


Article

User Authorization in Microservice-Based Applications

Niklas Sanger *  and Sebastian Abeck

Research Group Cooperation & Management, Karlsruhe Institute of Technology, 76131 Karlsruhe, Germany; sebastian.abeck@kit.edu

* Correspondence: niklas.saenger@kit.edu

Abstract: Microservices have emerged as a prevalent architectural style in modern software development, replacing traditional monolithic architectures. The decomposition of business functionality into distributed microservices offers numerous benefits, but introduces increased complexity to the overall application. Consequently, the complexity of authorization in microservice-based applications necessitates a comprehensive approach that integrates authorization as an inherent component from the beginning. This paper presents a systematic approach for achieving fine-grained user authorization using Attribute-Based Access Control (ABAC). The proposed approach emphasizes structure preservation, facilitating traceability throughout the various phases of application development. As a result, authorization artifacts can be traced seamlessly from the initial analysis phase to the subsequent implementation phase. One significant contribution is the development of a language to formulate natural language authorization requirements and policies. These natural language authorization policies can subsequently be implemented using the policy language Rego. By leveraging the analysis of software artifacts, the proposed approach enables the creation of comprehensive and tailored authorization policies.

Keywords: microservices; fine-grained authorization; ABAC; engineering; structure preservation



Citation: Sanger, N.; Abeck, S. User Authorization in Microservice-Based Applications. *Software* **2023**, *2*, 400–426. <https://doi.org/10.3390/software2030019>

Academic Editor: Manuel Mazzara

Received: 11 August 2023

Revised: 31 August 2023

Accepted: 5 September 2023

Published: 19 September 2023



Copyright:  2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

The microservice architecture has become a widely popular architecture style in research and industry [1–3]. Microservice-based applications replace monolithic applications by dividing the business logic into smaller services that perform a well-defined, relatively small task [4]. Each service exposes its functionality through a web Application Programming Interface (API). There are several web API paradigms, such as Representational State Transfer (REST) [5] or Remote Procedure Calls (RPC) [6], with their most popular specifications, OpenAPI [7] and gRPC [8]. With well-designed API specifications, microservices enable the creation of reusable services. In this paper, we follow an architectural style presented by Hippchen et al. [9] and Sidler et al. [10]. The business functionality is implemented in a microservice that resides in the application layer. A frontend located in the presentation layer accesses the respective microservice. However, with the distribution of business logic across multiple microservices, the overall complexity of the management microservices increases [4]. This includes the integration of access control mechanisms.

The Open Web Application Security Project (OWASP) frequently publishes a list of security risks in web applications. In their 2021 list, broken access control is listed as the number one security risk [11]. In a microservice architecture, the complexity of access control is further increased by the distribution of services and their responsibilities. As de Almeida and Canedo [12] report, communication, trust, and access control between microservices are major challenges. Authentication and authorization are aspects of access control [13]. Authentication involves the process of verifying the identity of a subject, while authorization determines the level of access an authenticated subject is granted.

Authorization for microservice-based applications has been primarily researched as a technical aspect of evaluating and enforcing authorization decisions in the microservice

architecture [14,15]. The loose coupling of microservices offers a separation of concerns between business logic and authorization logic. This allows the modification of authorization logic without modifying the business logic. In this paper, we apply ABAC to perform authorization decisions as it complements the loose coupling of microservices by adding authorization components to the overall architecture [16].

Besides architectural challenges, the questions of what must be authorized and how authorization can be systematically formulated and consequently enforced have not been fully addressed. This implies changes to the software development process for a microservice-based application. Authorization, as a subset of computer security, is a non-functional requirement [13] which is often performed as an afterthought in software engineering [17]. To counteract this, authorization should be collected during the requirements analysis and further realized throughout the design and implementation. However, there is no widely established procedure to define authorization policies. While there are model-driven approaches to derive ABAC policies based on Unified Modeling Language (UML) models [18,19], there is a lack of research on approaches to derive ABAC policies as an integral part of a software engineering process. This also applies to the definition and creation of authorization requirements, which are needed to define what needs to be authorized and lack systematic structure [20].

We identify the integration of fine-grained authorization using ABAC in the development of microservice-based applications as the primary research gap. To address this gap, we establish four research questions investigated in this paper:

RQ1 How can ABAC policies be systematically formulated?

RQ2 How can requirements for authorization be formulated?

RQ3 How can we systematically implement authorization policies?

RQ4 How can we integrate authorization into the development of a microservice-based application?

The leading research question is the formulation of ABAC policies (RQ1) as the central authorization artifact. The definition of ABAC policies affects what must be collected during the requirements analysis (RQ2) and how policies can consequently be implemented (RQ3). The holistic integration into the development (RQ4) depends on the authorization artifacts.

To answer these research questions, the contributions of this paper are the following: First, we propose an authorization policy language to provide a structure for the formulation of natural language authorization policies. This is done with an Augmented Backus–Naur Form (ABNF) structuring the necessary aspects of ABAC. Second, a subset of the authorization policy language is provided to establish a formalization for the definition of authorization requirements during the analysis phase. Third, we provide a structure to implement authorization policies using the policy language Rego. Fourth, we provide an authorization extension for a development process of a microservice-based application spanning over the analysis, design, and implementation and test phases.

In this work, we focus on microservice-based applications using RESTful APIs, as they have a fixed set of operations following the use of Create, Read, Update, and Delete (CRUD) operations [5]. Further, authentication is not considered in this paper. That is, it assumes that a human user is already authenticated (e.g., via Open ID Connect) and can prove it (e.g., via a valid JSON Web Token (JWT) [21]).

The remainder of this article is structured as follows: Section 2 presents related work on authorization and the state of the art on (fine-grained) authorization in (microservice-based) applications. In Section 3, the authorization policy languages, the implementation of authorization policies, and the authorization extension for a development process are introduced. To demonstrate the contributions of this work, Section 4 introduces a case study using the authorization artifacts and the authorization extension in a development process. The results are discussed in Section 5. Finally, Section 6 summarizes this work.

2. Related Work

2.1. Background

Access control is a fundamental aspect of security in IT systems [13]. It consists of three aspects: authentication, authorization, and auditing [22]. In access control, a subject (human or non-human) always attempts to perform an action on an object. A subject must be authenticated before the access request can be authorized or not. The access request is logged for auditing purposes. In this work, only the terms subject and object are used to provide a common terminology and to avoid misconception (e.g., use of a subject instead of a user).

Authorization is often divided into coarse-grained and fine-grained authorization [23]. Coarse-grained access control grants access to a system or resource to a broader set of subjects. In contrast, fine-grained access control allows flexible access rights to specific resources for individual users [24,25]. An overview presenting characteristics of fine-grained and coarse-grained authorization is provided in Table 1. Popular authorization paradigms are Role-Based Access Control (RBAC) and ABAC. RBAC ties a set of permissions (e.g., read or write to a folder) to a role that can be assigned to a subject [26]. Thus, RBAC is typically used to perform rather coarse-grained authorization. RBAC still allows creating roles for individual resources or users. However, this can lead to a phenomenon called role explosion, which implies an unmanageable number of roles in an access control system [27].

Table 1. Comparison of levels of authorization granularity adapted from [28].

Characteristic	Fine Grained	Coarse Grained
Granularity	Highly detailed and specific controls	Broader and generalized controls
Scope	Control access to individual resources or actions	Manage access at higher level (e.g., functions, services)
Flexibility	High	Low
Manageability	Complex	Simple
Changing Privileges	Simple	Complex
Popular Paradigm	ABAC	RBAC

ABAC can perform an authorization decision based on a set of attributes that can be received from a subject, an object, or an environment [29]. The attributes can be formulated as a set of conditions. To perform authorization, the conditions are collected in an authorization policy that can consequently be enforced. The collection of attributes and conditions allows the creation of arbitrarily complex fine-grained access policies [30]. ABAC policies can be implemented in a policy language such as eXtensible Access Control Markup Language (XACML) [31]. To enforce ABAC policies, there is a reference architecture consisting of four main components [29]: Policy Enforcement Point (PEP), Policy Decision Point (PDP), Policy Information Point (PIP), and Policy Administration Point (PAP). The PEP receives an authorization request from a client (e.g., web browser) and forwards it to a PDP, which decides whether to allow or deny the request. The result is returned to the PEP and the request is either granted or denied to the resource. To decide if the request is valid, the PDP can use the PIP, which provides the attributes required to make the decision. The PAP allows the formulation of authorization policies. While ABAC allows the creation of fine-grained authorization policies, the complexity of introducing it into applications and managing it is high, in part due to its distributed nature [28].

To define authorization policies, an Augmented Backus–Naur Form (ABNF) is used in this paper. The ABNF is a modified version of the Backus–Naur Form (Backus–Naur

Form) [32]. The ABNF is structured using rules and elements that represent the grammar of a language. Each rule consists of a name, followed by an equals sign "=", and the elements that define the rule. Elements can be terminal symbols (literals) or references to other rules. The notation allows for optional elements, repetition, and grouping, making it more flexible than the traditional BNF. The ABNF is used in specification of internet protocols. For example, the specifications of the HTTP protocol [33] or OAuth 2.0 [34] use the ABNF.

Listing 1 presents an excerpt from the HTTP 1.1 specification. The rule *Request-Line* (line 1) refers to five other rules followed by a Carriage Return Line Feed (CRLF). Line 3 presents a rule defining how an HTTP version is specified, e.g., HTTP 1.1. Finally, the rule *Method* in line 4 defines the available HTTP methods, e.g., GET.

Listing 1. Excerpt from the HTTP 1.1 specification [33].

```

1 Request-Line = Method SP Request-URI SP HTTP-Version~CRLF
2
3 HTTP-Version = "HTTP" "/" 1*DIGIT "." 1*DIGIT
4 Method = "OPTIONS" | "GET" | "HEAD" | "POST" | ...
5 ..

```

2.2. State of the Art

Integrating authorization into a microservice-based application addresses a subset of security that focuses on the application layer [35]. There are several approaches and solutions for integrating authorization into a microservice-based application. Banati et al. [36] propose an authorization orchestrator using JSON web tokens, OAuth, and OpenID. Each microservice has an IAM module that communicates with the authorization orchestrator and enforces decisions. Sauwens et al. [15] describe a distributed authorization middleware called ThunQ that can perform authorization decisions early (i.e., when the first service is reached) and lazily (i.e., evaluating decisions only when possible). ThunQ allows authorization decisions to be made at the microservice level by including a query modifier. Nehme et al. [14] present an access control solution for microservice architecture based on a combination of OAuth 2.0 and XACML technologies. Their solution requires a centralized access control server that holds the necessary data to evaluate a request. This contradicts the loose coupling of microservices. Further, the solutions presented focus on the technical implementation of authorization in a microservice-based application. They answer the question of how to implement authorization in a microservice architecture. However, these solutions neglect the question of what must be authorized. To answer this question, authorization policies must be created based on user requirements. The process of creating authorization policies is also called policy engineering.

Das et al. [37] provide a classification for engineering approaches for ABAC policies. The classes to emphasize are the top-down and bottom-up approaches. Bottom-up approaches create policies based on mining past access requests, examining logs, or mining existing role matrices. Bottom-up approaches require existing data to mine policies. Top-down approaches start from scratch, with no prior data other than existing natural language policy documents. The inherent complexity of these approaches is the transformation of natural language documents into an authorization policy. Thus, the authors identify the creation of authorization policies based on natural language documents as inexact.

Brossard et al. [20] propose a systematic life cycle for implementing ABAC based on enterprise experiences. The approach structures the development of authorization policies into the phases of a software development process. This includes gathering authorization requirements, identifying relevant attributes, implementing and testing policies, and deployment. To define an authorization requirement, the authors suggest the use of use cases. The authorization requirement is written in a natural language format. The authors use XACML to implement the authorization policies. The process by Brossard et al. [20] provides a promising structure for developing authorization policies. However, the process is rather coarse and lacks details on how to systematically create artifacts. For example,

the authors do not provide a structure for the creation of authorization requirements or authorization policies. Alohalay et al. [38] focus on automating the extraction of attributes based on those presented by Bossard et al. using natural language processing.

Another approach to automating ABAC policy development is presented by Narouei et al. [39], who describe a policy engineering framework for deriving ABAC policies from natural language documents using natural language processing. Similar to [20], the authors use existing requirements documents to identify and extract relevant policy elements (e.g., subject and object). Attributes are then extracted from the elements and formatted into an XACML policy. The approach of Narouei et al. does not include the definition of how the authorization requirements are documented in the requirements analysis.

Zolotas et al. [19] present RESTSec, which enables the generation of secure RESTful services using low code. The authors use model-driven engineering with a Unified Modeling Language (UML) metamodel for ABAC to generate XACML policies. Fatemian et al. [40] also propose a model-driven engineering approach that uses a UML metamodel for ABAC to generate XACML policies. This approach allows the creation of XACML policies based on the transformation of the UML metamodel. The authors provide a graphical interface to create the policies which can then be transformed. These approaches require manual definition of authorization requirements. In addition, the authorization requirements must still be analyzed for their objects, actions, and attributes, which leaves room for inaccuracy.

The approaches of Bossard et al., Narouei et al., Zolotas et al., and Fatemian et al. can be classified as top-down policies because they require existing documents to extract relevant information. In addition, Bossard et al. neglect the process of creating the required natural language documents. A bottom-up mining approach is proposed by Talukdar et al. [41]. The authors examine existing access requests and create authorization rules accordingly. For the development of new microservice-based applications, bottom-up mining approaches cannot be applied, as there are no existing data to derive policies. However, the creation of additional authorization policies based on, e.g., logs, can further strengthen the security of a microservice-based application.

3. Authorization in Microservice-Based Applications

The creation of authorization policies can take several forms. According to [29], Natural Language Policies (NLPs) and Digital Policies (DPs) should be considered. NLPs are statements governing the management and access of objects that are human readable, which can subsequently be transformed into machine-enforceable access control policies. Digital policies are access control rules that can be compiled into machine executable code [29]. Subject, object, attributes, and rules are the building blocks for a digital policy. The digital policy can be enforced by a PDP. Throughout the development phases of a microservice-based application, the available information regarding the authorization changes in clarity and structure. We address this by introducing three artifacts which range from a natural language policy to a digital policy.

Figure 1 provides an overview of the introduced artifacts in the respective development phases. The authorization requirement is an NLP which is created in the analysis phase. To create an authorization requirement, we propose an Augmented Backus–Naur Form (ABNF) providing a structure for natural, human-readable sentences. With the design phase, the knowledge regarding authorization is more consolidated and the relevant ABAC terms (e.g., subject and object) are known. Therefore, we introduce an artifact authorization policy which is a further defined authorization requirement containing the building blocks for the digital policy. Similar to the authorization requirement, we propose an ABNF containing more structured rules to create a further structured authorization policy. Thus, we classify an authorization policy as a natural language policy and an intermediate artifact to a digital policy. Finally, in the implementation and test phase, the authorization policy is transformed into a digital policy by implementing the policy using Rego.

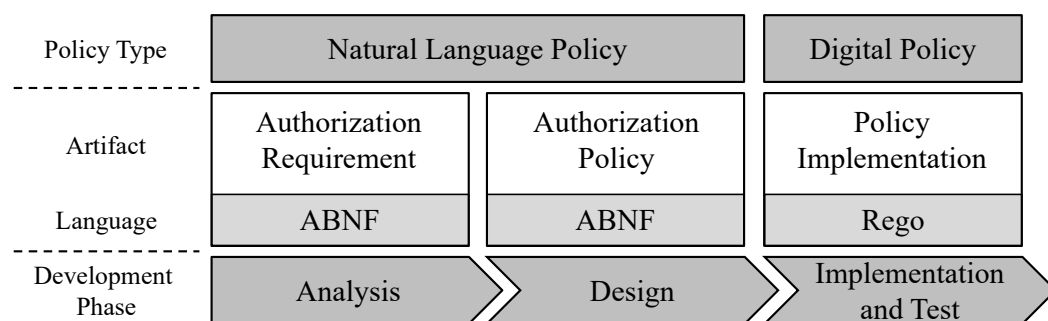


Figure 1. Classification of policy types and policy artifacts.

Since the authorization policy is an intermediate artifact between NLP and DP, we first introduce the ABNF for authorization policies in Section 3.1. Based on the authorization policy, we introduce the ABNF for the authorization requirement as a simplified authorization policy in Section 3.2. The implementation in Rego based on is introduced in Section 3.3. Finally, the integration into a development process is introduced in Section 3.4.

Complementing the introduction of the proposed authorization artifacts, we introduce two running examples:

1. Alice is working in the finance department of the Berlin branch of a company. She is only allowed to retrieve a list of projects she is assigned to. Following company guidelines, Alice is only allowed to access the records from Germany.
2. Bob wants to see the overview of a horror book in a digital book library. To perform that request, Bob must have less than EUR 10 debt at the library and meet the age restriction of the requested book.

These examples are created to cover the access to a single object as well as the access to a set of objects. Both commonly occur in microservice-based applications when using RESTful APIs.

3.1. ABNF for Authorization Policy

The purpose of the ABNF for authorization policies is to provide a uniform structure for natural language authorization policies. Authorization policies defined using this language are an intermediate step before the implementation of an ABAC policy using a corresponding policy language.

Listing 2 provides the authorization policies for the running examples using the ABNF introduced in Listing 3. In the first authorization policy, shown in line 2, the subject wants to see every project they are assigned to. There are two subject conditions for this policy. The first condition compares the subject's department with a value; the second condition compares the branch the subject is employed in with a location. The action is set to *GET*. Since a set of projects is being accessed, the text *every object in* is used to note that the authorization rules are applied to every project which is part of the object. The object is set to the HTTP path the projects can be reached at (i.e., */projects*). To check the ownership of the project, the ID of the subject is compared with the attribute assignee of the object. Finally, the location from which the request is made is checked, which is equal to Germany. The second authorization policy evaluates whether the debt of the subject is less than 10 and the performed action is *GET*. The object is a single object located in the path */book/{id}*. Finally, the attribute rating of the object is compared to the age of the subject.

To create the policy language, we must first identify the aspects that are required by ABAC. According to [29], we need to address the aspects of a subject, an action, an object, and their attributes. Since the policy is a representation of rules and relationships, the rules must also be defined. In addition, the environmental conditions must be defined in a policy language. The policy language is presented in Listing 3. The general structure of the authorization policy is written in natural language and thus readable for developers

without prior knowledge. Line 1 represents the initial ABNF rule of the authorization policy. The goal is to define what a subject can do to an object given a set of conditions.

Listing 2. Example authorization policies.

```

1 #AuthorizationPolicy1
2 A subject with subject.department == Finance AND subject.branch == Berlin can perform
   action GET on every object in /projects for which subject.id = object.assignee AND
   environment.location == Germany
3 #AuthorizationPolicy2
4 A subject with subject.debt < 10 can perform action GET on /book/{id} IF object.rating <=
   subject.age

```

Listing 3. Augmented Backus–Naur form for authorization policies.

```

1 AuthorizationPolicy = "A subject " subAttributes " can perform action " action " on "
   objectDecision conditions
2 action = "GET" / "POST" / "PUT" / "PATCH" / "DELETE"
3 object = *VCHAR
4 operator = "<" / "<=" / "==" / ">" / ">=" / "is" / "not" / "contains" / ..
5 attribute = *VCHAR
6 value = *VCHAR
7
8 subAttributes = "" / "with" subCond
9 subCond = "subject." attribute operator value
10 subCond = / subCond " AND " subCond / subCond " OR " subCond / " ( " subCond " OR "
   subCond " ) " / " ( " subCond " AND " subCond " ) "
11
12 objectDecision = object / object " IF " / " every object in " object " for which~"
13
14 conditions = "" / conditions " AND " conditions / conditions " OR " conditions / " ( "
   conditions " OR " conditions " ) " / " ( " conditions " AND " conditions " ) "
15 conditions = / "subject." attribute operator "object." attribute / "object." attribute operator value
16 conditions = / "environment." attribute operator value

```

The first rule is called *subAttributes* and is used to identify the subject attributes. A subject can either have no attributes or one or more attributes. This distinction is made in line 8. If a single subject attribute is used, another rule called *subCond* is used. As shown in line 9, a structure for the conditions of a subject attribute is provided. A subject attribute with the notation "*subject.attribute*". The rule *attribute* is shown in line 5. In ABNF, **VCHAR* allows us to write an arbitrarily long (*) set of all printable characters (VCHAR). The subject attribute must be compared to a value. Thus, the rules *operator* and *value* are created. The rule *operator* is presented in line 4. It contains a set of comparison operators such as <, >, or ==. Finally, the rule *value* is similar to the rule *attribute*, presenting an arbitrarily long amount of printable characters. Using these rules in place, a subject attribute can be compared to a value. However, if multiple subject attributes are required, the rule *subCond* depicted in line 9 is not sufficient. ABNF allows a rule to be extended by providing increment alternatives, which are presented by the notation "=/". Line 10 presents the alternatives for the subject conditions which provide logical operators such as *AND* and *OR*. Applying the logical operators allows the creation of a complex chain of statements to describe subject attributes. For example, the statement (*subject.age* > 10 *AND* *subject.age* < 18) *OR* *subject.age* > 30 allows restricting access in a policy to a specific range of ages.

Following the subject attributes, the authorization policy has the rule *action*. Since the authorization policy language is created for RESTful APIs, the rule *action* presented in line 2 provides HTTP operations (e.g., GET, POST). Using the HTTP operations in the authorization policy allows simplification of the implementation in a policy language.

Next, the object needs to be defined. Therefore, the rule *objectDecision* is used. In RESTful APIs, the request is always performed on a single resource that returns either a single object or a set of objects. For example, the endpoint *GET /projects* returns a list of projects, while the endpoint *GET /projects/{id}* returns a specific project. This behavior must be taken into account when creating an ABAC policy. In general, a subject should only perform an action (e.g., view) on the objects they have access to. To that end, the rule *objectDecision* provides three alternatives. First, the request is only performed for a single object without conditions. Second, the request is performed on an object with conditions which is characterized by an *IF*. Third, the request is performed on a set of objects which is marked by the statement "every object in" object "for which". Similar to the definition attributes and values, the rule *object* (line 3) defines the name of an object as an arbitrarily long set of printable characters.

Finally, the conditions for the access decisions must be defined. For this, the ABNF provides the rule *conditions* (lines 14 to 16). Similar to the conditions of subject attributes, the object conditions can be concatenated using logical operations such as *and* and *or*. Additionally, the conditions allow comparing subject attributes with object attributes, or object attributes with a specific value. The conditions use the rules *operators* and *values* presented in lines 4 and 6, respectively. Finally, environmental conditions (line 16) can be created by comparing the attributes with a given value.

3.2. Authorization Requirements Language

During the analysis phase of a software development project, functional and non-functional requirements are collected. Functional requirements describe what a system should be able to do [42]. At this stage of the project, the exact details are not defined and are rather coarse. Typically, the details are added during the design phase. However, throughout the requirements analysis, it is important to think about what actions a user can perform under what circumstances. This can be done by using use cases [43]. Firesmith calls the collection of such requirements authorization requirements [44].

To support the capture of authorization requirements throughout the analysis phase, we propose a further formalization using an ABNF presented in Listing 4. Compared to the authorization policies, the ABNF of the authorization requirements only provides a coarse structure that can be used in the design phase to formulate authorization policies. For this purpose, the ABNF for the authorization requirement can be seen as a subset of the ABNF of the authorization policies. The initial rule of the authorization requirement is presented in line 1. An authorization requirement provides a natural language structure similar to [20]. The primary goal of the authorization requirement is to capture the action, the object, and the conditions. Thus, for each of these aspects, a separate rule is created. The action (line 2) is an arbitrarily long amount printable characters (*VCHAR). Compared to the authorization policy, the action is constructed openly. In the analysis phase, the HTTP operation is unknown. Thus, an action might be *add* or *update*. The structure of the object (line 3) is the same as the action (i.e., *VCHAR). For example, an object could be *projects* or *books*. Finally, the conditions are either an arbitrary number of characters (line 5) or the concatenation of multiple conditions using the logical operators *and* and *or*.

Listing 4. Augmented Backus–Naur Form for authorization requirements.

```

1 AuthorizationRequirement = "A subject can perform action " action " on object " object " IF "
    condition
2 action = *VCHAR
3 object = *VCHAR
4 condition = *VCHAR
5 condition = / condition " AND " condition / condition " OR " condition / " ( " condition " OR "
    condition " ) "

```

Listing 5 contains two authorization requirements for the running example. The first authorization requirement is presented in line 2. The subject can only view their projects if they are assigned to the project, work in the finance department, and perform the request from Germany. Compared to the authorization policy presented in Listing 2, the authorization requirement is less formalized and primarily structures the content of the analysis artifacts. The second authorization requirement is presented in line 5. As can be seen from the authorization requirements, throughout the analysis phase, the exact names of the attributes, e.g., as necessary design artifacts, are not yet available. However, the authorization requirements allow the collection of requirements during the analysis phase, which can consequently be transferred to authorization policies.

Listing 5. Exemplary authorization requirements using the ABNF.

```

1 #AuthorizationRequirement1
2 A subject can perform action retrieve on object list of projects IF subject is in the finance
   department AND subject is assigned to project AND action is performed from~Germany
3
4 #AuthorizationRequirement2
5 A subject can perform action read on object book IF debt of subject has less than EUR 10 debt
   AND subject meets age restriction

```

3.3. Authorization Policy Implementation

The authorization policy using the ABNF presented in Listing 3 can be implemented directly in a policy language. We use the Open Policy Agent (OPA), an open-source policy engine supported by the Cloud Native Computing Foundation (CNCF) [45]. The OPA provides the policy language Rego, which allows policies to be written as code artifacts [46]. When using the OPA as a PDP, proxies such as Envoy [47] or Traefik [48] can be used as a PEP as they provide an integration for the OPA. This allows us to access the details of an HTTP request in a JSON format.

Listing 6 presents the structure for implementing an authorization policy. Each authorization policy is encapsulated by an allow statement (lines 1 and 13), which becomes true if all of its enclosed statements are true. The structure from the authorization policy can be transferred to implement the authorization policy in Rego. Lines 2 to 4 evaluate the subject attributes. Then, the action is evaluated in line 6 followed by the object in line 8. Finally, each condition from an authorization policy can be evaluated.

Listing 6. Authorization policy implemented in Rego.

```

1 allow {
2   # Subject Conditions
3   subject.attribute1 == X
4   Y in subject.attribute2
5   # Action
6   action == input.attributes.request.http
7   # Object
8   object := input.attributes.request.path
9   # Conditions
10  # condition1
11  data[object.id].owner == subject.id
12  # alternative
13  response := http.send({
14    "method": "GET",
15    "url": <PIP>
16  })
17  response.owner == subject.id
18 }

```

In Rego, it is possible to define variables outside an allow statement. In this case, the subject (line 3) is a variable that contains subject attributes, e.g., by extracting the content of a JWT. The action can be received from the variable input (line 6), which contains the content of the HTTP request performed by the client (i.e., user). The content of the HTTP request is provided by the PEP (e.g., Envoy). In line 6, the HTTP operation is retrieved from the variable *input.attributes.request.http*. In lines 7 and 8, the object is evaluated by matching the HTTP path. Starting in line 9, the conditions are evaluated, for example, by comparing the object owner to the subject identifier (line 11). To perform the evaluation, the attribute data of the object must be known. In ABAC, the attributes are provided by a PIP [29]. The OPA provides several options to access attribute data. One option is to store the required data in a JSON format in the OPA itself, which is utilized in line 11 through the variable data. An alternative is to perform an HTTP request to a dedicated PIP as displayed in lines 13 to 16. The result can then be compared to the identifier of the subject (line 17).

Using helper functions in Rego allows creating more sophisticated policy implementations by creating functions for common reusable logic. Listing 7 shows an overview of helper functions. For example, to identify the subject, a JSON Web Token (JWT) can be used [21]. A JWT can be verified in Rego by checking its signature. The claims of a JWT can be extracted from the JWT's payload (lines 14 through 22). This allows the creation of statements such as *subjectIsInFinanceDepartment* (lines 5 through 7) that can be used in multiple policies. Other examples include a statement for an action (lines 1 through 3) or a method for the determination of a project assignment (lines 9 through 12). By using helper functions, authorization policies can be structured in a simpler and more compact way. For example, the authorization policies described in Listing 2 are implemented using helper functions in Listing 8.

Listing 7. Helper functions implemented in Rego.

```

1 actionIsGet {
2     "GET" = http_request.method
3 }
4
5 subjectIsInFinanceDepartment {
6     claims.department == "Finance"
7 }
8
9 subjectIsAssignedToProject {
10    input.parsed_path = ["projects", projectid]
11    data[projectid].assignee == claims.sub
12 }
13
14 claims := payload {
15     [_ , payload, _] := io.jwt.decode(bearer_token)
16 }
17
18 bearer_token := t {
19     vs. := http_request.headers.authorization
20     startswith(v, "Bearer ")
21     t := substring(v, count("Bearer "), -1)
22 }

```

Listing 8. Example authorization policy implementation utilizing helper functions.

```

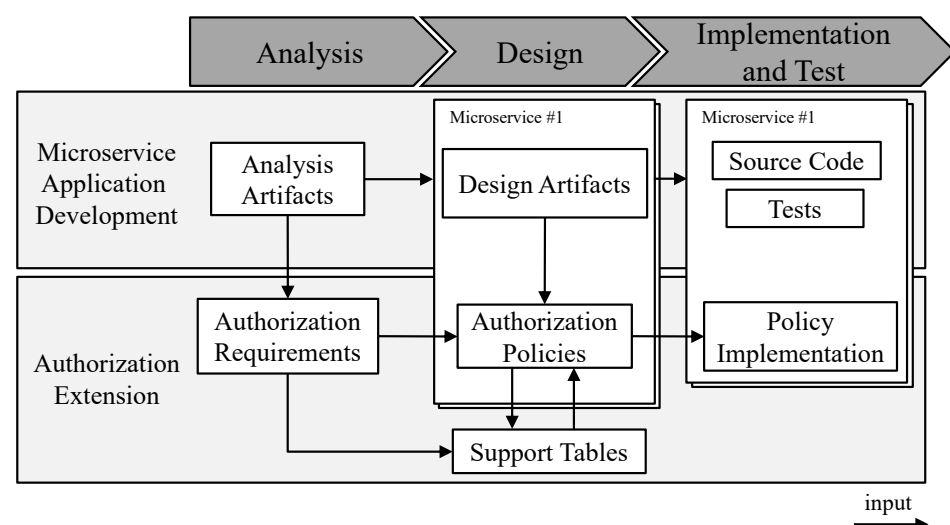
1 #AuthorizationPolicy1
2 allow {
3   subjectIsInFinanceDepartment
4   subjectBranchIsBerlin
5   actionIsGET
6   objectIsProjects
7   subjectIsAssignedToProject
8   locationIsGermany
9 }
10 #AuthorizationPolicy2
11 allow {
12   subjectDebtLessThan10
13   actionIsGET
14   objectIsBooks
15   objectRatingSmallerEuqualSubjectAge
16 }

```

3.4. Development Process Integration

To integrate fine-grained authorization into the development of a microservice-based application, the development process for creating such an application must be adapted. The development of such an application is highly individual and will vary between different developers. To support developers in using the previously presented authorization artifacts, we propose an extension to an existing development process.

Figure 2 provides an overview of the microservice-based application development process following the analysis, design, and implementation and test phases. Microservice-based application development creates analysis artifacts (e.g., use cases) during the analysis phase. In the design phase, the analysis artifacts are used to structure the microservice-based application into a set of microservices [9]. In addition, design artifacts such as API specifications or class diagrams are created for each microservice. Finally, in the implementation and test phase, the design artifacts are implemented in a programming language and tested (e.g., unit tests, component tests, and E2E tests) [49].

**Figure 2.** Analysis and design of a microservice-based application with artifacts for authorization extension.

To extend the development with authorization, each phase is complemented with the presented authorization artifacts. During the analysis phase, the authorization require-

ments are created. As described in Section 3.2, the authorization requirements provide a basic structure for formulating authorization requirements in natural language. To write such an authorization policy, the analysis artifacts can be used as an input, since the analysis artifacts should define what the application can do under what circumstances [42]. Based on the authorization requirements, authorization policies are created for each microservice. Additionally, we propose the use of support tables that contain information about attributes between multiple microservices. Finally, the policies are implemented in the implementation and test phase.

To create the authorization artifacts presented in Figure 2, one or more steps must be performed in each development phase. Figure 3 provides an overview of the tasks to be performed. In the analysis phase, the authorization requirements must be elicited to understand what needs to be authorized. This is performed by applying a process to existing analysis artifacts to extract the required information. In the design phase, the attributes required to enforce an authorization requirement with an ABAC policy are extracted from the design artifacts. Next, the authorization requirement is transformed into an authorization policy. Finally, in the implementation phase, the policy is implemented in Rego. We propose to preserve the structure in the integration of authorization into the development by creating one authorization policy for one authorization requirement and create one policy implementation for one authorization policy.

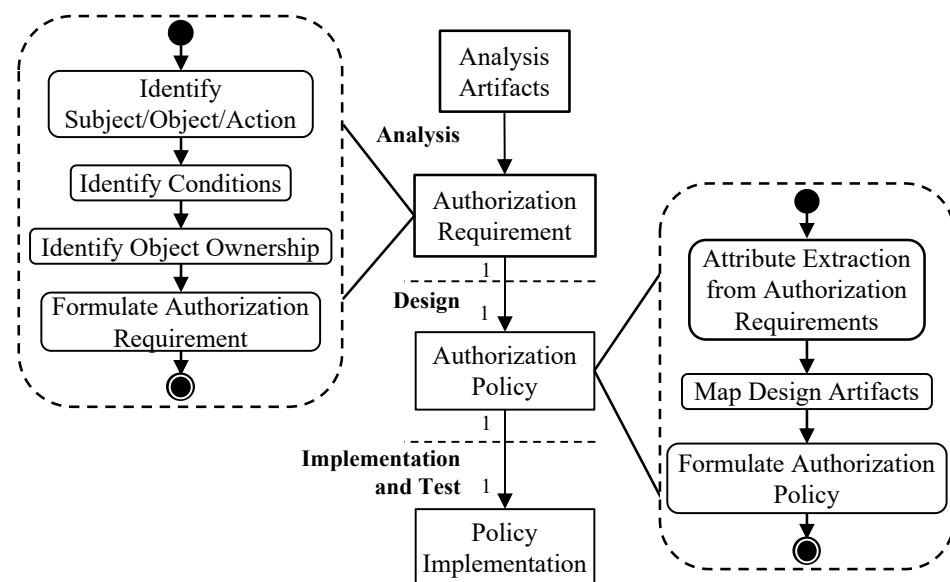


Figure 3. Development of authorization artifacts.

3.4.1. Analysis

The left side of Figure 3 shows the derivation process to create an authorization requirement. The basis for authorization requirements is the analysis artifacts of the software engineering approach. The quality and completeness of the authorization requirements strongly depend on the information present in the analysis artifacts. The first step is to identify the required aspects of ABAC. As described in Section 3.2, the subject, object, and action need to be identified. Second, the conditions for the access request must be defined. This includes conditions related to the subject, object, or the environment, which must be identified from the analysis artifacts. Third, the ownership of the object must be identified; while this is a condition itself, ownership is an important attribute for managing the access to an object [29]. Thus, developers should determine whether a single user or a large group is accessing an object. Depending on this, design decisions about the placement of attributes must be made in the design phase. Finally, the authorization requirement can be formulated by using the ABNF for an authorization requirement.

3.4.2. Design

Once the authorization requirements have been created, authorization aspects must be incorporated into the design of the application to maintain the overall structure. The right side of Figure 3 illustrates the tasks for creating an authorization policy. For each authorization requirement created in the analysis phase, one authorization policy is created. This allows us to maintain structure throughout the development. Further, if an authorization requirement changes, the authorization policy and the policy implementation can consequently be changed. Before the authorization policies can be formulated, the attributes are extracted from the authorization requirements, as also proposed in [20,38]. The conditions of the authorization requirements contain the names of the attributes. For example, the condition *age of subject is over 13* implies the subject attribute age. The condition *subject is assigned to project* implies an attribute assignee for an object project. The extraction of attributes based on the authorization requirements is individual and depends on the formulation of these authorization requirements. The second step is to map the authorization requirements to the design artifacts of a microservice. This step is necessary because the names may vary between the analysis phase and design phase, e.g., due to different stakeholders responsible for each phase. For example, the class diagram or the API specification may contain the attribute *projectAssignee*, while it is only named *assignee* in the authorization requirement. Furthermore, the names of objects may vary between the phases. For example, an object is called *book* in the analysis, is called *BookEntity* in a class diagram, and is accessible through the API endpoint */books/{id}*. Finally, the authorization policy can be formulated by applying the ABNF for each component of the authorization requirement.

We propose the use of support tables to store data (e.g., attributes and examples) required for the formulation of authorization requirements. The tables provide an overview for all participating developers. A list of attributes is also proposed by Brossard et al. [20], who introduced a single table including a short name, a namespace, a category, a data type, and a value range. However, compared to Brossard et al., we propose the use of one support table for each attribute type (i.e., subject, object, and environment). The format for the object support table can be found in Table 2 and contains four rows: The first is object, which contains the name of the object as found in the design artifacts (e.g., API specification) of the microservice. The second is the path to the REST endpoint of the microservice providing the object. The third is the name of the attribute as specified in the design artifacts. The fourth is example values for the attribute. The attributes for subjects and the environment are stored in separate tables. The tables contain the name of an attribute and an example value to provide information to developers.

Table 2. Structure of the object support table.

Object	Path	Attribute	Example
Projects	/projects	-	-
Project	/project/{id}	assignee	subject@mail.com
Book	/book/{id}	rating	12
...

3.4.3. Implementation and Deployment

Any authorization policy can consequently be implemented in an authorization policy language. As described in Section 3.3, we propose an implementation using the policy language Rego. To deploy the microservice with ABAC, the deployment architecture needs to be extended with the necessary logical components to enforce authorization decisions. Figure 4 provides a software architecture that includes the necessary logical ABAC components. The PEP communicates with the PDP, which requests the necessary attribute data from the PIP. The PEP enforces the decision and forwards the request to the microservice. The PAP provides the necessary policies to the PDP. The aspect of a software

architecture for a microservice-based application requires additional work in the future. This affects the number of PEPs, PDPs, or PIPs present in the architecture, which can affect the overall scalability or resource consumption.

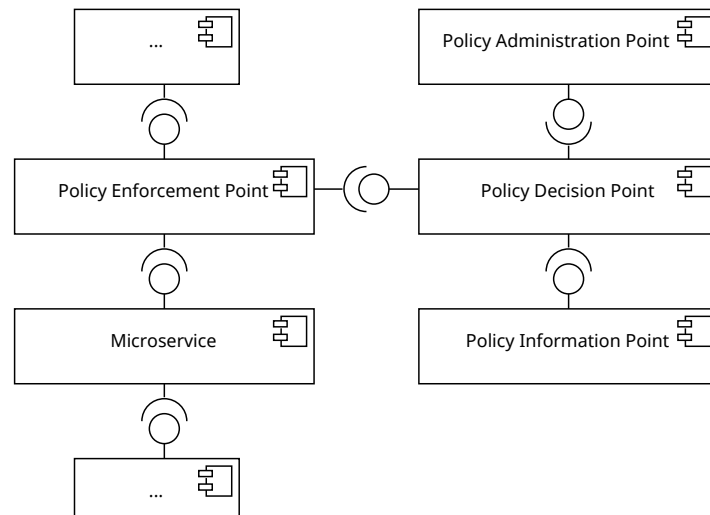


Figure 4. Exemplary software architecture including logical authorization components.

4. Case Study: FleetManagement

To validate the use of the authorization artifacts and the introduced extension of the development process, we perform a case study [50]. In our case study, we develop a connected car application that provides several business functionalities. One of the business functionalities is the management of fleets, which is selected as an example to describe the systematic implementation of authorization in this section. The targeted architecture of the connected car application is a microservice architecture. The business functionality management of fleets is provided by the microservice FleetManagement.

4.1. Analysis

For the requirements analysis, we apply use cases as primary analysis artifacts to formulate user requirements. Alternatively, other types of requirement analysis artifacts can be utilized, provided they contain the necessary information for authorization. The functionalities of the microservice FleetManagement are shown in the use case diagram presented in Figure 5. In total, there are two actors, the FleetAdministrator and the FleetManager, and four use cases. The FleetAdministrator can create a new fleet for a set of existing fleets. The FleetManager can view an overview of the fleets they are affiliated with. If the FleetManager has an affiliated fleet, they can view an overview of the cars within a fleet. A car can be deleted from a fleet by the FleetManager. The use case *View Fleets* is shown in Listing 9. An additional condition is that the request must be performed from the same location at which the fleet is located (line 6). The use case does not include a post-condition relevant for the authorization (line 8). The main flow in line 10 is omitted. However, the actors requests to view all fleets (step 1). Consequently, the system will search for the fleets belonging to the actor (step 2) and present the actor with the results (step 3).

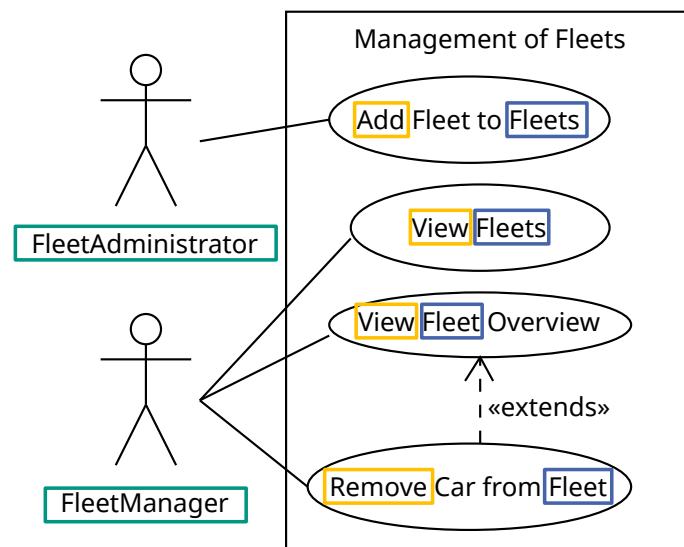


Figure 5. Use case diagram for the system FleetManagement.

Listing 9. Description of the “View Fleets” use case.

-
- 1 Title: View Fleets
 - 2 Primary Actors: FleetManager
 - 3 Secondary Actors: –
 - 4 Preconditions:
 - 5 – **Actor can only see fleets they manage**
 - 6 – **Request is performed in same location as fleet location**
 - 7 Postconditions:
 - 8 – ...
 - 9 Main Flow:
 - 10 ...
-

For each use case shown in the use case diagram in Figure 5, an authorization requirement is created in Listing 10. The primary objects, marked in blue, are a fleet and a set of fleets called fleets. The actions, marked in yellow, can be identified as add, view, and remove. The primary actors are marked in green. The conditions are either that the subject is a FleetAdministrator (line 1) or that the subject is the manager of the fleet on which it performs an action on (lines 2 through 4). For the use case *View Fleets*, only the fleets from the request’s location should be returned.

Listing 10. Authorization requirements of the case study.

-
- 1 **AuthZReq-10:** A subject can perform action Add on object Fleets IF the subject is FleetAdministrator
 - 2 **AuthZReq-20:** A subject can perform action View on object Fleets IF the subject is FleetManager for this Fleet AND location is same as Fleet location
 - 3 **AuthZReq-30:** A subject can perform action View on object Fleet IF the subject is FleetManager for this Fleet
 - 4 **AuthZReq-40:** A subject can perform action Remove on object Fleet IF the subject is FleetManager for this Fleet
-

4.2. Design

The class diagram for the microservice FleetManagement is presented in Figure 6. The FleetCollection contains a set of fleets. Further, it contains functions for the use cases *Add Fleet to Fleets*, *View Fleets*, and *View Fleet Overview*. The fleet contains an identifier and

attributes for a FleetManager, a location, and a set of cars which are managed by the fleet. The fleet also contains a function to remove a for the use case *Remove Car from Fleet*.

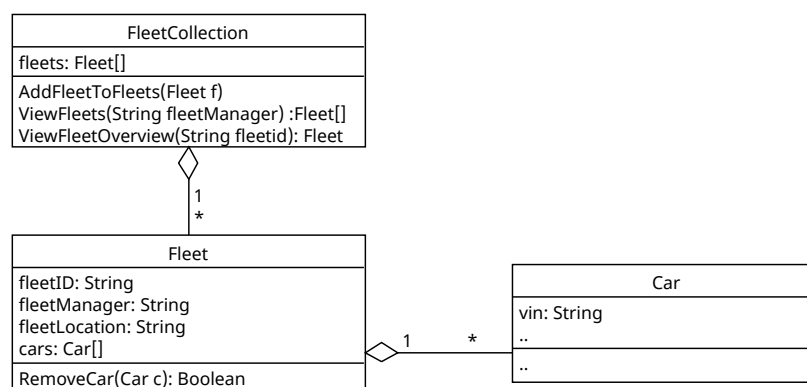


Figure 6. Class diagram for the microservice FleetManagement.

Following the process extension introduced in Section 3.4, the attributes are first extracted from the authorization requirements before the authorization policies are created. The support tables are used to store the extracted subject and object attributes and provide a coherent overview of attributes among developers.

The object support table for the case study is presented in Table 3. The identified objects from the authorization requirements are fleets and fleet which are called *FleetsCollection* and *Fleet*, respectively, in the class diagram. The object *FleetCollection* does not have an attribute that is required for the authorization. The object *Fleet* has an attribute *fleetManager* that holds an identifier of a subject. Further, the attribute location is called *fleetLocation* in the class diagram. The paths for the objects *FleetCollection* and *Fleet* are derived from the API specification and are added as */fleets* and */fleets/{fleetID}*, respectively.

Table 3. Object support table.

Object	Path	Attribute	Example
FleetCollection	/fleets	-	-
Fleet	/fleets/{fleetID}	fleetManager	subject@mail.com
		fleetLocation	Germany

Table 4 presents the subject support table, which contains the attributes required by the authorization requirements. Since authorization requirements mandate a check to determine if a subject is a *FleetManager* for a fleet (e.g., AuthZReq-30), it is imperative to accurately identify the subject. There are several options available to access the attributes of a subject, which depend on the employed technologies. During the design of the case study, a decision was made to utilize JWTs provided by an Identity and Access Management (IAM). The JWT contains an identifier referred to as *sub*, which corresponds to the subject's email address. As a result, the attribute *sub* has been incorporated into the subject support table. Additionally, for illustrative purposes, an example email address has been included in the support table. Given that the use case *Add Fleet to Fleets* can only be executed by a *FleetAdministrator*, it becomes necessary to assign a role for subject identification. The JWT also features a field called *roles*, which can encompass a role designation for the *FleetAdministrator*. Proper configuration of the IAM system is essential to ensure the provision of the appropriate role. In the context of this case study, the role is denoted as *cs-fleetAdm*, and an entry for *roles* has been established within the subject support table. Considering that a subject can possess multiple roles, the attribute *roles* includes example values represented using array notation enclosed in square brackets (i.e., []). Alternatively,

an attribute such as *department* could be employed if all employees within a department are deemed fleet administrators.

Table 4. Subject support table.

Attribute	Example Values
sub	fleet@manager.com
roles	[cs-fleetAdm]
department	FleetDepartment

Utilizing the authorization requirements and the information located in the support tables, authorization policies can be formulated. The authorization policies for each authorization requirement are presented in Listing 11. The first authorization policy, shown in line 1, requires the subject attribute roles, but no other conditions. The second authorization policy aims to return a set of fleets to which the subject has access to while taking the location into account. For each fleet, the condition whether the subject is *fleetManager* is checked. The authorization policies in lines 3 and 4 also require that the *fleetManager* is equal to the subject's identifier in order to perform the respective operation.

Listing 11. Authorization policies of the case study.

- 1 **AuthZPolicy-10:** A subject with FleetAdministrator in subject.roles POST on object /fleets
- 2 **AuthZPolicy-20:** A subject can perform action GET on every object fleet in objects /fleets IF
fleet.fleetManager == subject.sub AND environment.loc == fleet.fleetLocation
- 3 **AuthZPolicy-30:** A subject can perform action GET on object /fleets/{fleetID} IF fleet.
fleetManager == subject.id
- 4 **AuthZPolicy-40:** A subject can perform action DELETE on object /fleets/{fleetID} IF fleet.
fleetManager == subject.id

4.3. Implementation

Based on the authorization policies in Listing 11, the authorization policies are implemented in the Rego policy language in Listing 12. Exactly one Rego policy is implemented for each natural language authorization policy. An authorization policy is represented by an allow statement. Within an allow statement, multiple statements must be evaluated to true for the policy to become true. In the policies presented in Listing 12, the default value for allow is false (see line 1). As described in Section 3.3, support functions are used to implement the policies by offloading common functionality. For example, the statement *actionIsPost* checks whether the HTTP operation is POST.

For the first authorization policy, only the role *FleetAdministrator* must be present in the JWT of the subject represented by the helper statement *subjectIsFleetAdministrator*. Regarding the second authorization policy, the microservice should return only the fleets that a subject actually owns. There are several options to implement such a behavior, depending on the degree of authorization externalization and attribute management. In the case of the policy presented in lines 7 through 12, an HTTP header containing the subject's identifier is returned. This is called partial evaluation and allows the actual microservice to perform the filtering and return the correct fleet objects (see [15]). Another option is to tell the microservice which objects to return by filtering with Rego within the OPA. This could potentially lead to higher latency and an increased complexity when evaluating more complex requests [46]. The final two authorization policies in lines 13 and 17 evaluate the HTTP operation and the manager of the fleet, which is provided by a helper statement (lines 21 through 25).

Listing 12. Authorization policies implemented in Rego.

```

1 default allow = false
2 allow { # AuthZPolicy-10
3     subjectIsFleetAdministrator
4     actionIsPost
5     objectIsFleets
6 }
7 allow { # AuthZPolicy-20
8     actionIsGet
9     objectIsFleets
10    object.fleetLocation == input.loc
11    response_headers_to_add["x-fleetManager"] := subject.id
12 }
13 allow { # AuthZPolicy-30
14    actionIsGet
15    subjectIsManagerOfFleet
16 }
17 allow { # AuthZPolicy-40
18    actionIsDelete
19    subjectIsManagerOfFleet
20 }
21 subjectIsManagerOfFleet {
22    fleetID := getFleetID(input)
23    fleet := data[fleetID]
24    subject.id == fleet.fleetManager
25 }
26 ..

```

4.4. Excursus: Deployment and Performance Evaluation

This section provides an example deployment architecture and an evaluation of performance through latency testing. However, we do not discuss the process of deploying a microservice-based application including the required ABAC components. A Continuous Integration and Continuous Deployment (CICD) pipeline can streamline the deployment process to a Kubernetes cluster [51]. Given that Kubernetes describes deployments in manifest files, typically in YAML format, the necessary manifests can be generated using Helm charts within the pipeline [52]. This approach enables the creation of a single deployment configuration that remains adaptable to multiple microservices. Subsequently, the CICD pipeline can deploy the Helm chart to the cluster using Helm (i.e., “helm install”). Additionally, any implemented authorization policy can be automatically copied to OPA by the CICD pipeline. Leveraging a CICD pipeline alongside Helm charts effectively simplifies the complexity of deployment and configuration, facilitating the swift rollout of new changes, including updates to authorization policies.

The nodes presented in Figure 7 can be deployed in a Kubernetes cluster within a single Kubernetes pod. This can be executed using the sidecar pattern [53]. The microservice FleetManagement runs as the main container, and its functionality is extended by the sidecars Envoy and Open Policy Agent. The proxy Envoy acts as PEP and provides an authorization plugin that can route HTTP requests [54]. Envoy forwards the request to the PDP Open Policy Agent which holds the authorization policies from Listing 12. The OPA evaluates the request and returns the result to Envoy. Depending on the result, the request is routed to the microservice FleetManagement. If the result is positive, Envoy can add additional headers to the request if necessary such as in the authorization policy in line 11 of Listing 12. The final node is the load generator k6, which allows writing load tests using JavaScript.

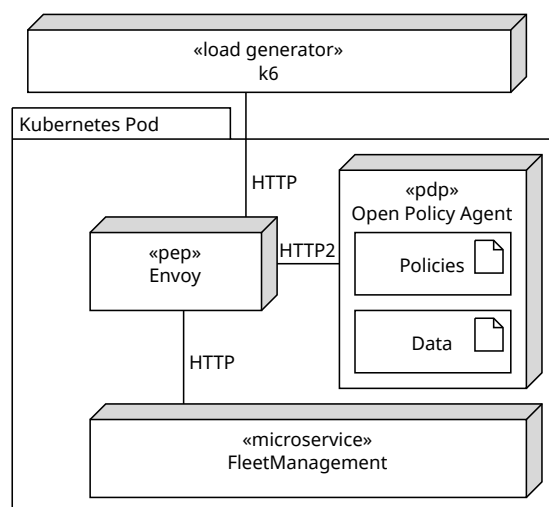


Figure 7. Case study deployment.

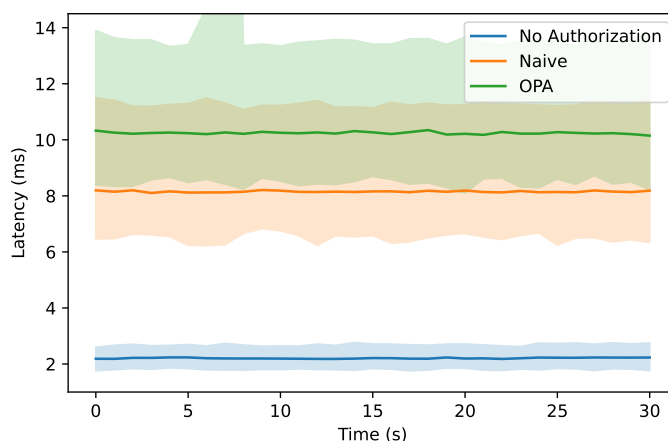
To evaluate the impact of authorization on the latency of the microservice FleetManagement, three implementations were compared. First, as a baseline, the microservice FleetManagement was implemented without authorization. Second, a naive implementation of authorization within the code of the microservice FleetManagement was executed. Third, fine-grained authorization using envoy and OPA was implemented. For load tests, the use case *View Fleet Overview* was used. The functionality for the load test was retrieved via the REST endpoints *GET /fleets/{fleet-id}*, respectively. The microservice FleetManagement was implemented in Golang. A PostgreSQL database was employed to store the data required by the microservice and comprises two tables. The first table contains information about the fleet, including a fleet ID as the primary key, along with the fleet manager. The second table is dedicated to storing car data, encompassing attributes such as an ID (primary key), a Vehicle Identification Number (VIN), the brand, and a reference to the corresponding fleet through its fleet ID (i.e., foreign key). A relational database was selected to model the relation between a car and a fleet. Other options, such as MongoDB, would also have been possible. However, the impacts of the database on the overall results should be comparable. The same database was utilized across all implementations.

The load tests present a synthetic load that is intended to produce comparable results for the different implementations. The goal is not to test the maximum throughput of the microservice but rather to capture the median latency for an authorization implementation. To provide comparable results, each load test contained the same configuration and lasted for 30 seconds. The target rate was set to 100 requests per second. The resources were selected according to the synthetic load applied to them. To ensure comparability, each component of the load test environment, including the load generator, was run in a container with a fixed set of resources within a Kubernetes cluster presented in Table 5. Throughout the load tests, the containers were assigned to the same Kubernetes node to ensure consistent latency. The PostgreSQL database (v15.2) has four CPU cores and 20 GB of RAM. The microservice FleetManagement has three CPU cores and 5 GB of RAM. The Envoy proxy (v1.23) has 2.5 CPU cores and 5 GB of RAM. OPA (v0.51.0) has 2.5 CPU cores and 5 GB of RAM. Finally, to avoid bottlenecks at the load generator k6 (v0.43.1), six CPU cores and 20 GB of RAM were allocated. During the load tests, the load on the allocated resources was monitored and found to be sufficient to perform 100 requests per second without running into a resource bottleneck. The database is populated with 2500 users for the experiment. Each user is responsible for four fleets, for a total of 10,000 fleets. Each fleet contains four cars. For the baseline implementation of the microservice FleetManagement, the database contains only four fleets with four cars each. Otherwise, the microservice would return all 10,000 fleets, resulting in incomparable results.

Table 5. Parameters for load testing components.

Component	CPU	RAM	Version
FleetManagement	3	5 GB	-
PostgreSQL	4	20 GB	15.2
Envoy	2.5	5 GB	1.23
OPA	2.5	5 GB	0.51.0
k6	6	20 GB	0.43.1

The results of the load tests are shown in Figure 8. The x-axis represents the elapsed time of the load tests in seconds. The y-axis represents the response time of the load tests in milliseconds. For each authorization architecture, the load tests were run 25 times and a line was plotted showing the median response time for that time frame. The median was chosen to remove outliers with a very high or low response time. The hue around each median response time represents the 90% quantile (Q90) of the response latency. Table 6 complements the results shown in Figure 8 with the 95% quantile (Q95) and the Requests Per Second (RPS). The baseline implementation (blue) without authorization has a median latency of 2.2 ms. The naive authorization (orange) has a median latency of 8.152 ms, which is a significant increase over the baseline implementation. The increase in latency can be explained by the larger amount of data that must be filtered by the microservice. In addition, the microservice must also validate a JWT to authorize the request. Compared to the naive implementation, the externalized authorization (green) adds ≈ 2 ms to the median latency. This increase is due to the additional components required for the authorization using an OPA.

**Figure 8.** Results of load tests performed on the microservice FleetManagement.**Table 6.** Results presenting response latency for executed load tests.

Implementation	Median	Q90	Q95	RPS
No Authorization	2.207 ms	2.579 ms	2.698 ms	100.001
Naive	8.152 ms	10.831 ms	11.280 ms	100.012
OPA	10.243 ms	12.934 ms	13.594 ms	100.010

k6 collects additional metrics, including the amount of data sent and received during the load tests. These metrics are presented in Table 7. For all implementations, $\approx 77,500$ requests were sent during the load tests. A difference occurs in the data sent by the different implementations. Since the baseline implementation does not require JWT tokens, the overall data sent is only 7.36 MB. In contrast, both the naive and OPA implementations transmit

26.89 MB and 26.96 MB, respectively. The difference in data sent between the naive implementation and OPA likely arises due to the random selection of JWT tokens which can have a slightly different size. The data received for the baseline and naive implementation total ≈ 62 MB. Compared to the other implementations, the OPA implementation returns an additional 4 MB of data. This is likely due to additional data (e.g., headers) added by the Envoy proxy. Throughout the load tests for both the naive and OPA implementations, every individual request undergoes successful authorization.

Table 7. Further metrics for executed load tests.

Implementation	Total HTTP Requests	Data Sent	Data Received
No Authorization	77501	7.36 MB	62.07 MB
Naive	77509	26.89 MB	62.25 MB
OPA	77508	26.96 MB	66.26 MB

5. Discussion

In this paper, we investigate user authorization in the development of microservice-based applications. To answer how ABAC policies can be systematically formulated (RQ1), we propose an ABNF to structure authorization policies. The methodology using the ABNF allows one to create authorization policies by structuring the aspects of ABAC to authorize a user request. This includes the subject, object, action, and the respective attributes. It provides developers with a natural language artifact. The authorization requirements are an intermediate technology-agnostic artifact. For the formulation of authorization requirements (RQ2), a syntax for authorization requirements is introduced to capture the aspects of authorization during the requirements analysis phase. The authorization requirements are formulated using an ABNF; while the process of deriving authorization requirements will be highly individual and depend on the individual stakeholders, the authorization requirement provides a structure among developers. Furthermore, using authorization requirements allows further derivation of an authorization policy. To systematically implement authorization policies (RQ3), we propose a systematic implementation by employing an OPA and its policy-as-code language Rego. Using the authorization policies defined with our ABNF, the content of an authorization implementation can be derived step by step. To answer the systematic integration of authorization into the development of a microservice-based application (RQ4), we propose an authorization extension to a generic development approach for a microservice-based application. The extension ranges from the analysis to the implementation and test phase and places the introduced authorization artifacts in the respective phases. In addition, a procedure for deriving the authorization artifacts is introduced.

By performing the introduced approach in a case study, we present that the approach is feasible by employing a UML use case diagram and use case descriptions. Furthermore, we presented a comparison between authorization implementations, which shows that externalizing the authorization logic from a microservice using an OPA and Envoy is a viable option. The overall increase in latency is acceptable and will not hinder the performance of microservice-based applications.

Compared to the previous work on authorization in microservice-based applications, which maintain a primarily technical focus on performing authorization, we provide a systematic approach to create authorization artifacts [14,15]. This approach can support developers in reducing the reported complexity of ABAC [28] while gaining a uniform understanding of what to authorize. The generation of policies based on UML models is an intriguing research area [40]. Nevertheless, to create high-quality models, developers must possess a firm understanding of what to authorize, beginning at the requirements analysis. Having a dedicated requirement authorization artifact will provide support and prevent the integration of authorization as an afterthought. With the emergence of advanced large language models such as ChatGPT, the formulation of authorization policies, as outlined

in the approach by [39], will likely become more sophisticated and dependable. However, given that authorization is a highly sensitive aspect in terms of security, the human factor (i.e., developer) is likely to remain prominent. This, in turn, underscores the necessity of possessing a sound understanding of what requires authorization.

5.1. Threats to Validity

In accordance with the research conducted by Wohlin et al. [50], we acknowledge and address the following threats to validity in relation to empirical software engineering and the implementation of our findings in our specific case study.

Construct Validity. The primary concern regarding the construction of the case study lies in the potential lack of overall complexity. There is a possibility that the provided example may be too small, thereby neglecting certain edge cases that were not considered during the creation of authorization requirements or authorization policies. This could impact the comprehensiveness and effectiveness of the derived authorization policies. In this case, the ABNF of the authorization requirements or authorization policies lacks completeness and the respective ABNF can be extended.

Internal Validity. Throughout the development of authorization policies, a strong dependency exists between the various authorization artifacts; while this interdependence is necessary for deriving effective authorization policies, an incorrect or poorly defined analysis artifact can result in insufficient policies, e.g., missing conditions. Hence, the experience and proficiency of developers in conducting thorough and precise requirements analysis pose a threat to internal validity.

External Validity. The successful application of authorization policies in an external context relies on the similarity of the development process structure. However, variations in the development process, including the creation of development artifacts, may exist. Consequently, certain artifacts required by the process may be absent, and the adoption of agile development methods like Scrum could potentially impact policy development. Although it is possible to adapt the process to different microservice development approaches, the derivation of authorization policies would not be straightforward and would necessitate modifications.

Reliability. The consistency of results when applying the authorization extension on a development process should ideally be independent of the individual developer. However, due to the presence of subjective decisions throughout the process (e.g., application structure and naming conventions for development artifacts), different developers may yield varying results. Nevertheless, the overall number of policies should remain constant, as the structure preservation principle ensures that each authorization requirement ultimately corresponds to one implemented authorization policy. However, the amount of authorization requirements will be subject to the individual developer, as the derivation of authorization requirements is not prescribed and depends on the requirements analysis.

5.2. Limitations

This paper assumes a structured development approach for a microservice-based application. The definition of analysis artifacts is a highly individual topic. Therefore, the derivation of authorization requirements is also highly individual, which will lead to different results among developers. The analysis artifacts must contain all necessary information regarding the authorization. Further, the proposed development process extension cannot provide a solution for every edge case. However, it provides a structure that can be extended by appropriate artifacts or derivations. Furthermore, the ABNF and Rego allow one to write policies of an arbitrary complexity. In its current state, the use of REST APIs limits the possible actions when creating authorization policies. Other API paradigms such as gRPC or GraphQL are not supported by the results of this

work. To use such APIs, changes to the authorization artifacts must be investigated and (possibly) performed.

The overall complexity of introducing attribute-based access control to a microservice-based application is increased compared to embedding authorization decisions into the code of the microservices. Thus, the question arises whether using ABAC is a suitable option. For small projects, the complexity of ABAC will most likely outweigh its benefits. When working in an enterprise project with a large team, the overall complexity will have a less significant effect on the overall workload.

In a real-world enterprise scenario, the complexity introduced by ABAC in a microservice architecture can present several challenges in operations; while the excursus presented only a small increase in added latency, the management of the overall architecture will likely increase due to the additional components. This will also impact factors such as scalability and maintenance, which are highly relevant in modern cloud environments.

5.3. Future Work

To further support developers using the approach presented in this paper, the automation of the processes should be investigated to reduce overall complexity. For instance, exploring the automated creation of authorization requirements derived from use case descriptions could be valuable. This would require natural language processing to identify aspects relevant for authorization such as subjects, objects, or conditions. Moreover, the generation of authorization policies based on authorization requirements should also be investigated since the structures of the ABNFs are similar. This can also include the creation of support tables. Lastly, we should delve into exploring Rego implementations generated from the structured authorization policies. This way, developers would not need an extensive comprehension of Rego.

To implement the proposed approach within an enterprise environment that already encompasses a range of established protocols, tools, and security practices, additional research is necessary. For instance, the utilization of subject attributes is likely to be contingent upon the existing IAM solutions, which concurrently serve as the means for authentication. In the case study, we employed JWT tokens issued by an OAuth provider. Thus, exploring the potential incorporation of other protocols, such as the widely employed Security Assertion Markup Language (SAML) in enterprise contexts, should be addressed in future research.

Additionally, we have identified three challenges to successfully integrate user authorization into the development of microservice-based applications. These challenges should be addressed in future work. First, in order to perform authorization based on ABAC, the attributes are essential and must be known. This is typically performed through the Policy Information Point. When using an Open Policy Agent as a PEP, access to the attribute data needed for authorization must be orchestrated. This raises several questions, such as where to store the attribute data, how to access the data, and how current the data need to be. In this context, the coupling between business logic and authorization logic must be investigated. Depending on the desired level of coupling, the involved components and the overall architecture may change. To distribute attribute changes in the architecture, additional components might be necessary. Second, there are several ways to integrate the required ABAC components into the deployment of a microservice-based application. For example, a single API proxy can be used as the policy enforcement point for all microservices. In contrast, the exemplary deployment presented in Figure 7 has a single PEP and PDP for each microservice deployment. This may have implications for scaling, as well as for the availability and topicality of attributes across multiple deployments. Thus, future work should explore different deployment architectures for authorization in microservice-based applications. Third, this publication focuses on user authorization. However, in a microservice architecture, requests between microservices also need to be authorized. This is especially important when implementing a zero-trust architecture which requires removing implicit trust among microservices [55]. Among other things, service-to-

service authorization requires specifying what resources a microservice can access and who owns those resources. In the context of ABAC, requests performed to the authorization components (e.g., PIP) must also be authorized.

6. Conclusions

The development of microservice-based applications and the use of cloud platforms to distribute them has become an established practice in modern software engineering. However, the integration of security mechanisms such as authentication and authorization remains a major challenge in the development of such applications.

In this paper, we propose a syntax for authorization requirements and authorization policies and a systematic, top-down process for the integration of authorization into the development of a microservice-based application. To enforce comprehensive user authorization, fine-grained authorization decisions must be made, for example, granting access to an object only to a specific user. To enforce such decisions, ABAC is used by the systematic process. ABAC and its required mechanisms fit into the distributed microservice architecture style by decoupling the necessary components. It enables the externalization of authorization for microservices by removing the authorization logic from the implementation of a microservice, which in turn reduces the microservice to providing its core business functionality. Additionally, externalization allows aspects of authorization to be changed without necessarily changing the logic of a microservice, thus providing greater flexibility. The authorization development spans all development phases and allows authorization policies to be derived from existing development artifacts. In the analysis phase, authorization requirements are created. Based on the authorization requirements, authorization policies are created in the design phase using support tables. Finally, in the implementation phase, the authorization policies are implemented in Rego. By doing this, we address the inherently complex problem of integrating fine-grained authorization into a microservice-based application. Our approach may provide a starting point for software developers to systematically address this topic to create reliable and secure software.

Author Contributions: Conceptualization, N.S. and S.A.; methodology, N.S.; software, N.S.; validation, N.S.; investigation, N.S.; data curation, N.S.; writing—original draft preparation, N.S.; writing—review and editing, N.S.; supervision, S.A.; project administration, S.A. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are openly available in KITopen at <https://doi.org/10.35097/1744>.

Acknowledgments: The authors would like to thank Rudy Ailabouni and David Boschert for the valuable discussions and the contributions to this work.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

ABAC	Attribute-Based Access Control
ABNF	Augmented Backus–Naur Form
API	Application Programming Interface
BNF	Backus–Naur Form
CICD	Continuous Integration Continuous Deployment

CNCF	Cloud Native Computing Foundation
CRLF	Carriage Return Line Feed
CRUD	Create Read Update Delete
IAM	Identity and Access Management
JWT	JSON Web Token
NLP	Natural Language Policy
OPA	Open Policy Agent
PAP	Policy Administration Point
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PIP	Policy Information Point
Q90	90% Quantile
Q95	95% Quantile
RBAC	Role Based Access Control
REST	Representational State Transfer
RPC	Remote Procedure Call
RPS	Requests Per Second
SAML	Security Assertion Markup Language
UML	Unified Modeling Language
VIN	Vehicle Identification Number
XACML	eXtensible Access Control Markup Language

References

1. Swoyer, M.; Loukides, S. Microservices Adoption in 2020. Available online: <https://www.oreilly.com/radar/microservices-adoption-in-2020/> (accessed on 20 June 2023).
2. Berardi, D.; Giallorenzo, S.; Mauro, J.; Melis, A.; Montesi, F.; Prandini, M. Microservice Security: A Systematic Literature Review. *PeerJ Comput. Sci.* **2022**, *7*, e779. <https://doi.org/10.7717/peerj-cs.779>.
3. solo.io. Microservices, Kubernetes and Istio—2022 Adoption Trends. Available online: <https://www.solo.io/resources/infographic/microservices-kubernetes-and-istio-2022-adoption-trends/pdf/> (accessed on 24 August 2023).
4. Newman, S. *Building Microservices: Designing Fine-Grained Systems*, 1st ed.; O'Reilly Media: Beijing, China; Sebastopol, CA, USA, 2015.
5. Fielding, R.T. Architectural Styles and the Design of Network-based Software Architectures. Ph.D. Thesis, University of California, Irvine, CA, USA, 2000.
6. Birrell, A.D.; Nelson, B.J. Implementing Remote Procedure Calls. *ACM Trans. Comput. Syst.* **1984**, *2*, 39–59.
7. Open API Initiative. Open API Specification—v3.1.0. Available online: <https://spec.openapis.org/oas/v3.1.0> (accessed on 24 August 2023).
8. Google LLC All. Protocol Buffers Documentation. Available online: <https://protobuf.dev/programming-guides/proto3/> (accessed on 24 August 2023).
9. Hippchen, B.; Giessler, P.; Steinegger, R.H.; Schneider, M.; Abeck, S. Designing Microservice-Based Applications by Using a Domain-Driven Design Approach. *Int. J. Adv. Softw.* **2017**, *10*, 432–445.
10. Sidler, J.; Braun, E.; Schmitt, C.; Schlachter, T.; Hagenmeyer, V. Microservice-Based Architecture for the Integration of Data Backends and Dashboard Applications in the Energy and Environment Domains. In *Advances and New Trends in Environmental Informatics*; Wohlgemuth, V., Naumann, S., Behrens, G., Arndt, H.K., Eds.; Springer International Publishing: Cham, Switzerland, 2022; pp. 37–48. https://doi.org/10.1007/978-3-030-88063-7_3.
11. OWASP Foundation. OWASP Top 10:2021. Available online: <https://owasp.org/Top10/> (accessed on 15 July 2023).
12. de Almeida, M.G.; Canedo, E.D. Authentication and Authorization in Microservices Architecture: A Systematic Literature Review. *Appl. Sci.* **2022**, *12*, 3023. <https://doi.org/10.3390/app12063023>.
13. Gollmann, D. Computer Security. *WIREs Comput. Stat.* **2010**, *2*, 544–554. <https://doi.org/10.1002/wics.106>.
14. Nehme, A.; Jesus, V.; Mahbub, K.; Abdallah, A. Fine-Grained Access Control for Microservices. In Proceedings of the 11th International Symposium (FPS 2018), Montreal, QC, Canada, 13–15 November 2018; Zincir-Heywood, N., Bonfante, G., Debbabi, M., Garcia-Alfaro, J., Eds.; Springer International Publishing: Cham, Switzerland, 2019; Volume 11358, pp. 285–300.
15. Sauwens, M.; Heydari Beni, E.; Jannes, K.; Lagaisse, B.; Joosen, W. ThunQ: A Distributed and Deep Authorization Middleware for Early and Lazy Policy Enforcement in Microservice Applications. In Proceedings of the 19th International Conference (ICSOC 2021), Virtual Event, 22–25 November 2021; Hacid, H., Kao, O., Mecella, M., Moha, N., Paik, H.Y., Eds.; Springer International Publishing: Cham, Switzerland, 2021; Volume 13121, pp. 204–220.
16. Yarygina, T.; Bagge, A.H. Overcoming Security Challenges in Microservice Architectures. In Proceedings of the 2018 IEEE Symposium on Service-Oriented System Engineering (SOSE), Bamberg, Germany, 26–29 March 2018; pp. 11–20. <https://doi.org/10.1109/SOSE.2018.00011>.

17. Devanbu, P.; Stubblebine, S. Software Engineering for Security: A Roadmap. In Proceedings of the Conference on the Future of Software Engineering, Limerick, Ireland, 4–11 June 2000; pp. 227–239.
18. Busch, M.; Koch, N.; Masi, M.; Pugliese, R.; Tiezzi, F. Towards Model-Driven Development of Access Control Policies for Web Applications. In Proceedings of the Workshop on Model-Driven Security, Innsbruck, Austria, 1–5 October 2012; pp. 1–6. <https://doi.org/10.1145/2422498.2422502>.
19. Zolotas, C.; Chatzidimitriou, K.C.; Symeonidis, A.L. RESTsec: A Low-Code Platform for Generating Secure by Design Enterprise Services. *Enterp. Inf. Syst.* **2018**, *12*, 1007–1033. <https://doi.org/10.1080/17517575.2018.1462403>.
20. Brossard, D.; Gebel, G.; Berg, M. A Systematic Approach to Implementing ABAC. In Proceedings of the 2nd ACM Workshop on Attribute-Based Access Control—ABAC '17, Scottsdale, AZ, USA, 24 March 2017; pp. 53–59. <https://doi.org/10.1145/3041048.3041051>.
21. RFC 7519; JSON Web Token (JWT). Internet Engineering Task Force, Fremont, CA, USA, 2015. Available online: <https://www.rfc-editor.org/rfc/rfc7519> (accessed on 15 June 2023). <https://doi.org/10.17487/RFC7519>.
22. Sandhu, R.; Samarati, P. Access Control: Principle and Practice. *IEEE Commun. Mag.* **1994**, *32*, 40–48. <https://doi.org/10.1109/35.312842>.
23. Kizza, J.M. Access Control and Authorization. In *Guide to Computer Network Security*; Springer: London, UK, 2015; pp. 185–204. https://doi.org/10.1007/978-1-4471-6654-2_9.
24. Goyal, V.; Pandey, O.; Sahai, A.; Waters, B. Attribute-Based Encryption for Fine-Grained Access Control of Encrypted Data. In Proceedings of the 13th ACM Conference on Computer and Communications Security, Alexandria, VA, USA, 30 October 2006; pp. 89–98. <https://doi.org/10.1145/1180405.1180418>.
25. Ghotbi, S.H.; Fischer, B. Fine-Grained Role- and Attribute-Based Access Control for Web Applications. In *Software and Data Technologies*; Cordeiro, J.; Hammoudi, S.; van Sinderen, M., Eds.; Springer: Berlin/Heidelberg, Germany, 2013; Volume 411, pp. 171–187. https://doi.org/10.1007/978-3-642-45404-2_12.
26. Sandhu, R.; Coyne, E.; Feinstein, H.; Youman, C. Role-Based Access Control Models. *Computer* **1996**, *29*, 38–47. <https://doi.org/10.1109/2.485845>.
27. Elliott, A.; Knight, S. Role Explosion: Acknowledging the Problem. In Proceedings of the 2010 International Conference on Software Engineering Research & Practice (SERP 2010), Las Vegas, NE, USA, 12–15 July 2010; pp. 349–355.
28. Aftab, M.U.; Qin, Z.; Zakria, Ali, S.; Pirah; Khan, J. The Evaluation and Comparative Analysis of Role Based Access Control and Attribute Based Access Control Model. In Proceedings of the 2018 15th International Computer Conference on Wavelet Active Media Technology and Information Processing (ICCWAMTIP), Chengdu, China, 14–16 December 2018; pp. 35–39. <https://doi.org/10.1109/ICCWAMTIP.2018.8632578>.
29. Hu, V.C.; Ferraiolo, D.; Kuhn, R.; Schnitzer, A.; Sandlin, K.; Miller, R.; Scarfone, K. *Guide to Attribute Based Access Control (ABAC) Definition and Considerations*; Technical Report NIST SP 800-162; National Institute of Standards and Technology: Gaithersburg, MD, USA, 2014. <https://doi.org/10.6028/NIST.SP.800-162>.
30. Yuan, E.; Tong, J. Attributed Based Access Control (ABAC) for Web Services. In Proceedings of the IEEE International Conference on Web Services (ICWS'05), Orlando, FL, USA, 11–15 July 2005; p. 569. <https://doi.org/10.1109/ICWS.2005.25>.
31. eXtensible Access Control Markup Language (XACML) Version 3.0. OASIS Standard. 22 January 2013. Available online: <http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html> (accessed on 20 June 2023).
32. RFC 5234; Augmented BNF for Syntax Specifications: ABNF. Internet Engineering Task Force, Fremont, CA, USA, 2008. Available online: <https://www.rfc-editor.org/rfc/rfc5234.html> (accessed on 22 March 2023). <https://doi.org/10.17487/RFC5234>.
33. RFC 2616; Hypertext Transfer Protocol—HTTP/1.1. Internet Engineering Task Force, Fremont, CA, USA, 1999. Available online: <https://www.rfc-editor.org/rfc/rfc2616?data1=dwnsb4B&data2=abmurltv2b> (accessed on 20 July 2023). <https://doi.org/10.17487/RFC2616>.
34. RFC 6749; The OAuth 2.0 Authorization Framework. Internet Engineering Task Force, Fremont, CA, USA, 2012. Available online: <https://www.rfc-editor.org/rfc/rfc6749> (accessed on 15 June 2023). <https://doi.org/10.17487/RFC6749>.
35. Chandramouli, R. *Security Strategies for Microservices-Based Application Systems*; Technical Report NIST SP 800-204; National Institute of Standards and Technology, Gaithersburg, MD, USA, 2019. <https://doi.org/10.6028/NIST.SP.800-204>.
36. Banati, A.; Kail, E.; Karoczkai, K.; Kozlovsky, M. Authentication and Authorization Orchestrator for Microservice-Based Software Architectures. In Proceedings of the 2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), Opatija, Croatia, 21–25 May 2018; pp. 1180–1184. <https://doi.org/10.23919/MIPRO.2018.8400214>.
37. Das, S.; Mitra, B.; Atluri, V.; Vaidya, J.; Sural, S. Policy Engineering in RBAC and ABAC. In *From Database to Cyber Security*; Samarati, P., Ray, I., Ray, I., Eds.; Springer International Publishing: Cham, Switzerland, 2018; Volume 11170, pp. 24–54. https://doi.org/10.1007/978-3-030-04834-1_2.
38. Alohal, M.; Takabi, H.; Blanco, E. Automated Extraction of Attributes from Natural Language Attribute-Based Access Control (ABAC) Policies. *Cybersecurity* **2019**, *2*, 2. <https://doi.org/10.1186/s42400-018-0019-2>.
39. Narouei, M.; Khanpour, H.; Takabi, H.; Parde, N.; Nielsen, R. Towards a Top-down Policy Engineering Framework for Attribute-based Access Control. In Proceedings of the 22nd ACM on Symposium on Access Control Models and Technologies, Indianapolis, IN, USA, 21–23 June 2017; pp. 103–114. <https://doi.org/10.1145/3078861.3078874>.

40. Fatemian, A.; Zamani, B.; Masoumi, M.; Kamranpour, M.; Ladani, B.T.; Rahimi, S.K. Automatic Generation of XACML Code Using Model-Driven Approach. In Proceedings of the 2021 11th International Conference on Computer Engineering and Knowledge (ICCKE), Mashhad, Iran, 28–29 October 2021; pp. 206–211. <https://doi.org/10.1109/ICCKE54056.2021.9721518>.
41. Talukdar, T.; Batra, G.; Vaidya, J.; Atluri, V.; Sural, S. Efficient Bottom-Up Mining of Attribute Based Access Control Policies. In Proceedings of the 2017 IEEE 3rd International Conference on Collaboration and Internet Computing (CIC), San Jose, CA, USA, 15–17 October 2017; pp. 339–348. <https://doi.org/10.1109/CIC.2017.00051>.
42. Lethbridge, T.C.; Laganier, R. *Object-Oriented Software Engineering*; McGraw-Hill: New York, NY, USA, 2005; Volume 11.
43. Cockburn, A. *Writing Effective Use Cases*; Pearson Education India: Noida, India, 1999.
44. Firesmith, D. Engineering Security Requirements. *J. Object Technol.* **2003**, *2*, 53. <https://doi.org/10.5381/jot.2003.2.1.c6>.
45. Cloud Native Computing Foundation. Open Policy Agent (OPA). Available online: <https://www.cncf.io/projects/open-policy-agent-opa/> (accessed on 16 March 2023).
46. Cloud Native Computing Foundation. Open Policy Agent: Documentation. Available online: <https://www.openpolicyagent.org/docs/latest/> (accessed on 16 March 2023).
47. Envoy Project. Envoy Documentation: What Is Envoy? Available online: https://www.envoyproxy.io/docs/envoy/latest/intro/what_is_envoy (accessed on 24 April 2023).
48. Traefik Enterprise Middleware: OPA—Traefik Enterprise. Available online: <https://doc.traefik.io/traefik-enterprise/middlewares/opa/> (accessed on 4 July 2023).
49. Schneider, M.; Zieschinski, S.; Klechorov, H.; Brosch, L.; Schorsten, P.; Abeck, S.; Urbaczek, C. A Test Concept for the Development of Microservice-based Applications. In Proceedings of the The Sixteenth International Conference on Software Engineering Advances (IARIA), Barcelona, Spain, 3–7 October 2021; pp. 88–97.
50. Wohlin, C.; Runeson, P.; Höst, M.; Ohlsson, M.C.; Regnell, B.; Wesslén, A. *Experimentation in Software Engineering*; Springer: Berlin/Heidelberg, Germany, 2012. <https://doi.org/10.1007/978-3-642-29044-2>.
51. Throner, S.; Hutter, H.; Sanger, N.; Schneider, M.; Hanselmann, S.; Petrovic, P.; Abeck, S. An Advanced DevOps Environment for Microservice-based Applications. In Proceedings of the 2021 IEEE International Conference on Service-Oriented System Engineering (SOSE), Oxford, UK, 23–26 August 2021; pp. 134–143. <https://doi.org/10.1109/SOSE52839.2021.00020>.
52. Cloud Native Computing Foundation. Helm Documentation. Available online: <https://helm.sh/docs/> (accessed on 24 August 2023).
53. Burns, B.; Oppenheimer, D. Design Patterns for Container-Based Distributed Systems. In Proceedings of the 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16), Denver, CO, USA, 20–21 June 2016.
54. Envoy Project. Envoy Documentation: HTTP Filters—External Authorization. Available online: https://www.envoyproxy.io/docs/envoy/v1.26.3/api-v3/extensions/filters/network/ext_authz/v3/ext_authz.proto, (accessed on 29 March 2023).
55. Teerakanok, S.; Uehara, T.; Inomata, A. Migrating to Zero Trust Architecture: Reviews and Challenges. *Secur. Commun. Netw.* **2021**, *2021*, 9947347. <https://doi.org/10.1155/2021/9947347>.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.