Why and How We Went Serverless, and How You Can Too

CNI Spring 2021 Virtual Membership Meeting

Yinlin Chen and Bill Ingram
Virginia Tech Libraries
March 15-26, 2021
The Presenters

Yinlin Chen
Digital Library Architect And Assistant Professor
University Libraries, Virginia Tech

William Ingram
Assistant Dean
University Libraries, Virginia Tech
Outline

• Challenges
• Virginia Tech Digital Library Platform
• Methodology
• What we did: Before vs Now
• Conclusion
• Future work
Challenges

- Numerous, web applications with similar stacks stretching resources
- Limited in-house capacity to address performance, resilience, and scaling
- Library-specific software requires training or competing for few experienced library Software developers and Sysops
- The complexity nature to modify or add new feature in the monolithic applications
Methodology

The twelve-factor app

1. Codebase
   One codebase, many deploys

2. Dependencies
   Explicitly declare and isolate dependencies

3. Config
   Store config in the environment

4. Backing services
   Treat backing services as attached resources

5. Build, release, run
   Strictly separate build and run stages

6. Processes
   Execute the app as one or more stateless processes

7. Port binding
   Export services via port binding

8. Concurrency
   Scale out via the process model

9. Disposability
   Maximize robustness with fast startup and graceful shutdown

10. Dev/prod parity
    Keep development, staging, and production as similar as possible

11. Logs
    Treat logs as event streams

12. Admin processes
    Run admin/management tasks as one-off processes
If I have seen further, it is by standing on the shoulders of Giants

- Isaac Newton

Team of 5

Thousands of engineers
Principles of Serverless Design

- Design push-based, even-driven pipelines
- Write single-purpose, stateless functions
- Execute functions on demand and use the resource that needs exactly
- Create thicker and more powerful front ends
- Three Pillars of Observability: Logs, Metrics, & Traces
Monolithic Architecture

- User Interface
- Business Logic Layer
- Data Access Layer
- Database

Serverless Architecture

- User Interface
- User Interface
- API Interface

- Microservices & Managed services
- Purpose-built databases
Multi-Tenant Web Applications
Multi-stage deployments through Dev/Test/Prod Environments

AWS Amplify Console

- **main**
  - https://main.appid.amplifyapp.com

- **uat**
  - https://uat.appid.amplifyapp.com

- **feature**
  - https://feature.appid.amplifyapp.com

AWS Amplify CLI

- **prod**
- **test**
- **dev**

GitHub

Dev Team
Multi-site deployment: Before

- Prod
- Test
- Dev

Ansible Playbooks, Shell scripts, etc.

Multi-site deployment: Now

- No Servers need us to perform maintenance
- Unlimited ephemeral Test & Dev deployments
- Step up deployment and let AWS do the rest
- Initially a small instance is created and let AWS to scale it up and down for us
- … and many many more
Fully Automated CI/CD Pipeline

Build 60

Provision | Build | Test | Deploy | Verify

Domain: https://vtulp-demo.cloud.lib.vt.edu
Source repository: https://github.com/VTUL/dlp-access/tree/demo

Started at 3/2/2021, 2:18:21 PM
Last commit message: add archive permission

Build duration: 18 minutes 32 seconds

All Cypress specs completed: 27 spec(s) passed

Spec name | Number of tests | Total duration | Video
---|---|---|---
About link | 2 passed | 00:05 | Download artifacts to see this video.
Displays and updates browse collections page configurations | 8 passed | 00:35 | Download artifacts to see this video.
Update attribute and change it back | 5 passed | 00:17 | Download artifacts to see this video.
Every Single Feature (Change) has a Preview
Before

Website performance

There are other things that we cannot compare the performance, because we can only do those in AWS.

Now

Resource Usage

<table>
<thead>
<tr>
<th>Page</th>
<th>IAWAv1</th>
<th>IAWAv2</th>
<th>Percent Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index Page V1</td>
<td>796 ms</td>
<td>766 ms</td>
<td>4% faster</td>
</tr>
<tr>
<td>Index Page V2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Item Browse Page V1</td>
<td>1.35 s</td>
<td>882 ms</td>
<td>35% faster</td>
</tr>
<tr>
<td>Item Browse Page V2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Collection Browse Page V1</td>
<td>926 ms</td>
<td>754 ms</td>
<td>20% faster</td>
</tr>
<tr>
<td>Collection Browse Page V2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Item Page V1</td>
<td>1.01 s</td>
<td>861 ms</td>
<td>16% faster</td>
</tr>
<tr>
<td>Item Page V2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
We have 11 collections of images need to be processed.

We only have 1 spare server for this, we will do 1 collection a time.

It looks like we need 7 days to process one set of images, is it OK?

That take too long, can you provision servers in AWS and do all at once?

Run 11 servers for a week for this… seems expensive and hard to manage them.

- Processed 11 collections of images in 2 days
- Expect to process 20, 30, … collections of images will still in 2 days
- There are still space to optimize microservices and make it even faster
Infrastructure as Code (DevOps)

SubmitSteps:
Type: AWS::Serverless::Function
Properties:
  CodeUri: fixitycheck/
  Handler: steps.lambda_handler
  Runtime: python3.8
  Timeout: 30
  Layers:
  - !Ref SharedUtils
Policies:
  - StepFunctionsExecutionPolicy:
    StateMachineName: !Ref StateMachineName
Events:
  FixitySOSEvent:
    Type: SQS
    Properties:
      Queue: !GetAtt FixitySqsQueue.Arn
      BatchSize: 10

Environment:
Variables:
  Region: !Ref Region
  StateMachineArn: !Sub "arn:aws:states:${Region}

FixitySqsQueue:
Type: AWS::SQS::Queue

FixitySNS:
Type: AWS::Serverless::Function
Properties:
  CodeUri: fixitycheck/
  Handler: sns.lambda_handler
  Runtime: python3.8
  MemorySize: 2048
  Timeout: 120
  Layers:

One click deploy
Monitor Metrics for Applications in Near Real-time

- Requests (sum)
  - Total: 95
- 4xx errors (sum)
  - Total: 10
- Time to first byte (average)
  - Average: 2.31 seconds
- Data transfer ($sum)
  - Total: 1.7 MB
- 5xx errors (sum)
  - Total: 1
Cost: Before

- Add one server, plus the cost
- Plan server capacity each year
  - Overestimation: shortage is even worse, so would rather purchase more than we need (Wasting resource is fine, we already paid money, but really?)

Cost: Now

- Pay what we use
  - Use it well than pay less
  - Pay 0 when services are idle
- Some (invisible) cost are vanished
  - Server maintenance cost
  - Labor, time, electric, etc.
Conclusion

- AWS services just like Lego blocks, we pick and choose blocks and build powerful, scalable applications that suit our business.
- With serverless, we write less code, fasten development speed, and enable creativities and innovations.
- Applications we built are more resilient, secure, scalable, and cost-efficient than in a traditional, server-based environment.
- AWS communities are larger than any software communities in the libraries.

Image source: Lego.com
Future Work

• New features and services
  – AI/ML/DL as a service
  – Large-scale text and data mining
  – Digital preservation and the management

• Continue performance tuning and cost optimization of cloud-native services

• Continue refactoring of legacy applications to AWS

• Collaborate with other institutions on the Serverless Cloud
Thank you!

Virginia Tech Digital Libraries
Cloud-native, multi-tenancy digital library software
https://github.com/vt-digital-libraries-platform

- **aws-batch-iiif-generator**
  A project that demonstrates the use of AWS Batch to create IIIF tiles and manifests. It is used for VTDLP Derivative Service
  - Python

- **DLPServices**
  VTDLP Services

- **resolution-service**
  VTDLP Resolution Service
  - Python

- **dlp-access**
  DLP Access Website: A Multi-Tenancy Serverless web application
  - JavaScript

- **NOID-mint**
  Mint Nice Opaque IDentifiers
  - Python

- **FixityService**
  VTDLP Fixity Service: Regularly scheduled fixity checks and comparisons to the files stored in the AWS S3
  - JavaScript