SYSOP SQUAD

Table Of Contents **Page Number**

KATA DESCRIPTION	2
ARCHITECTURALLY INTERESTING BITS	2
PROPOSED SOLUTION	3
SOLUTIONS CONSIDERED AND DISCARDED	9
REASONS FOR CONSIDERING	9
REASONS FOR DISCARDING	9
ARCHITECTURAL STYLE SELECT AND REASONS	9
REASONS FOR SELECTION	9
DEPLOYMENT ENVIRONMENT	10
FITNESS FUNCTIONS	11
NORMAL WARNING AND ALARM STATES	12
DATA GROWTH AND IMPACTS	13
REQUIREMENTS FROM PRODUCT OWNER	13
IMPACTS	13
BUSINESS CONTINUITY NEEDS	14
ADR	15

KATA DESCRIPTION

The Sysop Squad's trouble-ticket system is a nationwide platform for managing IT support requests across thousands of customers and hundreds of consultants and store staff. The system allows customers, call-center receptionists, and store staff to log trouble tickets, which are automatically routed to consultants based on location, skill set, and availability. Consultants can manage tickets and track work on mobile devices, while customers provide feedback after service.

ARCHITECTURALLY INTERESTING BITS

1. Scalability and Load Management:

A. The system must handle **thousands of daily users** (call-center receptionists, consultants, store staff, and customers) while supporting **real-time ticket creation and routing**.
B. **Graceful degradation** under heavy load is crucial. The system should shed non-critical features or delay non-essential tasks under load while ensuring core functionality, such as **ticket creation and routing**, remains operational.

2. Service Availability and Uptime:

A. The system is expected to have **99.5% availability**, focusing primarily on **core working hours**.
B. The system should be able to recover quickly from failures, supported by **disaster recovery** strategies like backup systems and failover regions.

3. Routing Logic:

A. Ticket routing is based on **location, consultant availability, skillset, and additional criteria** to ensure the right consultant is assigned..

4. Data Management:

A. Data size (KB to MB for each ticket) and the need to store it for **5 years**.
B. **Compliance** with data privacy regulations in both the USA and Canada (e.g., GDPR, PIPEDA) is necessary.

5. Compatibility

A. The system must be fully compatible with **iOS and Android** devices, enabling consultants to complete all tasks via **mobile devices**.

PROPOSED SOLUTION

We have designed a solution leveraging **microservices** and **event-driven architecture**. This approach ensures scalability, modularity, and flexibility, allowing independent services to communicate asynchronously via events. The system components are connected through an event broker to handle user interactions and background processes efficiently. Each microservice, such as the **Ticket Management Service**, **Survey and Notification Service**, and **Search Service**, focuses on specific functionalities, ensuring a decoupled structure that simplifies maintenance and future enhancements. With **caching layers** and **load management** integrated, the architecture can handle high traffic, ensuring low latency and seamless user experience.

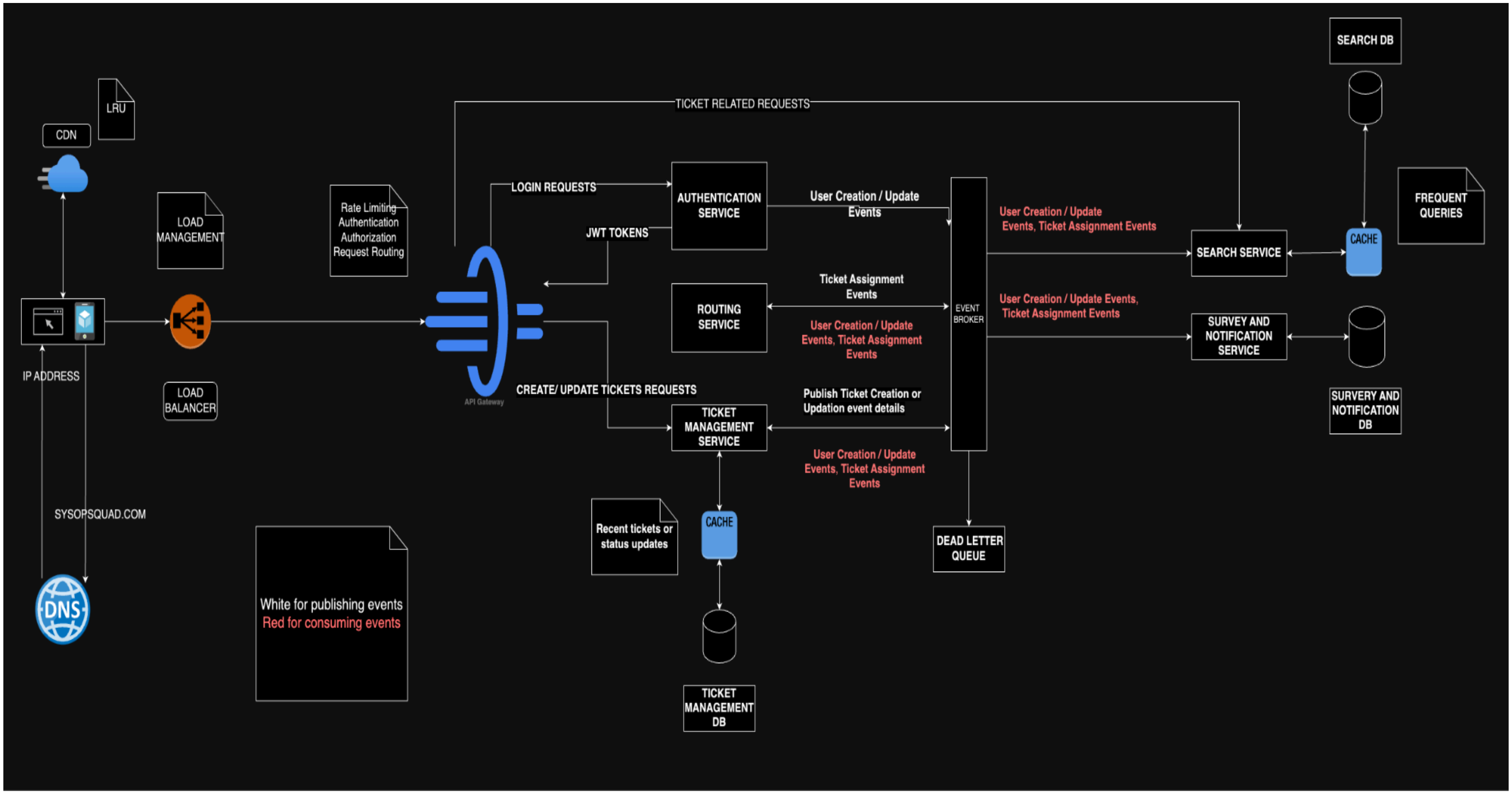
SEARCH DB
LRU
CDN
TICKET RELATED REQUESTS
LOGIN REQUESTS
AUTHENTICATION SERVICE
User Creation / Update Events
User Creation / Update Events, Ticket Assignment Events
FREQUENT QUERIES
LOAD MANAGEMENT
Rate Limiting Authentication Authorization Request Routing
JWT TOKENS
SEARCH SERVICE
CACHE
Ticket Assignment Events
ROUTING SERVICE
EVENT BROKER
User Creation / Update Events, Ticket Assignment Events
User Creation / Update Events, Ticket Assignment Events
SURVEY AND NOTIFICATION SERVICE
IP ADDRESS
LOAD BALANCER
API Gateway
CREATE/ UPDATE TICKETS REQUESTS
Publish Ticket Creation or Updation event details
TICKET MANAGEMENT SERVICE
SURVERY AND NOTIFICATION DB
User Creation / Update Events, Ticket Assignment Events
SYSOPSQUAD.COM
Recent tickets or status updates
CACHE
DEAD LETTER QUEUE
DNS
White for publishing events
Red for consuming events
TICKET MANAGEMENT DB

Figure : **PROPOSED SOLUTION**

For better Quality image : Link

ELABORATION OF THE SOLUTION

1. Content Delivery Network (CDN):
We have decided to go with CDN to **reduce the latency** and **offload traffic** from the origin server.

We are going to use LRU cache which evicts the recently used items.

Diagram for better visualization : Link

2. Load Balancer (LB):

We have decided on using load balancers for **intelligent routing** by ensuring **low latency**, **efficient load management** and **fault tolerance**.

Diagram for better visualization : Link

3. API GATEWAY

API gateway acts as a single point entry for our backend services.

1. Authentication & Authorization, Verifies JWT tokens from users for secure access.
2. Rate Limiting, Controls the number of requests to prevent overload.
3. Request Routing, Forwards requests to appropriate microservices (like Search or Ticket Management).

Diagram for better visualization : Link

4. MICROSERVICES AND EVENT DRIVEN ARCHITECTURE (EDA)

We have opted for an EDA architecture because it makes our system scalable, flexible, and resilient. It allows services to communicate asynchronously and respond to events in real-time, ensuring smooth performance and fault tolerance.

Here, services communicate asynchronously through an Event Broker (SQS). This architecture allows publishers to send events (like "Ticket Created" or "User Updated") to the Event Broker, which distributes them to subscribers/consumers. The consumers perform tasks based on these events, such as assigning tickets, updating user data, and sending notifications.

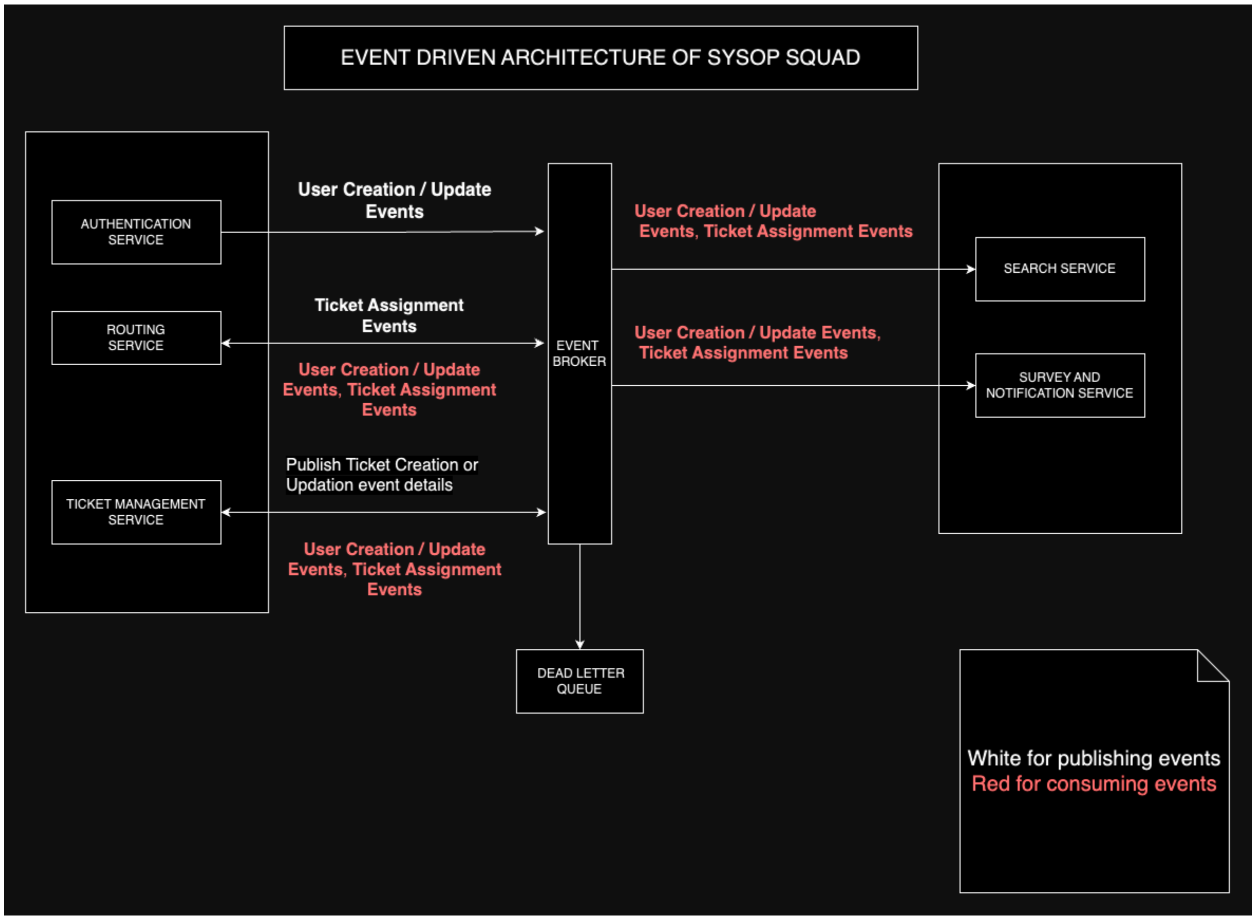
EVENT DRIVEN ARCHITECTURE OF SYSOP SQUAD
AUTHENTICATION SERVICE
ROUTING SERVICE
TICKET MANAGEMENT SERVICE
User Creation / Update Events
Ticket Assignment Events
User Creation / Update Events, Ticket Assignment Events
Publish Ticket Creation or Updation event details
User Creation / Update Events, Ticket Assignment Events
EVENT BROKER
User Creation / Update Events, Ticket Assignment Events
User Creation / Update Events, Ticket Assignment Events
SEARCH SERVICE
SURVEY AND NOTIFICATION SERVICE
DEAD LETTER QUEUE
White for publishing events
Red for consuming events

Figure : Event Driven Architecture of Sysop Squad

DEAD LETTER QUEUEE (DLQ)

It stores failed or unprocessed messages from the main queue, ensuring that no data is lost. It allows for **troubleshooting** and **retries** without blocking the system's workflow.

SERVICES OVERVIEW

Service	Requests	Publishes	Consumes	Database	Cache (External)
Ticket Management System Service (Diagram: Link)	Create/Update Ticket Requests from API Gateway	Ticket Created/Updated Events	User Creation/Update Events Ticket Assignment Events	Ticket Management DB, Relational database to store structured ticket data (e.g., ticket ID, status, timestamps).	Redis for Recent Tickets
Search Service (Diagram: Link)	Queries for Ticket or User Details from API Gateway	None	User Creation/Update Events Ticket Created/Updated Events	Search DB (Elasticsearch as it is ideal for full text search and indexing)	Redis for Frequent Queries
Routing Service (Diagram: Link)	None	Ticket Assignment Events	User Creation/Update Events Ticket Assignment Events	None (consultant data is kept in memory) (when restart fetch it from cognito)	None
Authentication (AWS Cognito)	Login Requests from API Gateway	User Creation/Update Events	None	Managed by AWS	None
Survey & Notification Service (Diagram: Link)	None	None	Ticket Assignment Events User Creation/Update Events	Survey and Notification DB(NOSQL db with fast read and writes)	None
Monitoring & Analytics (CloudWatch)	Collects metrics, logs, and traces	None	Receives logs/metrics from all services	Managed by AWS (No DB Setup Required)	NA

PS: Links for the working of each service have been added for better understanding

COMMUNICATION PROTOCOLS

Communication	**Protocol**	**Reason for opting**
Client ↔ API Gateway	HTTPS	Secure, Widely supported ,Standard
API Gateway ↔ Services	HTTPS	Simple, Standard
Services ↔ Event Broker	HTTPS	SNS uses HTTPS and AWS SDKs for reliable message publishing and delivery.
Services ↔ Databases	JDBC/SQL	Standard, reliable access to relational or NoSQL databases.
Services ↔ Cache	TCP	Fast, reliable in-memory access to cached data.
Authentication (Client ↔ Cognito)	OAuth2 over HTTPS	Standard protocol for secure user authentication and token-based access.

SOLUTIONS CONSIDERED AND DISCARDED

REASONS FOR CONSIDERING

In Microkernel architecture we thought we can offer **modularity and extensibility by isolating the core functions (kernel) from the extended features**. We thought that the core kernel would be handling the assignment and communication between the tasks and the tasks are done by separate modules that interact with the kernel.

REASONS FOR DISCARDING

1. Here the kernel manages all the communication between the modules and handles core services such as memory management, process scheduling and IPC. So the **more modules you add the more the kernel needs to bear** leading to performance degradation.

2. Since the kernel handles the core tasks and must interact with all other modules, **scaling becomes more difficult as traffic grows**.

3. Lots of context switching between the kernel and various modules. Each module relies on the kernel to manage communication, which can introduce **latency and overhead.**

4. The **kernel itself is a single point** of failure.

ARCHITECTURAL STYLE SELECT AND REASONS

REASONS FOR SELECTION

1. **Decoupling of services**, the producers and consumers are decoupled and they don't need to know about each other. This not only improves modularity but scalability.

2. Events can be processed **asynchronously,** allowing systems to continue performing other tasks while waiting for events to be handled. This helps in real time updates without blocking system performance.

3. Services can **scale independently based on the workload**.

4. Better **Resilience,** because even if one service goes down, it doesn't block the rest of the system.

DEPLOYMENT ENVIRONMENT

We have decided on choosing Amazon Web Services as the cloud provider. Because,

1**. Cheaper Costs** compared to its competitors and **broad services range**.

2. **More Data Centers and availability zones**.

3**. Proven track record** and also it has lesser number of down times compared to gcp or azure.

Component	**AWS Resources**	**Purpose**
CDN	Amazon CloudFront	Distributes Static Content (low latency)
Load Balancer	AWS Elastic Load Balancing (ELB)	Distributes incoming traffic across multiple instances
API Gateway	Amazon API Gateway	Provides a secure and scalable entry point for all requests to backend services.
Event Broker	Amazon SNS + Amazon SQS	SNS for pub/sub events and SQS for reliable queues.
TMS	Amazon RDS	Stores structured ticket data
Search Service	Amazon OpenSearch (Elasticsearch)	Optimized for search queries over tickets and user data with fast indexing.
Routing Service	In-Memory Storage (EC2/ECS)	Stores consultant details in memory without external storage for fast lookups.
Authentication Service	Authentication Service	Provides user authentication and identity management with OAuth2 support.
Survey & Notification Service	Amazon DynamoDB	NoSQL database for survey responses and user preferences with fast read/write operations.
Monitoring & Analytics	Amazon CloudWatch	Centralized monitoring of logs, metrics, and traces from all services.
Redis Cache	Amazon ElastiCache (Redis)	Provides in-memory caching for frequent ticket and query results to reduce load.
Services Hosting	Amazon Lambda	Hosting services.

FITNESS FUNCTIONS

Fitness Function	Objective	Metric	Validation
Scalability Fitness Function	Ensure the system can handle the expected load (e.g., thousands of users, real-time ticket creation, and routing) without degradation.	Measure the system's response time and throughput under increasing load, simulating peak usage	1. Using load testing tools (like **AWS CloudWatch and AWS Auto Scaling metrics**) to validate that the system can automatically scale up to maintain performance within acceptable thresholds. 2. Set a performance threshold, such as **response time < 300 ms** and **no more than 1% request failures**, to validate scalability.
High Availability Fitness Function	Ensure the system maintains 99.5% availability during **core hours**, with proper disaster recovery.	Measure the uptime of critical services (ticket creation, routing) and the recovery time during failures or region outages.	1. Simulate region failures using **AWS Fault Injection Simulator.** 2. Ensure the system can fail over to a backup region or availability zone within a **recovery time objective (RTO)** of less than **5 minutes**, meeting the 99.5% uptime requirement.

NORMAL WARNING AND ALARM STATES

FITNESS FUNCTION	**NORMAL**	**WARNING**	**ALARM**
DATABASE PERFORMANCE	Query Response Time (<100ms)	Query Response Time (100ms-300ms)	Query Response Time (>300ms)
EVENT DRIVEN ARCHITECTURE	Events are processed in near real time (<1s)	Delays in Event Processing(1-4s)	Major delays in Event Processing(>5s)
DISASTER RECOVERY	Backups within 5 minutes	Delayed failover (5-10 mins)	Backup missing or corrupted with a recovery time >10 mins
COST EFFICIENCY	Monthly Costs within budget (<= 100%)	Costs nearing the upper limit and occasional spikes in the resource manager. (101% to 125%)	Costs exceeding the budget, frequent scaling events, unused resources consuming budget. (>125%)

DATA GROWTH AND IMPACTS

REQUIREMENTS FROM PRODUCT OWNER

DAILY ACTIVE USERS : 100,000

TICKETS PER DAY (10% of users, assumption): 10,000 tickets/day

AVERAGE SIZE OF TICKETS: 5 MB

DATA RETENTION PERIOD: 5 YEARS

<u>ESTIMATE</u>

Daily data growth: 10,000 tickets/day × 5 MB = 50 GB/day

Annual data growth: 50 GB/day × 365 days = 18.25 TB/year

5-year storage: 18.25 TB/year × 5 years = ~91.25 TB total

IMPACTS

Data growth is not a problem for us, as we are moving data to cold storage for closed tickets. With 91.25 TB over 5 years, this approach will efficiently manage data growth and keep costs low, ensuring that data growth won't be an issue. But if we really need to point out problems.

1. Increased Storage Requirements and higher storage costs.
2. With larger datasets, **ensuring compliance with data retention policies** (e.g., 5-year retention) and implementing security measures more complex.

BUSINESS CONTINUITY NEEDS

CONTINUITY NEED	**DESCRIPTION**
High Availability	The system should maintain **99.5% uptime**, especially during core working hours. This requires using **multi-region** and **multi-AZ (Availability Zone)** deployments to ensure that the system can continue operating even if one region or zone goes down.
Disaster Recovery	Implement automated backups and failover mechanisms with an **RTO < 5 minutes** and near real-time **RPO** to minimize downtime and data loss.
Data Redundancy and Backup	Ensure regular backups per AWS schedule and **data replication** across regions to avoid data loss in case of system failures or outages. Additionally, data will be moved from S3 Standard to S3 Glacier after 1 year of inactivity.
Graceful Degradation	Under heavy load or during partial outages, the system should degrade gracefully by **shedding non-critical features** (e.g., notifications or surveys) while ensuring that **core services**, such as ticket creation and routing, remain operational.
Security and Compliance Continuity	Encryption, access control, and adherence to **GDPR and PIPEDA** regulations should be maintained during any disruption.

ADR

LRU over LFU: Simplicity Wins the Race

We chose LRU (Least Recently Used) because it is easier to implement and understand, making it ideal for managing recent ticket access. LRU efficiently handles workloads where recent usage is a good predictor of future access, aligning perfectly with our needs.

Load Balancer Before API Gateway: Resilience First

By placing the load balancer before the API Gateway, we ensure traffic can be routed across multiple regions or availability zones during outages. This setup enhances disaster recovery and prevents regional failures from impacting service availability, maintaining continuity.

User Sync from Cognito: Faster, Smarter, Uncoupled

Rather than querying AWS Cognito directly, we decided to sync user data into our own database via events. This reduces latency, avoids rate limits, and ensures faster performance for services like notifications, all while keeping user data up-to-date.

Routing as a Separate Service: Keep It Simple, Keep It Scalable

We opted to decouple routing logic from the Ticket Management System. Embedding complex routing inside TMS would overload responsibilities and hinder scalability. By separating it, we reduce bottlenecks, simplify maintenance, and allow routing to evolve independently.

Separate Databases for Each Service: Freedom to Scale

We decided that each service will maintain its own database to avoid bottlenecks and ensure scalability and fault tolerance. This approach aligns with our event-driven architecture, enabling services to operate independently and efficiently.

Consultant Data in Memory: Speed Without Complexity

Since only consultant details are needed, we'll store them in memory. This approach ensures low-latency access for quick ticket assignments, with event-driven updates keeping the data current, all while simplifying the architecture.