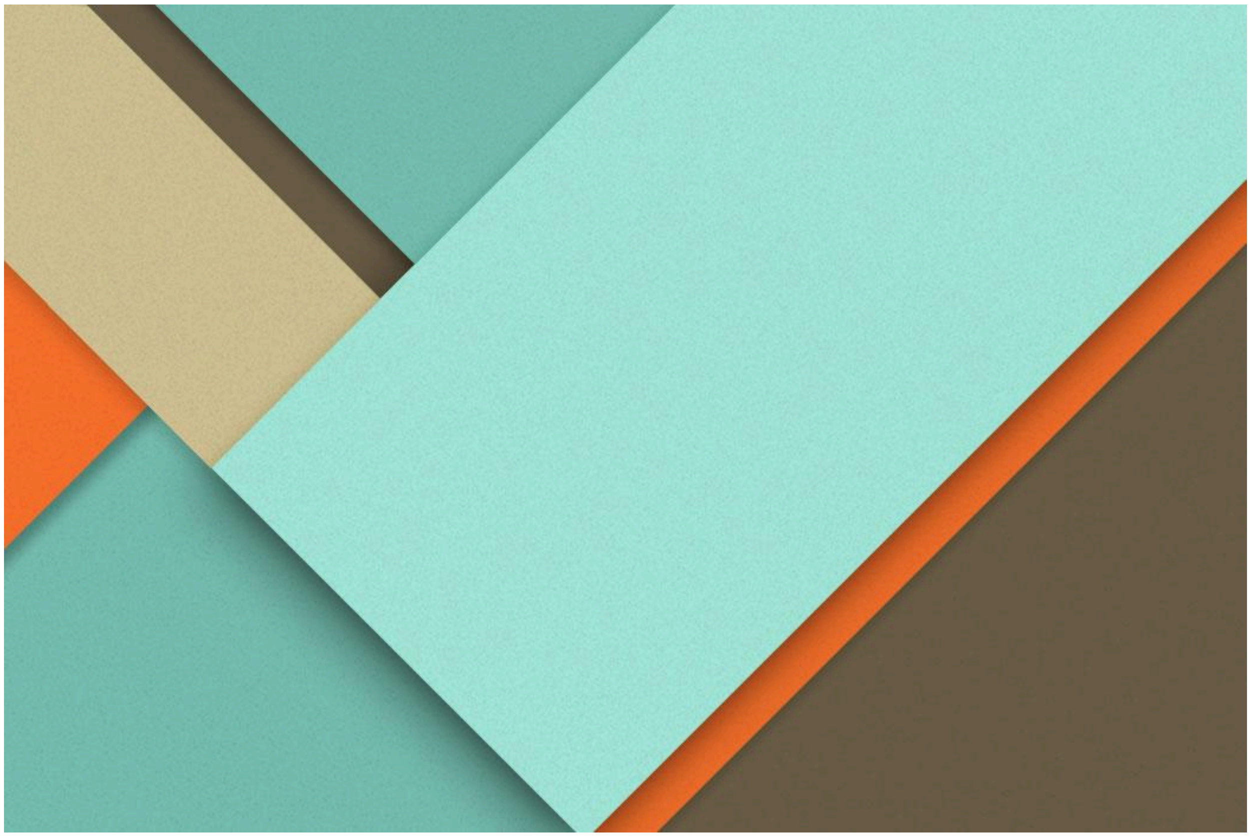


Architectural Document

Avalanche Analytics

—

The Skunkworks
For DNS Business PTY LTD

theskunkworks301@gmail.com

Capstone Project

University of Pretoria

Introduction

Avalanche Analytics is a project developed by The Skunkworks, that aims to provide a scalable and user-friendly data analytics platform to analyse the domain name space in South Africa

Design Strategy

The architectural decisions made by the team are based on the quality requirements as identified by the team and prioritised by the client. The functional requirements and user stories that have been identified have been used to identify the most important quality requirements.

1. Alignment with Business Requirements:

The selection of a design strategy based on quality requirements is primarily due to its significant alignment with the business requirements. In the context of a Management Information System (MIS), quality is of paramount importance. Quality attributes such as reliability, performance, scalability, usability, and security are crucial for an effective MIS.

2. User-Centric Design:

Designing based on quality requirements puts the users' needs and expectations at the forefront. Since Avalanche is typically used by managers and decision-makers, their needs and requirements are crucial to consider. This includes factors like ease of use, information accuracy, system availability, and responsiveness, which directly correlates to the quality requirements.

3. Facilitates Long-term Maintenance:

A system designed based on quality requirements is generally more maintainable in the long run. When a system is created with a focus on aspects such as modularity, performance, reliability, and security, it is typically easier to identify and fix issues, add new features, and scale as needed

While other strategies like decomposition and generating test cases are important and will be applied during different phases of the project, our main architectural design strategy will focus on quality requirements. This ensures the construction of a robust, reliable, maintainable, and user-friendly MIS that meets and exceeds the expectations of its users and stakeholders.

Quality Requirements

The following Quality Requirements have been identified by the team and the client. They are listed in order of importance and discussed in some detail below.

1. Security
2. Extensibility
3. Usability and Transparency
4. Interoperability
5. Efficiency/Performance

I. Security

Security has been identified as the most important quality attribute of the system. The data in the Snowflake Warehouse contains the personal information of individuals that are protected by the POPI Act. Furthermore, it contains important record-keeping information of the client that is needed for annual auditing and thus cannot be corrupted or lost. The CIA principle is used to quantify security measures. Confidentiality is realised by only allowing authorised users with the correct permissions checked on multiple levels to access sensitive endpoints, as well as using Snowflake masking. Integrity is enforced by making access from the system to the Snowflake warehouse where personal information is stored read-only. Authorization in a login being required and requiring the correct integrations.

Stimulus Source
Malicious actors / Unauthorised users
Stimulus
Attempts to access non-aggregated data/data authorised for another user profile.
Response
● The system should log all requests to keep track of unauthorised data access (Audit Trail). ● The system should only authorise users with the correct permissions/integrations to access data, by only allowing access to specific Snowflake views which are already aggregated ● The system should ensure access to the Snowflake Warehouse from the API is read-only
Response Measure
● Unauthorised access attempts identified and flagged ● Only authorised views are queried ● The Snowflake database has not been modified
Environment
System running normally
Artifact
Primarily any entry points into Snowflake like an API endpoints/SQL Builder queries

II. Extensibility

The product should be designed in such a way that additional data products can be added with ease. The client has identified this project as one with the potential to grow and expand into different areas besides only data analytics in the domain name space. The system should allow for additional products/services to be added with a limited effect on current system functionality.

Stimulus Source
Client/Developer
Stimulus
Wishes to add new data products (services) in addition to the existing system products (incremental deployment)
Response
● Add integration (service) ● Test integration
Response Measure
● None of the existing services needs to be modified other than the central Service Bus/Gateway API ● Minimal changes are needed to the user management database ● Minimise the cost (financial and time) of adding additional features
Environment
An already deployed system
Artifact
The source code of the system

III. Usability and Transparency

Usability is a crucial aspect of any software product. The client phrased their requirement more specifically as usability and transparency. The design should be intuitive, customisable and accessible, however, the information displayed should also be done so in a transparent manner. It should be clear to the user exactly what data is being displayed and how it has been gathered, ie. whether it is definitive or whether statistical analysis was applied. Details should be provided in the case of non-definitive data analytics.

Stimulus Source
End User
Stimulus
Wishes to interact with the system in a way that is user-friendly, intuitive and transparent
Response
● Create an interface that offers easy navigation ● Create customisation by allowing custom dashboards ● Create a system that offers users a clear understanding of the data including disclaimers to any data where there is no guarantee of correctness
Response Measure
● High levels of user satisfaction are achieved through the usability of the tool. It is measurable through usability testing sessions and feedback. ● Transparent presentation of data, especially in cases of statistical analysis. This transparency is measured by how well users understand the data and can make informed decisions based on it. ● Less time and fewer steps are needed to perform typical tasks. ● Reduced user errors and quick recovery when errors do occur.
Environment
The design phase of the system
Artifact
The User Interface (UI) and User Experience (UX) design of the system. This includes all the visual elements of the system, as well as the documentation and metadata explaining data sources, gathering methods, and applied statistical analyses.

IV. Interoperability

The analytics tool should be designed to easily integrate with the registry operator's existing systems. One of the main concerns is to authenticate registrars through their system. Thus, the architecture should prioritize seamless interoperability with the existing authentication mechanisms in the registry operator's system to ensure a secure and straightforward user experience.

Stimulus Source
Client
Stimulus
Wishes to integrate the analytics tool with their existing authentication system (interoperability)
Response
● Design and implement a module or functionality in the tool that integrates with the client's existing authentication system ● Test the integration
Response Measure
● The integration is successfully implemented without the need for modifications to the existing authentication system ● No significant changes are required to the underlying databases of the analytics tool or the client's system ● Minimal cost (both in terms of time and financial resources) involved in implementing this integration ● High success rate of registrar authentication through the integrated system ● Low latency during the authentication process to ensure a seamless user experience
Environment
A system under development, aimed to be deployed within the registry operator's infrastructure
Artefact
The module or functionality in the source code of the analytics tool that integrates with the registry operator's existing authentication system

V. Efficiency

Efficiency is an essential quality requirement for the proposed business analytics tool which will be analyzing the domain name space in South Africa. In the context of our system, efficiency refers to the ability of the system to provide timely and accurate data analytics results with optimal resource utilization. This is critical given the potential scale of data to be processed and the need to deliver insights in a timely manner to enable informed decision-making.

Stimulus Source
Client/User
Stimulus
User needs to process and analyze a large dataset within a short timeframe to derive meaningful insights.
Response
● The system quickly processes the data, performs analytics, and delivers results.
Response Measure
● The time taken to process and analyze the data is within acceptable limits defined by the client. ● The system successfully handles the load without significant degradation in performance or increase in errors. ● Optimal usage of computational and storage resources during the process, minimizing costs.
Environment
During peak usage times and under high data volume conditions.
Artefact
The system components involved in data processing and analytics – Snowflake warehouses, Domain Watch service, registry database services, API Gateway, and the frontend client.

(Note that this is the lowest priority as identified by the client and thus sometimes needs to be sacrificed in order to achieve quality requirements that are higher priority)

Architectural Constraints

1. The system requires the use of Snowflake Data Warehouse
    1.1. Snowflake tokens are charged per minute of execution time
2. The tool is to be deployed on-premise
3. Desired to be RESTful API calls
4. Integrate with existing API for registrar integration authentication
5. Security and Compliance Constraints:
    5.1. Given the sensitive nature of domain name data, there are significant security constraints.
    5.2. Secure authentication and access controls are needed.
    5.3. Moreover, there are compliance constraints related to privacy laws such as the Protection of Personal Information Act (POPIA) in South Africa.

Architectural Styles

The architecture of this project follows mainly a micro-service oriented architecture. Microservices allows the project to meet various quality requirements as set out by the client.

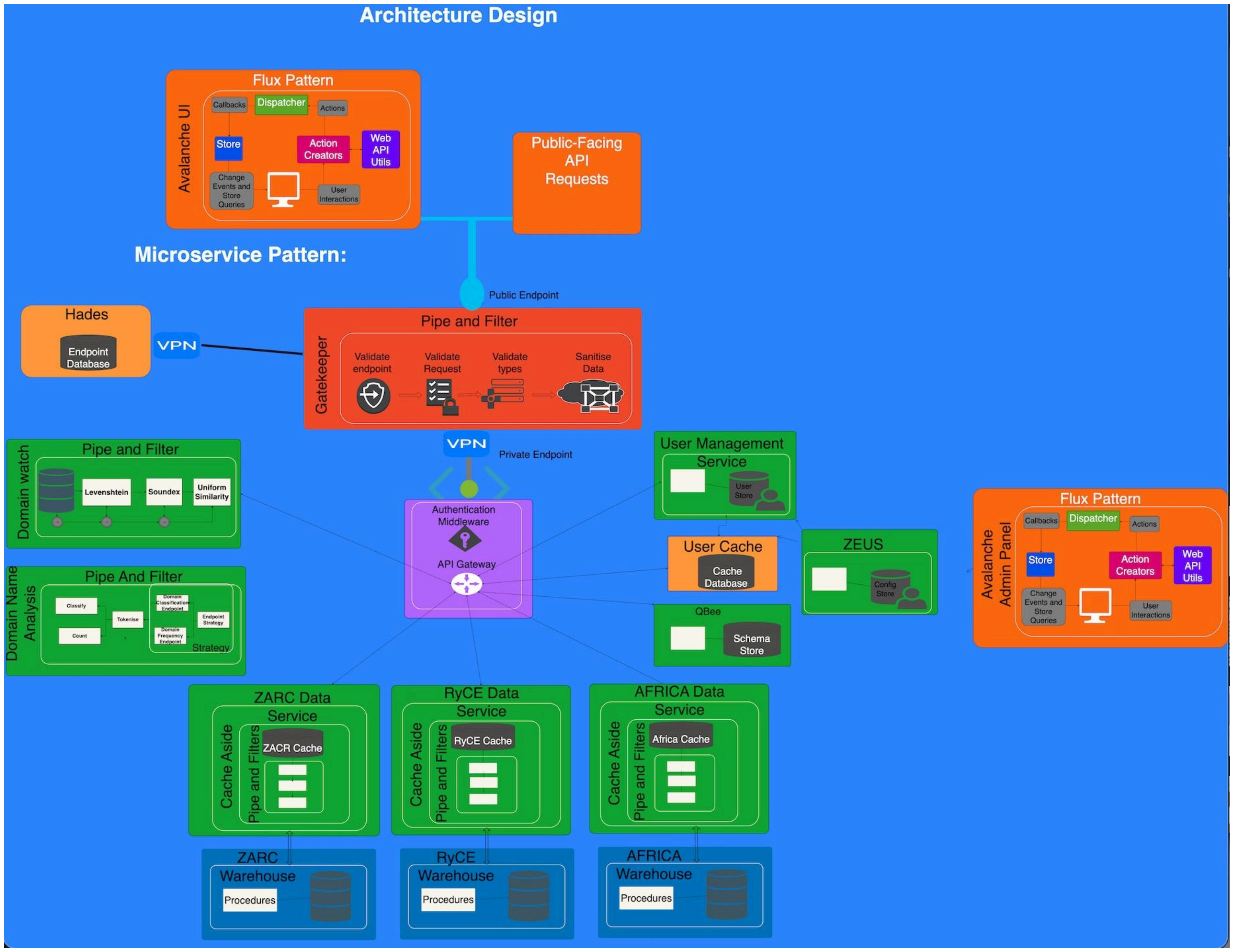
Interoperability is achieved through the services being separated into services which can be easily replaced, maintained or swopped for other services that adhere to the contracts defined. Extensibility is achieved through the low coupling generated by the use of a Gateway which routes between services. Furthermore, additional services can be deployed with minimal effect on currently deployed services. Services can be distributed across servers or can be located on a single server. Resources can be scaled per service helping to achieve scalability as well as performance by allocating resources where they are most necessary.

While microservice architecture does not result in improved security out of the box, using the Gatekeeper pattern in combination with it results in a secure pattern with reduced vulnerabilities. The Gatekeeper can validate and control access to the system and resources while all other services are hidden behind this layer on a private network. This Gatekeeper is designed to be extensible and modular, and through the deployment of multiple instances working independently also achieves scalability.

To improve usability on the frontend the components are based on the Flux architecture which manages state and separates the data from the view. Furthermore, a component based frontend allows for reuse and improves extensibility, since existing components can be employed in new services.

Various pipe-and-filter patterns are employed in the system. Given the Avalanche project's nature, which involves a considerable amount of data processing, transformation, and transmission, the Pipe and Filter pattern becomes very suitable. It enables high throughput, flexibility, and reusability. The pattern allows components to be reused across different parts of the system and promotes decoupling, as each filter has no knowledge of what happens beyond its input and output pipe.

System Architectural Design



System Overview

Our system is composed of various components including a frontend client, an API Gateway, microservices for separate registry databases and user management, a Domain Watch service, and separate Snowflake warehouses for each registry. These components are arranged in a microservices architecture, with a central gateway, user management micro-service, registry products services, and the Domain Watch service running in a separate container. Some of these microservices also employ their own architectural patterns to meet specific needs. An overarching Gatekeeper acts as a filter between clients and the API Gateway, ensuring all incoming requests are clean and valid. This Gatekeeper is the only publicly accessible endpoint and the single entry point to our system.

Flux Architectural Pattern

At its core, Flux is a pattern that encourages a unidirectional data flow, which can significantly improve the usability of the application for both developers and end-users. The Avalanche project is expected to have a rich user interface with multiple user interactions. The Flux pattern, with its unidirectional data flow, offers a predictable data flow, which makes the application easier to track and understand. It ensures consistency across multiple components, leading to fewer unexpected behaviors and more manageable code

Gatekeeper

In order to address the security quality attribute a Gatekeeper architecture has been employed. The gatekeeper protects applications and services using a dedicated host instance that acts as a broker between clients and the services, validates and sanitizes requests, and passes requests and data between them. It is the single point of entry for the system. The rest of the system runs on an internal network protected by a VPN. This ensures that all requests that are passed through to the system have been screened for malicious data.

One of the weaknesses of the Gatekeeper is that it may affect performance. Since Security is the primary Quality Requirement set out by the client, it is of higher priority in the system architecture design. However, to alleviate the potential bottleneck and single point of failure it is possible to replicate the Gatekeeper. This can then be scaled as necessary to

improve performance and improve reliability by reducing the impact of the failure of a Gatekeeper.
It is enhanced by using a pipe and filter to add more security to the system

Microservices

Microservices are used to meet the extensibility requirement of the client. Extra data products can be added as additional microservices without affecting the deployed system, each with its own database if needed. The central Gateway is configured to route to all microservices as they are requested. An additional layer of security is also implemented at this stage by ensuring that the user is authenticated and that the user has the correct permissions to access the requested product/service.

Various patterns are employed within the Microservice Architecture:

1. Gateway

A central Gateway Routing is used to expose all the microservices at a single endpoint. The Gateway handles the routing of the requests to the correct microservices. This allows the client to connect this endpoint regardless of the services available. It also can help to make services more scalable. If a single service becomes overloaded, multiple instances of the service may be spun up. Instead of changing the client the Gateway can manage the routing to these.

The Gateway also employs a layered approach. The entry-point is the first “layer” in lorder to analyse where the request should be routed. Thereafter, the request could be sent through a token based authorisation middleware or bypass this middleware in the case of login, register and verify.

2. Cache Aside

Cache Aside is used to decrease latency and improve performance. This in turn also addresses usability since data can be delivered to users at a higher pace. The microservices that handle the requests to the various data Warehouses each have a cache database to store recent retrievals. Thus when the same data is requested again, the service can access it from cache instead of making a call to the external warehouse. This also saves on the cost of Snowflake “tokens” addressing client constraints.

3. Pipes And Filters

Pipes and Filters are used to process data in the Domain Watch microservice. The process contains various steps in order to identify strings that are a close match to he passed in parameter. In order to make this service scalable and to be able to edit/optimise parts of the processing in isolation a pipes and filters approach is used. Additional processing components can then be added, or existing ones replaced or removed without refactoring the code.

Practically there are multiple pipelines of pipes and filters running concurrently in order to improve processing efficiency. A key advantage of the pipeline structure is that all pipelines can run in parallel or only slow filters can run in parallel while faster filters have a single instance.

Addressing Quality Attributes

1. Security

The system addresses the crucial attribute of security with a layered approach. First, at the entry point, the Gatekeeper validates and cleans all incoming requests, offering a first line of defense against malformed or malicious data. Next, the API Gateway, acting as the entry point to the system, further screens the incoming requests before routing them to the appropriate services.

The Gateway employs token based middleware to protect the system from unauthorized access. This token is session based improving security once again.

Within the system, individual services, such as user management, ensure that only authorized users with correct permissions can access sensitive information. The user management service, with its dedicated PostgreSQL database, maintains strict access controls. Moreover, our data storage system—individual Snowflake warehouses for each registry—complies with the POPI Act by masking sensitive information and providing read-only access, thus ensuring data integrity and confidentiality.

2. Extensibility

The system is designed with extensibility in mind, allowing for easy addition of new products or services with minimal effect on existing functionality. The microservices architecture employed by our system inherently supports extensibility, as each service is isolated and independently deployable. This design makes it straightforward to add, modify, or remove services as necessary, allowing the system to adapt to evolving requirements and growth opportunities.

3. Usability and Transparency

Our frontend client ensures high usability with server-side rendering, providing a smooth and responsive user interface with low loading times. The design is intuitive, customizable, and accessible. Moreover, the system prioritizes transparency, particularly in terms of data representation. Information displayed to the user is clearly marked as definitive or statistically analyzed, providing clarity on data sources and processing methods.

The client will be responsive for all screen sizes, ensuring that the users can use the frontend client in the office or on the go. There will also be methods to alter data, such as changing the graph type and zooming into the graph to help with usability.

Data will be displayed in a hierarchical manner, and the ability to filter and sort this data is also essential to our aim of usability. We will also be using the FATE principles when it comes to data visualization, ensuring that the data will be fair, the methods we used to get this data is accountable, the methods we used are transparent and that they can be explained. By using FATE, we are ensuring that we meet the requirements of transparency.

4. Interoperability

Our architecture prioritizes interoperability to seamlessly integrate with the registry operator's existing systems. The microservices design, with its service-specific databases, allows for easy and flexible integration with external systems. The user management service is a key example, as it is designed to authenticate registrars through the client's system.

5. Efficiency

Our architecture achieves efficiency by using the cache-aside pattern which allows the system to respond to requests that have been made recently much faster. This will be employed both in the User Management subsystem as well as the Registry subsystems. This will decrease latency by avoiding unnecessary access to the remote warehouse. The micro-service approach also ensures that multiple instances of the same microservice can be deployed to decrease load.

Conclusion

Our system architecture successfully addresses the client's defined quality attributes of security, extensibility, usability and transparency, and interoperability. The thoughtful application of a microservices architecture, coupled with specific security measures, transparent data handling practices, and a user-centered frontend design, positions our system as a secure, adaptable, user-friendly, and integrable solution for domain name space analytics.

Technology

Frontend:

1. NextJS built on ReactJS

Overview

React is a JavaScript library for building user interfaces, primarily for single-page applications. It's used for handling the view layer in web and mobile apps. React allows you to design simple views for each state in your application, and React will efficiently update and render the right components when your data changes.

Pros

- Fast rendering and out of the box server side rendering,
- SEO friendly,
- Strong community support,
- Component-based structure,
- and easy to learn.

Cons

- High pace of development, which might be hard to keep up with,
- JSX as a barrier

Fit

React's component-based architecture aligns perfectly with the Microservices architecture.

It allows developers to build large scale applications with reusable components, improving maintainability, and testability.

React also allows for state management in order to implement the Flux Pattern.

2. Angular

Overview

Angular is a platform for building web applications. Developed by Google, it offers a unified approach towards developing web applications by combining declarative templates, dependency injection, end-to-end tooling, and integrated best practices.

Pros

- High performance,
- robust tooling
- and excellent community support.

Cons

- Steep learning curve,
- complexity,
- and verbosity of the code.

Fit

Angular provides libraries to manage Redux state to achieve the Flux pattern in the frontend. It also allows for component based design perfect for implementation of the microservice architecture.

While Angular offers a comprehensive solution for building web applications, its complexity and the steep learning curve make it less suitable for our project compared to React.

3. Vue.js

Overview

Vue.js is a JavaScript library for building web interfaces. It provides data-reactive components with a simple and flexible API.

Pros

- Simplicity and ease of use,
- flexibility,

- and detailed documentation.

Cons

- Less community support
- and fewer resources compared to React and Angular.

Fit:

Vue offers vuex for state management, suitable to implement a Flux-like pattern.

While Vue.js offers simplicity and flexibility, its relatively smaller community support makes it less preferable.

Final Choice:

NextJS due to its large community, server side rendering, and fast rendering capabilities and component based design which align perfectly with our Microservices architecture.

Gatekeeper:

1. NestJS

Overview

NestJS is a framework for building efficient, scalable Node.js server-side applications. It uses progressive JavaScript and is built with TypeScript. NestJS makes use of robust HTTP server frameworks like Express and Fastify

Pros

- Leveraging the power of TypeScript,
- supporting a modular architecture,
- a well-defined programming model,
- and compatibility with JS libraries.

Cons

- Steep learning curve for developers not familiar with TypeScript

Fit

NestJS's robust, modular architecture and TypeScript support makes it a perfect fit for implementing a secure Gatekeeper.

2. ExpressJS

Overview:

ExpressJS is a web application framework for Node.js, released as free and open-source software under the MIT License. It is designed for building web applications and APIs.

Pros:

- Simple,
- unopinionated,
- and flexible.

- It's easy to customize and extend.

Cons:

- Lack of structure can lead to unorganized code,
- More lightweight for smaller APIs

Fit

While ExpressJS is a popular choice, lack of structure might not be the best fit for implementing a secure Gatekeeper.

3. Koa.js

Overview

Koa.js is a next-generation web framework for Node.js created by the team behind Express. It aims to be a smaller, more expressive, and more robust foundation for web applications and APIs.

Pros

- Lightweight, expressive, and robust
- Easy error-handling
- supports ES6 features out of the box.

Cons

- Smaller community support, not as popular as Express.

Fit

Despite its lightweight and robust nature, Koa's smaller community and less popularity make it less suitable.

Final Choice:

NestJS for its robust, modular architecture, TypeScript support, and capability to build efficient and scalable server-side applications aligning with our Microservices architecture.

User Management Database:

1. PostgreSQL (Postgres) - constraint: client required

Overview:

PostgreSQL, also known as Postgres, is a free and open-source relational database management system emphasizing extensibility and SQL compliance. It offers advanced data types and robustness.

Pros:

- ACID compliant, robust and stable
- excellent support for JSON,
- strong community support.
- Define entities from JS to load database schema

Cons

- Could be slower than other databases for read-heavy workloads,
- relatively high complexity.

Fit

Given the need for a reliable, robust, and ACID-compliant solution for user management, Postgres is a strong candidate. EXtensive JSON support which works well with JavaScript backends.

2. MySQL

Overview

MySQL is a widely used open-source relational database management system. It is popular for web applications and is a central component of the LAMP open-source web application software stack.

Pros

- Fast read operations,
- easy to set up and use,
- wide acceptance in the industry.

Cons

- Less robust for complex queries and transactions,
- might not handle large databases as well as other systems.

Fit

Although MySQL is quite popular, it does not have extensive json support makingit less flexible for our use case.

3. MongoDB

Mongo is a NoSQL, document-based database using key values and storing data in JSON format for easy interpretation and transmission.

Pros

- Designed for large scale data aggregation and processin
- Easy input and output of data due to being in JSON format

Cons

- Processing can become expensive for large data sets
- Relationships can become messy and complex

Fit

Although Mongo is a popular choice and supports JSON, relationships can become quite complex. The nature of the User data introduces many relationships making it a less suitable choice

Final Choice:

PostgreSQL (Postgres) for its ACID compliance, robustness, and excellent JSON support. PostgreSQL was a client constraint.

Cache:

1. Redis

<u>Overview</u>

Redis is an open-source, in-memory data structure store, used as a database, cache, and message broker. It supports various types of data structures such as strings, hashes, lists, sets, and more.

<u>Pros:</u>

- Exceptionally fast,
- supports various data types,
- supports replication and persistence.

<u>Cons</u>

- Data size can't exceed available memory,
- relatively complex clustering.

<u>Fit</u>

Redis's speed and data structure support make it a perfect fit as a cache for our microservices architecture’s cache aside pattern.

2. Memcached

<u>Overview</u>

Memcached is a high-performance, distributed memory object caching system intended for use in speeding up dynamic web applications by alleviating database load.

<u>Pros</u>

- Easy to install and use,
- speeds up web applications by caching database queries.

Cons

- Doesn't support data structures like Redis, not as versatile as Redis.

Fit

Although Memcached can be a good caching system, it lacks the versatility and data structure support that Redis offers.

3. Hazelcast

Overview

Hazelcast is an open-source in-memory data grid based on Java. It allows you to spread data and computation across multiple machines.

Pros

- Supports a wide range of distributed data structures,
- provides out-of-the-box distributed computing capabilities.

Cons

- More complex to install and set up than Redis,
- smaller community.

Fit

Hazelcast's features are useful, but its added complexity and smaller community make it less suitable than Redis for our project.

Final Choice:

Redis due to its exceptional speed, versatile data type support, and replication capabilities. The client also uses Redis and preferred it over other choices to maintain consistency.

Domain Watch:

1. Java

Overview:

Java is a widely used programming language expressly designed for use in the distributed environment of the internet. It's known for its robustness, simplicity, and cross-platform capabilities.

Pros

- Platform independent, object-oriented, robust and secure,
- Multithreading support
- Vast open-source library ecosystem.

Cons

- Can consume more memory and CPU resources,
- Slower than some other languages like C++.

Fit

Java's robustness, security, threading, and vast open-source libraries make it a perfect fit for our Domain Watch service.

2. Python

Overview

Python is an interpreted, high-level, general-purpose programming language. It's popular for its simple syntax and readability.

Pros

- Simple and easy to learn,
- large community and library support,
- good for data analysis and machine learning.

Cons

- Slower than languages like Java or C++.
- Singlethreaded

Fit

Despite its strengths, Python's performance may not meet the demands of our Domain Watch service.

3. Rust

Overview

Rust is a statically-typed programming language designed for performance and safety, especially safe concurrency and memory management with syntax similar to C

Pros

- Supports concurrency
- High performance
- Type and memory safety

Cons

- Newer programming language thus may have less library support
- Learning curve

Fit

Rust is high performance and memory safe making it a good option for the resource intensive domain watch service

Final Choice:

Java due to its robustness, security, cross-platform capabilities, and vast open-source library ecosystem.

Python for some Machine Learning and AI components due to extensive library support.

Final Technology Stack

- Snowflake Data Warehouse:
    - Procedures using Javascript
- Frontend:
    - NextJS
    - Chart.js
- Gateway:
    - NestJS
    - JWT
    - Redis
- User Management
    - NestJS
    - Redis
    - PostgreSQL
- Services:
    - Java
    - Python

To achieve the requirements, the project team will leverage **Snowflake®'s data warehousing** technology and develop microservices using **Java/NestJS**. The team will also use **NestJS** as the primary API framework to provide a robust and scalable interface for the microservices. The team will also utlise **Redis** as a caching service to enhance user experience through faster load times. The team will user a **postgreSQL** database to manage users

For the frontend, the team will use **NextJS** to create an intuitive and user-friendly interface that enables clients to access and visualise statistical data. Graphical information will be displayed using **chart.js.**

1. Frontend
    a. UI: Use NextJS for building the web-based user interface for the BIT. NextJS provides a component-based approach, which enables a modular and maintainable structure for the frontend.
    b. Graphing Service: This service will use chart.js to generate graphs that can then be displayed by the Frontend UI. This allows for different graphing libraries to be plugged into the system with minimal effects
    c. Reporting Service: Implement a reporting service that generates daily/weekly reports in PDF format containing graphical and tabular data, as well as analyses of the data. This will utlise the graphing service as well.

2. API NestJS will be used as the main API gateway, while different microservices will be developed using Java, NestJS and other languages as needed. Each microservice will handle a specific set of functionalities, such as data analytics, authentication, and profile management.
    a. API Gateway: The NestJS API gateway will handle the public-facing API, providing secure, token-based access to Avalanche data for clients. The API gateway will route API requests to the appropriate microservices for processing and data retrieval. This provides a single entry point for external clients to access the system's services. The API gateway handles request routing, composition, and protocol translation, simplifying the client's interaction with the system.
    b. Authentication, Authorization and User Management: Implement an authentication and authorization system using JWT tokens to ensure secure access for different types of users (Third-Party Users, Registrar Users, Registry Users). This will also handle role-based access control for different functionalities within the system.
    c. Data Processing and Analytics: Implement data wrangling and analytics microservices using suitable languages. These services will be responsible for transforming, cleaning, aggregating, and analysing the data stored in Snowflake®. They will provide insights and statistics for the frontend and API.
3. Data Storage: Use Snowflake® data warehouse technology for storing the domain data and statistics. Snowflake® provides a scalable and performance-optimised solution for data storage and retrieval.