CS 752 Software Architecture and Design Practices

Technical Architecture Deliverable

CS752-2024-11

Project Overview

The Single Window Utility Management Portal simplifies access to utility services like water, gas, and electricity for users and providers. It uses scalable architectures like Microservices (MSA) and Event-Driven Architecture (EDA) to ensure smooth operations. Key modules include Billing and Payment Management for generating invoices, processing payments, and integrating with payment gateways; Usage Monitoring and Analytics for trend analysis and usage recommendations; and Service and Issue Management for handling service requests, issue reporting, and technician updates. The portal connects with external systems like Payment Gateways for secure transactions and Service Provider APIs for real-time data sharing.

Functional Modules

Internal Modules

1. Billing and Payment Management Module

The Billing and Payments Module in the Single Window Utility Management Portal enables seamless management of billing and payment operations for utility services like water, gas, and electricity. It provides functionalities to generate detailed invoices, process customer payments, and integrate with multiple payment gateways, ensuring a smooth and efficient financial transaction process.

Key functionalities are as follows:

a. Invoice Generation:

- Aggregates usage data from service providers via external service API (e.g., water, gas).
- Calculates charges based on service usage, pricing rules, and applicable taxes.
- Generates detailed, itemised invoices with total amounts and breakdowns.

b. Payment Processing:

- Accepts customer payment requests for generated invoices.
- Supports multiple payment methods (e.g., UPI, Credit Card).

- Validates payment requests and routes them to the appropriate payment gateway.

c. Payment Gateways Integration:

- Provides seamless integration with external payment providers.
- Processes transactions securely through gateways like UPI and Credit Card systems.
- Manages responses from payment gateways (success/failure) and updates the payment status.

d. Notification:

- Sends notifications to customers for billing events (e.g., invoice generated) and payment confirmations.
- Provides real-time updates on the status of payments.

2. Usage Monitoring and Analytics Module

The Usage Monitoring and Analytics Module in the Single Window Utility Management Portal provides users with comprehensive insights into their utility consumption patterns for services like water, gas, and electricity. It enables utility usage trend analysis and personalised usage recommendations, empowering users to make informed decisions about resource usage and optimise their consumption.

Key functionalities are as follows:

a. Historical Data and Trends Analysis:

- Retrieves historical consumption data from Billing and Payments module.
- Visualizes usage patterns using charts and graphs to help users identify trends over time.
- Compares current usage with historical data (e.g., this month vs. last month) for better insights.

b. Usage Optimization and Recommendations (future scope):

- Provides personalised suggestions to optimise consumption based on usage patterns (e.g., reduce peak electricity usage).
- Displays environmental impact metrics (e.g., carbon footprint from gas usage) and tips to reduce it.
- Integrates with energy efficiency programs offered by utility providers to encourage sustainable practices.

3. Service and Issue Management Module

The Service and Issue Management Module in the Single Window Utility Management Portal facilitates efficient handling of service requests, complaints, and upgrades for utility services like water, gas, and electricity. It ensures seamless interaction between users, utility providers, and technicians, enhancing service quality and resolution times.

Key functionalities are as follows:

a. Service Request Management:

- Allows users to raise service requests for issues like repairs, upgrades, or new installations.
- Tracks request details, such as request type (e.g., outage, leakage, upgrade), status, and timestamps.
- Provides users with a centralised interface to view, update, or cancel their service requests.

b. Issue Reporting and Resolution:

- Enables users to report utility-related issues (e.g., outages, malfunctions) through the portal.
- Automatically routes issues to the appropriate utility provider or technician based on type and location.
- Tracks resolution progress and provides real-time status updates to users.

c. Technician Assignment and Management:

- Assigns reported issues to technicians based on specialisation, availability, and proximity.
- Tracks technician progress and allows them to update issue resolution details in real time.
- Provides utility providers with insights into technician performance and workload distribution.

d. Alerts and Notifications:

- Sends users timely updates on the status of their service requests (e.g., issue logged, technician en route, issue resolved).
- Notifies technicians and service managers about new assignments or changes to existing tasks.
- Alerts users about planned service disruptions or maintenance schedules.

Note: Notification service is a common functionality to both Billing and Payments module as well as Service and Issue management module. It is a pretty standard service so it will be an abstract class that can be later implemented concretely based on the exact functionality. To avoid over complicating things we have not mentioned it as a separate module.

External Modules

1. Payment Gateways

The Payment Gateways Module facilitates secure and seamless payment processing for utility bills, service charges, and other transactions. It ensures smooth financial operations by integrating with various payment providers and offering a user-friendly experience.

Key functionalities are as follows:

a. Secure Payment Processing:

- Supports multiple payment modes (e.g., UPI, credit/debit cards, net banking, digital wallets).
- Ensures secure transactions with encryption and compliance with payment standards.

2. Service Provider APIs

The Service Provider APIs module acts as the interface between the portal and external utility service providers. It enables seamless data exchange for usage metrics, billing details, and issue handling, ensuring an integrated experience for both users and service providers.

Key functionalities are as follows:

a. Data Integration for Utility Services:

- Provides APIs to fetch user-specific billing and consumption data from service providers.

a. Issue and Service Request Management:

- Supports the creation, update, and status tracking of service requests or complaints with external service providers.

- Allows service providers to send updates regarding resolution status or service interruptions directly to the portal.

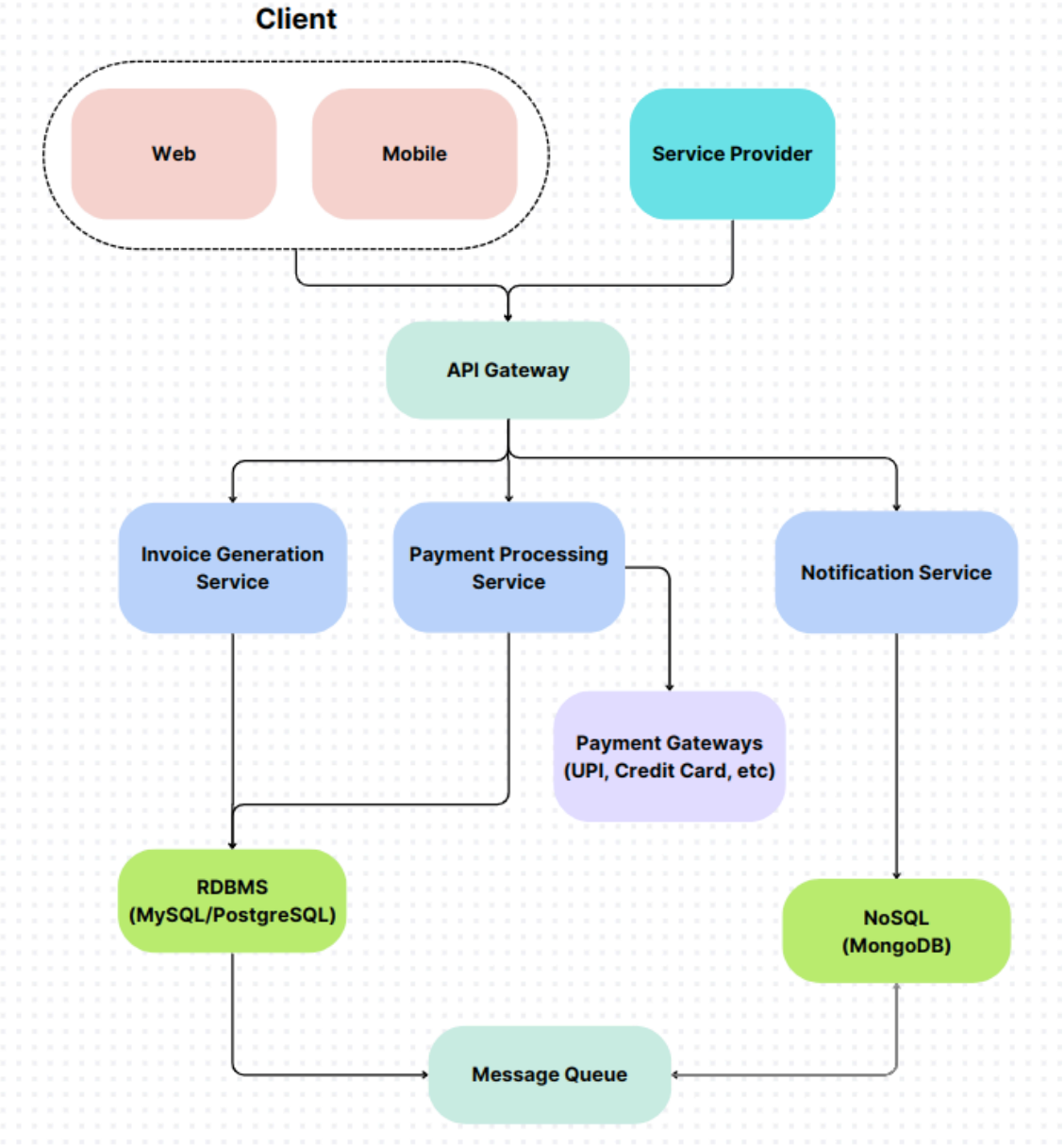
Dependencies

Module 1	Module 2	High Level Input	High Level Output
Billing & Payments	Payment Gateways	Request a payment session	Acknowledgement of payment status (success of failure)
Billing & Payments	Usage Monitoring	Payment history & invoices	NA
Service Provider API	Billing & Payments	Citizen Billing Info	Payment Status
Service & Issue Mgmt	Service Provider API	Service and Issue Requests	Acknowledgement of corresponding requests
Service & Issue Mgmt	Billing & Payments	Citizen Billing info	Payment Confirmation
Service & Issue Mgmt	Usage Monitoring	Service Data	NA

Software Applications

Billing and Payments Application

The **Billing and Payments Module** is broadly designed using a **Microservice** architecture which is composed of independent services that work together to handle the entire billing and payment process. There is utilisation of **message queues** for communication between services introduces elements of an **Event-Driven Architecture** within the broader **Microservices** Architecture. In this context, EDA complements MSA by enabling asynchronous communication between services through events.



Explanation of MSA

1. Invoice Generation

- **Input**: The **Invoice Generation Service** receives usage data from external utility providers via APIs.
- **Processing**:
    - Parses the usage data (e.g., gas, water).
    - Applies service-specific billing rules, taxes, and discounts.
    - Generates a detailed invoice containing usage breakdowns and the total amount due.

- **Output**: Stores the invoice in its database and triggers an event (e.g., "Invoice Generated") in the messaging queue.

2. Notification of Invoice

- **Trigger**: The **Notification Service** subscribes to the "Invoice Generated" event.
- **Processing**: Fetches the invoice details and sends notifications (via email/SMS) to customers, informing them about the newly generated invoice.

3. Payment Processing

- **Input**: The customer initiates a payment via the frontend, which sends a payment request to the **Payment Processing Service**.
- **Processing**:
  - Validates the payment request (e.g., amount matches the invoice, payment method is supported).
  - Routes the payment to the appropriate **Payment Gateway Service** (e.g., UPI, Credit Card) based on the chosen method.
  - Receives a success or failure response from the gateway.
- **Output**: Updates the payment status in its database and triggers a "Payment Completed" or "Payment Failed" event in the messaging queue.

4. Notification of Payment

- **Trigger**: The **Notification Service** subscribes to "Payment Completed" and "Payment Failed" events.
- **Processing**: Sends payment confirmation (or failure) notifications to the customer, providing the status and details of the transaction.

Justification of MSA

1. **Modularity and Scalability**:
   - Services (e.g., Invoice, Payment Processing, Notification) are independent, allowing easy scaling based on demand (e.g., scaling Payment Processing during peak times).
   - New features (e.g., adding more payment gateways) can be added without affecting existing services.
2. **Fault Isolation**:
   - Failures in one service (e.g., a down payment gateway) do not impact others, improving reliability.
3. **Technology Independence**:
   - Each service can use the most suitable technology stack (e.g., NoSQL for logs, relational DB for billing).
4. **Easy Integration**:

- Integration with external systems (e.g., utility APIs for billing data, payment gateways) is simplified as each service has a focused purpose.

Overall this was the best fitting architecture compared to MKA or Layered.

Data Management

Relational Database (e.g., PostgreSQL, MySQL):

- Suitable for **Invoice Generation** and **Payment Processing** due to ACID compliance and complex relational queries.
- Entities: `Invoices`, `Customers`, `Payments`.

NoSQL Database (e.g., MongoDB, Cassandra):

- Suitable for **Notification Service** and logging due to high scalability and simple document-based storage.

Each service has its own database which enables independent scaling and management, a trait of MSA architecture

Usage Monitoring and Analytics Application

The **Usage Monitoring and Analytics Module** is designed using an **Event-Driven Architecture (EDA)** to handle historical data analysis and provide optimization recommendations for utility usage. EDA enables the system to process usage data and deliver insights to users by decoupling services and promoting asynchronous communication.

In this architecture, the system leverages **event queues and event mediators** to ensure that each functionality, such as historical data and trend analysis, and usage optimization, operates independently. This design allows for scalable, reactive, and modular development while ensuring that services remain loosely coupled. The events trigger specific workflows, ensuring real-time responsiveness and insightful analytics.

Explanation of EDA

1. **Historical Data and Trends Analysis:**
   - **Trigger:** The **Historical Data Channel** subscribes to usage events.
   - **Processing:**
     - Aggregates monthly usage data into structured historical records for analysis.
     - Generates trend visualizations and comparisons (e.g., this month vs. last month, yearly usage bar chart).
   - **Output: Stores analyzed data** in the relational database and generates trends for user dashboards.

2. **Usage Optimization and Recommendations:**
   - **Input:** Subscribes to **historical data and trends events** published by the **Historical Data Channel**.
   - **Processing:**
     - Analyzes **user consumption patterns** and identifies areas for optimization.

        - Generates personalized suggestions (e.g., reduce peak electricity usage, reduce overall usage in a particular month) and environmental impact metrics.
    - **Output:** Publishes recommendations for user dashboards or as notifications.

Justification of EDA

1. **Loose Coupling:**
    - The functionalities (**Historical Data and Trends Analysis** and **Usage Optimization and Recommendations**) are decoupled from each other and interact asynchronously using events. This ensures modularity and reduces dependencies between components.
2. **Scalability:**
    - Each functionality operates **independently**, allowing individual components to **scale horizontally** based on load (e.g., a spike in usage events does not affect trend analysis or recommendations).
3. **Ease of Maintenance:**
    - **New features or event consumers** can be **added/modified without disrupting or redesigning** the system. For example, integrating a machine learning model for predictive analytics or integrating third-party optimization engines.

Data Management

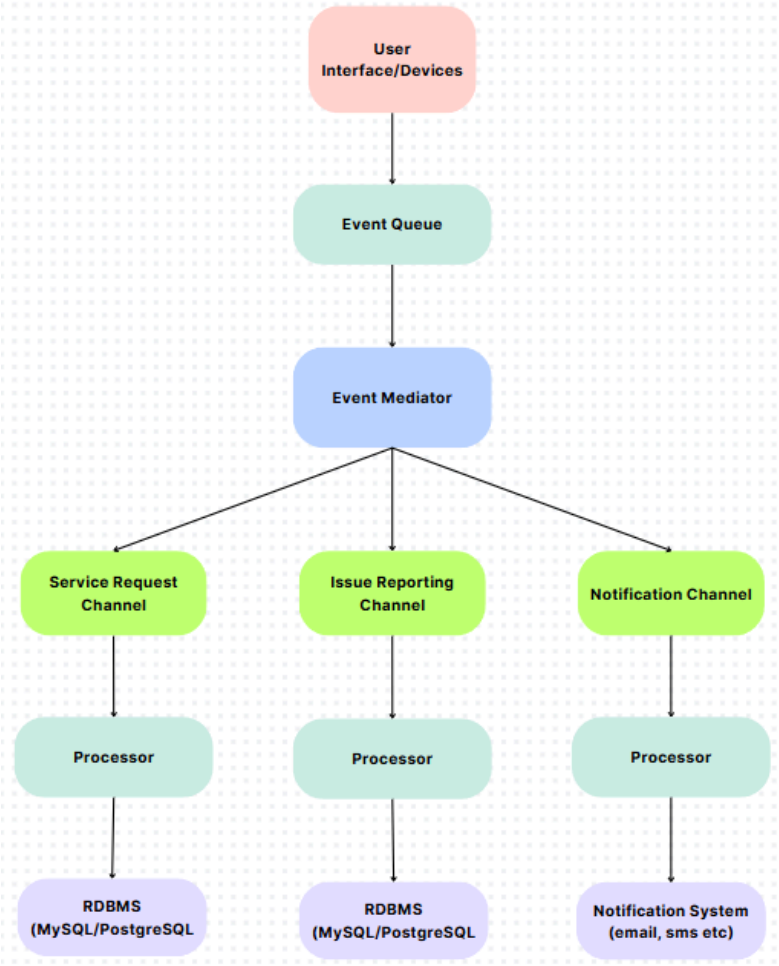
1. **Data Types:**
    - **Event Data:** Monthly usage events (e.g., water usage, electricity consumption).
    - **Aggregated Data:** Yearly usage summaries and comparisons (e.g., "2023: 2100 kWh vs. 2024: 2500 kWh").
    - **Recommendations:** Personalized optimization suggestions and environmental impact metrics.
1. **Data Storage:**
    - **Relational Databases (RDBMS):** Store structured data for historical trends, aggregated usage, and optimization metrics (e.g., PostgreSQL or MySQL).
2. **Data Flow:**
    - Monthly usage data flows into the **Historical Data Channel** for processing.
    - Aggregated data is stored in the relational database.
    - The **Optimization Channel** fetches historical trends from the database, processes the data, and publishes actionable insights back into the system.
3. **Retention and Querying:**
    - Historical data is retained long-term for trend analysis and seasonal optimization.
    - Optimization metrics and patterns are stored temporarily but can be archived for deeper insights.

EDA Components

1. **Event Broker:**
    - A centralized **Event Queue** (e.g., Apache Kafka, RabbitMQ) ensures reliable delivery of events to subscribed services.
    - Serves as the backbone for asynchronous communication, decoupling producers from consumers.
2. **Event Mediator:**
    - Routes events to appropriate channels based on their type:
        - Monthly usage events - **Historical Data and Trend Analysis Channel**.
        - Aggregated data and trends - **Optimization and Recommendations Channel**.
3. **Event Consumers:**
    - **Historical Data Channel:** Consumes usage events, aggregates them into structured data, and generates trend visualizations using aggregated data from the database.
    - **Optimization and Recommendations Channel:** Consumes aggregated data and trends to analyze patterns and generate actionable insights for users.
4. **Processors:**
    - Each channel has dedicated processors for business logic:
        - **Trends Processor:** Aggregates, stores, and visualizes data.
        - **Optimization Processor:** Analyzes trends, identifies areas for improvement, and publishes recommendations.
5. **Databases:**
    - **Relational Database (RDBMS):** Stores structured data (e.g., historical usage trends, optimization metrics) for long-term use and dashboard updates.

Service and Issue Management Application

This module is to be designed using the **Event Driven Architecture (EDA).**



Justification of EDA

1. **Real-Time Processing:**
   - Asynchronous event handling enables instant notifications, escalations, and updates.
2. **Flexibility and Extensibility:**
   - Adding new events or processors is seamless without disrupting the system.
3. **Decoupling:**
   - Producers and consumers operate independently, ensuring modularity and easier maintenance.
4. **Resilience:**
   - Failures in one component don't impact the entire system; events persist in the queue.

Requirement	EDA	Microkernel	Layered	SOA	Monolithic
Real-time	✅	❌ Limited	❌ High	❌ High	❌ Limited

notifications	Asynchronous events enable instant updates		latency	latency	scalability
Decoupling	✅ Producers and consumers are independent	❌ Tight plug-in coupling	❌ Tightly coupled layers	✅ Independent services, but complex orchestration	❌ Tightly coupled
Scalability	✅ Scale processors individually	❌ Limited	❌ Entire layer must scale	✅ Scalable services, but more overhead	❌ Hard to scale
Asynchronous processing	✅ Built-in	❌ Not supported	❌ Not supported	❌ API-dependent	❌ Not supported
Ease of maintenance and extensibility	✅ Add new events or processors seamlessly	❌ Hard to extend core features	❌ Hard to isolate changes	✅ Services can be added, but more complex to manage	❌ Hard to maintain

EDA Components

1. **Event Queue**
   Acts as the central data structure that stores all events being fired.
   Examples of events queued:
   - `ServiceRequestCreated` (user creates a service request).
   - `IssueReported` (user reports an issue).
   - `TechnicianAssigned` (system assigns a technician).
   - `IssueResolved` (technician resolves the issue).
2. **Event Mediator**
   The mediator processes incoming events from the queue and fires additional events to the appropriate channels.

   Responsibilities:

- Validation: Validates the event payloads (e.g., ensure ServiceRequestCreated has all required data like user ID and request type).
- Orchestration: Based on the event type, triggers follow-up events for specific channels.

Example Actions:
Upon receiving `ServiceRequestCreated`, the mediator might:

1. Route it to the Service Request Channel for processing.
2. Fire a NotifyUser event to the Notification Channel.

3. **Event Channels**
   Dedicated pipelines for handling specific types of events. These channels decouple processing and allow parallel handling of different event types.

   Key Event Channels:
   - Service Request Channel: Processes events related to service requests (e.g., create, update, cancel).
   - Issue Reporting Channel: Handles issue-related events like IssueReported or IssueResolved.
   - Notification Channel: Sends notifications and alerts to users and technicians.

4. **Event Processors**
   The processors listen to specific event channels and perform necessary actions.

   Key Event Processors:
   - Service Request Processor:
     Handles events like ServiceRequestCreated and updates the service database. Also assigns the request to the appropriate technician or team based on priority and location.
   - Issue Resolution Processor:
     Processes the IssueReported and IssueResolved events to update the issue status in the database. Also verifies resolution details provided by technicians.
   - Notification Processor:
     Listens to events like NotifyUser or NotifyTechnician and sends messages via email, SMS, or in-app notifications.

Event Flow Example

User Creates a Service Request

1. **Event Queue:**
   - The `ServiceRequestCreated` event is published to the queue by the User Interface.
2. **Event Mediator:**

    - Validates and enriches the event with additional details (e.g., user location).
    - Fires the event to the **Service Request Channel** and **Notification Channel**.
3. **Service Request Processor:**
    - Creates a new record in the service request database and assigns it a status of "Pending."
4. **Notification Processor:**
    - Sends a confirmation notification to the user.

Technician Resolves an Issue

1. **Event Queue:**
    - The technician's app emits an `IssueResolved` event to the queue.
2. **Event Mediator:**
    - Routes the event to the **Issue Reporting Channel** and the **Notification Channel**.
3. **Issue Resolution Processor:**
    - Updates the issue status in the database to "Resolved."
4. **Notification Processor:**
    - Notifies the user that their issue has been resolved.

Data Management

Relational vs Non-Relational:
A hybrid approach using both relational and NoSQL databases:

- Relational DB for core structured data (e.g., service issues, user profiles). Relational DBs are suitable to manage relationships between users, service requests, and resolutions.
- NoSQL DB for unstructured or high-velocity data (e.g., event logs, notifications, or attachments).

Local vs Shared Database:
A local Database would suffice for storage of event data of individual processors.

Internal Integrations

The following interactions take place among the internal modules:

1. Billing and Payments Module ↔ Usage Monitoring Module:

Strategy: **Function Interface**

Description:
The Usage Monitoring Module retrieves payment history and invoice data from the Billing

and Payments Module via synchronous function calls, such as RESTful API requests. These calls allow the Usage Monitoring Module to fetch user-specific details, including payment dates, amounts, and outstanding balances, in real time. The retrieved data is used to enhance consumption analysis and provide context for usage trends.

Justification:

- **Real-Time Access:** Function interfaces enable on-demand retrieval of up-to-date payment and invoice data, ensuring the Usage Monitoring Module reflects accurate information for user dashboards.
- **Consistency:** The Billing and Payments Module remains the single source of truth for financial data, reducing the risk of data inconsistency.
- **Low Latency Needs:** Since payment data is accessed only when needed (e.g., for display on user dashboards), synchronous calls are sufficient without requiring constant data synchronization or high-frequency updates.

2. Billing and Payments Module ↔ Service and Issue Management Module:

Strategy: **Messaging Interface**

Description:
The Service and Issue Management module sends billing-related data, such as technician charges, to the Billing and Payments module using asynchronous messaging. Each service-related event (e.g., `ServiceChargeAdded`, `InvoiceGenerated`) is published to a message broker, where it is queued and consumed by the Billing and Payments module for further processing, such as updating invoices or initiating payment workflows.

Justification:

- **Asynchronous Communication:** Messaging allows the Service and Issue Management module to send billing data without waiting for immediate confirmation, ensuring non-blocking operations.
- **Decoupling:** The two modules remain independent, making it easier to update or scale either module without impacting the other.
- **Event-Driven Design Fit:** This approach aligns with an event-driven architecture, where service-related billing events naturally trigger invoice updates or payment workflows asynchronously.

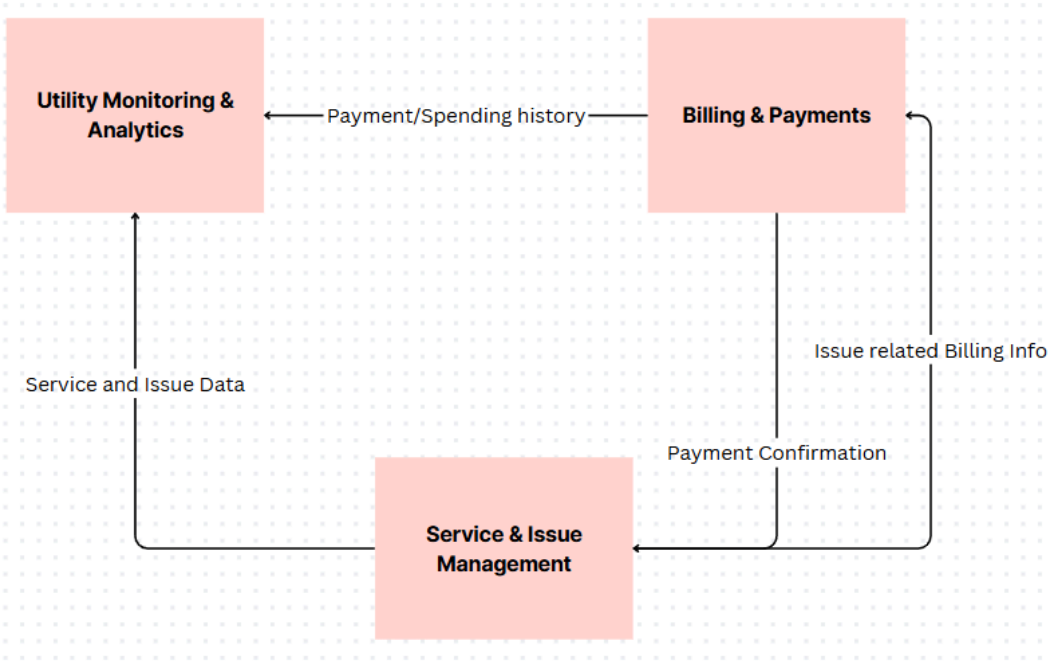
3. Service and Issue Management Module ↔ Usage Monitoring Module:

Strategy: **Function Interface**

Description:
The Usage Monitoring Module retrieves issue resolvement and service usage data from the Service and Issue Management Module via synchronous function calls, such as RESTful API requests. These calls allow the Usage Monitoring Module to fetch user-specific details in real time. The retrieved data is used to enhance consumption analysis and provide context for usage trends.

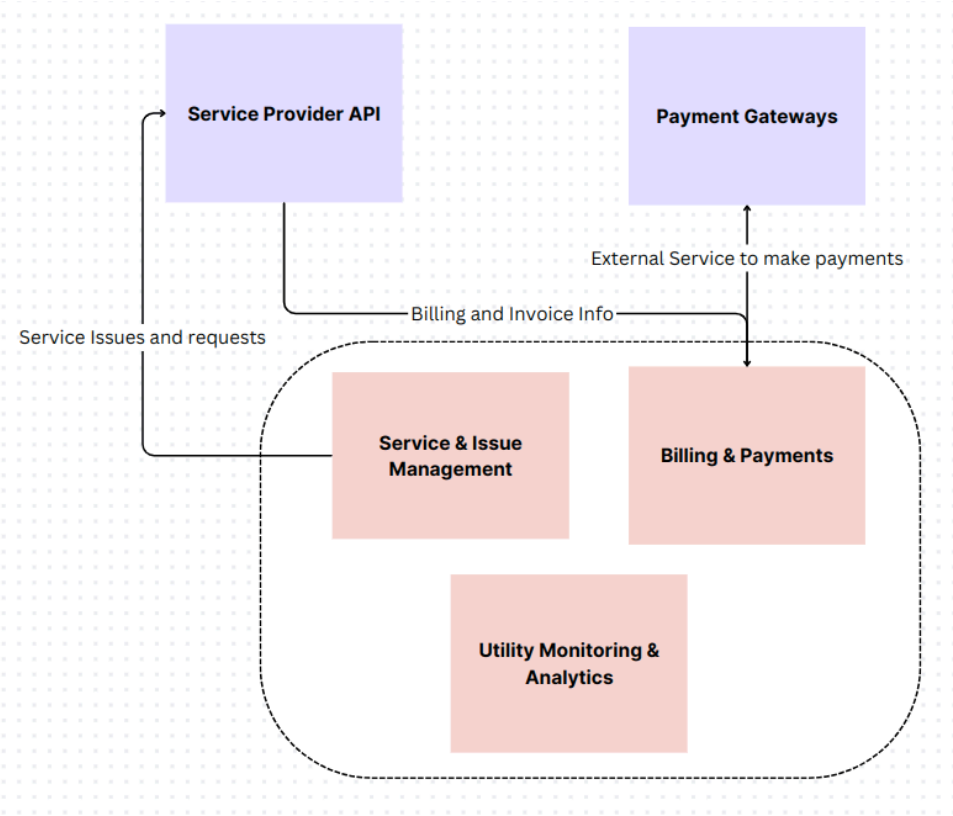
Justification:

- **Real-Time Access:** Function interfaces enable on-demand retrieval of up-to-date issue and service data, ensuring the Usage Monitoring Module reflects accurate information for user dashboards.
- **Consistency:** The Service and Issue Management Module remains the single source of truth for service and issues related data, reducing the risk of data inconsistency.
- **Low Latency Needs:** Since service and issue data is accessed only when needed (e.g., for display on user dashboards), synchronous calls are sufficient without requiring constant data synchronization or high-frequency updates.



External Integrations

The following interactions among internal functional modules and external modules take place



1. Billing and Payments Module [int]↔Payment Gateway [ext]

Strategy: **Web Service**

Description:

The **Billing and Payments Module** communicates with external **Payment Gateways** (e.g., Razorpay, Stripel) via **RESTful web services**. Payment requests are sent to the gateway's API with necessary details such as the invoice amount, customer information, and payment method. The gateway responds with the payment status (success/failure) and a transaction reference.

Justification:

- **Standard Integration**: Most modern payment gateways provide APIs, making web services the natural choice for integration.

- **Real-Time Communication**: Web services enable synchronous, real-time communication for initiating and verifying transactions.
- **Security**: Secure protocols like HTTPS ensure safe data transmission during sensitive operations like payments.
- **Flexibility**: Easily switch or integrate additional gateways by following similar API specifications.

Popular Payment Gateways -: Razorpay, Stripe, PayU

RazorPay is a Payment Gateway that creates a secure pathway between a customer and the business to facilitate payments securely. It involves the authentication of both parties from the banks involved. You can accept payments from customers on your website and mobile apps using the Razorpay Payment Gateway as a business owner.

https://razorpay.com/integrations/

2. Service Provider API [ext] ↔ Billing and Payments Module [int]

Proposed Strategy: **Web Service**

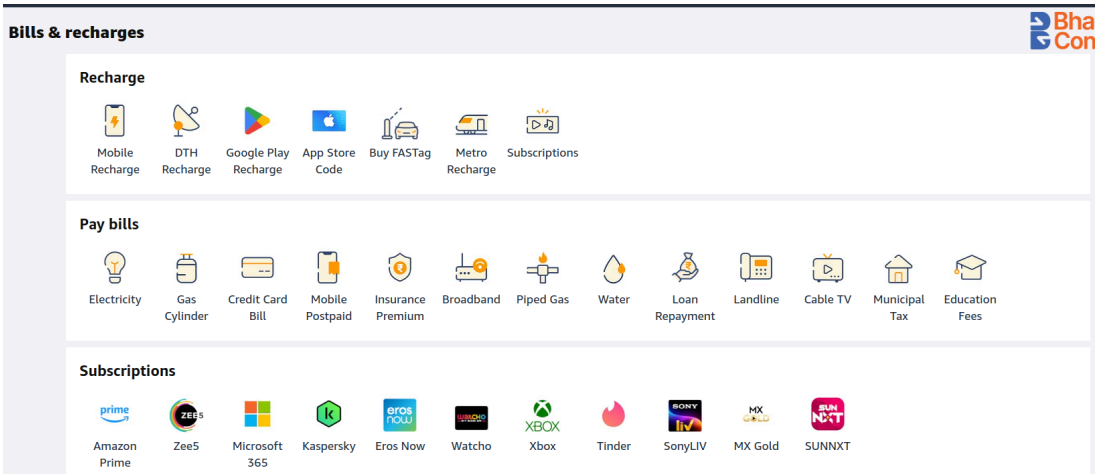
Description:

The **Billing and Payments Module** fetches utility consumption data (e.g., water, gas usage) and pricing details from the **Service Provider API** via RESTful web services. It sends requests with identifiers (e.g., customer ID, period) and receives responses containing usage metrics, rates, and additional details required to generate invoices.

Justification:

- **Flexibility**: Web services allow the Billing and Payments Module to interact with multiple service providers, each potentially having different APIs.
- **Real-Time Data Retrieval**: Ensures up-to-date billing information by fetching data on demand.
- **Standardization**: Most utility providers already support web service-based APIs for data sharing, making this strategy compatible with industry practices.
- **Scalability**: A web service strategy enables the system to integrate with additional service providers as needed, without significant architectural changes.

Real life example - Karnataka One, Amazon



Amazon Platform to pay bills

3. Service Management Module [int] ↔ Service Provider API [ext]

Strategy: **Data Interface**

Description:

The **Service Management Module** allows citizens to create issues (e.g., complaints about utility disruptions) in a shared database. The **Service Providers** access this database to view, update, and resolve these issues through a data interface.

Justification:

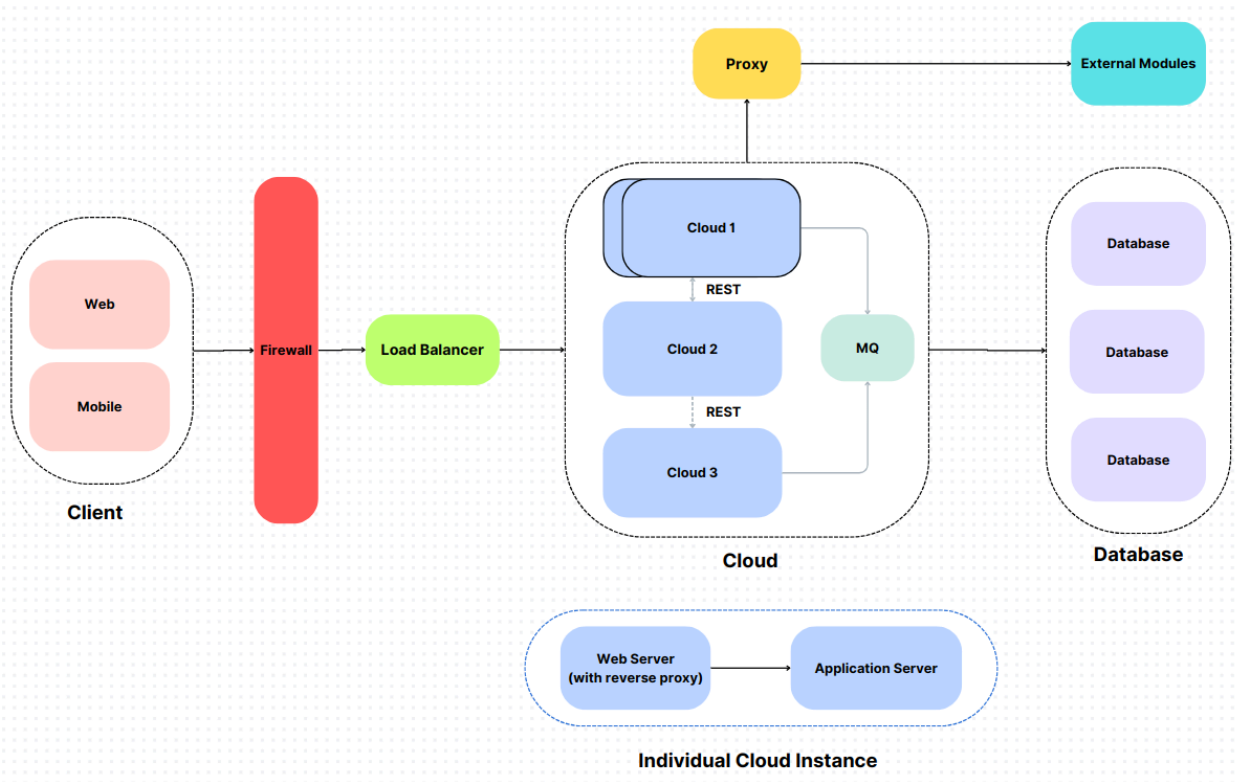
- **Direct Collaboration**: A shared database ensures that both citizens and service providers have direct access to the same data for transparency.
- **Simpler Integration**: Using a shared database avoids the overhead of creating and maintaining a complex API.
- **Real-Time Updates**: Changes made by service providers are instantly visible to citizens, improving user experience.
- **Cost-Effectiveness**: Database-driven communication reduces the need for complex middleware or custom APIs.

Deployment Architecture

Component	Hardware	Software
	- CloudInstance-1: 8 vCPUs, 32 GB RAM, 2 TB SSD, auto-scaling for high availability	- Billing and Payments Application (Spring Boot)

Compute	- CloudInstance-2: 4 vCPUs, 16 GB RAM, 1 TB SSD, configured for event-streaming	- Service and Issue Management Application (Node.js)
	- CloudInstance-3: 4 vCPUs, 16 GB RAM, 1 TB SSD, used for analytics and monitoring	- Usage Monitoring and Analytics Application (Python/Django)
Storage	- CloudDatabase-1: RDBMS like PostGres and NoSQL DB like redis	- PostgreSQL Database (for Billing and Payments), Redis
	- Storage: NAS/SAN system with 50 TB capacity for backup and archival	-
Network	- Load Balancer: Cloud-managed or hardware load balancer for request traffic management	- Nginx (Web Server/Load Balancing)
	- Firewall: Enterprise-grade firewall with IDS/IPS for secure communication	- Elastic Stack (ELK) for log monitoring

Deployment Architecture Diagram



Physical Hardware Components

Details of physical hardware components required for the solution:

S. No.	Component Name	Component Description	Software Applications
1	Storage	Dedicated NAS/SAN system with 50 TB capacity for on-premise backup and archival.	Backup for Billing and Payments, Usage Monitoring, and Service Data.
2	End User Devices	Mobile phones, tablets, and desktop computers for accessing the portal. These include standard web browsers and mobile OS platforms.	User-facing portal (web/mobile app).
3	Firewall	Enterprise-grade network firewall with intrusion detection/prevention system (IDS/IPS). Ensures secure communication and prevents unauthorized access.	Network Security.
4	Load Balancer	Enterprise-grade hardware load balancer or a cloud-managed load balancer. Balances user traffic across instances and ensures high availability.	Managing request traffic

Virtual Servers/Cloud Components

Details of cloud services required for the solution:

S. No.	Component Name	Component Description	Software Applications
1	CloudInstance-1[2]	Cloud VM with 8 vCPUs, 32 GB RAM, 2 TB SSD storage. Configured with auto-scaling for high availability.	Billing and Payments Application (Springboot)
2	CloudInstance-2	Cloud VM with 4 vCPUs, 16 GB RAM, 1 TB SSD storage. Configured for event-streaming and issue management.	Service and Issue Management Application (Node.js).
3	CloudInstance-3	Cloud VM with 4 vCPUs, 16 GB RAM, 1 TB SSD storage. Used for historical data analytics and monitoring.	Usage Monitoring and Analytics Application (Python/Django).
4	CloudMessageBroker-1	Managed message broker service (e.g., AWS SQS, Google Pub/Sub, or RabbitMQ). Handles asynchronous messaging between modules.	Messaging Interface for EDA architecture.
5	CloudDatabase-1	Managed relational database service (e.g., AWS RDS, Google Cloud SQL) with PostgreSQL/Aurora/Redis. Includes automated backups and replication.	Non relational and Relational database for Billing and Payments and Service and Issue Management.

Infrastructure Software

Additional software required for the solution:

S. No.	Infrastructure Software Name	Purpose	Type	Component Description	Cloud Component Name
1	PostgreSQL\Aurora	Relational database for core module data.	Open Source	Managed PostgreSQL database for scalability and fault tolerance.	CloudDatabase-1
2	Rabbit MQ	Message broker for asynchronous communication in event-driven architecture.	Open Source	Managed RabbitMQ service for reliable message queuing and routing.	CloudInstance-2, CloudInstance-3, CloudMessageBroker-1
3	Nginx	Web Server/Load balancing	Open Source	Reverse proxy for web requests.	CloudInstance-1, CloudInstance-2
4	Elastic Stack (ELK)	Log monitoring.	Open Source	Elasticsearch, Logstash, and Kibana for centralized logging.	CloudInstance-3

5	Ubuntu/CentOS/Red Hat Linux	Operating System.	Open Source/Proprietary	Linux OS for running software applications.	CloudInstance-1, CloudInstance-2, CloudInstance-3
6	Redis	NoSQL DB	Proprietary	Stores unorganised data	CloudInstance-1, CloudInstance-2

Quality Attributes

How the cloud-based deployment architecture ensures quality for each software application:

Billing and Payments Application

- **Availability:** Hosted on CloudInstance-1 with auto-scaling and a managed PostgreSQL/Aurora database on CloudDatabase-1.
- **Security:** Cloud services include built-in encryption, firewall configurations, and role-based access control (RBAC).
- **Performance:** High-performance cloud VMs and optimized database queries ensure real-time invoice generation.

Usage Monitoring and Analytics Application

- **Scalability:** Hosted on CloudInstance-3 with elastic scaling to handle large-scale analytics.
- **Real-Time Processing:** Event-streaming via Kafka ensures timely anomaly detection and alerts.
- **Reliability:** Managed cloud services ensure data redundancy and disaster recovery.

Service and Issue Management Application

- **Availability:** Hosted on CloudInstance- with auto-scaling and a managed PostgreSQL/Aurora database on CloudDatabase-1.
- **Scalability:** Hosted on CloudInstance-2 with elastic scaling to handle large-scale analytics.
- **Monitorability:** ELK stack tracks system performance and logs in real time.

This cloud-based deployment architecture ensures scalability, high availability, and resilience while reducing infrastructure management overhead.