Tuijin Jishu/Journal of Propulsion Technology
ISSN: 1001-4055
Vol. 45 No. 2 (2024)

Visualization of Information System Architectures Based on Microservices

Kornienko D.V. Mishina S.V.
Bunin Yelets State University

Abstract: This article substantiates the importance and necessity of visualizing the architecture of information systems built on the principles of microservice architecture. This task is key both in the process of developing new systems and in optimizing the operation of existing ones. To provide complete and up-to-date information about the structure and interaction of microservices, an approach is required that is based on the automatic collection and processing of information about the relationships between microservices and their internal structure, followed by visualization in architecture diagrams. The study revealed that specialized software trace logs must be used to visualize the architecture. As part of the work, aspects of tracing were considered in the context of the OpenTelemetry project, an open-source tool designed for collecting telemetry data from programs and analyzing them. The author suggested using the C4 model as a visualization method. This approach is a relatively new concept for modeling software systems, including context, containers, components, code, and their relationships. The main goal of the research was to automate the process of creating diagrams based on data obtained from OpenTelemetry to simplify the understanding of the structure and interactions of microservices in the system. The result of the research was the development of algorithms for collecting and converting trace data to create C4 diagrams, as well as the development of software for their implementation.

Keywords*: automation, information systems, architecture, visualization, microservices, C4, OpenTelementry, Zipkin.*

1. Introduction

Microservices have gained great popularity in the development and construction of modern complex information systems [1]. They are an approach to software development in which a large application is broken down into small, independent modules or services, each of which performs a specific function and communicates with the others through simple interfaces, such as a RESTful API [2]. One of the key benefits of microservices is flexibility. Breaking the system into separate modules allows development teams to work on one service independently of the others, which speeds up the development process and simplifies scaling. Microservices also help increase system reliability, since an error in one service does not necessarily affect the operation of others. This can significantly reduce application downtime and make it more resilient to errors [3].

Flexible methodology, in particular Agile, is one of the main approaches in modern software development, including microservices [4]. The use of Agile in microservices development typically involves incremental and iterative approaches to create functionally independent system components. This allows teams to effectively adapt to changing business needs and technology trends. However, Agile also has its challenges. One of the most common problems is the lack of detailed and up-to-date documentation [4]. According to Agile principles, a working product is more important than detailed documentation. This means that Agile teams tend to focus on continuous value delivery and a working product rather than on creating detailed documentation. Most Agile documentation consists of task descriptions, system changes, and other work notes. This is incremental documentation and is created and updated as development progresses. Description of a complete system map is usually a labor-intensive task and is not always kept up to date. However, accurate and up-to-date system documentation is very important for a systematic understanding of the operation of the information system. Documentation helps analysts, developers and architects make timely and correct decisions about the development of the system, taking into account its technical features.

The most reliable information about the system is provided by the system itself, in the form of source code, binary files, system logs and trace logs. These artifacts can be used to reverse engineer the system [5, 6]. Through code analysis, logs, and traces [7], the structure and behavior of the system can be reconstructed, which in turn allows automatic generation of documentation. For example, you can automatically generate API descriptions, dependency diagrams, data diagrams, and other types of documentation. Additionally, tools can be compiled to visualize the microservice architecture.

For visualization of microservice architecture, trace systems are of greatest interest, in particular OpenTelemetry is a set of open-source tools designed for collecting, processing and exporting traces, metrics and logs [8]. This project was the result of the merger of two other projects, OpenTracing and OpenCensus, and is part of the Cloud Native Computing Foundation's cloud architecture. When it comes to microservice architectures, OpenTelemetry is especially valuable for its tracing capabilities. Splitting a system into many microservices can make it difficult to track the flow of requests between all of these services, as well as identify bottlenecks, performance issues, and other errors. OpenTelemetry allows you to trace request paths across all services, providing a complete final execution context [9]. The main applications for visualizing and analyzing OpenTelemetry traces are Zipkin and Jaeger [9]. The results of the study of these applications showed that these tools have limitations in terms of visualizing system components. Zipkin and Jaeger demonstrate the ability to present trace data in a useful and readable format, but they do not provide a comprehensive visualization of the system architecture. This shortcoming can lead to an incomplete view of integrations within a system, as well as make it difficult to identify complex interoperability issues between microservices. In addition, to visualize a microservice architecture, it is convenient to represent components and their relationships in the C4 model - a hierarchical system architecture visualization scheme that covers systems at other levels of complexity, offering contextual, container, component and class representation [10, 11]. Applying the C4 model to the visualization of OpenTelemetry trace data can improve the understanding of a system's architecture and make it easier to analyze the interactions between components.

The task of visualizing software architecture remains at the forefront of information technology and continues to become increasingly relevant in modern conditions. Visualization simplifies the process of perceiving and analyzing complex information, and helps both programmers and architects gain a better understanding of the structure and function of the system. In the work of Namiota D.E., Romanov V.Yu. “On 3D Visualization of Software Architecture and Metrics,” emphasizes the importance of this aspect [12]. The authors rightly point out the benefits of this approach, including speeding up the process of learning software and improving its understanding. Johann Mortara, Philippe Collet, Anne-Marie Dery-Pinna “Visualization of object-oriented software in a city metaphor: Comprehending the implemented variability and its technical debt” describes the use of visualization in a city metaphor to study the variability of object-oriented software [13]. In the work of Vyugina A. A. “Visualization of the work of client-server architecture using delegates in the C# language”, the tasks of visualizing the client-server architecture of applications are considered and it is noted that this approach will allow you to see which requests come to the server, and which requests are sent back to the client [14]. However, at the moment, the problem of visualizing the architecture of an information system based on microservices has not been completely solved. Modern approaches to obtaining and processing data on connections and the use of a modern convenient model for representing architecture, such as C4 diagrams, are required. The article discusses the option of building an architecture based on OpenTelemetry data, where Zipkin is used as a data collector. The process of obtaining and processing this data is described to visualize the architecture in the C4 model of the first and second levels of presentation - the context level and the container level.

2. Results and discussion

This article discusses the developed algorithm for visualizing the architecture of an information system based on microservices using OpenTelemetry, which can be divided into the following stages: 1. Preparing the information system infrastructure for collecting trace data, metrics and logs. This phase involves deploying and configuring data collection tools, such as OpenTelemetry agents, that will monitor the execution of microservices and collect trace information, metrics, and logs. 2. Uploading trace data. At this stage, the collected trace data, containing information about the interaction between various components of the information system, is retrieved from the warehouse and prepared for further processing. 3. Data processing and preparation. The collected trace data is processed and aggregated to create a structure that is easy to visualize. This stage includes data merging, filtering,

eliminating duplicates, and converting the data into a format that is used to build graphical visualizations. 4. Direct data visualization. At the last stage, the prepared data is visualized using suitable tools that allow you to create visual graphical representations of the architecture of the information system based on microservices. In Fig. Figure 1 presents the above-described stages of the algorithm for visualizing the architecture of an information system based on microservices using OpenTelemetry.

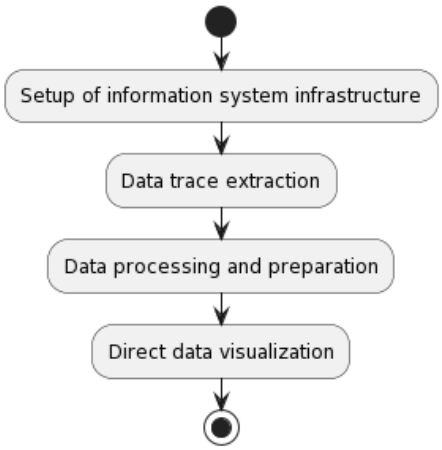

Fig. 1. Stages of an algorithm for visualizing the architecture of an information system based on microservices using OpenTelemetry

The infrastructure provisioning phase involves configuring and initializing relevant agents for each microservice. These agents intercept requests to other services or databases and, in the background, pass information about such requests to the appropriate data collection module. OpenTelemetry provides such agents for most modern programming languages. For example, a special instrumentation agent has been developed for the Java platform, which supports most Java frameworks, including Spring Boot. This agent introduces additional elements into the bytecode of classes when the application starts, allowing you to intercept calls to internal service functions and calls to external components, passing this data to pre-configured information collection modules. An alternative approach for setting up data collection agents is to use Istio Service Mesh in a Kubernetes infrastructure. Istio serves as a proxy for requests to other services, providing the ability to collect this information and subsequently transmit it to data collection modules in accordance with the OTPL protocol. It is proposed to use Zipkin as a collector for accumulating trace data. Zipkin is an easy-to-use yet feature-rich tool that offers support for a variety of data store types, such as MySQL, Apache Cassandra, and Elasticsearch. This expands the possibilities of its integration into various architectural solutions. A key feature of this tool is the availability of a tool for searching and viewing traces, which greatly facilitates the process of analyzing them. Zipkin also has an API for retrieving data that will be used for further processing in terms of architectural visualization. In Figure 2 shows one of the options for an infrastructure diagram using OpenTelemetry.

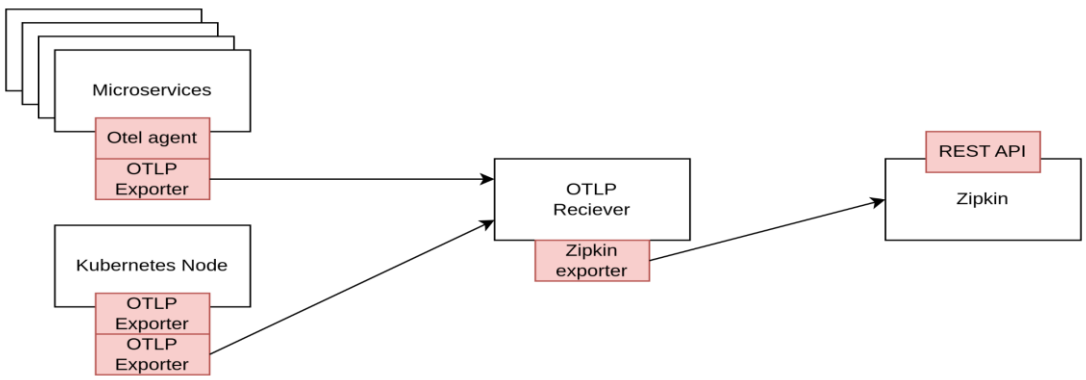

Fig. 2. Infrastructure diagram using OpenTelemetry

The Trace Data Upload phase is a process that involves setting a time period to collect trace information and downloading the data from the Zipkin system. It is obvious that it is critically important that the collected data reflect the actual state and behavior of the system as accurately and completely as possible. In this context, one of the key parameters is the depth of coverage of various system operating scenarios. To download data, it is proposed to use the REST API of the Zipkin service, namely the GET / traces method. The time interval is specified by the endTs and lookback parameters. The result of this method is expressed in the form of a two-dimensional array in a JSON structure, where each element corresponds to a specific trace episode. These episodes represent individual events that took place within the system, for example, initializing interaction with another service, activating an internal component, making a database query, etc. An example of a Zipkin trace element is shown in Figure 3.

JSON		
traceId	c8c30c35ea801559f88d613e99c37677	
parentId	2bc16a413cf4d657	
id	e26f5ee751a313a7	
kind	CLIENT	
name	insert 3ac7d9ef-d829-4717-af38-adb4782f53f5.test_entity	
timestamp	1708181860505789	
duration	148	
localEndpoint	serviceName	service_4
	ipv4	172.17.0.1
tags	db.connection_string	h2:mem:
	db.name	3ac7d9ef-d829-4717-af38-adb4782f53f5
	db.operation	INSERT
	db.sql.table	test_entity
	db.statement	insert into test_entity (name,id) values (?,?)
	db.system	h2
	db.user	sa
	otel.library.name	io.opentelemetry.jdbc
	otel.library.version	2.0.0-alpha
	otel.scope.name	io.opentelemetry.jdbc
	otel.scope.version	2.0.0-alpha
	thread.id	60
	thread.name	http-nio-8084-exec-4

Fig. 3. Zipkin Trace Element Example

This structure is intended for the convenience of analyzing and processing the received trace data, which contributes to a more accurate interpretation of the processes and relationships operating in the system. It is worth noting that many such elements form an unconnected graph, which consists of a number of directed subgraphs organized like a tree. The number of such subgraphs corresponds to the number of unique traceId values, and the root vertices in these subgraphs are elements with a parentId value equal to null. An example of such a subgraph in a disconnected OpenTelementry event graph is shown in Figure 4.

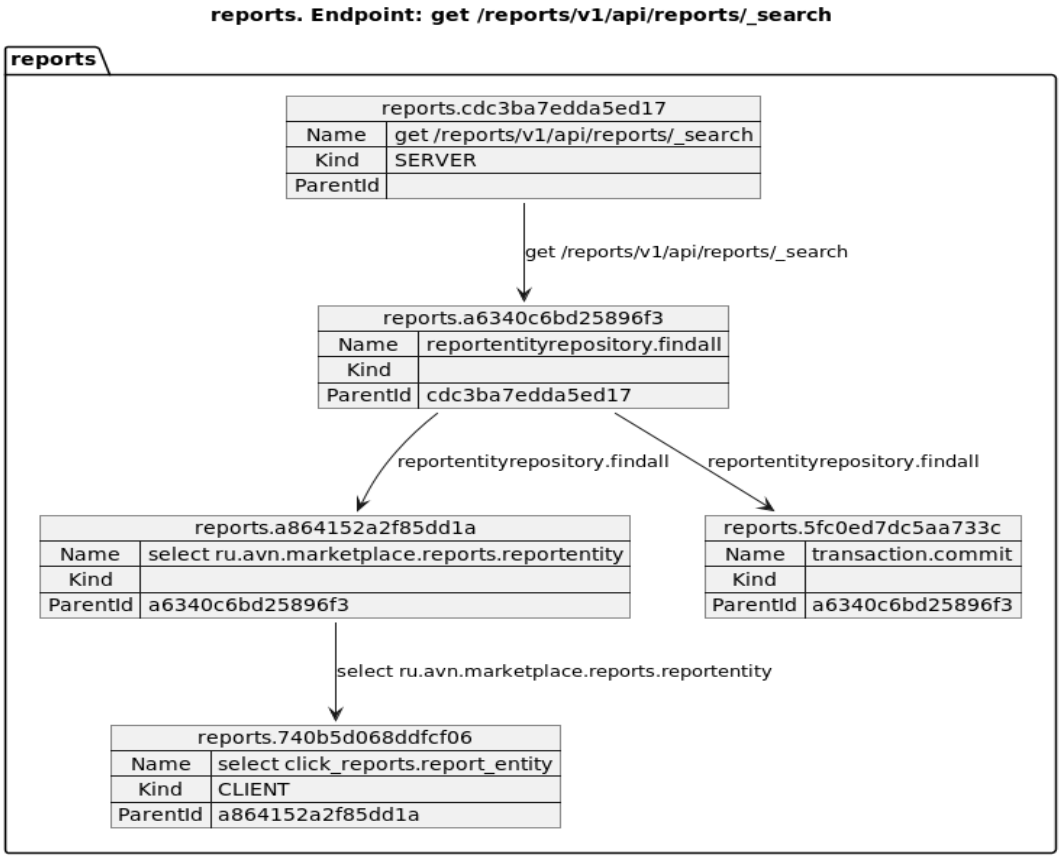

Fig. 4. Example of a subgraph in a disconnected OpenTelemetry event graph

The data processing stage for the purpose of visualization involves the construction of a new graph. In this context, the vertices of this graph are produced by extracting the unique values of the localEndpoint.serviceName attribute from the vertices of the source graph. The edges in the newly created graph represent connections between different services, their identification is carried out based on a search for the sequence of changes in the values of localEndpoint.serviceName. For example, if the current vertex of the graph contains the value service1, and the next node of the tree contains the value service2, then based on this information, an edge of a new graph is formed connecting the vertices service1 and service2. Thus, the new graph allows you to most clearly and conveniently represent the interaction between various services. Additionally, you need to perform a search procedure for all root vertices in the original graph with non-empty values in the tags[‘user_agent.original’] field. According to the developed algorithm, for each service where the tags[‘user_agent.original’] value in the original graph is not empty, a separate vertex is created in the new graph. Then an edge is created that connects the new vertex to the service vertex. Thus, the new vertex represents a specific service user in the context of the new graph. As a continuation of the developed algorithm, an additional search for vertices in the source graph occurs, based on the detection of non-empty values using the tags[‘db.name’] label. Based on each unique value, consisting of a combination of the values tags['db.name'], tags['db.system'], tags['db.user'] and tags['db.connection_string'], a set of new vertices is formed . These newly created vertices are database containers that are used by services. To ensure the connectivity of the graph, edges are established that connect these vertices to services. The criterion corresponding to service nodes is determined based on the value of localEndpoint.serviceName. The process of constructing a new graph is complemented by searching for vertices in which the value of tags[‘messaging.destination.name'] is non-empty. For each unique set of values composed of tags['messaging.destination.name'], tags['messaging.system'], tags['network.peer.address'] and tags['network.peer.port'] , a vertex is formed, which is subsequently included in a new graph between existing vertices. It is worth noting here that the vertex of the input graph, which is the source of information (with the kind attribute as PRODUCER), is displayed in the new graph as the source vertex with the service name corresponding to the value of localEndpoint.serviceName. Accordingly, the vertex representing the receiving

service (with its kind attribute equal to CONSUMER) becomes the destination for the new vertex's data. Thus, the organization of the vertices of the new graph is realized, which are message brokers participating in the interaction between services. This provides a deeper and more accurate understanding of the interactions that occur during system operation, since message brokers play a central role in the exchange of information between services. An example of the operation of the data preparation algorithm is shown in Figure 5.

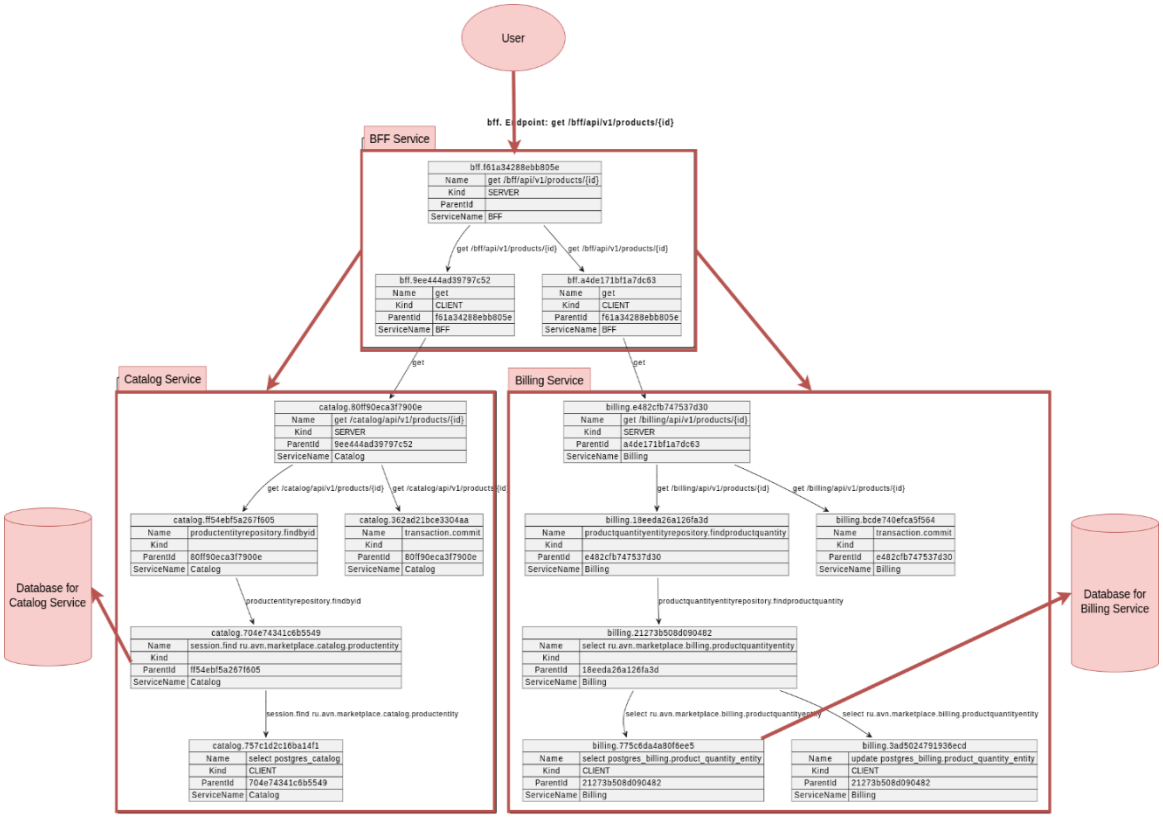

Fig. 5. An example of preparing trace graph data for architectural visualization

The resulting new graph will be used to visualize the C4 diagram of the second level - the container level, which shows the interaction at the service level within the information system. In the context of analyzing complex systems that have the form of a graph of service interactions, it is possible to use the Label Propagation Clustering clustering algorithm. This allows us to identify an additional level of abstraction - the level of subsystems, and in the C4 model the level of context. The essence of the Label Propagation Clustering algorithm is to implement an iterative approach, during which class labels are transmitted from the selected vertex to all neighboring ones. In this case, the vertex takes the most frequently occurring label among its neighbors, which allows similar objects to be grouped into common clusters. The main goal of this algorithm is to minimize the intra-cluster distance and maximize the distance between clusters, that is, grouping interconnected vertices into common clusters. This approach allows you to identify the structure of service interactions at more refined levels and analyze complex systems more effectively. Applying the Label Propagation Clustering algorithm to the service interaction graph allows us to identify subsystems that are larger, functionally related groups of services.

The stage of direct visualization of the architecture at the context and container level of the C4 model consists of creating text files that are syntax diagrams in the PlantUML format. PlantUML is a diagram visualization tool that allows you to describe the structure of diagrams as text. For ease of working with the C4 model, there is a special C4 library for PlantUML, which simplifies the process of visualizing diagrams of this model. A particularly important aspect of this stage is the use of the graph structure obtained in the previous stages, together with the use of template engines such as Mustache. Template engines allow you to create templates for generating PlantUML text files, which ensures automatic generation of text diagrams based on specified templates. An example template for creating a PlantUML C4 diagram using the Mustache template engine is shown in Figure 6.

```
@startuml
!include https://raw.githubusercontent.com/plantuml-stdlib/C4-PlantUML/master/C4_Container.puml

{{#users}}
Person({{name}}, "{{originName}}", "{{#devices}}{{.}}{{/devices}}")
{{/users}}
{{#databases}}
    ContainerDb({{name}}, "{{name}}", "{{classifier}}")
{{/databases}}
{{#services}}
    Container({{name}}, "{{name}}", "{{address}}")
{{/services}}
{{#broker}}
    ContainerQueue({{name}}, "{{name}}", "{{classifier}}")
{{/broker}}
{{#connections}}
    Rel({{source.name}}, {{destination.name}}, "", "{{#protocols}}{{.}}{{/protocols}}")
{{/connections}}
@enduml
```

Fig. 6. Example template for creating a PlantUML C4 diagram using the Mustache template engine

As a result of this stage, ready-made files are created that contain all the necessary information to describe the diagrams of the C4 model of the first and second levels. These text materials are then used to construct a visual diagram of the system architecture, which provides a more visual representation of the structure and relationships of components in the information system in question.

3. Conclusion

In conclusion of this article, it can be noted that the author has done work to develop an algorithm for visualizing the architecture of an information system built on the basis of microservices. A unique algorithm was proposed that includes several key stages: preparing the information system infrastructure, including setting up OpenTelemetry to collect accurate and reliable data, downloading data, processing data and direct visualization. For data analysis and processing, data downloaded from Zipkin was used, which made it possible to obtain the most complete picture of the interaction between system components. To create a second-level C4 model graph, a specially developed data processing algorithm was used to accurately reflect the structure of microservices. In order to test the operation of the developed algorithm, a diagram of the microservice architecture of the prototype marketplace system was obtained, which consists of the microservices BackendForFrontent, Billing, Delivery, Notification, Catalog, Discount and Payment. A diagram of the architecture of the prototype marketplace system in C4 notation of the container level is presented in Figure 7.

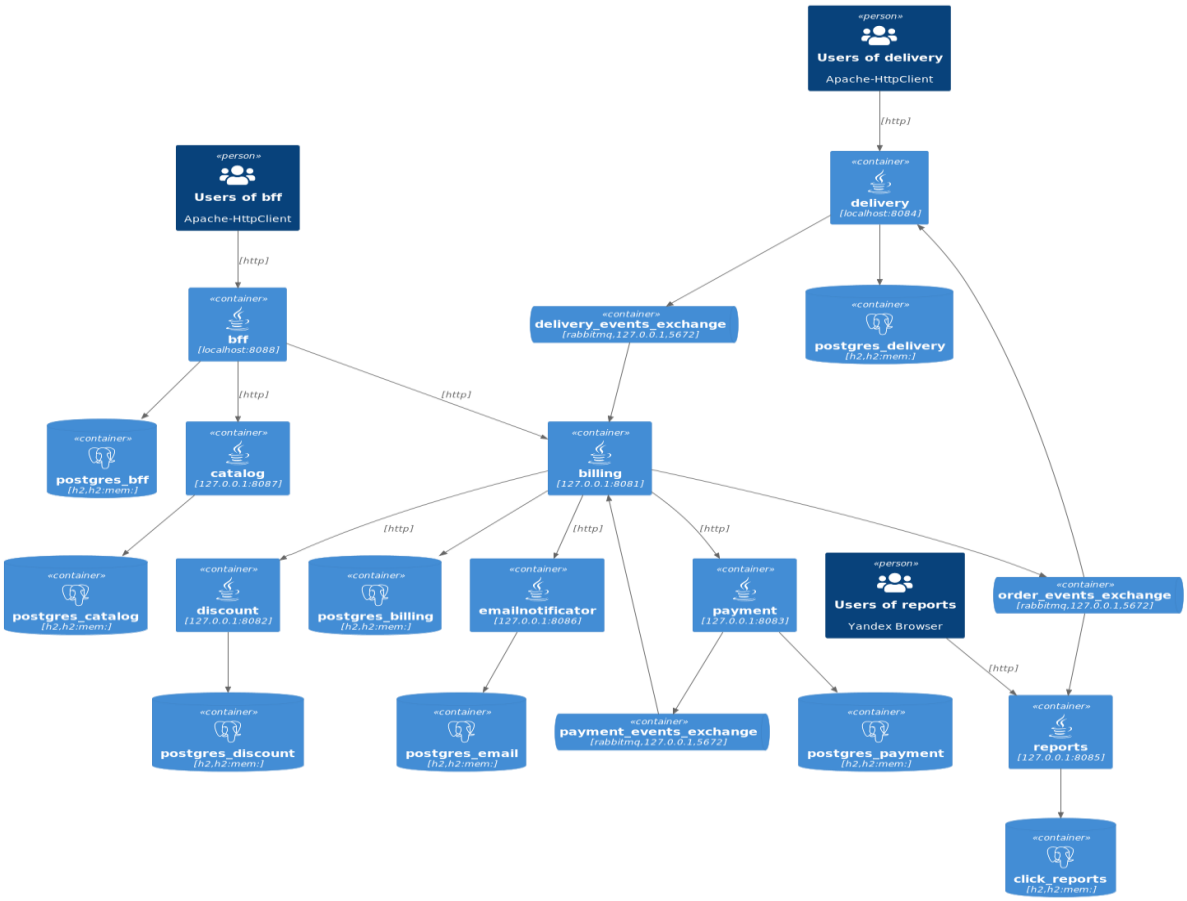

Fig. 7. Architecture diagram of a prototype marketplace system in container-level C4 notation

The graph of this diagram was supplemented by using the Label Propagation Clustering algorithm, which made it possible to extract the C4 model of the first level of context, which is the structure of subsystems. Diagram C4 of the first level of context for a prototype marketplace system after applying the Label Propagation Clustering algorithm is shown in Figure 8.

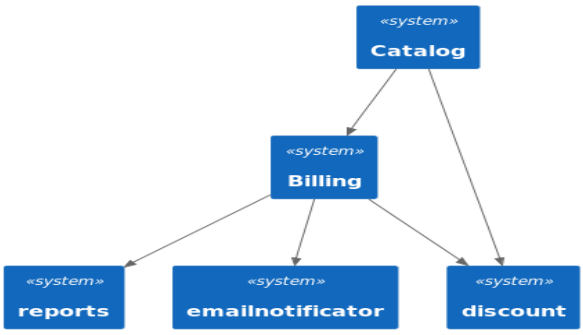

Fig. 8. Diagram C4 of the first level of context for a prototype marketplace system after applying the Label Propagation Clustering algorithm

The work also proposed a visualization option based on the use of the PlantUML tool. The PlantUML text file template was created using the Mustache template engine. This approach allows you to customize the visualization to the specifics of the information system. Thus, the results of the author's work represent a work algorithm for visualizing the microservice architecture of an information system. This opens up new opportunities for analyzing and optimizing the structure of the system, simplifies the process of design, development and maintenance, which in general can significantly increase the efficiency of specialists in this field.

The authors would like to thank the management of Bunin Yelets State University for financial support of this study.

Refrences

[1] *Nadeikina, L. A., Cherkasova N. I.* Creating applications based on microservices // Informatization and communication. 2019. No. 4. pp. 107-112. https://doi.org/10.34219/2078-8320-2019-10-4-107-112

[2] *Kornienko D.V., Mishina S.V., Shcherbatykh S.V. and Melnikov M.O.* (2021) Principles of securing RESTful API web services developed with python frameworks. Journal of Physics: Conference Series. 2021, 2094(3), 032016. https://doi.org/10.1088/1742-6596/2094/3/032016

[3] *Valdivia H. A., Laura-Gonzalez A., Lemon K.* Patterns of microservice architecture: a multidisciplinary literature review // Proceedings of the Institute of System Programming of the Russian Academy of Sciences. 2021. Vol. 33, No. 1. pp. 81-96. https://doi.org/10.15514/ISPRAS-2021-33(1)-6

[4] *Hüseyin Ünlü, Dhia Eddine Kennouche, Görkem Kılınç Soylu, Onur Demirörs.* Microservice-based projects in agile world: A structured interview. Information and Software Technology, Volume 165, 2024, 107334, ISSN 0950-5849. https://doi.org/10.1016/j.infsof.2023.107334

[5] *Lulu Wang, Peng Hu, Xianglong Kong, Wenjie Ouyang, Bixin Li, Haixin Xu, Tao Shao.* Microservice architecture recovery based on intra-service and inter-service features. Journal of Systems and Software, Volume 204, 2023, 111754, ISSN 0164-1212. https://doi.org/10.1016/j.jss.2023.111754

[6] *Romanov V. Y.* A tool for reverse engineering and refactoring software written in Java // International Journal of Open Information Technologies. 2013. Vol. 1. No. 8. pp. 1-6.

[7] *Andrea Janes, Xiaozhou Li, Valentina Lenarduzzi.* Open tracing tools: Overview and critical comparison, Journal of Systems and Software, Volume 204, 2023, 111793, ISSN 0164-1212. https://doi.org/10.1016/j.jss.2023.111793

[8] *Rudometkin V. A.* Monitoring and troubleshooting in distributed high-load systems // Cybernetics and programming. 2020. No. 2. pp. 1-6. https://doi.org/10.25136/2644-5522.2020.2.32996

[9] *L. Giamattei, A. Guerriero, R. Pietrantuono, S. Russo, I. Malavolta, T. Islam, M. Dînga, A. Koziolek, S. Singh, M. Armbruster, J.M. Gutierrez-Martinez, S. Caro-Alvaro, D. Rodriguez, S. Weber, J. Henss, E. Fernandez Vogelin, F. Simon Panojo.* Monitoring tools for DevOps and microservices: A systematic grey literature review. Journal of Systems and Software, Volume 208, 2024, 111906, ISSN 0164-1212. https://doi.org/10.1016/j.jss.2023.111906

[10] *Boitsov B. V., Minakova O. V., Potsebneva I. V.* An architectural approach to creating software tools for working with evaluation tools of an information system according to quality parameters // Quality and Life. 2022. № 1(33). Pp. 23-30. https://doi.org/10.34214/2312-5209-2022-33-1-23-30

[11] *Kitanin, S. S., Makarevich A.D.* Building the architecture of a software system for a geoinformation application of augmented reality // Modern science: actual problems of theory and practice. Series: Natural and Technical Sciences. 2023. No. 6-2. pp. 90-100. https://doi.org/10.37882/2223-2982.2023.6-2.20

[12] *Namiot D. E., Romanov V. Yu.* 3D visualization of architecture and software metrics // Scientific visualization. 2018. Vol. 10. No. 5. pp. 123-139. https://doi.org/10.26583/sv.10.5.08

[13] *Johann Mortara, Philippe Collet, Anne-Marie Dery-Pinna.* Visualization of object-oriented software in a city metaphor: Comprehending the implemented variability and its technical debt. Journal of Systems and Software, Volume 208, 2024, 111876, ISSN 0164-1212. https://doi.org/10.1016/j.jss.2023.111876

[14] *Vyugina A. A., Kroshilina A. A.* Visualization of the client-server architecture using delegates in C# // Methods and means of information processing and storage: Interuniversity collection of scientific papers / Edited by B.V. Kostrov. – Ryazan: Ryazan State Radio Engineering University named after V.F. Utkin, 2022. pp. 140-144.