Architectural Patterns

Architectural Constraints

Adherence to Legal Standards and Regulations:

POPIA Compliance

- The system architecture must incorporate robust data encryption mechanisms and access control measures to ensure compliance with the Protection of Personal Information Act (POPIA). This includes safeguarding personal information during data handling, storage, and processing.

Regular Legal Reviews

- The system architecture should be modular and scalable to facilitate regular reviews and updates without causing disruptions to the entire system. This allows for seamless integration of changes based on legal requirements and ensures continuous compliance with relevant laws.

Data Protection Officer

- Implement role-based access control within the system architecture to ensure that the Data Protection Officer (DPO) has appropriate access privileges to oversee data protection strategies and compliance activities.

Balancing Automation and Human Oversight:

Human-in-the-loop System

- The system architecture must support workflow orchestration to seamlessly integrate human oversight into critical decision-making processes. This involves designing workflows that enable efficient collaboration between automated systems and human operators.

Limiting Bias in the Dispute Resolution Process:

Algorithmic Transparency

- Architectural design should prioritize the use of explainable AI models, which provide clear explanations for their decisions, thereby reducing bias and facilitating audits. This involves structuring the system architecture to accommodate algorithms that offer transparency in decision-making processes.

Architectural Styles

In developing the Dispute Resolution Engine (DRE), we have selected several architectural patterns to address key quality requirements.

The Event-Driven Architecture (EDA) enables DRE to handle large volumes of events efficiently by allowing components to react to events asynchronously. This pattern significantly enhances the system’s scalability by facilitating horizontal scaling through the addition of more event processors. Additionally, EDA boosts performance by decoupling the production and consumption of events, allowing for real-time processing and efficient handling of high-throughput scenarios. Reliability is also improved, as events are typically stored in a persistent event log that can be replayed in case of system failures, ensuring data integrity and consistent operation.

Service-Oriented Architecture (SOA) is employed to enhance DRE’s scalability and maintainability. By encapsulating business logic within discrete, independently deployable services, SOA allows each service to be scaled independently, effectively managing growth of the system. This modular approach also simplifies maintenance as services can be updated, replaced, or extended without impacting other parts of the system. This ensures continuous improvement and adaptability to changing requirements and technologies.

The Gatekeeper pattern serves as a security layer within DRE, addressing essential quality requirements of security, reliability, and compliance. Acting as a single entry point for all incoming requests, the Gatekeeper enforces strict access control policies to ensure only authorized requests reach internal services. This pattern enhances system reliability by incorporating load balancing, caching, and failover mechanisms, thus maintaining smooth operation even under high demand or in the event of partial system failures. Moreover, the Gatekeeper helps ensure compliance with data privacy and security regulations by validating requests against predefined policies, thereby safeguarding sensitive information and ensuring regulatory adherence.

We make use of the Flux pattern to enhance the usability of the DRE by simplifying data flow and state management. Its unidirectional data flow ensures predictable and traceable state changes. By structuring the frontend into modular components, Flux makes the system easier to maintain and debug. This approach improves the user experience, offering developers an intuitive framework and providing end-users with a responsive and reliable interface.

Event-Driven Architectural Pattern

Quality Requirements Addressed:

1. Scalability
    - Event-driven architecture allows components to react to events asynchronously, enabling the system to handle large volumes of events efficiently and scale horizontally by adding more event processors.
2. Performance
    - By separating the responsibilities of the production and consumption of events, the system can process events in real-time and handle high-throughput scenarios efficiently.
3. Reliability
    - Events are typically stored in a persistent event log, which can be replayed in case of system failures, ensuring data integrity and system reliability.

Service-Oriented Architecture (SOA) Pattern

Quality Requirements Addressed:

1. Scalability
    - Services in an SOA can be deployed and scaled independently, allowing the system to grow and manage increased loads effectively.
2. Maintainability
    - By encapsulating business logic within discrete services, SOA makes it easier to update, replace, or extend functionalities without affecting other parts of the system.

Gatekeeper Architectural Pattern

Quality Requirements Addressed:

1. Security
    - The Gatekeeper pattern acts as a security layer that enforces access control policies, ensuring that only authorized requests are allowed to reach the internal services.

2. Reliability
    - By routing requests through a single entry point, the Gatekeeper pattern can provide load balancing, caching, and failover mechanisms to improve the reliability of the system.
3. Compliance
    - The Gatekeeper pattern can enforce compliance requirements by validating requests against predefined policies and ensuring that data privacy and security regulations are met.

Flux

Quality Requirements Addressed:

1. Usability
    - Flux architecture simplifies the data flow in the system, making it easier to understand and maintain. This enhances the usability of the system by providing a clear and predictable data flow.

Conclusion of Choices

Event-Driven Architecture

1. Scalability and Performance
2. Reliability

Service-Oriented Architecture (SOA)

1. Scalability
2. Maintainability

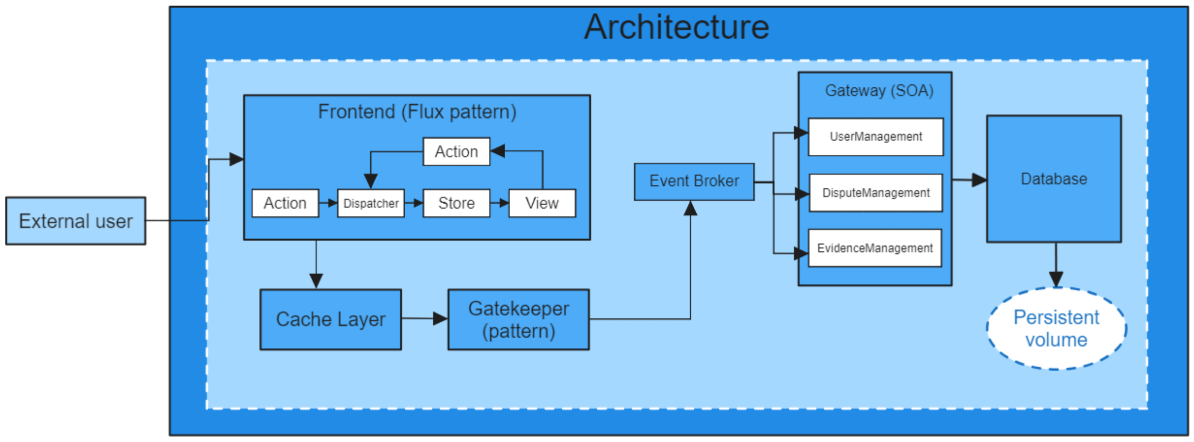
Gatekeeper Pattern

1. Security
2. Reliability
3. Compliance

Flux Pattern

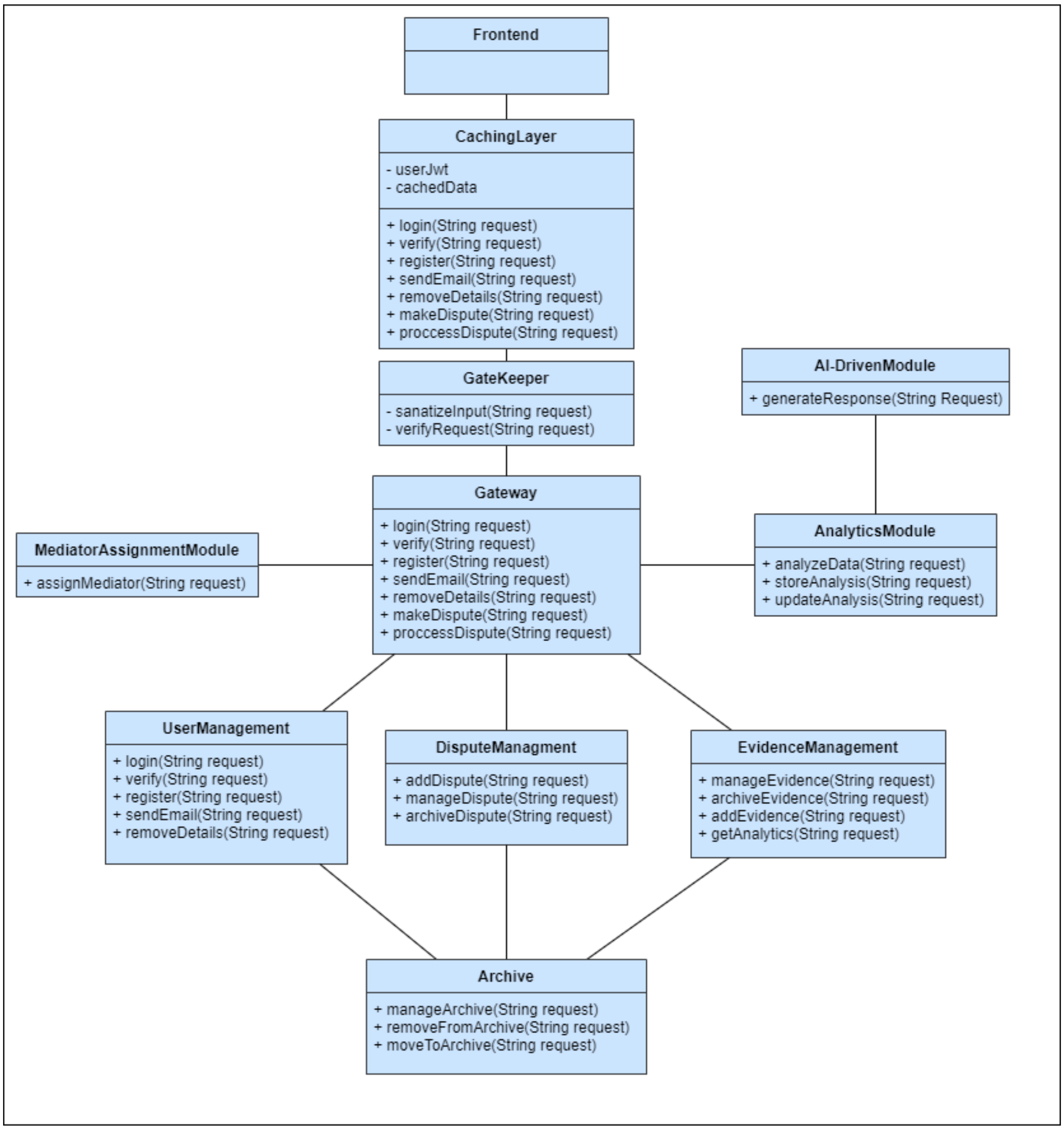
1. Usability

Architectural Diagram



image

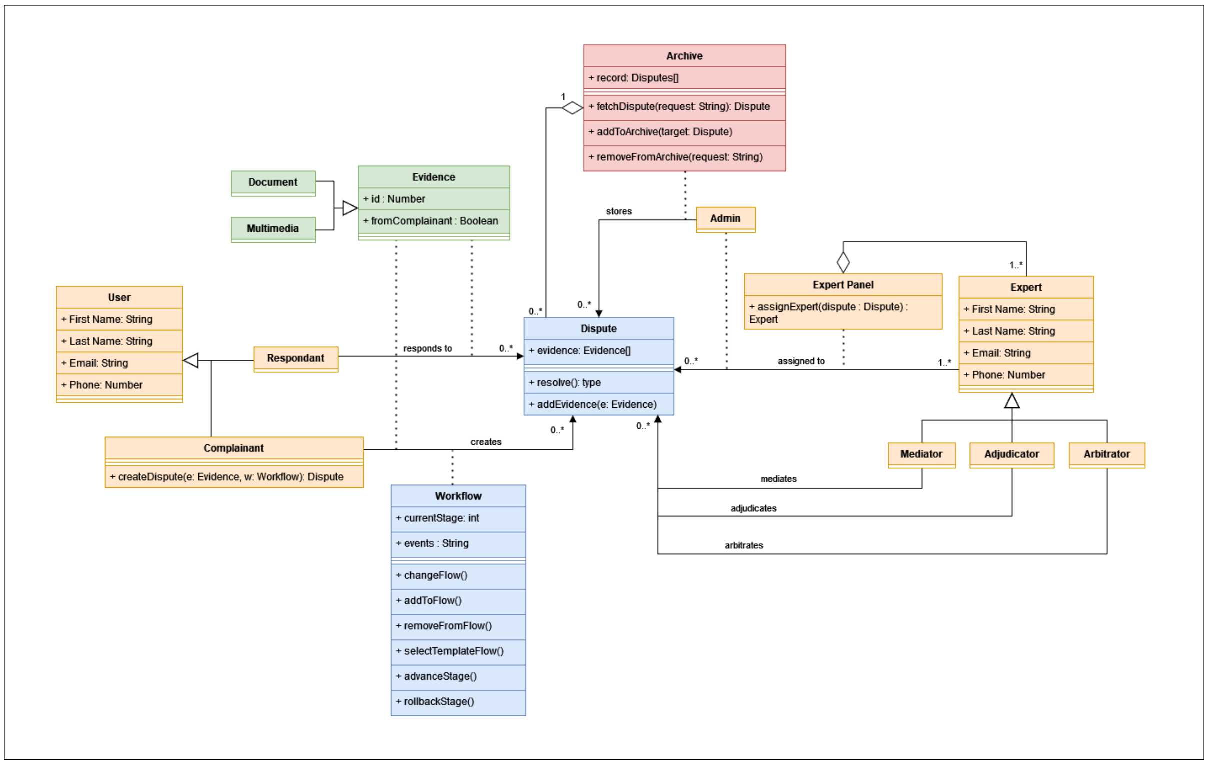
Class Diagram


Frontend
CachingLayer
- userJwt
- cachedData
+ login(String request)
+ verify(String request)
+ register(String request)
+ sendEmail(String request)
+ removeDetails(String request)
+ makeDispute(String request)
+ proccessDispute(String request)
GateKeeper
- sanatizeInput(String request)
- verifyRequest(String request)
AI-DrivenModule
+ generateResponse(String Request)
Gateway
+ login(String request)
+ verify(String request)
+ register(String request)
+ sendEmail(String request)
+ removeDetails(String request)
+ makeDispute(String request)
+ proccessDispute(String request)
MediatorAssignmentModule
+ assignMediator(String request)
AnalyticsModule
+ analyzeData(String request)
+ storeAnalysis(String request)
+ updateAnalysis(String request)
UserManagement
+ login(String request)
+ verify(String request)
+ register(String request)
+ sendEmail(String request)
+ removeDetails(String request)
DisputeManagment
+ addDispute(String request)
+ manageDispute(String request)
+ archiveDispute(String request)
EvidenceManagement
+ manageEvidence(String request)
+ archiveEvidence(String request)
+ addEvidence(String request)
+ getAnalytics(String request)
Archive
+ manageArchive(String request)
+ removeFromArchive(String request)
+ moveToArchive(String request)

classDiagram

Version 2

Class Diagram: Ver 2



ClassDiagram ver2

Architectural Design Strategy: Design based on Quality Requirements

Customer Satisfaction and Continuous Delivery

Agile emphasizes delivering valuable software to the customer frequently, ensuring that the product meets their needs. By focusing on quality requirements like reliability, usability, and security, we ensure that each iteration produces a working product that satisfies customer expectations. High-quality software reduces the number of issues and enhances user satisfaction, which is a core principle of agile.

Adaptation to Changing Requirements

Agile methodologies are built to adapt to changing requirements, even late in development. Designing based on quality requirements such as scalability and maintainability ensures that the system can handle evolving demands without extensive rework. This flexibility is crucial in an agile environment where priorities can shift rapidly.

Sustainable Development

Agile advocates for sustainable development, maintaining a constant pace. By prioritizing performance and maintainability, the system remains efficient and easier to enhance or modify, preventing technical debt and burnout among developers. High-maintainability reduces the long-term effort needed for development, aligning with agile’s emphasis on sustainable work practices.

Promoting Technical Excellence

Agile practices promote continuous attention to technical excellence and good design. By focusing on security and reliability, we ensure that the system is robust and dependable, adhering to high standards of technical excellence. Good design based on quality requirements enhances the overall architecture, making the system more resilient and easier to extend.

Collaboration

Agile values collaboration between cross-functional teams and stakeholders. When design is based on clear quality requirements, it provides a transparent framework for decision-making and prioritization. Everyone involved understands the criteria guiding development, facilitating better communication and collaboration.

Incremental Development

Agile development is incremental and iterative, allowing teams to build and refine the system step-by-step. By focusing on scalability and performance, we ensure that each increment can handle increased loads and improved efficiency. This iterative approach helps in continuously enhancing system quality while accommodating growth and new features.

Delivering Working Software

One of the key agile principles is delivering working software at a constant pace. Focusing on usability ensures that each iteration results in a product that is not only functional but also user-friendly. High usability encourages user feedback, which is vital for agile’s iterative improvement process.

Version 3

Architectural Patterns

Event-Driven Pattern

Scalability - Components to react to events asynchronously - Enables the system to scale horizontally by adding more event processors

Performance - Separating the responsibilities of the production and consumption of events - Processing events in real-time - Handle high-throughput scenarios

Reliability - Events can be stored in a persistent event log - Replayable after system failures, ensuring data integrity and system reliability.

Service-Oriented Pattern

Scalability - Services in an SOA can be deployed and scaled independently - The system can grow and manage increased loads effectively

Maintainability - Encapsulating business logic within discrete services - Easier to update, replace, or extend functionalities in isolation

Gatekeeper Pattern

Security - Acts as a security layer that enforces access control policies - Only authorized requests are allowed to reach the internal services

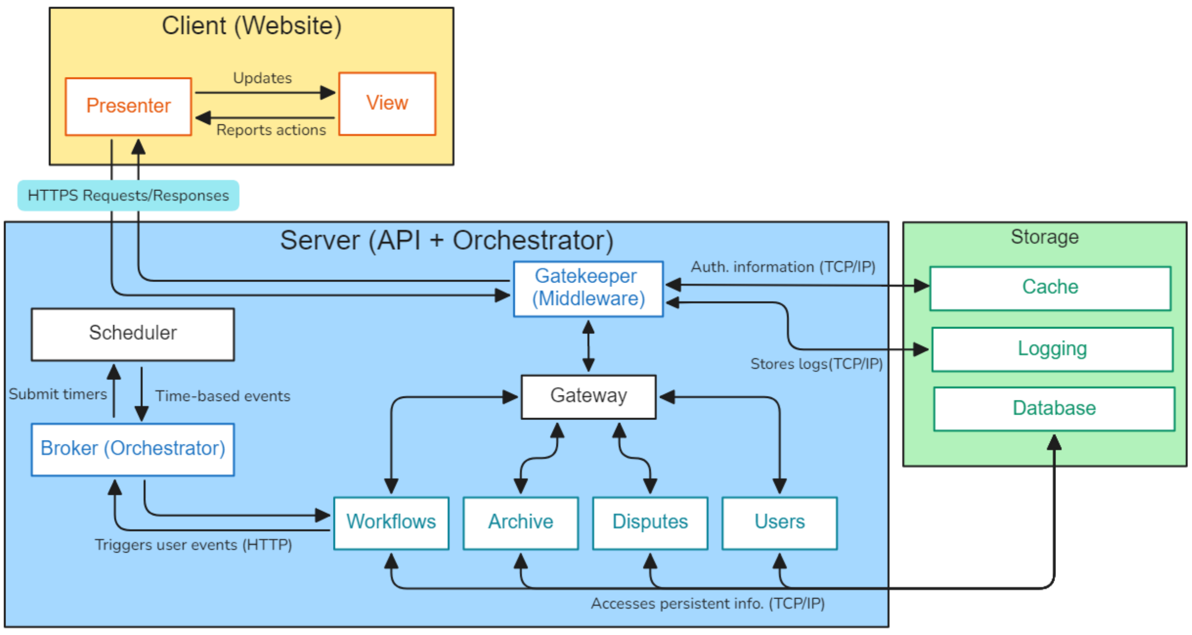
Reliability - Routes requests through a single entry point - Provide load balancing, caching, and failover mechanisms

Client-Server Pattern

Usability - Separates the user interface (client) from the data storage and business logic (server) - More intuitive and responsive user interface - Not bogged down by backend concerns

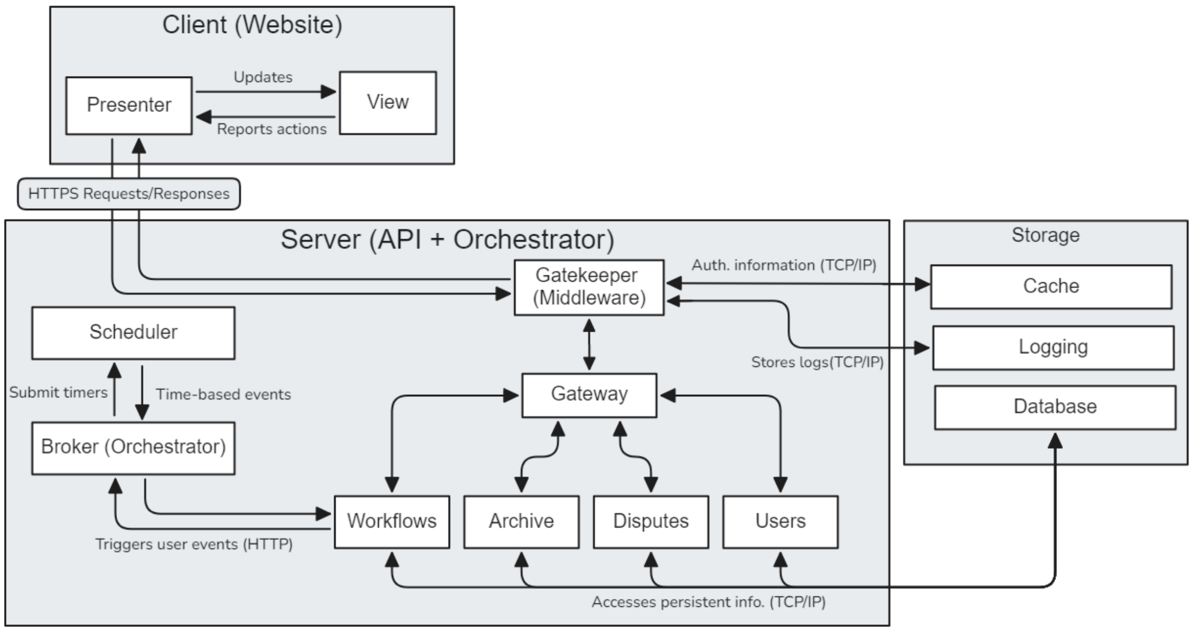
Maintainability - Easily update policies - Ensures that current data privacy and security regulations are met

Architectural Diagram (Color)



20240811_v1

Architectural Diagram (B&W)



archdiagram_20240811vBW