On supervising and coordinating microservices within web applications on the basis of state machines

Oleksiy Oletsky[1,*,†], Vitalii Moholivskyi[1,†]

[1] *National University of Kyiv-Mohyla Academy, Skovorody St.,2, Kyiv, 04070, Ukraine*

Abstract

An approach to supervising and coordinating microservices by applying state machines has been developed. Both orchestration and choreography approaches to coordinate microservices are considered. A prototype of a library based on this approach is reported. Such advantages as central control over complex distributed execution flow, declarative description of the system and the workflows within it, rich facilities for visualization, possibilities for applying mathematical methods of analysis, keeping track of long-running background jobs, monitoring the current state of a system, testing coordination logic with mocked implementation details, and debugging coordination issues using only state machine without running microservices are illustrated on examples. A way to estimate the importance measures of specific microservices based on PageRank-like techniques is suggested. Some suggestions for using other mathematical methods are discussed.

Keywords

Microservice architecture, state machines, coordination of microservices, orchestration of microservices, choreography of microservices, distributed workflows, PageRank-like algorithms, ranking microservices

1. Introduction

Microservice architecture has gained wide acknowledgment among software architects since its introduction in 2011 [1-3]. Traditional monolithic architecture exposes more and more weaknesses and vulnerabilities in coping with such challenges as scalability demands, fault tolerance standards, flexibility requirements [2]. Such issues become especially crucial for the situation, currently very typical, when an application is deployed in cloud, and facilities for replacing or updating separate modules independently of other parts of the application matter very much.

On the other hand, microservice architecture puts forward new challenges, imminent just for it. Sometimes such challenges are caused by not very high proficiency of developers and even of architects, who may have troubles even with more or less serious monolithic applications, not saying about much more sophisticated microservice-based ones. But even having been designed perfectly, microservice-based applications are prone to such integral troubles as complexity of overall management and such as establishing and maintaining proper coordination across microservices within the application. Developers and admins suffer from a lack of control, observability and monitoring capabilities over the distributed execution flow within microservice-based web applications. Testing and debugging complexity have also increased significantly compared to the monolithic architecture.

To our knowledge, now there is no strong theoretical base for microservice architecture, commonly recognized and accepted. Heuristic approaches prevail in this realm, even though a theory

Information Technology and Implementation (IT&I-2024), November 20-21, 2024, Kyiv, Ukraine

[*] Corresponding author.

[†] These authors contributed equally.

oletsky@ukr.net (O.Oletsky); mogolivskiy@gmail.com (V. Moholivskyi)

0000-0002-0553-5915 (O.Oletsky); 0009-0001-2654-7798 (V. Moholivskyi)

of microservice based systems might find its roots in such comprehensive and fundamental disciplines as theory of complex systems, theory of parallel computing and distributed systems, etc. There are many sound results in the field of distributed systems [4], but a microservice-based application as an example of distributed systems has many specific features [5], which makes it impossible to apply those results directly without any upgrades taking into account those specific features. In particular, classic theory of distributed systems paid insufficient attention to the problem of coordinating across nodes in real-time mode. Another problem is related to scaling distributed systems, and within microservice architecture new approaches can be suggested [6]. The problems of fault isolation [7], deployment and technology flexibility [8] are worth mentioning as well.

Not a single, but a very important issue is to entrench facilities for declarative describing the overall build of a microservice-based application as well as actual and possible workflows within it, easy for supervising and interpreting them. Moreover, such declarative descriptions appear to be not only tools for representing the architecture, but also for triggering control actions as well.

In this paper we are suggesting the approach based on so-called state machines addressing these issues. A prototype of a software package implementing this approach and demonstrating main possibilities addressing issues mentioned above is reported as well. In addition to this, some ways of increasing effectiveness, reliability and security of applications by using certain mathematical methods are discussed.

2. State machines as the theoretical model for describing microservices

A state machine can be defined as a 5-tuple [9]

$$(Q, \Sigma, \delta, q_0, F), \tag{1}$$

where Q is a finite set called the states, Σ is a finite set called the alphabet, δ is a state transition function $\delta\colon Q \times \Sigma \to Q$ (determines the next state given a current state and an input symbol), q_0 is the start state, where $q_0 \in Q$, F is a set of final state, where $F \subseteq Q$.

Within the context of microservices, the alphabet Σ is a set of messages (events) occurring between or within microservices. Therefore, a state transition function $\delta\colon Q \times \Sigma \to Q$ receives an internal or external event with a current state and returns the next state. Moreover, the function δ could trigger control procedures for meeting the functional requirements of an application. Those control procedures could be as follows (including but not limited to): persisting data to the database, sending messages (events) to the message broker, executing remote procedure calls, etc.

State machines provide a well-defined structure for representing different states of a system and the transitions between them and thus can be regarded as a tool for declarative descriptions. As a result, they enable to improve understanding of overall structure, organization, and behavior of the system. Software engineers get the possibility to effectively monitor and visualize how the system transitions through various states. Explicitly defined transitions between states make the execution flow control simpler. Each state can be provided with a specific configuration as a condition for transitioning to another state. In this way, the risk of unexpected behaviors is decreased as there shall be a limited set of defined in advance well-known transitions between states. State machines also can improve maintainability as the system can grow and evolve by adding new states or transitions without the need to redesign the entire system. Each state machine can operate independently or interact with other state machines, allowing the system to be scaled. Since each state is distinct and transitions are well-defined, it is easier to isolate and identify where issues may occur. This makes debugging and testing much simpler, as engineers can focus on individual states and transitions rather than on the entire system. Also, testing a state machine is more straightforward because you can verify each state and transition to ensure the system behaves as expected in every scenario. State machines encourage reusable state logic. As states and transitions can be abstracted and reused across different parts of the system, it leads to more efficient development. Complex workflows involving multiple concurrent tasks or parallel processes can be handled efficiently by multiple interacting state machines. State machines are ideal for modeling long-running processes, such as business workflows or distributed transactions, by clearly defining each step (state) and what needs to happen to move to the next step. State machines provide mechanisms to define error states and recovery paths when a failure or unexpected event occurs. For example, the system can transition to a safe state or invoke compensating actions to undo failed operations. The system can be designed to transition into safe or idle states in the event of failure, allowing for graceful degradation of the system instead of a complete crash.

3. Methods and models of microservices coordination using state machines

To our knowledge, using state machines for coordinating microservices have not been studied enough so far. This is one of the main questions our paper is focused on.

When a microservice web application is designed, the first factors which should be taken into account are scalability, reliability, and performance. The main goal of a software architect is to create a highly effective system. The effectiveness of the system is the main criteria for designating service boundaries. However, established service boundaries not only bring undisputed value but also create new challenges in terms of communication and coordination. As microservices are typically deployed as independent containers running on different servers, having effective means of communication and coordination between microservices is essential for managing distributed execution control flow between independent services. Therefore, several methods of microservices coordination have emerged:

1. Orchestration [10]
2. Choreography [10]
3. Saga Pattern for distributed transactions [11]
4. API Gateway Pattern [12]
5. Service Mesh [13]

Without a doubt, each of the above methods has its use cases. But, in this research, we will concentrate on the first two methods as they may benefit the most from using state machines. The third method surely can benefit from state machine utilization, too. But as this pattern builds on top of orchestration or choreography [14], there is no need to examine it separately. As methods applied to orchestration and choreography can also be indirectly applied to the Saga pattern too.

3.1. Orchestration and choreography coordination methods using state machines

Orchestration-based coordination focuses on centralized control over all microservices communications [15, 16, 17]. It provides an effective way to manage complex workflows, letting each service focus on specific business logic instead of scattering on communication with other services [15]. The orchestrator defines a sequence of tasks and handles service invocations, does retries, and manages transactions. One prominent disadvantage of the orchestration approach is the increasing complexity for an orchestrator as the number of services grows. This problem can be resolved using a state machine to take over control of orchestrator logic. In this way, the behavior of a service acting as coordinator can be defined by a state machine. However, modeling the whole microservice as a one single state machine is not practical in a real-world scenarios. Much more practical approach would be to model specific activities aka tasks, which need to be performed, as an individual state machines. Thereby, the orchestration will have many state machines applicable for specific tasks at its disposal.

Choreography-based coordination focuses on decentralized control over workflow between microservices [15, 16, 17]. Each microservice is responsible for coordinating the execution flow by passing execution control to a corresponding microservice. Such a model is also often called event-driven architecture. In this approach, microservices communicate by producing and consuming events without direct knowledge about other microservices. The following approach creates a lot of autonomy allowing each microservice to operate independently. However, in such a system, it's hard to guarantee eventual consistency at all times as thoughtful handling of workflow order and failures is required. Also, debugging and monitoring flow through microservices is much harder than with the orchestration approach. Those issues may be softened using state machines. As the choreography approach implies independence of services, each service will need to get a state machine. Having many state machines per service to manage different workflows can also be considered for a more voluminous service from a domain logic perspective. As a result, a system will become more manageable because of a clear definition of states for each microservice. Reliability will improve because of the structured error-handling and lowered chance of getting into inconsistent states.

The comparison of orchestration and choreography coordination with state machines is shown at Figure 1.

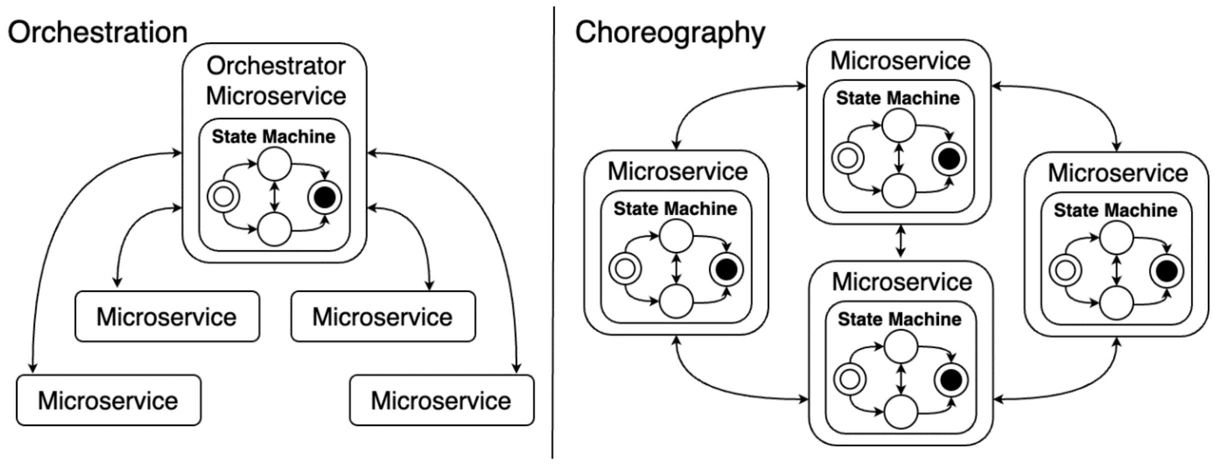


Figure 1: The comparison of orchestration and choreography coordination with state machines

The comparison summary of both approaches is given in Table 1.

With orchestration, control is centralized. A single orchestrator microservice manages the interactions between other microservices, directing the overall workflow. This orchestrator uses a state machine to determine the sequence of actions and transitions, guiding each service on what to do and when to do it. The advantage of this approach is that it simplifies management, monitoring, and debugging, as everything is controlled and tracked by the orchestrator. However, it also introduces potential drawbacks, such as creating a bottleneck at the orchestrator and making the system more vulnerable to failure due to this single point of control.

In contrast, choreography relies on decentralized control, where each microservice operates independently and communicates directly with others as needed. Every microservice manages its own behavior through an internal state machine, allowing for greater flexibility and scalability. This approach is more resilient to failure, as there is no central orchestrator that could become a point of failure. However, the downside of choreography is that it can be more complex to manage and debug, as the coordination between services happens in a distributed manner, requiring careful design to avoid issues.

Thoughtful consideration is required to choose a more appropriate approach for a given circumstance. In general, the orchestration based coordination style is more compatible with state machines and can benefit the most from their capabilities. The comparison summary of both approaches is given in Table 1.

Table 1
The summary of orchestration and choreography coordination with state machines comparison

Characteristic	Orchestration	Choreography
Control	Centralized through an orchestrator microservice	Decentralized, each microservice is independent
Coordination	The orchestrator directs the flow and interactions	Microservices communicate directly
State Machine	One main state machine inside the orchestrator	Each microservice has its own state machine
Advantages	Easier to monitor and manage	Greater flexibility, scalability, and resilience
Disadvantages	Orchestrator can become a bottleneck and single point of failure	Harder to debug, more complex coordination between services

4. Library for coordinating microservices using state machines

A prototype of a library addressing the issues mentioned above was developed and is reported in the paper. We called the developed library SMMC (State Machine Microservice Coordinator). The architecture of the created library is shown at Figure 2.

To ensure the proper operation of a state machine in a microservice environment, the library implements some fundamental functionality as follows:

- registering state machine definitions
- handling the creation and execution of a state machine instance, its suspension and persistence, retrieval and revival when required
- creating required data structures in ArangoDB database and Kafka message broker
- handling the exchange of events between microservices
- handling graceful shutdown
- guaranteeing scalability, high availability, and fault tolerance based on both infrastructure and implementation choices.

A very important point is to choose the proper software tools for implementing the units of the architecture shown in Figure 2.

As a state machine engine, XState library is used [18]. This choice is based on a wide range of functional abilities the library provides. Even though the library derives from classic Mealy or Moore machines [19, 20]. It complements them by state machine approaches suggested by David Harel, Grady Booch, and SCXML standard, such as nested and parallel (composite) states, internal and external events, conditional transitions, context dependence (internal memory), and triggerable control procedures [21 - 23].

As a communication mechanism, Apache Kafka message broker is used [24]. A message broker is an essential tool in a microservice architecture that provides reliability, resiliency, and scalability [12]. Thus, it is a much better method of communication than, for instance, a remote procedure call.

As a persistence layer, a multi-model ArangoDB database is used [25]. This database supports persisting both documents and graphs. Document collection is used to persist the current state of a state machine. Graph collections are used to gather data for further analysis and recommendations for software engineers.

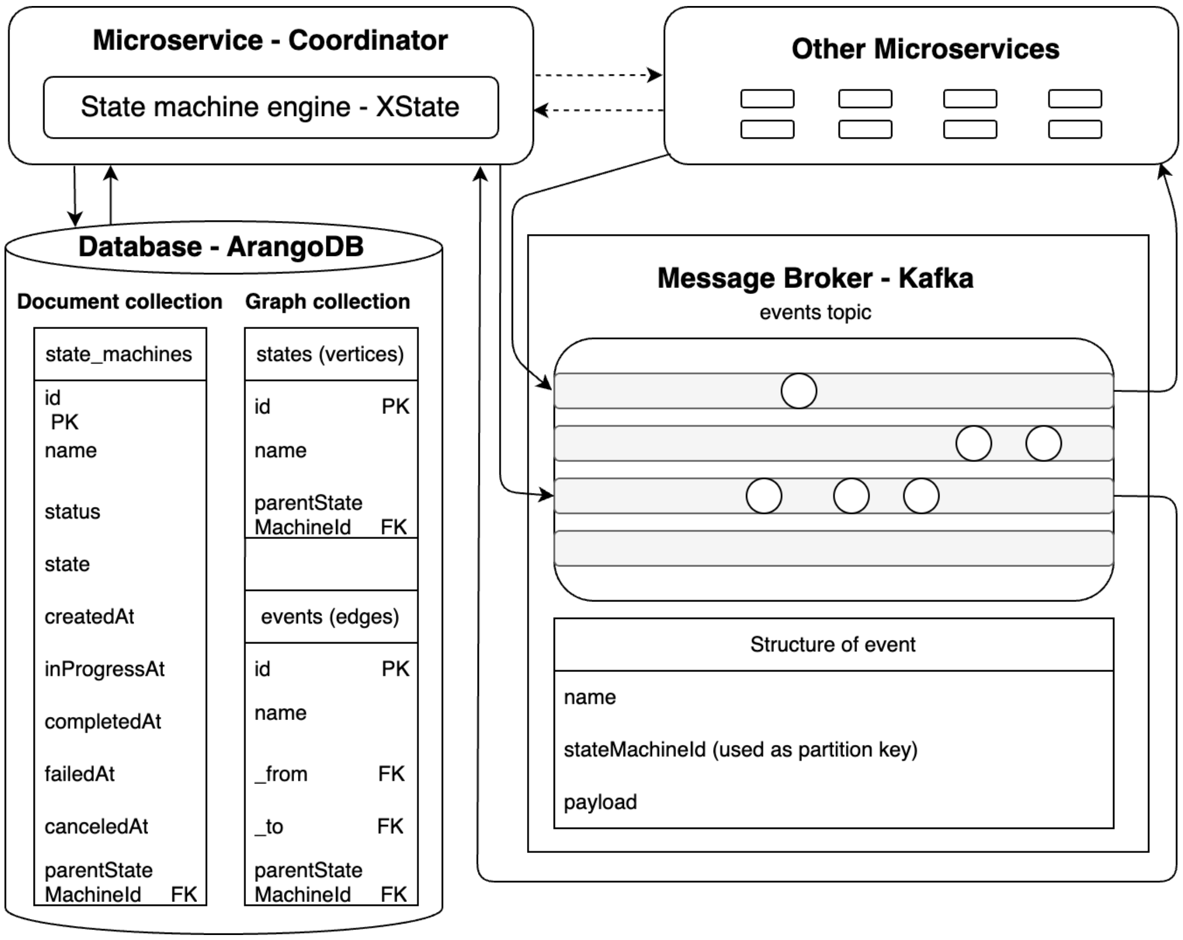


Figure 2: The architecture of the SMMC library

The library aims to handle execution, communication, and persistence issues. Another objective is to provide extended analytics capabilities based on defined system workflows. Finally, the library works toward elevating security standards for microservice architecture.

A very important thing supported by the library is providing a good possibility for visualizing the overall build of the system and the workflows within it. Basic features of visualization are taken from the XState library. These basic features were supplemented in the developed library. This includes, for instance, representing in the form of a graph illustrated on the Figure 3, which was generated by the SMMC library.

The library requires a user to have an environment with deployed ArangoDB and Kafka instances. On library initialization, a user should provide ArangoDB and Kafka clients to the main class constructor. Only the official ArangoDB client [25] and KafkaJS library [26] are supported at the moment. The main class called Smmc contains methods of migration and rollback responsible for creating and deleting data structures required for the library to function. Calling the migrate method will create database smmc in the ArangoDB with state_machines, states, and edges collections. The migration method will also create event topics in Kafka. Calling rollback will remove created resources. A user may call the migrate method on a microservice startup or alongside other migrations. Then, a state machine should be registered using the registerStateMachineDefinition method. A state machine definition and its logic should be provided according to the XState syntax [18]. The reverse method, unregisterStateMachineDefinition, is also available. After all state machine definitions are registered, the start method should be called. It will check connections to the ArangoDB and Kafka. If connections are healthy, the consumer and the producer for events topic in Kafka will be initialized and started. On a microservice shutdown, the stop method should be called to ensure a graceful shutdown of state machines processing inside the SMMC. Before returning, the stop method will wait for the completion of in-progress events processing and active state machines' persistence to the database. Once the SMMC is started, the getStateMachineManager method can be called to get an instance of StateMachineManager, which allows creating state machine instances, getting them to check the current state, and deleting them.

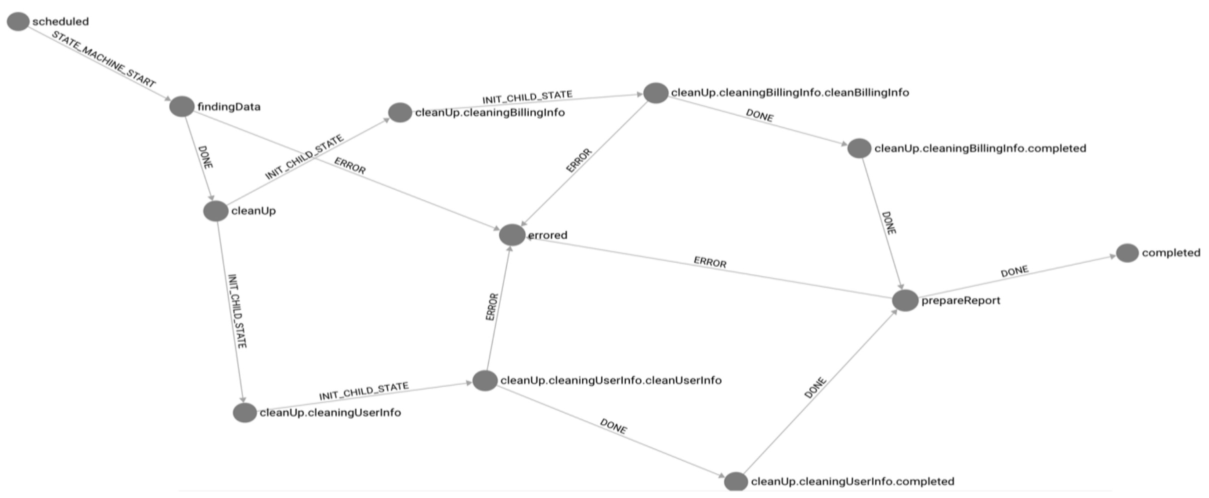


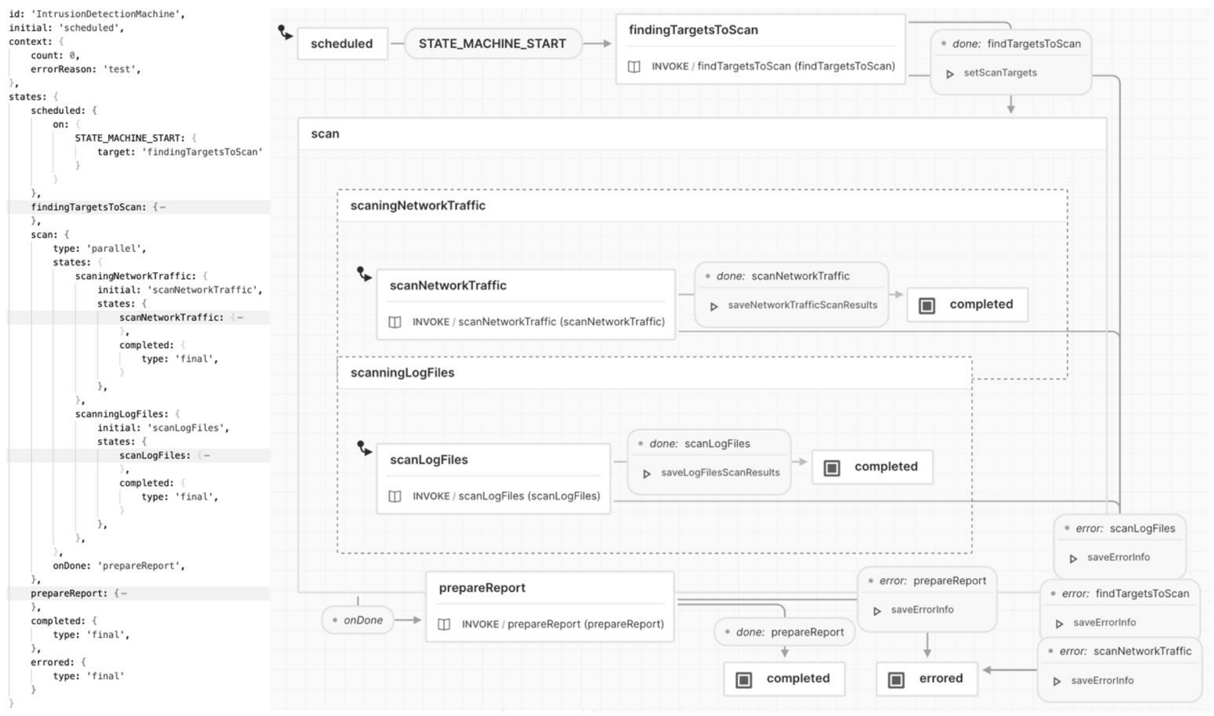
Figure 3: Visualization of workflows within the system in the form of a graph

Figure 4 shows how distributed microservice execution workflow can be defined and how that definition can be visualized in the form of a flow chart. Meaningfully, this is the workflow related to scanning of the system in order to detect possible intrusions. Firstly, targets for security scanning are retrieved from a database based on the defined security policies. Secondly, scanning of the network traffic and of the system log files is performed simultaneously. Finally, a report containing the results of the scanning is created. Then procedures for counteracting possible intrusions can be triggered on this basis.

The displayed example illustrates the solution of some previously discussed issues. The domain logic can be defined using state machines without worrying about cross-microservice communication and state machine persistence. The library provides a convenient means of forking and joining workflows using composite states defined as parallel state machines. In Figure 4, the state called "scan" is defined with the type "parallel" forks the workflow, whereas the "onDone" hook joins it. The example also shows how all the information about microservices events exchange is concentrated in one place. In contrast, with typical free-coded microservices, where each of the services holds some data and looking through the code base of each service is required to get even a high-level overview of workflows existing in the system. We also see a predictable error handling. Error states are explicitly defined together with transition, which will happen if an error occurs. This approach makes easier to perform corrective actions based on a state where an error occurred. The right part of Figure 4 pictures a state machine visualization automatically generated based on the programmed definition at the left part of Figure 4. Such visualizations can be very useful for both getting a high-level understanding of a system at a glance and inspecting it in depth to study each tiny detail about the system. Additionally, such a visualization can visually show the current state of a workflow, which is especially useful for the long-running workflows like cleanup or indexation jobs.

The library guarantees scalability, high availability, and fault tolerance based on both infrastructure and implementation choices. Firstly, infrastructure dependencies such as ArangoDB and Kafka provide such guarantees themselves [24, 25]. Secondly, the library implementation approach follows a stateless philosophy [12], all state is kept in ArangoDB database, and an orchestrator microservice running the library is exclusively a stateless processor of events from Kafka. Such an approach allows scaling an orchestrator microservice to as many replicas as required.

The design of Kafka events topic also enables scaling by increasing the number of partitions. State machine instance id is used as a partition key for events topic to avoid concurrency issues.



```
id: 'IntrusionDetectionMachine',
initial: 'scheduled',
context: {
    count: 0,
    errorReason: 'test',
},
states: {
    scheduled: {
        on: {
            STATE_MACHINE_START: {
                target: 'findingTargetsToScan'
            }
        }
    },
    findingTargetsToScan: {⋯
    },
    scan: {
        type: 'parallel',
        states: {
            scaningNetworkTraffic: {
                initial: 'scanNetworkTraffic',
                states: {
                    scanNetworkTraffic: {⋯
                    },
                    completed: {
                        type: 'final',
                    }
                },
            },
            scanningLogFiles: {
                initial: 'scanLogFiles',
                states: {
                    scanLogFiles: {⋯
                    },
                    completed: {
                        type: 'final',
                    }
                },
            },
        },
        onDone: 'prepareReport',
    },
    prepareReport: {⋯
    },
    completed: {
        type: 'final',
    },
    errored: {
        type: 'final'
    }
}
```

Figure 4: The example of state machine definition and its visualization created using XState visualizator

Currently, the SMMC library is best designed for running on a single microservice, which should act in the role of an orchestrator. Usage according to the choreography approach on many microservices is possible, but a library user should take care of spawning instances of state machines on each service separately. Automation of related state machines creation for choreography coordination is a subject for future improvement. Having a separate state machine on each microservice shall create a multi-agent model, which could be a foundation for modelling decision making in a multi-agent environment based on a Markov chain. State machines provide a solid theoretical model for the workflows between microservices, which enables implementing further analytics capabilities.

5. Use of PageRank-like techniques for analyzing importance of microservices

Analyzing measures of significance that might be ascribed to specific microservices appears to be a very important point. This can be carried out by means of the classical PageRank algorithm [27] and techniques based on it and similar to it. Within this approach, relations and dependencies between microservices should be considered.

In general, PageRank-like algorithms are based on getting a stochastic matrix P representing different associations between nodes, and then vector evaluating measures of importance for each node can be obtained as a main left eigenvector of P.

Such an analysis could identify crucial microservices having a central role in the system. Knowing the most important microservices allows to set priorities in allocation of computational resources, development afford and monitoring focus for each microservice. A quantity of event dependencies between microservices defined by state machines will be a base for computing weight of a relationship. This is a difference between our research and related ones [28]. We can compute the weight of a relationship between microservice M_j and microservice M_i using the formula:

$$w_{ji} = \frac{N_{ij}}{N_{max}} \tag{2}$$

where N_{ij} is a number of event dependencies specified in state machines between M_j and M_i, and N_{max} is a maximum number of event dependencies detected between any M_j and M_i.

Then, PageRank-like technique can be applied to the graph in order to get microservices importance scores. Brin and Page [27] defined PageRank algorithm as:

$$PR(i) = \gamma \sum_{j \in V} \frac{a_{ji}}{d_j^{(out)}} PR(j) + \frac{1 - \gamma}{\mathrm{n}} \tag{3}$$

By adding Ding concept of weight [29] we get:

$$PR(M_i) = \gamma \sum_{M_j \in V} \frac{w_{ji}}{\mathrm{s}_j^{(out)}} PR(M_i) + \frac{1 - \gamma}{\mathrm{n}} \tag{4}$$

where:

- $PR(M_i)$ is the score of importance of a microservice M_i,
- γ is the damping factor (usually 0.85), it represents the probability that an event will follow actual dependencies versus a random jump,
- w_{ji} is the weight of the interaction between microservice M_j and microservice M_i (2),
- $\mathrm{s}_j^{(out)}$ is the sum of the weights of all outgoing interactions from microservice M_j, normalizing the contribution
- $\frac{1-\gamma}{\mathrm{n}}$ represents a random jump probability needed to give a chance of importance event to isolated microservices.

Based on knowledge of events correspondence to microservices and definitions of state machines, the graph with weights w_{ji} in edges can be computed for the set of microservices M. For example, let’s take the security platform with the microservice architecture, which has eight microservices: policies management service, firewall management service, treat detection service, vulnerability management service, vulnerability scanning service, incident response service, monitoring service and metrics service. When using state machines, relationships between those microservices are defined by state machine definition and can be converted to a graph. The weight w_{ji} of a relationship between M_j and M_i can be automatically computed based on the quantity of communications between a pair of microservices. As a result, the graph pictured at Figure 5 with weights of microservices relationship importance is received.

The importance of each microservice is computed using Formula 4. As a result, a score for each microservice from the given set of microservices M is received. Scores calculated for a given example (Figure 5) are presented in Table 2. Table 2 shows that incident response service, firewall management service, and policies management service are the most significant services. To be specific, it means that other services depend on them the most. Consequently, failure of those services could be extremely critical for the system.

The suggested approach allows software engineers to identify the most important microservices within their applications to take preventive actions to avert potential issues. Such preventive actions may include increasing test coverage, leveling up requirements for code review, writing supporting documentation, having more detailed logging, adding additional monitoring metrics, creating extra real-time alerts, etc.

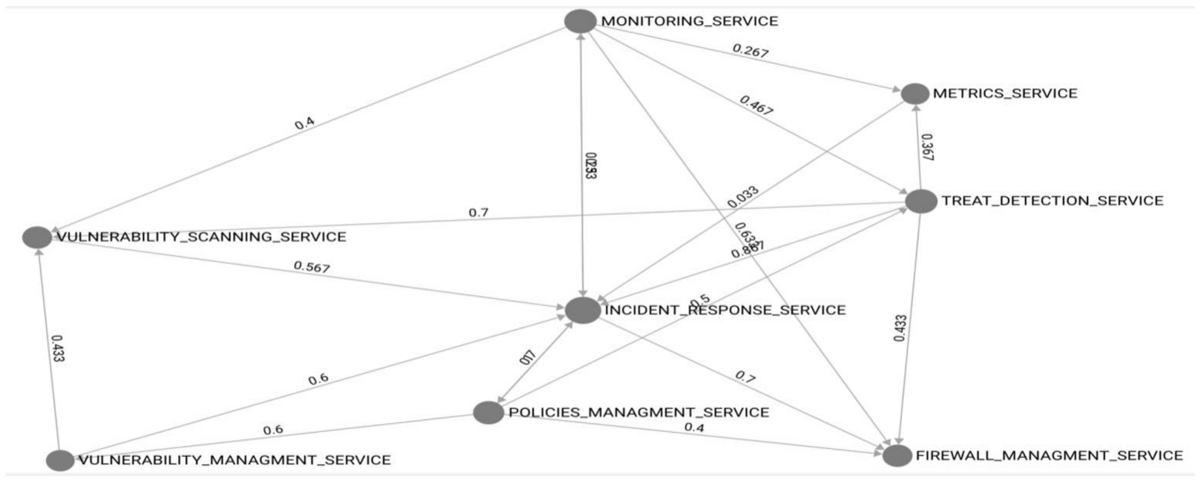

MONITORING_SERVICE
METRICS_SERVICE
TREAT_DETECTION_SERVICE
VULNERABILITY_SCANNING_SERVICE
INCIDENT_RESPONSE_SERVICE
POLICIES_MANAGMENT_SERVICE
VULNERABILITY_MANAGMENT_SERVICE
FIREWALL_MANAGMENT_SERVICE
0.267
0.467
0.4
0.367
0.7
0.033
0.567
0.433
0.6
0.6
0.4
0.7
0.433
0.5

Figure 5: The graph with weights of microservices relationship importance

Table 2
Microservices and their importance scores

Microservice	Importance score
Policies management service	0.141
Firewall management service	0.179
Treat detection service	0.082
Vulnerability management service	0.069
Vulnerability scanning service	0.098
Incident response service	0.279
Monitoring service	0.088
Metrics service	0.064

Additionally, having a reliable measure of importance for each microservice is beneficial for service ranking in regression testing as it enables testing of the system in the most efficient way [28].

6. Conclusions and discussion

The paper considers how to improve processes of control and monitoring within a microservice-based application, mainly by means of entrenching coordination across microservices. We are suggesting an approach based on using state machines for declarative describing the overall build of an application and workflows within it.

A prototype of a software package implementing this approach and demonstrating main possibilities for addressing the issues mentioned above is reported as well. The main functions implemented by the prototype are as follows:

- registering state machines definitions
- handling the creation and execution of a state machine instance, its suspension and persistence, retrieval and revival when required
- creating required data structures in ArangoDB database and Kafka message broker
- handling the exchange of events between microservices
- handling graceful shutdown
- guaranteeing scalability, high availability, and fault tolerance based on both infrastructure and implementation choices.

Advantages of using state machines for typical tasks such as defining the business logic of a web application, coordinating complex workflows between many microservices, keeping track of long-running background jobs, monitoring the current state of a system, testing coordination logic with mocked implementation details, and debugging coordination issues using only state machine without running microservices are showcased on the workflow depicted on the Figure 4.

For constructing more formalized models of coordinating microservices, especially for models involving modelling forking and joining workflows like the one mentioned above, use of Petri nets appears to be very promising [30].

Declarative descriptions provided by state machines enable to explore the microservice-based application by means of various mathematical methods and on this base to work out recommendations for how to improve certain features of it. Namely, this paper illustrates the use of PageRank-like techniques for evaluating measures of importance for different microservices.

There are many other possible applications of mathematical models and methods based on descriptions provided by the state machine. For instance, explicit specifying the states and possible transitions across them enables to consider a Markov chain of transitions. Provided that we are able to get transitional probabilities across the states, we can calculate the probabilities that the system shall be in the certain state in the long run on this basis. This may be related, for instance, to estimating expected loads on certain microservices or to other, more complicated, issues. So, the ability to find the stationary distribution for a Markov chain provides valuable insight into the long-term behavior of a state machine, especially in scenarios where transitions between states are probabilistic.

An explicit graph representation enables to apply techniques typical for graphs. For instance, use of the A* algorithm [31] enables to find the optimal sequence of operations aimed at moving the system from its known current state to the desired target state.

Another promising possibility is related to triggering certain control procedures connected with a certain node of the state machine. For instance, if a light- or middle-threating cyberattack is detected, a system built either on orchestration or choreographic principles might make a probabilistic decision about which microservices should be stopped and which of them should be kept alive. Within this context, the model "state-probability of action" suggested and developed in [32, 33], which addresses probabilistic decision making on the base of probabilities for being in certain states, is worth mentioning. Use of this model appears to be quite promising, especially for situations when there is a lot of similar microservices, and decisions are made by majority of votes. Another point is to take into account different factors influencing probabilistic decisions by means of constructing multi-level models the "state-probability of action" [34, 35].

In a certain sense, the state machine for a specific microservice-based application can be regarded as a certain unit of knowledge. It focuses mainly on behavioral aspects. However, it appears fruitful to combine it with the ontological-oriented approach by building proper descriptive logical systems, ontologies and/or knowledge graphs [36, 37].

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] J. Lewis, M. Fowler, Microservices. URL: https://martinfowler.com/articles/microservices.html.

[2] M. Villamizar, O. Garces, H. Castro, M. Verano, L. Salamanca, R. Casallas, S. Gil, Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud, in: 2015 10th computing colombian conference (10CCC), IEEE, 2015. doi:10.1109/columbiancc.2015.7333476.

[3] M. Loukides, S. Swoyer, Microservices adoption in 2020. URL: https://www.oreilly.com/radar/microservices-adoption-in-2020/.

[4] S. Ghosh, Distributed Systems, Chapman and Hall/CRC, 2006. doi:10.1201/9781420010848.

[5] I. Shabani, E. Mëziu, B. Berisha, T. Biba, Design of Modern Distributed Systems based on Microservices Architecture, Int. J. Adv. Comput. Sci. Appl. 12.2 (2021). doi:10.14569/ijacsa.2021.0120220.
[6] B. Christudas, Distributed Computing Architecture Landscape, in: Practical Microservices Architectural Patterns, Apress, Berkeley, CA, 2019, pp. 1–19. doi:10.1007/978-1-4842-4501-9_1.
[7] C. M. Krishna, I. Koren, Fault-Tolerant Systems, Elsevier Science & Technology Books, 2020.
[8] L. Baresi, M. Garriga, Microservices: The Evolution and Extinction of Web Services?, in: Microservices, Springer International Publishing, Cham, 2019, pp. 3–28. doi:10.1007/978-3-030-31646-4_1.
[9] M. Sipser, Introduction to the theory of computation, Thomson South-Western, 2012.
[10] C. Surianarayanan, G. Ganapathy, P. Raj, Service orchestration and choreography, in: Essentials of microservices architecture, Taylor & Francis, 2019, pp. 175–197. doi:10.1201/9780429329920-6.
[11] H. Garcia-Molina, K. Salem, Sagas, in: The 1987 ACM SIGMOD international conference, ACM Press, New York, New York, USA, 1987. doi:10.1145/38713.38742.
[12] S. Newman, Building microservices, O'Reilly Media, Incorporated, 2015.
[13] R. Sharma, A. Singh, Getting started with istio service mesh, Apress, Berkeley, CA, 2020. doi:10.1007/978-1-4842-5458-5.
[14] S. Aydin, C. B. Cebi, Comparison of choreography vs orchestration based saga patterns in microservices, in: 2022 international conference on electrical, computer and energy technologies (ICECET), IEEE, 2022. doi:10.1109/icecet55527.2022.9872665.
[15] A. Stutz, A. Fay, M. Barth, M. Maurmaier, Orchestration vs. choreography functional association for future automation systems, IFAC-PapersOnLine 53.2 (2020) 8268–8275. doi:10.1016/j.ifacol.2020.12.1961.
[16] A. Megargel, C. M. Poskitt, V. Shankararaman, Microservices Orchestration vs. Choreography: A Decision Framework, in: 2021 IEEE 25th International Enterprise Distributed Object Computing Conference (EDOC), IEEE, 2021. doi:10.1109/edoc52215.2021.00024.
[17] C. Surianarayanan, G. Ganapathy, P. Raj, Service Orchestration and Choreography, in: Essentials of Microservices Architecture, Taylor & Francis, 2019, pp. 175–197. doi:10.1201/9780429329920-6.
[18] XState documentation. URL: https://xstate.js.org/docs/.
[19] G. H. Mealy, A method for synthesizing sequential circuits, Bell Syst. Tech. J. 34.5 (1955) 1045–1079. doi:10.1002/j.1538-7305.1955.tb03788.x.
[20] E. F. Moore, Gedanken-Experiments on sequential machines, in: C. E. Shannon, J. McCarthy (Eds.), Automata studies. (AM-34), Princeton University Press, Princeton, 1956, pp. 129–154. doi:10.1515/9781400882618-006.
[21] D. Harel, Statecharts: a visual formalism for complex systems, Sci. Comput. Program. 8.3 (1987) 231–274. doi:10.1016/0167-6423(87)90035-9.
[22] G. Booch, J. Rumbaugh, I. Jacobson, Unified modeling language user guide, the (2nd edition) (the addison-wesley object technology series), 2nd. ed., Addison-Wesley Professional, 2005.
[23] State chart XML (SCXML): state machine notation for control abstraction. URL: https://www.w3.org/TR/scxml/.
[24] Apache Kafka documentation. URL: https://kafka.apache.org/documentation/.
[25] ArangoDB documentation. URL: https://docs.arangodb.com.
[26] KafkaJS Documentation· KafkaJS. URL: https://kafka.js.org/docs/getting-started.
[27] S. Brin, L. Page, Reprint of: The anatomy of a large-scale hypertextual web search engine, Comput. Netw. 56.18 (2012) 3825–3833. doi:10.1016/j.comnet.2012.10.007.
[28] L. Chen, J. Wu, H. Yang, K. Zhang, Does PageRank apply to service ranking in microservice regression testing?, Softw. Qual. J. (2022). doi:10.1007/s11219-021-09579-6.
[29] Y. Ding, Applying weighted PageRank to author citation networks, J. Am. Soc. Inf. Sci. Technol. 62.2 (2010) 236–245. doi:10.1002/asi.21452.
[30] K. H. Rueda, Applications of Petri Nets petri applications, Sci. J. Appl. Soc. Clin. Sci. 3.34 (2023) 2–10. doi:10.22533/at.ed.2163342314122.1111

[31] S. J. Russell, P. Norvig, Artificial intelligence: A modern approach, Pearson Education, Limited, 2021.
[32] O. V. Oletsky, E. V. Ivohin, Formalizing the Procedure for the Formation of a Dynamic Equilibrium of Alternatives in a Multi-Agent Environment in Decision-Making by Majority of Votes, Cybern. Syst. Anal. 57.1 (2021) 47–56. doi:10.1007/s10559-021-00328-y.
[33] O. Oletsky, Exploring dynamic equilibrium of alternatives on the base of rectangular stochastic matrices, in: CEUR Workshop Proceedings, 2021.
[34] D. Dosyn, O. Oletsky, An approach to modeling elections in bipartisan democracies on the base of the "state-probability of action" model, in: CEUR Workshop Proceedings, 2024.
[35] E.Ivokhin, O.Oletsky, Restructuring of the Model "State–Probability of Choice" Based on Products of Stochastic Rectangular Matrices, Cybern. Syst. Anal. 58-2 (2022) 242-250. https://doi.org/10.1007/s10559-022-00456-z
[36] L. Bellomarini, D. Fakhoury, G. Gottlob, E. Sallinger. Knowledge graphs and enterprise AI: the promise of an enabling technology. In ICDE, pp. 26–37. IEEE, 2019.
[37] P. Atzeni, L. Bellomarini, M. Iezzi, E. Sallinger, A. Vlad. Weaving enterprise knowledge graphs: The case of company ownership graphs. In EDBT, pp. 555–566. OpenProceedings.org (2020).