Elasticity

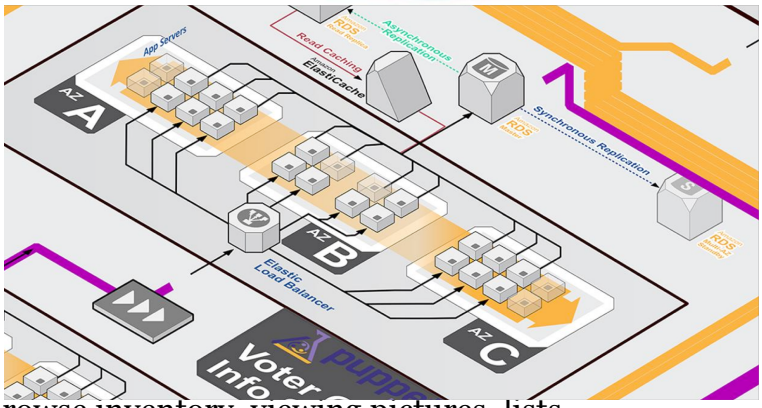
15-719/18-709: Advanced Cloud Computing

George Amvrosiadis
Greg Ganger
Majd Sakr

Advanced Cloud Computing Elasticity Readings

- Req'd Ref 1: "Dynamically Scaling Applications in the Cloud." Luis M. Vaquero, Luis Rodero-Merino, and Rajkumar Buyya. ACM SIGCOMM Computer Communications Review (CCR), v 41, n 1, 2011. http://www.sigcomm.org/ccr/papers/2011/January/1925861.1925869
- Opt Ref 2: "Jockey: guaranteed job latency in data parallel clusters." Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. 7th ACM European conf. on Computer Systems (EuroSys '12), 2012. http://doi.acm.org/10.1145/2168836.2168847
- Opt Ref 3: "Split/Merge: System Support for Elastic Execution in Virtual Middleboxes." Shriram Rajagopalan, Dan Williams, Hani Jamjoom, Andrew Warfield. 10th USENIX Symp. on Networked Systems Design and Implementation (NSDI '13), 2013. https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final205.pdf
- Opt Ref 4: "ElasTraS: An elastic, scalable, and self-managing transactional database for the cloud."Sudipto Das, Divyakant Agrawal, Amr El Abbadi. ACM Trans. Database Syst. 38, 1, Article 5, 2013. http://doi.acm.org/http://dx.doi.org/10.1145/2445583.2445588

Web Servers



- The “killer app”
- Online retail service
- Most cycles spent on web page rendering
- Customers use http to browse inventory, viewing pictures, lists, technical specs, price comparisons before requesting a product be purchased and shipped to them
- Number/”size” of server machines needed to keep up with client load fluctuates with customer interest, social media exposure, inventory and marketing changes, time of day, etc.
  - BUT, want to pay only for needed servers!

Basic web service parallelization: “Load Balancer”

- Incoming stream of independent user requests broken into multiple separate per-server-machine streams
- Done with a component generally called a “load balancer”
  - partitions requests among server machines
  - not necessarily “balanced”, but called “load balancer” anyway

Lets remember how a web request works

http://www.cs.cmu.edu/~15719/

- 1. Client translates DNS name (www.cs.cmu.edu) to server IP address
  - use DNS to get list of one or more IP addresses
- 2. Client opens TCP connection to a server IP address
  - use default port number, if none specified in URL
  - usually use first IP address in list, if multiple
- 3. Client sends "GET" HTTP request (e.g., for "~15719/") via connection
  - sent as bytes in TCP stream, formatted as per protocol

Load Balancing Approaches #1

- 1. DNS load balancing
  - have one IP address per server machine
  - have DNS reorder the list for each client asking for translation of name
  - expect each client to try IP addresses in list order
  - PLUS: out of band of actual TCP/HTTP requests
    - can distribute arbitrary bandwidth (not limited by bandwidth of a router)
  - MINUS: takes a long time to change
    - Tells client a binding of a name to an IP (list), which makes dynamic changing of the binding hard, or at least up to the client

Load Balancing Approaches #2

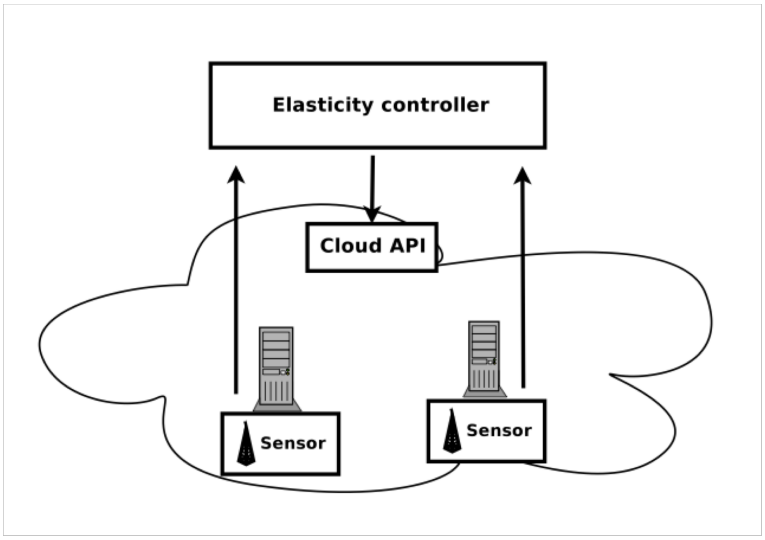
- 1. DNS load balancing
  - have one IP address per server machine
  - have DNS reorder the list for each client asking for translation of name
  - expect each client to try IP addresses in list order
- 2. Having Router distribute TCP connection open packets
  - have one IP address for entire web service, which goes to a “load balancing” router
  - Have that router spread SYN packets sent to web server among different server machine
    - Recall: client opens TCP connection by sending SYN packet
  - PLUS: router doesn’t have to think too much
  - MINUS: decision is for the entire life of the connection which may be too long to do good balancing

Load Balancing Approaches #3

- 1. DNS load balancing
  - have one IP address per server machine
  - have DNS reorder the list for each client asking for translation of name
  - expect each client to try IP addresses in list order
- 2. Having Router distribute TCP connection open packets
  - have one IP address for entire web service, which goes to a “load balancing” router
  - Have that router spread SYN packets sent to web server among different server machine
    - Recall: client opens TCP connection by sending SYN packet
- 3. Having Router distribute individual requests embedded in connections
  - Again have one IP address for entire web service, going to router
  - Have router be endpoint for TCP connection and interpret the bytes
    - Thus, it can make routing decisions for specific requests
  - PLUS: most dynamic approach
  - MINUS: requires the most processing in the router

How is elasticity provided to web servers?

- Abstractly, an elasticity controller monitors allocated machines
- When overloaded, add load capability
- When under used, reduce load capability
- Adding/reducing done by cloud framework, as directed by elasticity controller



Elasticity: Scale-out or Scale-up?

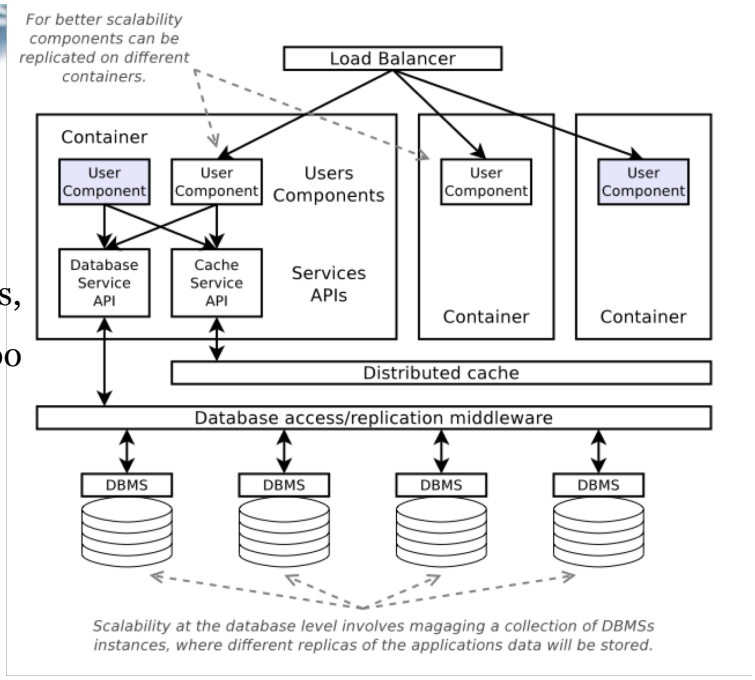
- Horizontal scaling (Scale-out)
  - Adding more (usually identical) instances
  - Most use of elasticity is done this way
- Vertical scaling (Scale-up)
  - Resizing the resources allocated to an existing instance
  - But, does your (IaaS) OS accept and utilize more resources on the fly?
    - More network bandwidth is probably easy
    - More memory is harder but possible (eg. VM ballooning)
    - Changing cores is even harder
  - PaaS containers might hide resource representation
    - Eg. could provide more MapReduce slots in same machine

Elasticity Controller Capabilities

- User takes monitoring offered, defines rules for when to take actions (models the appl.)
- Monitoring
  - Monitor resource usage at specified instances, threshold individually or in total
  - Monitor request sequence, looking for patterns to reconfigure for predicted load
- Triggering
  - Trigger on simple conditions, thresholds, on monitored instances, request stream
  - Trigger on schedule (simple prediction)
  - Trigger on complex formula of many monitored instances (a model of overall service quality)
- Actions:
  - Launch single instances, or identical instances, or modify existing instances
  - Execute sequence of launches or modifications according to a dependency graph or workflow
  - Execute programs that implement a more abstract action (launch and configure multi-tier service)
- If a system is composed of a scalable set of processes feeding/interacting with another scalable set of processes implementing a very different function, are they monitored and scaled independently or is their elasticity managed by one elasticity controller?
  - How good are your models for how you application actually depends on workload and resources?

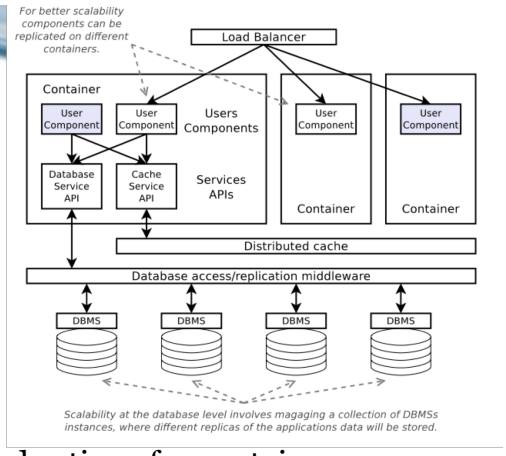
Two-tier services

- Most cycles in web servers, but want to take orders too
- Originally order taking wasn't even in cloud
  - Just send message
- Now frontends (web) & backends (database)



- Elasticity is certainly possible in IaaS – but database scaling is not simple replication of identical web server, so user scripts complex
  - Databases involve state and distributed DBMSs involve consistency issues

Two-tier web server scaling



- Ahhah, PaaS where P == Web Service
- E.g. Google's AppEngine
  - provides a web server framework
- Built-in elastic load balancing and scheduled actions for containers
  - Most invoked servers have to complete in less than 60 seconds
- Built in persistent key-value store (datastore) & non-persistent memcache for simple database tier
- Users can instantiate Backends: bigger, long running, billed differently
  - Not autoscaled, but user code can request (actuate) horizontal scaling
  - Used for running traditional database services, whose scaling is still hard

Scaling the virtual network

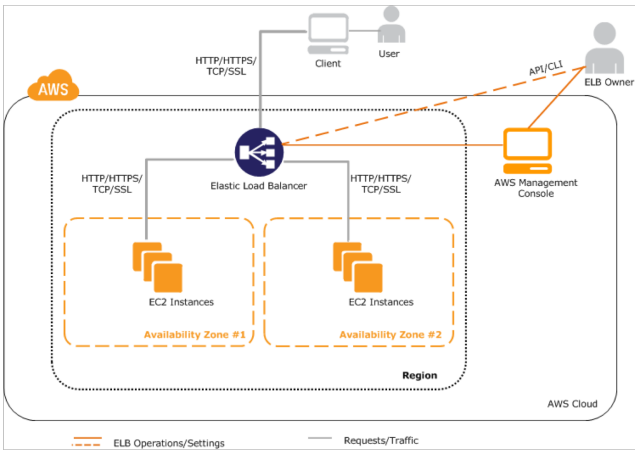
- Router-based load balancing is an example of a network middlebox
  - Eg., intrusion detection software, protocol accelerators, etc
  - Services that forward or mutate a network flow
- Scaling middleboxes is often overlooked, but may need its own tier
  - Especially if the function can be CPU intensive (ie., intrusion detection)
- Basic approach: split the flows
  - Eg. OpenFlow allows network switches to have flows defined & switched
  - "Load balancing the load balancers"
- Working with network switches & routers facilitates bandwidth allocation as well

Next up

- Next up (today): P1 discussion
- Next lecture will be examples of cloud infrastructure architectures

Basic service parallelization: Load Balancer

- Incoming stream of independent user requests broken into multiple separate streams, one per allocated machine
- Load balancer is not necessarily elastic
  - AWS charges for CloudWatch to monitor your EC2 instances
  - AutoScaling of the EC2 instances and reconfiguring of AWS Elastic Load Balancer at no add'l charge

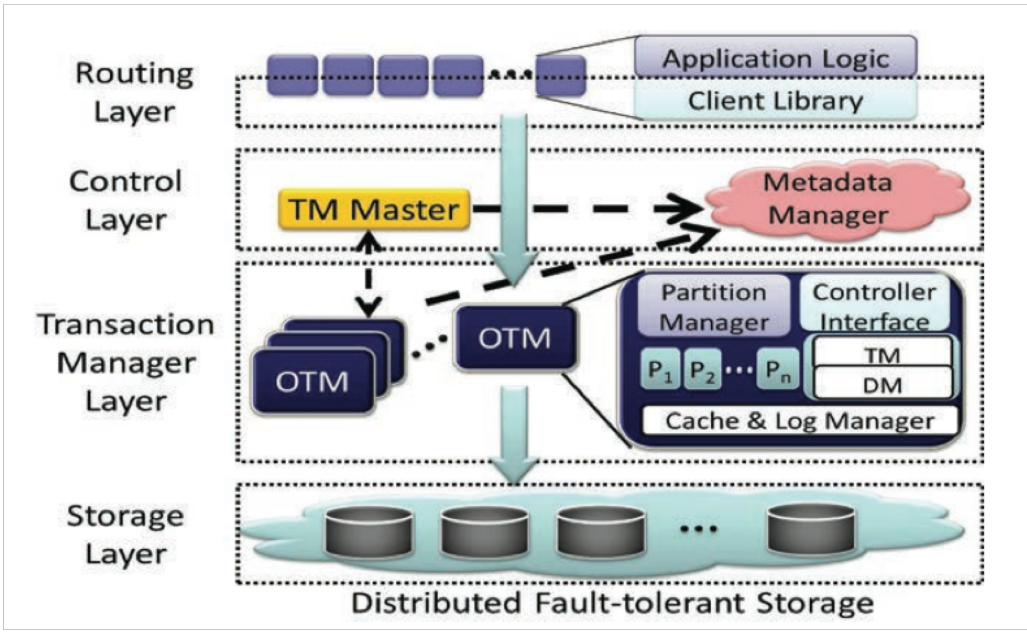


- docs.aws.amazon.com/ElasticLoadBalancing/latest/DeveloperGuide/SvcIntro_arch_workflow.html

What about a scalable relational database?

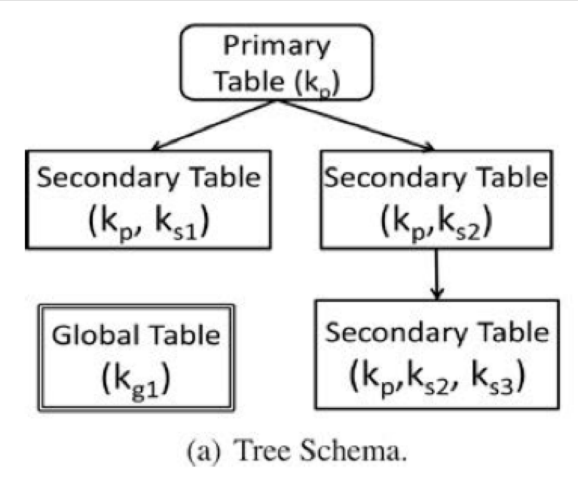
- Not a feature of traditional database, where users owned machines
- With a few possibly important restrictions, pretty easy (e.g. ElasTraS)
- Separate data at rest from ongoing or recent access & mutation
  - Data at rest is stored in (non-elastic) distributed pay-for-use storage (HDFS)
  - Recent access & mutation servers are elastic (called Owning Transaction Managers)
- Database can be partitioned but all transactions restricted to one partition
  - Distributed transactions usually block on locks and bottleneck performance scaling
- Elastic controller is also fault-tolerance manager
  - Servers can shutdown (flush to storage) or start up when controller re-assigns partitions
  - For the controller itself, reliability provided by replication (Zookeeper)

ElasTraS architecture scales OTM machines



Primary restriction: limitations on transactions

- Eliminate distributed transactions by rule – each transaction can touch only data from one partition
- But want dynamic repartitioning
- So establish a lot of mini-partitions allowing each OTM to manage many
- ElasTraS data model expresses this as the tree rooted at each row of the primary table
- All transactions restricted to the tree beneath one row in the root table



(a) Tree Schema.

Migrating OTMs with minimal pause

- First, push completed work to shared storage to reduce OTM state
- Next, "fuzzy migrate" of OTM state between servers
- Finally, stop processing requests for final part of migration

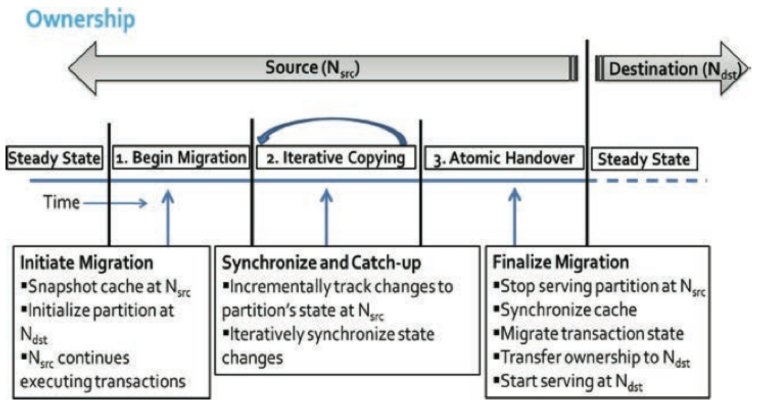


Fig. 3. Migration timeline for Albatross (times not drawn to scale).

Advanced Cloud Computing 15-719/18-709 S19

Project 1: Elasticity & Auto-Scaling on AWS

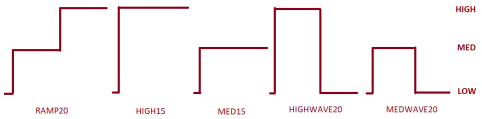
ALEX GLIKSON
AGLIKSON@CS.CMU.EDU

JANUARY 16, 2019

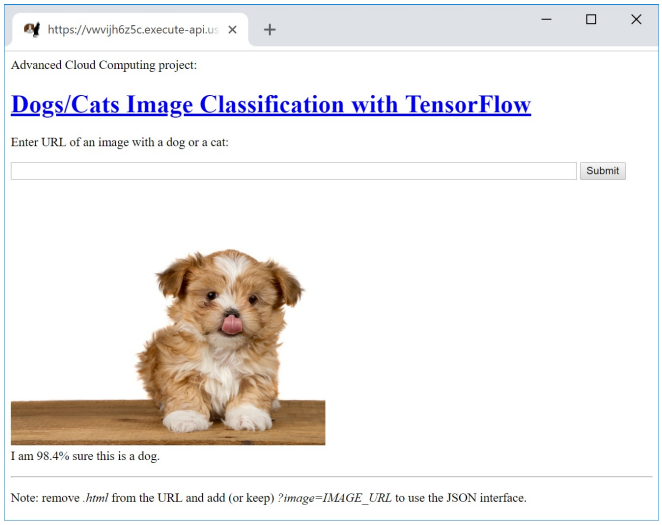
Carnegie Mellon University School of Computer Science

Overview

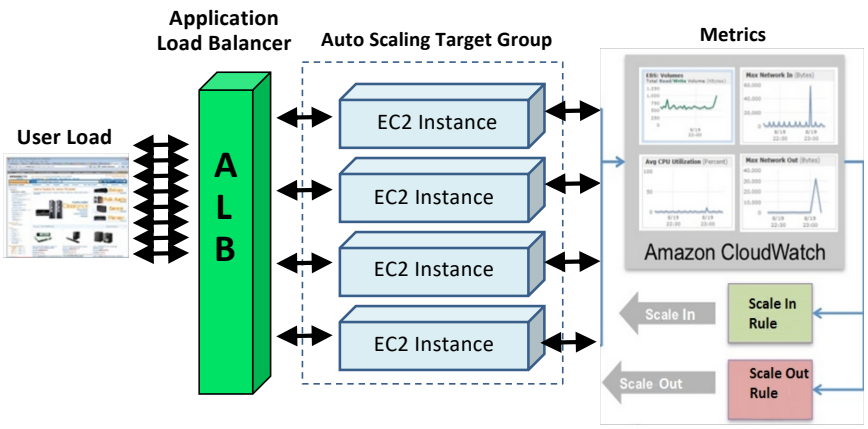
- **You are Given**:
  - Web service that performs Image Classification of Dogs vs Cats
- **Goal**: Run the application on AWS so that:
  - It can accommodate varying rates of incoming requests
    - Optimized for a fixed set of workload patterns



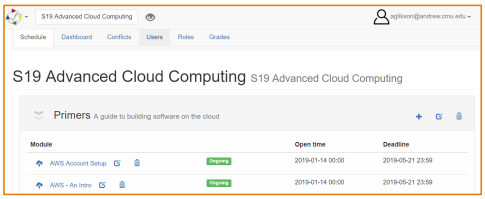
  - ...By scaling the #instances aiming at:
    - High RPS (average request-per-second) score
    - Low INST (average instance-count) score
    - Combined score = 0.7*RPS + 0.3*INST
      - Part 1: Score>65 for full credit, top #5 bonus



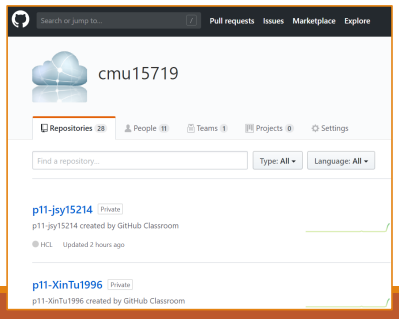
Architecture



Project Environment



- TheProject.Zone
  - Profile: update AWS Account ID, github.com username!
  - Project Writeup, Submissions, Scoreboard, etc
- Student VM
  - Ubuntu-based AMI with all the software needed to develop, test and submit assignments
  - Need to customize AWS credentials ('aws configure'), TPZ_USERNAME/PASSWORD env vars
- Github Classroom
  - Private per-student repositories under **cmu15719** org
  - Shared repository holding the latest starter code (see below)
- Starter code, README.md with instructions
  - Initial version provided when github repo is created
  - Instructions on pulling updates in README



Part 1: Auto-Scaling Policies

- Main Tool: **Terraform**
  - Deployment automation tool, using a declarative language to describe the desired AWS **resources**
  - Example (fragment):

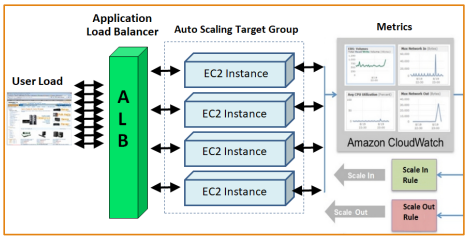
```
resource "aws_autoscaling_group" "asg" {
  name                      = "${var.label}-asg-${local.suffix}"
  availability_zones        = ["${aws_default_subnet.default_az1.availability_zone}"]
  min_size                  = "${var.asg_min_size}"
  max_size                  = "${var.asg_max_size}"
  desired_capacity          = "${var.asg_init_size}"
  health_check_type         = "ELB"
  health_check_grace_period = 30
  default_cooldown          = "${var.asg_cooldown}"

  enabled_metrics           = ["GroupDesiredCapacity", "GroupInServiceInstances", "GroupPendingInstances", "GroupStandbyInstances",

  launch_configuration = "${aws_launch_configuration.launch-config.name}"
  target_group_arns    = ["${aws_alb_target_group.target-group-ws.arn}", "${aws_alb_target_group.target-group-admin.arn}"]
```


Part 1: Auto-Scaling Policies (Cont'd)

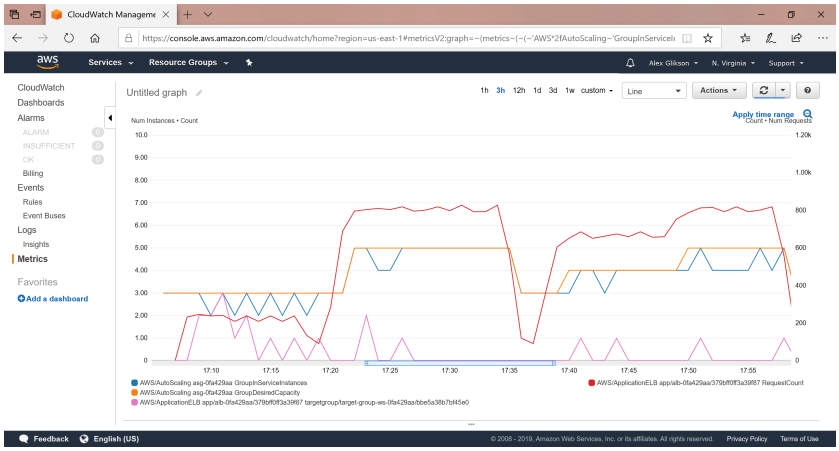
- Starter
  - README.md
  - p11.tf - partial Terraform template
  - bin/p11test, bin/p11submit
  - .gitignore



- Resources to be defined by the student include:
  - AutoScaling Group – encompasses all the following resources
  - Security Group – firewall rules for the LB and EC2 instances
  - Launch Configuration – EC2 instance configuration "template"
    - Used when instances need to be add/removed
  - Load Balancer – externally accessible; forwards the incoming requests to a group of EC2 instances
  - Target Groups, LB Listeners (image classification API on port 80, admin API on port 8080)
  - CloudWatch Alarms – a rule that defines a metric to monitor, and a threshold that fires an alarm
  - Scaling Policies – a combination of alerts and actions (e.g, add/remove N instances)
- Tip: all resources MUST be tagged with "Project":"15719.p11"

Hints

- CloudWatch is your best friend! If you have a doubt, don't guess – check in CloudWatch!
  - Metrics are persisted even when the resources are deprovisioned (can analyze 'post-mortum')

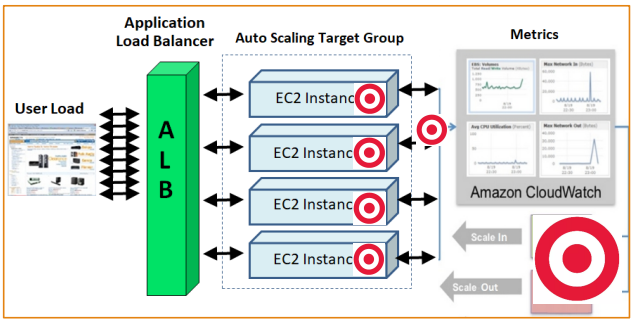


Hints (Part 1)

- Examine the incoming RPS patterns by looking at the RequestCount metric of the ALB
  - You can do it even without having scaling policies
- Determine key ingredients of the system behavior **before** designing policies
  - Workload characteristics, e.g.: Incoming RPS levels (as explained above), RPS each instance can serve, etc
  - Metrics that might correlate well with the load (Instances, Load Balancer, etc)
  - System delays
    - E.g., VM boot time, cooldown, metrics aggregation time, alarm-to-action time (may vary between metrics!), etc
- When you run tests, **always** monitor health checks. Health check failures **will** reduce your score!
  - E.g., ALB/TG (Un)HealthyHostCount, ASG GroupInServiceInstances vs GroupDesiredCapacity, etc
  - Health-check failures are also reflected in Alarm History, as well as ASG Scaling Activity Log
  - Tip: if health checks on port 80 fail for unknown reason, try moving the health check to port 8080
- Test your policies on individual patterns (provided or custom) first – this will save you lots of **time**!
- Consider using step policies (multiple thresholds), adding/removing >1 instance at a time, etc
- Before running a long test, check if there are **starter updates** (e.g., updated test/submit programs)

Part 2: Auto-Scaling Controller

- Instead of relying on AWS scaling policies, you are going to develop your own **controller**
  - Web Service (each VM):
    - Collection of metrics that can reflect the load
    - Shipment to CloudWatch
  - Controller:
    - Retrieval of metrics from CloudWatch
      - shipped from all VMs
    - Detection of conditions requiring to add/remove instances
    - Adjustment of the DesiredCapacity of the corresponding ASG



- Same StudentVM, similar tester and submitter logic, higher Score target

Part 2: Auto-Scaling Controller

- Starter:
  - controller: Lambda function, Step Functions state machine
  - webservice:
    - Django application implementing image classification
    - Skeleton of Django middleware to collect metrics and apply rate limiting (to avoid health check failures)
  - Terraform templates
  - README.md, bin/p12test, bin/p12submit

Part 3: Per-Request Scaling with AWS Lambda

- AWS Lambda
  - 'Function-as-a-Service' model
  - For each invocation, function code is executed in a separate lightweight runtime sandbox
    - API Gateway for HTTP requests as triggers
  - Pay-per-use in granularity of 100ms
  - No need to warry about:
    - Security Groups, Load Balancing, Target Groups, Auto-Scaling Groups, Cloud Watch Metrics, Alarms, Auto-Scaling Rules, Health-Check, etc
- Task
  - Image classification logic packaged as a Lambda function
  - Deployment automation with Terraform
    - Lambda Function, API Gateway
  - Latency optimization
- Goal: even higher RPS, lower cost

BACKUP

Research Idea: Elastic ("Serverless") VM

- Think of an SSH client, connected to an Ubuntu VM (e.g., Student VM in this course)
- What if we could replace the VM (and potentially the client) so that:
  - User does not need to pay for the VM when nothing is running
  - User does not need to decide on the VM size up-front
  - User experience remains roughly the same
- Envisioned solution approach:
  - Docker Containers
  - Kubernetes, Knative-like middleware
  - Client-side logic (e.g., caching)
- If you are interested (e.g., Independent Study), contact Alex Glikson (aglikson@cs.cmu.edu)