SMU | LYLE SCHOOL OF ENGINEERING

Show Me The Money

An Architectural Kata

CS7346

Submitted by Group 10

Monarch Nigam

Balakrishnan Arumugam

Index

Introduction and Overview

Our Project "Show Me the Money" is a given situation where a community college needs a new accounting system. Their current system have become old and is no longer able to fulfil the needs of the institution. The college has over 1000 users having the faculty, administration personnel and some members who are not within the locality that is nationwide and even internationally. They require a sound, effective, and safe system of accounting to assist daily financial transactions needs.

One important requirement for this project is that the system must not be web based. The president of the college personally requested that the solution must not be web-based, might be due to security concerns or performance issues or end-user preference. So the new solution will be created as a desktop application to run directly on the actors desktops. The system must also be cost-saving by not needing expensive licensing fees or excessive maintenance.

The college would also like the system to comply to a standard accounting process a collection of guidelines and procedures that have been established by the finance department. Individual departments, however, must be allowed the flexibility to make slight changes or modifications to more closely accommodate their in-house processes. This is to allow the system to be flexible but to also keep all departments under one umbrella.

The second essential requirement is for the system to maintain a full and detailed audit trail of every fiscal transaction. What is meant by this is that every transaction a payment, expense, or budget modification have to be logged along with the user who made it, the date, time, and what exactly was altered in fact.

Requirements

Following are some Functional Requirements for the project:

Follow the College's Standard Accounting Procedure
The designed system must comply with financial rules and procedures set by the college's finance department. This includes how transactions are done, approved, and entered. All users must work within this to ensure accuracy and compliance with the law.

Let departments customize their own workflows

Every department may have slightly different needs or methods of processing money. For ex, the IT department might use a unique approval procedure compared to the Biology department. The process should be able to accommodate small changes like the addition of an extra step of approval but without breaking the process.

Ensure a Complete and Secure Audit Trail
Every transaction or change conducted in the system should be automatically logged in an audit trail. The records should contain details such as what was done, by whom it was done, when and at what time, and what was altered. This is for auditing, investigation, and accountability.

No Web Applications
Since the college administration does not need a web application, we will make the new accounting system a desktop application. The new system would have to work offline (partially at least) and wouldn't need any user to open a browser. The application has to be working on any major operating systems like Windows and Mac.

Keep Software Costs Low
The college doesn't desire expensive software. This includes the use of open-source software, inexpensive services, or cloud services with reasonable pricing. The system must also be easy to upgrade and maintain, such that the costs of ongoing upkeep are minimal.

Key Assumptions

Following are the Assumptions:

AWS Will Be Utilized for the Cloud Hosting
We assume that the college will be using AWS to host the backend of the system. AWS offers many strong tools for computing, storage, messaging and database management. It also includes good security and backup capabilities.

Desktop Application Shall Be Built Using ElectronJS
After a quick research we found that ElectronJS is a framework that enables us to build desktop applications using HTML, CSS and JavaScript, is best for this scenario. It enables us to develop cross-platform applications that can run on Windows, Mac, and Linux. Although it uses web technologies internally, the application does not run on any web browser, fulfilling the requirement of the college.

Hybrid Database Approach: SQL + NoSQL

We will be using both MySQL (relational database) and MongoDB (NoSQL database). MySQL is appropriate for storing the structured data like user accounts and transactions. Also MongoDB is flexible and appropriate for storing audit logs or department workflows that may vary in data structure.

They Will Possess Internet and Compatible Devices
We assume that the users/Actors will have computers that can run the desktop application and they have good and stable internet connectivity whenever required (to synchronize data with the cloud). However, some restricted offline capability can also be provided in the event of temporary disconnection.

Data Analytics Tool will be used if needed:
Advanced reporting features will be included to assist the finance team in making more informed decisions, the system will be integrated with analytics tools such as QuckSight and/or Databricks. These tools enable one to create visualization like graphs, charts and customised financial report from the data in storage.

System Actors and their Roles

The accounting system will be used by many different types of people in the college. There are different categories of users who will have some roles and responsibilities when using the system as follows:

1. Students
Role: See and control their own financial information, such as tuition fees payment and scholarships. Students can log into the system and view what they owe, payments they've already made, and outstanding balances. They might also receive reminders or make payments right within the app interface.

2. Academic Staff / Faculty
Oversee departmental budgets, approve or deny transactions, and track expenditures. Faculty members in departments require tools to view how much of their budget they've already spent, authorize purchase requests, and monitor their budget. The system must provide them with a clear and easy means of doing so.

3. Administrators Role
Install the system, administer user accounts, and offer technical assistance.
System administrators will update and install the app, set up new user accounts, configure roles, and assist users when things go awry. They'll also be responsible for configuring backup schedules and checking system performance.

4. Finance Department Staff
Input financial information, process transactions, and generate reports.
These personnel are at the core of the system. They record expenses, make journal entries, keep accounts, and prepare financial reports. Their actions ensure the college's finances are recorded uniformly and accurately.

5. Auditors Role
Monitor audit trails and scan for anomalies or fraud. Details: The auditors will ensure the system is being utilized as intended. They will review the audit logs to determine whether the users adhered to procedures and whether there are any unusual activities to be investigated further.

Key Use Cases

1. Secured Login and Access Control
The system will have strong login protocols to make sure only the appropriate users have access to the system. This will include usernames and passwords and the two factor authentication for more senior users like administrators and accountants. Users will only be able to access the parts of the system that are relevant to their function.

2. Accounting Workflow Automation
The system will also assist in automating normal accounting processes such as approval of expenditures, recording of transactions, and transfer of funds. Rather than performing all this manually, the users will follow well-documented procedures, and the system will verify that all the appropriate approvals are in place before it continues.

3. Basic Financial Data Entry
They will be able to enter financial information, like budget revisions or payments into plain text forms. The system will verify for errors, like omitted mandatory fields or improper data types, in an effort to prevent errors.

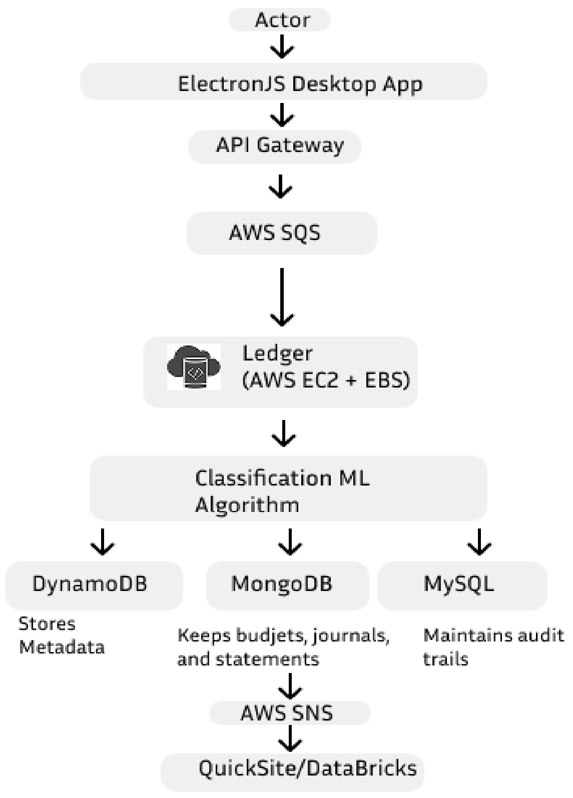
4. Department Budget Management
Each of the departments will have the ability to generate and edit their own budgets. The software will indicate to users how much has been spent and how much remains available. It will even issue warnings as users are approaching overexpenditures, which will keep departments comfortably within their budgets.

5. Continuous Audit Trail Logging
Any activity a user does be it updating a budget or approving a transaction that will be automatically logged in an audit log. The logs are irreversible and will assist in tracking

everything that occurs in the system. The log can be checked by auditors and administrators to ensure all is well.

6. Financial Reporting and Analytics
The system will also generate pre-formatted and tailored financial reports. The reports will display items such as monthly expenditure, annual trends, or unusual activity. This will enable the leadership and financial personnel to make choices from actual data.

Proposed Solution and Architectural Diagram



Our suggested architectural solution tries to meet the complex needs of the community college's accounting system by combining a cloud-native and decentralized solution on Amazon Web Services. The architecture is specifically designed to provide robustness, extensibility, low latency and fault tolerance and adherence to security and auditability standards. One of the core principles of this solution is a balance between non web based interaction needs with the scalability and automation features.

At the frontend, the system Uses an ElectronJS-based desktop client application as the main graphical user interface. The cross platform client application offers a smooth, responsive user experience, obscuring the inherent complexity of the cloud services it interacts with. By bypassing the browser model while still taking advantage of JavaScript, HTML, and CSS, ElectronJS facilitates rich, interactive desktop applications without relaxing the "no web app" requirement.

All user interactions initiated by client applications are directed through API Gateway which also acts as a secure broker managing request routing, protocol translation and authentication enforcement. The requests are then queued using Amazon Simple Queue Service (SQS), facilitating decoupled communication among services, fault isolation, and asynchronous message processing—critical to guaranteeing system responsiveness despite variable workloads.

Moving on to the Accounts Sequence Manager, a central element hosted on Amazon EC2 instances backed by the Elastic Block Store for transactional storage, acts as the financial transaction choreographer. It commits all of the transaction records sequentially hence ensuring transactional integrity and traceability. The Ledger Service, hosted on EC2, combines all the financial entries into one immutable data structure that represents the authoritative financial record of the institution.

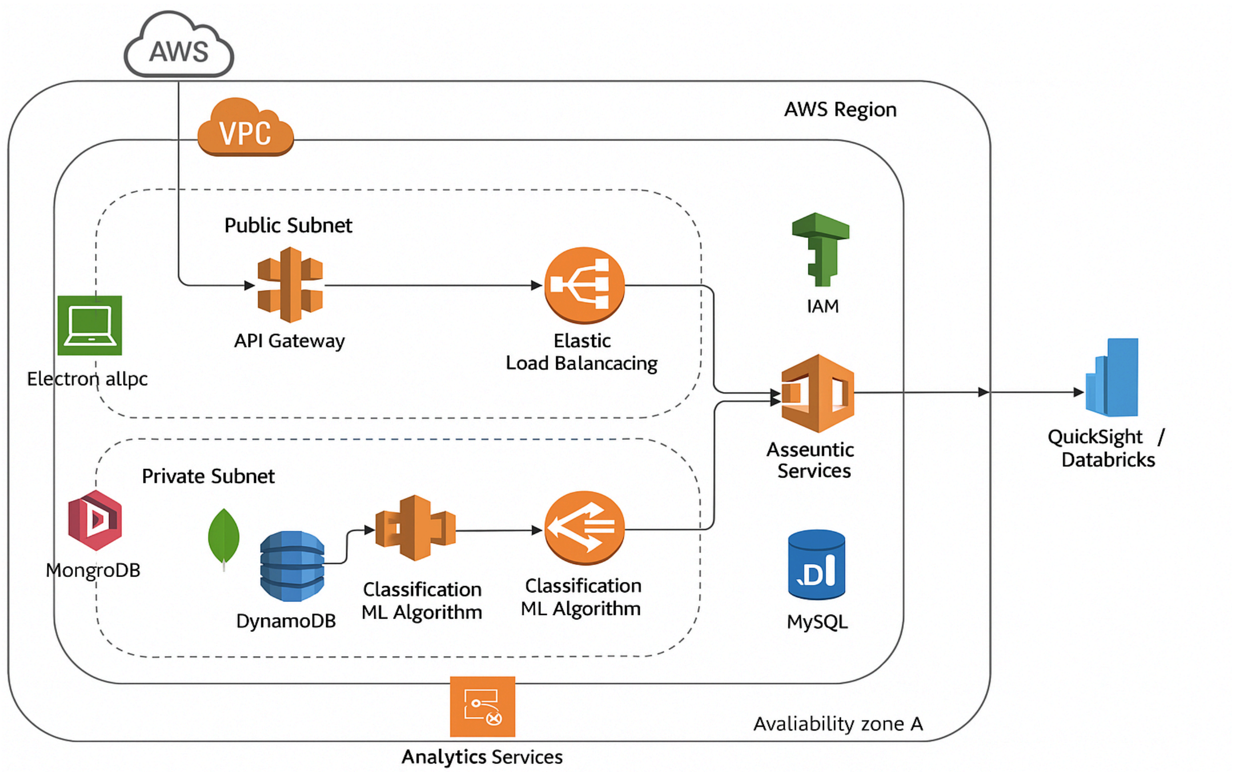
For high-speed and low-latency key-value data access the system utilizes Amazon DynamoDB, particularly for metadata management and in accommodating financial metadata such as transaction tags, real-time budget and approval status. A ML Classification algorithm, containerized and orchestrated via AWS Fargate (proposed addition), introduces machine learning into predictive analytics and fraud detection. The module imparts intelligence to the system by detecting anomalies in spending patterns or anomalous transactional behavior.

Normalized and formatted financial data like user roles, transaction metadata and audit logs is stored in Amazon RDS (MySQL engine). The Audit Engine is a serverless Lambda function , is constantly reading from and writing to an RDS instance like this, ensuring full audit trail and compliance with internal and regulatory requirements.

For semi-structured or unstructured financial data like receipts, departmental reports, and budgetary justifications, MongoDB in a VPC private subnet is the storage layer. Segregation provides data sovereignty and security from unauthorized internet access.

The solution also have Amazon SNS (Simple Notification Service) for real-time notification of alerts, approval workflows, and financial event triggers. For example, when a budget crosses a threshold or suspicious transaction is logged, interested stakeholders are notified in real-time via SNS.

Lastly, security of data is provided by encryption-at-rest and encryption-in-transit using AWS KMS (Key Management Service). Hash functions on sensitive identifiers like student IDs and transaction numbers are used to avoid unauthorized disclosure, tampering, and inference attacks.



As we can see from the above diagram, our application is hosted within a Virtual Private Cloud (VPC) that is further divided into public and private subnets. The ElectronJS desktop application talks to the application via API Gateway, which securely directs the requests to backend services hosted on EC2 instances. The EC2 instances are managed by Auto Scaling Groups and positioned behind Elastic Load Balancers to manage the traffic gracefully and provide system availability.

Data is held in various databases for particular requirements MongoDB for financial data, MySQL for audit trails, and DynamoDB for account ordering. All of these databases are placed in private subnets for an extra layer of security.

Amazon Cognito and IAM are used for user authentication and access control so that the system can be used by valid users only. AWS SQS and AWS SNS are used by the system for background task execution and real-time alert notification. Amazon CloudWatch is used for system activity monitoring and health checking.

In case of analytics and reporting, Databricks and QuickSight are utilized so that users can make sense of the data via dashboards as well as visualizations.

List of Services Used and Justification

In this section we define the cloud services utilized in the architecture, with the detailed justification for every choice.

1. **Amazon EC2 + Amazon EBS**

They constitute the system's computational core. EC2 offers virtual machines for hosting backend services like the Ledger, ML classifiers, and APIs, and EBS offers persistent block-level storage for storing financial data securely and durably. They both facilitate scalability and performance management in a self-managed environment.

2. **Amazon SQS (Simple Queue Service)**

SQS facilitates decoupled communication between asynchronous components. For instance, after posting a transaction, it is put into a queue to enable the system to process requests without overloading any one component. This improves system responsiveness and manages load spikes effectively.

3. **Amazon SNS (Simple Notification Service)**

SNS is used for publishing events and notifications in real-time. It is used in combination with SQS in the publish-subscribe model for informing users or systems about updates to transactions, audit trails, or identification of suspicious behavior.

4. **Amazon API Gateway**

This is the secure gateway between ElectronJS frontend and the backend microservices. It validates, routes, and throttles API calls to grant controlled access to the internal workings.

5. **Amazon DynamoDB**

Selected for the Accounts Sequence Manager, DynamoDB is a fully scalable, serverless NoSQL database with low-latency and high-throughput transaction sequencing that enables reliable ID creation and ordering.

6. **MongoDB**

Utilized for holding flexible financial documents like budgets, journals, and financial reports. Its JSON-like document data model allows for flexible and intricate data modeling, particularly in applications where data is semi-structured and prone to alteration.

7. **MySQL**

Driving the Audit Engine MySQL provides robust data consistency and ACID compliance, essential to store structured audit trails that demand relational integrity.

8. **AWS IAM**

IAM provides granular access control to the solution. Auditors, students, and professors are assigned custom roles to implement the principle of least privilege, granting secure and controlled access to services.

9. **JWT Authentication**

Employed to securely authenticate users and preserve session state in client-server communication. JWTs contain user roles and claims, facilitating fine-grained authorization without maintaining session state on the server.

10. **AWS VPC**

The application is deployed in a Virtual Private Cloud (VPC) for network isolation. Sensitive services such as databases are put in private subnets to avoid unwanted public access.

11. **AWS CloudWatch**

Utilized for central monitoring, logging, and alerting of all AWS services. Provides insight into system usage and health, allowing for proactive maintenance.

12. **AWS QuickSight**

QuickSight provides a serverless BI experience that makes it easy for users to visualize financial performance metrics and build interactive dashboards without the need to transfer data out of AWS.

13. **DataBricks**

Used optionally for more in-depth analytical work, including machine learning and real-time anomaly detection. It adds additional analytical capability to the system beyond visualization.

14. **AWS Auto Scaling & Load Balancer**

Auto Scaling adjusts EC2 instances automatically in response to load, and Load Balancer splits traffic evenly. High availability and reduced costs are the outcomes.

Advantages and Disadvantages of the services we used

TheCloud Services	Their Advantages	Disadvantages
Amazon EC2 + EBS	It is Customizable, scalable compute and storage resources	But it requires manual scaling and instance management
Amazon SQS	It has Decoupled components and supports large-scale async processing	Adds message latency, needs retry/dead-letter handling
Amazon SNS	It gives us Real-time notifications, simple to integrate	Basic filtering and it is not suitable for complex orchestration
Amazon API Gateway	It Secures API access, supports throttling and logging	Higher cost for large-scale usage, setup complexity for advanced configs
Amazon DynamoDB	IT is Serverless, highly scalable, fast read/write performance	But it has Limited complex querying, requires careful partitioning
MongoDB	It has Flexible schema, supports nested JSON, fast query on document data	Requires self-hosting on AWS or use of Atlas and it might be less secure without proper config
MySQL	Strong relational database capabilities, ACID compliant	Less flexible with unstructured data, vertical scaling limitations
AWS IAM	Fine grained access control, integrates across all AWS services	But the Complex policies can be difficult to audit or debug
AWS VPC	Full control over networking and isolation	Complex setup and management
AWS CloudWatch	It gives Integrated monitoring and alerting across services	Logging costs can scale rapidly

AWS QuickSight	The Native AWS integration, BI dashboards, user-friendly	Limited advanced analytics; per-user pricing
DataBricks	It gives Powerful ML for big data and streaming analytics	Requires Spark/ML expertise; cost can be high
Auto Scaling & Load Balancer	Enhances fault tolerance and handles dynamic traffic	Needs proper configuration for cost-effective scaling

Software Qualities and Trade Off

1. Scalability
Scalability was recognized as a important requirement because of the need to support growing numbers of actors and transactions, with peaks in financial processing such as end-of-term refunds or payroll activity. The system uses Amazon DynamoDB for horizontally scalable database activities, EC2 Auto Scaling for dynamic scaling of compute resources, and SQS for decoupling workload to enable more agile scaling patterns.

Trade-Off:
The capacity of the system to scale quickly is at the expense of higher operational cost. Provisioned throughput services and auto-scaling infrastructure need to be correctly configured and continually monitored to prevent wastage of resources and billing surprises.

3. Ease of Use
User experience is a medium level requirement but is important, given the fact that faculty and students with varying technical skills will utilize the system. Leveraging an ElectronJS desktop application has the advantage of having a standard, user-friendly interface like native applications on all platforms.

Trade-Off:
Developing and maintaining a feature-complete ElectronJS application introduces a front-end development overhead committed to managing application packaging on various platforms.

3. Security
Due to the sensitive nature of personal and financial data processed by the system, security is of the utmost priority. Role-based access control with AWS IAM, secure user

authentication with JWT tokens, and encryption of data by AWS KMS, both in transit and at rest, are part of the architecture.

Trade-Off:
Though these mechanisms dramatically improve data security and compliance with regulations, they also demand committed resources for managing policies, key rotation, vulnerability scanning, and regular audits. Security overhead needs to be constantly maintained to avoid exploitation.

4. Performance
It is again a medium-priority requirement but one of the most important user expectations, particularly when there is a period of high-frequency transaction or bulk upload. Adopting SQS for async message queuing, DynamoDB for fast query performance, and stateless microservices maintains responsiveness under load.

Trade-Off:
Asynchronous design adds complexity to data consistency and reliability of message handling. Developers need to cater to eventual consistency, retries, and message duplication—needing robust error handling and test processes.

5. Maintainability
Maintainability is relatively high-priority to provide the system's long-term viability and flexibility to integrate new requirements or functionality. Through modular microservices, containerization, and CI/CD pipelines, the application is designed to be updated and deployed with minimal friction and downtime.

Trade-Off: Although these contemporary practices make system evolution leaner, they have a steep learning curve for development teams with no experience in DevOps tools, container orchestration, or cloud pipelines. There needs to be investment in training developers and documentation to ensure system agility.

6. Flexibility
Flexibility is required to address evolving institutional needs or adding features in the future. Modular, service-oriented architecture ensures new functionality can be introduced with minimal effect on existing components.

Trade-Off:
While modularity supports adaptability, it also introduces complexity in inter-service communication and require sophisticated orchestration frameworks.

7. Reliability
Reliability is a key attribute, particularly because the app deals with core financial functions where downtime can hold up reimbursements or payroll cycles. AWS services such as multi-AZ EC2 instances, load balancers, and SQS queues make services always available and resilient to failure with zero data loss and no disruption to users.

Trade-Off:
High reliability necessitates architectural redundancy, which adds to infrastructure expense and operational complexity. Keeping standby resources, determining failover techniques, and monitoring service health across availability zones can result in more ongoing maintenance and monetary investment.

8. Cost-Efficiency
Cost-effectiveness matters, especially while dealing with budgetary limitations in public schools. The platform achieves cost-effectiveness through EC2 spot instances, on-demand scaling of DynamoDB, and open-source analytics software such as Metabase. Pay-as-you-go pricing on AWS provides financial flexibility.

Trade-Off: Cost-saving measures need to be watched and optimized constantly. Misconfigured services, unutilized resources, or unplanned peaks can cause budget excesses. A properly established cost governance process and alarming mechanism need to be in place to ensure long-term affordability.

In brief, our architecture plan prioritizes scalability, reliability, security, and performance as they are non negotiable in an financial setup. In the meantime, the architecture also takes into account maintainability, cost-effectiveness, usability, and flexibility to ensure the solution remains viable and easy to use in the long run.

Risks and mitigation techniques
We believe our solution is very reliable, secure and scalable system but still it should expect potential threats and we should prepare in advance for them:

1. Data Examination
Risk: Databases with inconsistent data and inefficient or obsolete code could slow down system performance or cause havoc in the form of unforeseen malfunctions.

Mitigation: It Require regular code and database updates, including the regular patching, schema validation, and regression testing to ensure consistency, reliability, and optimal performance.

2. Audit System Alert Deficiency
Risk: Inability to provide real-time audit subsystem alerts can allow anomalies and/or unauthorized changes can go undetected.

Mitigation: Activate real-time logging, alerting and anomaly detection mechanisms like CloudWatch Alerts + Lambda triggers to notify auditors and admins of suspicious behavior.

4. Delays in Account Creation from Batch Processing
Risk: Batching account creation will result in onboarding delays and undermine user experience.

Mitigation: Transition to event-driven architecture on AWS SQS or Lambda triggers for real-time processing to onboard users efficiently.

5. Ordering Issues in Event-Based Architecture
Risk: Out-of-order processing of events would make financial records inconsistent or transactional errors

Mitigation: Using a sequencing mechanism like DynamoDB with sequence key and idempotent message processing to achieve ordered processing and recoverability in the event of inconsistency

6. Human Error Leading to Calculation Errors
Risk: Manual entry or inconsistent data formats (currencies, units) can result in financial errors.

Mitigation: Normalize currency/unit conversions at the system level, validate input, and automatically calculate key calculations based on pre-defined rules or ML classification.

7. Data Breach or Unauthorized Access
Risk: Unauthorized users gain access to sensitive accounting and user data, causing data breaches.

Mitigation: Enable role-based access control (IAM), encrypt data in transit and at rest (KMS, SSL), and audit logs for monitoring of access patterns. JWT-based authentication is secure identity verification.

8. Misconfigured Cloud Services
Risk: Misconfigured cloud service like open security groups, public subnets for databases can expose the system to external exploits.

Mitigation: Regularly run cloud security checks with AWS Config and Trusted Advisor, and use infrastructure as code policies to validate configuration before deployment.

9. Auto-Scaling or Event Flooding Cost Overruns
Risk: Auto-scaling and/or uncontrolled message queue expansion could result in unexpected cloud costs.

Mitigation: Seting up budget alerts and service usage limits on services. Apply backpressure or throttling controls to handle traffic surges without overwhelming systems or budgets.

10. Single Point of Failure
Risk: Some critical services (e.g classification ML model or Accounts Sequencer) could be single points of failure.

Mitigation: Use high-availability configurations (multi-AZ deployments), auto-recovery capabilities, and fallback methods to provide continuity in the case of component failure.

Conclusion

The design and deployment of the cloud accounting system, Show Me The Money, provide a scalable, secure, and high-performance solution tailored to the financial management needs of today in an educational environment. With a carefully considered combination of AWS services and open-source elements, the system satisfies the primary needs of scalability, reliability, security, and maintainability—yet still delivers an end-user-friendly experience via the ElectronJS desktop client.

Through the inclusion of such pieces as the Ledger, Accounts Sequence Manager, Audit Engine, and Classification ML Algorithm, the design offers not only accurate and well-organized tracking of transactions but real-time fraud detection, auditing, and financial analysis. Services like Amazon EC2, DynamoDB, MongoDB, SQS, and API Gateway are handpicked to ensure high performance and modularity, while additional outside tools like Metabase, Databricks, and QuickSight exist to further develop reporting and visualization power.

The system has been architected with a comprehensive understanding of numerous software-ilities, striking a fine balance between performance, cost, security, usability, and flexibility. Additionally, risks have been foreseen and mitigated through visionary architectural decisions to facilitate long-term reliability and compliance.

Lastly, the suggested architecture is not merely a functional solution but a foresighted platform that can expand with user requirements, adapt to emerging developments, and uphold the utmost best practices of data integrity and access. It is a stable platform for financial applications in educational settings and can be easily extended or redefined for broader institutional use.

References

- https://aws.amazon.com/sns/
- https://aws.amazon.com/ec2/
- https://aws.amazon.com/products/compute/
- https://www.electronjs.org/docs/latest/tutorial/tutorial-adding-features
- https://www.electronjs.org/
- https://www.databricks.com/
- https://aws.amazon.com/sqs/
- https://aws.amazon.com/sqs/
- https://aws.amazon.com/dynamodb/
- https://www.electronjs.org/docs/latest/tutorial/tutorial-adding-features
- https://aws.amazon.com/sqs/features/
- https://en.wikipedia.org/wiki/MD5
- https://aws.amazon.com/ec2/?p=pm&c=mt&pd=ec2&z=4