DALHOUSIE UNIVERSITY

CSCI 5409 - Advanced Topics in Cloud Computing

Term Assignment Report

Name: Abhinav Acharya Tirumala Vinjamuri

Banner: B00929073

Email: ab806657@dal.ca

Table of Contents

Application Overview

Browser Notes – Reforming the way you take notes down.

Motivation

Being a master's student at Dalhousie, I had to search for and apply to co-op programs as part of my academic curriculum. Applying for over 200 jobs is extremely exhausting, let alone the fact that you forget most of the jobs that you apply for. To keep track of the jobs I applied for, to make notes about certain points about the job role, or to filter some jobs out of the list, I always had to maintain a notepad/excel. But this did not make my work easier, as I always had to switch tabs to write down something, which gets frustrating at some point. Having the ability to note down something while still browsing a page is a necessity, especially while dealing with lengthy job role and responsibilities sections.

Inspiration

This is when I came across an extension called Edge Notes [3]. It is a browser extension that lets you write down your notes about anything without having to leave the tab that you are browsing. You can create as many notes as you want to, and they are stored in your local storage session data [3].

Overcoming the limitations in current implementation

A problem with the current implementation of Edge Notes is that the data won't be retained if you choose to reinstall, install on another device, and there is no way to use this on any other device [3].

To tackle these problems, I chose to architecture the backend of such a note-taking application using AWS serverless architecture. I carefully chose different services for my backend, as we want the functionality of the application to be reliable, efficient, and most importantly, fast.

Context of my application

I derived my project idea inspiration from this extension and tried to recreate the application in a versatile enough way to work on any setting. I wanted to allow users to not worry about the data retention, ease of transfer to other devices, and efficiency of the application.

With my application/extension, Browser Notes, a user can register, authenticate, create as many notes as they wish, export them to their email, not worry about data retention as it is backed up every 5 minutes.

Target Users

The target users of my application are everyone who has access to a computer and a browser that can support extensions. Students, researchers, debuggers, etc., are some of the categories that the potential users of the application could belong to.

Performance Targets

I expect my application to be responsive with minimal single digit latency or delay while creating, editing, or accessing notebooks. As the number of users grows, the application should be able to handle the increased traffic and the data volumes without experiencing performance degradation. The users must be able to export the notebook of their choice to their email with ease. The data must be backed up in a reliable storage that allows for recovery in times of disaster.

Notes (Acknowledgements)

For the sake of creating a browser extension, the frontend of the application is meant to be published in the browser's extension store. For the same reason, **I chose to only architecture the backend of this application on AWS cloud as my term assignment**, not the entire application. This decision allows for a lot of versatility in the application regarding which I will be talking about later in the report in the architecture section.

The frontend of the application can be maintained locally during development, maintained using Git, or **once published on the extension store (which typically takes about 3-4 weeks) [4],** it is automatically managed by the browser.

Since publishing the extension takes such lengthy times, I decided to showcase the front-end of the application which was built locally and loaded on to the browser's extensions. This is because of the impending deadline that is in a week which would be insufficient to publish this extension.

Menu Item Requirements

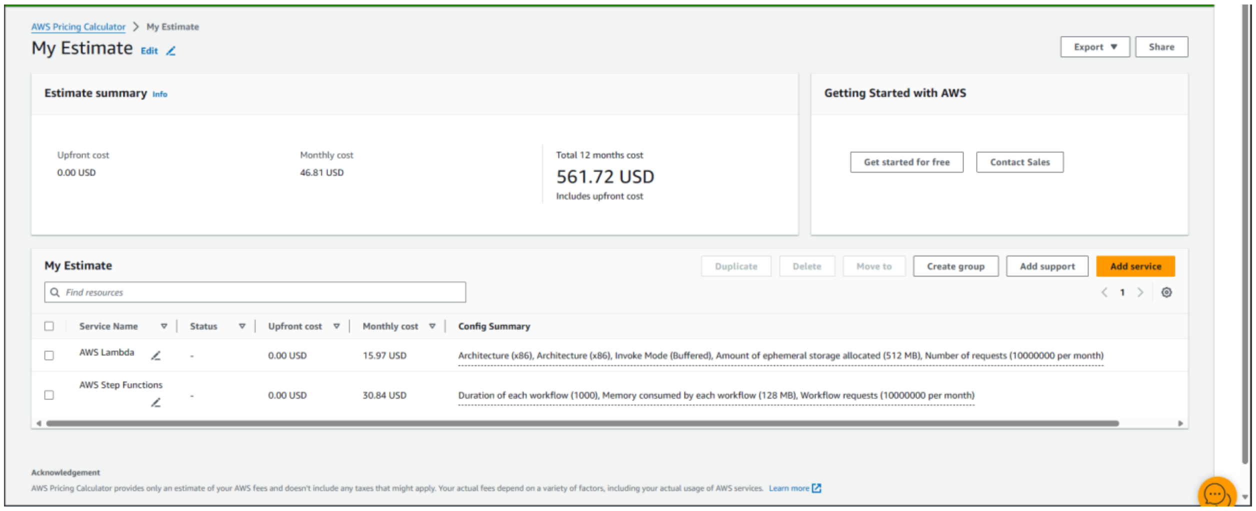
Compute Section

Choice of services

I opted for a serverless architecture for Browser Notes to ensure automatic management and scaling, as well as to mitigate costs during periods of inactivity. This choice aligns with the application's goal of being versatile and cost-effective for users.

AWS Lambda was selected for executing functions due to its event-driven nature, allowing for efficient scaling and cost optimization. Lambda's pay-per-execution pricing model ensures cost-effectiveness by charging only for the actual compute time consumed, minimizing expenses during idle periods [5].

Additionally, **AWS Step Functions** were utilized to build a serverless state machine, providing a robust framework for orchestrating and coordinating workflows. Step Functions offer scalability and fault tolerance, ensuring reliable execution of complex processes. By leveraging these serverless services, Browser Notes benefits from automatic scalability, cost efficiency, and reliable workflow orchestration [6].



My Estimate

Estimate summary

Upfront cost	Monthly cost	Total 12 months cost
0.00 USD	46.81 USD	561.72 USD Includes upfront cost

Getting Started with AWS

My Estimate

Service Name	Status	Upfront cost	Monthly cost	Config Summary
AWS Lambda	-	0.00 USD	15.97 USD	Architecture (x86), Architecture (x86), Invoke Mode (Buffered), Amount of ephemeral storage allocated (512 MB), Number of requests (10000000 per month)
AWS Step Functions	-	0.00 USD	30.84 USD	Duration of each workflow (1000), Memory consumed by each workflow (128 MB), Workflow requests (10000000 per month)

Acknowledgement

AWS Pricing Calculator provides only an estimate of your AWS fees and doesn't include any taxes that might apply. Your actual fees depend on a variety of factors, including your actual usage of AWS services. Learn more

Figure 1 Cost estimation of the services of my choice in compute section., source: [1]

Alternatives

Alternatively, I could have chosen services like **EC2**, **Elastic Beanstalk**, **ECS** & **ECR**, or **EKS** for the compute section. However, these alternatives have notable disadvantages compared to the chosen serverless architecture. For instance,

- **EC2** instances require manual provisioning and scaling, leading to higher operational overhead and potential underutilization during low-traffic periods.
- **Elastic Beanstalk** offers easier deployment but lacks the automatic scaling capabilities of serverless architectures, potentially resulting in higher costs and slower response times during peak loads.
- **ECS** & **ECR** and **EKS** provide container orchestration but require additional management efforts and infrastructure provisioning compared to serverless options.

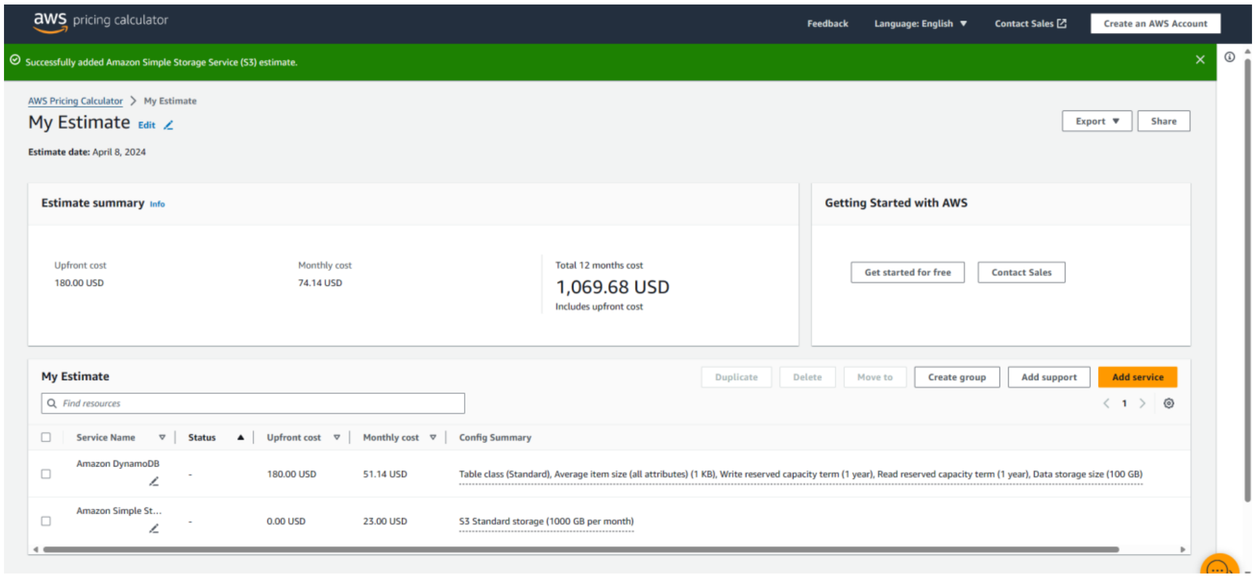
Overall, these alternatives may offer more control but come with increased complexity, higher operational costs, and slower scalability compared to the serverless approach adopted for Browser Notes.

Storage Section

Choice of services

For the database, I opted for **DynamoDB** due to its reliability, single-digit latency **(>1ms),** speed, high availability, and scalability. DynamoDB's architecture ensures consistent and predictable performance with low-latency responses, making it ideal for real-time applications like Browser Notes. Additionally, DynamoDB's fully managed service model eliminates the need for manual provisioning and maintenance, allowing for seamless scalability as user demand fluctuates. Metrics indicate that DynamoDB consistently delivers single-digit millisecond response times, ensuring rapid access to user data [7].

In terms of data retention, the good old **S3** was chosen for its efficiency and scalability. S3 provides unlimited storage capacity with a maximum file size of 5TB, making it suitable for storing and backing up large volumes of user data. The managed nature of S3 simplifies data storage and retrieval processes, while its durability and availability ensure that user data remains secure and accessible.



Successfully added Amazon Simple Storage Service (S3) estimate.

AWS Pricing Calculator > My Estimate

My Estimate

Estimate date: April 8, 2024

Estimate summary

Upfront cost	Monthly cost	Total 12 months cost
180.00 USD	74.14 USD	1,069.68 USD Includes upfront cost

My Estimate

Service Name	Status	Upfront cost	Monthly cost	Config Summary
Amazon DynamoDB	-	180.00 USD	51.14 USD	Table class (Standard), Average item size (all attributes) (1 KB), Write reserved capacity term (1 year), Read reserved capacity term (1 year), Data storage size (100 GB)
Amazon Simple St...	-	0.00 USD	23.00 USD	S3 Standard storage (1000 GB per month)

Figure 2 Estimated cost of 1 year for the services of my choice in Storage section. Source: [1]

Alternatives

Alternatively, **AWS Aurora** or **AWS RDS** could have been considered for the database. However, these services may not offer the same level of performance and scalability as DynamoDB.

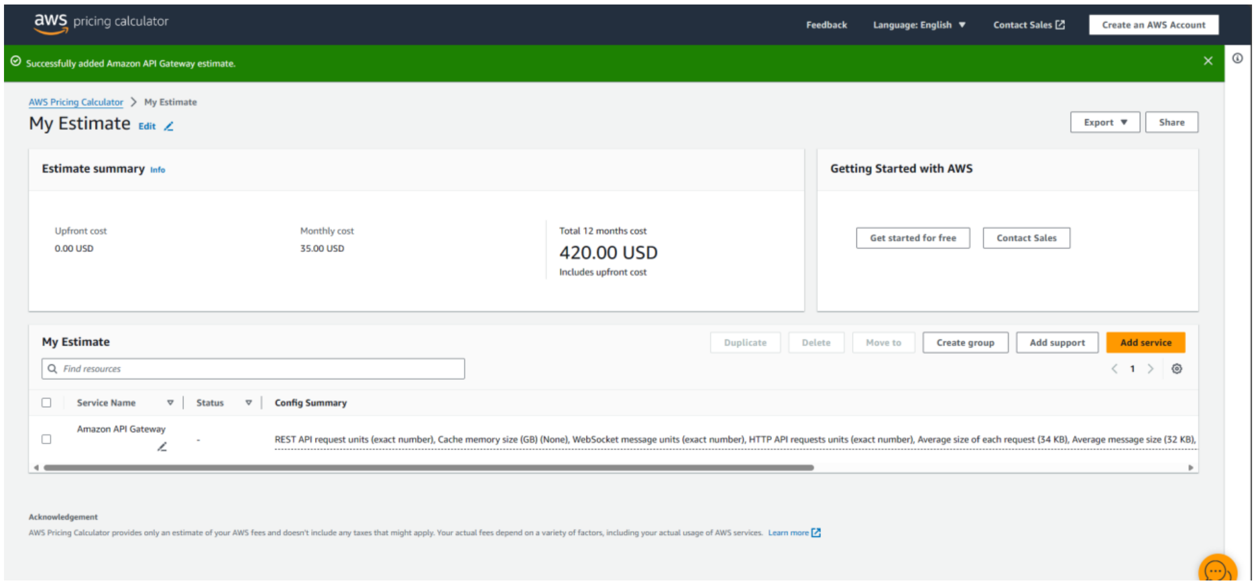
- **AWS Aurora**, while providing excellent performance and compatibility with MySQL and PostgreSQL databases, may incur higher costs compared to DynamoDB, especially as data volume increases. Metrics indicate that Aurora may have slightly higher latency compared to DynamoDB, impacting real-time responsiveness.
- Similarly, **AWS RDS**, while offering relational database capabilities, may face limitations in scalability and performance under high load conditions. The manual management required for scaling and maintenance tasks can introduce operational overhead and potential points of failure, making it less suitable for dynamic and rapidly evolving applications like Browser Notes.

Therefore, **DynamoDB** was deemed the optimal choice for its superior performance, scalability, and ease of management.

Network Section

Choice of services

I chose **AWS API Gateway** which provides a secure route for API requests to AWS services. I needed an entry point for my backend that allowed users to execute and use the backend functionality. API Gateway is the only solution that allows us to do this, given the microservices that I have in place (**AWS Lambdas**) [8].



Successfully added Amazon API Gateway estimate.

AWS Pricing Calculator > My Estimate

My Estimate Edit

Estimate summary Info

Upfront cost	Monthly cost	Total 12 months cost
0.00 USD	35.00 USD	420.00 USD Includes upfront cost

Getting Started with AWS

My Estimate

Service Name	Status	Config Summary
Amazon API Gateway	-	REST API request units (exact number), Cache memory size (GB) (None), WebSocket message units (exact number), HTTP API requests units (exact number), Average size of each request (34 KB), Average message size (32 KB),

Acknowledgement

AWS Pricing Calculator provides only an estimate of your AWS fees and doesn't include any taxes that might apply. Your actual fees depend on a variety of factors, including your actual usage of AWS services. Learn more

Figure 3 Estimated cost of 1year for the services of my choice in Network section. Source: [1]

Alternatives

Alternatively, had I chosen **EC2** for my Compute section, I would have preferred using **AWS VPC** in this section to segregate my application from the rest of the cloud resources on AWS. This segregation is crucial for enhancing security and minimizing the risk of unauthorized access to sensitive data. While EC2 instances offer flexibility and control over virtual machines, AWS VPC provides network isolation and control, significantly reducing the likelihood of security breaches. Additionally, VPC ensures better performance and scalability by allowing fine-grained control over network traffic and resources allocation.

General section

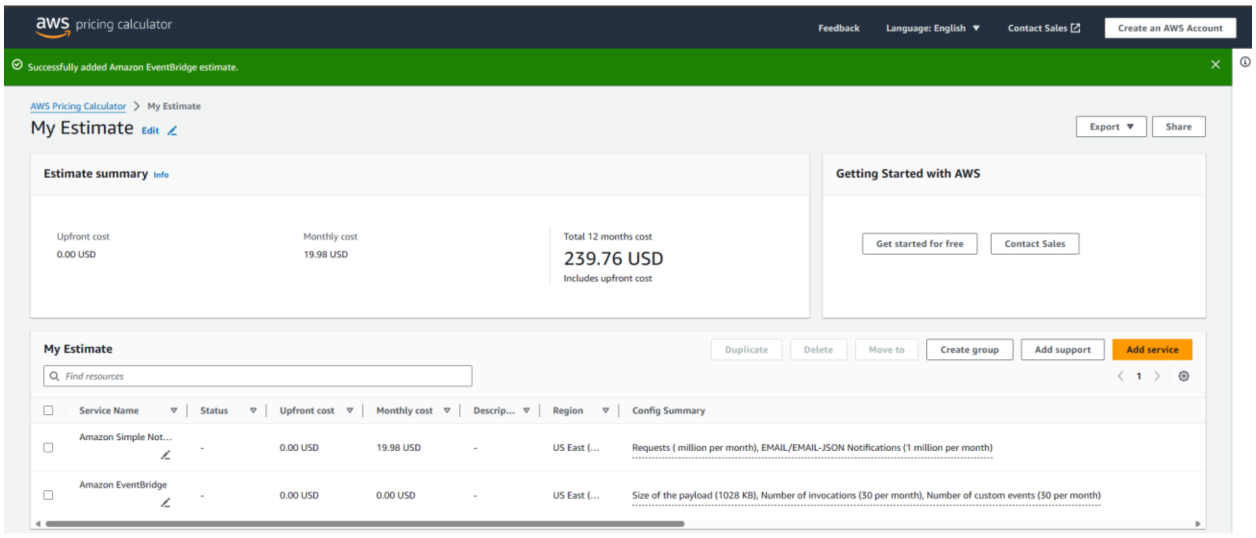
Choice of services

I strategically selected services to address specific challenges encountered in Edge Notes, focusing on exporting notebooks to users' email addresses and ensuring robust data retention.

For exporting notebooks, **AWS SNS** was chosen as it offers a reliable and scalable solution for sending notifications and messages via email. Its seamless integration with other AWS services allows for efficient delivery of exported notebooks to users' designated email accounts [9].

Additionally, to address data retention concerns, **AWS EventBridge** was utilized to automate the periodic backup of data from **DynamoDB** to **S3**. This ensures that user data is consistently backed up, mitigating the risk of data loss and ensuring data integrity [10].

Also, being a CSCI 5409 student, I am required to use **Cloud Formation** (Infrastructure as Code) to provision and architecture my services on the AWS cloud. **IaC** allows for easy integration of new services into the existing architecture by simply adding its creation schema into the JSON or YAML template for cloud formation.



Successfully added Amazon EventBridge estimate.

My Estimate

Estimate summary

Upfront cost	Monthly cost	Total 12 months cost
0.00 USD	19.98 USD	239.76 USD Includes upfront cost

Getting Started with AWS

My Estimate

Service Name	Status	Upfront cost	Monthly cost	Descrip...	Region	Config Summary
Amazon Simple Not...	-	0.00 USD	19.98 USD	-	US East (...	Requests (million per month), EMAIL/EMAIL-JSON Notifications (1 million per month)
Amazon EventBridge	-	0.00 USD	0.00 USD	-	US East (...	Size of the payload (1028 KB), Number of invocations (30 per month), Number of custom events (30 per month)

Figure 4 Estimated cost for 1 year of choice of services in General section. Source: [1]

Alternatives

Alternatively, I initially considered implementing additional features such as translating notes using **AWS Translate**, implementing speech-to-text functionality with **AWS Transcribe**, extracting text from uploaded files using **AWS Textract**, or enabling text-to-speech capabilities with **AWS Polly**.

However, limitations imposed by the AWS Academy account restricted the usage of these services, prompting the selection of **SNS** and **EventBridge**. While these alternatives could potentially enhance the application's functionality, their implementation was not feasible due to account restrictions. Nonetheless, it's worth noting that services like **AWS Translate, Transcribe, Textract,** and **Polly** offer advanced capabilities that could significantly enrich the user experience by providing translation, transcription, text extraction, and speech synthesis functionalities. The inability to utilize these alternatives due to account restrictions underscores the importance of resource availability and constraints in the decision-making process.

Deployment Model

The deployment model for Browser Notes follows a serverless architecture deployed on **AWS (Amazon Web Services)**, which is a **Public Cloud**. I chose a public cloud deployment model primarily for its scalability, reliability, and cost-effectiveness.

1. **Scalability:** Public cloud providers like AWS offer virtually unlimited scalability, allowing applications to handle fluctuating workloads seamlessly. With AWS **Lambda**, for example, resources automatically scale up or down based on demand, ensuring that Browser Notes can accommodate varying levels of user activity without manual intervention. This scalability is crucial for an application like Browser Notes, which may

experience spikes in usage during peak times, such as when many users are browsing job listings simultaneously.

2. **Reliability:** Public cloud providers invest heavily in infrastructure redundancy and disaster recovery measures, resulting in high levels of reliability and availability. By hosting Browser Notes on AWS, I can leverage AWS's robust data centers and global network infrastructure to ensure that the application always remains accessible to users. Features like **DynamoDB's** multi-region replication and **S3's** data durability contribute to the overall reliability of Browser Notes, minimizing the risk of downtime or data loss.
3. **Cost-effectiveness:** Public cloud services operate on a pay-as-you-go pricing model, allowing organizations to only pay for the resources they consume. This cost-effective pricing structure eliminates the need for upfront hardware investments and allows for greater flexibility in resource allocation. For Browser Notes, which may experience variable workloads and traffic patterns, this pay-as-you-go model helps optimize costs by scaling resources dynamically based on demand. Additionally, the elimination of upfront infrastructure costs reduces financial barriers to entry for hosting and managing the application.

Overall, the decision to deploy Browser Notes on a public cloud like AWS aligns with the application's requirements for scalability, reliability, and cost-effectiveness. By leveraging AWS's extensive suite of services and global infrastructure, Browser Notes can deliver a seamless and reliable user experience while optimizing costs and resource utilization.

Delivery Model

The delivery model chosen for Browser Notes involves a serverless architecture deployed on AWS (Amazon Web Services), leveraging various AWS services such as AWS Lambda for the backend, DynamoDB for the database, and S3 for data backups. This deployment model is characterized by its use of Function as a Service (FaaS) for the backend, Platform as a Service (PaaS) for the database, and Database as a Service (DBaaS) for data backups.

Reasoning for Choosing this Delivery Model:

1. **Function as a Service (FaaS) for Backend (AWS Lambda):** I chose AWS Lambda as the backend solution for Browser Notes due to its serverless architecture, which offers several advantages such as automatic scaling, pay-as-you-go pricing, and reduced operational overhead. With AWS Lambda, I can focus on writing code to handle specific functions or tasks without worrying about provisioning or managing servers. This allows for greater agility, cost-effectiveness, and scalability, making it an ideal choice for a dynamic application like Browser Notes. Additionally, AWS Lambda integrates seamlessly with other AWS services, enabling efficient development and deployment workflows.

2. **Platform as a Service (PaaS) for Database (AWS DynamoDB):** DynamoDB was selected as the database solution for Browser Notes due to its serverless architecture, high availability, and scalability. As a fully managed NoSQL database service, DynamoDB

offers automatic scaling, built-in replication, and low-latency access to data, making it well-suited for applications with unpredictable workloads like Browser Notes. By using DynamoDB as a PaaS solution, I can offload the responsibility of managing database infrastructure to AWS, allowing me to focus on developing and optimizing the application's features and functionality.

3. **Database as a Service (PaaS) for Data Backups (AWS S3):** AWS S3 was chosen for storing backups of data from DynamoDB, effectively serving as a PaaS solution for Browser Notes. S3 offers durable object storage with high scalability, availability, and data durability, making it an ideal choice for storing backups of critical application data. By leveraging S3 for data backups, I can ensure the integrity and availability of user data, while also benefiting from features such as versioning, lifecycle policies, and cross-region replication to enhance data protection and compliance.

Overall, the combination of **FaaS** for the backend (AWS Lambda), **PaaS** for the database (AWS DynamoDB), and data backups (AWS S3) aligns with the goals of Browser Notes in terms of scalability, reliability, cost-effectiveness, and ease of management. This delivery model allows for efficient development, deployment, and maintenance of the application while leveraging the scalability and reliability of cloud-native services provided by AWS.

Architecture

Browser Notes follows a serverless architecture deployed on **AWS (Amazon Web Services)**. This architecture leverages various AWS services to host and manage the application without the need to provision or manage servers. Key components of the deployment model include:

1. **AWS Lambda:** I utilized AWS Lambda for executing code in response to triggers, such as HTTP requests or events from other AWS services such as **State Machine**. Each function within Browser Notes is deployed as a Lambda function, allowing for automatic scaling and eliminating the need to manage server infrastructure. Each of the Lambda is created using **Python3.10** as the programming language and **x84** architecture.
2. **AWS Step Functions (State Machine):** Browser Notes uses a state machine to route the incoming requests from the API gateway to respective invoke specific lambdas, based on an action parameter which is passed as an input to the API gateway. I had created an express state machine which supports synchronous API calls using HTTP requests. I configured the state machine such that it invokes different lambdas based on different values of **"action"** parameter in the input.
3. **AWS API Gateway:** AWS API Gateway serves as the entry point for Browser Notes, providing a secure and scalable API interface for accessing Lambda functions. It handles incoming REST API requests, routing them to the **State Machine**, which routes them to the appropriate Lambda function for processing. I created REST API Gateway and created a resource and a method to connect my state-machine which is configured to invoke different lambdas on different conditions. I used **StartSyncExecution** API as the action to connect to the state machine [11], as I wanted to return the output of the lambda

to the frontend. Without **StartSyncExecution**, the calls to state machine are going to be asynchronous which would return a timestamp and invocation arn which we need to keep polling to fetch results [12].

4. **AWS DynamoDB:** Browser Notes utilizes DynamoDB as its database solution for storing user notes and related data. DynamoDB offers seamless integration with serverless architectures, providing high availability, scalability, and low-latency access to data.
5. **AWS S3:** Browser Notes utilizes Amazon S3 for storing DynamoDB data backups. S3 offers durable object storage with high scalability and availability, making it suitable for storing large volumes of data, including backups of DynamoDB tables.
6. **AWS SNS**: AWS SNS is used for sending notifications and messages via email, facilitating the export of notebooks to users' email addresses. A topic is created and the arn of this topic is passed as environment variable to the lambdas that are involved in sending an SNS notification.
7. **AWS EventBridge:** Additionally, EventBridge is employed to automate periodic backups of data from DynamoDB to S3, ensuring data retention and integrity. In cloud formation, I had to create an additional policy for the event bridge rule to invoke the respective lambda successfully.

Reasons for choosing this architecture.

I chose a serverless deployment model on **AWS** for several reasons:

1. **Scalability:** Serverless architectures, leveraging services like AWS Lambda, allow for automatic scaling based on demand. This ensures that Browser Notes can handle fluctuating workloads efficiently without the need for manual intervention or over-provisioning of resources.
2. **Cost-effectiveness:** With serverless computing, we only pay for the compute resources consumed during execution, rather than for idle server capacity. This results in cost savings, especially for applications with variable workloads or low traffic volumes.
3. **High availability and reliability:** AWS services like Lambda, DynamoDB, and S3 are designed for high availability and durability, offering built-in redundancy and data replication across multiple availability zones. This ensures that Browser Notes remains accessible and reliable, even in the event of hardware failures or outages.
4. **Flexibility and agility:** Serverless architectures promote rapid development and deployment cycles, enabling quick iteration and updates to the application. This agility is essential for adapting to changing requirements and user feedback in a timely manner.

Overall, the serverless deployment model on AWS aligns with our goals of building a scalable, cost-effective, and reliable application like Browser Notes, while also providing flexibility and ease of management for ongoing development and operations.

The chosen deployment model, focusing solely on architecting the **backend** on AWS while leaving the frontend for local maintenance or automatic management by the browser extension store, aligns with the goal of versatility in application usage. By concentrating on the backend architecture, the same APIs can be seamlessly integrated into various applications, including

extensions, mobile apps, or web applications, without necessitating any alterations in the backend. This approach ensures consistency and ease of development across different platforms, enhancing the application's versatility and adaptability to diverse user needs and preferences.

Moreover, if I want to extend this application to a web application, I can still browse different AWS services such as **Fargate**, or **EC2**, or **Elastic Beanstalk** to support my backend architecture.

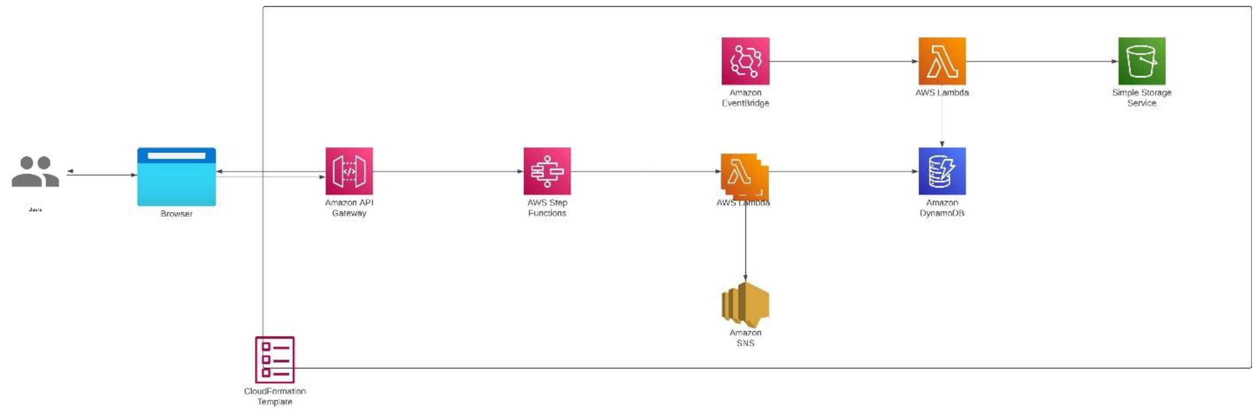


Figure 5 Architecture Diagram of my deployed backend. Source: Author, [2]

Coming to the frontend, which is maintained locally, and is not part of the assignment, is created using **ReactJS** and **Tailwind CSS** to create a minimalistic UI that is attractive and easy to use. This frontend needs 2 variables in order to successfully call the backend APIs: state machine arn, and the API Gateway url. Both of these variables are available and can be fetched manually after the stack is created from cloud formation and are provided to the frontend by updating values on the config file in the frontend.

Security

Browser Notes prioritizes data security by implementing **encryption at rest** and **in transit** for both **DynamoDB** and **S3**, ensuring the confidentiality and integrity of user data by default. Data stored in **DynamoDB** is encrypted at rest using AWS encryption mechanisms, which utilize **AWS Key Management Service (KMS)** for encryption key management. This ensures that sensitive information remains protected even if the underlying storage media is compromised. Similarly, data transferred between the application's components and external services, such as between the frontend and API Gateway, is encrypted in transit using HTTPS/TLS protocols. This encryption prevents unauthorized access to data during transmission and protects against eavesdropping and tampering.

To maintain the integrity of requests and responses, Browser Notes implements secure configurations in AWS **API Gateway**. API Gateway is configured to enforce HTTPS communication, ensuring that all data transmitted between clients and the backend remains encrypted and secure. Additionally, API Gateway is configured with input validation and parameterized queries to prevent common web application security threats such as injection attacks. By enforcing secure communication and validating inputs, Browser Notes mitigates the risk of data manipulation or interception, preserving the integrity of user interactions with the application.

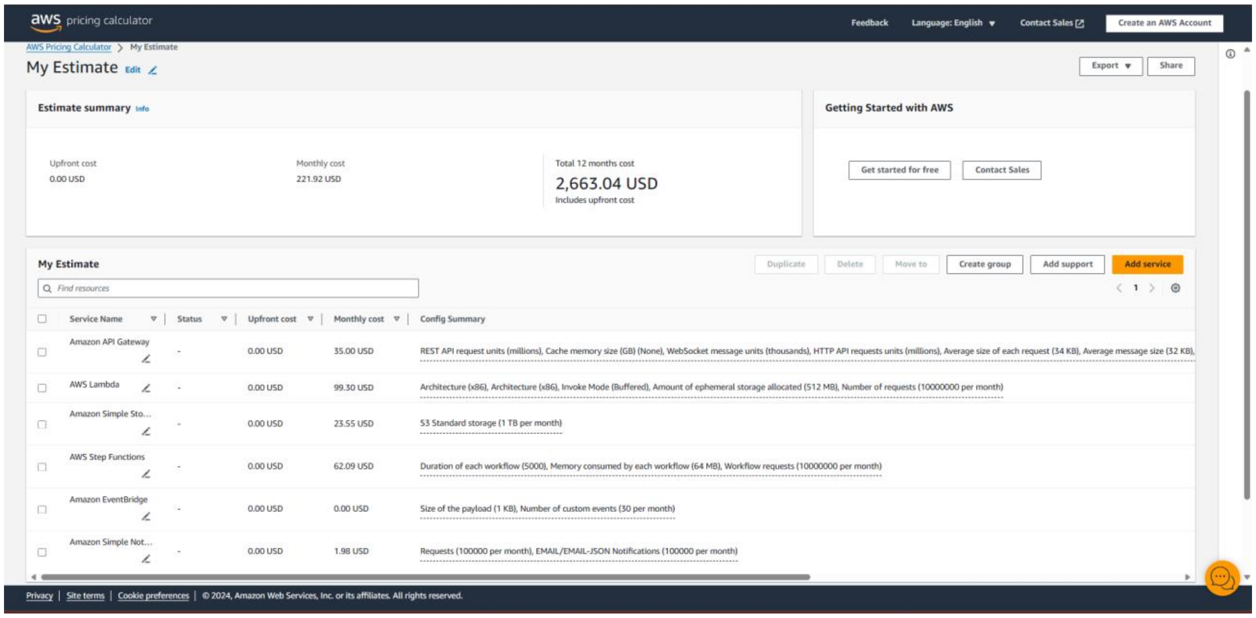
Furthermore, Browser Notes implements route guarding to authenticate users and control access to protected resources. This ensures that only authenticated and authorized users can access sensitive features or data within the application. By implementing route guarding, Browser Notes effectively safeguards against unauthorized access and protects user privacy and data confidentiality.

In summary, Browser Notes employs encryption at rest and in transit for DynamoDB and S3, secures API Gateway configurations to maintain request integrity, and implements route guarding to authenticate users and control access to protected resources. These security measures collectively enhance the confidentiality, integrity, and availability of user data within the application, ensuring a secure and trustworthy user experience.

Costs, Alternatives, Reproducing your architecture in a private cloud.

Cost

Cost Calculator link - https://calculator.aws/#/estimate?id=495cd7b11f3377d6e4a295b07267303447f867b3.



AWS Pricing Calculator > My Estimate

My Estimate

Estimate summary

Upfront cost	Monthly cost	Total 12 months cost
0.00 USD	221.92 USD	2,663.04 USD Includes upfront cost

My Estimate

Service Name	Status	Upfront cost	Monthly cost	Config Summary
Amazon API Gateway	-	0.00 USD	35.00 USD	REST API request units (millions), Cache memory size (GB) (None), WebSocket message units (thousands), HTTP API requests units (millions), Average size of each request (34 KB), Average message size (32 KB),
AWS Lambda	-	0.00 USD	99.30 USD	Architecture (x86), Architecture (x86), Invoke Mode (Buffered), Amount of ephemeral storage allocated (512 MB), Number of requests (10000000 per month)
Amazon Simple Sto...	-	0.00 USD	23.55 USD	S3 Standard storage (1 TB per month)
AWS Step Functions	-	0.00 USD	62.09 USD	Duration of each workflow (5000), Memory consumed by each workflow (64 MB), Workflow requests (10000000 per month)
Amazon EventBridge	-	0.00 USD	0.00 USD	Size of the payload (1 KB), Number of custom events (30 per month)
Amazon Simple Not...	-	0.00 USD	1.98 USD	Requests (100000 per month), EMAIL/EMAIL-JSON Notifications (100000 per month)

Privacy | Site terms | Cookie preferences | © 2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

Figure 6 Estimated cost of maintaining my current architecture on AWS cloud, Source: [1]

The cost estimation for Browser Notes has been calculated based on the usage of various AWS services in the US East 1 (N. Virginia) region. Here's the reasoning behind each cost component [13, 1]:

1. **Amazon API Gateway**: The estimated cost is $35.00 per month, based on 10 million REST API requests and 10 WebSocket message units. The pricing is determined by the number of requests and message units consumed, along with the average size of each request and message.
2. **AWS Lambda**: The estimated cost is $99.30 per month, based on 10 million requests and 512 MB of allocated ephemeral storage. The pricing is calculated based on the number of requests and the allocated memory for each invocation.
3. **S3 Standard**: The estimated cost is $23.55 per month for 1 TB of S3 Standard storage. The pricing is determined by the amount of storage used per month.
4. **Data Transfer**: There is no estimated cost for data transfer as it's not selected, indicating that there are no outbound or inbound data transfer charges associated with the selected services.
5. **Step Functions - Express Workflows**: The estimated cost is $62.09 per month, based on 10 million workflow requests, each consuming 64 MB of memory for a duration of 5000 milliseconds. The pricing is determined by the number of workflow requests and the memory and duration consumed by each workflow.
6. **Amazon EventBridge**: There is no estimated cost for Amazon EventBridge as it's not selected, indicating that there are no charges associated with the selected service.
7. **Standard Topics (Amazon SNS)**: The estimated cost is $1.98 per month, based on 100,000 requests and notifications for standard topics. The pricing is determined by the number of requests and notifications sent per month.

Overall, the total monthly cost of Browser Notes is calculated by summing up the individual costs of each selected AWS service. This cost estimation provides an approximate understanding of the recurring expenses associated with running the application on AWS infrastructure in the specified region. Additionally, it's important to note that the actual fees may vary based on factors such as usage patterns, data transfer volumes, and any additional AWS services or features utilized within the application.

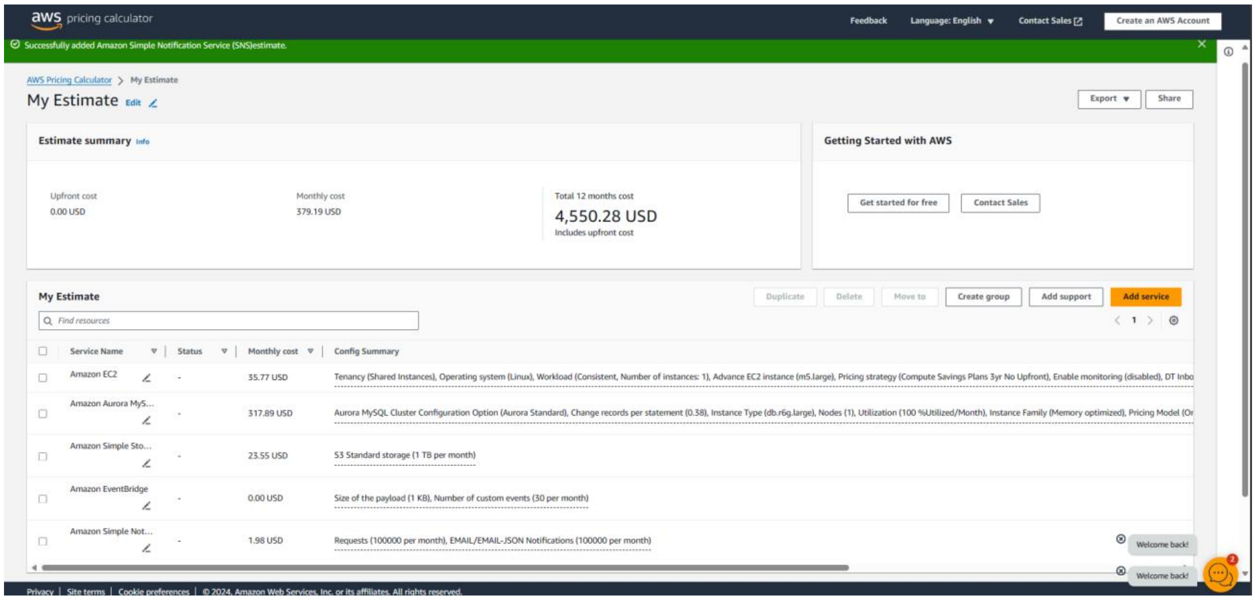
Cost of Alternate/ Expensive Services

Now, to give a comparison, let's calculate the cost estimate of using alternate services in AWS. Let's use **EC2** instead of Lambda, API Gateway, and Step Functions. Instead of DynamoDB we can opt for **AWS Aurora** [13, 1].

I chose to replace AWS Lambda, API Gateway, and Step Functions with **EC2**, specifically the **m5.large** instance type, due to the need for more control over the underlying infrastructure and potential cost savings for consistently high workloads. The **m5.large** instance type offers a balance of compute and memory resources, with **8 GB** of RAM and **2 vCPUs**, which exceeds the memory consumption of the current Lambda configuration. This choice allows us to handle the current request rate of **10 million** requests per month more efficiently and cost-effectively by leveraging EC2's scalable compute capacity while reducing the operational overhead associated with managing multiple serverless services.

Aurora r6.large is a good replacement for DynamoDB due to its balanced performance and scalability, aligning well with the needs of Browser Notes. The **r6.large** instance type offers **2 vCPUs** and **16 GB** of memory, providing ample resources to handle the application's workload. Additionally, Aurora's storage scales automatically up to **64 TB**, ensuring that Browser Notes can accommodate growing data volumes without manual intervention. The choice of Aurora also allows seamless migration of existing data models and queries to one of its supported MySQL or PostgreSQL versions. Moreover, Aurora's built-in high availability and durability features, such as automated failover and continuous backups, enhance data reliability and resilience.

Cost Calculator Link - https://calculator.aws/#/estimate?id=ec0d127d83d9b271afafac7bf314f23d16ff856e.



Upfront cost	Monthly cost	Total 12 months cost
0.00 USD	379.19 USD	4,550.28 USD Includes upfront cost

My Estimate

Service Name	Status	Monthly cost	Config Summary
Amazon EC2	-	35.77 USD	Tenancy (Shared Instances), Operating system (Linux), Workload (Consistent, Number of instances: 1), Advance EC2 instance (m5.large), Pricing strategy (Compute Savings Plans 3yr No Upfront), Enable monitoring (disabled), DT Inbo
Amazon Aurora MyS...	-	317.89 USD	Aurora MySQL Cluster Configuration Option (Aurora Standard), Change records per statement (0.38), Instance Type (db.r6g.large), Nodes (1), Utilization (100 %Utilized/Month), Instance Family (Memory optimized), Pricing Model (Or
Amazon Simple Sto...	-	23.55 USD	S3 Standard storage (1 TB per month)
Amazon EventBridge	-	0.00 USD	Size of the payload (1 KB), Number of custom events (30 per month)
Amazon Simple Not...	-	1.98 USD	Requests (100000 per month), EMAIL/EMAIL-JSON Notifications (100000 per month)

Figure 7 Estimated Cost of choosing ***alternative, expensive*** *AWS Services, Source: [1]*

As we can see, the cost drastically increased **(almost doubled)** when using server architecture. Now let's see what the estimated costs would look like if we deployed this solution in a private cloud.

Reproduce architecture in Private Cloud

To reproduce Browser Notes' architecture in a private cloud, we would need to consider both hardware and software costs. Specifically, in hardware, we would need [13, 1]:

1. **Servers** to host the application and data. The cost of maintaining a server can range from around $1300 to $2000 (Inspired from similar EC2 costs for 1 year).
2. **Storage systems** would take majority of the expenditure as maintaining, optimizing a large storage system (1TB a month) would need excessive care. These can cost around $5000.

3. We would also need **routers, firewall**, and **other networking equipment** that allows for secure data and request flow within the application.
4. Lastly, we need to invest in software that runs the application such as, VMWare, OS, DBMS, and other automation / monitoring tools to manage the application.

Overall, I think the current architecture of serverless backend on a public cloud is a sustainable, and cheap option in the long run due to the extra work offload that involves maintaining, scaling, provisioning, load balancing various metrics of the application. Moreover, public cloud provides the services for a way lot much cheaper price than private cloud due to various factors like multitenancy, etc. If we were dealing with an application that cannot be hosted/deployed on a public cloud for compliance, legislation, data sensitivity, etc., it would be logical to go for the private cloud.

Monitoring

I feel that the **AWS Lambda** service has the most potential to cost the most money. This is because the cost of **AWS Lambda** is directly tied to the number of requests and the duration of each invocation. In **Figure 6**, AWS Lambda accounts for **$99.30** per month, which is the highest among all the services listed. Since Browser Notes relies heavily on AWS Lambda for executing backend functions in response to various triggers, monitoring Lambda usage and performance is crucial to ensure costs do not escalate out of budget unexpectedly. Implementing thorough monitoring and alerting mechanisms for AWS Lambda will help in closely tracking usage patterns, identifying any abnormal spikes in requests or durations, and optimizing resource allocation to control costs effectively. Additionally, monitoring Lambda functions can provide insights into performance bottlenecks and inefficiencies, enabling proactive measures to optimize function execution and minimize costs [1].

Upcoming Features

As I have said earlier, I was very intrigued about introducing some interesting features into this application using AWS services that are new to me. In a note taking application, it would be very convenient for users to have an ability to speak and let the application convert that speech to text and for this I would look into using **AWS Transcribe**. I would also look into using **AWS Translate** to convert the text into various languages for the users' convenience. It would also make sense to add a feature to extract text from scanned documents/images using OCR which is handled by **AWS Textract.** Finally, I would use **AWS Polly** to convert the written notes of a notebook to speech to narrate it.

Apart from the features of the application, I would also love to work on creating this application as a web app, and a mobile app just so that users can transition from one platform to another with ease and without worrying about data retention.

References

[1] "AWS Pricing Calculator," *Calculator.aws*. [Online]. Available: https://calculator.aws/#/estimate. [Accessed: 09-Apr-2024].

[2 "Lucid visual collaboration suite: Log in," *Lucid.app*. [Online]. Available: https://lucid.app/lucidchart/8eed77fb-039b-4961-8945-5e39455c37d4/edit?invitationId=inv_12ae2c2c-254a-485c-a03e-b73cfc34464b&page=0_0. [Accessed: 09-Apr-2024].

[3] "Edge notes - Microsoft Edge addons," *Microsoft.com*. [Online]. Available: https://microsoftedge.microsoft.com/addons/detail/edge-notes/fiigjfolhmeakepmplcmljcnonjggine. [Accessed: 09-Apr-2024].

[4] "Publish in the Chrome Web Store," *Chrome for Developers*, 28-Feb-2014. [Online]. Available: https://developer.chrome.com/docs/webstore/publish. [Accessed: 09-Apr-2024].

[5] *Amazon.com*. [Online]. Available: https://aws.amazon.com/lambda/. [Accessed: 09-Apr-2024].

[6] L. Cherian, "AWS step function state machines," *Medium*, 20-Jul-2023. [Online]. Available: https://medium.com/@leocherian/aws-step-function-state-machines-c7ab2e598aff. [Accessed: 09-Apr-2024].

[7] *Amazon.com*. [Online]. Available: https://aws.amazon.com/dynamodb/. [Accessed: 09-Apr-2024].

[8] *Amazon.com*. [Online]. Available: https://aws.amazon.com/api-gateway/. [Accessed: 09-Apr-2024].

[9] *Amazon.com*. [Online]. Available: https://aws.amazon.com/sns/. [Accessed: 09-Apr-2024].

[10] *Amazon.com*. [Online]. Available: https://aws.amazon.com/eventbridge/. [Accessed: 09-Apr-2024].

[11] *Amazon.com*. [Online]. Available: https://docs.aws.amazon.com/step-functions/latest/apireference/API_StartSyncExecution.html. [Accessed: 09-Apr-2024].

[12] "Api gateway get output results from step function?," *Stack Overflow*. [Online]. Available: https://stackoverflow.com/questions/44041821/api-gateway-get-output-results-from-step-function. [Accessed: 09-Apr-2024].

[13] "Amazon RDS Instance comparison," Vantage.sh. [Online]. Available: https://instances.vantage.sh/rds/. [Accessed: 09-Apr-2024].