Designing a comprehensive architecture for a banking audit system involves addressing various aspects from functional and non-functional requirements to detailed design elements like capacity estimation, database schema, API implementations, and scalability considerations. Below is a detailed explanation covering each of these aspects:

### 1. Functional Requirements

Functional requirements define what the system must do to support its users. For a banking audit system:

- \*\*Data Ingestion:\*\* Automatically ingest logs from S3, parse them, and store in MongoDB categorized by log types (Large Withdrawal, Potential Scam/Phishing, User's Usage Pattern).

- \*\*Data Retrieval:\*\* Provide APIs to query logs based on date range, log type, and other relevant filters.

- \*\*Reporting:\*\* Generate daily reports summarizing log activities and trends.

- \*\*Compliance:\*\* Ensure logs are securely stored and accessible only to authorized personnel with proper authentication and authorization mechanisms.

- \*\*Audit Trail:\*\* Maintain an audit trail of all activities related to log data access and modifications.

### 2. Non-Functional Requirements

Non-functional requirements specify the quality attributes of the system:

- \*\*Performance:\*\* Ensure low latency in querying and retrieving log data.

- \*\*Scalability:\*\* Handle increasing volumes of log data and user requests efficiently.

- \*\*Security:\*\* Implement data encryption, secure APIs with authentication (JWT tokens), and role-based access controls (RBAC).

- \*\*Reliability:\*\* Ensure high availability with minimal downtime for critical operations.

- \*\*Maintainability:\*\* Facilitate easy updates and maintenance of the system components.

- \*\*Auditability:\*\* Provide detailed logging and monitoring of system activities for auditing purposes.

### 3. Use Case Diagram

A use case diagram would illustrate the interactions between actors (users) and the system:

- \*\*Actors:\*\* Reporting Team, Compliance Officer

- \*\*Use Cases:\*\*

- Upload Log Files

- Query Logs

- Generate Daily Reports

- View Audit Trail

### 4. Capacity Estimation

Capacity estimation involves calculating the expected load and resources needed:

- \*\*Storage:\*\* Estimate storage requirements based on daily log file sizes and retention policies.

- \*\*Compute:\*\* Calculate Lambda function invocations based on S3 event frequency and processing time.

- \*\*Database:\*\* Estimate database storage based on the volume of log data and query patterns (read/write operations per second).

### 5. Low-Level Design

Low-level design involves detailed component-level design:

- \*\*Lambda Function:\*\* Code to parse log files, validate data, and insert into MongoDB.

- \*\*MongoDB Schema:\*\* Define collections and indexes for efficient querying.

- \*\*API Endpoints:\*\* Detailed implementation of RESTful APIs for log querying and reporting.

### 6. High-Level Design

High-level design focuses on system architecture and integration:

- \*\*AWS Services:\*\* Use S3 for storage, Lambda for serverless processing, CloudWatch for monitoring.

- \*\*Node.js Backend:\*\* Express.js for API development, MongoDB for data storage.

- \*\*Security Architecture:\*\* IAM roles for AWS services, JWT tokens for API authentication.

### 7. Database Design

Database design for MongoDB would involve:

- \*\*Collections:\*\*

- `logs`: Store log entries with fields like `type`, `date`, `message`.

- \*\*Indexes:\*\*

- Index on `type` for fast retrieval by log type.

- Time-to-Live (TTL) index on `date` for automatic data expiration based on retention policy.

### 8. Microservices

Microservices architecture could be implemented by separating components into:

- \*\*Log Ingestion Service:\*\* Handles S3 events and processes log files.

- \*\*Data Service:\*\* Manages MongoDB interactions and data retrieval.

- \*\*API Gateway:\*\* Exposes APIs for external access, integrating with microservices.

### 9. APIs Used in System Design of Banking Audits

APIs used include:

- \*\*AWS SDK:\*\* S3 API for retrieving log files.

- \*\*MongoDB Node.js Driver:\*\* CRUD operations for MongoDB.

- \*\*Express.js:\*\* Define RESTful API endpoints for querying logs.

### 10. API Code Implementation

Example API endpoint (`/logs`) implementation in Express.js:

```javascript

app.get('/logs', async (req, res) => {

const { type, startDate, endDate } = req.query;

const query = {};

if (type) query.type = type;

if (startDate && endDate) {

query.date = { $gte: new Date(startDate), $lte: new Date(endDate) };

}

try {

const collection = db.collection('logs');

const logs = await collection.find(query).toArray();

res.json(logs);

} catch (err) {

console.error(err);

res.status(500).json({ error: 'Internal server error' });

}

});

```

### 11. Scalability

To ensure scalability:

- \*\*AWS Lambda:\*\* Scales automatically based on the number of S3 events and concurrent executions.

- \*\*MongoDB:\*\* Use sharding and replica sets for horizontal scaling and high availability.

- \*\*Load Balancing:\*\* Distribute API requests across multiple instances using AWS Elastic Load Balancer.

- \*\*Caching:\*\* Implement caching strategies (e.g., Redis) for frequently accessed data to reduce database load.

By addressing these aspects comprehensively, the banking audit system can meet its functional requirements, adhere to non-functional requirements, and handle scalability challenges effectively. Adjustments and optimizations can be made based on specific operational needs and ongoing performance monitoring.