BEST AMAZON WEB SERVICES (AWS) DEPLOYMENT PRACTICES:

Lessons Learned From 3 Years in the Trenches with AWS

CONTENTS

PREFACE	3
PART 1: AMAZON EC2 INSTANCE TYPES	4
PART 2: ADVANCED OPERATIONS	5
SUMMARY	14

PREFACE

Since the time of our founding, in June 2010, IDERA has been using Amazon Web Services® (AWS). It was critical to us to use cloud technologies to build our product, as it too would be running in the cloud. Before deciding on AWS, we evaluated many different cloud technology providers and tools. From

the time of that initial investigation in 2010, and spanning the years since, we have had the opportunity to experience the good, the bad, and the ugly.

From our perspective, we have sought the best technologies possible at the best price point. Amazon offers an amazing array of technologies, with elastic pricing that is favorable to most operating expenditure budgets. So, we offer this whitepaper as a perspective to organizations exploring

the public cloud and considering AWS. Further, we will provide information regarding popular components/ topics such as Amazon EC2®, cost allocation, EBS Provisioned IOPs®, and DynamoDB®.

PART 1: AMAZON EC2 INSTANCE TYPES

Choosing Amazon instance types can be tricky. Commonly, companies first select a starting infrastructure and then tweak it until they find a stable combination of assets. Through this process you may find that you are asking frequent questions like, “My database server needs more memory, which size do we upgrade to from a c1.xlarge?” or “I have these types of physical servers in my datacenter I need to migrate. What should I consider procuring in AWS to replace these with?” In order to make these questions easier to answer, CopperEgg created an **Amazon Web Services Cheat Sheet** to highlight some of the qualities of the different instances. This makes it easier to translate the hourly cost to a monthly cost, while showing the reserved instance cost as well.

While there are many interesting things to learn on the Cheat Sheet, a few are worth noting here:

- A **t1.micro** has the cheapest cost per CPU, but has horrible CPU steal.
- A **cg1.4xlarge** has the most expensive cost per GB memory.
- The cheapest cost per CPU system (that is NOT a micro) is a **cc2.8xlarge.**

Here is our list of invb stance sizes that we think are best:

- **c1.xlarge:** great compute, but still cost effective to have them spread across several zones. At publishing time, the c3 class is attempting to replace these.
- **hi1.4xlarge:** the ephemeral storage is INSANELY fast. We’ve seen massive IO on these, and great compute. Hint: use ‘hvm’ instead of ‘pv’ (paravirtualized) or you will have instances crashing a lot. If these traits are interesting to you, you will also want to consider the i2 class. They may actually be performing better.
- **cc2.8xlarge:** lots of compute, and 10gbit networking make these compute workhorses cost effective for heavy compute applications.
- **t1.micro:** for test systems…and nothing more!

PART 2: ADVANCED OPERATIONS

2.1 AWS Cost Allocation Tips and Tricks

Amazon announced their new cost allocation feature in Fall 2012. For many users, trying to associate different parts of a complicated infrastructure with a multitude of components and cost groups can be a struggle. For example, how do you assign and track costs as they relate to cost of goods sold (COGS), marketing, development, etc.? Many companies have raced to try to solve that problem by building tools to help you manage this scenario. Yet, many of them aren't able to truly provide you with what you need to be the most effective: the ability to see these components in the most simple of terms so you may slice and dice as needed.

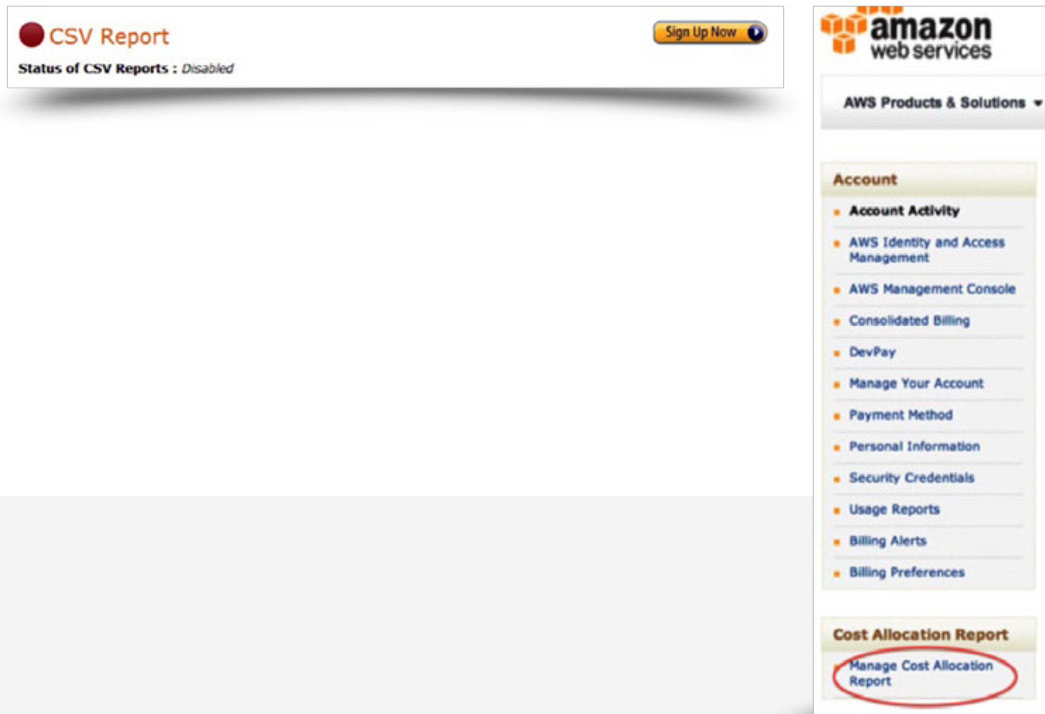
Luckily, AWS created the 'Cost Allocation Report' feature. If you sign up for Programmatic Billing Access, you can get CSV reports for your estimated and final charges from an Amazon S3 bucket that you specify. The file contains charges for the account, broken down by AWS product and individual type of usage. There are some critical steps you should take upfront to make sure you are set up for success:

1. First, you need to enable 3 options in your 'Billing Preferences' page: 'CSV Reports', 'Programmatic Access' and 'Cost Allocation Report'. These must be done in sequence. To get started, log into the 'Billing Preferences' page at AWS, and then enable 'CSV Reports'.

2. Second, enable 'Programmatic Access', where you provide an S3 bucket location for the CSV file to be uploaded to. This has to be a globally unique name, so be creative. You will then set a policy on the bucket to grant AWS to publish the CSV reports to that newly created bucket (otherwise AWS does not have access to it). You will be billed for any S3 usage, even if it is for the billing statements.

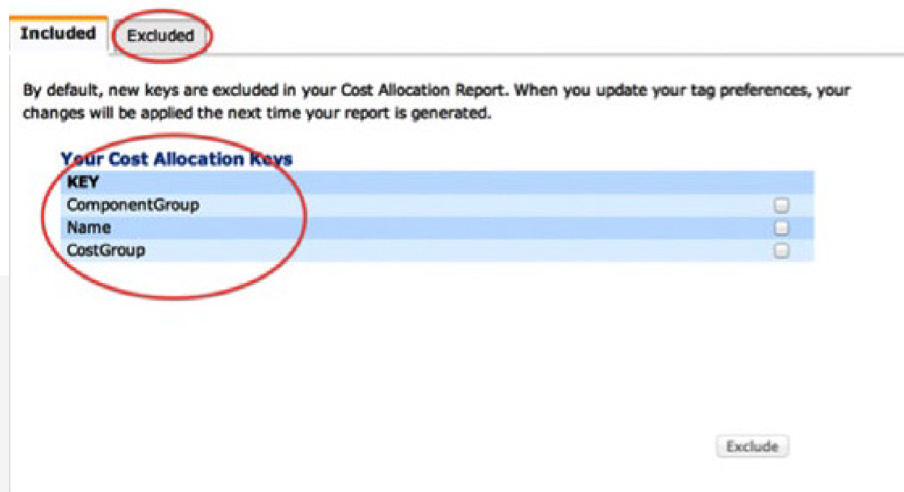
3. Next, enable the 'Cost Allocation Report' and begin tagging. A tag is a label you assign to an AWS resource. Each tag consists of a key and a value, both of which you define. AWS uses tags as a mechanism to organize your resource costs on your 'Cost Allocation Report'. You can tag lots of different AWS items like EC2 instances, EBS volumes, S3 buckets, autoscaling groups, RDS, VPC, etc.

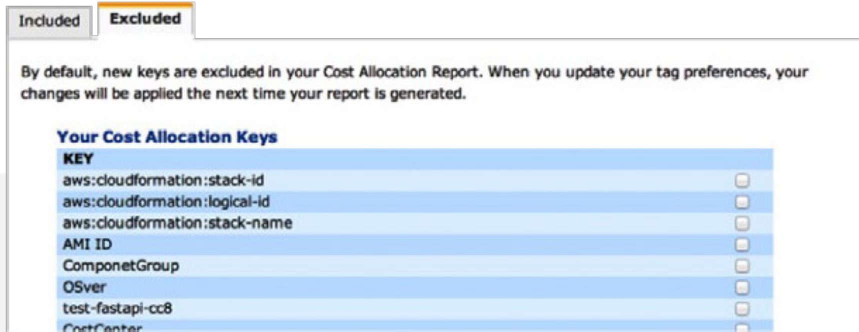
4. Once you have that set up, you can turn on the new 'Cost Allocation Report' billing feature. Log-in to AWS, and find the 'Manage Cost Allocation Report' option in the bottom left corner of the 'Account' navigation:



This allows you to enable the more complex cost allocation reporting, which uses the S3 bucket we just set up. Once you have done that, select the 'Manage Cost Allocation Report' page (which is also accessible through your 'Billing Preferences' page). You should see a screen that looks like this:

It this example, it is apparent that there are several keys not being included. In this example, we included three very important keys: ComponentGroup, CostGroup, and Name. Every EC2 instance, EBS volume, snapshot, etc., has all three of those keys filled in with a particular value. You can easily group a set of keys by various components.One example would be to see how much a certain part of your infrastructure is costing you.

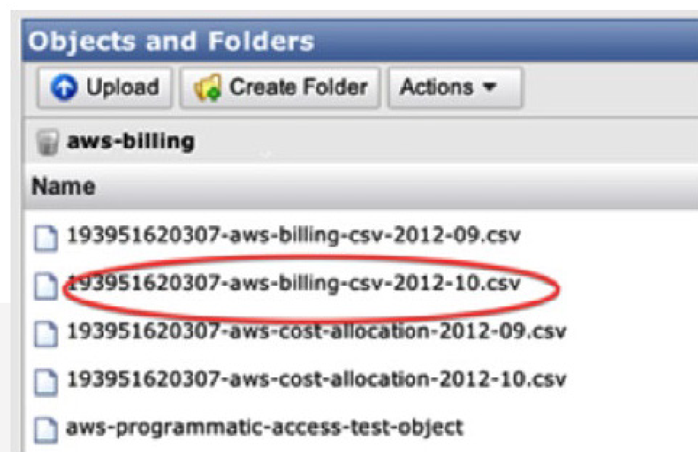




Grouping by cost is effective for determining various components as well – dev/test, marketing, COGS, sales, and name. The reason we have name as its own key, is so we can track how much a system has cost us over time, since sometimes we spin up/down a system, or change instance sizes, etc., but we keep the same name.

KNOWING WHAT YOU HAVE INCLUDED IS JUST AS IMPORTANT AS WHAT YOU HAVE EXCLUDED AS FAR AS COST ALLOCATION GOES.

It's nice to know that your Redis server has grown in cost over time. You need to make sure you pick values for keys (when you start an EC2 instance for example) that are consistent each time, so when you get the report it aggregates the data correctly. Consistency is key with this aspect. If you have a key of 'user' and values of 'eric' and 'eric a' then you will see two line items for that even though it was the same user. Once you have added the keys you want to the included list, you have to wait for the report to process. That initial report could take as much as a few hours to process. When complete, you will see something like this:



Rule #1 with tagging: be consistent

Tips and Tricks – Some Good to Know Restrictions on AWS Tags:

- Maximum key length: 128 chars
- Maximum value length: 256 chars
- Maximum tags for object: 10 (yep, only 10)
- 'AWS:' is a reserved prefix
- Key can be used only once per object
- You can only tag an object AFTER it is created
- Allowed chars regex: [a-zA-Z0-9 +-=._:/] After analyzing the cost of your own AWS environment, it is important to look at the storage options to determine what is (or isn't) working for your requirements.

2.2 Amazon Provisioned IOPS – EBS

IDERA uses a fair amount of storage for all the metrics we capture. At the time of publishing, we process over 100 billion records a day. To accommodate that amount of records, we use a variety of storage options for different reasons. Some of our data resides within MySQL, some in Redis, and the largest amount of our data (the metrics themselves) actually reside in a purpose-built datastore we internally call NestDB. One of the most important requirements for most data stores is consistent, high performance storage. Let's put down our wish list of requirements for storage:

1. Persistent – does not go away when the system reboots or crashes.
2. Stable – available when you need it.
3. Performant– does the requested amount of actions per second (be it operations or bytes).
4. Predictable – does those actions in a predictable amount of time over time.
5. Cost effective – can't break the bank to fulfill the requirements.
6. Amazon has a few block storage options (not object stores like S3): Instance store, Elastic Block Storage (EBS), and Provisioned IOPS EBS (PIOPS EBS).

IDERA processes over 100 billion records a day.

Instance Store

Instance store is fairly predictable since the storage is 'local' to the instance (it is not shared with other instances). The performance is mediocre and you can't get any more from it by striping it since you get a fixed amount. Unfortunately if an instance disappears, all storage will be lost. Instance store effectiveness:

Persistent: [   ]

Stable: [####]

Performant: [##   ]

Predictable: [###]

Cost effective: [#####]

EBS Volume

EBS volumes are persistent, and you can take snapshots of them and create new volumes from those snapshots (which is handy when you want to 'clone' a volume or use that for backups). You can also copy them to new regions, which is very powerful when you want to migrate a service or application to a new region for resilience, cost, etc. Since EBS volumes are on a shared storage platform, the performance is ok, but inconsistent. This is because most users use storage in bursts by doing many reads and writes in a batch (that's how applications typically use storage). If those bursts are high enough, or there are many customers using the same set of storage disks, the performance becomes unpredictable. Those bursts can easily overlap and max out the peak IO capacity or throughput capacity of the storage it is using. If you are trying to get a consistent amount of IO or throughput, you will quickly learn that EBS volumes are unpredictable. They typically perform quite well, and we recommend striping or RAIDing them together for increased performance (stripe, RAID 5, RAID 10, etc are all options but you should research and determine what works best for your needs). Of course, EBS is not free, so you end up paying for the persistence of that storage.

EBS effectiveness:

Persistent: [#####]

Stable: [###]

Performant: [##   ]

Predictable: [#   ]

Cost effective: [####]

Provisioned IOPS EBS

Provisioned IOPS EBS (PIOPS EBS) volumes are a newer offering from AWS, and are Amazon's answer to EBS performance capacity and predictability. PIOPS EBS have the same qualities of EBS volumes in that they can be snapshotted, copied, and are persistent over time. You can attach them to any system as well, just like EBS. The main difference here is simple, but very important: you can provision a specified amount of IO (operations per second, not throughput) for a given volume, and you will get it. What this means is that you can now get great performance that is extremely predictable with the other benefits of EBS. Of course, there is a price to pay for this – however it is relatively small.

EBS effectiveness:

Persistent: [#####]

Stable: [###]

Performant: [####]

Predictable: [#####]

Cost effective: [###]

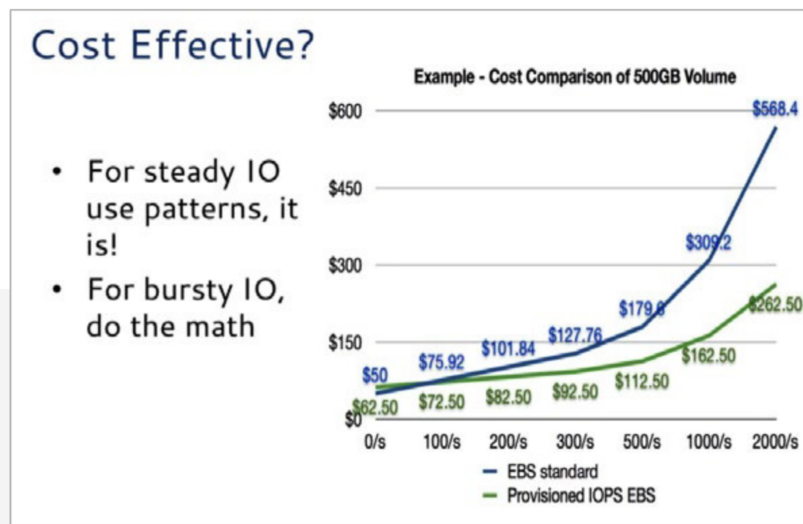
YOU CAN TEST YOUR OWN SCENARIOS WITH THE AMAZON SIMPLE MONTHLY CALCULATOR.

As an example, a 100GB EBS volume would cost about $10/mo for the storage, and we will use 200 IO per second for the whole month (that is about what you will get from it on average). This results in 520 million IO's for a month, so that would be an additional charge (EBS charges on volume and IO usage) of about $52 for a month. The total is roughly $62 for traditional EBS. Now, what about PIOPS EBS?

Provisioned IOPS are a bit more expensive for the capacity (25% actually), so 100GB of PIOPS EBS will run you $12.5 for the storage, and then we need to add the IOPS. Since you don't pay- per-use on PIOPS but instead pay-per-requested-use, it works like this: you pay $0.10 per provisioned IOPS-month. So, for 200 IO/s (using the same example), you would pay 200 * $0.10, or $20 additional, so $32.50. Wait – isn't PIOPS more expensive? Well, in some cases it is not, like the example here. You can test your own scenarios with the **Amazon Simple Monthly Calculator.**

Keep in mind though that when you provision 200 IOPS for a volume,you pay for those whether you use them or not, and you cannot burst over that. You will get 200, almost EXACTLY 200, IOs per second. We tested this many times, with different settings, and watched using our own metrics, and we also used some other tools like iostat, and we can say that the storage is extremely predictable and the performance is precisely what you request.

Some downsides to PIOPS EBS (Amazon will probably fix soon) is there is a max amount of IOPS you can provision per volume, which is currently 2000, and you cannot change this once the volume is provisioned. This means that on some Linux systems which only can have 11 volumes attached (this may be changing in the near future) you can get a max IO capacity of 22,000 IOPS for a single system. If you need more than that and can ignore persistence, we highly recommend the hi1.4xlarge systems.



Provisioned IOPS EBS volumes make wonderful database storage volumes due to the extreme predictability and high performance (about 10x per volume compared to standard EBS). Keep in mind though that you are still limited by throughput (the 'bytes per second' you can get) to those volumes. However, if you need to eek out some more storage bandwidth (keep in mind that storage bandwidth and network bandwidth travel on the same 1gbit network on standard instance types – unless you use a 10gbit system), then you can choose an EBS optimized instance. This uses a dedicated network interface for storage so your network and storage traffic do not fight over a single pipe.Overall, if you need predictable performance, Provisioned IPOS is best. If you have less than 100 ops/s on average, Standard EBS would be suitable to your needs.

2.3 Need Big Data -DynamoDB- the Good, the Bad, and the AWSome

2.3 Need Big Data -DynamoDB- the Good, the Bad, and the AWSome DynamoDB is a key-value store with extras. You can store items in it based on a key, like any other key-value store, but it can do much more than that. For instance, you can store many individual key-value combinations in a single item. You group these items together into tables, which organizes items together in some logical way. You are also able to store many items in a single table. So, for example, you could have an item for each person on earth. You might have a table for each country, and each

table would hold items for everyone in that country. Each item then might contain different data about the people: city, state, and zip code for US residents. But for other countries, they won't have states, so they may have provinces, zones, or some other delineation.

That's perfectly ok to do in DynamoDB – in fact, as long as you have a single key name (the 'primary key' which indexes all the items together) that exists in every item in a table, you're good to go.

The rest of the keys in the items can be whatever you want, and you can store nearly anything in the values for those as well. Amazon calls keys within an item, 'attributes', and every attribute is a name/value pair. The values can be things like strings, numbers, binary, or even string sets, number sets, and binary sets. For our example, we might pick a primary key with a name of 'fullname' or maybe a unique single-digit identifier like 'person_id', as long as everyone (each item) has that attribute.

Some Things We Love About DynamoDB:

- No operational cost since it is hosted by Amazon (also known as the 'Kool-Aid').
- Triple zone redundancy (means we don't have to think about that part, much).
- Price is ok considering what you get (3 zones, growable IO performance, and 'infinite' expansion).
- Super easy to get started

Cool DynamoDB features

One feature that we find particularly powerful in DynamoDB is composite primary keys. The reason composite primary keys are awesome is because it turns this key- value store into a powerful time series data store with key-value combinations that widely surpass the capabilities of other simple key-value stores. Specify one key as your partition key (how it divides up the keys), and another as your range key. The important things to know here are: your partition key should be very unique (hashes like md5 or sha1 are great), and the range key is great if you use something like seconds from the epoch to do time series range queries.

DynamoDB Performance

We tested using cc2.8xlarge instances and c1.xlarge instances at Amazon and used Ruby and the Amazon Ruby SDK. We wrote a simple script that built a set of items that mimicked our dataset, and keys that made sense to us (for what it's worth, our keys were fairly big, compositing several pieces of information into a single key). First we had to create a table to hold the test data. We started with a table that would take 10,000 inserts per second, and 5000 reads per second. Creating the table took us quite some time. So, you will need to plan accordingly and not depend on create tables at run-time. This also means that if you are going to pre-create tables, you will want to do it as late as possible, since you begin paying for that table as soon as you create it, whether it is being used or not.

The Bad DynamoDB news

Once the table was created we were ready to run the test. Running a single instance of the test script showed right away that the round-trip time and authentication was going to be troublesome without doing some more parallel testing since we were testing the single 'put' interface (not the 'batch put' interface which will perform better). Once several tests ran in parallel, we achieved the precise throughput we provisioned on the table. We were very pleased to see the performance of the table! Overall, the performance was exactly what they claim, and extremely consistent over time.

The Bad DynamoDB news

DynamoDB is missing just one thing that may not be important to everyone. While it wasn't a deal breaker, it was certainly something The ones we missed the most were:

- No expiration of items (or tables). So, you must delete them and expire them on your own.
- The scan feature: don't use scan unless you don't care about performance. It's slow, and has to walk all items in the table. So, this is not a solution for the delete problem, and you don't want to use it for run-time use.

Bad news aside, DynamoDB is great for new companies or startups getting set up on AWS and needing a 'just works' key-value store solution that they can trust.

SUMMARY

Overall, we believe Amazon's technologies and elastic pricing are favorable when exploring the public cloud. Yet, it is necessary to have adequate knowledge regarding AWS pricing, cost allocation, Provisioned IOPS, and DynamoDB before setting up an AWS environment. View the live demo or send us an email with additional questions.

IDERA'S SOLUTION

To help solve some of the challenges that database professionals face concerning the cloud, IDERA provides a portfolio of products to ease the burden of managing complex database environments and simplifying migrations of data.

IDERA makes the complex easy.

Learn more