unlocking-the-power-of-nano-services-a-new-era-in-microservices-architecture-22647ea36f22
75. Bettini A. *Should you use Serverless for your API?* [Electronic resource] – 2023 – Available at: https://medium.com/@aurelien.bettini/should-you-use-serverless-or-containers-cb28ce38632e

76. Egilsson E. *Serverless: 15% slower and 8x more expensive*. [Electronic resource] – 2019 – Available at: https://einaregilsson.com/serverless-15-percent-slower-and-eight-times-more-expensive/
77. Clark M. *Powering BBC Online with nanoservices*. Technology and Creativity at the BBC. [Electronic resource] – 2018 – Available at: https://www.bbc.co.uk/blogs/internet/entries/5bdabd53-090e-4611-a5d5-4faea05aeb35
78. Kolny M. *Scaling up the Prime Video audio/video monitoring service and reducing costs by 90%*. Prime Video Tech. [Electronic resource] – 2023 – Available at: https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90
79. Kanter Z. Post in X: TIL Amazon engineers think AWS serverless docs say serverless can't be used in production. [Electronic resource] – 2023 – Available at: https://twitter.com/zackkanter/status/1654105783498608641
80. John A.S. *Q1 cloud infrastructure spending surpasses $63B globally; Google, Microsoft & Amazon hold 65% marketshare*. Wire19. [Electronic resource] – 2023 – Available at: https://wire19.com/cloud-infrastructure-spending/
81. AWS Documentation. *AWS Free Tier*. [Electronic resource] – Available at: https://aws.amazon.com/free/?all-free-tier.sort-by=item.additionalFields.SortRank&all-free-tier.sort-order=asc&awsf.Free%20Tier%20Types=*all&awsf.Free%20Tier%20Categories=*all
82. Serverless Framework Documentation. *Serverless Framework Concepts*. [Electronic resource] – Available at: https://www.serverless.com/framework/docs/providers/aws/guide/intro

APPENDICES

Appendix A

Frontend application during loading of the data



Appendix B

Frontend application when the data is loaded