Assignment 3: Ark
ECE1779

2020.04.11

Group member:

- Ruixin Huang 1001781565
- Lichen Liu 1001721498
- Baiwu Zhang 1001127540

1 Introduction **1**

2 Motivation **2**

3 Usage **3**

3.1 Archive a web page 3

3.2 Search for an old archive by URL 4

3.3 Search for an old archive by dates 4

4 Architecture **5**

4.1 Lambda Archive Function 5

4.2 Database 5

4.3 Persistent Storage 6

4.4 Infrastructure 7

5 Cost model **8**

6 Conclusion **8**

1 Introduction

In Assignment 3 for ECE1779: Cloud Computing, our team built a web archiving application called *Ark.* The name of 'Ark' is taken from the ship built by Noah to save his family and two of every kind of animal from the Flood. In the Internet era flooded by information, our web application *Ark* aims to preserve the memory of the Internet by providing a place for users to record and retrieve websites of their interest at any given time. What sets *Ark* apart from other Internet archiving services is its screenshot functionality, which will be discussed later.

Ark is primarily built for scalability with AWS technologies, including S3 for storage, Lambda for serverless computation, and DynamoDB for database access. *Ark* can be accessed at https://ahiptn0b0k.execute-api.us-east-1.amazonaws.com/dev.

This document provides a comprehensive review of *Ark*. Section 2 discusses our motivation for this application, as well as how this website will benefit users. In section 3, we provide a walkthrough of *Ark* to demonstrate its usage. Section 4 describes the Ark's architectures and implementation details. Section 5 will illustrate the cost model we have projected if Ark is rolled out to the public.

2 Motivation

With the development of economy and technology, the Internet has become an inseparable part of more people's life more than any time in the past, especially under the current social distancing policy required by COVID-19, where Internet has become the place for people to work, study and express themselves. Digital information, such as websites shared on the Internet, has the glamorous property to be easily distributed and stored. However, with the amount of data presented and the flexibility to modify, history of digital information, if not actively stored, would be easily lost.

To preserve the memory of digital information and archive such information for future records, numerous Internet Archiving sites have been developed, including the famous archive.org. These tools perform well on websites that are mainly composed of texts as they save the HTML resources of the website from a GET request. However, many modern websites are filled with rich media such as images. To provide better mobile experiences, those sites utilize optimization technologies such as lazy loading, which unfortunately results in unloaded images in archiving records as images are loaded on the go.

A better option is to mimic a real user's behaviour to scroll down and capture a complete screenshot of the website of interest. This operation is ideal for an AWS Lambda function to perform for two reasons: 1. Each operation will require a certain amount of computation time to wait for the loading. 2 Each operation could be performed independently in an unsynchronized and scalable way. Thus, we decided to build this functionality with additional user interfaces for our archiving site: Ark.

3 Usage

To use Ark, a user will need to register an account first. A registered user is capable of using Ark for three major purposes: archive a public web page; search for an URL for related archives; search for archives by dates.

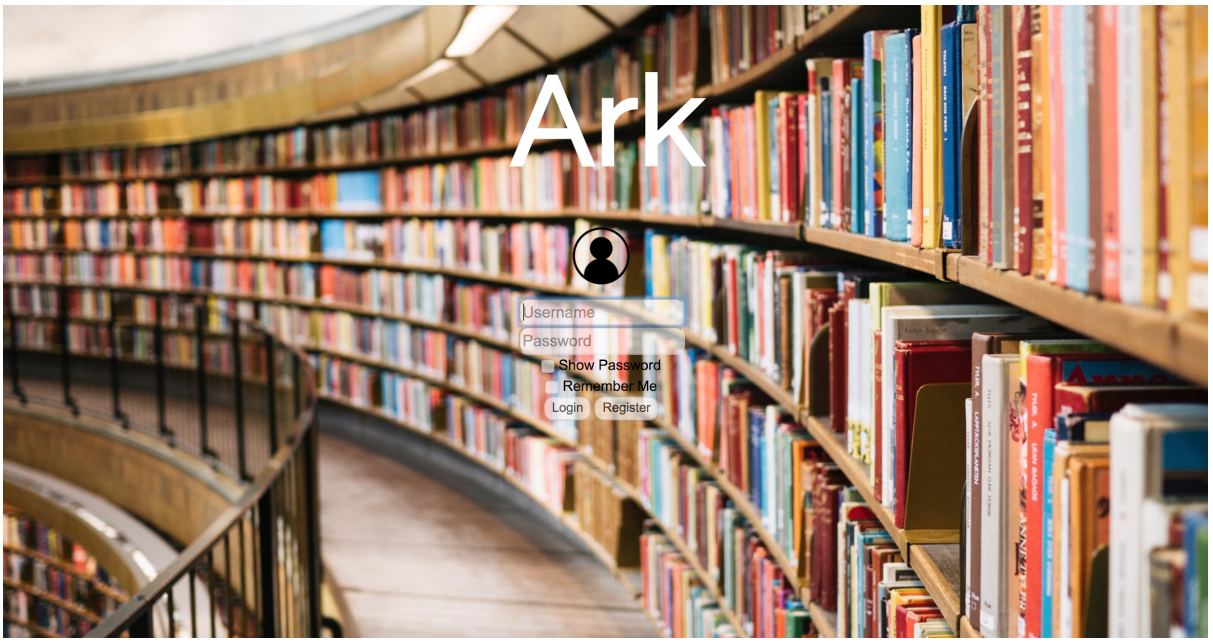

Image 1: User login and registration page

3.1 Archive a web page

A user is able to type in a public URL in the search bar and make an archive request. The request will be queued first and processed immediately when previous requests are finished.

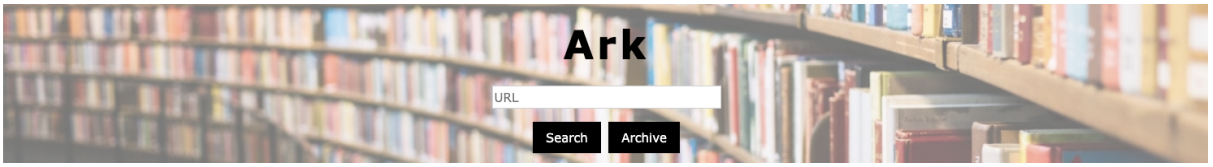

Image 2: Search/Archive bar

During an archive request, the requested URL will be normalized into a standardized HTTP format by our application first, and then Ark will take a snapshot of the website denoted by the URL, along with the time that the snapshot was taken. The user is able to see the snapshot of the web display on our page when the related archive request is completed. The information about an archived request, including the snapshot image, the snapshot date time and the HTTP URL is stored in our cloud database/storage.

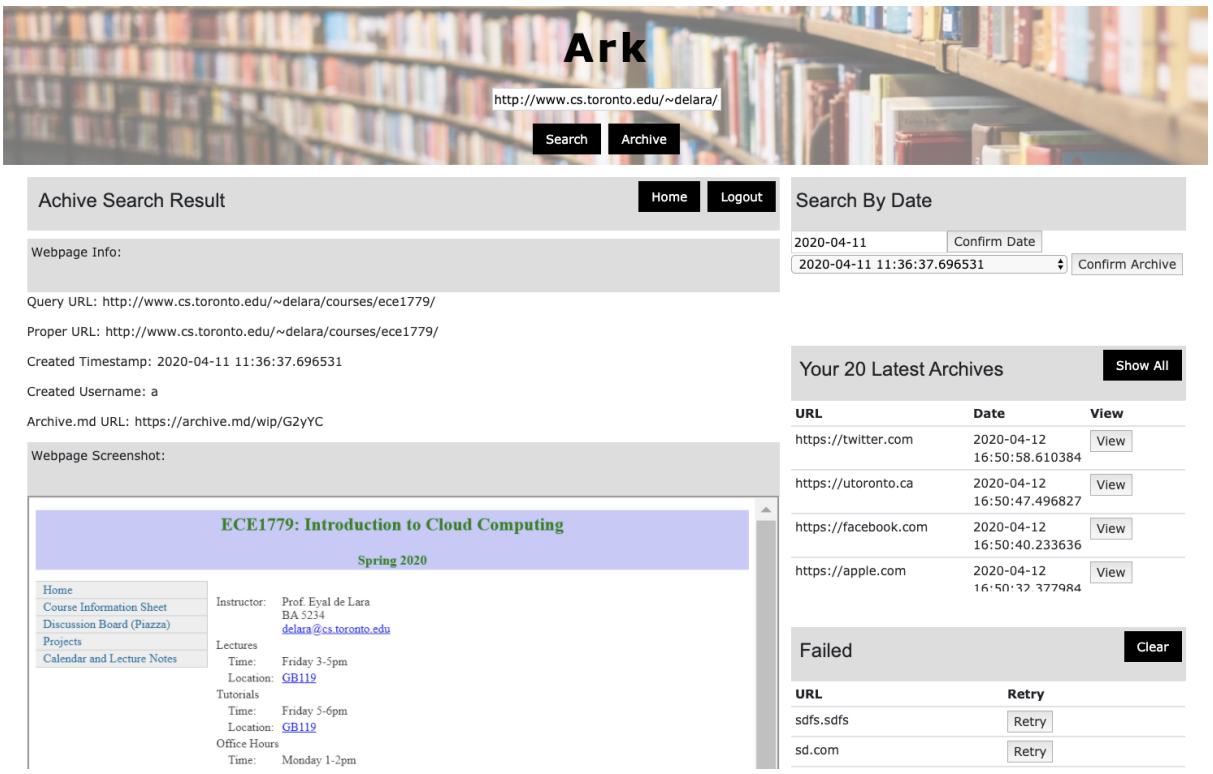

Image 3: An screenshot of an Ark user page.

3.2 Search for an old archive by URL

A user can search for an URL in the search bar, which gives the user the latest snapshot about that URL if it exists. The snapshot will be displayed on our application, along with the date and time when that snapshot was taken.

3.3 Search for an old archive by dates

If a user wishes to see all the archive history of an URL, he can use our browse-by-date functionality. After an archive request or a search request about a particular URL is made, the user can pick a date on the calendar, and only the dates when there was an archive request made against the same URL are selectable. After a date is selected, our application will display a list of selections representing the exact moments of snapshots taken on that date and then the user can click on one of the selections to open that particular snapshot on our website.

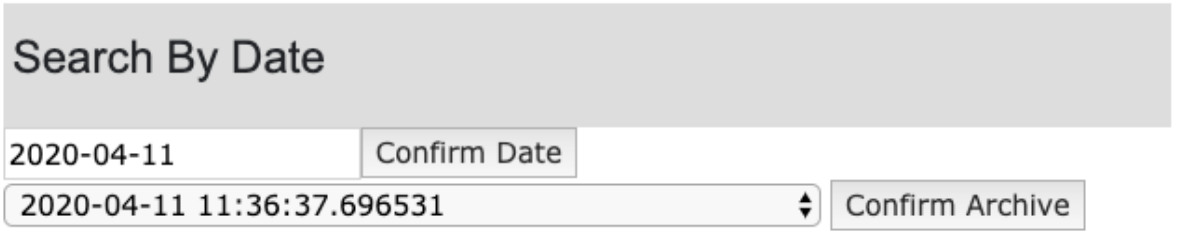

Image 4: Select archive history

4 Architecture

4.1 Lambda Archive Function

The main archiving functionality is implemented on AWS Lambda. By separating it from the Flask server, the archiving requests can be processed asynchronously, which avoids blocking the Flask server from handling other incoming requests. From a high-level-view, the user's single URL archive requesting action is divided into three nonblocking stages: request enqueue, processing and result notification.

Once the user has submitted an archiving request, the request is enqueued into the DynamoDB database as a pending request. Upon any modifications to the database, an event is triggered and effectively activates the lambda archive function. Entering the lambda archive function, the pending request is firstly popped from the pending queue and proceeds to the actual archiving logics that involve a few time-consuming (IO intense) steps. It first tries to fix the user input URL scheme (ie., "Http" and "Https") if it is missing. The fixing is done in an error and trial fashion, where each candidate URL is sent a "GET" request to determine validity. Then, the function tries to normalize the URL, so that URLs can be archived and stored in a consistent format. Finally, the website is taken a full-size screenshot by the serverless-chrome, and all archiving info about the website are stored into the database (DynamoDB) and the persistent storage (S3). If the URL fails to be legalized, then the original archiving pending request will be stored in a failure queue for notifying the user.

The most tricky part for implementing this lambda archive function is the screenshot. The screenshot is done by Selenium, a project that aims to automate browsers, which is backed by Chrome and its driver (chromedriver). To run Chrome on AWS Lambda, which lacks necessary OS support, we used a modified version called serverless chrome. Binaries for Chrome and all related python library codes (including our own) are zipped and then updated to the Lambda. Then the lambda handler manually copies the binaries to the proper location with the proper permission.

4.2 Database

The Ark application uses DynamoDB as its backend database. DynamoDB provides a very flexible structure for storing data, and can easily scale up to keep the performance under a huge amount of traffic and data. Two tables are used for accounts and archives.

The account table keeps username, hashed password, password salt, pending archive request list and failed archive request list. The pending and failed archive request lists are of StringSet type, which can dynamically grow and shrink and can also remove duplicated requests automatically. The primary partition key is the accountId (the username). The account table has been attached to a trigger that is triggered whenever the table is modified.

accountId	passwordHash	passwordSalt	archivePendingRequestList	archiveFailedRequestList
yonghu	134522b92bfc52e275d1127f671bff3ddc581c6327e74ea70057be0cbae8011f	0f2d33f4		
ray3	29906787ea6732637212a3f1ecacd0a2d1989eda991eda50d609374d7d53…	1c253f36		
c	cde344fcfd2c2a164b24b99375072ef420e143a914833cc2705f750074c22d6f	26d193e1		
qq	244500e63e53ffcc070e171dfab23834c366c40c9760386306285c0abf89df33	5f2c7271	{ "apple.com" }	{ "ss" }
p	5d91d3f8af7fb150e20afc1cc3ba6a5ac398a5225e7bae7ebb56f1976f0e5441	8ec0f609		
y	65c84e85c9d54eee8c2840f72de378b2f4cd0a070cd33d512f04549aa9a2b…	9b961c22		
xx	577f20f738978d5992433419a72fc952b43e9b6d59535ee6985a13e7aacb0…	c4e4c7d8		
a	5da7d5b469a250be94d5840ba74c28e40b479848fe3611ce50fd56bc0cdb3…	e5283cef		
b	926a705936de0c584d9449ea958fd3f81fc5eebf646e91d2af7f932952319098	f3d8869a		
user	888a62cbc3454c1e597cd56250a6f27839a59f27a939a8aef7897c4d2b773…	fef3b2d7		

Image 5: Account Table Schema

The archive table keeps all URL archives, including URL, timestamp, username, unique ID, and archiveMD URL. The primary partition key is archiveUrl, and the primary sort key is archiveDatetime. As a single URL can be archived at multiple timestamps with time granularity to 1 millionth of 1 second, using archiveUrl as partition key allows the server to query for all archives on a single URL, while the archiveDatetime can be used for sorting by time. A global secondary index (GSI) is also created, where the partition key is archiveAccountId and the sort key is archiveDatetime. This GSI allows the server to perform efficient queries for all archives created by a certain user, with the ability to be sorted by timestamp. Each archive entry also has a unique ID assigned to be used as an S3 screenshot storage key. Ths unique ID is generated by using UUID. To avoid creating a non-unique ID under the extremely-low likelihood, a trick on the primary key has been used for guaranteeing that. The unique ID itself is inserted as primary range and sort key as primary keys are guaranteed to be unique by the DynamoDB.

archiveUrl	archiveDatetime	archiveAccountId	archiveId	archiveMDUrl (NULL)
archiveId#dff26638-9358-4184-a114-89ff6fff697b	archiveId#dff26638-9358-4184-a114-89ff6fff697b			
archiveId#e4853c2b-5096-4b55-9ba9-b944c6428c5d	archiveId#e4853c2b-5096-4b55-9ba9-b944c642…			
archiveId#ea724578-0868-45ec-ad6c-33e89b9e49df	archiveId#ea724578-0868-45ec-ad6c-33e89b9e…			
archiveId#f721949c-18c6-4c29-9ede-4953ab970993	archiveId#f721949c-18c6-4c29-9ede-4953ab97…			
archiveId#f9149bd8-71c6-4d64-a349-d8174149ec84	archiveId#f9149bd8-71c6-4d64-a349-d8174149…			
archiveId#f95449d9-1431-4bd5-8189-8602540c94d0	archiveId#f95449d9-1431-4bd5-8189-8602540c…			
https://www.google.ca	2020-04-11 10:43:00.746643	a	b7498130-20d7-4904-8171-178a45c5eee1	https://archive.md/wip/y0lvh
https://www.qq.com	2020-04-11 10:43:12.834321	a	d3244084-25ab-4a97-8c66-0b01c360a75b	https://archive.md/wip/og0jZ
https://www.bilibili.com	2020-04-11 10:44:43.306191	a	f7d5a85f-05e7-4c70-8543-1b8f9a98e701	https://archive.md/wip/q6ghI
https://www.google.ca	2020-04-11 10:45:17.538587	a	3151c6f8-9d37-4f73-8eb6-c8dd6ef3f615	https://archive.md/wip/y0lvh
https://www.google.ca	2020-04-11 10:46:23.665380	a	65344e2c-b706-4ad0-b5e0-fa4e6fe0f3a4	https://archive.md/wip/y0lvh
https://www.google.ca	2020-04-11 10:54:38.175826	a	2de513a6-3de1-400d-a5b1-f4fde4aa7cf7	https://archive.md/wip/fBdGZ

Image 6: Archive Table Schema

4.3 Persistent Storage

The Ark application is using AWS Simple Storage Server (S3) as backend persistent storage. Two buckets are created, one for storing website screenshots, while the other one is for storing the static resources for Ark itself. The use-case of S3 storage is fairly simple. The Lambda Archive Function will upload the website screenshot when it processes the archive request.

4.4 Infrastructure

In a project with this scope and complexity in deployment, infrastructure is as important as the code that implements core logic. To minimize dependency the application is separated into three modules: *corelib*, *archivelib* and *ark_app*.

ark_app is the Flask-powered frontend server. If running the application locally, it relies on both *corelib* and *archivelib*. However, once deployed onto AWS lambda via zappa, a framework that eases the deployment of server-less event-driven Python applications, it now only relies on corelib. By eliminating the dependency with *archivelib* (will be explained), the Lambda deployment of ark_app becomes super easy.

corelib is a collection of utilities and AWS query algorithms. It contains useful AWS APIs for performing quick and efficient queries on S3 and Dynamodb. These APIs are created on top of AWS-provided Boto3 library. *corelib* is used by both the Flask server on Lambda and the Lambda archive function.

archivelib is the library that defines algorithms for performing the archiving. *archivelib* is used by the Lambda archive function (if deployed on cloud), but not by the Flask server on Lambda. This is due to its dependency with Chrome and chromedriver backend, which makes setting up AWS Lambda complicated. Fortunately, with the well-designed structuring of the code, the Flask server on Lambda is not affected by the Chrome dependency.

As the application has been divided into three libraries for better dependency, and the two different approaches for AWS Lambda Deployment, and also the two backend services, it is crucial to have a centralized tool for fast management on all these aspects. As a result, a helper script was created to improve the team's productivity. The script can show all contents inside the two DynamoDB tables, the storage usage on S3. It is also able to clear all contents stored in the backend. In addition, it also can update/deploy all local static resources, the zappa-backed-Flask server, and even the Lambda archive function onto S3 and Lambda with a few arguments.

An example to demonstrate how the script has improved the team's productivity is as follows: in order to update the Lambda archive function code, the team member needs to manually replace the modified python files in a zipped package including binary files and python library codes. Then this zipped package needs to be uploaded onto S3 due to its file size, and then update the Lambda function code by the new S3 key. All AWS operations need to be done by using the AWS console. After the helper script is introduced, the team member just needs to type a single line of command to do all the code replacement, packing, uploading and updating automatically. This has significantly improved the team's productivity, so that the team can do more iterations over bug fixings and feature improvements.

5 Cost model

To project future costs of maintaining Ark, we developed a cost model that calculates costs for Lambda function, S3 storage and DynamoDB. Our model is based on the assumption that each user will on average archive 50 URLs, and perform search and display operation for 1000 times during 1 month period.

Lambda: Each archiving function will take 2 seconds with 512MB memory allocation, and all other operations will take 100ms and 128MB in memory.
Cost for Lambda: (50 * 0.5GB * 2s + 1000 * 0.125GB * 0.1s) * 0.00001667 = $0.00104

S3: Each archived screenshot will take on average 2MB in S3 storage.
Cost for S3: 50 * 0.023 * 2 / 1024 = $0.0022246

DynamoDB: DynamoDB charges $1.25 for 1M write request, $0.25 for 1M read request. Each operation will on average take 5 read requests, with archiving operations takes one more for a screenshot. Rows in both DynamoDB tables will have 1KB in storage as an upper bound estimation.

Cost for DynamoDB access: $1.25/1M*50 + $0.25 / 1M * 5 * 1000 = $0.0013125
Cost for DynamoDB storage: 51 * 0.25 / 1024 / 1024 = $0.00001216

Scenario 1: 10 users, 6 months. DynamoDB storage cost is ignored as it's below 25GB/Month free use limit (1KB * 51 * 10 = 510KB):
Total cost: 60 * ($0.00104 + $0.0022246 + $0.0013125) = $0.2746

Scenario 2: 1,000 users, 6 months. DynamoDB storage cost is ignored as it's below 25GB/Month free use limit (1KB * 51 * 1000 = 49.8MB):
Total cost: 6000 * ($0.00104 + $0.0022246 + $0.0013125) = $27.4626

Scenario 3: 1,000,000 users, 6 months. DynamoDB storage cost is incurred:
Total cost: 6000000 * ($0.00104 + $0.0022246 + $0.0013125) + 6 * (1KB * 51 * 1000000 - 25GB)* $0.25 = $27497.45

6 Conclusion

Ark gives users their own spaces of preserving the public information on the internet based on their interest. It is scalable, reliable and cost-effective. Ark makes accessing to past archive history effortless by providing users to search both based on url and time.

An user can issue as many archive requests as they want, and eventually those requests will be processed by Ark and become permanently accessible by that user. The serverless architecture capabilitate Ark to handle massive amounts of requests simultaneously, and its scalability is an inherent advantage provided by AWS lambda solution. The cloud

architecture ensures Ark is always working for our users when they want to document anything in today's instantaneously changing environment of internet information. Most costs of running Ark are only charged when users' requests are being processed, when users do not issue any archiving requests there are no additional costs except the storing costs of their archive history.

Each user has their own library of archives and they can easily look up their own personal collection of archives, which is another characteristic differentiates Ark from the most other web page archiving services currently available. All the information about a user's archive history is kept in the AWS dynamoDB and S3 storage, thus the combination of these two storing strategies enabled a more efficient structure of storing users information and their archive histories.

The future work of Ark both includes enhancement of functional features and performance optimization. The currently envisioned additional features include periodically archiving web pages and search of webpage images by keywords. A user can set up a watch list that contains a list of web pages that he wants to periodically archive. Also a user can search for a webpage when they can not explicitly spell the full http address, but using keywords to find possible archived web pages. It is possible by extracting the body text of the webpage and summarizing key words using language processing techniques.