Accenture Cloud Platform
Serverless Journey

Tom Myers, Sr. Cloud Architect, Accenture Cloud Platform
Matt Lancaster, Lightweight Architectures Global Lead

November 29, 2016
The common denominator of all platforms is that they facilitate interactions between distinct groups (the two “sides”) that want to interact with and need each other.*

What is the Accenture Cloud Platform?

Accenture Cloud Platform (ACP) offers clients an on-ramp to the cloud.

ACP provides governance, analytics, and services to support the transition to and provide ongoing support of their cloud journey.

ACP services enable clients on one side, and Accenture solution providers on the other.

ACP has transitioned over time from on-premises data center, to cloud virtual data center, and is now transitioning to serverless platform.
How Does ACP Support its Ecosystem?

- Accenture Business Groups
- Development Process
- AWS
- Service Catalog
- Clients
ACP is fundamentally a collection of services coordinated by an orchestrator and ordered via service catalog.

A microservices architecture – services orchestrated and interacting via APIs – is well-suited for such an offering.

A serverless design can optimally host highly available and scalable microservices.
Why AWS?

- The serverless options available from AWS are the most mature from any cloud provider. Scalability, availability, reliability, performance and other architectural concerns are well-addressed in AWS.
- A number of good tools exist to design and deploy complex microservices to the AWS Cloud service environment.
- The Accenture AWS Business Group is our 1-year old partnership with Amazon Web Services, to bring clients the power of cloud through AWS.
ACP Ring Model

Ring 0
- Core Services
- APIs to support Ring 1+
- Continuous Delivery

Ring 1
- Applications/ Services
- Rely on Ring 0
- Agile Releases

Ring 2
- Client Applications/ Services
- Rely on Rings 0 & 1
- Client Control
Serverless Architecture
Serverless Architecture

DNS Owned by Enterprise IT Dept.
Serverless Architecture

Global CDN
Lambda execution with SNS, Redis, Amazon Kinesis for communication
Serverless Architecture

DynamoDB, Amazon Redshift, Elasticsearch for persistent storage
Serverless Architecture
## Serverless Architecture

<table>
<thead>
<tr>
<th>Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>• API Gateway</td>
</tr>
<tr>
<td>• Lambda</td>
</tr>
<tr>
<td>• Amazon ElastiCache for Redis</td>
</tr>
<tr>
<td>• DynamoDB, Amazon Redshift</td>
</tr>
<tr>
<td>• Amazon Kinesis &amp; DynamoDB Streams</td>
</tr>
<tr>
<td>• Elasticsearch</td>
</tr>
<tr>
<td>• SNS Topics/Subscriptions &amp; SQS</td>
</tr>
<tr>
<td>• CloudWatch Logs</td>
</tr>
<tr>
<td>• CloudFront/S3</td>
</tr>
</tbody>
</table>

Flexible design allows components to be swapped

<table>
<thead>
<tr>
<th>Accounts</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Rings</td>
</tr>
<tr>
<td>– Core</td>
</tr>
<tr>
<td>– Apps</td>
</tr>
<tr>
<td>• Lifecycle</td>
</tr>
<tr>
<td>– Sandbox</td>
</tr>
<tr>
<td>– Dev</td>
</tr>
<tr>
<td>– Test</td>
</tr>
<tr>
<td>– Stage</td>
</tr>
<tr>
<td>– Prod</td>
</tr>
</tbody>
</table>

10 accounts with cross-account roles Core ⇔ Apps
DevOps Pipeline

- Node
- Python
- APEX
- Custom Framework
- Jenkins
- Accenture DevOps Platform
- AWS
- State Files
DevOps Model

- **Terraform + Apex**
  - CloudFormation didn’t fully support API G/W when we needed it
  - Custom Framework (code consistency, name checking, etc.)
    - Solved a lot of the initial inherent complexity in Lambda and API G/W deployment
    - Higher-level abstraction for all infrastructure with simple config files to generate Terraform
    - Supports cross project & AWS account Terraform dependencies and stores state on S3

- **Accenture DevOps Platform**
  - We currently use BitBucket, Jenkins, Sonar
  - Also supports Nexus, Kibana, Selenium, etc.
  - Deploys via ACP as an automated stack
  - [https://github.com/Accenture/adop-docker-compose](https://github.com/Accenture/adop-docker-compose)
Challenges

• CloudFormation
  – Occasionally gets stuck requiring AWS Support to resolve
  – Monitoring status and completion of stacks
  – Diagnosing and fixing errors not always straightforward
  – Verbose JSON format for large stacks at times unwieldy (even with Troposphere) - YAML a welcome addition!!

• API Gateway
  – Limited initial support in CloudFormation
  – Overhead of creating large API Gateway definitions
  – HTTP method mappings complex
  – Recent enhancements such as Swagger import and passthrough will help a lot!

• Need to explicitly define all infrastructure, which gets complex at scale
  – IAM Roles, Policies, Lambdas & SNS/Stream mappings, etc.
  – Handled by custom framework
• Lightweight frameworks can allow greater flexibility (e.g., AWS account per environment/stage)
• Pick a Lambda runtime – lots will work, but less reuse opportunities
• Operability and security must be considered and designed for
• Plan for upgrades ahead of time (e.g., ElasticSearch)
• Pay attention to limits (e.g. CloudWatch logs writes, batch size, and message dates in batch events, DynamoDB read/write)
• Define Lambda logging conventions and how these fit into the monitoring components (ElasticSearch/Kibana)
• Being an early adopter comes with risk and often requires workarounds
Benefits

• Deploy in minutes (no delay waiting for other teams)
  – 10 deploys per day (or more)
  – Deployment triggers on code-commit
• Bug fixes deploy more quickly with isolated functions and CD
• Use of custom framework allows for rapid creation of complex components that use multiple AWS services
• Swagger provides rapid deployment of documentation
• API Gateway simplifies API versioning and deployments
• Write functions in Lambdas – design for parallel execution
  – Small, idempotent routines
  – Built in restart, retry
  – Cache state (Redis, ElastiCache)

• Surface Lambdas with API Gateway
  – User Custom Domain Names and pathing to gather APIs together
  – This enables independent build with a consistent experience

• Communicate with services
  – SNS to send messages
  – Kinesis to pass data

• Target storage to ElasticSearch
  – Search queries are probably what you need anyway

• Persist data with Dynamo DB
  – Keep a date key (epoch) to partition and purge data
  – Send backup to low cost storage
Hybrid Integration Example
Operations and Monitoring Example
## The Future of Serverless Applications

<table>
<thead>
<tr>
<th>Current State</th>
<th>Future State</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Heavy use of the console to manage different services</td>
<td>• Use a framework to manage resources</td>
</tr>
<tr>
<td>• Lambda is over-utilized</td>
<td>• Use platform services (e.g., velocity templates, Kinesis streams) to</td>
</tr>
<tr>
<td>• Shoving existing code into Lambda</td>
<td>supplement Lambda</td>
</tr>
<tr>
<td>• No real system health visualizations</td>
<td>• Develop applications to ensure minimal code deployments</td>
</tr>
<tr>
<td></td>
<td>• Use open source time series visualization tool for operational readiness</td>
</tr>
</tbody>
</table>
Don’t Use Lambda for *Everything*

- Consider the capabilities available within AWS first, and use Lambda only when necessary
- Take advantage of Apache Velocity for transformations in the API Gateway instead of using a Lambda function
- If data ingestion is required, consider using an AWS Proxy to Kinesis for greater throughput and scalability
- Most AWS services expose a RESTful endpoint for an API Gateway AWS Proxy interaction
- Use Lambda where you need to execute business logic
Use Velocity Instead of Lambda for Transformations
Use Streams to Optimize Throughput

- Using DynamoDB or Kinesis Streams allows bulk processing of data
- Tuning the EventSourceMapping BatchSize to just under 100 ms ensures you’re getting the greatest value from your Lambda functions
Be Nice to Your Lambda

• Use bundlers like Webpack or Browserify to ensure only required dependencies are present in deployed Lambda functions.
• Ensure that no dev dependencies are being included in your packaged Lambda functions.
• Lambda containers have the ability to persist across invocations. Make sure to group multiple related methods in the same handler file to increase invocation rate, in order to avoid cold boot delays.
• Instantiate persistent objects outside of methods to increase Lambda execution performance during container reuse conditions.
Don’t Bother with CloudWatch for Operations Metrics

- CloudWatch is not a realistic choice for enterprise operations metrics
- Dedicated time-series systems like Prometheus.io are a far better choice for near real-time operations dashboards
Story Time!
Alexia – An Alexa NodeJS Framework

- Written in ES6
- Speech assets generation
- Cards support
- Works with Lambda, Hapi, Express
- Build in state machine
- Starter kit
const alexia = require('alexia');
const app = alexia.createApp();
app.intent('HelloIntent',
'Hello', () => {
    return 'Hello from Alexia app';
});
Demo Architecture

- Echo
- Alexa Skills Kit
- Lambda Handler
- API Gateway
- GoodRx Catalog API
- NIH Interaction API
“Aspirin has been added to your cart. But I’ve found interactions between your medications. Please consult with your doctor or pharmacist”
Case Study – Pharmacy Advisor

Showcase modern architecture principles and leverage from Amazon Web Services
Demo Conversation

“Add Ibuprofen to my cart.”

“Ibuprofen has been added to your cart.”

“Add Aspirin to my cart.”

“Aspirin has been added to your cart. But I’ve found interactions between your medications. Please consult it with your doctor or pharmacist.”
Visit us at the Accenture booth #829

High performance. Delivered.

Tom Myers
thomas.w.myers@accenture.com
@TomMyersACP

Matt Lancaster
matthew.d.lancaster@accenture.com
@urnamma
Remember to complete your evaluations!