Tributary: spot-dancing for elastic services with latency SLOs

Aaron Harlap, Andrew Chung, Alexey Tumanov*, Greg Ganger, Phil Gibbons

Carnegie Mellon University

* UC Berkeley
5.2 Scaling Policies Evaluated

We implement three popular scaling policies: Reactive, Predictive Moving Window Average (MWA), and Predictive Linear Regression (LR) to evaluate our system. In all three policies, the utility function implemented is linear with respect to the amount recommended by the scaling policy. We are able to make this assumption since our workload characteristic is embarrassingly-parallel — if a workload exhibits different scaling characteristics, a different utility function generator can be implemented.

The Reactive Scaling Policy scales out immediately when demand reported by the MM is greater than what the available resources are able to handle. It scales in slowly (only after three minutes of low demand), as recommended by Gandhi et al. [15], to prevent premature scale-in in case the demand fluctuates widely in a short period of time. Tributary’s smart termination mechanism helps with the Reactive Scaling Policy’s strategy of scaling in slowly. With smart termination, Tributary keeps a resource available if the resource has not yet been revoked or met the end of its billing period even if the scaling policy recommends a scale in.

The Predictive-MWA Scaling Policy maintains a sliding window of a fixed size, with each window entry consisting of the number of requests received in each monitoring period. The policy takes the average of the window entries to predict the number of requests on the next monitoring period. The policy then adjusts the utility and scaling functions according to the predicted number of requests, and reports the updated functions to the ResMgr to scale in expectation of future requests. The Predictive-LR Scaling Policy also maintains a sliding window of a fixed size, but rather than using the average in the window for prediction, the policy performs linear regression on data points in the window to estimate the expected number of requests in the next monitoring period.

Our experiments show that regardless of the scaling policy used, Tributary beats its competitors in both meeting the service latency target and the cost of operation.

5.3 Improvements with Tributary

This section evaluates Tributary’s ability to reduce cost and latency target misses. We compare Tributary to three alternate resource acquisition approaches: using on-demand resources, using AWS AutoScale with spot instances, and using Smart AutoScale.

AWS Autoscale.

The default AWS AutoScale only supports the simplest reactive scaling policies. To provide better comparison between approaches, we implement the AWS AutoScale resource acquisition algorithm as closely as possible according to its documentation [2] and integrate it with Tributary to work with...
Elastic Service Architecture

- Load Balancer
- Scaling Policy
- Resources
- Resource Manager
Elastic Service Architecture

User Requests

Load Balancer

Scaling Policy

Resources

Resource Manager
Elastic Service Architecture

User Requests

Load Balancer

Scaling Policy

Fwd Requests

Resources

Resource Manager
Elastic Service Architecture

User Requests → Load Balancer → Stats → Scaling Policy → Fwd Requests

Fwd Requests → Resources → Resource Manager
Elastic Service Architecture

User Requests → Load Balancer → Stats → Scaling Policy → Resource Manager → Resources

Fwd Requests

How many resources currently needed
Elastic Service Architecture

User Requests → Load Balancer → Stats → Scaling Policy → How many resources currently needed

Fwd Requests → Resources → Add Remove

Resource Manager
Why Tributary?

• CSPs offer cheaper resources that come with potential of being taken away
  - GCE preemptible instances
  - AWS EC2 spot instances

• Preemptions are bad for services w/ SLOs
Transient resources much cheaper

- Often 75-85% cheaper to use Spot Instances
Transient resources much cheaper

- Often 75-85% cheaper to use Spot Instances

![Graph showing price per hour for c4.xlarge instances from Jan 19 to Jan 24, with On-Demand pricing as a reference. The graph highlights the significant cost savings available through the use of Spot Instances.](attachment:image.png)
Transient resources much cheaper

- Often 75-85% cheaper to use Spot Instances

Low Cost

<table>
<thead>
<tr>
<th>Date</th>
<th>Price per Hour ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jan 19</td>
<td>0</td>
</tr>
<tr>
<td>Jan 20</td>
<td>5</td>
</tr>
<tr>
<td>Jan 21</td>
<td>5</td>
</tr>
<tr>
<td>Jan 22</td>
<td>5</td>
</tr>
<tr>
<td>Jan 23</td>
<td>5</td>
</tr>
<tr>
<td>Jan 24</td>
<td>5</td>
</tr>
</tbody>
</table>
Spot Market Details

• Many different spot markets
  - each instance type, in each availability zone, in each datacenter
  - empirically, markets are uncorrelated
• If pre-empted, Amazon issues refund
  - during first hour only
• Aquire resource(machines) by specifying:
  - <spot market, bid price, number of machines>
Tributary Changes how we Acquire Resources

- Uses transient instead of reliable resources
  - while addressing bulk preemptions
- Uses resource from multiple spot markets
  - predicts allocation $P[\text{preemption}]$
  - tracks inter-market correlations
  - maintains diverse resource buffer
Tributary Components

- Predicting resource reliability
- Constructing resource footprint
Influencing P[preemption]

- User’s bids influence P[preemption] of spot instances
  - bid delta = user bid price - spot market price
- Bigger Delta
  - lower P[preemption] and higher cost
- Smaller Delta
  - higher P[preemption] and lower cost
Predicting $P[\text{preemption}]$

- Predict $P[\text{preemption}]$ as a function of bid deltas
- Extract features
  - calendrical
  - temporal
- Plug features into LSTM Model
  - models EC2 as a sequence of events
Constructing the Resource Footprint

• Need to achieve capacity to satisfy SLO of client workload

• Need sufficient diversity across markets

While expected request capacity < SLO:

Add resource that increases expected cost the least and increases request capacity the most.
Computing Expected Request Capacity

- Compute probability of exactly 0 - N resources not pre-empted
- Accounts for spot market dependencies
- Encourages diversity
Computing Expected Request Capacity

- Compute probability of exactly 0 - N resources not pre-empted
- Accounts for spot market dependencies
- Encourages diversity

50%
Computing Expected Request Capacity

- Compute probability of exactly 0 - N resources not pre-empted
- Accounts for spot market dependencies
- Encourages diversity

\[
\begin{array}{cc}
50\% & \\
\square & \square \\
50\% & \\
\square & \\
\end{array}
\]
Computing Expected Request Capacity

- Compute probability of exactly 0 - N resources not pre-empted
- Accounts for spot market dependencies
- Encourages diversity

\[ 1 \times 0.5 + 0.5 \times 0.5 = 0.75 \]
So Why Does this Work?

- Creates a diversified, oversized footprint
  - able to tolerate preemptions
  - little or no extra cost
- Handles unexpected workload spikes
  - handled via oversized natural resource buffers
Time for an Example

AutoScale

Tributary
Time for an Example

AutoScale

Tributary
Tributary Serves More Requests

AutoScale

Tributary
Request Rate Decreases

**AutoScale**

**Tributary**
Tributary’s Resources are Pre-empted

AutoScale

Tributary
Experimental Setup

- 4 Traces Evaluated
  - show Clarknet
- 3 Scaling Policies
  - show reactive
- Comparisons
  - Autoscale on spot
  - Autoscale+Buffer on spot
  - Tributary

Figure 4: Traces used in system evaluation.

Figure 5: Cost savings (red) and percentage of "slow" requests (blue) for the ClarkNet trace.

5.2 Scaling Policies Evaluated

We implement three popular scaling policies: Reactive, Predictive Moving Window Average (MWA), and Predictive Linear Regression (LR) to evaluate our system. In all three policies, the utility function implemented is linear with respect to the amount recommended by the scaling policy. We are able to make this assumption since our workload characteristic is embarrassingly-parallel — if a workload exhibits different scaling characteristics, a different utility function generator can be implemented.

The Reactive Scaling Policy scales out immediately when demand reported by the MM is greater than what the available resources are able to handle. It scales in slowly (only after three minutes of low demand), as recommended by Gandhi et al. [15], to prevent premature scale-in in case the demand fluctuates widely in a short period of time. Tributary’s smart termination mechanism helps with the Reactive Scaling Policy’s strategy of scaling in slowly. With smart termination, Tributary keeps a resource available if the resource has not yet been revoked or met the end of its billing period even if the scaling policy recommends a scale in.

The Predictive-MWA Scaling Policy maintains a sliding window of a fixed size, with each window entry consisting of the number of requests received in each monitoring period. The policy takes the average of the window entries to predict the number of requests on the next monitoring period. The policy then adjusts the utility and scaling functions according to the predicted number of requests, and reports the updated functions to the ResMgr to scale in expectation of future requests.

The Predictive-LR Scaling Policy also maintains a sliding window of a fixed size, but rather than using the average in the window for prediction, the policy performs linear regression on data points in the window to estimate the expected number of requests in the next monitoring period.

Our experiments show that regardless of the scaling policy used, Tributary beats its competitors in both meeting the service latency target and the cost of operation.

5.3 Improvements with Tributary

This section evaluates Tributary’s ability to reduce cost and latency target misses. We compare Tributary to three alternate resource acquisition approaches: using on-demand resources, using AWS AutoScale with spot instances, and using Smart AutoScale.

AWS Autoscale. The default AWS AutoScale only supports the simplest reactive scaling policies. To provide better comparison between approaches, we implement the AWS AutoScale resource acquisition algorithm as closely as possible according to its documentation [2] and integrate it with Tributary to work with...
Comparing to AutoScale

- **AWS AutoScale**
  - AWS service that acquires cheapest spot instances
Other Interesting Results

- Across 4 traces Tributary reduces cost by 47-62%
- Outperformed recent research systems
  - ExoSphere [Sharma 2017]
  - Proteus [Harlap 2017]
- Only ~50% of cost saving come from preemptions
Conclusion

• Provides reliable service using transient resources
• Uses diversified buffers of resources
• Reduces cost by ~85% over on-demand