Cost-Optimizing Existing Amazon DynamoDB Workloads

Greg Krumm
Sr. Solutions Architect
AWS

Juhi Patil
Solutions Architect
AWS

New vs. Existing Cost Optimization

Optimize utilization, not design

Focus on efficient operations

Iterative process, not one-time

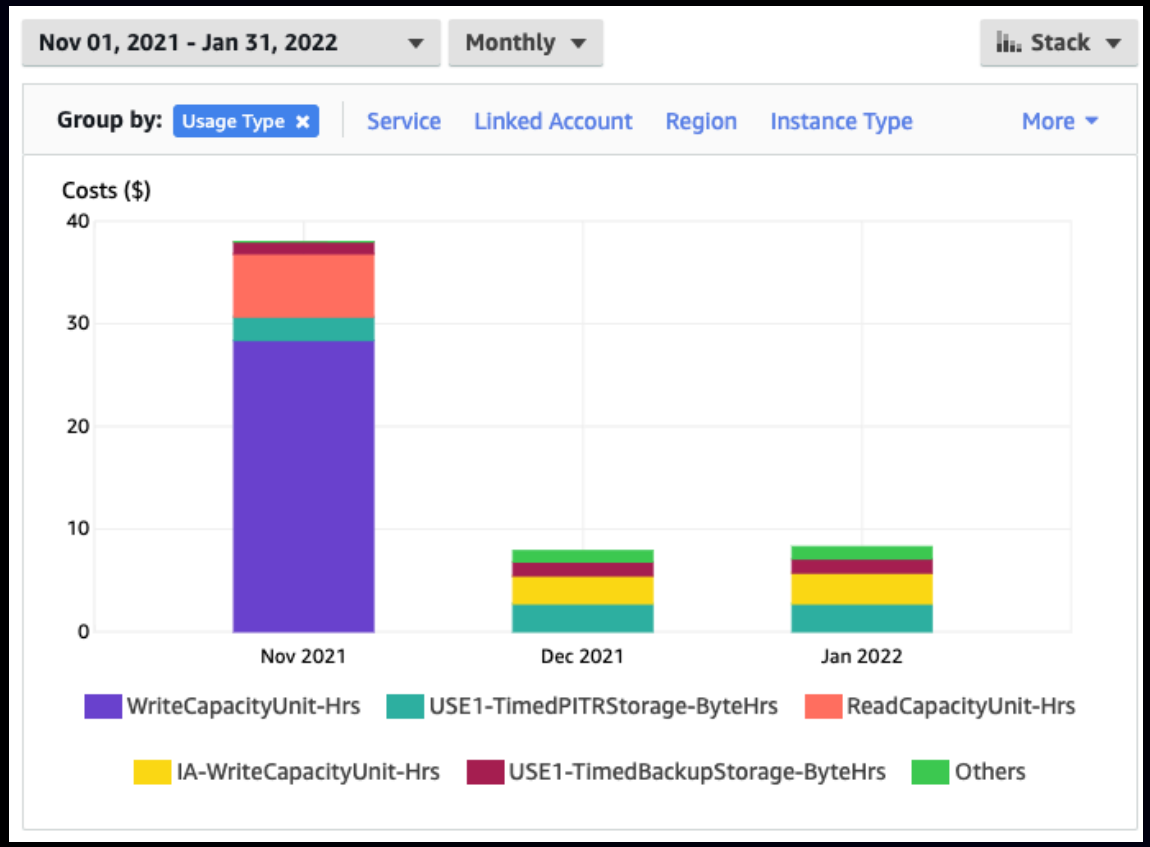
Cost Optimization Approaches

1. Tag tables eponymously
2. Pick the right capacity mode
3. Right-size provisioned capacity
4. Choose the right auto scaling settings
5. Use the right table class
6. Find unused tables and GSIs
7. Identify poor table usage patterns
8. Filter streams events in Lambda functions

aws

Untagged tables

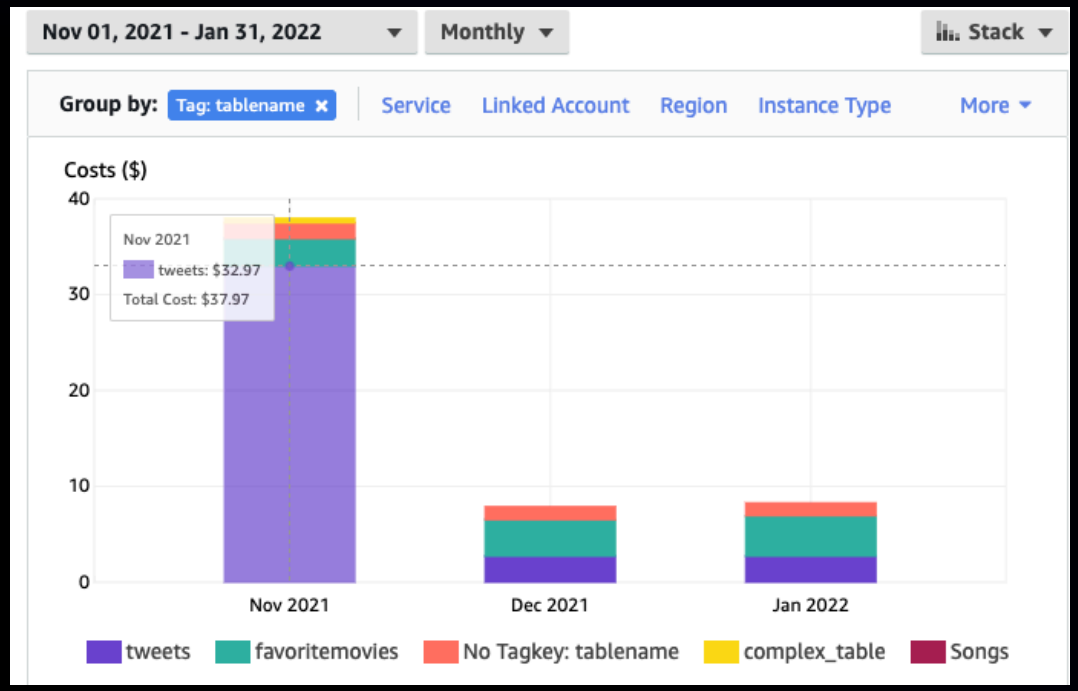
Default grouping (usage type only)



What are my DynamoDB usage costs per region?

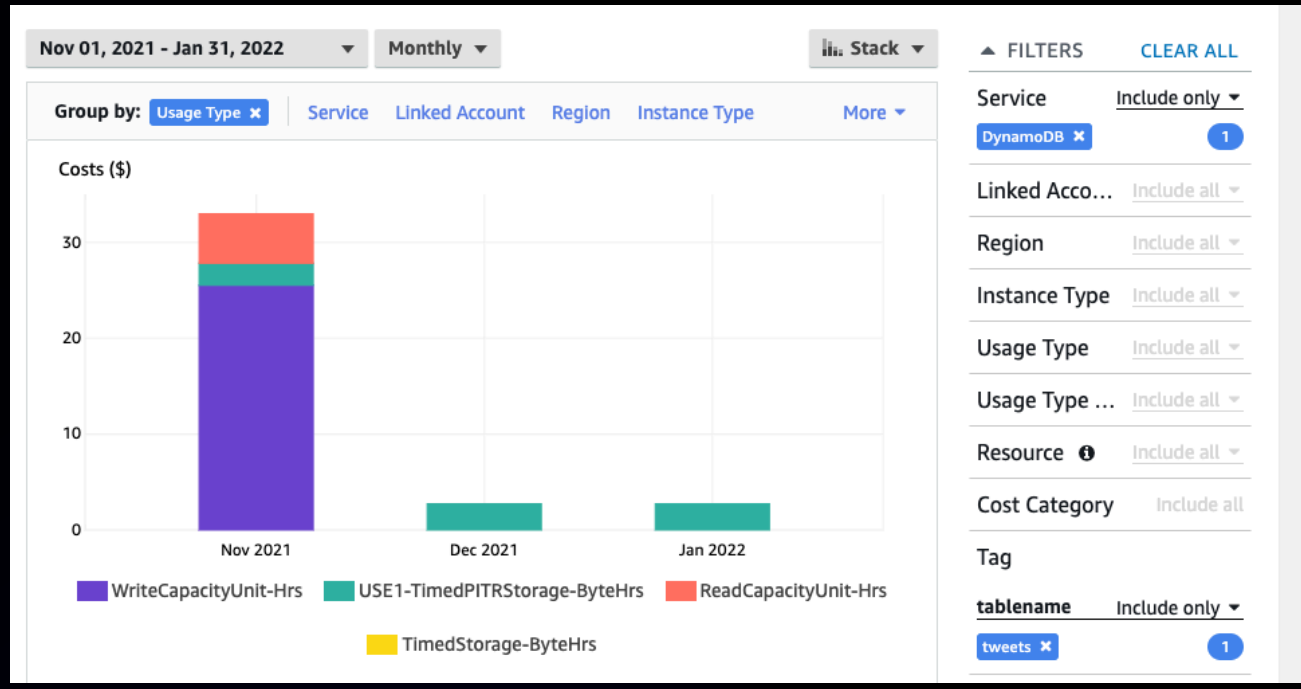
Tables tagged eponymously

Group by tag (table)



What is my most expensive DynamoDB table?

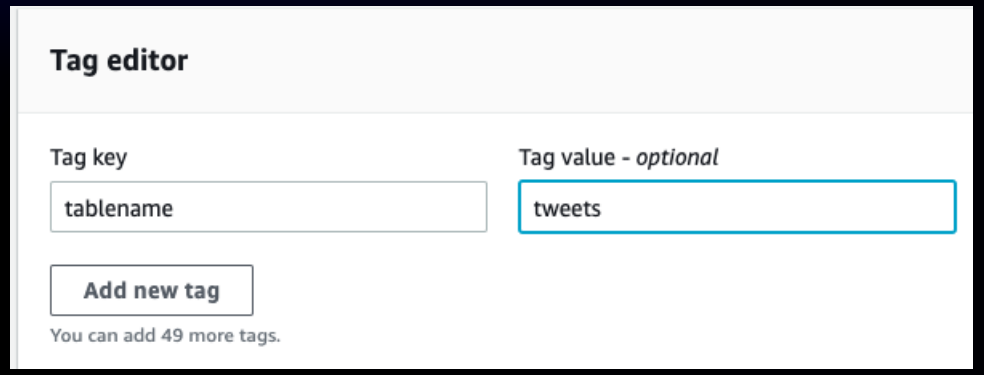
Group by usage for tag (table)



What usage contributes to a table's costs?

How do I eponymously tag tables?

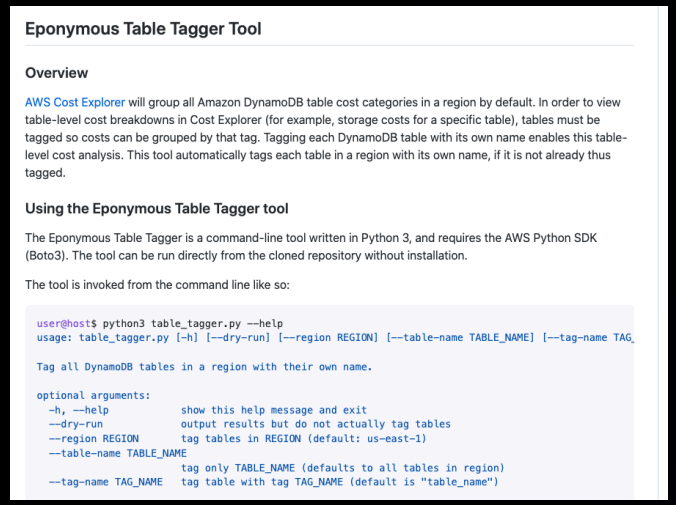
In the console



Via the CLI

```
aws dynamodb tag-resource \
  --resource-arn arn…:table/MusicCollection \
  --tags Key=tablename,Value=tweets
```

Table tagger utility



Eponymous Table Tagger Tool

Overview

AWS Cost Explorer will group all Amazon DynamoDB table cost categories in a region by default. In order to view table-level cost breakdowns in Cost Explorer (for example, storage costs for a specific table), tables must be tagged so costs can be grouped by that tag. Tagging each DynamoDB table with its own name enables this table-level cost analysis. This tool automatically tags each table in a region with its own name, if it is not already thus tagged.

Using the Eponymous Table Tagger tool

The Eponymous Table Tagger is a command-line tool written in Python 3, and requires the AWS Python SDK (Boto3). The tool can be run directly from the cloned repository without installation.

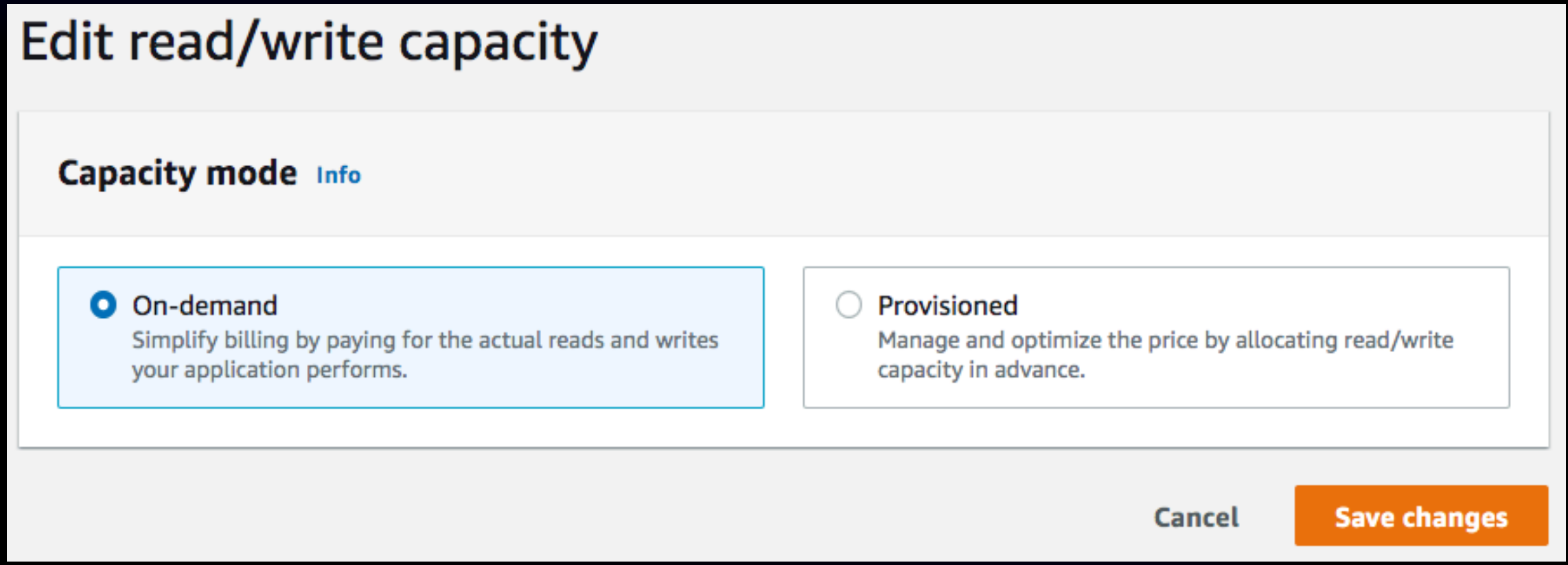
The tool is invoked from the command line like so:

```
user@host$ python3 table_tagger.py --help
usage: table_tagger.py [-h] [--dry-run] [--region REGION] [--table-name TABLE_NAME] [--tag-name TAG_

Tag all DynamoDB tables in a region with their own name.

optional arguments:
  -h, --help            show this help message and exit
  --dry-run             output results but do not actually tag tables
  --region REGION       tag tables in REGION (default: us-east-1)
  --table-name TABLE_NAME
                        tag only TABLE_NAME (defaults to all tables in region)
  --tag-name TAG_NAME   tag table with tag TAG_NAME (default is "table_name")
```

You must enable the tag in cost explorer!

https://github.com/awslabs/amazon-dynamodb-tools

Choose or change your table's capacity mode



On-Demand capacity mode

Features

- No capacity planning, provisioning, or reservations—simply make API calls
- Pay only for the reads and writes you perform

Key benefits

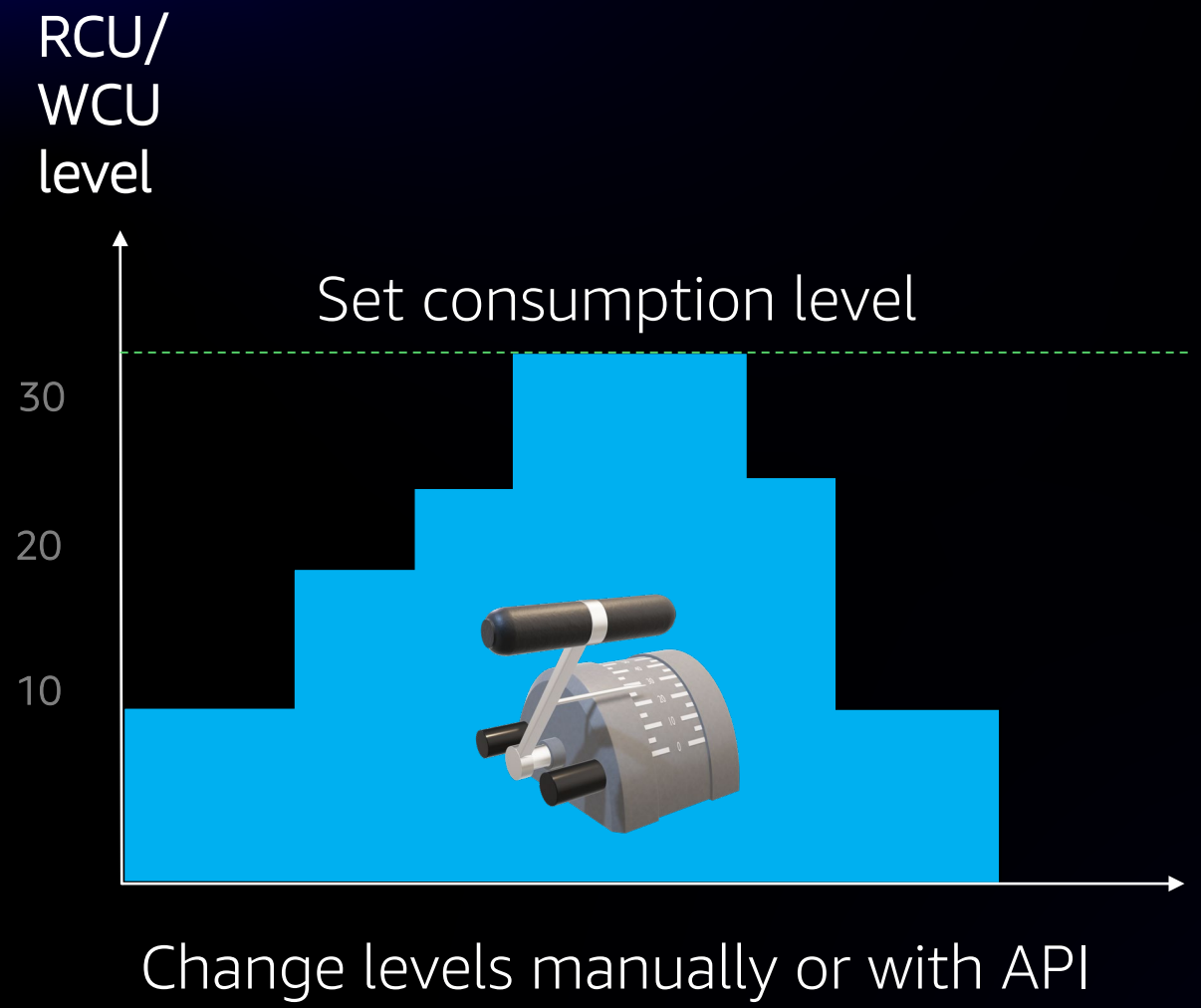
- Eliminates tradeoffs of overprovisioning or underprovisioning
- Instantly accommodates your workload as traffic ramps up or down

On-demand scaling properties

- Base throughput
  - Up to 4,000 WCU
  - Up to 24,000 RCU
  - Any linear combination of the two

- Maximum throughput
- Unlimited!
  - Pay per request: Use nothing, pay nothing

Provisioned Capacity



Features

- Set read and write levels separately

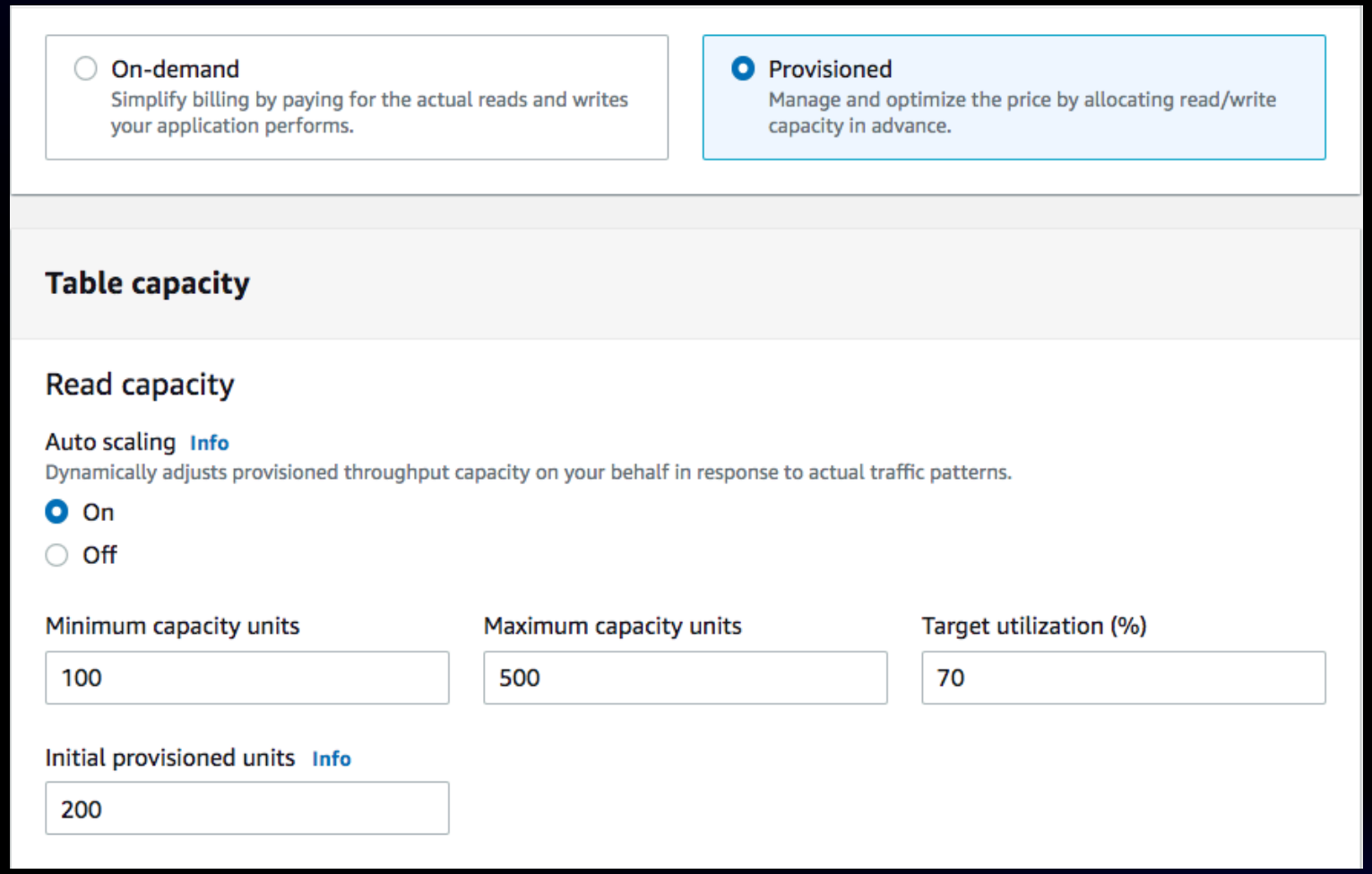
Key benefits

- Can be much less expensive for steady workloads

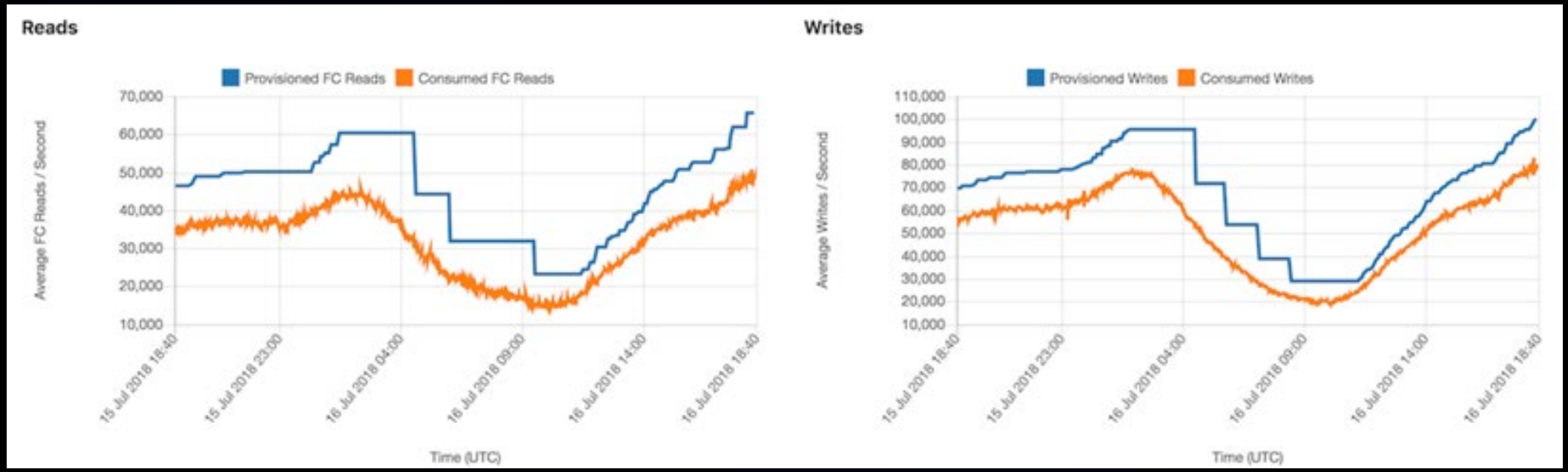
Note:

- Throttling possible if traffic spikes
- Waste possible if capacity not used

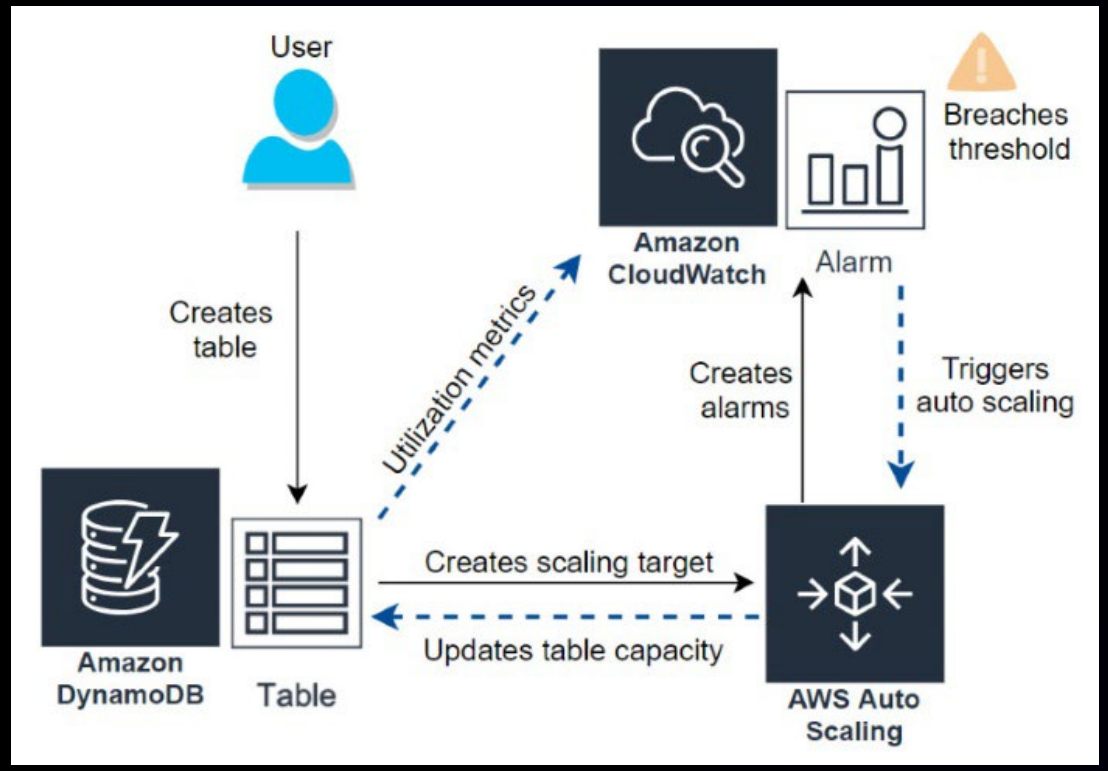
Provisioned Capacity with Auto Scaling option



Provisioned Capacity with Auto Scaling



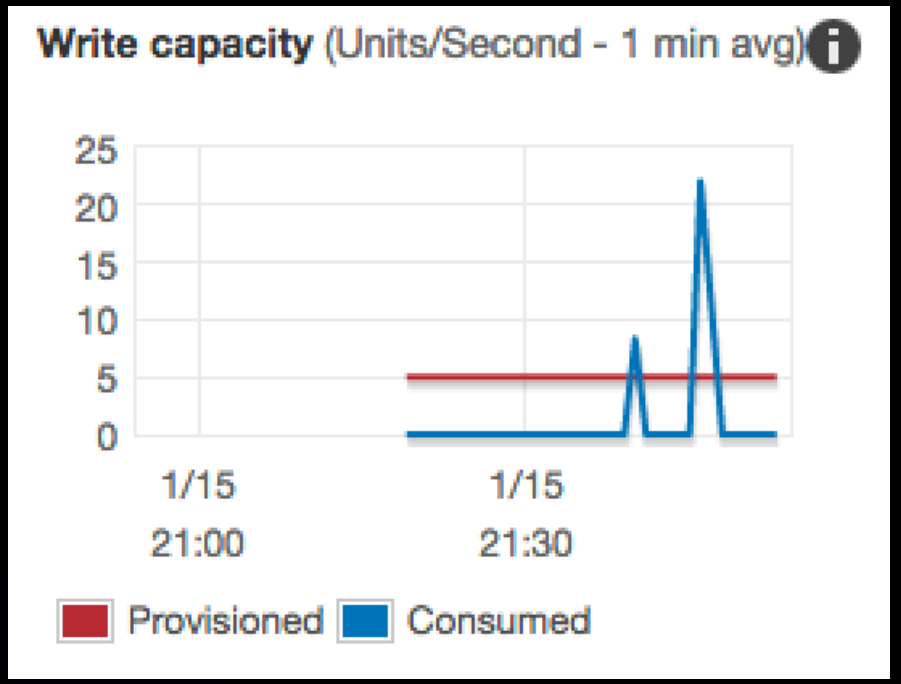
Auto Scaling creates Amazon CloudWatch Alarms for you



- Two consecutive minutes of traffic above target utilization causes scale up to begin
- It takes a few minutes to apply new levels
- Burst capacity (grace) may handle sudden spikes for a few minutes
- Sudden and sustained spikes can result in throttling, retries, and throttle errors

Burst Capacity

- Sudden spikes happen!
- Expect up to five minutes of grace capacity to handle spikes, if capacity unused before the spike
- Sustained traffic above provisioning levels will be throttled



Provisioned Capacity vs On-Demand cost

Provisioned is 80% less than On-Demand? True?

- For flat steady workloads, perfectly provisioned capacity, yes
- For unpredictable workloads, overprovisioning is inevitable

aws

Using CloudWatch Charts to decide

For tables with On-Demand capacity:

- Look for tall spikes, with drops to zero – Keep in On-Demand
- Steady traffic that rises modestly and steadily - Move to Provisioned

aws

Using CloudWatch Charts to decide

For tables with Provisioned capacity:

- Look for tall spikes that cause throttling – move to On-Demand
- Look for gaps of an hour or two – move to On-Demand

- If your table has throttling while seemingly having plenty of capacity, consider if hot access patterns are causing trouble

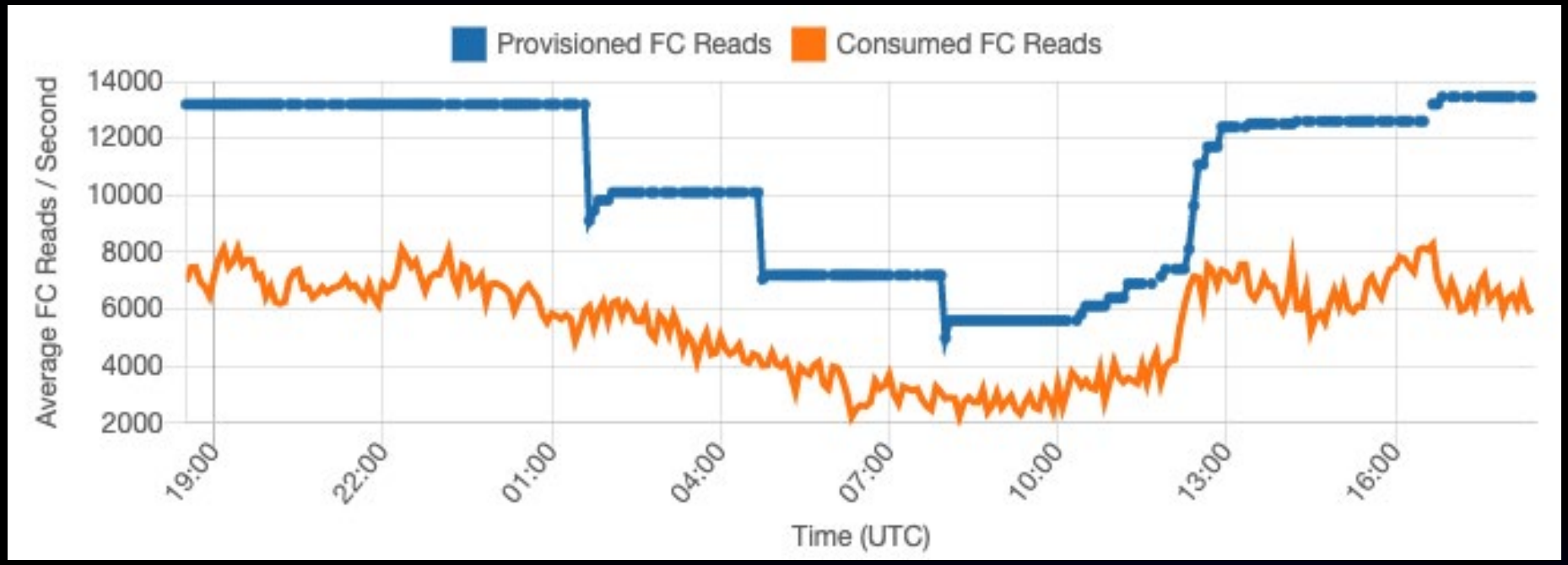
- CloudWatch Contributor Insights

- Bulk loads to a single PK in excess of 1000 WCU/sec can throttle

aws

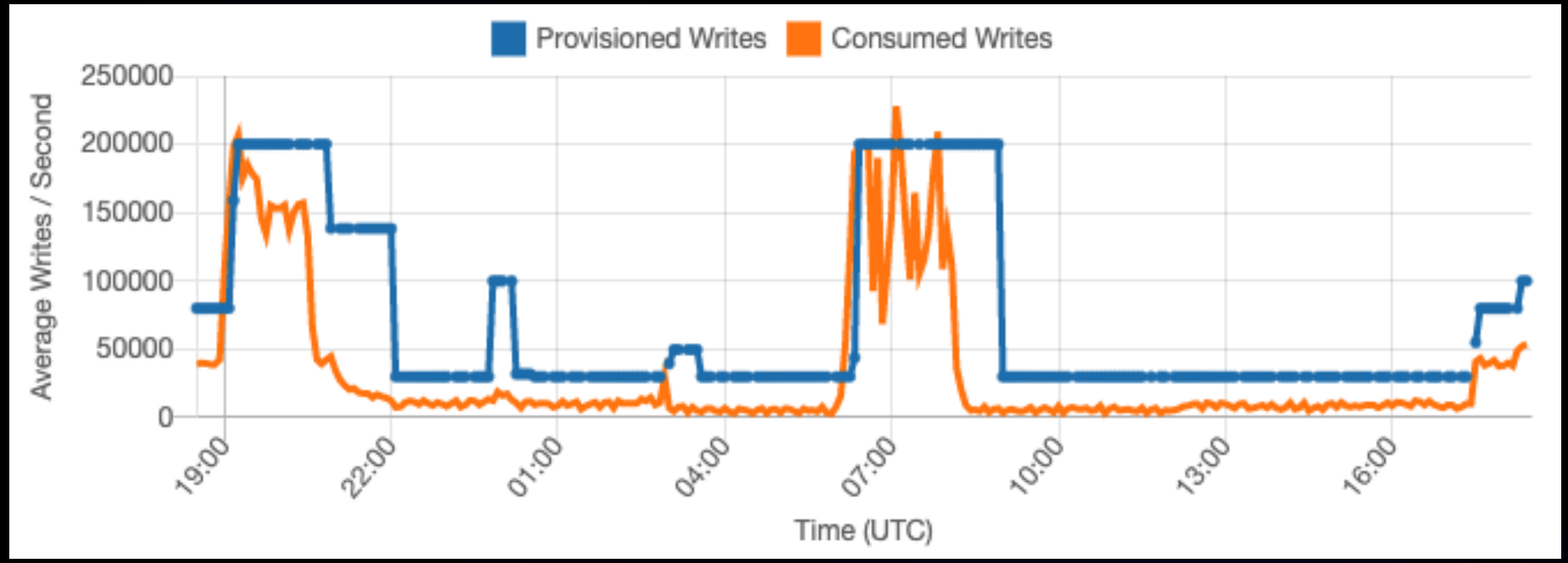
Ideal Provisioned Capacity with Auto Scaling

Read units provisioned/consumed



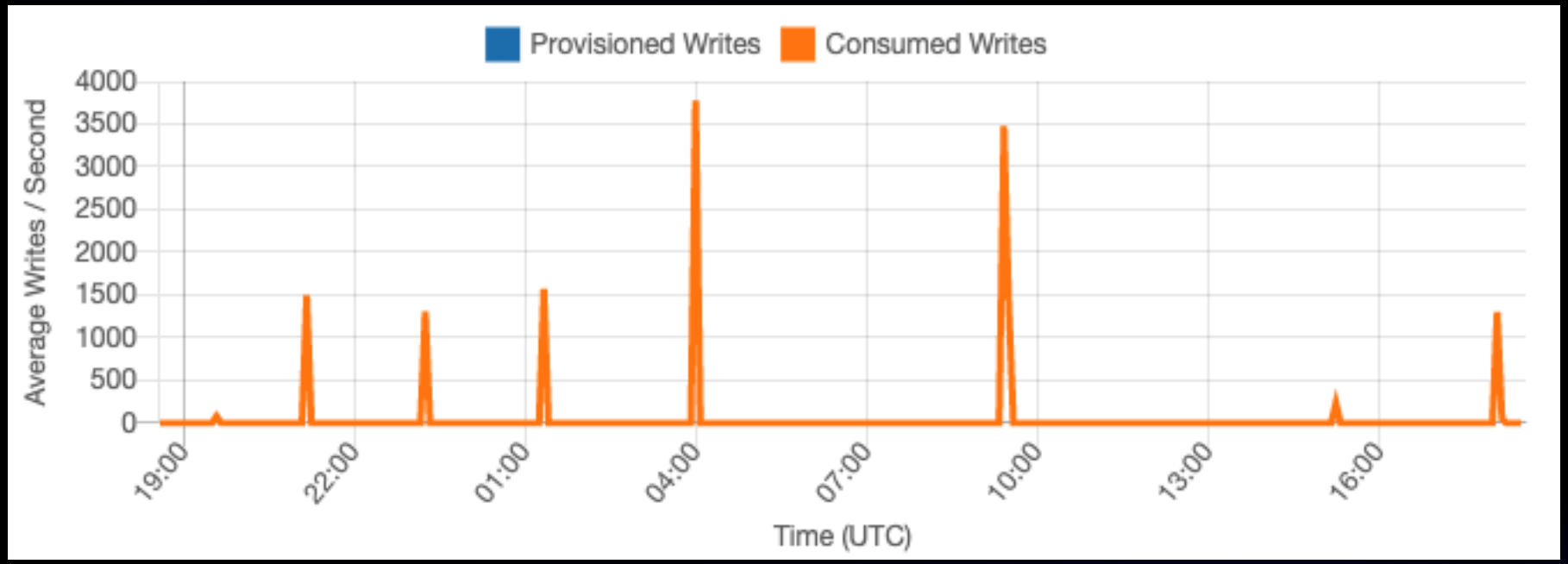
Ideal Provisioned Capacity with Auto Scaling

Write units provisioned/consumed



On-Demand optimized usage

Spiky Write requests



Demo – AutoScaling Modeler

bit.ly/ddbcost

When does Autoscaling fit your workload?

Workloads that are spiky may not work well with Auto Scaling due to inherent scale up and scale down delays

Change the model by adjusting the green numbers.

	RCU	WCU		Show
Unit size (KB)	4	1	Provisioned Capacity	Y
Provisioned Pricing/hour	$0.00013	$0.00065	Throttle Risk	Y

Workload Shape		
Load Rate	60,000	Items/min
Items to Load	3,000,000	
Item Size	2	KB
Ramp Duration	0	Minutes
Peak Rate in WCU	2,000	WCU/second
Peak Duration	50	Minutes
Peak Size Total	6.7	GB
Provisioned Capacity Auto Scaling		
Min	500	WCU
Max	3500	WCU
Target Utilization	70%	
Baseline WCU level	200	
Total WCU Required	7,035,747	
Total Provisioned WCU	12,019,192	
WCU Utilization	59%	
WCU Price OnDemand	0.00000125	
On Demand Price	$8.79	
Autoscaled Provisioned Price	$2.17	

Table Classes

DynamoDB table classes

- Standard: Balances storage and read/write costs
- Standard-Infrequent Access (IA):
  ~60% lower storage costs, ~25% higher read/write costs

Pick Standard-IA when storage costs >50% of total throughput costs

Table class affects:

- Read/Write request costs (RCU, WCU)
- Data storage costs
- Global Tables costs (rWCU)
- GSI costs (GSIs share the table class of their parent)
- Reserved capacity

How to choose the right table class

Calculate storage and read/write costs for each table and GSI for both classes

Read/Write Costs

- Easier to calculate for provisioned capacity mode
  - Autoscaling activity can affect the calculations
- For On-demand capacity mode, will have to look at utilization history in CloudWatch metrics

Storage Costs

- Storage usage is not yet a trackable CloudWatch metric
- Growth estimates are manual

Global Tables Costs

- Tables in different regions can have different table classes
- rWCUs are higher cost for Standard-IA

aws

Table Class Evaluator tool

Using the Table Class Evaluator tool

The Table Class Evaluator is a command-line tool written in Python 3, and requires the AWS Python SDK (Boto3) >= 1.23.18. You can find instructions for installing the AWS Python SDK at https://aws.amazon.com/sdk-for-python/. The tool can be run directly from the cloned repository without installation.

The tool is invoked from the command line like so:

```
user@host$ python3 table_class_evaluator.py --help
usage: table_class_evaluator.py [-h] [--estimates-only] [--region REGION] [--table-name TABLE_NAME]

Recommend Amazon DynamoDB table class changes to optimize costs.

optional arguments:
  -h, --help            show this help message and exit
  --estimates-only      print table cost estimates instead of change recommendations
  --region REGION       evaluate tables in REGION (default: us-east-1)
  --table-name TABLE_NAME
                        evaluate TABLE_NAME (defaults to all tables in region)
```

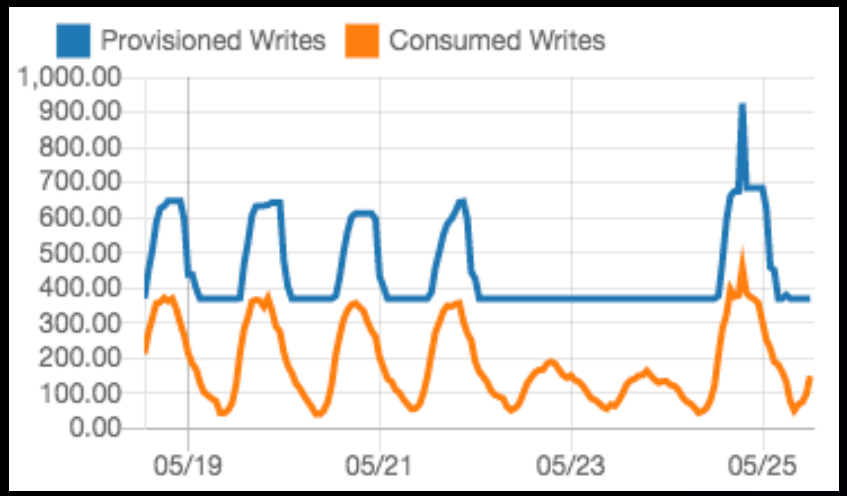
- Provisioned capacity only
- Lots of autoscaling activity could give wrong results
- LSIs are not calculated
- Can be used to calculate costs

https://github.com/awslabs/amazon-dynamodb-tools

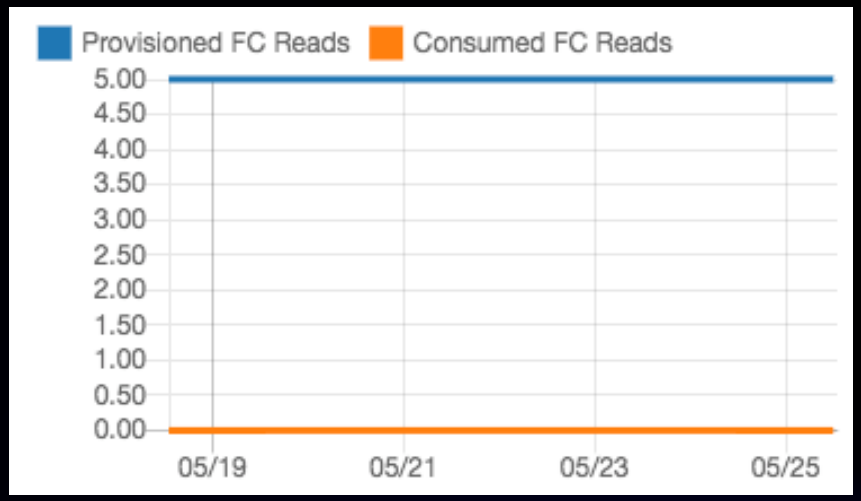
Cost Driver : Storing, Writing too much to an unused GSI

- Check CloudWatch for any Global Secondary Indexes without read traffic in the last 30 days.
- Consider dropping these GSIs to save costs!

GSI Writes



GSI Reads



zero

Identify Poor Table Usage Patterns

Using only strongly-consistent reads

- Eventually-consistent read (default) uses .5 RCU per 4KB
- Strongly-consistent read uses 1 RCU per 4KB

Using transactions for all operations

- Eventually-consistent read (default) uses .5 RCU per 4KB
- Transactional read uses 2 RCUs per 4KB
- Transactional write uses 2 WCUs per KB
- Can look at Operation = TransactWriteItems / TransactGetItems in CloudWatch metrics

Identify Poor Table Usage Patterns

Scanning tables for batch operations

- Export to S3 is often a better option (requires PITR)
- Can look for Operation = Scan in CW metrics

Not using Time-To-Live (TTL)

- Implement archival/tiering using streams
- Removing items via TTL is free

aws

Identify Poor Table Usage Patterns

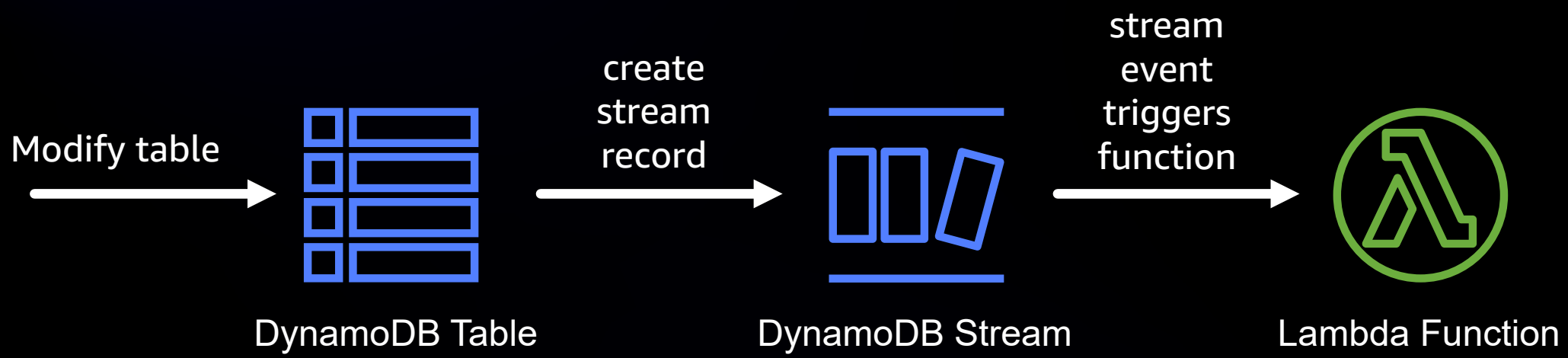
Not using AWS Backup vs. DynamoDB Backup

- AWS Backup offers tiered storage with lower costs
- Backups inherit tags and can be used for cost allocation tagging
- You must opt-in to AWS Backup

Using Global Tables for DR with high RPO/RTO

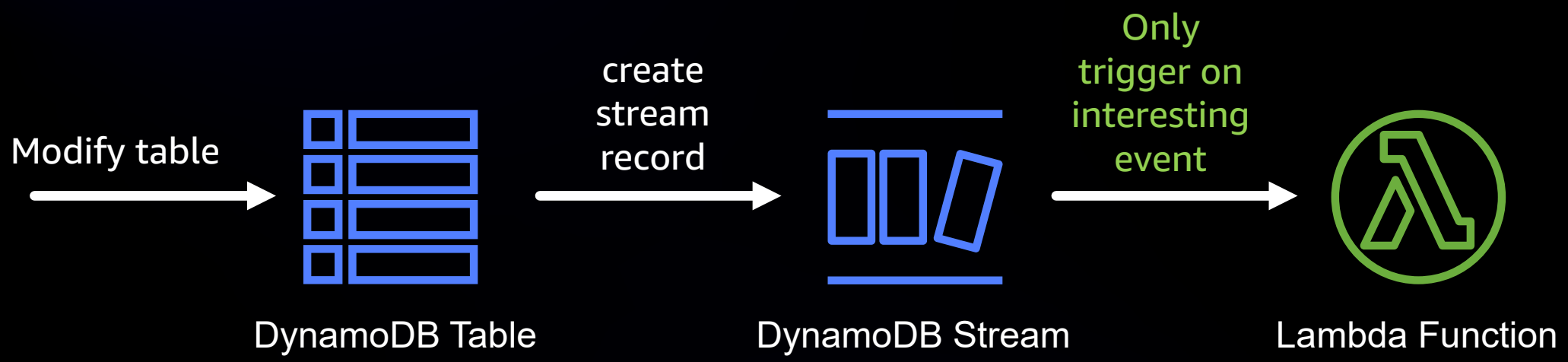
- Scheduled cross-region backups are easy with AWS Backup

Filter streams events for Lambda



Lambda functions subscribed to DynamoDB streams trigger on every event by default

Filter streams events for Lambda



Event filters let you only trigger functions on events of interest, including:

- Event name (INSERT/MODIFY/REMOVE)
- Partition Keys
- Arbitrary data in new or old version of item

How do I filter streams events?

```
{
  "dynamodb": {
     "Keys": {
        "Id": {
           "S": "someactor#somemovie"
        }
     }
...
}
```

Event data

```
{
   "dynamodb": {
      "Keys": {
         "Id": {
            "S": [{"prefix": "someactor" ]
         }
      }
   }
}
```

Filter Pattern

```
{ "Filters": [ { "Pattern": "{ \"dynamodb\": { \"Keys\": { \"Id\": { \"S\": [{"prefix:"someactor"}] } } } }" } ] }
```

Actual definition

aws

Filter streams events for Lambda

- DynamoDB streams GetRecords call is already free for Lambda
- Filtering events saves on Lambda costs, not DynamoDB costs
- 5 filters soft limit, 10 hard limit per stream (logical OR)

Q&A

Thank you!

Greg Krumm
Sr. Solutions Architect
AWS

Juhi Patil
Solutions Architect
AWS