AWS Lambda

In the Cloud and on Devices

Marc Brooker, Senior Principal Engineer, AWS Serverless

6 June 2017
About Me

• 9 years at Amazon Web Services

• Joined in 2008, in Cape Town, South Africa

• Worked on EC2, EBS, IoT and Lambda

• PhD in EE from University of Cape Town
  • Cool stuff with passive radar and simulation
I’m Very Excited To Be Here
Encoding, Fast and Slow:  
Low-Latency Video Processing Using Thousands of Tiny Threads

Sadjad Fouladi $, Riad S. Wahby $, Brennan Shacklett $,  
Karthikeyan Vasuki Balasubramaniam φ, William Zeng $, Rahul Bhalerao φ,  
Anirudh Sivaraman $, George Porter φ, Keith Weinstein $  

Stanford University $, University of California San Diego φ, Massachusetts Institute of Technology $φ$

Abstract

We describe ExCamera, a system that can edit, transform, and encode a video, including 4K and VR material, with low latency. The system makes two major contributions.

First, we designed a framework to run general-purpose parallel computations on a commercial “cloud function” service. The system starts up thousands of threads in seconds and manages inter-thread communication.

Session relies on temporal correlations among nearby frames. Splitting the video across independent threads prevents exploiting correlations that cross the split, harming compression efficiency. As a result, video-processing systems generally use only coarse-grained parallelism—e.g., one thread per video, or per multi-second chunk of a video—frustrating efforts to process any particular video quickly.

In this paper, we describe ExCamera, a massively parallel, cloud-based video-processing framework that we
### Teraflops and microservices

- **468 commits**
- **38 branches**
- **7 releases**
- **7 contributors**
- **Apache-2.0**

**Branch:** master

**shivaram committed on GitHub:** Merge pull request #133 from pywren/empty-list

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
<th>Last Updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>pywren</td>
<td>Handle empty input list in map</td>
<td>a month ago</td>
</tr>
<tr>
<td>tests</td>
<td>Mark numba test as flaky with 3 max runs</td>
<td>a month ago</td>
</tr>
<tr>
<td>.gittignore</td>
<td>Ignore IntelliJ idea file.</td>
<td>3 months ago</td>
</tr>
<tr>
<td>.travis.yml</td>
<td>Merge pull request #99 from pywren/serialization-bugs</td>
<td>2 months ago</td>
</tr>
<tr>
<td>CONTRIBUTORS.md</td>
<td>added echo for install; updated contributors</td>
<td>3 months ago</td>
</tr>
</tbody>
</table>
What I’m Talking About

AWS Lambda

AWS Greengrass
Serverless overview
Let’s take a look at the evolution of computing
## Each progressive step was better

<table>
<thead>
<tr>
<th>Higher utilization</th>
<th>Trade CAPEX for OPEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Faster provisioning speed</td>
<td>More scale</td>
</tr>
<tr>
<td>Improved uptime</td>
<td>Elastic resources</td>
</tr>
<tr>
<td>Disaster recovery</td>
<td>Faster speed and agility</td>
</tr>
<tr>
<td>Hardware independence</td>
<td>Reduced maintenance</td>
</tr>
<tr>
<td></td>
<td>Better availability and fault tolerance</td>
</tr>
</tbody>
</table>

### Virtual Servers in the Cloud

![Diagram of virtual servers in the cloud](image-url)
But there are still **limitations**

- Still need to administer virtual servers
- Still need to manage capacity and utilization
- Still need to size workloads
- Still need to manage availability, fault tolerance
Evolving to **serverless**
Serverless means..

- Simple but usable primitives
- Scales with usage
- Never pay for idle
- Availability and fault tolerance built in
# Building blocks for serverless applications

<table>
<thead>
<tr>
<th>Compute</th>
<th>Storage</th>
<th>Database</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="https://example.com/aws-lambda" alt="AWS Lambda" /></td>
<td><img src="https://example.com/amazon-s3" alt="Amazon S3" /></td>
<td><img src="https://example.com/amazon-dynamodb" alt="Amazon DynamoDB" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>API Proxy</th>
<th>Messaging and Queues</th>
<th>Analytics</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="https://example.com/amazon-api-gateway" alt="Amazon API Gateway" /></td>
<td><img src="https://example.com/amazon-sqs" alt="Amazon SQS" /></td>
<td><img src="https://example.com/amazon-kinesis" alt="Amazon Kinesis" /></td>
</tr>
<tr>
<td><img src="https://example.com/amazon-sns" alt="Amazon SNS" /></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Orchestration and State Management</th>
<th>Monitoring and Debugging</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="https://example.com/aws-step-functions" alt="AWS Step Functions" /></td>
<td><img src="https://example.com/aws-x-ray" alt="AWS X-Ray" /></td>
</tr>
</tbody>
</table>

---

*Note: The images are placeholders and should be replaced with actual images from the serverless services.*
Customers

- airbnb
- AdRoll
- ALT/s
- asurion
- Atlassian
- Benchling
- BUSTLE
- CITRIX
- Coca-Cola
- COMCAST
- Discovery Channel
- Expedia
- Experian
- FireEye
- instacart
- LexisNexis
- Localytics
- McGraw Hill
- MLB
- Nextdoor
- Scholastic
- The Seattle Times
- The Honest Company
- Thomson Reuters
- vevo
- VidRoll
- Zapproved
- Zillow
- 23andMe
Data and IoT
Most machine data never reaches the cloud

Medical equipment

Industrial machinery

Extreme environments
Why this problem isn’t going away

Law of physics

Law of economics

Law of the land
Laws of Physics

Speed Of Light in Fiber
$2 \times 10^8 \text{ ms}^{-1}$

(or 33% slower than a vacuum)
(or 5ms per 1000km)
Law of Economics

Law of economics
Law of the Land
AWS Greengrass extends AWS onto your devices, so they can act locally on the data they generate, while still taking advantage of the cloud.
AWS Greengrass Overview
Greengrass Components

Greengrass is software, not hardware (you bring your own)

2 Components that work together:
- Greengrass Core
- IoT Device SDK
AWS Greengrass Core (GGC)

A runtime offering local Lambda execution, pub/sub messaging, device shadows, and security.
AWS Greengrass Core (GGC)

- Min single-core 1 GHz
- Min 128 MB RAM
- x86 and ARM
- Linux (Ubuntu or Amazon)

- The sky is the limit
Devices work together locally

An AWS Greengrass group is a set of cores and other devices configured to communicate with one another.
Devices work together with the cloud

AWS Greengrass works with AWS IoT to maintain long-lived connections and process data via the rules engine.

Your Lambda functions can also interact directly with other AWS services.
Shadows

JSON documents that represent state of your devices and Lambda functions

Define them however is logical to you—a car, an engine, a fleet

Sync to the cloud or keep them local
Shadows – What you can do

- Device state (current and desired)
- Granular device state (only synched to the cloud for debug)
- Lightweight configuration
Messaging

Local MQTT pub/sub messaging

Define subscriptions between publishers and subscribers

Apply MQTT topic filters
Security

Mutual auth, both locally and also with the cloud

Certificate on your devices can be associated to SigV4 credentials in the cloud

You can directly call any AWS service from AWS Greengrass
Local Lambda

Lambda functions are event-driven compute functions

With AWS Greengrass you can write Lambda functions in the cloud and deploy them locally
Local Lambda

AWS Greengrass runs Lambda functions written in Python 2.7

Invoke Lambda functions with messaging and shadow updates
Local Lambda—What you can do

- Command and control
- Offline operation
- Data filtering and aggregation
- Get smarter over time
Local Lambda—What you can do

- Command and control
- Offline operation
- Data filtering and aggregation
- Get smarter over time
Lambda and Greengrass Together
Example: Combining Readings From Multiple Sensors

- Temperature
- Humidity
Example: Combining Readings From Multiple Sensors
Fleet of Sensors

AWS Greengrass Core

AWS IoT

AWS Lambda

Amazon SNS

Operator

Amazon DynamoDB
Law of Economics
Fleet of Sensors

AWS Greengrass Core

AWS IoT

AWS Lambda

Amazon SNS

Operator

Amazon DynamoDB

Operator
Lambda

MQTT

Topic

IoT

Sensor

Shadow to Record

High Water Mark

Shadow to Record

Last Publish Time

MQTT Topic

Publish to AWS IoT

AWS

IoT
Loading and Parsing A Shadow

def current_record_data(cls):
    try:
        # The get_thing_shadow API loads a shadow
        shadow = iot_client.get_thing_shadow(thingName=cls.name)['payload']
        record_shadow = json.loads(shadow)
        version = record_shadow['state'].get('version', 0)
        ...
        return record_value, last_push_time, version
    except ShadowError as e:
        # The shadow doesn't exist yet, so return some default values
        if str(e).startswith('Request for shadow state'):
            return None, None, 0
        else:
            raise e
def update_shadow(self):
    # Loop until shadow state update successfully occurs
    while True:
        current_shadow, last_push, version = self.current_record_data()

        if not self.should_update(self, current_shadow):
            return False

        new_shadow = self.new_shadow(version)
        if self._update_shadow(new_shadow):
            return True

    # Sleep with jitter
    time.sleep(random.random())
Publish an MQTT Message to the Cloud

def push_record(self):
    # Loop until shadow state update is successful
    while True:
        record, last_push_time, version = self.current_record_data()

        if not self._should_push_record(last_push_time):
            # Another Lambda has already posted. Job done!
            return False

        # Try to update the thing shadow
        if self._try_update_shadow(version):
            # If that works, publish the message
            iot_client.publish(topic=self.record_reporter_topic, payload=str(record))
            return True

    # Sleep for a short while, with jitter
    time.sleep(random.random())
Laws of Physics
Why Greengrass is important

**Embedded developer**

- Program devices with modern languages, deployment APIs, and workflows

**Cloud developer**

- Cloud-based development that adds value to data that never reaches the cloud
Thank You!