Tenant Separation on a multi-tenant microservice platform

Thesis for the degree of Master of Science in electrical engineering

Axel Sandqvist

August 8, 2023

Abstract

Axis Communications wishes to investigate their PaaS system, Axis Connected Services(ACX), with regard to separation of the tenants of the platform to ensure the implemented separation technologies are used correctly and to find out whether more separation is necessary. ACX ties together several previously separate services under a single umbrella, with the goal of improving usability and increasing inter-service functionalities and centralisation of the software products Axis has developed for their devices.

This thesis investigates alternative tenant separation technologies especially for data at rest and access management but also for data in use. The different technologies for at rest separation are logical separation, separate schema, separate encryption and separate database. For access management 6 technologies are presented; the three models access control list(ACL), role based access control(RBAC) and attribute based access control(ABAC), and also three specifically multitenant technologies for access management; Secure logical isolation for multitenancy(SLIM), Obejct tag access control strategy (OTACS) and Key insulated attribute based data retrieval scheme with keyword search (KI-ABDR-KS). Data in use separation technologies are shared instances, division of processing, VM separation and server separation. The technologies above and ACX's implementation are analysed and compared to arrive at a resulting proposition for the tenancy separation and access management solutions for ACX.

The investigation found that as ACX contained minimal sensitive information, separate database and encryption are too complex and costly to be worth the increased confidentiality, and separate schema is not an increase in separation compared to a well implemented logical separation solution. Access management is too decentralised and opaque in access enforcement, thus centralisation of access evaluation through a policy agent is proposed. To enforce tenant separation during sessions, the tenant identifier is also added as a parameter of the session to increase the distinction between tenant contexts.

In conclusion, the chosen technologies for data at rest, data in use and access management, being logical, shared instances and RBAC, are good choices for the system. The chosen technologies are mainly kept however the logical separation of data can be improved, and access control enforcement should be centralised with a policy agent.

Acknowledgement

This project was performed by Axel Sandqvist during the autumn and spring of 2022/2023, at Axis Communications headquarters in Lund with the purpose of investigating the current tenant separation of the Axis Connected Services PaaS, and suggesting other technologies for reducing risk and effect of attacks and data leaks.

The thesis was supervised by employees of Lund university faculty of engineering, and Axis Communications thanks to whom this thesis was possible.
Axis: Per-Anders Söderqvist, Olle Palmgren, Joakim Karlsson, Johan Jacobsson
LTH: Christian Gehrmann

Contents

Acronyms

- ABAC - Attribute Based Access Control
- AC - Access Control
- ACAP - Axis Camera Application Platform
- ACL - Access Control List
- ACX - Axis Connected Services
- ADMX - Axis Device Management Extend
- API - Application Programming Interface
- AT - Access Token
- AWS - Amazon Web Services
- CRUD Create, Read, Update, Delete
- CSP - Cloud Service Provider
- CSP - Content Security Policy
- CU - Callable Unit
- DB - Database
- DI - Device Inventory
- DM - Device Management
- IAM - Identity and Access Management
- IdP - Identity Provider
- JSON - JavaScript Object Notation
- JWT - JSON Web Token
- KI-ABDR-KS - Key Insulated Attribute Based Data Retrieval Scheme with Keyword Search
- KMS - Key Management Service
- LC - Login Cookie
- LS - Login Service
- MIDP - Machine IdP
- OIDC - OpenID Connect
- orgId - Organisational Identifier
- OTACS - Object Tag Access Control Strategy
- PA - Policy Agent
- PaaS - Platform as a Service

- RBAC - Role Based Access Control
- RDS - Relational Database Service
- RG - Resource Group
- RLS - Row level Security
- SaaS - Software as a Service
- SC - Session Cookie
- SLA - Service Level Agreement
- SLIM - Secure Logical Isolation for Multitenancy
- SoD - Separation of Duties
- SSS - Security Scope Service
- TLS - Transport Layer Security
- TM - Tenant Manager
- TTL - Time To Live
- Tzr - Tokenizer
- UUID - Universally Unique Identifier
- VM - Virtual Machine
- VMM - Virtual Machine Manager/Monitor, AKA Hypervisor

1 Introduction

Software architectures are complex structures which consist of smaller elements of direct functionality, their properties and connections to peers [1]. Due to its applicability, all larger organisations have to consider it for continued prosperity. Cloud software systems have a very advantageous feature in an improved ability to share resources, which is one of the key reasons for its market growth [2]. In the cloud there are three fundamental services to sell to customers, all of which under the umbrella of the term CSP or Cloud Service Provider; Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service (IaaS) [3][4][5]. The terminologies are not clearly defined as borders between these are not determined although they in general have some key properties. SaaS is a system with all background processing left up to the service's CSP, where the selling point for the creator of the SaaS is a program for the user to reach and use for processes like computation, modelling, content delivery, etc.. A platform is the underlying framework for the SaaS, creating a homogeneous environment for the service. The platform still has an underlying CSP that provides the means for a stable system yet has a greater responsibility for the configurations of the service. IaaS is the hardware and the underlying network of internet connections, data processing and storage [5]. Due to its many uses and versatility, larger systems quickly become intricate and difficult to oversee, and also so unique that finding optimal properties has no simple solution but must instead be found for each case. The aspects to take into account are the userbase, what data is stored, average and peak workloads, geographical spread, service level agreements (SLA), etc.. When considering security the main issues become confidentiality, access control, encryption, resilience, load balancing and data sharing [6].

Most cloud services subscribe to multitenancy, meaning the sharing of resources between unaffiliated users [7], here called tenants. Tenants are groups of users that share certain aspects like organization, billing, owned data and possibly other factors [7][8]. Application instances running on servers may serve several users simultaneously, and by allowing different tenants on the same application instance, it becomes multitenant. The advantage of hosting several tenants using the same resources is the compaction of resources, meaning higher efficiency and lower costs [1][3][8]. The cost of multitenancy's benefits is the increased risk of lost privacy since privileged data passes through shared nodes where there is a possibility of leakage between tenants, to attackers or to the platform owner [9]. This can happen accidentally through malfunctioning separation technology or intentionally through deliberate attacks on vulnerable resources. The principal factors to take into account when designing a multitenant system are to maximise the isolation between tenants while also maximising the use of the resources allocated to the system [1]. Figure 1 shows a general multitenant architecture with two services both serving two separate tenants. The system in the figure uses logical separation with row-level security (RLS) for data at rest separation. RLS means adding metadata to all entries in the DB which specify the tenant the data belongs to, this is described further in 3.8.1.

1.1 The ACX platform

Axis communications is a video surveillance company based in Lund with more than 4.000 employees which until recently mainly developed network cameras, however now also develops several other security-related systems including the system in focus for this thesis. The system is Axis Connected Services (ACX), a PaaS mostly deployed in AWS and some in Azure. AWS is ACX's CSP, and as such is responsible for the up-time and secure implementation of the infrastructure ACX is built upon. Implemented on the instances bought from AWS is the ACX PaaS microservice architecture with many small instances which perform small and specific tasks like managing job queues, creating and validating certificates, session authentication, performing operations on devices through the cloud and more, dividing the system into smaller and more easily handled modules. ACX ties most microservices Axis provides for their users together on

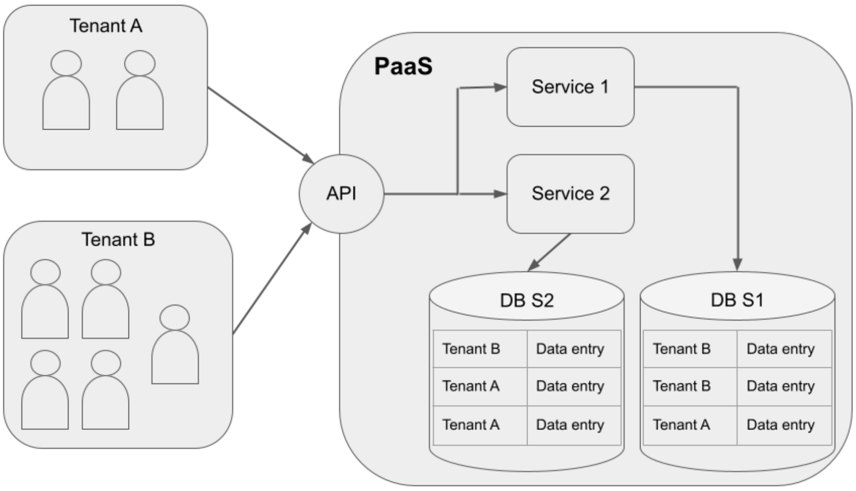


Figure 1: Example of multitenant architecture

the same platform with the goal of standardised protocols, policies and a controlled environment for all software services Axis has for their devices. The platform works as a foundation for core services like device management but also for additional value adding services like video management services and IoT applications. ACX provides ease of integration and diverse functionalities for both Axis and its partners. The prevalent features of the platform are universal identity and access management, device onboarding, device management and remote access.

When an entity wishes to use the platform, they first call the ACX API, which redirects to the Identity and Access Management (IAM) services for authorisation and then the intended service that performs the requested jobs, e.g the Device Management(DM) bundle of services. The service receives the request, possibly also communicating with other services if some information/part of the requested job is not with this service, then sends the response back through the API which then sends it to the entity. The architecture is developed with independence in focus, where most services contain some database (DB) with the data necessary to perform its tasks, meaning minimal communication between nodes on ACX as it is stored in the DB and remains within the service unless specifically requested from authorised parties.

The motivations behind maintaining multitenancy for ACX are that the services it provides are non-specific to tenants, so no functionalities are lost by sharing resources. Furthermore Axis can in this way take responsibility for the QoS of interactions between the devices and users, it is cheaper to share resources between tenants and software updates are simpler on fewer instances. The alternatives to multitenancy either mean more costs for Axis as it would require more servers and infrastructure, or it means more responsibility for the customers to implement parts of solutions that now are parts of ACX. Although by owning the servers Axis would control all non-transport infrastructure meaning greater assurances of the safety of the platform, AWS is trusted by many and is therefore a reasonably safe choice for Axis as well to use.

Axis has numerous partners whom provide services for camera systems as well, and ACX integrates the partners' platforms at the level they are working on, whether they have an independent app and UI, their own camera software, video monitoring systems, etc.. Depending on what kind of partner they receive different certificates and identities on the platform which correspond to the necessary access points to integrate their systems with ACX.

Figure 2 illustrates the steps an operator of surveillance cameras would administer their equipment using ACX. The list follows the operator's steps while the figure visualises the list as described.

1. An operator wishes to use the app for Device Management(DM) and logs in on the app

with username and password via their chosen client/web browser.

2. The operator's credentials reach Axis' login service, performs the OIDC authorisation code flow with the IdP used by the operator's organisation, or myAxis IdP. The login service receives a token set from the IdP and sends the access token to the client.

3. The token is then sent to the ACX API to generate another token, the one used in ACX.

4. In the IAM service, the token is received and the Tokenizer in IAM creates a new token representing the operator's session with ACX. The ACX API returns the session token to the operator's client.

5. The operator requests an operation upon a resource in the DM service, and sends all information necessary for this and the session token from the Tokenizer.

6. The ACX API directs the traffic to DM, DM reads the request.

7. DM request the access control list (ACL) of the operator by checking the cached value in IAM by querying the ACL connected with the request's session token. IAM generates the access control list(ACL) if it was not stored in the cache previously, then responds with it to DM. The ACL has entries of roles and operations authorised by ACX for this operator to perform on specific resources.

8. Following the verification in the ACL, DM enforces the permissions of the operator within the service by either granting the request or denying it.

9. The result of the request is sent back to the user.

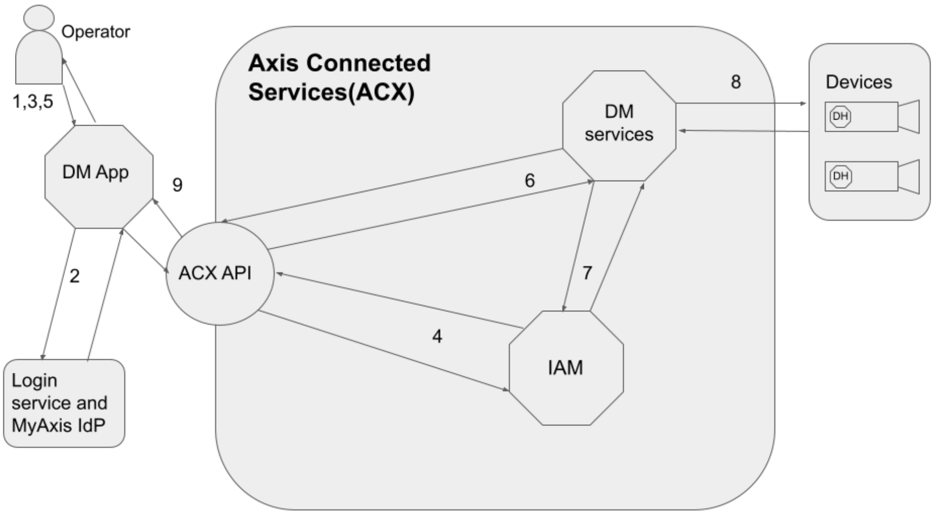


Figure 2: High-level ACX Device Management login and operation flowchart

The part of DM most important to tenancy separation is the Device Inventory (DI), a database which stores all devices ever registered to ACX with their serial number, device Id, resource group and other device-specific information. The DI is used to provide an overview of all devices for Axis administrators, as well as a reference for microservices to retrieve e.g device identification and resource group from. All devices in the DI and the other DM services are in a single table respectively and implement either plain logical separation or RLS which is the most commonly used separator when working with logical separation [10].

Access management is implemented with Role-Based Access Control (RBAC) [11] where there are principles, roles, access and the objects/resources. A user is a principle whom has certain access, and when performing an operation, the access represents a connection between the role and the object and through the role, certain operations can be performed on the objects by a user with the correct access. The RBAC model stored in IAM is converted to ACLs and then used to enforce the AC policies locally.

1.2 Purpose and goal of the thesis

In this master's thesis, the risks and opportunities of multitenancy are investigated on behalf of the PaaS Axis Connected Services (ACX) which is developed and maintained by Axis Communications AB. This investigation is launched because the platform's administrators wished to find areas of improvement especially within security and gain knowledge of the latest technologies within cloud platform administration. The purpose is to investigate the current architecture of ACX with focus on multitenancy and tenant separation, to research possible implementations for increased separation between tenants especially for data at rest and identity and access management, weighing their advantages and disadvantages. The goal and the contribution of this thesis is to analyse cloud multitenancy technologies that increase the separation of tenants and to find a compromise between assured confidentiality and usability.

1.3 Limitations

Cloud platforms have many areas with rich diversity in implementation and so it is necessary to reduce the technologies this thesis analyses to keep the scope of the investigation within the timeline of the project.

The objective first included implementation of the result which due to the size of implementation was taken to the design stage only. Since ACX was a system with many dependencies and decentralised services, changes in data storage, identity, access management, or other areas would mean changes in most services making the scope of an implementation in these areas a work of several weeks, more time than allocated for the project.

The other limitations of this thesis are that there is no analysis as to the difference in security for on premise solutions compared to CSP solutions, no analysis of tenancy separation on the lower layers of e.g architecture hardware, OS and kernel. AWS was used as basis for CSP technology and the other CSPs were not researched, this thesis focused on technologies and not CSP implementations of them. Since the ACX architects wished to keep their serverless instances, processing separation proposals were not included in the results but were still compared to the current solution in the system analysis and discussion sections.

1.4 Disposition

The report has 7 sections below, numbered 2-8. Section 2 is a review of how information was gathered and how different technologies were compared during the thesis, also specifying the focus of the project, Section 3 is a general overview of multitenancy and the technologies there are to use for multitenant architectures. Section 4 is a system overview for ACX, the PaaS in focus for this thesis with deeper description of certain functionalities later used for the analysis. Section 5 contains the analysis of ACX as to the chosen implementations of the areas in focus. Section 6 contains a comparison between technologies introduced in section 3 and ACX while section 7 presents the results of this comparison where the recommended technologies to implement are described. Section 8 is the discussion and conclusion of the thesis.

2 Methodology

Below the approach of attaining information about technologies as well as how they are analysed is described.

2.1 Research Design

The thesis has the following objectives to be realised during the project: to first analyse the architecture of ACX, what CSP services are implemented, what tenant separation technologies are utilised and how the platform is deployed, secondly perform a literature study to find differing separation methods and to list the advantages and disadvantages of each. Thirdly compare the technologies found to suggest a solution for ACX which improves the tenancy separation while not requiring much increased costs or intricate implementation methods.

The information regarding ACX was delivered by both the architects of the platform and members of several development teams within ACX. The architects provided general information regarding the purpose of the platform, the guidelines used during development, the session parameters and all CSP services implemented on the platform. The teams provided more specific details, mainly of data storage, data querying and access management procedures.

The literature study has as its goal to find technologies for ACX that provide tenant separation to the degree where the administrators could be reasonably sure that users only have the ability to see and operate on the tenant's own data. The technologies in focus are within cloud technologies, cloud security, multitenancy, tenancy separation and CSP services, possible solutions are analysed and compared to the current solution with the most important factors being technologies which provide clear benefits to ACX, and provide risk reduction for data leakage, especially between tenants. Main contributors of sources are IEEExplore, Researchgate, NIST and the Cloud Security Alliance (CSA).

By researching papers of cloud security and cloud service case studies, a comparative analysis is made where implemented technologies and the motivations behind their implementation is compared to ACX's situation to find what technology is generally used for that type of service, and why. Also general best practices are researched and applied in the analysis to ascertain whether they would provide value to the platform. The best practices are specified by NIST and are taken into account during the analysis for a broader perspective on PaaS-systems in general.

2.1.1 Assumptions

Several assumptions are made out of necessity to reduce the scope of the project, these are that fewer running instances always is cheaper, a reasonable assumption since more instances mean more resources to pay for. Another was that the services provided by AWS and Curity(IdP) are trusted entities. SLAs are assumed to be approximately equal in size between tenants thus no solution for special treatment of large tenants are presented. Data stored in a certain structure e.g document storage, is assumed to not be changeable unless specifically mentioned.

2.2 Validity and Reliability

The papers chosen as sources for the thesis are technology white papers or scientific papers published in trusted journals and conferences which are vetted before publication to e.g IEEE where most are found. All papers of non-fundamental information within the target subject are written later than 2015 to ensure the relative similarity in the internet environment as to threats and broadly incorporated technologies like CSP resources and attack vectors for different still used architectures. Several papers are older because they provide the building blocks of today's cloud technologies, like RFC- and NIST standards which have the best overview of the fundamentals they introduce, namely OAuth, RBAC, ABAC, etc..

As services in the cloud store different kinds of data, have different levels of vulnerabilities in the processing of data, and all have several unique aspects to the architecture or the goal of the system, comparisons are made with this in mind. Personal information demands other technologies than other information since laws are more strict and the consequences of a breach in privacy are more serious for people than machines. This weighing of consequences and cost is critical for a useful result.

2.3 Area of Focus

The focus of this thesis is ACX, and as it is a PaaS, issues only concerning IaaS and SaaS systems are overlooked. This is especially the case for the responsibilities visible in figure 3.

As ACX uses cloud services and has no on-premise servers, the hardware, software, network and OS layers are difficult to influence and instead became the responsibility of the CSP as shown in figure 3. Some decisions could be made by Axis to increase control over the processes, like utilising AWS instances where OS can be decided by the subscriber e.g EC2 however since Axis uses serverless(described in section 3.10.1) instances rather than server instances this is not included to a large degree in the analysis. Separating data in use is researched to find alternatives above the OS layer however encrypted processing [8] is not considered viable since the costs for all processes increase significantly when both processing time and power is increased.

As a result of the size of the platform researched, many aspects of it were abstracted from the investigation mostly due to it providing little new information to the analysis, e.g a complete mapping of all microservices, Id generation algorithms, API authentication and similar processes does not provide more material than performing a deeper analysis of a smaller but complete ecosystem within ACX. The security aspects of the CSPs choice of implementation for the infrastructure of ACX's services are not investigated due to the smaller scope of the thesis.

The security analysis of the platform focuses on the aspects the tenancy separation technologies added to the privacy and integrity of tenant data in relation to users of other tenants. This means the general security of the system is not analysed unless affected by multitenancy, and all separation technologies are assumed to not affect the stability of the system against entities outside of the system.

2.3.1 Technology comparisons

All technologies researched for comparisons has several criteria to meet, these were: the technologies must be cloud-based, with specific solutions, no or little responsibility for the users of the service, clear motivations of design choices and with the main aspect analysed being multitenancy or tenancy separation. Access control technologies are here excepted from the last criteria due to the same concepts being applicable for both single tenant and multitenant access management.

Data in use separation is researched but as with separate encryption for data at rest is not researched and analysed to the same extent as the other technologies. This is due to most of the implementations of the technologies with higher separation than the shared instances are in systems where data was highly confidential, or the tenants were large enough to use private instances themselves without as large increases in cost as would occur for systems with a smaller userbase.

3 Technology background

In this section, the theory and technologies necessary for the analysis below is presented. The focus is on tenancy separation, thus all potential alternatives are described and their traits including resilience to workload stress, risk of breach to privacy and relative performance are presented.

3.1 The environment of services in the cloud

For the last 10-15 years, companies have been utilising the cloud more and more. With server halls and shared databases (DB), both administration duties and costs have been reduced with less control over the resources used as a consequence. Several CSPs e.g AWS and Azure provide scaling hosting solutions for systems of any size at low latency all over the world, this relatively cheap service is significantly more expensive for any one organisation to build on their own [3].

The tendency to move to collectivisation of internet resources has both advantages and disadvantages. With private information existing in the cloud, there are three states where the data is at risk of leakage: at rest in the DBs, in transit on its way to another location either within the CSP system or without, and lastly in use by a server or other processors [12]. These three categories demand different analyses and solutions. No technology has the best implementation for all systems and so the best adaption needs to take into account the most important aspects of the surrounding architecture of that system. Below, data in use and data at rest will be presented at depth while data in transit already has a consensus on reasonable security in Transport Layer Security (TLS) and requires no tenancy separation, since communication is separated by client already. Although not a separate state of data, identity and access management (IAM) is another aspect vital to multitenant architectures because of its critical responsibility to distribute access to data [11][13], which is why it also is part of the investigation.

Services on the internet can be divided into four categories: On-Premise, IaaS, PaaS, SaaS which represent different levels of control of resources by the service. Figure 3 shows the areas of responsibility depending on which type of system is built. The eleven categories are required for a working service in the cloud, and each pillar visualises the division of responsibility between the CSP and the service and its developers [14]. Since ACX is a PaaS, the alternative tenancy separation technologies investigated here are limited to what the categories a PaaS is responsible for.

3.2 Multitenancy

Systems in the cloud are diverse, providing their services on different levels of cost, speed and confidentiality. The most common architecture for larger systems is multitenancy, meaning users with no relationship share the underlying hardware and software of a service [7]. This system is the most efficient way to utilise hardware resources since users seldom utilise all capacity of a server at all times [8]. The tenants process their data and store it in the system, and it is generally encrypted and decrypted in the cloud(except for the TLS-cryptography during communication) and is transported using internet infrastructure wherever the service is provided.

Resource sharing can be implemented in several ways, and in general leads to more vulnerabilities for the data on the service because of the shared space between authorized and unauthorized users for all data. The difficulty in efficiently adopting multitenancy lies in the isolation of tenants [3]. Since the concept of multitenancy means greater diversity of users on the same system, the distribution of access and enforcement of policies become more complicated and also more important issues. Depending on the technology adopted the responsibility of upholding rules set by the technology lie on the system or the developers and architects implementing separation in varying degrees.

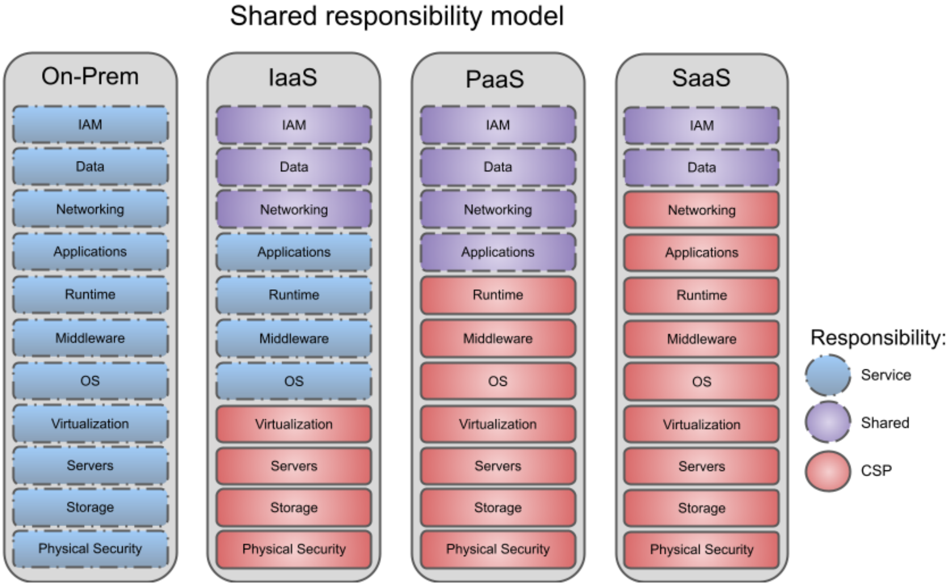


Figure 3: Shared responsibility model

3.3 Data at rest

All data on the internet is at rest where its relatively permanent location is, in general this is in a DB. Permanent here signifies the data store has no time to live (TTL) for its entries shorter than the timescale of days. Caches also store data, and if it is only stored there for a short period of time it does not count as at rest, neither does data in a queue or in RAM on a client, as it is only temporary. Data at rest is either stored in tables of rows and columns or as nested objects resulting in three major categories of data stores. The categories are object, document, and relational storage, they are utilised for different scenarios and offer different functionalities.

3.3.1 Relational storage

Relational storage is a two-dimensional matrix with a large amount of both columns and rows, they may use attributes to data points which prove relations between different matrices and can be used to build intricate models of how data is related and can be retrieved. Relational storage is stored in tables, where each table is a context within the database instance currently viewed which has to be specified in the request to reach, this is also known as a schema, and it determines how the data is presented as well.

3.3.2 Document storage

Document-, NoSQL- or file-storage is built up of e.g JSON objects to store data in a single blob with a set amount of keys, providing great flexibility to the system [15] but also makes larger sets of relations more complicated and in general is used when fewer connections exist between different attributes of entries or when there are too many connections resulting in them being specified in the each entry instead of forming the relations of the above storage type. Instead of storing all data in row and columns, document based DBs have objects within objects using key:value-pairs where a value may be a list of keys with values. The data is grouped into collections or domains, translating to different schemas when compared to relational storage.

3.3.3 Object storage

Object storage is similar to document storage, but with objects being stored instead of documents. This is e.g videos, pictures and other non-text based formats as a principle, although text also is stored there. Object storage has the fewest attributes in common between sets of data, since it only stores three values for each object; key, value and meta-data. This in contrast to document storage which can have keys within a key that all correspond to values further within the document in a tree-structure. Relational storage has no real distinction between keys and values, it depends on what is queried as all fields for a data entry are represented identically.

3.3.4 Storage security

At rest, the data can be protected in different ways. Almost all storage in the cloud is encrypted at rest. The encryption key can either cover several databases worth of data, a database, parts of a database or individual entries. Another important protection for databases are APIs, used to regulate who asks for what from where. The responsibility of the API becomes validating the source, verifying the parameters, and performing the request as well as sending a response to the requester and possible also logging the interactions.

3.3.5 Multitenant data at rest

The four ways to store data in multitenant architecture is no separation(logical separation), separate schema, separate encryption or separate DB [16]. For data security, the above four data at rest separation models can be applied to provide different systematic protections which help the system developers keep tenant-specific data private, this is described in section 3.8. Separation of resources can be performed logically with varying techniques. E.g different DBs instances for each user on the same application instance means different physical machines are connected to the same application and yet are separated logically.

3.4 Data in use

Data in use is all data that is currently in a temporary location or has an operation exerted upon it. This could be data in caches, queues, RAM, or the result of a method or function used. As data in use represents what the platform does with its data, it is very specific to the system. It could be machine learning algorithms, or displaying images that have been compressed and therefore has to be expanded to be presented. Since most data is decrypted before being processed the risk for information leakage is higher than in transit or at rest where data generally is encrypted.

Processing units handle data in use, and PaaS- and SaaS-owners buy the processing power from CSPs, e.g Google Cloud, IBM Cloud, AWS and Microsoft Azure. The instances a service buys could either be a specific location where the server is located, or a serverless unit. Serverless means that customers buy complete runtime environments and only have to write an executable which the CSP's container triggers when the customer wishes [17], the concept is explained further in section 3.10.1.

3.4.1 APIs

Application Programming Interfaces (API) are frequently used in cloud services, and are meant as a function to call from either a library or another service to simplify the production of value by the service. In this report all APIs are RESTful which is the most common type, except the ACX API being a GraphQL API. APIs are common to implement as serverless services since they have a set of actions for a given event and therefore do not require many configurations or much CPU-allocation for processing. This is how serverless processing is used by ACX, and how they are meant to be used according to the CSPs [17].

3.4.2 Multitenant data in use

Ways to separate user data include process isolation and dynamic operations on data blocks [3] but also by dividing the jobs and processing each part individually [18]. Process isolation can itself be performed either through virtual machines(VM) or through separate physical servers. All alternatives are presented in detail in section 3.9.

CSPs providing services for processing use containers/VMs for better efficiency of the servers they own as they are smaller units with discrete, configurable resources that work well for most workloads. The hypervisor, or Virtual machine manager/monitor(VMM) is the host for VMs. Each VM has their own OS instance running, and are essentially separate machines although they run on the same memory, CPU, GPU and all other hardware. This is done by assigning a fix amount of CPU performance and memory size for each VM. The VMM has control over its VMs and may receive data from the jobs executed on the layer below [19]. This results in tenants of the CSP generally sharing the same physical instances and causes separation of tenants within the customers' architecture to be more difficult since most solutions sold are multitenant via the serverless functionality.

The VMM has more responsibilities than VMs and therefore also contains some sensitive information e.g data necessary for supervising the jobs performed by VMs which could include which user the job is for and its forwarding destination, information that can be abstracted from the VM. A tool called a rootkit is malicious code that may take control over the VMM from the VM which makes them very dangerous, and proper isolation between environments is critical to dissuade this attack as the rootkit could be designed to try all known vulnerabilities of any and all types of software machines. By simply controlling the request a VM and by extension VMM receives, it is possible to gain much information about the system [20]. This could be e.g what OS and which processor a machine runs on. The choices ACX has here are limited as it for the most part relies on IaaS solutions where another party makes those decisions, although ACX still may choose which VM solution offered to use.

Serving multiple tenants on shared instances may cause problems when the workload exceeds processing power [5]. DoS or DDoS attacks could result in an unresponsive application when the servers are overloaded with requests [21]. When a privileged attacker uses their access to a cloud server they can cause performance degradation for all sharing that component, perhaps even keeping other regular users outside [22]. Not only is DoS a threat, but also unfair workload balancing between tenants, where service level agreements(SLA) may not be upheld because another tenant exceeds their quota. A server with too high workload compromises the system and could result in serious consequences to its separation [8]. By forming SLAs and enforcing them with hypervisor monitoring and also placing a load balancer at the point of entry, the system increases its resilience to workload related attacks or 'noisy neighbor' issues [23].

For a multitenant system with API-calls and resource loading from several geographic locations, content usage control becomes a high priority as a malicious user may otherwise inject bad resources that could contain malware like ransomware or viruses through e.g an input field on a service's application. Content security policy is the best method for loading resources only from trusted sources [24], as it can be used for white-listing the sources that may provide e.g data, scripts and images. Having a same-origin policy is also protective and is more strict than most content security policies, since it requires virtual private networks or clouds to load content from other sources [3].

3.5 Identity and access management (IAM)

IAM is the core of all platforms as it defines how users are identified and dictates their permissions for the resources of the platform. It is the logical dissuasion of unauthorized actions, without which no clear boundaries between users exists except the token identifier. Encryption is at times used as access control(AC) instead of logical control and ensures only those with keys may read the data that is encrypted. Both systems have their uses and should be used in tandem for maximum security [25]. Without a stable authentication system, no process within the platform is fully under controll. The threats to incomplete implementation of IAM are weak query control where users may gain insight to other resources than those they should have access to, weak privilege separation where operation privileges meant for one resource still works on another, and weak authentication where users may be able to impersonate other users, or no user. These threats are principal factors in a system with requirements of confidentiality, integrity and privacy.

Larger systems with many services and complex AC often use dedicated access management services which centrally cater all AC within to both separate duties, and make the system more structured and easily managed [26].

On similar platforms to ACX, a breach in authentication often leads to successful attacks by internet users, like in the case of the Verkada attack in 2021 [27] where authentication credentials visible in the public version of the application were used by an outside user to gain access to resources belonging to several different entities.

Authentication and other IAM concepts will be described below, while authorisation is both one of the more central parts of this thesis and also has a greater effect on multitenancy, and is presented at depth in section 3.7.

3.5.1 Identity and authentication

Usually, identification is provided through username and password, and is represented by a token in the system which the system or a third party has generated. Managing identification and authentication has become more standardised due to the OAuth 2.0 and OIDC framework and protocol [28][29] which both are widely implemented. There are other protocols for identification however since ACX uses these and it is a technology adopted by a significant portion of the internet this is the only identification algorithm presented here and is considered a valid implementation choice.

The OpenID connect(OIDC) protocol and the OAuth framework are used to handle identity and authentication by simplifying login and sessions. OAuth adds an authorization layer to HTTP which allows the system to provide weaker access rights to the request and thereby increase protection of the system's resources [28]. OIDC adds an identity layer to the OAuth framework to provide user information and enable login sessions for the system. By forwarding the user information through the protocol, other systems also using OIDC can read the credentials and the user may therefore use the same session for several services [29]. In a sentence, OAuth stands for limiting access to resources-, and OpenID for authentication in HTTPS communication.

When negotiating the initialisation of a session, the OIDC protocol standardises the exchange and results in a token or cookie representing the completed authentication process of the user via their client, often a JSON web token(JWT) [30]. Authentication usually occurs via an identity provider(IdP) which performs the validation of users' credentials, creates tokens and verifies the tokens or cookies during a session as a third party, trusted by both the users and the system outsourcing the authentication.

3.5.2 IAM from third parties

Different parts of IAM and in some cases the entire IAM module is outsourced for many systems. This is due to it being a very sensitive part of a system where many things may go wrong, thus resulting in third parties selling authentication and authorisation services, e.g AWS IAM, Ory, Curity, Okta, Casbin and more, all providing parts of or whole solutions for IAM with the selling point of having IAM as the only focus of their business. Especially identity management is outsourced to a more trusted platform whom then act as IdPs, and access management is also outsourced as a full or partial service, with libraries like Casbin and Ory or frameworks like open policy agent(OPA) which can provide access management on several levels of responsibility for the enforcement of its policies. The third party access management services mentioned use two models presented in section 3.7, RBAC [11] and ABAC [31], where a centralised policy engine is used to both increase consistency in how authorisation is granted within the system, and to offer flexible models which can take a plethora of parameters into account. Especially OPA and Casbin are commonly used by cloud systems for central AC [32][33].

AC through e.g OPA means centralising access evaluation and simplifies the pooling of several mostly independent services by providing an access evaluation engine that uses the policies written with OPA which can handle both general and very specific requests for the AC structure chosen by the system. The differences between OPA and Casbin are minute and do not impact their effectiveness, motivations for choosing one over the other are subjective.

3.5.3 Separation of duties (SoD)

Separation of duties mainly concerns separation of actions within services, so SoD does not mean tenancy separation although it may support it. SoD is a well established way of building robust systems and signifies the division of privileges to operations and resources through AC [34]. The more the system separates the duties of the services, the less can go wrong if any access point is compromised or a mistake is made within one of the contexts.

An issue with larger cloud services is their large contexts, with no strong division between them which may cause unforeseen permissions of actions with no clear policy to either grant or deny it for the services responsible. A well thought-out SoD implementation results in reduced conflict of interests as it attempts to provide one interpretation of each task as to who, what and how for each context [8].

3.5.4 Representation of access rights

How access rights are represented is as a rule either tied to a universally unique identifier(UUID) acting as a sessionId, or encrypted values and is the sessionId itself as e.g a JWT. Two other options could be used; the service may require the user to authenticate themselves for each operation however this becomes taxing quickly and rarely is a useful solution, and the access rights of the user could be sent with each operation request. The last option could be subject to tampering and spoofing attacks since the rights would be sent from the client, where malicious users, men-in-the-middle or eavesdropping network attackers could manipulate the requests.

If the sessionId or JWT is used, the sessionId/JWT could be used by attackers to impersonate another, or gain another's rights. The prominent protection for this is to provide temporal access since this means leaked sessions are only functional for a set amount of time set by the platform administrators [35][36], often set to as short time-spans as 5 minutes. When the TTL expires the user of the session is re-authenticated to ensure it still is the same user that initialised the session.

3.6 Tenant separation concepts

Security in the cloud covers all possible vulnerabilities from the hardware physical security to the OS, kernel, VMs, VM monitors, networks, routing, virtual environments and to the client, resulting in an area larger than the scope of this thesis. Through focusing on multitenancy, most of the vulnerabilities that are in common with non-multitenant architectures are not covered here, and instead the investigation concerns the consequences of users of different tenants sharing resources.

Multitenancy is implemented to reduce costs and complexity of running an internet service as its main appeal is fewer instances and cheaper maintenance when several users share an instance since it leads to a more efficient use of resources [10][3]. The resources the system provides its users with are shared, and need to be separated in some aspects for privacy and security to ensure information is not disclosed and make the system more resilient to e.g heavy workloads. Separating the users and tenants is the solution for reducing risk of either intended or accidental data leakage, to reduce the amount of data leaked if credentials are publicised and to more easily enforce SLAs. This can be implemented in several ways which this report will go over below as to access management, data storage and data processing. There are three basic principles of data at rest separation: Logical like the example above in figure 1, cryptographic and physical [16]. These principles have solutions for data in use as well, although the cryptographic separation is computationally taxing and not considered here due to its great increase in costs. AC is however separated logically or cryptographically in the vast majority of cases, the alternatives of which are listed in section 3.7. The three data at rest separation methods can be listed as four categories as below.

- Shared DB, shared schema (AKA logical separation)
- Shared DB, separate schema
- Shared DB, separate encryption
- Separate DBs, separate encryption

A similar list can be made for data in use. Instead of cryptographic separation, a more useful separation is virtual machines (VM).

- Shared instances
- Division of processing over several VMs
- VM separation
- Server Separation

Whenever tenant separation is sought, three aspects are of particular importance; performance, stored data volume, and access privileges [22]. Performance depends on how much overhead and the intricacy of the processors' jobs are which both depend on separation. The larger the tenant data volume, the more resources they require, possibly making less separation not as economically advantageous as the technologies with higher separation. AC is on of the principal separators thus making it of utmost importance for separation.

Some risks cannot be handled by the platform owner and must be offered by the infrastructure provider, like the ability to create virtual environments on physical machines, and allow for separate DBs whenever the platform owner wishes. CSPs have these options available for their customers and so the platform has a choice of which infrastructure technology to purchase. These duties for different layers of the service that is provided means that some risks are covered mainly by actors like AWS as they provide the hardware solutions while others are necessarily handled by the platform itself.

3.7 Access control

Access management has several types of technologies with the same purpose: to build a framework for administrating the access of operations by subjects upon objects [8][11]. The typical access control models used are Role-Base Access Control (RBAC) [11], Attribute-Based Access Control (ABAC) [31], and Access Control Lists (ACL) [4][31]. Other technologies of IAM are generally adaptations of the previously mentioned, including OTACS by [13] and KI-ABDR-KS by [35] described below. These other alternatives implement the basic models and further develop algorithms to use surrounding access control, answering who stores the keys/attributes where, what the keys represent, how resources are stored and grouped and how operations on the resources are performed.

The technologies below in sections 3.7.3 to 3.7.8 are presented as alternative implementations of AC, the three first are the basic cases which most use and the other 3 are advancements more focused on multitenancy which base their solutions on the three first. Some terminology is listed here first for clarification:

Subject is a user of the system, requesting access and performing operations, could also be a process or machine user.

Object is a resource within the system that requires access privileges to be reached, it is generally some form of data. Objects could contain objects or be the resource sought after.

Operation An action performed by a subject which affects an object or several objects. Could be a process unique to the system or a CRUD-operation(Create, read, update, delete) on a DB. Similar to objects it can have several operations within one or be one type of operation.

3.7.1 General practices

Technologies divide the subject and its access rights to individual or groups of objects differently and each is useful for different cases. Access control can be very specific to a system design as the goal is to separate duties as much as possible as mentioned in section 3.5.3. This can be done in varying degrees depending on what the system should do and is strongly influenced by how intuitive and flexible the system is meant to be to the end users and how tailored the permissions should be.

Access management for a PaaS means both many and diverse cases to be handled, both between users with different tenants and between users within tenants. A service-specific AC that is useful for a smaller system becomes opaque when the AC for each service must be managed both through a centralised IAM module and locally in the services. Due to this the recommendation for larger systems is a dedicated component to handle all the types of requests the system receives at one location [26], while the smaller ones have fewer recommendations since the risk of convoluted schemas is lower. The most important aspect is providing minimal access rights to all parties, ensuring they only perform their designated tasks and nothing else.

3.7.2 Multitenant access control

Access control separates privileges to ensure resources in a system are as secure as possible. This is not a multitenant issue, but an issue for all systems. Tenant separation can be performed using AC which works as an addition to the data at rest separation to only grant access to the user's tenant's data. The sources investigated that discuss separation techniques have data at rest separation thus a separated multitenant system should at least use at rest separation. Multitenant AC is used as additional separation and the separators can be e.g tenant specific attributes, roles or operations. Multitenant AC is a complement to at rest separation however a system could also have no multitenant AC and still consider its resources separated.

3.7.3 Role-based access control (RBAC)

RBAC was designed by NIST as a civilian AC model to centralise the management of permissions to privileged information [11] as the military grade of mandatory access control(MAC) was seen as unnecessarily demanding and unyielding in this case. It has a mapping between subjects, roles they may access, and the access the roles have vis-a-vis the objects concerning operations. RBAC is the most commonly implemented AC system since it allows both simple and scalable configurations and covers all common AC cases desired by software systems.

Roles are similar to an employee's title in an organisation as both correspond to a set of privileges/responsibilities and resources to utilise. The role is what is connected to the resources and what operations that can be performed on them while the subjects/users take on the roles/titles to perform their operations/tasks. Figure 4 shows a concept of role based access control. The subject has a set of roles they can operate with, while the roles have sets of operations and resources which these operations may be used upon.

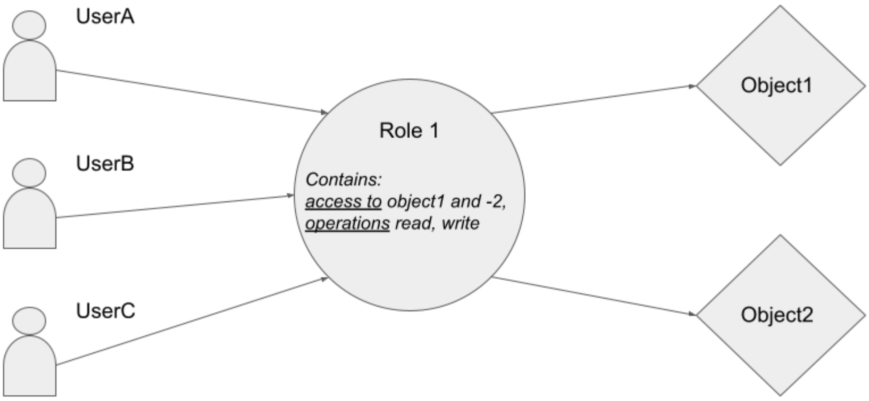


Figure 4: RBAC visualisation

Just as there are experts within a field, roles can be inside roles, creating a tree of inheriting access rights where for example a doctor could be a basic role, and physics, medicine, math or philosophy all mean expertise in separate areas, resulting in some shared access due to the doctor title, and with more privileged access within the respective field with the addition to the title.

RBAC has a straightforward implementation method- to use roles to connect subjects with access- and operation pairs and then apply the properties of the role upon the object. Which subjects have what roles is stored in a DB and the roles' connection to the resources and their access and operations is specified in yet another DB. This system places the access management close to the roles since they specify what can be done and it is extra important for the management to be able to evaluate the roles and understand what purpose each has as a large system may have many roles with almost identical privileges resulting in a difficult system to understand for human eyes, especially if the same roles have different purposes in different contexts.

Since roles can apply specifically to certain resources within a system, RBAC access evaluation sometimes occur independently within each service on a larger system, but centralised evaluation and control is also possible by specifying which roles belong to which services in the evaluation engine, or specifying the operations each role pertains in the services.

Unlike figure 4, the access to objects can also be connected to subjects directly instead of through the role, thus putting less focus on the role. It would offer more customisation since another parameter of access to objects offers more possible ways to distribute access. Access is then an attribute of the subject and only shows that a subject has rights to an object, however not which rights. The rights are still defined in the role and have to be combined with the access attribute for valid operations. Thus the difference is that in the standard RBAC, users need a access to a role for operating in a system, while the alternate design requires the user to have access to both the role and the resource to operate on.

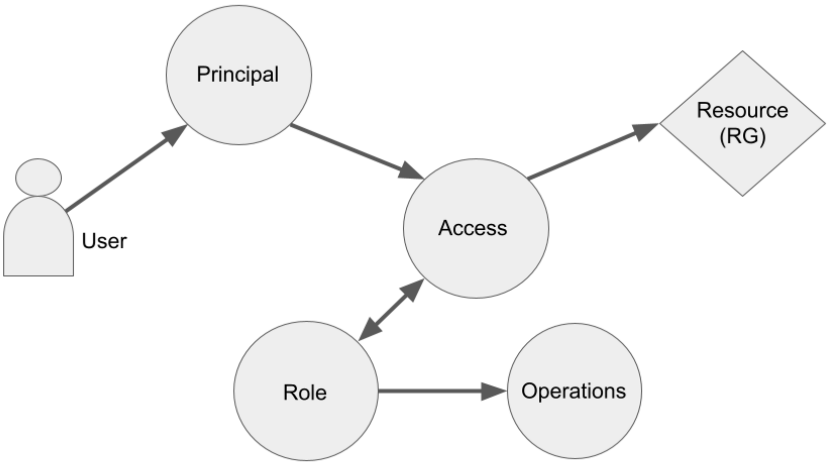


Figure 5: ACX access control (RBAC)

ACX uses this alternate technique as seen in figure 5, where a principal is a user's identity in ACX. The role needs to be paired with the principal's attribute of access for the principal to perform operations determined by the role, on resources accessed by the principal.

3.7.4 Access control list (ACL)

An ACL is a simple table with {subject, object, operation} entries which describes all relationships for all objects. It is also known as IBAC or identity based AC [31] and is very transparent as each identity, each resource and each operation is clearly stated together, which both means easily evaluated requests but also more difficult configuration. If any access is to be changed, or new resources are added to the system, all subjects whom should be granted or denied access needs to be stated in the ACL individually, and cannot use the same grouping as is possible in RBAC through the roles. ACLs are often stored in relational DBs, making all access easily queried to ascertain validity of a request.

Generally ACLs are stored locally with each API/service and thus only the sum of different operations are limited to what is possible in each service with the number of subjects and objects being limited to those within the service. The operations meant to be used in the ACL are CRUD-operations and if not, are explicitly declared as a single operation resulting in a single specific action, and the object is meant to be the identifier of a single tangible resource. Neither operation nor object needs to be this concrete yet changing them to e.g have an object cover several resources means that the evaluation becomes more advanced and slower as further investigation by the service becomes necessary to evaluate the request. The effect is due to the access management engine now needing to see which users have access to what object groups as well as what action each operation entails.

ACLs provide all information necessary for the AC evaluation context in the request, making enforcement intuitive. Because this simplicity ACLs are useful for systems with either small user pools or few resources. It scales poorly as each access right needs to be explicitly written, thus resulting in more work especially when users gain access rights within the system. The new rights all have to be specified and added, instead of e.g RBAC where this could be fixed by adding access to one or a few roles instead.

Figure 6 shows the structure of ACL AC, where the redundancy is shown by the users having access to the same operation for different objects.

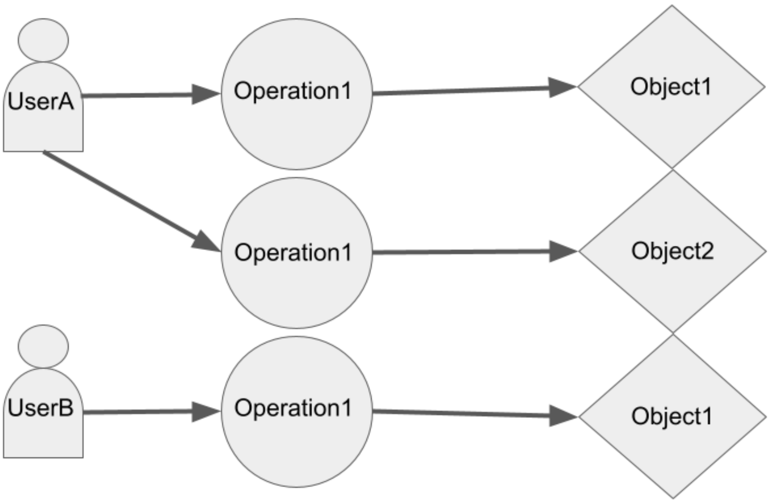


Figure 6: ACL AC

3.7.5 Attribute-based access control (ABAC)

As stated in the introductory paper to ABAC [31], RBAC and ACL are special cases of ABAC where the ACL uses user identity as an attribute and RBAC uses roles. The permissions of ABAC lie in whether the user's attributes match any of the attributes of the object and whether the permission request also fulfills potential system policies/environmental factors.

Attributes could consist of very basic information regarding the user, like age, seniority within an organisation, office location, a certain trust level, etc.. The attributes are inherent to the user, and act as a more broad use of roles since a role definitely grants certain access to resources, while attributes may need to reach a threshold, need to be combined or need to be missing for access to be granted.

Policies and environmental factors are similar to attributes yet are instead parameters of the situation rather than of the parties in the exchange. For environmental factors. a user may have all attributes necessary to access a user but is accessing it at invalid locations, times, in a wrong context, or similar factors which depend on other parameters than the context within the system. Environmental factors are thus external factors to dictate the AC enforcement within a system. Policies depend on the context within the system, e.g if there are certain chains of operations that are not allowed, if several user of a specific role performs operations on the same resource too often or seldom or other rules that depend on a wider context than attributes provide. These policies and factors are specified centrally in the AC engine, the manager of the ABAC model. Figure 7 shows the connections of all entities where neither policies nor environmental conditions are necessary yet add customisability to the ABAC system.

ABAC is a very flexible tool as it offers more parameters to customise than the other technologies. With attributes as the propagator of access rights, user identities are not necessarily recorded in the access management system and so providing and removing access to resources becomes as simple as either removing the attribute of the user, the object or changing policies/environmental factors.

Unlike ACL and RBAC, the standard ABAC model has a central policy engine that evaluates all parameters in each request requiring authorisation. ACL and RBAC can however also use a central engine, although ACLs advocate local AC, while RBAC can be effective in both types.

There is no restriction for how attributes behave, and so similar hierarchical solutions as in RBAC are possible, with even more granular configurations without making the evaluation of

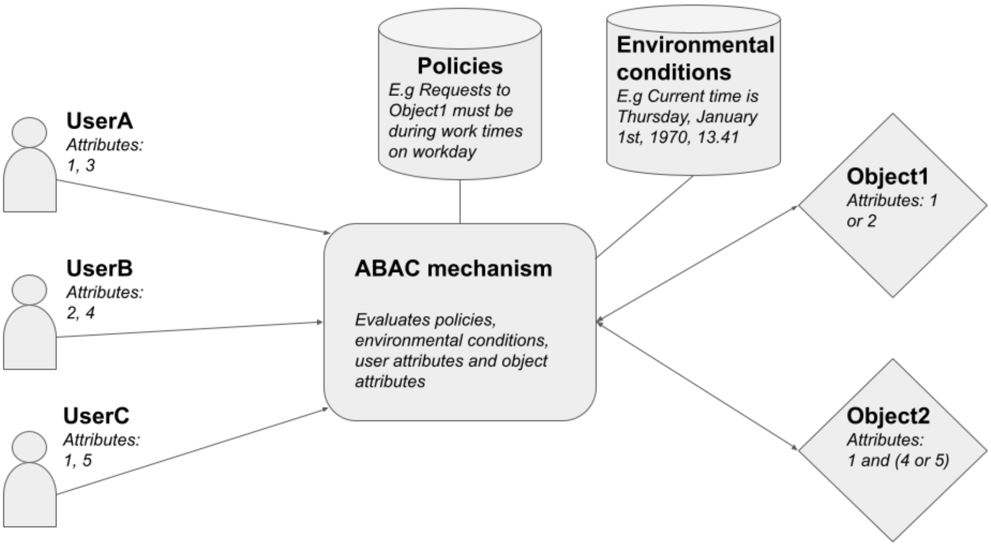

Policies
E.g Requests to Object1 must be during work times on workday
Environmental conditions
E.g Current time is Thursday, January 1st, 1970, 13.41
UserA
Attributes: 1, 3
UserB
Attributes: 2, 4
UserC
Attributes: 1, 5
ABAC mechanism
Evaluates policies, environmental conditions, user attributes and object attributes
Object1
Attributes: 1 or 2
Object2
Attributes: 1 and (4 or 5)

Figure 7: ABAC access control

access as diluted as a RBAC model with too many roles.

ABAC is both the most complex and simple alternative for AC as it provides infinite possibilities in how attributes and other factors are managed. This can also be seen as a disadvantage to the system since the responsibility of a secure implementation lies completely with the architect of the AC-model, and there are few guidelines to support decisions when the system implements a unique ABAC solution. This is unlike RBAC and ACLs which have more clearly defined properties making them more intuitive to implement, even if a simpler solution can be achieved with ABAC.

3.7.6 Secure logical isolation for multitenancy (SLIM)

Some experts find that the basic AC methodologies above do not separate users and resources adequately in multitenant cloud systems, and instead promote other more adapted strategies, where SLIM is an addition to AC to address this [37]. SLIM is less focused on how the access is distributed, and more about how the access can be contextualised and enforced within and between tenants in the system.

In SLIM, both the user and the tenant are authenticated before a request is granted, and tenant affiliation is enforced across all services by a guard and proxy service overlooking the active sessions. The load balancer/gateway is also responsible for dividing the users into parts of varying privilege requirements within the system.

Whenever a shared instance is accessed by a user, the gatekeeper is used to ensure only resources owned by the user's tenant are accessible first, then viewing the AC policies for within-tenant privileges. Which policies are utilised depends on the type of resource, what is shared and how, however the priority of first separating the tenants, then roles/attributes/userIds within the tenant depending on if the applied model is RBAC, ABAC or ACL is meant to allow for tenant-specific AC.

Figure 8 shows the introduced entities and their connections. The proxy and guard keep the user in the correct context and authenticates both the user and tenant respectively. When the service of context 1 needs data or a process in context 2, this is first channeled through a proxy for the context to retain information of its origin, second a guard which retains the privilege of the session in context 1. The gateway is a combined load balancer and privilege separator for requests from users and the gatekeeper separates requests based on the context of the request first by tenant, then by privilege. If a user within context 1 requests an operation which requires action in another context e.g context 2, this must first go through the proxy and guard which ensure the process requested remains within the bounds of the privileges granted to the tenant and user by both providing information of how the process request came to be and who requests it.

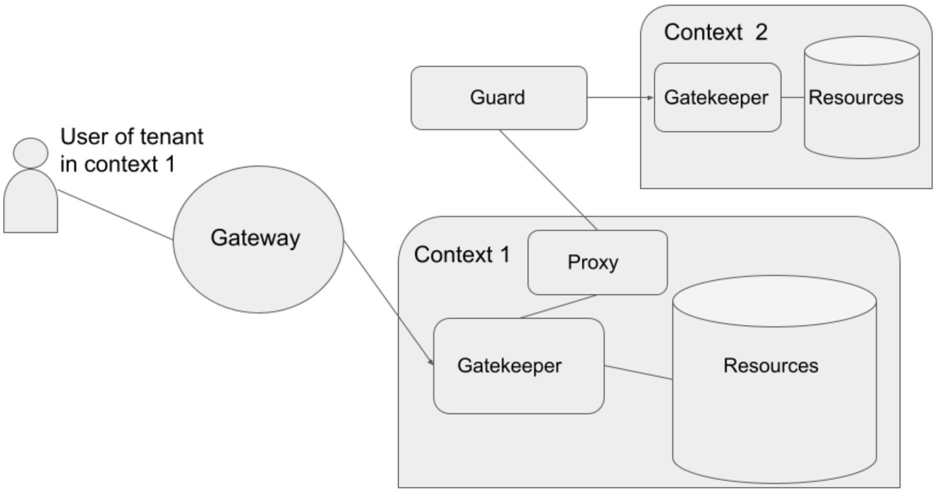


Figure 8: SLIM AC

3.7.7 Object tag access control strategy (OTACS)

Multitenant cloud systems have additional complexity within AC since there are more cases to cover due to the increased user pool and required divisions of access rights. OTACS is meant to reduce the risk of any cases not being covered, including users with partial access gaining full access or sharing resources unknowingly to unauthorised users through second or third party instances [13].

The strategy for evaluating requests is through logical relationships between subjects, between objects and between subjects and objects. All subjects have subject security- and collision tags, and all resources have object security- and collision tags. The tags are used to evaluate the access rights and can be used in a logical computation where $st_o \wedge ct_s = 0, st_s \wedge ct_o = 0$ means that the object security tag and subject collision tag do not create a collision and neither does the security tag of the subject and the collision tag of the object, resulting in granted access. The collision tags are for stopping unauthorised access, while the security tags are for granting access for authorised users.

Whenever access needs to be updated, either the subject or object collision and security tags are updated depending on which operation was performed, read or write, through $st_s = st_s \vee st_o, ct_s = ct_s \vee ct_o$ for read, the same but for the object when the operation was write. Deleting objects and subjects means removing their tags, this could either be done by deleting the object reference, or remove its tag from all other tags which have been used in conjunction with it.

The tags are attributes distributed to all entities in the system thus forming a type of ABAC but with focus on fast computation and potentially very stable access policies, at the cost of a complex set up process. OTACS does not have functionality for separating different operations since the authors advocated an all or nothing access to resources, especially due to the implementation mainly being designed for services that store tenant data, not process the data.

By decentralising the authorisation evaluation and reducing it to a mathematical computation, performance is increased by a significant factor. Also being able to modify access through mathematics makes for an effective solution. Evaluation misinterpretations are almost impossible in theory since it is a simple computation for the evaluator.

3.7.8 Key insulated attribute based data retrieval scheme with keyword search(KI-ABDR-KS)

Encrypting resources and only decrypting it for users with access to them is considered best practice, through encryption no subject can read any data they want without authorisation from within the system. [35] motivates encrypted separation within a system between tenants in a similar manner, as the tenants generally have nothing in common in the system except the processing, the structure of their data and possibly DB instances.

By combining encryption and AC, unauthorised access to objects is more difficult to attain than through the other technologies presented since the data reached in an unauthorised manner still is encrypted. KI-ABDR-KS has encryption of data based on attributes so the subjects with the correct set of attributes may decrypt the data using the keywords belonging to these attributes and their own keys for the system. Those without the correct attributes and keys cannot receive the data, however if they did through malicious action it would not result in lost privacy since the data then still would be encrypted.

The scheme requires an attribute authority, a server, data owners and data receivers. To use this system there is a process to implement:

1. The attribute authority distributes public and private keys to the data owners and generates public parameters identifying each tenant's data. At a set interval the authority also updates all keys to keep the system confidentiality.

2. The data owners generate an index for each encrypted object and send the index to the data receivers whom create trapdoors for it. The index contains the system parameters, the tenant's access structure and keywords for each data entry owned by the tenant.

3. The server stores the data and receives requests from the data receivers to the encrypted data. The server takes the request which contains a trapdoor to the data and responds to the data receiver with the data the trapdoor corresponds with.

4. The trapdoor is generated by the users/data receivers and is a computation containing the keywords of the data entry and the user's private key and a nonce.

KI-ABDR-KS has a complex setup process yet provides good protection from any privacy intrusion. The level of privacy can become an issue for system administrators as they can not process the data when the tenants generate the trapdoors for fetching and decrypting data. If the generation became the responsibility of the system, separating the receivers of the data from the users, the model becomes more useful for systems which both provide data storage and processing. The data owners, similar to the trapdoor generators, also have the ability to either be part of the system or not. This ability to shift the responsibility of different AC modules means it can be suitable for use in systems with low levels of trust to everything except the users, or for systems with high trust in the system.

Figure 9 shows an overview of the system. As seen the attribute authority distributes all keys for creating the relationships between data owners, data receivers and the data itself. The data receivers use the keywords and their key to access the data.

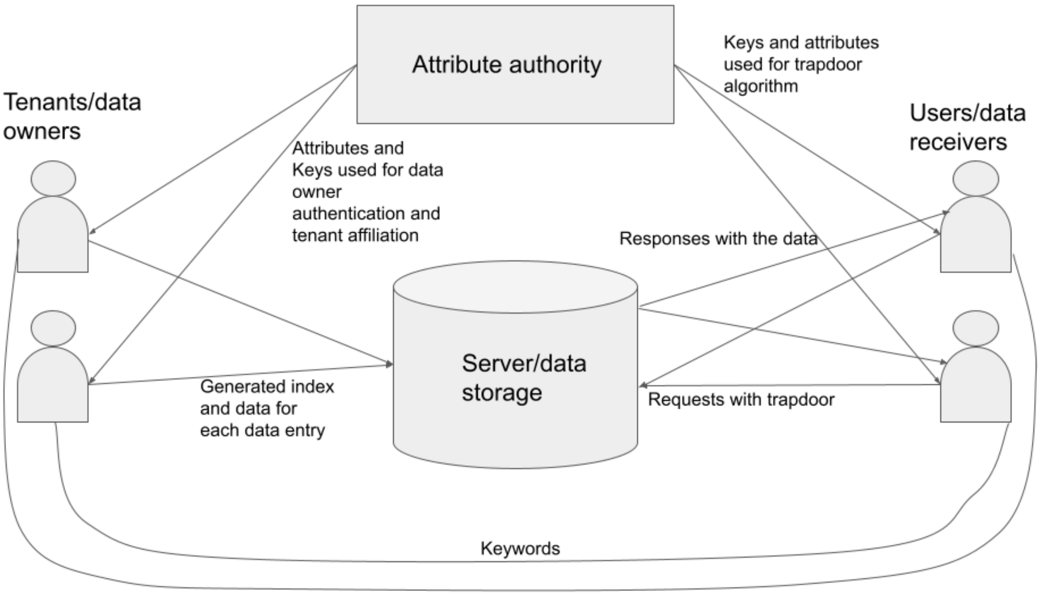
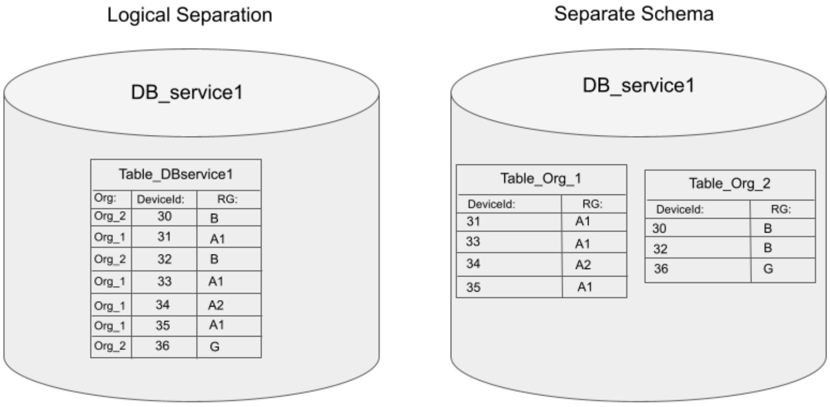

Attribute authority
Tenants/data owners
Users/data receivers
Keys and attributes used for trapdoor algorithm
Attributes and Keys used for data owner authentication and tenant affiliation
Responses with the data
Server/data storage
Generated index and data for each data entry
Requests with trapdoor
Keywords

Figure 9: KI-ABDR-KS overview

3.8 Data storage separation

Below are the four different alternatives for tenant data storage in a cloud. The alternatives all work for document and relational DBs, while object stores do not have an equivalent for separate schema. The alternatives are presented and their implementations are described.

Figure 10 shows the difference between logical separation and separate schema in a relational database. All data is in the same table on the left instance and is reached using the same instance/table name, whereas the right one requires the schema/table name to retrieve data, resulting in increased systematic barriers between tenants.



Table_DBservice1

Org:	DeviceId:	RG:
Org_2	30	B
Org_1	31	A1
Org_2	32	B
Org_1	33	A1
Org_1	34	A2
Org_1	35	A1
Org_2	36	G

Table_Org_1

DeviceId:	RG:
31	A1
33	A1
34	A2
35	A1

Table_Org_2

DeviceId:	RG:
30	B
32	B
36	G

Figure 10: Separation technologies

3.8.1 Logical separation

ACX currently stores its data using logical separation, the simplest separation between tenants. The level of separation describe systems where all tenants have their data stored using the same parameters, with the same encryption key for all data on the instance resulting in all data being decrypted whenever the DB is queried [15][16]. In some cases a single DB instance stores all data, in other cases the data necessarily is divided into several instances when the capacity is surpassed for a single instance which still means data is logically separated since the storage strategy is the same for all instances. The separation is logical, meaning implemented using code that differentiates access to resources between users. This could either be implemented through no differentiation in storage and the separation lies in AC, or the data has meta-data appended which is used for tenancy separation during queries.

For increased separation row level security(RLS) can be implemented, for e.g relational storage this means an extra column containing an organisational identifier (orgId) is appended to all data entries and for a complete implementation is also added to the query after a user requests their data. The orgId can then be used for all queries as a filter to separate the tenants' data and also improves AC since the orgId has to match with the user's tenant. This method of separation puts less responsibility on the developers, since there now is a clear affiliation for each object/file/entry located in the data store that adds another method of validating access. Faulty design and implementation can lead to unauthorised data access more easily than the other separation technologies since there are fewer barriers in place to overcome, either intentionally or by mistake. It does not lead to misunderstandings or accidental bugs at the same level as the other technologies though since all data only needs to be tagged with an orgId and thus wrongful implementation is a smaller risk. Logical separation can be sufficient for systems if IAM and platform best practices are used and enforced in a complementary way [10].

Although logical separation is the least private multitenancy technology it still is recommended and used by many in combination with RLS due to its simplicity, low cost and scaling capabilities [16][37]. The risks involved in using this technology compared to the others are concerning the encryption/decryption and AC. Since the entire DB is encrypted and decrypted using the same key, a user within a tenant context prompts the DB to decrypt all data on the DB when requesting it, although it is only permitted to read a subset. Mistakes made in AC could then result in the user reading data they are not permitted to, which would be plaintext in a logically separated DB unlike e.g using separately encrypted data and the data is encrypted by another key.

Logical separation implementations

Logical separation with API-enforced RLS: In [38], the authors showcased a cloud system architecture using shared components. The case was a business process management application, with users from different tenants and in different levels of management and thereby levels of access using it. The focus of the case was implementing a simple solution with low cost but still with focus on data at rest and in use SLA performance guarantees. This case is interesting as it is the most similar to the current solution on ACX, and provides motivations for implementation of similar separation by other parties. Figure 11 illustrates the proposed architecture of the system.

The data at rest was stored using AWS SimpleDB, a document data storage service. All data presided in a single domain/collection, with the key of each item being the organisational Id and each value being the data stored, resulting in document based logical separation RLS. The database with all data was replicated and spread out geographically for increased stability and faster response times All users of this system only had access to one tenant's data, resulting in the organisation being part of the userId, which the API calling the DB could parse and see. The API was another part of the separation as it did the querying and could isolate the tenant's

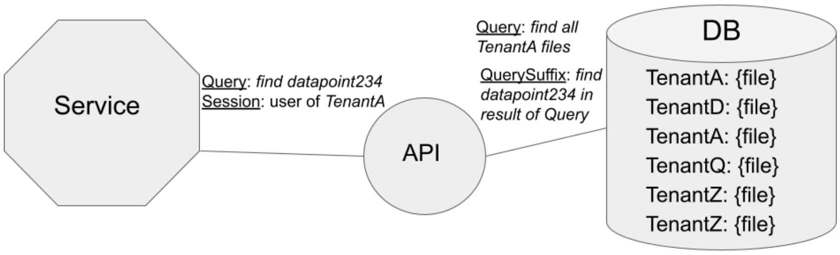


Figure 11: Shared component case data at rest implementation

data for each query through filtering with the orgId. Whenever the data was queried, the API would remove all data not owned by the requester's tenant before forwarding any information.

The motivation of using the RLS logical separation was *"that they are more systematically presented, easier to implement, with more logical headings and detailed pattern description"*. The implementation was simple with logical separation, and with more shared resources the result was also a cheap system to maintain, scale up or down and develop. Although this implementation was chosen, the conclusion was that:

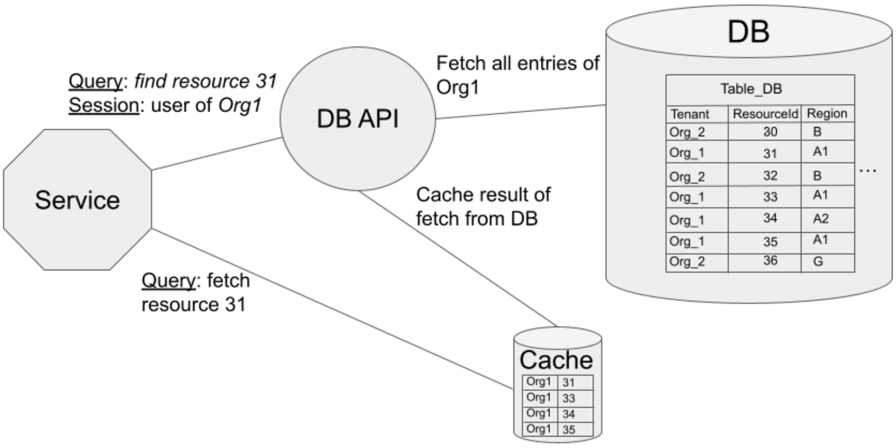
"The Shared Component pattern minimizes resource overheads by making efficient use of critical components. Tenants are each allocated a quota of the shared resource. However, sharing components could compromise privacy, performance and security"

The privacy, performance and security to be compromised are the consequences of poor implementation of the system proposed, and not due to greater vulnerabilities than those of separate schema and separate database.

Logical separation with caches: [16] made a taxonomy of data at rest multitenancy approaches, listed their uses and when to avoid them. The paper also proposed a multitenant design for a general cloud service with several tenants storing large amounts of data and where the speed of the service was a high priority. The system had all data at rest in the cloud and the goal was to find efficient tenant isolation while keeping performance quality. The technologies discussed were separate database, separate schema and logical separation. The chosen approach was logical separation with an architecture illustrated in figure 12.

The data storage structure was relational storage RLS with each data entry having the tenant's UUID as the orgId. When a tenant wished to reach their data a service queried all entries for that tenant and stored this in an empty cache resulting in a separate schema from the tenants' perspective since all owned data is in a separated table when they retrieve it. The DB had many columns, since the paper assumed the tenants had similar but not identical structure to their data thus including many empty points for all entries. The DB service took all columns the tenant had defined values in before storing it in the cache, making queries during the session from the DB quicker after the initial query since all superfluous data already had been removed.

When a user of a tenant logged in, they were first authenticated both as the user and as part of the tenant, after which a session is generated with a UUID as sessionId. The tenantId was stored in a cache with a relationship to the sessionId thus determining the context scope. Whenever a user wished to access their data the mapping between sessionId and tenantId was verified after which all data belonging to the tenant was fetched to the empty cache. The two steps of authentication, first between the sessionId and tenantId, then the tenantId and the



Table_DB		
Tenant	ResourceId	Region
Org_2	30	B
Org_1	31	A1
Org_2	32	B
Org_1	33	A1
Org_1	34	A2
Org_1	35	A1
Org_2	36	G

Org1	31
Org1	33
Org1	34
Org1	35

Figure 12: Data storage and querying architecture for the second logical separation case

DB entries, resulted in adequate tenant isolation according to the authors. This system has no AC within tenants and therefore assumes only authorised personnel within the tenant has credentials to use the system, and if AC was desired, this could be implemented by the tenant after fetching their data. The tenant data could also be queried when the user logged in instead of when they wished to access their data, thus reducing the response time for the user when they query the data.

The motivation for the increased separation through caches was that first removing all data not owned by the tenant, then allowing the user access meant higher level of privacy than only using RLS. This is due to the isolation of tenant data when searched by the user, as there only was data from their tenant available unlike RLS where it is present in the search but removed before results are presented. The first case above could result in security loop-holes within access management when tenants query the shared DB directly, and since the performance of this technology was still close to the RLS implementation despite the increase of isolation it was according to the authors sufficiently motivated. The authors did a performance analysis and found that the RLS and cache approach gave essentially the same response time for queries. E.g for 40 tenants, the response time was 4.1ms for the cache approach, while it was 4.3ms for only RLS. When users queried data they could only query the subset of data belonging to their tenant. The ease of implementation compared to the alternatives of separate database and separate schema and the increased separation from RLS resulted in the decision of logical data storage separation.

3.8.2 Separate schema

The next level of separation offers increased levels of privacy and confidentiality of tenant data compared to logical separation. The separation consists of providing a table for each tenant within the same DB instance. This means the schema first needs to be specified before a search over the data can be performed, compared to logical separation where there either is no separation or a RLS filter first needs to be executed before a search. The increased security is in the data of all tenants not being searchable together, instead each request must have a specified table with e.g the orgId as value. This can be done for both relational and document DBs, with the first having separate tables [16], and the other separate collections [15]. Queries to the DB are otherwise made the same way as in logical separation, the difference lies in the now obligatory field of schema name that specifies which tenant the data is fetched from. Since the data still is in the same DB instance they share the storage pattern [2], resulting in simple data management.

Separating data into different schemas means it may be easier to find bugs when a single tenant is experiencing issues, as an overview of a smaller data set is clearer. The system has more responsibility of separating tenants instead of the developer writing the queries since the orgId/schema name is mandatory for valid requests. Through the increased responsibility, the system itself supports the developers to a greater extent than logical separation can.

Separate schema implementations

Document store sharding:

[15] recommended a separate schema implementation between tenants for a multitenant cloud system by using separate collections for each tenant in a document DB, in this case MongoDB. Figure 13 illustrates the system design.

The paper focused on systems with a lot of data per tenant and with many tenants, as well as possibly different structures of data per tenant. Document DBs offer high customisation for data as each entry is a key with an e.g JSON blob of non-specified structure which was desired in this case. The case was implemented on a SaaS system providing venue booking for e.g movies, music festivals and conferences for the users of the tenants.

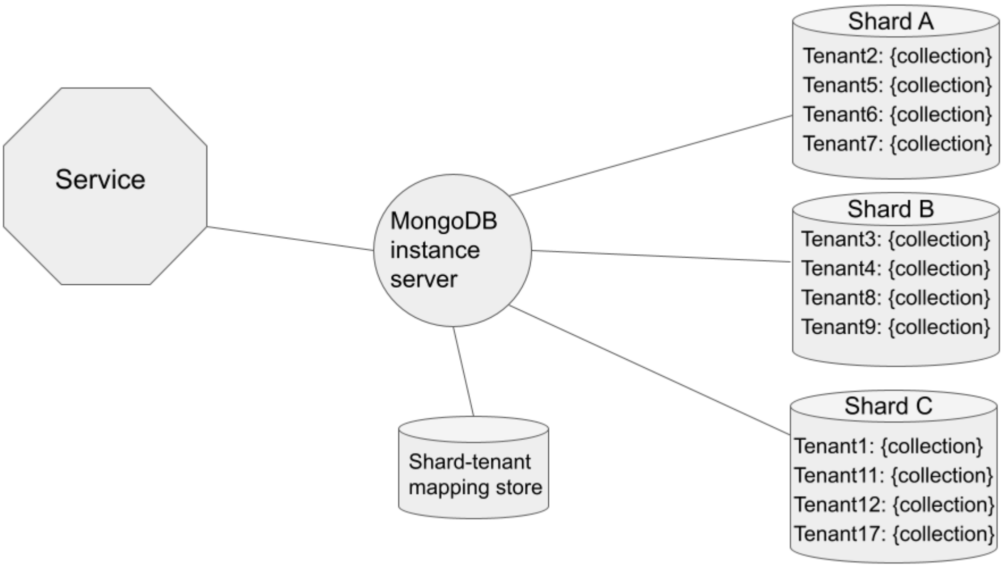


Figure 13: Storage architecture for the document store sharding system

The system used document data storage, with collections containing data belonging to one tenant. Collections were on the same DB instance yet were separate due to the collection needing to be specified before data was fetched. MongoDB has a feature called sharding which was utilised by the authors for built-in separation of instances.

Sharding means dividing the DB over several instances with several complete collections in each, so that the DB in practice had many instances, but is still multitenant since several collections are in each shard/server. Fetching data results in using the shard key the tenant is connected to and finding the collection in the shard with the corresponding orgId. Although data is spread out over several instances and possibly physical hardware, it is still a separate schema separation due to the interconnections of the instances through the MongoDB server and the shared shards between collections of different tenant's data.

By separating all data in collections the resulting isolation between tenants of the system was equivalent to separate schema. With sharding, fewer tenants share instances thus increasing the physical separation between groups of tenants and performance was increased for greater workloads.

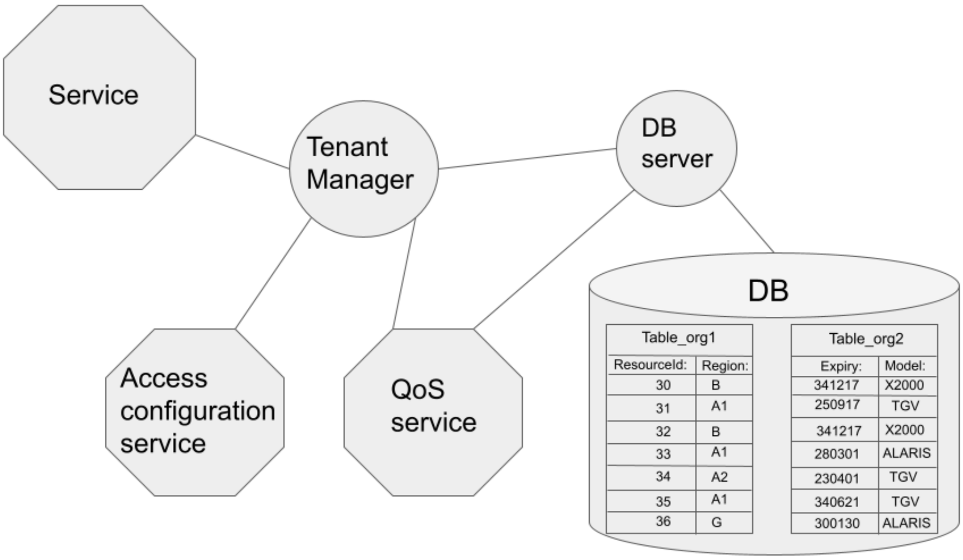
The authors motivated dividing into several instances of document DBs by mentioning the nimbler instances, the increased scalability and the increased isolation between tenants.

Access management was defined through the MongoDB instance which had been configured for the tenantId to be the authorisation needed to access a given collection connected to the specified tenant. Within the collection no authorisation separation was defined in the paper since no entry was seen as privileged information in venue booking which the system was designed for.

Separate schema with a dedicated tenant manager (TM):

In the multitenant system presented in [23], the goal of the design was to share resources efficiently between tenants whom share processing functionalities but not data. Resources in the system were meant to be as close to the SLAs as possible for maximum revenue. The system designed in the paper was a database management system where tenants store their data and operate on that data within the system using shared resources.

The overall structure was shared processing instances and a schema separated database with centralised AC, illustrated in figure 14.



Table_org1	
ResourceId:	Region:
30	B
31	A1
32	B
33	A1
34	A2
35	A1
36	G

Table_org2	
Expiry:	Model:
341217	X2000
250917	TGV
341217	X2000
280301	ALARIS
230401	TGV
340621	TGV
300130	ALARIS

Figure 14: System architecture for the separate schema and TM case

Separate schema data storage was employed with relational databases containing many tables, but with an equivalent to the sharding technique in the previous case. Each instance of the relational database contained a set amount of single-tenant tables, thus spreading out the workload on several DBs when necessary but with the ability to combine them in one instance for cost efficiency. All instances had the same API endpoint connected to the service, making them functionally shards of a group of DB instances

Separate schema was chosen due to tenants storing different types of relational data which meant many columns for each entry, decreasing the performance of the database. Having no columns in common between many entries means separate schema is the better approach when considering efficiency of relational data stores since there are fewer fields per entry than if all data was in the same table. All data within a tenant therefore had to implement a template so that it fit in the fields of the table.

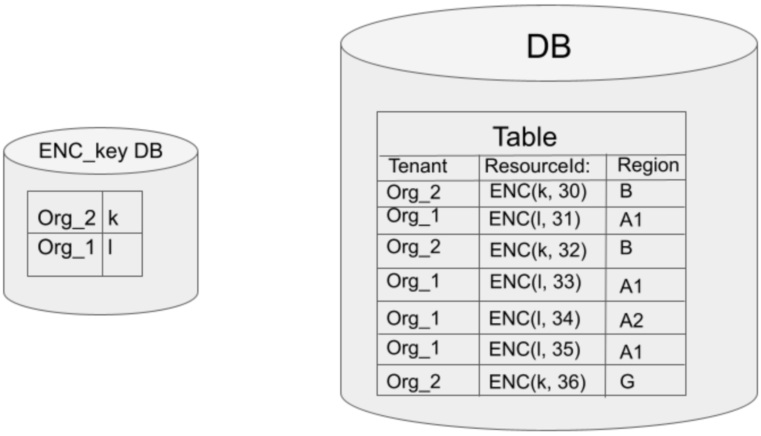
The access management was centralised through the TM which evaluated and formatted the requests sent from users into the system to ensure the requests reached the correct table and performed the correct action. The AC between tenants was the most critical and was described in the system while the AC within the tenants depended on the agreement between the service and the tenants. If AC within the tenant was included in the system SLA this was specified and used in the access configuration service, where a policy engine evaluated the request. Otherwise the user had access to all their tenant's data.

The access rights of the requester were reviewed by the access configuration service and the SLAs of the tenant combined with the current workload of the DB was measured by the Quality of service (QoS) service. The information gathered by the services was delivered to the TM which then sent the request to the DB server which performed the request if all parameters were satisfactory to the requirements of the access configuration service and QoS service.

3.8.3 Separate encryption

This separation technology is a spectrum of possible implementations depending on what is encrypted, and how. The broader terms of separate encryption include tenant specific, user specific and attribute based and are all useful in specific cases.

Tenant specific encryption is the simplest alternative and enforces the policy of tenant separation by making all other tenants' data unreadable even if reached by an unauthorized party. Each user would have the tenant-specific decryption key, used on the whole database when a user queries it and leads to only the data encrypted with the correct key to become readable. A simple example of this is shown in figure 15. The DB with the keys can have greater protection to ensure the separation actually is increased, and not logical which would question the efficiency of the other protective measures since tenant data then still would be adjacent.



Org_2	k
Org_1	l

Table

Tenant	ResourceId:	Region
Org_2	ENC(k, 30)	B
Org_1	ENC(l, 31)	A1
Org_2	ENC(k, 32)	B
Org_1	ENC(l, 33)	A1
Org_1	ENC(l, 34)	A2
Org_1	ENC(l, 35)	A1
Org_2	ENC(k, 36)	G

Figure 15: Example of tenant specific separate encryption

If the system has an AC policy this technology would work best in tandem so that it does not interfere with the already established separation of users within a tenant. The encryption of data could be specific to authorisation levels thus supporting the enforcement of AC policies.

Since a tenant may have hundreds of thousands of objects stored, the encryption key may be possible to break through finding patterns of the entries. This can be thwarted by encrypting every entry with yet another key, unique to that entry, and then adding that key to the data entry as another column. By encrypting the data entries with this technique, no information about the key from one entry can be used on another since they only share one of the two levels of encryption [12].

If the tenant holds any of the keys, The service cannot provide a faulty key or faulty data either [8]. A challenge with this method is that either the entire database is subject to the decryption where only one tenant's data becomes plaintext, thus limiting the DB to queries from one tenant at a time, or an orgId indicates where to use the decryption key in which case the encryption becomes another level of separation upon the RLS of logical separation.

Another separate encryption implementation separates between users, either through an administrator with all keys [8] or by encrypting data each user has access to individually. This would either mean that all data entries would have many copies with different keys to decrypt resulting in data redundancy, or very isolated users where one or few users have access to each object. This case is mostly useful as a special case of the tenant-specific encryption where the user is essentially a tenant on their own.

Encrypting data by using attributes that define who should have access is possible [25], but requires an intricate administration of encryption keys. This method of encryption either relies more on the users and tenants of the platform since attributes for all users need to be specified, as well as a scheme for which attributes correlate with what resources resulting in complex tenant-specific access management that potentially provides excellent privacy, but has difficult implementation architecture. Separation through users and attributes will likely lead to redundancy in the DBs [3] as most data is accessed by several users and therefore has to be encrypted using several keys for each set of attributes or for each user whom has access unless a threshold scheme of numbers of matching attributes can be used. The attribute-based encryption can also be viewed as an IAM implementation as users may be granted certain attributes that allow them to decipher the data they have access to. This way access management may only be enforced by the encryption of data entries, the data you do not have keys to you do not have access to.

The papers presenting encryption as a separation technique pair the data at rest separation with IAM separation, meaning the decryption key becomes the tenant-, user- or attribute identifier. This structuring of a system is quite different from ACX and would therefore require much change in most parts of the platform. An example of separate encryption is KI-ABDR-KS, introduced in 3.7.8.

3.8.4 Separate Database

Separate database means higher costs for the platform and whether the separation is more dependable than other, less costly ones is determined by how it is realised. The system architecture of separate database does however provide the best possible systematic separation technologies; e.g tenant specific encryption, several levels of encrypted data depending on attributes/users, DB address separation and more. When all data is separated like this, the APIs used to communicate with the DBs do not require the same enforcement of policies as there are fewer things that can go wrong between tenants, provided the correct user is connected to the correct API and DB.

Separate DB implementation

Separate DB and separate schema data storage:

In [2] a case study was performed on a business process management system which had its DBs migrated to the cloud from previously on-premise operations. Two different techniques

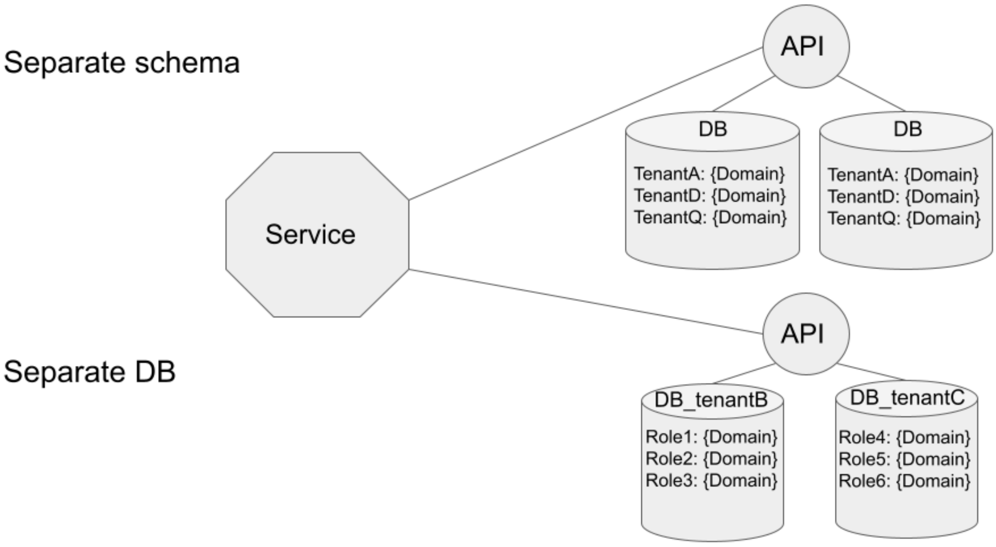


Figure 16: Illustration of the system with both separate schema and separate DB as multitenancy implementation

were used depending on the tenant's requirements within the system, some tenant-isolated components and some dedicated components. These two solutions correspond to separate schema and separate DB for data at rest. The purpose of the case study was to analyse the different separation techniques that can be used for especially data at rest, but also data in use and come to a conclusion of which technology to use for a stable tenancy separation where cost was less of an issue.

The data at rest implementation was document storage realised with the AWS SimpleDB service. Since SimpleDB is a dynamic document store service that scales with use, the data storage was cost efficient and stable. Separate schema was implemented for the tenants with similar structure to each other, and which had the same AC structure, while separate DB was for the more unique data storage and access management solutions.

Figure 16 visualises the system design, the top API serves the separate schema implementations, the bottom one the separate DB implementation.

The separate schema implementation meant each collection/domain within the DB instance was accessed by users of the domain's owner, being the tenant. All domains had the tenant's orgId as key for each domain, meaning the orgId was used when finding a user's tenant's data. The tenants were divided into groups using AWS IAM and access to each domain was limited to one IAM group. The authors meant that separate schema allowed better control of each tenant's use of storage compared to logical, since database performance could be analysed as to each tenant's use with a simple overview of both the number of requests and storage use and modeled subscriptions could be adapted to this analysis. Motivations for choosing separate schema was according to the authors:

"*To address a large number of customers, utilize underlying resources and in turn leverage economies of scale. However,[the] application can be configured to meet individual needs, AC is given to each tenant, and workload of one tenant does not affect the other*"

The shared concepts in the DB instances were the items defined as RBAC roles. These roles were reused between tenants and so access was distributed in the same way, with the domainId and the actual data stored being isolated through the AWS IAM groups and the schemas.

The separate DB instances were implemented by using the domains within each instance as a division of the roles of the organisation. Each DB instance then became an identification of the tenant, allowing the tenant to be less constrained as to how the data was stored. The motivation for separate DBs was:

" *While sharing of resources is possible by some tenants, some components offer critical functionality that requires specific configuration for individual tenants*"

Mentioned with this implementation was that the consequences were *"the lowest degree of sharing, but the highest degree of privacy"*. This technique is both more expensive and more technically advanced than logical or separate schema technologies as maintenance of all DBs demand more of the ones responsible for it. Separate DBs in this implementation allowed greater differences in use by the tenants as they could divide access and define roles more freely when not sharing the AC with other tenants.

Access management was implemented through the AWS IAM service with an RBAC solution. AWS IAM divided tenants into one group each, with the users of the tenant being members of the group. For the instances with shared schema, access for each domain was granted after the user had been authenticated and resulted in the API forwarding the query made by the user to the user's tenant's domain. The IAM groups had userIds and groupIds which represented the user and organisation, where the groupId also was used as the domainId in the database instance. The schema was accessed through an API that ensured the user in the session belonged to the group, meaning a check that the groupId was the same as the domainId was implemented.

Reaching data meant the user had to query the service, which in turn communicated with either the separate schema API or separate DB API, depending on which tenant the user belonged to and their data storage technology. The separate DB API had to match the DB instance name with the orgId, while separate schema matched domain names with the orgId.

Having a DB instance per tenant allowed each domain within to be allocated to RBAC access rights, thus making most of the AC built in to the system where boundaries are made by the DB instead of developer-made logic.

3.9 Processing Separation

Process isolation prevents any interference between processes or access of privileged resources determined in service level agreements. The architects of ACX do not wish do adopt a process isolation technology due to the increased cost for any separation therefore alternatives are mentioned but not proposed although still compared to the current implementation in the analysis. Below the alternatives of separating processes in the cloud are listed.

For the majority of papers concerning separation of data in use there are two motivators: highly confidential information e.g personal information in healthcare, and technologies for upholding SLAs [6][23][39][40][41], this suggests that systems generally use logically separated multitenant processing to reduce costs, and separates processing more when personal information is present. The SLA-motivated technologies used shared instances except those where some tenants due to their high demand in workload could motivate separate VMs.

3.9.1 Shared Instances

No separation means all data is processed equally by all instances owned by the service. There may be distinctions like tenant-specific SLAs in place but in general the data is processed without separation [7]. This works for most cases but depends on how the shared processes are implemented, e.g caches may require greater separation due to critical information being stored by many tenants simultaneously which due to the sensitivity of the data can motivate the increased cost [3]. Processing instances have methods for separating different processes within that function as intended and does not use additional resources to perform, which is why this is the most common implementation. During high workloads the separation may fail resulting in data leakage in unpredictable manners. The most efficient method to avoid failure for shared instances is therefore to ensure unsafe workloads are never reached.

3.9.2 Division of Processing

Some jobs have discrete elements within them that can be processed independently, and by utilising this to spread a single job out over several instances, no single instance may grasp the complete result of the job. The result need to be gathered somewhere however this can be set to a more secure location, e.g the client or in the hypervisor [18] where the result is sent to its destination and only one user can view it. Dividing jobs is possible for many services and the effect on latency is dependent on how the division is performed as well as where the processes are performed in relation to each other. In [19] all jobs are performed on the same physical machine but in separate VMs and where the result and the sensitive data used in the processes are stored in the VMM which is not directly communicable by clients.

3.9.3 Virtual machine Separation

Dividing a hardware server into several logically separate instances, each of the virtual machines may only process information from a single tenant in a given time period, or may only process a single tenant's data at any given time [12]. The VM has its processing dimensions decided by its owner thus efficient use of resources is achievable with chosen dedicated capacity per VM. This is achieved through standard technologies like memory segmentation and page mapping, constructing a separated address space for each process effectively creating VMs for data in use since a process cannot access memory regions outside its address space [42].

Whenever VMs do not use their entire capacity, each unused capacity may add up to significant losses in performance. As such more processing power than the minimum results in either higher costs for the system, or standby VMs serving the extra workload which may be shared with other tenants and results in reducing the privacy built up by the separate VMs. The compromise between adapting to everyday use or adapting to processing spikes leads to unused capacity somewhere in the system [7].

3.9.4 Server Separation

Since vulnerabilities in VM separation may create a false sense of security while attackers still are able to reach several VMs on the same hardware through e.g rootkits, separating the servers completely ensures those with access to one server cannot reach the other without vulnerabilities in other, more trusted domains [12]. Accidental errors in the VM may lead to data of different tenants being mixed and leaked unintentionally to a random user. This may be a fault in the hardware, the result of an overloaded system, or faulty code running on the instance [3]. By not sharing processing power, the worst case of accidental leakage is within the tenant. This separation results in only sharing processing software between tenants and therefore means expensive and complex system to manage. There is no multitenancy in this solution, resulting in high costs of maintenance, and since the technology is highly inefficient with resources it has little support in scientific papers.

3.10 AWS Technologies and Services

ACX uses almost exclusively AWS services, which provide the backbone of the platform. In this section the AWS services related to multitenancy used by ACX are described.

3.10.1 Serverless

Serverless is not a service in itself but a concept that several services utilise. It means that customers of serverless services pay for amount of data processed and stored. Through its inherent properties serverless offers automatic scaling, service integration and faster response times [17]. Some serverless services are called FaaS, or function as a service, signifying the use cases of the services as smaller APIs. The customer specifies a callable unit, what runtime environment it uses and events for which this unit is called and the code within is executed.

The CSP has containers with a runtime environment and the specified callable unit distributed geographically to fit the specifications for latency the client has. Although serverless has no specific location of execution for the costumer, the CSP treats it as a containerised service with more responsibility for the container than other services the CSP provides. The customer still has to specify the location of the serverless service as to which CSP-owned server hall it will be installed on, but only has one instance from their perspective while the CSP provides the scaling and load balancing. The simplicity the costumer buys with serverless also reduces their control over the specifications of the hardware and software the service is running on meaning more trust is required for CSP integrity. All services described below are serverless, and are used to great extent by ACX because of their simplicity.

The main contributing factors for services to move to serverless functionalities are that the speed of deployment is greater, the cost of running instances and maintenance is cut and less responsibility is on the customer, more on the CSP [17].

Axis requires ACX to only utilise serverless services, thus these are the only services considered for ACX in this thesis.

3.10.2 DynamoDB

DynamoDB is a NoSQL-, or document-DB, and therefore stores data in a file format for easier retrieval and greater management options. DynamoDB has many possible configurations for how the data is stored, shared metadata parameters like data types, and offer several querying techniques for high configurability. DynamoDB can implement other DB technologies, e.g MongoDB to offer more configuration, which further increases its malleability.

In ACX DynamoDB is used to store device data within its services, with a few keys and a document as value which contains the description of all properties the devices has, which varies depending on what information the services need, it could be firmware version, applications installed, its model, serial number, etc..

3.10.3 PostgresDB for RDS

The relational database service (RDS) offers many relational storage types, of which ACX uses PostgresDB. RDS is a scalable data storage service like DynamoDB, with ease of use and high availability as its main features. ACX uses it for storing relational device data, where more structured information is to be processed, and it is necessary to be able to search all the devices' fields i.e in the device inventory which stores the device groupings, deviceIds, serial numbers, models, etc..

3.10.4 NeptuneDB

Neptune is a serverless graph database service, which provides fast response times for scaling solutions where data is highly connected with complex relationships. It has built in data security and backup which makes it a good candidate for auth-services. Graph storage makes storage with many relationships easier to handle and most importantly makes queries of entries simpler than standard relational storage. Neptune is meant for highly connected data and a key feature is the ability to give properties to the connections between different data entries to specify their relationship.

ACX uses Neptune to store the access management model, which contains mappings between users, roles, operations and resources. It is in a graph to reduce data redundancy, and to better map how access is distributed in ACX. Graphs become necessary since regular relational storage cannot indicate the roles, their operations, the users that have access to the roles, and which resources the users have access to the role on in a simple way.

3.10.5 S3

The simple storage service (S3) is a serverless object storage service which cheaply stores large amounts of data with high scalability as its greatest advantage. It stores unstructured data and is used mainly for large sets of data that do not require the greatest performance.

ACX uses S3 for unstructured data, including especially videos from cameras, and sound from speaker-microphone devices. It is also used to store user access rights which the services query to then use in access enforcement.

3.10.6 Lambda

Lambda is a serverless computing resource that executes when called, e.g by being triggered through an event in other AWS services adding value and increasing SoD for that transaction. No configurations for OS, hardware or other specifications are available making it an intuitive service that has direct functionality. An example could be expanding a zipped file fetched from an S3 bucket before the file arrives in the client. Because of its few configurations cannot be adapted to fit all needs, however its purpose is to supply simple processing to its owners and there are options like the EC2 which offer more configurations when desired.

ACX uses Lambda for most processing, as APIs to send data to the correct services and for computations for the services, e.g generate new Ids for devices and users, perform the requests sent by users, etc..

4 ACX Platform Specification

A brief introduction to the platform is found in section 1.1, in this section the system design is presented with more details for specific parts of the platform.

The goal for ACX is to provide a federated integration point to Axis devices, meaning the focus is on simplifying the installation, maintenance and use of Axis devices for all business partners and customers. The thesis' main focus is device management (DM) as all concepts of multitenancy are similar across ACX. To cover all microservices and analyse each's separation and general security implementations would also take too much time for this thesis' scope.

Figure 17 below is an illustration of how ACX is used. Partner users, Axis users and machine users all connect with ACX to interact with the services Axis provide for their devices. The ACX API is integrated with an ELB to distribute work across several identical service instances not illustrated here. Each service has their own DB with data essential for the service's purposes, and AC is enforced in each service while the management and structure of the RBAC-implementation is in IAM.

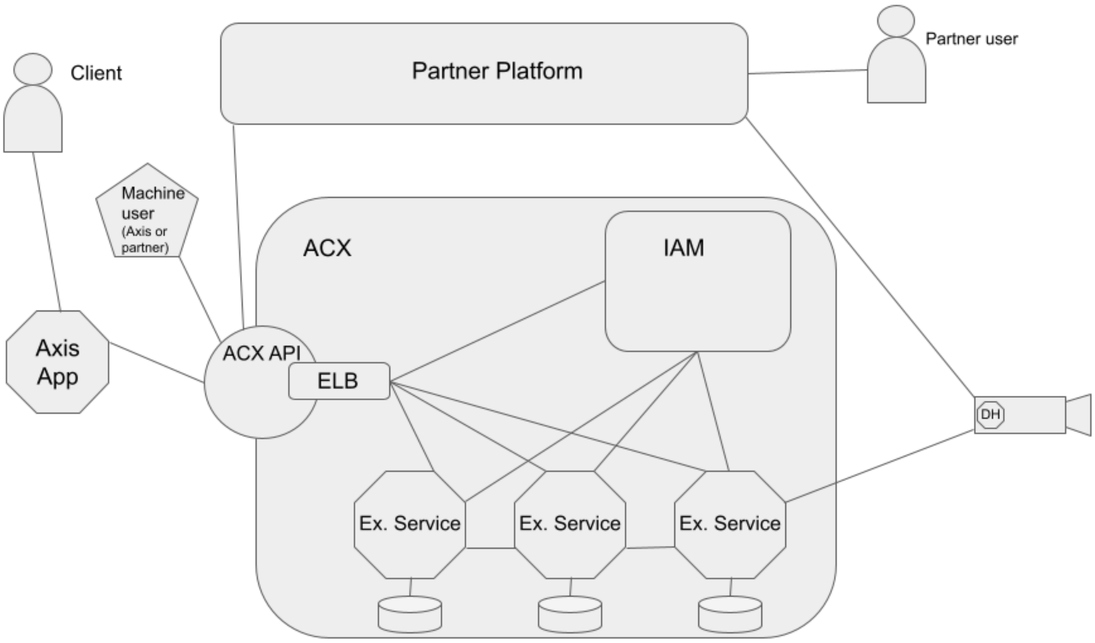


Figure 17: Overview of the architecture of ACX

4.1 Project development and deployment

The platform and its services are developed by DevOps teams working independently on each service. It is configured with few overarching policies and is made that way to accommodate for individual solutions on a service level by each team. The services developed are divided into microservices the teams have responsibility for, resulting in many teams' efforts being combined to an end-to-end solution. The independence of each team leads to tailored solutions for each service and thus ACX becomes many confined services federated through their interdependence to execute complete solutions and the overarching policies determined centrally. The independence and by extension decentralisation of responsibility leads to a certain degree of SoD where sensitive data only is reached by the microservices in need of it. The decentralisation also means that all requests within a service are locally evaluated and enforced, thus permission is granted or denied in a decentralised manner.

As the teams are independently developing their services, the resulting products generally have different properties although they mostly use the same tech stack. All services have repositories on Github, and are deployed into mainly AWS containers: several Lambdas with simple notification services (SNS) and simple queuing services (SQS) helpers for them. SNS and SQS are small services performing the job specified in the name, SNS sending notifications for events that developers or the other services have determined are notification-worthy, and SQS storing requests when the e.g Lambda instances are at capacity. Data storage in the cloud is mainly in AWS DynamoDB, RDS and S3, with IAM also using Neptune, all of which are serverless AWS services introduced in section 3.10.

The DevOps development cycle means continuous delivery and integration with both automated and manual testing to ensure the use cases are actualised as intended. The constant upkeep of dependencies means fewer vulnerabilities inherent to the system, an otherwise great risk factor for software systems. Vulnerabilities found in packages or in the code are discovered both manually and with automatic processes, which keeps the services' foundation updated.

4.2 Architecture

4.2.1 Microservice interconnections

The system's services revolve around a few important central nodes, the ones of interest to the thesis being the Tokenizer (Tzr) which generates JWTs that all other services on ACX use for the session, validates user authentication, and regenerates new tokens when session TTL is surpassed; the security scope service (SSS) which provides authorisation scopes for all services; and the Device Inventory (DI), a data storage service that maps serial numbers, Resource Groups (RG) and device Ids to the owner. All devices have a deviceId which are saved together with the serial number in the DI. The DI distributes the deviceId so that services only know what action to perform to which device.

The services above are connected to most other services, and provide the processes required by other microservices to in turn perform their responsibilities. Most other services are isolated with few connections to other resources except where it receives its instructions and where the service's results are delivered. The microservices have their own databases, queuing systems, and other surrounding automated processes attempting to make the system as a whole more stable with fewer outside dependencies.

4.2.2 Authentication use case

The figure below in conjunction with the enumerated list illustrate the steps ACX takes for signing a user in to a new login session. Authentication here signifies the starting of a session between a client and the login service after the user of the client is verified. The login flow is ACX's implementation of the OIDC authentication protocol.

1. The user locates the app containing the ACX resources the user wishes to reach through their client, and presses the login button.

2. The app redirects the user first to the login service (LS) to view the current state of the session. The session has a refresh token which could be used to renew the session if it is not too old.

3. If the last session is deleted or older than 24 hours, the user is once again redirected, this time to the identity provider (IdP).

4. IdP is reached and displayed in the client as a login prompt. This initiates the OIDC authorisation code flow.

5. The user inserts their credentials into the client and submits it to the IdP.

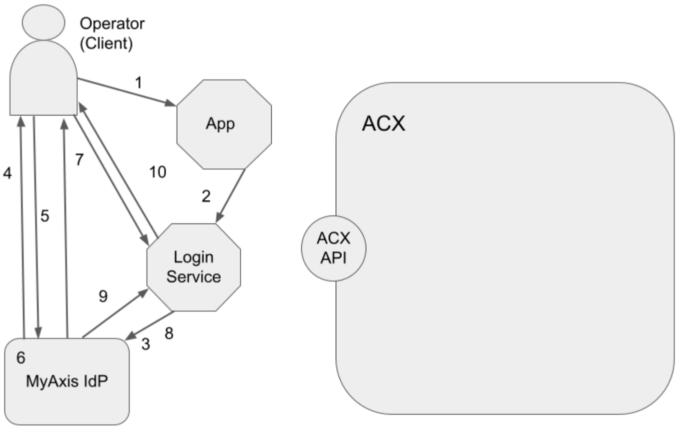


Figure 18: ACX Login flowchart

6. The IdP receives the credentials and verifies them by comparing them to its auth store.
7. If the user is authorised, the IdP responds to the client by redirecting it to the login service with the OIDC code.
8. The LS receives the OIDC code and client secret from the client and the LS forwards this to the IdP.
9. The IdP answers the LS with a token set containing an Id token, access token(AT) and refresh token.
10. The LS receives the token set, encrypts it and a nonce homomorphically thereby creating a session cookie (SC). The SC is sent to the client.

The SC contains three tokens, the AT represents the authorisation of the login session both for ACX and all other internet services Axis provides, making contextualisation of a session for ACX specifically unfeasible with this token. The second token is the identity token verifying the session is started by a certain user, and the third is the refresh token used for refreshing the login session when the TTL of the refresh token runs out. If the user belongs to an organisation with an external AD, the AD is called and provides a token to myAxis which transforms it into an axis SC. The login session lasts 24 hours and requires a new login after this TTL.

4.2.3 Authorisation use case

Below is a figure illustrating the steps taken when a user initiates a session with ACX and attempts to perform an action within the platform. The steps requiring authorisation use the OAuth framework presented in section 3.5.1.

1. The client has a SC from the login phase above, but the SC is only useful as a login SC. To start a session with ACX the client needs to send the SC back to the LS.
2. The LS receives the SC, decrypts it and returns the AT within, which is generated by the IdP previously in step 9 in figure 18.
3. The client sends the AT to the ACX API which lists the organisations corresponding to this user to the client.
4. The user sends the AT, the organisation, an action, and the resource the action is to be performed on to the app which forwards it to the ACX API.

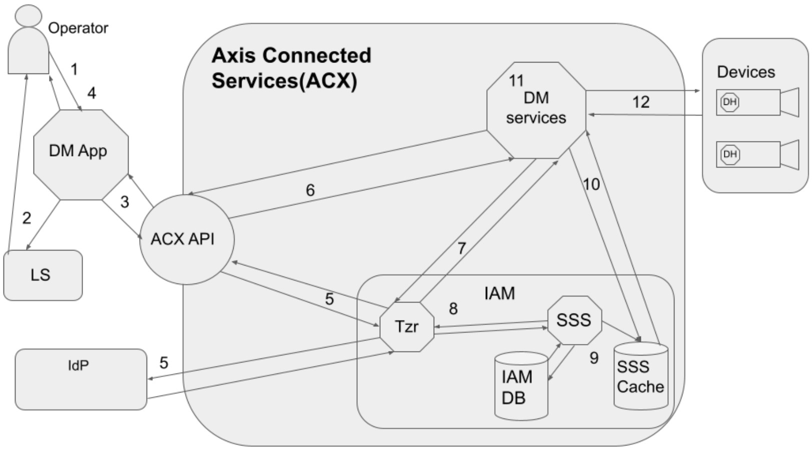


Figure 19: ACX authorisation flowchart

5. The API calls the Tzr, first verifying the AT by sending a validation request to the IdP, then responding with the AT used for authentication within ACX, called the JWT.
6. The API sends the client's request to the service the operator wishes to use, in this case device management (DM), with the new JWT.
7. DM requests validation of the JWT from the Tzr.
8. The Tzr validates, then requests the security scope service (SSS) to fetch the security scope of the user.
9. The SSS queries the IAM DB for the security scope of the user, security scope being the access control list (ACL) of all the users access rights. The security scope is stored in a cache with the JWT as the key.
10. After receiving validation from the Tzr, DM queries the SSS cache for the security scope and subsequently receives it.
11. DM parses the scope, finds the correct organisation from the request, action/operation and the resource(if the resource is within the user's access privileges) from the ACL. DM evaluates the request as to whether it is permissible for that service by that user.
12. DM acts upon the result of the evaluation.

After the user enters ACX, the login SC is replaced by a JWT containing the AT and the SC. This JWT is refreshed every 5 minutes(the TTL of the AT). When the TTL is reached, the Tzr communicates with the IdP to receive a new AT used in the JWT for continuation of the session. Every time the JWT is refreshed, the security scope in the SSS cache is as well.

The authorisation within ACX is designed to allow users to operate within several contexts simultaneously since several users have access to many tenants' data as installers or other operators of the Axis devices. The users cannot access different tenant's data simultaneously, as this is dissuaded by AC policies.

The IAM module is meant to centralise access management, storing the RBAC relations as well as generating the session JWTs. Due to the intended decentralisation of the platform, more division of duties has not been performed, resulting in most decisions being made independently in the services.

4.2.4 Identity and access management

The identity management is centralised through the LS and Tzr, performing the verification and validation of a user's credentials in a session. As seen in the authentication use case above, identity is provided by an IdP by performing the OIDC protocol and results in an AT being delivered to ACX. The JWT is used to authenticate the user every TTL by the service by requesting validation of the JWT by the Tzr and the Tzr requesting validation of the JWT by the IdP.

The access permissions are stored centrally in a NeptuneDB instance called the IAM DB, these permissions are fetched by the SSS and stored in its cache for all services to retrieve. When a request is received by a service, the enforcement of the access rights is performed in the services as described in section 4.2.3. Thus the AC model is central while the enforcement is decentralised in the services.

Tenant administrators may create RGs and assign access to them for principals, human or machine users, whom then have access to perform certain operations on the devices depending on which role they are paired with. A RG could mean all cameras serving a geographical area, certain device types, or other divisions the tenant administrator wishes. The tree is as vertical or horizontal as the tenant wishes, giving them freedom to design their AC structure as they please.

All requests within ACX result in communication to IAM from the service, where the principal's access is fetched. Users with access in several different organisations may perform operations on any resource they have access to irrespective of what organisation the user used when logging in although only in one context per request. The SSS queries the IAM DB to retrieve the privileges of the user, and converts them into an ACL, also known as the security scope, to make the access more easily read by the services. The services then use the ACL, the orgId, the JWT and the request to determine whether access is granted.

The IAM of ACX subscribes to Role-Based Access Control(RBAC), abstractly presented in section 3.7.3. A RG access tree is generated for each tenant and has a UUID as root, which also is the orgId. Access rights are inherited in the tree, meaning a user with access to the topmost RG have access to all RGs, a user with access to the second highest RG have access to that RG and all its branches. Access to a RG also means access to all of its child nodes down to the leaf, this way an overview of the organisation is both intuitive and inclusive. The access provided is slightly different depending on which service is in use, yet it is all RBAC-implementations. A principal with certain access took on a role to perform an operation upon an object within the access right of that principal. Roles are reused between services, but with different operations defined for them depending on the service. The tree is illustrated in figure 20 where an example RG is in focus in the right corner. The IAM DB only contained the RGs of the tenants, while another service is responsible for translating a RG to a list of resources within by relating their deviceIds, a UUID used for identification within ACX. Outside the platform the serial number is used instead.

The principal of ACX's RBAC is connected to a certain access as well as one or several roles, and when performing an operation, the access represents a connection between the role and the object specifically for this principal and through the role, certain operations can be performed on the objects by a principal with the correct access in the SSS's ACL. The ACL is always the same for any one principal unless updated by the tenant; it contains all access rights of that principal. The ACL is a list of entries of roles-, access- and resource entries which specify how the principal may operate on any given resource.

The IAM model of ACX contains a set of roles which the tenant administrators use to distribute privileges within the organisation which are not used outside the IAM module, and another set of roles exist in each service. Although the two sets of roles are separate in use, they are stored together in the IAM DB.

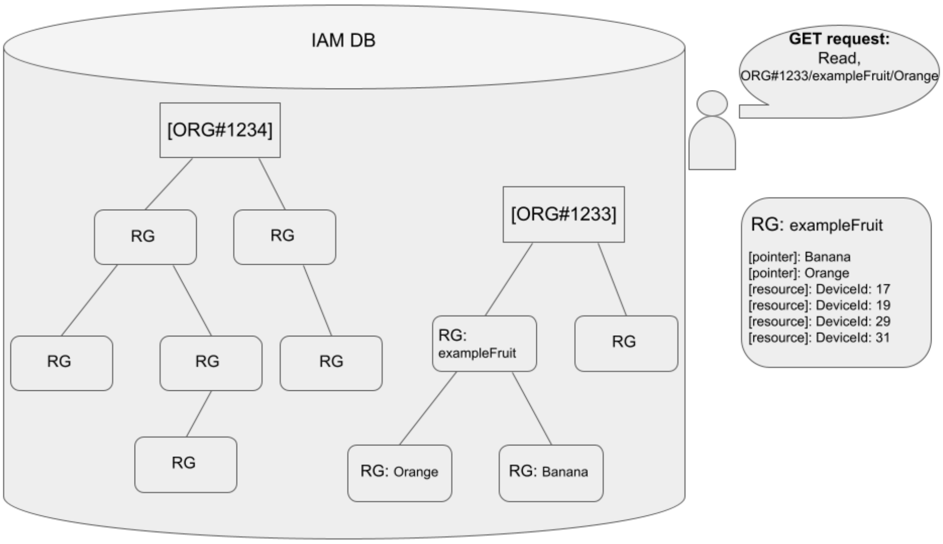


Figure 20: The structure of data in the RG access tree

In figure 21 the RBAC system is visualised. A principal is both machine users and human users, the role specifies the operations a principal can perform, the access specifies whether the principal may interact with the resource at all and the resource is the device, usually a camera that the principal operated on if the access and role are valid.

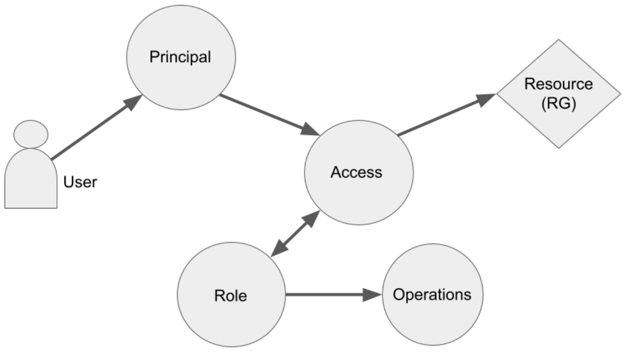
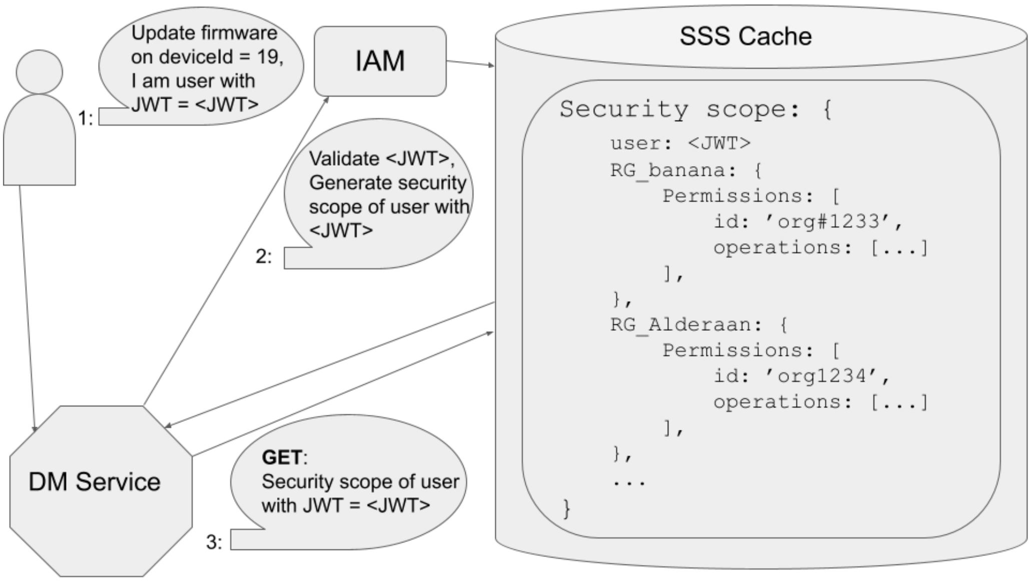


Figure 21: ACX IAM

The security scope/ACL contains all of the access rights of the user, a JSON object with the structure visible in figure 22. The figure shows a request for an operation from a user on ACX. The 'operations' mentioned are determined by the services independently and may correspond to roles with several- or a single action. ACL is a broadly used AC implementation with clearly stated permissions that the service can iterate over. The translation of RBAC to security scope is simple in ACX since the roles and RGs remain as fields in the security scope, making the adaption of the access tree into the ACL intuitive. The roles correspond to 'operations'-field and the RGs correspond to the 'objects'-field.

1. The request reaches the appropriate service, in this case a Device management(DM) service
2. The service requests JWT validation and security scope generation from Tzr and SSS respectively
3. The security scope, generated by the SSS is stored in the SSS cache, which the service then queries to retrieve the scope.


1:
Update firmware on deviceId = 19, I am user with JWT = <JWT>
IAM
2:
Validate <JWT>, Generate security scope of user with <JWT>
SSS Cache
DM Service
3:
GET: Security scope of user with JWT = <JWT>

```
Security scope: {
    user: <JWT>
    RG_banana: {
        Permissions: [
            id: 'org#1233',
            operations: [...]
        ],
    },
    RG_Alderaan: {
        Permissions: [
            id: 'org1234',
            operations: [...]
        ],
    },
    ...
}
```

Figure 22: Security scope illustration

The service searches the security scope/ACL to find the operation and resource pair matching the request. The service also evaluates whether the operation in the request is equivalent to the role specified in the ACL. The ACL contains all permissions for all services and RGs of the user it corresponds to, resulting in large ACLs being generated for users with access to many tenants and RGs. The ACL is in its entirety fetched from the SSS cache to service before the service evaluates the permissions.

The services of IAM use NeptuneDB for the IAM DB, S3 for the SSS cache, and otherwise lambdas and smaller modules like SQS and SNS for processing and monitoring.

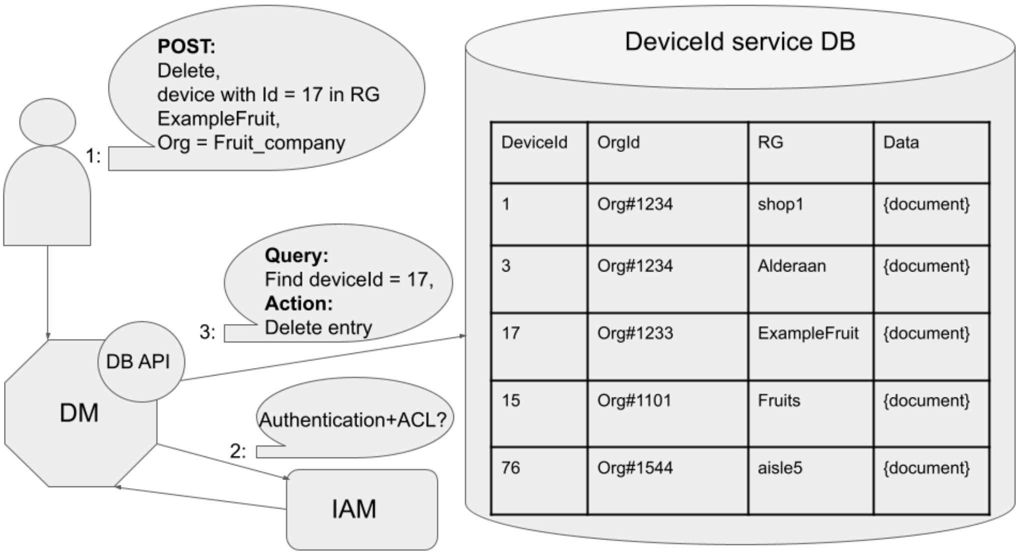
4.3 Current state of Multitenancy

4.3.1 Data at rest

Axis devices on ACX are divided into RGs. The RGs can be parts of other, larger RGs in a tree structure with devices on all nodes except the root where the orgId is presented. This tree is stored in the IAM module by a graph database in AWS NeptuneDB, and shows which devices belong together in RGs, described more in section 4.2.4.

All DB instances on ACX have a single service that queries it. If an application requires information from several services, requests are sent to the services which then fetch the data from their DBs. Wherever data is stored on ACX, the DB instance is encrypted with a key for all data irrespective of tenant-alignment. The logical separation used on the ACX DBs is partially RLS implemented by having data belonging to a specific tenant also contain its tenant's orgId as one of the fields, used for all queries as a filter to only retrieve data with this orgId. All requests to a service have the orgId added to the query, but each service has their own implementation of the filter. Not all services require the use of the orgId field in DB queries, e.g the DeviceId service where orgId filtering is optional. This results in query responses filtered with the user's tenant when that field is used while there is no at rest separation between tenants for the requests not using it. AC still enforces its policies on all requests to ensure the principal has the access, role and operation necessary. The orgId is both fetched and used differently depending on the service since they are independently developed. Figure 23 visualises the document storage partial RLS used for the Device Id service where DeviceId is the primary key, and orgId and RG are optional. The figure shows an example of how a query to a DB can look in ACX. The organisation value sent by the user is translated to its corresponding orgId within ACX. The orgId being optional means RLS is partially implemented yet not utilised, since queries to the DB by the service can omit it. This results in the implementation not being an efficient separator.

Not including any separation between data is not recommended by any source found, since this means no at rest structural defenses exist to support the isolation of users and tenants for those services that do not utilise RLS. AC verifies unauthorised access is dissuaded however this should not be the only separator for DBs. The ACX approach allows the data to be identified as belonging to a certain tenant, yet a user belonging to another tenant can still query it and no issues are found by the service unless the ACL or the orgId the user sent in specifies this operation is not allowed. The ACL should always work to separate users, but using AC as the only separator between tenants mean there are effectively no tenants, only users. The users have access rights, which belong to different groups of resources and are stored together.


1:
POST: Delete, device with Id = 17 in RG ExampleFruit, Org = Fruit_company
3:
Query: Find deviceId = 17, **Action:** Delete entry
2:
Authentication+ACL?
DB API
DM
IAM
DeviceId service DB

DeviceId	OrgId	RG	Data
1	Org#1234	shop1	{document}
3	Org#1234	Alderaan	{document}
17	Org#1233	ExampleFruit	{document}
15	Org#1101	Fruits	{document}
76	Org#1544	aisle5	{document}

Figure 23: ACX data at rest and query structure

ACX has minimal information about its users and devices by decree of GDPR, with the email address being the only direct information possible to be attained by a user. The email address however, is not stored in any ACX service for longer than user invites are pending. The emails of users are the mapping between users' Ids and email addresses in Axis' IdP (myAxis), which is not part of ACX's area of responsibility but instead of Axis as a whole, making it out of scope for this thesis. The sensitivity of ACX data is concerning this email address protected using GDPR processes via the IdP and it does not warrant drastic increase in separation since it is handled carefully and separate from other platform data already.

The data at rest on ACX is in several AWS storage services: AWS S3, DynamoDB, PostgresDB for the relational database service (RDS) and Neptune are used, all of which are serverless and have different purposes. S3 storage is used as caches and for storage of larger objects, DynamoDB is a document based database where most device meta-data is stored, PostgresDB for RDS is a relational database used for non-IAM relational storage, and NeptuneDB is used for storage of access rights in graphs. For sensitive metadata like keys and other secret values, the AWS KMS is used to control access to the values as tightly as possible.

All storage in ACX uses AWS's built in service for encryption and decryption, AES-256, which AWS handles for ACX so that the data is decrypted given that the requests are authenticated and authorised as ACX traffic.

4.3.2 Data in use

ACX strives to use as much serverless processing as possible, meaning structured processing safeguards are difficult to uphold since the physical machine the service runs on may change at any time at the whim of AWS. Therefore the AWS security measures are necessarily trusted for intra-AWS communication. The processing performed is spread out over several AWS server locations, including several European and US locations. The services used for serverless are Lambda, S3, dynamoDB, Neptune, SNS and SQS. Of these, Lambda, SNS and SQS are processors. Since the physical instances are not controlled the separation between tenants within ACX is completely determined by AWS, probably implemented with shared instances as ACX is a single tenant from AWS' perspective. No other separation within the serverless processing instances are performed, although division of processing or separate processors are possible with more instances yet not implemented as it is not considered sufficiently motivated for ACX.

4.3.3 Data in transit

The platform has minimal transport of data as a goal, translating to data being stored by every service so that the service have as independent functionality as possible. However communication between services is still necessary and this is protected with mTLS connections, upheld whenever there is machine to machine communication, and TLS otherwise. In some cases even certificate pinning is implemented, also between machines in those cases. Here AWS Elastic Load Balancer and AWS Cloudfront are used to divide the workload evenly between instances and ensure data reaches its destination respectively.

4.3.4 Identity and access management

Except the AC model, there is no other separation between tenants than the filters used for fetching data from the DBs which as mentioned above, is optional. The security scopes/ACLs taken from the IAM DB include all access rights across all tenants the principal has access within, however access can only be utilised within one tenant at a time. The session JWT used on ACX only specifies a session for the user, not for which tenant is specified in each request, and the AC within the services requires the tenant to be specified in the request sent from the client. These three technologies are the multitenant implementations of ACX.

4.4 Device Management(DM)

As ACX is a large structure, this thesis could only focus on some parts to reach a conclusion in the time set for the project. Device management and IAM are the focus of this thesis and are the parts analysed for tenant isolation. This is done because all services share multitenant architecture, and to include the other services would bring little new architectural information.

DM has several services all built up of microservices, and makes use of AWS SQS, Lambda, SNS, DynamoDB and S3. Axis DM is a constellation of services which provide functionality to the application ADMX, designed for simplifying the overview of products from Axis a customer has. It serves as a gathering of all necessary information and functionalities to maintain the products in use. ADMX provides an overview of how the system is connected to the internet and what devices contain which technologies for the users while the microservices in DM on ACX does the processing and storing of ADMX's data. Its responsibilities are to contain an inventory of devices a given user has access to, their attributes and possible actions for these devices, including Axis camera application platform (ACAP) applications, which are software that is processed and detected events in the video stream from the camera it is installed on. The information stored for each device in the DM services is:

- Device serial number

- Device model
- Resource Group
- Warranty
- End of Service
- End of Life
- Firmware/OS updates
- Currently installed ACAP applications
- Pending ACAP installations
- Available ACAP updates

Through ADMX, tasks intended for devices are created and sent to the ACX API which forwards it to the correct microservice which has SQS for queues, Lambdas for validating the operations, logging them, checking licensing and warranty, and performs the operation. The DBs connected to DM has replicated instances with only read-capabilities which has a load balancer between to handle heavier workloads, with one instance representing the truth where CRUD (create, read, update, delete) are all possible for authorised users, with a backup not reached from outside the system.

4.4.1 Separation within DM

DM data is stored in shared databases with all tenants, using logical separation with RLS. Whenever a user belonging to any tenant requests access to DM, the DB API has a filter added to the request within the services which are implemented in a case by case manner, where different solutions are necessary for DynamoDB and S3 since the storage of that data is differently structured. For the DI, which stores deviceId, serial number, resource group affiliation and more device specific data, the structure is relational with RLS, and the query sorted first on orgId or RG specified in the request, then the rest of the request. For document DBs like the deviceId database, separation with RLS is optional and implemented by filtering requests by either RG or orgId which are additional key fields in each document, however requests only need to specify the deviceId thus resulting in no at rest separation for those requests. AC separation is still performed thus the tenants are separated even without RLS.

The IAM of DM is as described above, where the service fetches the security scope from the IAM DB and enforces the rights established in it.

5 System analysis

This section contains an analysis of ACX regarding its practices and implementation of tenancy separation technologies, i.e the data at rest, data in use, access management and their interactions. Section 5.1 presents an issue concerning the AC of ACX that depended on the multitenancy of the platform. The analysis results in a collection of flaws that introduce or could introduce unnecessary vulnerabilities.

5.1 A multitenancy issue

An issue concerning multitenancy was found by testers during the thesis caused by a simple coding mistake; access rights of principals could be modified in one tenant from a principal in another tenant in the IAM module. A user Alice in tenant A could increase the privileges of Bob in tenant B within the administration of the tenant for ACX, meaning Alice could e.g promote Bob to an admin of all users of tenant B.

The error was an if-statement which did not cover all possible request parameters, and this error could have happened in any system, yet due to the choice of logical separation and its partially incomplete implementation this was a larger issue than it could have been with greater separation of tenants. Since users are signed in without tenant context the use cases that need to be covered by the if-statements of all services become unnecessarily many.

The issue was easily fixed once found, however its prevalence proves the tenancy separation needs be increased or at least be more thoroughly implemented as this is a bug involving manipulation of another organisation's access structure.

5.2 Data at rest

The data of ACX is encrypted with one key per DB, and when queried its decryption key is used to convert it to plaintext by AWS in the background. ACX trusts AWS performs this action, thus ACX can assume their data always is decrypted when queried correctly and encrypted otherwise. Although holding the key themselves would mean greater control of their resources, Axis gains nothing from that control since there has been no incidents of AWS not fulfilling their responsibility and AWS's reputation for fair and competent treatment is more important than any data ACX may hold.

All tenants have the same types of data stored in ACX and all utilise the access management on the platform, therefore the implementation of logical separation means efficient sharing of features within these areas between tenants. Due to this high degree of similarities low separation is preferred for maximising resource efficiency.

Queries to the DBs are evaluated by the DB's service to find what resources the requester has access to, an especially important part for logical separation since this is the only isolation between tenants as well as between users. The three fields used for filtering the data once the DB is searched include the orgId thus ACX can implement RLS although it is optional for some services, e.g in the deviceId service. Queries within the deviceId service can use the orgId if this is specified in the request from the client however do not have to since the orgId is an optional field. Thus queries search the entire database and separation is reliant on the AC of the service when the orgId is not utilised. The data is still protected through the security scope/ACL from the SSS used for the AC enforcement which verifies the principal has access to the resource to be fetched. The AC becomes the only tenant separator for the deviceId service when orgId is not used thus it has the lowest level of separation that still is separated.

Not including any separation between data other than AC is avoided by all sources found, since this means no structural defenses exist to support the isolation of users and tenants. As stated in section 3.7.2, multitenant AC should only supplement at rest separation, not be the main separator. The implementation is partially the cause of the issue in section 5.1 since tenants are only separated through the ACL which may contain vulnerabilities that include incorrect interpretation or lack coverage of some use cases. The ACX approach allows the data to be identified as belonging to a certain tenant through the orgId column in the DBs, yet a user belonging to another tenant can still query any data and no issues are found by the service unless the ACL and/or the API specifies this operation is not allowed through either viewing the request's parameters or the ACL's policies. The ACL should always work to separate users, but using AC as the only separator between tenants means there are in fact no tenants, only users. Below is an example of the potential issues with this implementation.

A query by a user of organisation A requests to view the information of the device with deviceId 17 which belongs to organisation B(a tenant the user does not have privileges in):

1. The user logs in, and since they have an account receive their SC, and after more communication, the JWT corresponding to a session within ACX.
2. The user requests service1 to read the information of the device with deviceId 17, and sends organisation B as their organisation.
3. Service1 validates the JWT by communicating with the Tzr, then requests the security scope of the user from the SSS cache.
4. The security scope/ACL reaches the service which evaluates the user's permissions. The service checks the user's requested tenant(org B), the user's rights within this tenant, and possibly other policies specific to the service or in general to the platform.
5. The service reaches a result, and enforces the decision by acting on it, either by querying its DB for device 17, or denying the request.

Figure 24 shows the steps above. The request parameters contain the organisation for many services but not all.

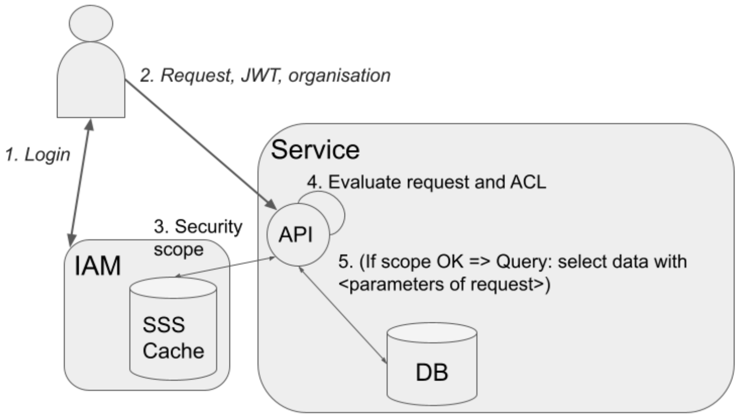

2. Request, JWT, organisation
1. Login
Service
4. Evaluate request and ACL
3. Security scope
API
IAM
5. (If scope OK => Query: select data with <parameters of request>)
SSS Cache
DB

Figure 24: Current DB queries

A simple mistake could here be that the service sees the user only has access to one tenant, which also is stated in the ACL where the service reads access to org A as B by it automatically trusting the user and only taking the input without AC, or if the service assumes orgIds are secret values which only users of the organisation know and therefore starts checking access rights immediately using the organisation from user input, regardless of the orgId stated in the ACL. Another possible bug could be if the RG of device17 in org B is named identically to another RG in org A the service may not look into the organisation the device belongs to.

The current tools against the incorrect retrieval of data belonging to another tenant are the ACL evaluations done by each service and the testing and analysis of the services performed by developers including threat modeling, code review, regression testing and penetration testing. Although these tools ensure the code works almost always, an increase in separation would support the AC and the policies tested thus improving ACX's reliability in tenant isolation.

ACX is best suited for separate schema or logical separation given the sensitivity of the data stored, the amount of features that can be shared between tenants, and the need for scalability and economic optimisation however the current implementation has greater vulnerabilities than advised and should therefore increase its separation. The vulnerability lies in the lack of structural support for the policies of ACX, currently enforced in the API only for some services and the API's ability to enforce the security scope.

5.3 Authentication

The first authentication before reaching ACX, being the one done towards Axis as a whole, is performed with an IdP and the LS, both outside of ACX's domain, acting as the parties involved with the client to log in the user to Axis as shown in figure 18. The recommendation from IdPs is to use an intermediary for authentication of a user [43], which is the method used by ACX with the LS. The mediator makes the client unable to tamper with the protocol except when the login credentials are sent, a functionality that also could be implemented through input validation, RegEx and similar [44] at the IdP and Axis side, however this way it can be gathered in the LS making analysis of the process of authentication for Axis clearer.

Authentication for sessions within ACX are performed differently as the user already has the SC from the authentication process above, and sends it back to the LS which decrypts it and returns the AT from within the SC. The AT is then used to authenticate the user in ACX. As seen in figure 18 and 19, the round-trip of the AT is unnecessary and adds nothing to the reliability, integrity or confidentiality of the session. Instead this extra step with the AT from the client to ACX ensures the session is with the user, but is not tied to any tenant. The AT contains a reference to the principal's access rights within ACX but not which tenant they are acting within.

The flow chart in figure 25 shows the client providing the AT to ACX and it results in all access of the user being represented by the JWT in ACX created with this AT. The IdP has

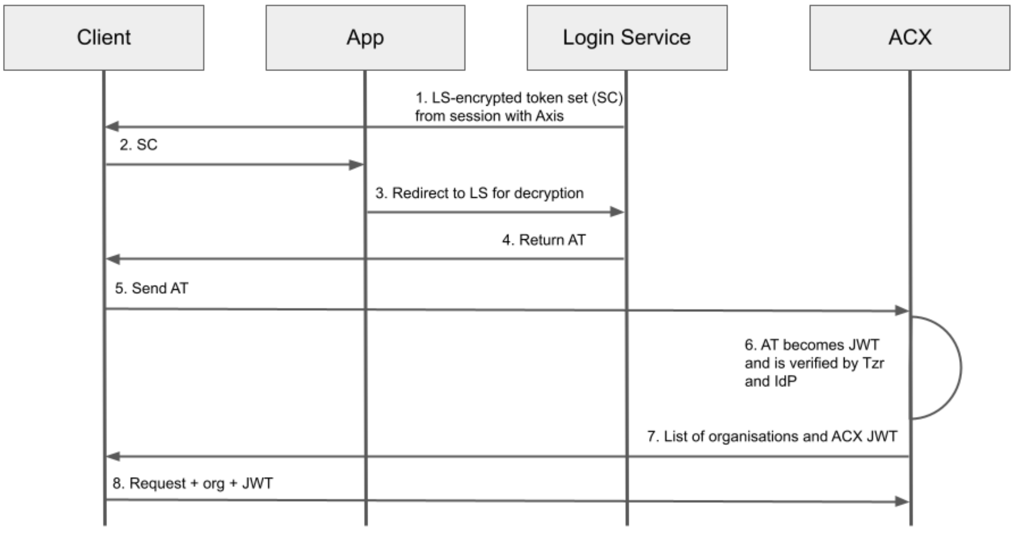

Client
App
Login Service
ACX
1. LS-encrypted token set (SC) from session with Axis
2. SC
3. Redirect to LS for decryption
4. Return AT
5. Send AT
6. AT becomes JWT and is verified by Tzr and IdP
7. List of organisations and ACX JWT
8. Request + org + JWT

Figure 25: Current authentication in ACX with AT

no knowledge of anything within ACX except the login credentials of the users of the platform thus the AT represents the user and not their context, thus making the context the user's. This user context is not what ACX wants to use which is why the organisation is sent with each request. ACX requires both the user to be authenticated and their affiliation with a certain tenant verified before requests are authorised. While this is possible in the implementation used by ACX, having the organisation be dependant on each request and use the larger context of the user as the session's parameters is a combination of two separate session management solutions. Sessions are for having restrained and specific access rights derived from the context of the authentication performed before its initiation[36] which ACX could improve upon by combining the user scope of the session, and the more granular access scope of the orgId.

Figure 26 visualises the difference in the session context, and the request context that contains the organisation the request is intended for. The request context and session context should logically match since it is natural that the requests should use the context of the user and vice versa.

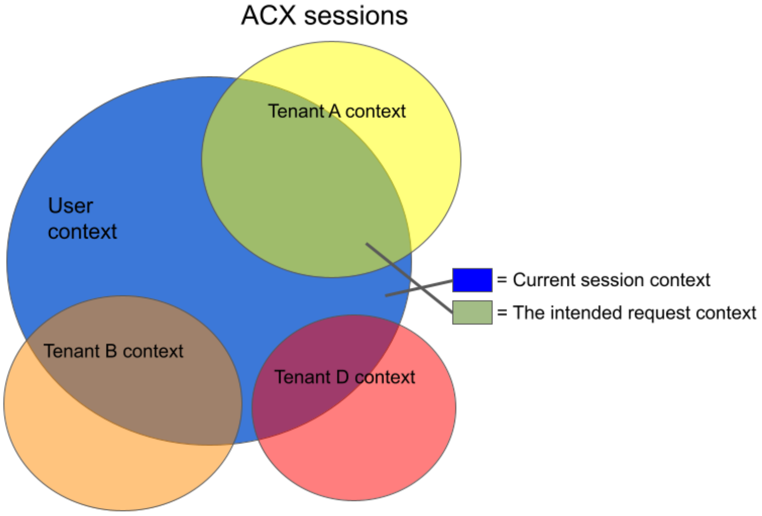


Figure 26: Contexts

Two issues with the authentication processes of ACX has room for improvement, the way ACX receives the AT from the client, and how tenant association is represented during sessions. The method for ACX to receive the AT is cumbersome and can be simplified for increased efficiency, and having a user context with tenant-contextualised requests is inferior to a tenant context with requests that inherit the context, and instead open up for possible mistakes in the handling of requests. Not including a tenant identifier in the session but instead using the tenant the user sends with each request is unnecessarily cumbersome and has provides possibilities for users attempting to breach the separation by giving them a parameter to try any input they like which is then read and helps determine the scope of the user's access on the platform. A user could try other tenant values, or some code injection which would cause serious issues as the process using the parameter has enough privileges to determine the user's access scope.

5.4 Access control

The current request authorisation, meaning the AC evaluation and enforcement, is split up with the former being centralised by having the access rights of the context be stated in the security scope, the latter being decentralised with each service reading the request and scope for the final decision of whether the request is granted. The scope is stored in the SSS in the IAM module during the session, and is called to retrieve the user's ACL for all services on ACX.

The roles specified in the IAM DB are determined by the services, thus they are not designed the same way, although all development teams are instructed in how to design them. Some roles become an operation, and some roles may contain the same operations, but for different services. This inconsistency could cause issues with e.g naming, where if two roles share a name the user would gain both roles as access, otherwise it adds redundancy with a plethora of roles that perform identical or very similar actions. The issue lies in the decentralisation of AC enforcement which results in many AC contexts which are difficult to keep track of. For smaller systems, locally enforcing AC decisions is preferred as it keeps all code in a smaller domain, yet when systems grow in size and each individual locale has a highly developed AC schema, centralisation is preferred as it offers a dedicated domain where AC resources can be pooled and a more homogeneous system with encompassing policies can be designed.

The services' interpretation of the ACL and subsequent comparison to the user's request to enforce the rules set by the ACL means each service has the responsibility for all actions within the system. Having decentralised access enforcement is motivated by each service's development team mostly independently developing its functionalities and was the more logical methodology when the services had fewer roles, operations and principals. Now, AC increases the responsibility of each team with something that should be a focus area instead of only a part of the implementation.

The decentralised AC increases the amount of roles existing in the IAM DB and ACL due to several roles with identical properties exist for many services. Although many services may have very similar roles, they are not shared, making the ACLs longer than necessary. If by coincidence the same names are used for roles in multiple services, the service needs to ensure the role is within the correct context and whether the principal has permissions for the correct one, as the consequences would be a breach in AC and result in data leakage if the current data at rest separation is used. Figure 27 shows the processes for AC in ACX due to its decentralisation.

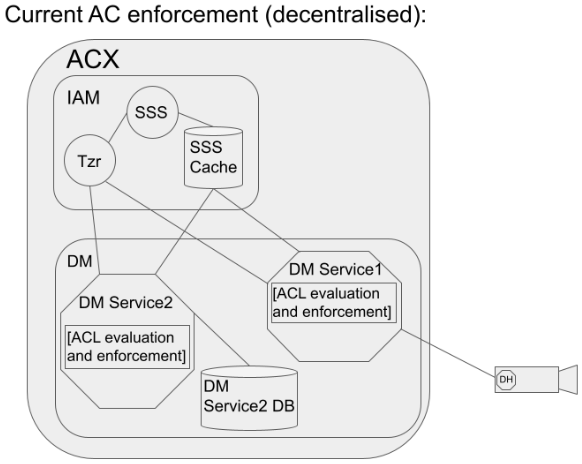

Current AC enforcement (decentralised):
ACX
IAM
SSS
Tzr
SSS Cache
DM
DM Service1
[ACL evaluation and enforcement]
DM Service2
[ACL evaluation and enforcement]
DM Service2 DB
DH

Figure 27: The decentralisation of AC enforcement on ACX

When systems have many separate services with many policies to follow, they benefit from having centralised IAM as discussed in section 3.7.1. Although it has its advantages, decentralised AC enforcement is a greater weakness than strength for the current platform. It makes the IAM DB unnecessarily large and complicated, it places almost all responsibility of correct handling of requests to the services, and means both heterogeneous policies due to the independent development of each service as well as duplicate policies for some services that they cannot share thus further increasing the amount of AC policies.

5.5 Data in use

No separation is implemented for data in use, resulting in data during processing being handled as if for a single user, subsequently and without any division of processes. The issues with this are malicious users inserting viruses to the instance and/or listening to the traffic in the instance where they then would be able to read all tenants' data and vulnerability to attacks. However since the processes are serverless, provided by a CSP and with an ELB, the issues are mainly the responsibility of AWS that Axis has chosen to trust to protect ACX from.

In the majority of papers concerning separation of processing, the two most common concern upholding tenant SLAs or critical confidentiality e.g personal information. Technologies that support greater handling of SLAs are currently not interesting for implementation on ACX since all tenants are served within their SLAs by first come first serve methodology. ACX stores no personal information other than the email-address, thus the same motivations used in the papers for data in use separation are not applicable to ACX. It does not store the email address permanently either, reducing the amount of sensitive data used.

Cases that implement greater separation of processing than logical were only found within healthcare, which have another level of confidentiality of data, and consequences of data leaks. Data leaked from healthcare systems is very problematic legally, and also has potentially severe consequences for the people affected. In those cases the increased costs of separating processes are worth the increased assurance of data confidentiality.

There is no need for ACX to increase the separation between tenants for processing, since the CSP still would utilise shared instances for serverless services even if ACX does not, thus an increase of tenancy separation means a shift from serverless, which is not an alternative for ACX.

5.6 Conclusion of the analysis

The above analysis presents multitenancy issues in three areas, data at rest, authentication and authorisation. Since separation is low any issues found become more important as the platform is without support that ensures tenant isolation that the other technologies within especially data at rest can offer. The three mentioned issues are:

1. The optional RLS of some services, not implementing the orgId-filter and resulting in users searching the entire DB instead of only the tenant's data. This is presented in section 5.2.

2. The session in ACX being user-based while the requests are handled as if sessions are tenant-based, i.e the ACL has tenant-based entries while the session allows the user to work across tenant boundaries within the same session. This is presented in 5.3.

3. The decentralised AC enforcement which means heterogeneous handling of requests and few over-arching policies which are important for large platforms to reduce AC complexity. This is presented in section 5.4

Another issue that does not involve multitenancy is in the authentication process where there are unnecessary steps to produce the AT for ACX. The issue is in the optimisation of the login flow

6 Potential system modifications

In this section the issues presented by the analysis in section 5.6 are used to find technologies that remove the vulnerabilities without introducing large new costs. The suggested technologies/changes described are either from sections 3.5 to 3.10 or are from the author and architects of ACX.

The important factors for all suggestions are how significant of a change the implementation brings, the difference in tenant separation, cost, scalability, maintenance, security aspects and complexity. The three main areas are access control, authentication and data at rest, which each have their own section below.

Since both AC and data at rest are intertwined and affect each other in most aspects, they are both taken into account for each comparison. The focus for each technology analysis are the introduced features, with the consequences to other parts of the system as secondary factors.

6.1 Access control

ACX has decentralised AC enforcement with a centralised AC model used for storing the access rights. As concluded in section 5.4, the implementation has points to improve upon.

The main point is the decentralised AC which could be centralised to follow the principles of section 3.7.1 and increase SoD and the transparency of AC decisions on ACX as all requests would be treated equally regardless of their destined service.

The AC models introduced in section 3.5 both include changes in how access is distributed and how it is evaluated and enforced. The models that address the issues from the system analysis are mainly ABAC and OTACS, with SLIM also containing useful attributes. ACL and RBAC are too broad subjects to determine whether they inherently can improve the situation, however implementing a similar policy engine as described in the ABAC model in section 3.7.5 would mean they become centralised and thus become valuable implementations as well. Therefore both ACL and RBAC are included in the section considering ABAC as an AC model. KI-ABDR-KS is very different compared to ACX and its features are not useful for improving the ACX model as there are few features they have in common.

6.1.1 ABAC

ABAC is interesting as a possible technology to adopt because of its customisability and its structure that is relatively intuitively centralised. The advantages of ABAC are that attributes allow further fine-grained policies where each resource has its conditions for access stored with it, ABAC offers great variety of application and centralises evaluation of access making the process more transparent and controllable by specialised IAM teams that can ensure homogeneity in AC enforcement. Due to its malleability it offers plethora of possible tenant-specific policies if desired, enforced equally to all services by the centralised AC. The disadvantage is the potential complexity of the solutions, since a thorough guide would have to be made for understanding which attributes relate to what access for the developers building and maintaining the platform. Figure 28 shows a possible implementation of ABAC on ACX.

If RBAC and ACL is used as currently but with AC enforcement centralised through a policy engine, this would mean all the advantages of ABAC with the possibility of a hybrid ABAC-RBAC solution through added attributes without the disadvantage of increased complexity sine it would decrease the responsibility of all services, and add one new entity that only handles AC enforcement.

Due to its centralisation and ability to adapt to ACX's current implementation, ABAC or at least the way AC is centralised with ABAC is proposed as a improvement for ACX.

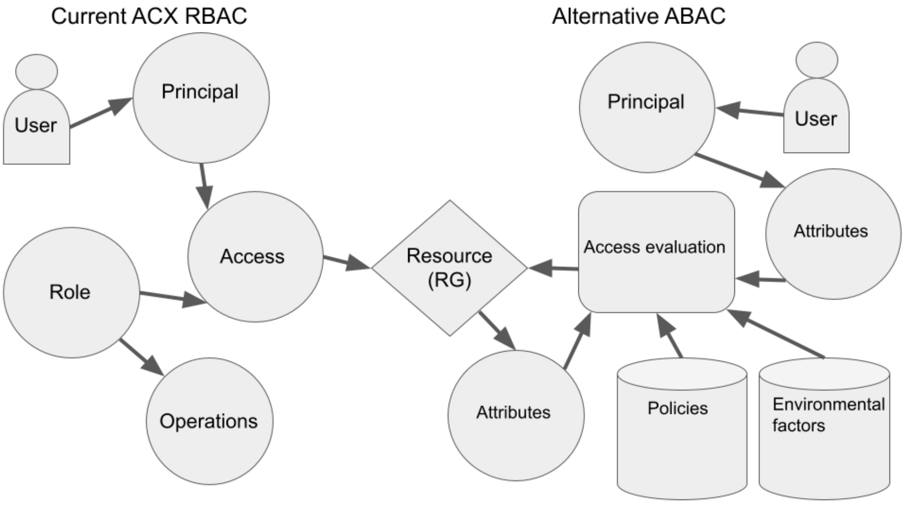


Figure 28: Alternative ABAC implementation

6.1.2 OTACS

Essentially, OTACS is a multitenant ABAC-implementation as described in 3.7.7, which uses mathematics to separate authorised requests from unauthorised. When implemented correctly it benefits of no access evaluation neither centrally or locally in each service which reduces the cost since fewer instances are needed for implementation and also increases performance due to the evaluation being a logic computation. The disadvantages are the recalculation of all subjects' tags when they gain new access to objects, the increased complexity for distributing permissions across several operations as each operation requires two correct computations with the tags, and the risk of miscalculation from incorrect code, resulting in very difficult bugs to fix.

Although OTACS has many great qualities, it has no known adapters thus a great risk for ACX potentially being the first. The safer choice of the more established technologies is more attractive while this technology could be tested for possible future implementations. It could even be used in parallel with another ABAC implementation as additional attributes.

6.1.3 SLIM

SLIM does not change the AC model nor completely replaces the AC enforcement of its adopter, instead adding isolation between services and the tenants using them through new instances. Figure 29 shows the difference between current ACX and the additions by SLIM.

The advantages of implementing SLIM would be its many systematic separators of context which ensure tenant separation enforcement in the system, the proxy and guard would keep all non tenant-owned data from the user when requests are made across several services, and the gatekeeper would do the same for the current service. The disadvantages are the increase in number of instances and thus cost due to the proxy and guard, as well as the superfluous protection between services by the gatekeepers as inter-service communication in ACX already is at minimum interaction.

The separation is higher using SLIM in conjunction with the suggestions made in this report and is the next step if further separation is desired as the increased costs are relatively low, and its implementation can be performed in stages making compromises between cost and separation simple.

6.2 Authentication

There are no specific technologies found to replace or change the current process, but using best practices from section 3.7.1, the implementations from the technologies in sections 3.8 and 3.7

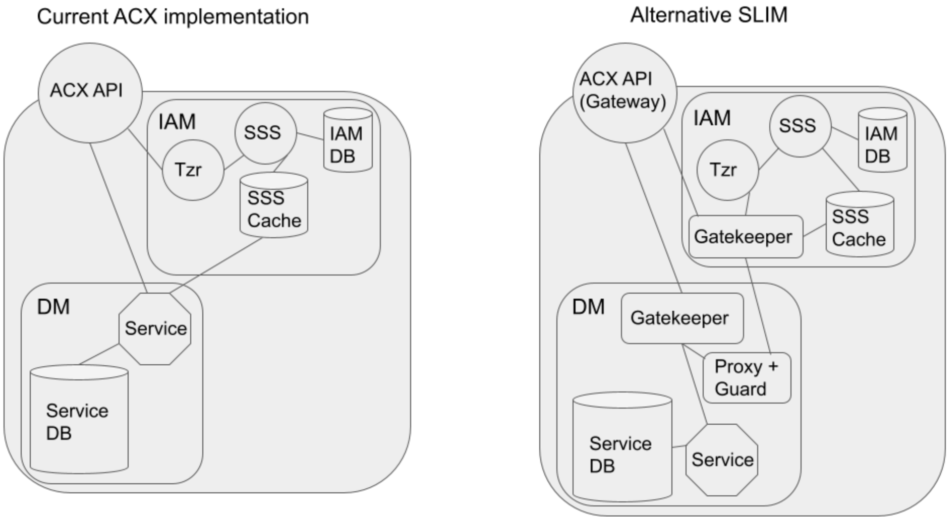


Figure 29: Alternative SLIM implementation

as well as the input from the architects of ACX, changes are proposed to remove the issues from section 5.3. The two sections cover one issue each.

6.2.1 Session context

One of the issues from the analysis of ACX authentication is the way sessions are contextualised on ACX. By adding a tenant identifier e.g the orgId to the JWT, the session and the requests have the same scope thus interactions with ACX become more uniform where the request no longer needs to specify the organisation it is acting within. It can be implemented by simply adding the orgId to the JWT which the Tzr has the homomorphic key to, thus allowing the services to view the requester's tenant at all times via a call to the Tzr. It could also be added as plaintext to the JWT, enabling the services to read the tenant of the user from the context themselves. For ACX, the suggestion is to store the orgId encrypted in the JWT, meaning the service has to request the orgId from the Tzr. Since the service always communicates with the Tzr for session validation, this only adds another parameter sent between them.

The tenants are separated at approximately equivalent level with this suggestion as they currently are, yet it creates a clearer demarcation by moving the context from being based on each request, to being based on the session. It also makes RLS more rigid as the orgId can be fetched from the session instead of the request, making it a more constant value, ensuring a degree of separation of tenants between requests.

In the paper that described logical separation with caches introduced in section 3.8.1, the tenantId is used as a session context and is also used for AC. SLIM uses sessions tied to both the user and their organisation as well to separate tenants as described in section 3.7.6. Thus this change can also be motivated by other implementations for increased separation.

6.2.2 Simplified session initiation

A smaller issue and one that can be remedied intuitively is the way ACX receives the user's AT. Changing the initiation by having the LS send the user's AT to ACX immediately, the negotiation for a session with ACX starts when the user logs in to Axis and ACX can subsequently send the list of organisations the user has access within, instead of the more tedious route of first sending the SC to the LS, receive the AT back and then send the AT to ACX, for ACX to then send the organisation list.

One step from the process is removed and has mainly the result of reducing the complexity of the session initiation although this also could be faster. There are no negative consequences of this change, making it a simple proposal.

Figure 30 shows the difference in session initiation on ACX between the current solution and the proposed solution. It includes the changes both for simplified session initiation and the tenant contextualisation of sessions proposed in the section above. The JWT is the session identifier and contains both the AT and the orgId for the tenant. Requests only contain the request for an action and the session JWT, instead of the current action, organisation and JWT.

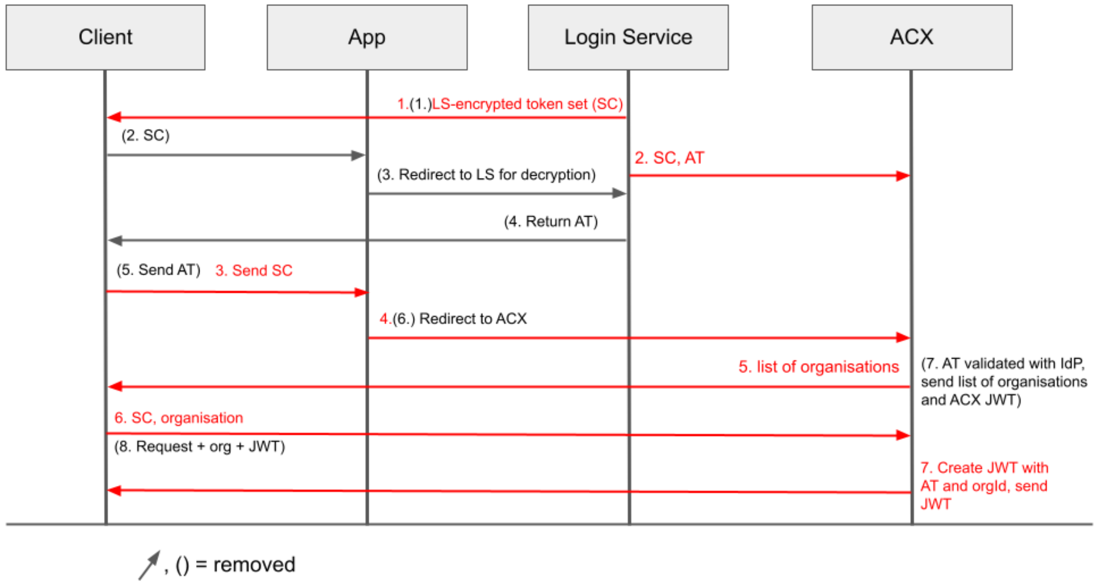


Figure 30: Proposal for ACX session initiation

6.3 Data at Rest

ACX uses logical separation RLS with both relational and document storage for data storage. The issue with this implementation presented in section 5.2 is the incomplete RLS implementation, resulting in AC being the only tenant separator for some services.

The technologies addressing at rest separation as introduced in section 3.8 are logical separation, separate schema, separate encryption, and separate database. Separate encryption and separate database have greater separation than is needed for the data stored by ACX, and since the tenants all utilise the same structure for both their data and AC, e.g separate database only results in costs that outweigh the increased separation, especially since RLS and separate schema are the technologies most implemented by other systems with a high degree of shared features, as ACX has. The remaining technologies of interest are the two logical separation technologies, and the two separate schema technologies.

6.3.1 Logical separation with API-enforced RLS

The first case in section 3.8.1 presents the technology with most similarities to ACX's current separation, using document storage RLS both in the DB and API. The requests for data by the service and user are appended to a query template the API uses. The template requires tenant-affiliation to be present before queries are performed.

Other than the use of SimpleDB, the only difference in implementation compared to ACX is advantageous as it standardises data retrieval on the platform. The template could be used for several services that share similar storage structure, and an additional safeguard between tenants helps AC in ensuring only users of the tenant may read its data. The difference between ACX and this alternative is shown in figure 31. All features that are currently used can be kept and few additional costs are introduced for a greater separation of tenants.

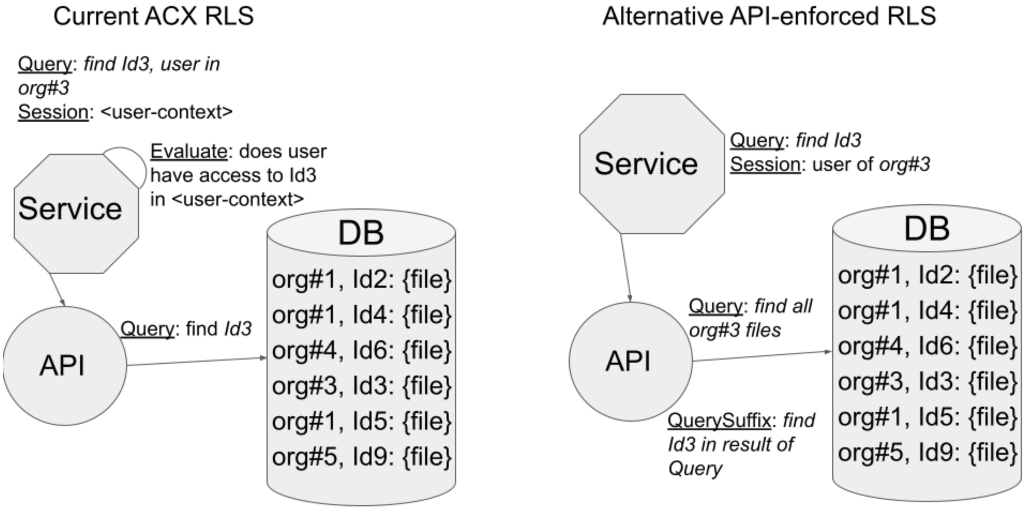


Figure 31: Alternative RLS implementation

Due to its similarity to ACX and increase in separation through a simple addition to the data query, this technology is proposed for implementation onto the platform.

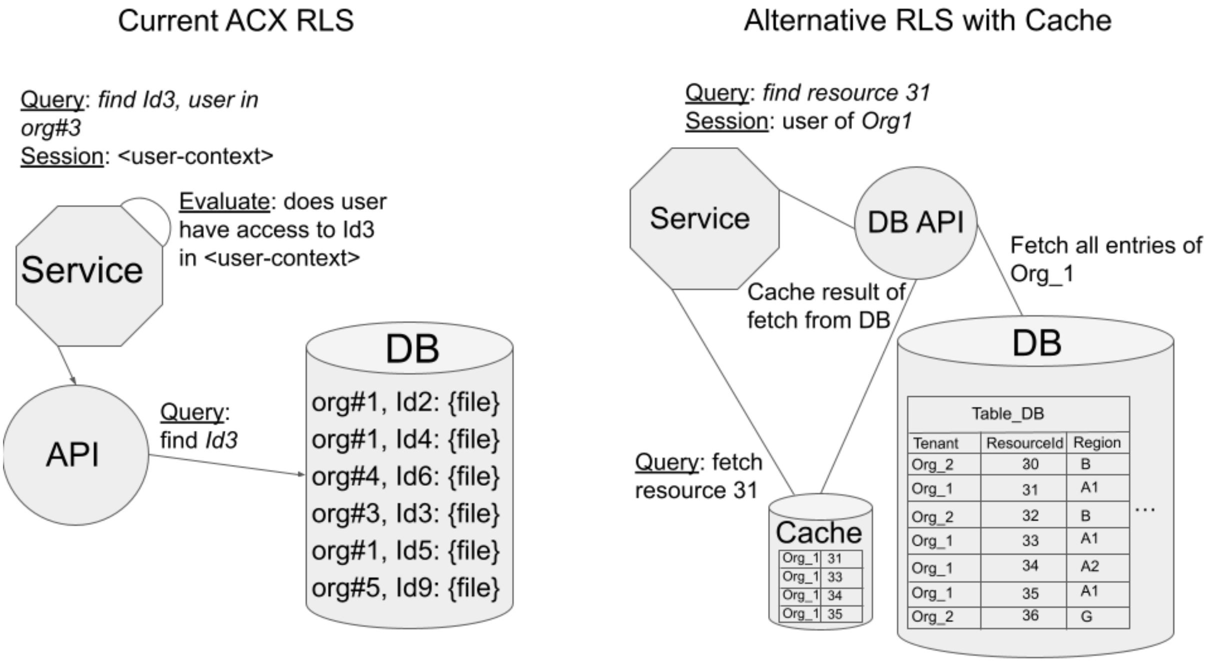
6.3.2 Logical separation with cache storage

To first filter the tenant-owned data before the user queries increases performance since the majority of data already has been removed from the pool when the user queries it, and storing it in a cache means the already populated cache can be used for longer sessions without querying the entire DB again. RLS caches could be a useful update for larger databases however the system would cost more which is dissuaded by the architects. The simplicity of RLS is once again mentioned and preferred over the more structured separation alternatives, especially for tenants with similar data stored. The differences of the systems are shown in figure 32.

By not having the service ever reach the DB, the user only ever comes in contact with their tenant's data and thereby effectively making the system seem like a separate schema implementation. The advantages are the increased separation of reached tenant data, enforced single tenant context and fast performance when the cache is queried, while disadvantages are the increased costs of running the caches, and the mapping of the correct cache to the right session in the service when data is queried by the user. For ACX currently, this solution is not suggested but could be interesting if greater separation than the one suggested in section 6.3.1 is wished further on.

6.3.3 Separate schema storage with sharding

Separate schema is especially useful for systems where tenants store data differently, as this is the use case of the two papers suggesting separate schema in section 3.8.2. The advantages of separate schema is that the schema has to be included in the query to get any data, a structurally enforced policy, while the tenant is mandatory due to the code for RLS being a policy of the same effect, but can be removed by a developer unlike separate schema. Both systems have an increased risk for to developers mistakenly incorrectly implementing the technology, yet this is



Table_DB

Tenant	ResourceId	Region
Org_2	30	B
Org_1	31	A1
Org_2	32	B
Org_1	33	A1
Org_1	34	A2
Org_1	35	A1
Org_2	36	G

Figure 32: Alternative RLS with Cache implementation

critical in RLS while the solution probably does not work during tests if incorrect in separate schema. The differences between the systems are shown in figure 33.

The disadvantages are that all tenants' data are structured identically on ACX, thus receiving no advantage of using separate schema in that aspect. If tenants on ACX wished to store their data in another way within their own schemas, they would not be able to use some or all of the services ACX provides which only handled data structured in the current manner. Implementing separate schema with identical storage structure within the tables/collections would work well but would gain no functional advantage over logical separation. Separate schema scalability is worse than the current solution, requiring more work as new tables need to be generated and all APIs need to be updated with this new schema in the DB connection. Due to its reduced

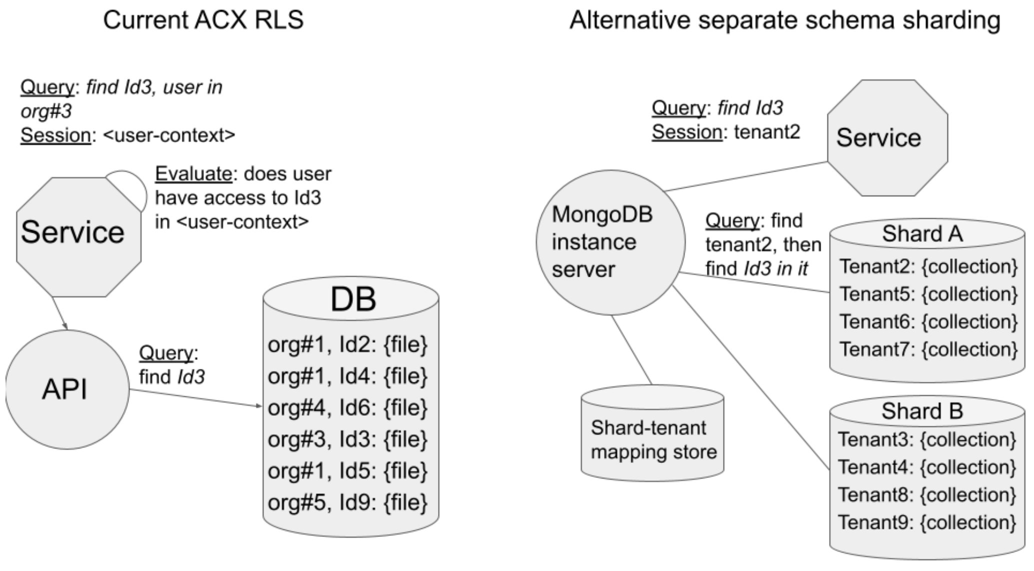


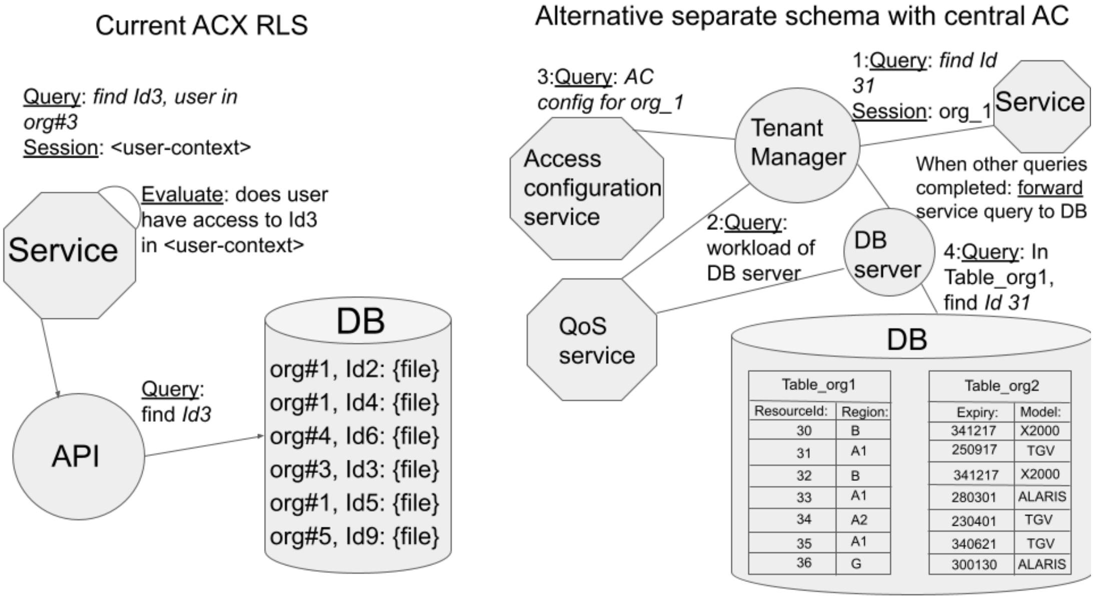
Figure 33: Alternative implementation with separate schema and sharding

scalability and the architect's trust in the developer's ability in separating tenants logically, separate schema is not proposed for ACX. Separate schema mainly increases administration for improved structural support that can be implemented with the same results by RLS with no increase in administration. It is an improvement compared to the current solution, however the solution of section 6.3.1 is more resource-efficient.

6.3.4 Centralised access management with separate schema

Similar to the analysis above, separate schema is not optimal for ACX to implement since the same policies and structures are used in data and AC for all tenants, the scalability is lower and the developers of ACX are trusted to implement tenant separation correctly, suggesting logical separation is more viable.

The additions of this technology are the centralised AC in the TM and the QoS service. The QoS service is an interesting addition however as ACX are fulfilling their SLAs already it adds no value to the current platform. Centralised AC is recommended but not as implemented here, being customised for each tenant. Since all tenants use the same functionalities in the platform a homogeneous AC model is a more efficient choice. If the tenants ask for a higher degree of adaptability it may become a contender in the future. Figure 34 shows how this technology could be implemented onto ACX. As seen, the data retrieval schema increases in complexity with the additions from the technology.


Current ACX RLS
Query: find Id3, user in org#3
Session: <user-context>
Service
Evaluate: does user have access to Id3 in <user-context>
API
Query: find Id3
DB
org#1, Id2: {file}
org#1, Id4: {file}
org#4, Id6: {file}
org#3, Id3: {file}
org#1, Id5: {file}
org#5, Id9: {file}
Alternative separate schema with central AC
3:Query: AC config for org_1
1:Query: find Id 31
Session: org_1
Tenant Manager
Service
Access configuration service
When other queries completed: forward service query to DB
2:Query: workload of DB server
DB server
4:Query: In Table_org1, find Id 31
QoS service
DB

Table_org1	
ResourceId:	Region:
30	B
31	A1
32	B
33	A1
34	A2
35	A1
36	G

Table_org2	
Expiry:	Model:
341217	X2000
250917	TGV
341217	X2000
280301	ALARIS
230401	TGV
340621	TGV
300130	ALARIS

Figure 34: Alternative implementation with separate schema and a centralised AC

6.4 Implementing other CSP services

Below, the AWS services ACX implements are described and compared with similar AWS services to evaluate whether other services would be useful for the platform. Only computational and database services are viewed to reduce the scope of this analysis. The analysis is performed to find whether there are other, better suited services for ACX than the currently implemented ones.

The services used in ACX that are of interest to exchange are mainly the ones affecting data at rest and access management, where AWS DynamoDB, PostgresDB for RDS, NeptuneDB, and S3 are implemented, with Lambda being the main service for data in use. DynamoDB is the service used for most data storage which uses document storage, and otherwise PostgresDB for relational data storage. NeptuneDB is used for the relational graph storage of the IAM DB and S3 is used for the SSS cache. Lambda is used for most processing in ACX, and the instances act as APIs for all services, performing computations, queries and more.

The alternative AWS services are few due to serverless being a required feature for ACX adoption. For storage there is SimpleDB, Aurora, Elasticache, Keyspaces and MemoryDB where SimpleDB is an older version of DynamoDB and Elasticache and MemoryDB are caches, for computation there is only Fargate that offers serverless processing.

DynamoDB is the one of the most commonly used document stores for AWS, and since the data stored in ACX fits well for this service's purpose it is most likely the best service for it within AWS. SimpleDB offers no additional features, instead offering fewer, and with no benefits, which is why it is not considered. Keyspaces is very similar to DynamoDB in features, except it only can use Apache Cassandra storage, meaning DynamoDB remains the best document storage for ACX although tenancy separation possibilities are the same.

Aurora is similar to PostgresDB for RDS and there is no difference that affects tenancy separation, except Aurora being more adaptable to workloads. Both Aurora and RDS can change the type of relational data storage, and both scale automatically. According to AWS themselves, the SaaS/PaaS with low to medium workloads should use RDS, and high workloads should use Aurora [45]. To better handle the increased workflows of the future, Aurora should be used instead. Other services are not as useful where Neptune is used, since they cannot store the same high level of complexity in relationships which is why it should be kept.

The two cache-services Elasticache and MemoryDB are interesting for the security scope, which currently is stored in an S3 bucket. S3 is an object storage service, and therefore not meant for the temporarily stored JSON documents the security scopes/ACLs are. Both Elasticache and MemoryDB are viable key-value caches which implement Redis, a commonly used open source caching store. MemoryDB is for storing data for longer sessions while Elasticache is meant to be used for shorter time periods, which is why Elasticache is recommended to replace the S3 SSS cache. Elasticache stores data for shorter time periods and since the data only is stored for as long as the AT is valid, 5-15 minutes, longer caching is superfluous. Redis is a typical caching solution used for fast responses for smaller stores.

Lambda is an efficient service which performs small processes, distributed throughout the system. In AWS there is Fargate as an alternative, yet it provides unnecessary configuration possibilities not utilised by the processes ACX performs on Lambdas.

The AWS services used are mostly implemented for the use cases they were designed, except the S3 that stores the data of the security scope service. There AWS Elasticache for Redis is better as it can increase performance and provide more tailored functionalities to the use case.

6.5 Third party AC

As introduced in section 3.5.2, many systems choose to use open source libraries for AC, e.g OPA, Casbin and Ory. Their purpose is providing simple AC and they are trusted by many large corporations making them a suitable candidate for ACX as well. OPA and Casbin are open source, offer diverse solutions that can be tailored for most systems and both Casbin and OPA are solutions that can be utilised through a central policy engine that e.g ABAC implements. Due to their widespread adoption, and libraries/frameworks supporting all cases ACX could need, either Casbin or OPA are suggested to be adopted for the AC of ACX.

OPA is the chosen recommendation to implement the policy engine with as suggested in section 6.1.1 although Casbin also is a valid choice.

6.6 Tables of comparisons

The alternative multitenant technologies within AC and data at rest above introduce varying methods of separating tenants with different methods, and with diverse results. There are two types of tables below, one with a digit signifying the relative efficiency the technology has within that property compared to the other technologies in the table. The second table lists the general advantages and disadvantages of each technology if ACX were to implement it.

Since all technologies that mentioned sessions had them in a tenant-context, there is no comparison made for the suggestions within session management above in section 6.2. This is because encrypted/unencrypted JWTs or a sessionId as a UUID mapped to data stored safely on the platform all have their uses and would each be effective on ACX, although the chosen recommendation is to use an encrypted JWT.

6.6.1 AC comparisons

In table 1 the impact to ACX if alternative AC technologies are implemented is presented.

Table 2 is a comparison of the AC technologies presented. The AC type specifies how AC performed in a system with the technology, i.e whether the technology is made for local AC for each service or central for a dedicated module for the entire system. The values of the other categories are of a ranking scale, where 1 is the lowest and 5 highest of the category specified. They are the result of comparisons between the technologies where they are placed in descending order for each category separately. In most categories, some technologies have the same value, this is because the are equally effective in that aspect. Each category analysed the following properties for each technology:

- **Complexity** How difficult it is to implement, how many instances must be created for it to fulfill its purpose, what algorithms are used and how many steps they contain as well as how many steps that can be mistakenly incorrectly implemented and their possible consequences.
- **Malleability** How much it can be adapted to the demands of the user/platform, how many different AC models that can be used with it(e.g RBAC, ACL, ABAC) and how limited they are in implementation.
- **Compaction** How much space the model occupies in terms of lines of code and bytes of storage, how advanced the AC evaluation is and how large the payload of information that is sent to the different instances on the platform is.
- **Separation** How many structural and logical separators tenants are from each other and what safeguards there are to hinder inter-tenant leakage of information.

Whether the ranking is better than another is determined by if the category is useful or not. A 1 does not mean it is the lowest possible in that category, rather that it is the lowest of the technologies presented. The same is true for a 5 in a category, meaning the technology has the highest rank of that category when considering all technologies in the table.

In some fields the rank is a span of several digits, and this is an effect of the malleability category where the technology is shown to be very adaptable to the desired functionality, i.e ABAC and RBAC.

6.6.2 Data at rest comparisons

In table 3 a comparison of the current implementation's and the alternatives' features is made.

In table 4 the cases presented above are compared superficially. The numbers have the same functionality as above in table 2. Each category analysed the following properties for each technology:

- **Complexity** How difficult it is to implement, how many instances it uses, what algorithms are used and how many steps they contain as well as how robust it is to mistakes in

AC technology	Advantages	Disadvantages
ACX, RBAC+ACL (4.2.4)	Each service has a tailored AC, local policies could be used, roles are very specific	Complex IAM structure, many roles therefore not compact, possible naming issues for roles, no tenant-boundaries except for RGs
RBAC (3.7.3)	Clear role definitions regarding both operations and access to RGs	Many objects/resources need to belong to several RGs if roles with different operations are to access the resource, less intuitive administration of access for principals and potential data redundancy
ACL (3.7.4)	Simpler IAM DB, all information for AC is compact and results in more intuitive AC enforcement	Tedious administration, all operations would need to be service-specific
ABAC (3.7.5)	Greater adaptability, more functionalities, centralised AC which makes AC enforcement more transparent and uniform	Possibly complex and unintuitive results from the policy engine makes it difficult to review, easy to misunderstand attributes due to their abstract nature
SLIM (3.7.6)	Systematic separators help developers retain AC context both between services and within	Greater amount of instances increase costs, ACX seldom needs inter-service requests(except core services where a partial implementation could be useful)
OTACS (3.7.7)	Quick access evaluation, fewer instances therefore cheaper, more compact AC codebase, less complex IAM DB	Few levels of access for RGs, difficult bugs when present, only user- and object attributes, more tedious administration
KI-ABDR-KS (3.7.8)	Increased separation between users, AC depends on decryption and not evaluation, centralised AC with policy engine provides flexible AC if attributes for operations are added.	Increased complexity, more computation, more storage, more sensitive data(enc., dec. keys)

Table 1: AC alternative technology comparisons

Technologies	AC type	Complexity (1-5)	Malleability (1-5)	Compaction (1-5)	Separation (1-5)
ACX (RBAC+ACL)	Central+local	2	3	2	1
RBAC	Central/local	2	4	2-4	1-3
ACL	Local	1	1	1	1
ABAC	Central	1-4	5	3-5	1-4
SLIM	Local	2	3	-	4
OTACS	Local	4	2	3	4
KI-ABDR-KS	Central/local	5	2	1	5

Table 2: Comparison of the presented AC technologies

implementation, meaning how easy it is to implement incorrectly and what its consequences would be.

- **Shared resources** How many instances that are needed for each tenant, how many are necessary in total and how much of the codebase that can be reused for the tenants.

- **Structural protections** What support it provides for separation other than the logical level, how difficult the protections are to circumvent and how easily/quickly they can be made effective again.

- **Scalability** How much work is required when tenants are added or removed for them to gain full functionality, and how great the effect on performance could be if the amount of tenants increased.

As mentioned for the table of AC comparisons, the digits are the rank of the technology in that category, relative to the other technologies ion the comparison. 5 is the highest rank and 1 the lowest in all categories. In most categories, some technologies have the same value, this is because the are equally effective in that aspect.

At rest separation	Advantages	Disadvantages
ACX implementation	Scalable, can operate within several tenants simultaneously	No structural protections between tenants (only enforced by AC)
Logical API-enforced RLS (first in 3.8.1)	More clearly defined border between tenants, greater assurance of users only reaching resources within their tenant	Since this implementation only forces ACX to use the almost implemented RLS, it has no disadvantages to ACX currently
Logical cache storage (second in 3.8.1)	Services never query the shared DB, increased tenant separation with no lost performance	More instances means higher costs, more logic required to map sessions to the correct cache and ensuring the cache is flushed regularly
Separate schema sharding (first in 3.8.2)	Increased separation in storage, mandatory tenant-specification for queries	The mapping between schema and tenant needs to be stored somewhere, increasing costs and complexity of data querying
Separate schema with TM (second in 3.8.2)	Centralised AC means increased transparency for the platform as a whole, mandatory tenant-specification for queries	The additional features of the separate schema are for less homogeneous platforms, the QoS service reduces overheads
Schema and DB separation (3.8.4)	Increased separation, mandatory tenant-specification for queries, mistakes in implementation are easily found	Increased costs, much unused capacity, mapping between tenant and instance and tenant and schema must be stored somewhere thus increasing costs further

Table 3: Data at rest alternative technology comparisons

Separation	Complexity (1-5)	Shared resources (1-5)	Structural protections (1-5)	Scalability (1-5)
ACX implementation	1	5	1	5
API-enforced RLS	1	5	2	5
Separation with caches	2	4-5	3	4
Sharding DBs	2	3	3-4	3
Schemas and TM	3	3	4-5	2
DB and schema	2	1-2	5	1

Table 4: Comparisons of the data at rest technologies presented

6.7 Conclusion of potential modifications

6.7.1 Access control

As mentioned in table 1, the multitenant AC implementations of ABAC, SLIM, OTACS and KI-ABDR-KS have more structural safeguards that can be used for tenant separation than ACX currently has, e.g the environmental factor/attribute of tenant-context in ABAC, the Gatekeeper, Guard and Proxy for SLIM and the tenant-chosen keywords in KI-ABDR-KS used for decryption of the data. KI-ABDR-KS and OTACS do however not provide all features ACX needs within AC to implement the diversity of operations on ACX while SLIM adds too many safeguards to be worth the increased costs. ACL is partially used already and works well for storing the access of the user within the current session, however only using ACLs would have no advantages to the current solution on ACX and would increase the work for all tenants distributing their access. SLIM is already partially implemented, and the proxy and guard are only useful for requests that require communication with several services which is only the case for those concerning the main services i.e the Tzr and SSS and they already have the parameters of the session with the JWT.

The additions from ABAC would make issues like the one in section 5.1 much less likely, and would work to support the developers ensuring mistakes affecting several tenants are more easily avoided. The increased protection offered by centralised AC enforcement is useful for ACX and covers the concluded AC issue from section 5.6 while also adding new potential ways to improve AC through global policies and attributes which is why it is proposed. ACX should keep its RBAC, making the resulting AC model a mix between RBAC and ABAC where the same RBAC model can be utilised, however with addition the centralisation in ABAC. Figures 28 and 35 show the changes in AC enforcement.

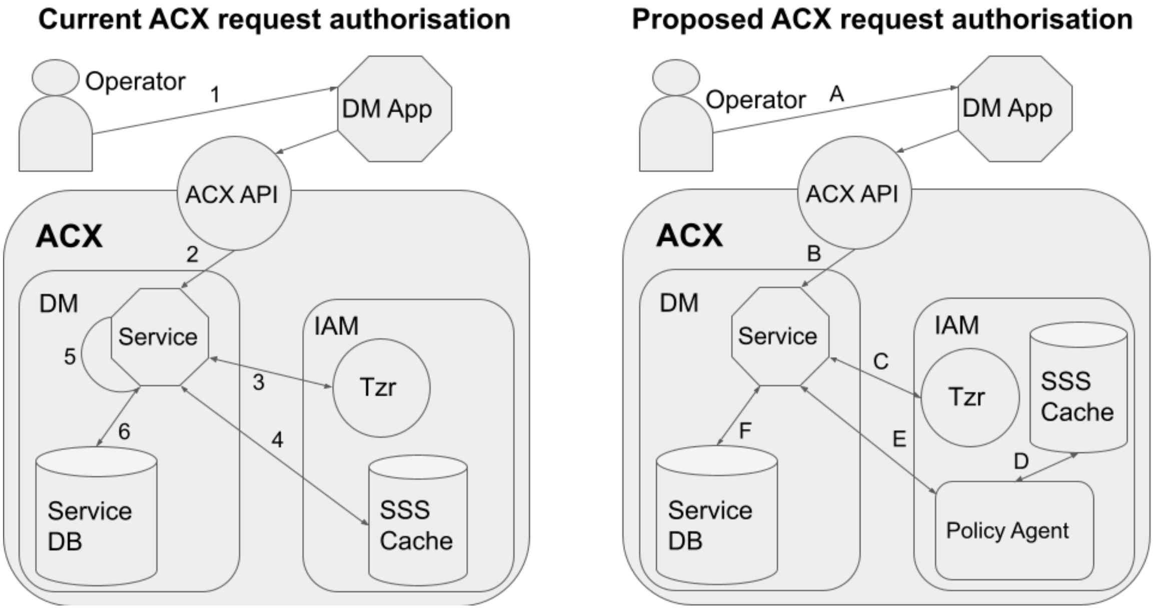


Figure 35: Comparison between current authorisation flow for a request, and the proposed flow

6.7.2 Authentication

The two modifications suggested in section 6.2 remove redundant processes and in the case of the change in session, also clarifies the user's rights and constrains their context to one tenant at a time. Currently, users can only act within one tenant at a time and through this change the context will also follow that policy.

The two changes that are proposed are to add the tenant of the user to the JWT that represents the session, and to send the AT directly from the LS to ACX instead of via the LS-client-LS-client-ACX route. Figure 30 shows the suggested authentication process.

6.7.3 Data at rest

The technologies with greater separation than logical, i.e separate schema, -encryption and database, have motivations concerning confidentiality or customisability for tenants. Since ACX has minimal personal data the higher costs of separate encryption and -database are not worth the increased separation. Separate schema can be implemented for logical as well via API-enforced RLS, thus ensuring only the current tenant's data is reachable to the user. The TM from the second separate schema case is useful for tenants with different AC structure, which they do not in ACX. Separate encryption e.g KI-ABDR-KS offers much separation however demands higher costs both for more storage, and more computations. Separate DB is not proposed for the same reasons as separate encryption.

Logical separation remains, where both the technology using caches the the one not using it work well with ACX, with the advantage to the first case with API-enforced RLS, without the caches. RLS is supported by several sources, it is resource efficient, and has the best scaling of all technologies analysed, it is chosen because of this and because it is a foundation with which it is possible to build further upon tenancy separation within AC. The caches add assurance that only the tenant's data is available for the query, however the API of the first case does the same with fewer instances. The caches can optimise query response times however this is not prioritised in this thesis, and thus it increases costs with slight improvement to separation.

The proposal means that the API-enforced RLS becomes mandatory to utilise for all services compared to the current implementation which only has most services with mandatory RLS. A change that results in a universal policy for ACX and a clear distinction between tenants throughout the platform. Figure 31 shows the change.

6.7.4 Other modifications

In section 6.4, a change in the service used for the SSS cache is proposed, from S3 object storage to Elasticache Redis key-value storage.

In section 6.5 the open source AC services OPA and Casbin are discussed, which both work well with a centralised AC and a combination of RBAC and ABAC policies, which is why either are recommended to implement AC with.

7 Results and proposed system design

In section 5, three major issues and one smaller are listed as areas with potential for improvement, in section 6 proposals are presented which remove some or all the issues with the areas. Here the system changes proposed in sections 6 are used to present a system design with a high-level proposal for implementation as well.

7.1 Proposed system design

Here the system changes from section 6 are presented. The areas not mentioned remain as currently implemented.

7.1.1 Authentication

The user signs in to two services when initiating a session, the first one with the login service(LS) using OIDC and the second one with ACX using the resulting tokens from the LS login, it is for the second flow changes are made. The issues found in the authentication of ACX are described in section 5.3 and the proposed changes are introduced in section 6.2.

The session context is changed as proposed in section 6.2.1 from being only user dependant to be user- and tenant-dependant. The LS is changed to send the SC to the client and the AT to ACX instead of via the client as proposed in section 6.2.1, meaning a step in the login flow is removed thus simplifying it with no negative consequences. The proposals are seen in figure 36 which is the same as figure 30.

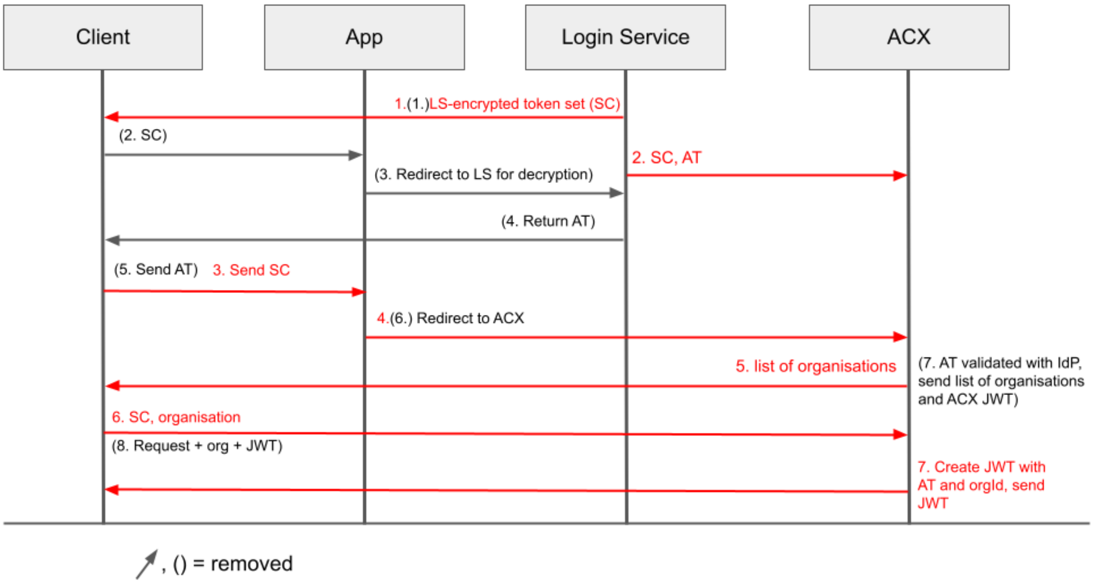


Figure 36: Proposal for ACX session initiation

As seen in the figure, the client still receives its SC after the first authentication against the IdP while it no longer receives the AT as it instead is sent to ACX which then sends the list of organisations the user has access within to initiate the tenant-contextualised session by including it in the JWT used in ACX. The JWT now contains both the user's access scope, and with which tenant the session is. This change also means that the data queries fetch the orgId from the JWT as presented in section 7.1.3.

7.1.2 AC enforcement

The current AC enforcement is decentralised, using services with distributed ACLs from the central SSS cache. In section 5.4 issues with decentralised AC is discussed and in section 6.1 a proposal is made to centralise enforcement by taking inspiration from the ABAC model. Both changes from section 6.7.4 are also affecting AC and are therefore included in this section.

The proposal is to change the AC enforcement to a central policy engine/agent(PA) and change how the security scope is stored in the SSS cache. The PA will be in the IAM module, and is connected to the SSS cache and all services. It contains the policies of each service and uses the policies to contextualise the requests as if the evaluation was performed in that service.

The PA receives a request from a service containing the JWT, the request and the service it is sent from. The PA queries the SSS cache for the security scope corresponding to the JWT. The SSS either generates the security scope if this is the session's first request or immediately responds with it. The security scope/ACL contains all access rights of that user within the tenant specified in the JWT. The PA evaluates the JWT, the service's policies and the request. It then sends the resulting authorisation decision to the service which receives the OK/Not OK from the PA and acts accordingly. Figure 37 shows the difference in request AC enforcement. The letters indicate the proposal, the numbers the current design.

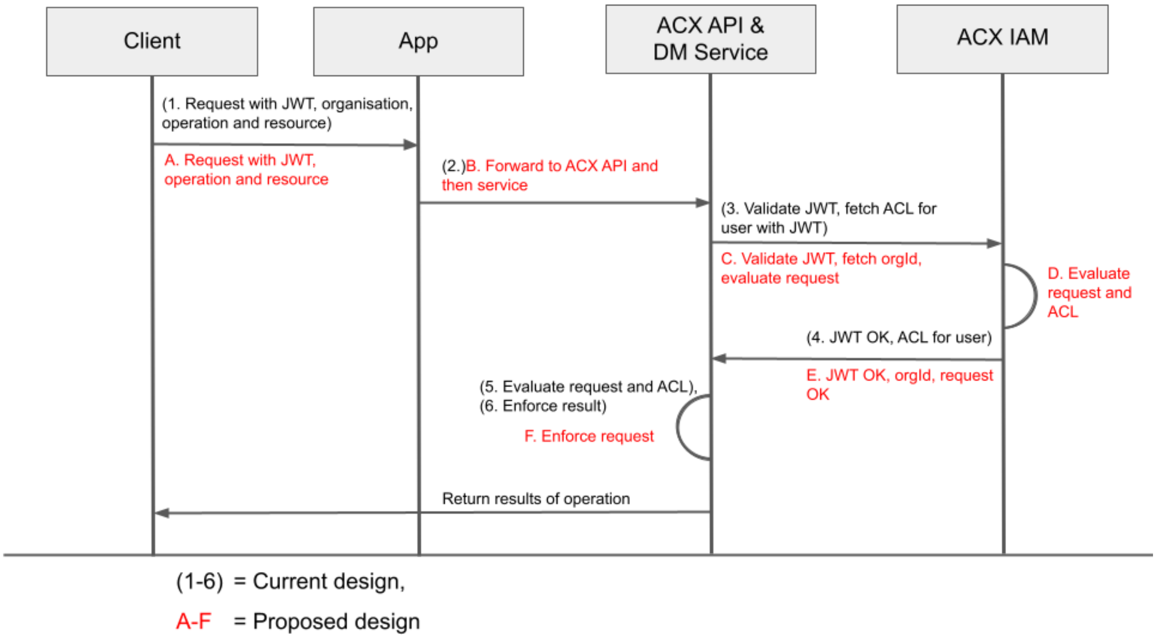

Client
App
ACX API & DM Service
ACX IAM
(1. Request with JWT, organisation, operation and resource)
A. Request with JWT, operation and resource
(2.)B. Forward to ACX API and then service
(3. Validate JWT, fetch ACL for user with JWT)
C. Validate JWT, fetch orgId, evaluate request
D. Evaluate request and ACL
(4. JWT OK, ACL for user)
E. JWT OK, orgId, request OK
(5. Evaluate request and ACL), (6. Enforce result)
F. Enforce request
Return results of operation
(1-6) = Current design,
A-F = Proposed design

Figure 37: Comparison between current authorisation flow for a request, and the proposed flow

In figure 38, the different entities involved are visualised and uses the same numbering and lettering as figure 37. As seen, the responsibility of AC lies completely with the IAM module after this change, which results in less responsibility for the services, increasing the SoD. Centralising AC opens up to more possible configurations using the ABAC policies and environmental factors that can further improve AC enforcement in the future while it initially mainly increases the transparency of the process of AC on ACX.

The policy agent is implemented using open policy agent(OPA). This is suggested in section 6.5 as a way to introduce global policies for ACX. It will be used as a REST API for the services, receiving all parameters of the request to then read the security scope and make an executive decision, relating it back to the service. OPA can be installed on several AWS Lambdas which evaluate different policies, and store the policies in a dynamoDB instance, with the applicable services or broader applicable areas as keys.

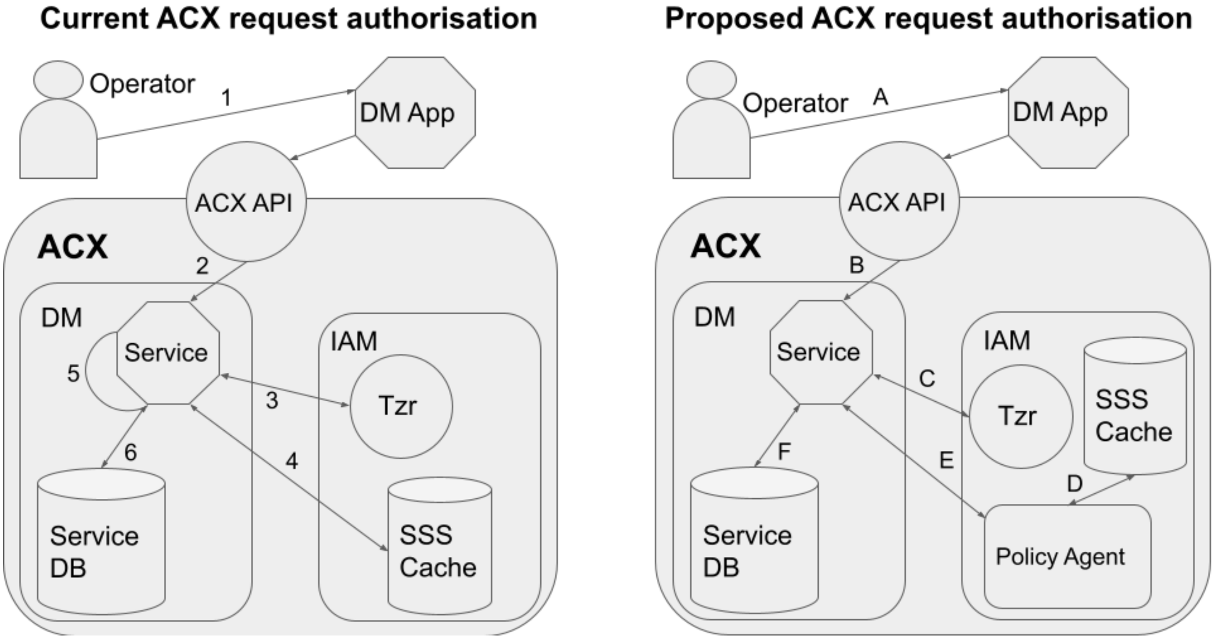


Figure 38: Comparison between the current ACX design and the proposed design of AC

The SSS cache is currently a S3 bucket instance, which is not optimised for caching purposes. This is why the change from S3 to Elasticache Redis is proposed, as analysed in section 6.4.

7.1.3 Data querying

The ACX implementation of logical separation analysed in section 5.2 had issues with how data is queried. In section 6.3 a similar solution to the current one is proposed for ACX, the only change being the forced instead of optional tenant context for DB queries.

The proposal is to enforce the tenant context by using the orgId in all queries as in the first case presented for logical separation in section 3.8.1, analysed in section 6.3.1. The APIs will have a new template for requests to data storage, containing a mandatory field for the orgId. The orgId value is received from the JWT and Tzr as proposed in section 7.1.1, while the other parameters of the request as before are from the request made by the client. The queries are not to be performed until the PA has granted the request, which the service is notified of by the OK response from it. Figure 39 shows the current solution and the proposed change.

7.2 Proposed implementation method

Below a summary of changes, and the affected entities are presented, to indicate the scope of the changes proposed. There must be a strategy for applying the updates in a way that is both time-efficient and easy to follow since all suggestions result in other parts of the platform having to adapt as well.

7.2.1 Changes to session management

The changes to the session are

- Do not send the AT to the client.
- LS does not decrypt SC, instead only provides the client with the SC.
- LS sends SC, AT, sessionId to Tzr instead of only AT.

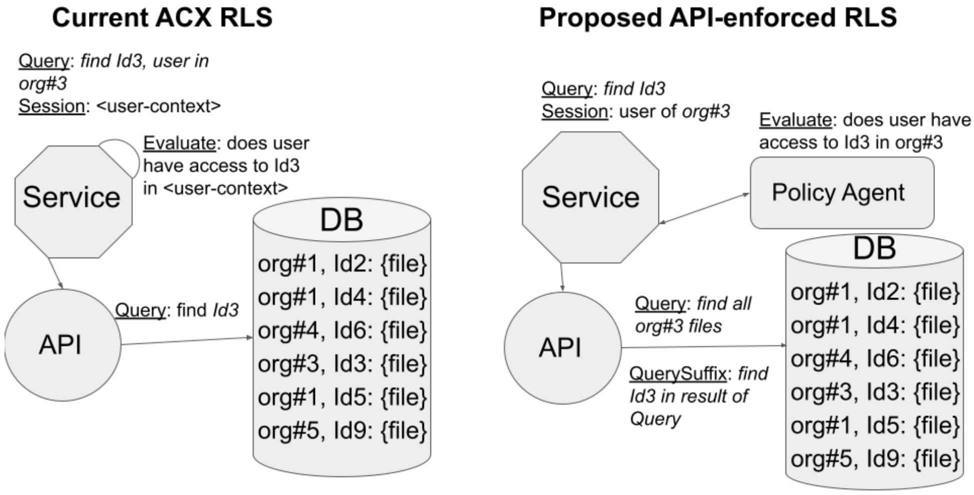


Figure 39: Current RLS in ACX compared to proposed RLS

- The JWT is tied to one tenant and user instead of only one user.
- Session scope is constant, not dependant on each request.

The affected entities are

- The Axis apps connected to ACX which need to update the session initiation protocol.
- The LS with the changes in above list.
- The Tzr which needs to handle the updated JWT to be able to serve requests for orgId by the services
- All ACX services, since the way they receive the orgId is changed

7.2.2 Changes to AC enforcement

The changes to AC enforcement are

- All AC enforcement code is moved to the PA
- The services need to include API-requests to the PA
- The ACL can at most contain the privileges of a user within one tenant
- The SSS cache is changed to Elasticache Redis instance from an S3 instance
- The current policies are rewritten to function as OPA policies

The affected entities are

- The Axis apps connected to ACX which need to remove the organisation field for each request
- All ACX services, since their AC enforcement is moved to the PA
- The SSS cache, since it changes instance type from S3 to Elasticache Redis, and it is now queried by the PA instead of the services

7.2.3 Changes to enforcement of RLS

The changes to data querying are

- Change API requests to require orgId instead of it being optional.
- All DB queries contain the orgId as one of the fields/keys.
- The orgId is received from the Tzr instead of the client's request.

The affected entities are

- All ACX services, since the way they query data is changed

7.2.4 Updating the services

The changes proposed for ACX result in the adaption of the existing platform to the new solutions. After rigorous testing of the new design, the two systems should first both be functional with the new one only being accessible for testers and developers to ensure that errors in the new design are caught and fixed so that the platform continues to behave as expected. As the number of errors decrease, the old system can be gradually removed from production environments.

The removal of the AT traffic can be done almost immediately, so long as the LS has the updated protocol of generating the sessionId and sending it with the AT and SC to the Tzr, and the Tzr maps them together for the session. Then the orgId also needs to be mapped and stored in a new cache in the Tzr.

The Tzr will accept both the old and proposed requests from services, while incrementally introducing the changes for each service and thus moving the service from the list the Tzr uses the old protocol with, to the list using the new one where it validates the JWT and sends the orgId from within the JWT back to the service.

The PA could first only include all policies for one service and then add new services as it proves to correctly evaluate(same result as service evaluation) all requests. The services would instead first use both evaluations but first trusting its own evaluation more while sending notifications when the PA disagrees. When the PA is trusted the services' own evaluation can be removed.

Once the PA covers all services' AC enforcement, it can be built upon further with environmental factors and policies that can be used in tandem if ACX wishes to to further increase separation or to better control the usage of the platform.

The API for RLS needs only a few changes, to always use the orgId and to deny the query otherwise. The service needs to fetch the orgId from the Tzr instead of the request and always include it in the DB query.

7.2.5 Implementation considerations

Especially the new AC enforcement will take time to implement, due to its importance in all aspects of the platform and the magnitude of change. The code in the PA can be adapted from the individual services' AC code while constantly performing unit tests to see if the code from the services and the PA arrive at the same results for all requests. The new code in the services are all the same, requesting AC evaluation in the PA and then acting on its decision.

Since the PA is within the IAM module only, one group works on it while the changes to the services are sent to their teams by that group since all services receive the same responses.

The SSS cache has all its configurations changed from being adapted to S3 object storage to Redis storage instead, and since ACX does not use Redis currently its adoption requires time to understand the technology and how to best configure it.

The session changes affect all entities on the platform yet are fairly small from a development perspective. The exception is the Tzr cache, which is a new entity that needs to first receive the SC and AT, store them together, then replace the AT with the JWT as session representation.

The RLS changes are also small, only changing the API, thus only taking a few minutes to change. The larger changes each service encounters are instead how the orgId is received, and AC enforcement mentioned above.

The verification of the new implementations' correct functionalities is continually shown when both systems are used in tandem while incrementally updating the services. Penetration testing and integration testing should be performed before introducing any of the new changes to production environments as well.

8 Discussion and Conclusion

The thesis' purpose was to investigate Axis' PaaS Axis Connected Services, find alternative multitenant technologies, compare them to find what ACX does right and what could be improved. The goal was to find improvements to the tenancy separation on ACX. Both the purpose and goal of the thesis is fulfilled as it provides an analysis of all multitenancy concepts and their different uses, presenting useful information for other cloud service implementations.

The results are forcing tenant contexts for the users both with the session and data querying, and centralisation of AC enforcement. Results were motivated both through the needs of the platform, and of designs by other systems for a solution specifically for ACX, that includes general guidelines from the design of multitenant technologies that any cloud service could utilise.

Data at rest separation is increased through further verification of tenant association by services resulting in greater assurance of correct separation and a more clear separation policy. Access control enforcement is centralised and thus its processes become more transparent and also more intuitively reviewed and controlled to ensure homogeneity across ACX. Sessions are contextualised further by the JWT containing the orgId which further distinguishes users and makes separation for data at rest and AC more intuitive as all sessions are assigned an identifier. Only the centralisation of AC enforcement introduces significant costs however in time centralisation should help in decreasing costs with its many advantages for larger platforms.

The choice of not considering other data in use separation technologies proved well founded due to the level of privacy required for those using them was greater than that of ACX. By including access management in the investigation, more options for data at rest is presented and a more encompassing result is achieved.

8.1 Future work

There are many ways to build upon this work, especially in the implementation of the resulting proposal. Other areas of improvement include more use cases to compare with for the models ABAC and RBAC which can provide alternatives to the chosen AC model. Other investigations include the access evaluation services e.g Ory, OPA and Casbin for a more informed choice of policy agent technology or comparisons between CSPs e.g Google Cloud, IBM cloud, Azure, et.c to find whether the other CSPs offer greater options for ACX. Lastly investigating the efficiency of serverless processing(e.g Lambda) compared to dedicated servers(e.g EC2 or on-premise) could prove insightful for ACX's choice of only using serverless.

References

[1] L. C. Ochei and A. Petrovski, "Evolutionary computation for optimal component deployment with multitenancy isolation in cloud-hosted applications," 2018.

[2] A. Adewojo, J. Bass, and I. Allison, "Enhanced cloud patterns: A case study of multitenancy patterns," 2015. DOI: `10.1109/i-Society.2015.7366858`.

[3] T. Takahashi, G. Blanc, Y. Kadobayashi, D. Fall, H. Hazeyama, and S. Matsuo, "Enabling secure multitenancy in cloud computing: Challenges and approaches," in *2012 2nd Baltic Congress on Future Internet Communications*, 2012, pp. 72–79. DOI: `10.1109/BCFIC.2012.6217983`.

[4] X. Li, Y. Shi, Y. Guo, and W. Ma, "Multi-tenancy based access control in cloud," in *2010 International Conference on Computational Intelligence and Software Engineering*, 2010, pp. 1–4. DOI: `10.1109/CISE.2010.5677061`.

[5] K. Munir and S. Palaniappan, "Secure cloud architecture," *Advanced Computing: An International Journal*, vol. 4, 2013. DOI: `10.5121/acij.2013.4102`.

[6] P. Chinnasamy, B. Vinothini, V. Praveena, A. Subaira, and B. Ben Sujitha, "Providing resilience on cloud computing," in *2021 International Conference on Computer Communication and Informatics (ICCCI)*, 2021, pp. 1–4. DOI: `10.1109/ICCCI50826.2021.9402681`.

[7] R. Krebs, C. Momm, and S. Kounev, "Architectural concerns in multi-tenant saas applications," 2012.

[8] W. Brown, V. Anderson, and Q. Tan, "Multitenancy - security risks and countermeasures," in *2012 15th International Conference on Network-Based Information Systems*, 2012, pp. 7–13. DOI: `10.1109/NBiS.2012.142`.

[9] S. Walraven, W. De Borger, B. Vanbrabant, B. Lagaisse, D. Van Landuyt, and W. Joosen, "Adaptive performance isolation middleware for multi-tenant saas," in *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*, 2015, pp. 112–121. DOI: `10.1109/UCC.2015.27`.

[10] P. Seamark and T. Martens, "Row-level security," in 2019, ISBN: 978-1-4842-4896-6. DOI: `10.1007/978-1-4842-4897-3_11`.

[11] D. Ferraiolo and D. Kuhn, "Role-based access control," in *Proceedings of the 15th National Computer Security Conference*, 1992, pp. 552–563.

[12] T. Eltaeib and N. Islam, "Taxonomy of challenges in cloud security," in *2021 8th IEEE International Conference on Cyber Security and Cloud Computing (CSCloud)/2021 7th IEEE International Conference on Edge Computing and Scalable Cloud (EdgeCom)*, 2021, pp. 42–46. DOI: `10.1109/CSCloud-EdgeCom52276.2021.00018`.

[13] M. Zou, J. He, and Q. Wu, "Multi-tenancy access control strategy for cloud services," in *2016 10th International Conference on Software, Knowledge, Information Management and Applications (SKIMA)*, 2016, pp. 258–261. DOI: `10.1109/SKIMA.2016.7916229`.

[14] CISA, "Trusted internet connections 3.0: Cloud use case," 2022.

[15] A. Sharma and P. Kaur, "Implementing a multitenant system using a document-based nosql database," in *2021 5th International Conference on Information Systems and Computer Networks (ISCON)*, 2021. DOI: `10.1109/ISCON52037.2021.9702362`.

[16] K. Gupta, S. Kumar, and O. Agnihotri, "Data isolation in multi-tenant saas environment," in *2016 International Conference on Computing, Communication and Automation (ICCCA)*, 2016, pp. 1290–1292. DOI: `10.1109/CCAA.2016.7813917`.

[17] A. Chetal, V. Manral, M. Bregkou, R. Ferreira, and D. Hadas, "How to design a secure serverless architecture," 2021.

[18] J. Gao, G. Wang, Z. Yang, and Z. Zhao, "A decentralized runtime environment for service collaboration: The architecture and a case study," in *2021 IEEE World Congress on Services (SERVICES)*, 2021. DOI: 10.1109/SERVICES51467.2021.00044.

[19] D. Chu, K. Zhu, Q. Cai, *et al.*, "Secure cryptography infrastructures in the cloud," in *2019 IEEE Global Communications Conference (GLOBECOM)*, 2019, pp. 1–7. DOI: 10.1109/GLOBECOM38437.2019.9014033.

[20] J. Franklin, "Remote detectfion of virtual machine monitors with fuzzy benchmarking," in *SIGOPS Oper. Syst. Rev.*, 2008.

[21] J. Huang, D. M. Nicol, and R. H. Campbell, "Denial-of-service threat to hadoop/yarn clusters with multi-tenancy," in *2014 IEEE International Congress on Big Data*, 2014, pp. 48–55. DOI: 10.1109/BigData.Congress.2014.17.

[22] L. C. Ochei, J. M. Bass, and A. Petrovski, "Evaluating degrees of multitenancy isolation: A case study of cloud-hosted gsd tools," in *2015 International Conference on Cloud and Autonomic Computing*, 2015, pp. 101–112. DOI: 10.1109/ICCAC.2015.17.

[23] G. B. Pallavi and P. Jayarekha, "An efficient resource sharing technique for multi-tenant databases," in *2017 2nd IEEE International Conference on Recent Trends in Electronics, Information and Communication Technology (RTEICT)*, 2017, pp. 90–95. DOI: 10.1109/RTEICT.2017.8256564.

[24] S. Stamm, B. Sterne, and G. Markham, "Reining in the web with content security policy," 2010, pp. 921–930. DOI: 10.1145/1772690.1772784.

[25] Z. Xia, X. Wang, X. Sun, and Q. Wang, "A secure and dynamic multi-keyword ranked search scheme over encrypted cloud data," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 2, pp. 340–352, 2016. DOI: 10.1109/TPDS.2015.2401003.

[26] V. Hu, M. Iorga, W. Bao, A. Li, Q. Li, and A. Gouglidis, "General access control guidance for cloud systems," 2020. DOI: https://doi.org/10.6028/NIST.SP.800-210.

[27] K. Randolph and M. Hunt, "March 9, 2021 security incident report," 2021.

[28] D. Hardt. "Rfc6749." (2012).

[29] OpenId. "Openid connect." (2023), [Online]. Available: https://openid.net/connect/.

[30] M. Jones, J. Bradley, and N. Sakimura. "Rfc7519." (2015).

[31] V. Hu, D. Ferraiolo, R. Kuhn, *et al.*, "Guide to attribute based access control(abac) definition and considerations," in *NIST Special Publication 800-162*, 2014.

[32] OPA. (2023), [Online]. Available: https://github.com/open-policy-agent/opa/blob/main/ADOPTERS.md.

[33] Casbin. (2023), [Online]. Available: https://casbin.org/docs/adopters.

[34] NIST. "Separation of duties." (2023), [Online]. Available: https://csrc.nist.gov/glossary/term/Separation_of_Duty.

[35] H. Hong, Y. Xia, and Z. Sun, "Towards secure data retrieval for multi-tenant architecture using attribute-based key word search," *Symmetry, [s. l.], v. 9, n. 6*, 2017. DOI: 10.3390/sym9060089.

[36] OWASP. "Session management cheat sheet." (2023), [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html.

[37] M. Factor, D. Hadas, A. Hamama, *et al.*, "Secure logical isolation for multi-tenancy in cloud storage," 2013, ISBN: 978-1-4799-0217-0. DOI: 10.1109/MSST.2013.6558424.

[38] A. Adewojo, J. Bass, K.-y. Hui, and I. Allison, "Cloud deployment patterns: Migrating a database driven application to the cloud using design patterns," 2015.

[39] Y. Zhang, Y. Sun, R. Jin, K. Lin, and W. Liu, "High-performance isolation computing technology for smart iot healthcare in cloud environments," *IEEE Internet of Things Journal*, vol. 8, no. 23, pp. 16 872–16 879, 2021. DOI: 10.1109/JIOT.2021.3051742.

[40] A. D. Williams, T. Adams, J. Wingo, *et al.*, "Resilience-based performance measures for next-generation systems security engineering," in *2021 International Carnahan Conference on Security Technology (ICCST)*, 2021, pp. 1–5. DOI: 10.1109/ICCST49569.2021.9717388.

[41] H. Lin, K. Sun, S. Zhao, and H. Yanbo, "Feedback-control-based performance regulation for multi-tenant applications," 2009. DOI: 10.1109/ICPADS.2009.22.

[42] L. Rodero-Merino, L. M. Vaquero, E. Caron, A. Muresan, and F. Desprez, "Building safe paas clouds: A survey on security in multitenant software platforms," vol. 31, pp. 96–108, 2012, ISSN: 0167-4048. DOI: 10.1016/j.cose.2011.10.006..

[43] Okta. "Authorization code flow." (2023), [Online]. Available: https://auth0.com/docs/get-started/authentication-and-authorization-flow/authorization-code-flow.

[44] OWASP. "Owasp top ten proactive controls 2018: Validate all inputs." (2018).

[45] V. Singh and S. Patel. "Is amazon rds for postgresql or amazon aurora postgresql a better choice for me?" (2021), [Online]. Available: https://aws.amazon.com/blogs/database/is-amazon-rds-for-postgresql-or-amazon-aurora-postgresql-a-better-choice-for-me/.