Cost Attribution for Multi-cloud / Multi-platform Applications

Mya Pitzeruse

September 23, 2020

Abstract

This paper details a solution to attributing cost across common infrastructure. The original target for the solution was a hybrid-cloud infrastructure. This led to a comprehensive, yet flexible design. As a result, its concepts are generic and support both cloud and non-cloud ecosystems. The goal is to provide a single view into spend across platforms. It needs to to support a large variety of use cases, from leaders analyzing spend to teams understanding the cost of systems they consume.

Introduction

Products run across a variety of platforms. Many use a combination of platforms to perform work efficiently. Managed database services allow teams to focus on their application logic. Apache Hadoop is popular amongst the data science community for analyzing data [3]. Meanwhile, platforms like Kubernetes have gained popularity amongst engineers for orchestrating systems [5]. Each of these platforms play a role in the development of a single product.

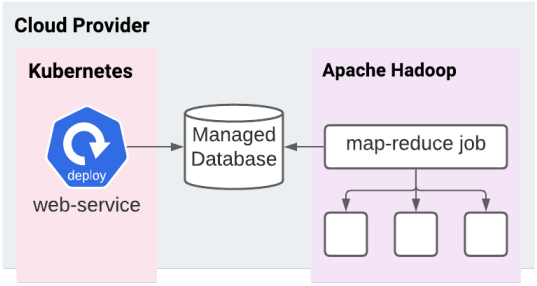

Figure 1: A multiple platform application

Smaller organizations are often less concerned about attributing spend to each product. In most cases, these organizations need to know if they're profitable. As an organization grows, they often add new products. With each new product, this figure grows more complex. As a result, the need to attribute spend increases as well. Executives often want to know how much each of their products cost. This helps determine what their return on investment is. When asked, leaders often need to take time to track down this information.

Solutions like *cloudtamer.io* help organizations better manage their spend on cloud providers [2]. But these solutions often lack visibility into clustered environments. You can pair this with in cluster solutions like *kubecost* provide visibility into the cost of Kubernetes pods [11]. But in the end, you wind up back to where you started. Needing to query several sources for information.

While these account for common cases, it fails to consider less common ones. Suppose one part of your organization runs a service on behalf of the rest. You might apportion usage of that service out to it's heaviest consumers.

Background

In November 2018, Indeed was feeling pain-points scaling our Apache Mesos infrastructure [4]. Some efforts towards a Kubernetes based ecosystem had been underway.

The first focused on a short-term migration. It would allow us to move off Apache Mesos while preserving the structure of our ecosystem. The second focused on how to run upstream Kubernetes safely and securely. As we migrated to Kubernetes, teams built out some cost attribution. We wanted to report on changes as teams underwent these migra-

tions.

We raised similar questions as we began our journey to the the cloud. During this time, we considered several solutions. Many of them focused on solving in cluster or out of cluster costs, not both. To accomplish this, many organizations pair solutions together to fill in gaps. However, many platforms lack this capability out of box.

In discovering a lack of comprehensive solutions, we started in on some research. Understanding how information is already being collected and reported was key. This would allow the solution to hook into many existing sources of information. It also needed to be able to allow for other types of solutions later on too. The following were some observations about systems that we analyzed.

- One or more workloads compose a product
- Workloads have a unique platform identifier and need resources to run
- Resources between each data set varied with some commonality
- Each represented an exchange of services between one group and their customers

These discoveries lead to the design a development of a common reporting platform. Designed with decomposition in mind, the platform allows for the addition of layers. This allows organizations to start out broad and get more granular over time.

Concepts

This section contains concepts that are fundamental to the solution. Understanding them will help map them to potential internal concepts.

Workloads

One or more workloads compose a product Workloads represent a variety of processes an organization might run. They can be a long running, public facing web services or background jobs. Workloads can run across a variety of platforms but are often deployed to one.

Resources

Workloads need resources to run. Common cases need things like CPU, memory, and disk. At the same time, resources are available through a variety of platforms. Kubernetes finds room within the cluster to run containers based on their requests. Virtual machines provide a similar allocation of resources, while also providing virtualized hardware. Regardless of where and how the workloads run, resources are uniform.

Standard Units of Measure

With anything we measure, standardization and documentation of units is important. Without it, one group might code against one standard while another codes using a second. Some standardized units of measure include:

- **CPU** is measured in **millicores**
- **GPU** is measured in **cores**
- **Memory** is measured in **bytes**
- **Disk** is measured in **bytes** and **iops**
- **Transport** is measured in **bytes**
- **Time** is measured in **milliseconds**
- An **API** is measured in **requests per second**
- **Energy usage** is measured in **kilowatt-hours**

Finer grained units can be used if needed.

Uniform Resource Names

A uniform resource name (URN) is a globally unique, persistent, location-independent identifier [8]. While deprecated in 2005, they continue to see use today. To help understand URNs a bit more, let's consider an example using RFCs. The following URN is for RFC-2141 [9], the specification for URNs.

```
urn:ietf:rfc:2141
```

This URN uses the *ietf* namespace with the *rfc* namespace-specific string. To ensure uniqueness, you must register namespaces with the Internet Assigned Numbers Authority (IANA). For simplicity, we will use the *workload* namespace. This namespace has **not** been registered and only serves as demonstration.

A URN in this document refers to specific workloads. Let us consider a database deployment MySQL named appdb [12]. At some point, you migrate appdb from MySQL to PostgreSQL [7]. During this time, both a MySQL and PostgreSQL resource will exist with the name appdb. As a result, you should scope your identifiers. The block below enumerates several examples of further scoping by class and kind.

```
urn:workload:aws:642135246531
urn:workload:compute:host:appname
urn:workload:compute:vm:appname
urn:workload:compute:pod:appname
urn:workload:compute:container:appname
urn:workload:storage:postgres:appdb
urn:workload:storage:mysql:appdb
urn:workload:storage:rds:appdb
urn:workload:storage:aurora:appdb
```

Amazon's ARN

Amazon leverages it's own resource naming scheme called ARNs [14]. Like URNs, ARNs represent a unique resource in AWS. The solution described herein should support ARNs in place of URNs. It's recommended that URNs are consistent for simplicity and ease of use.

Double-entry Bookkeeping

Double-entry bookkeeping is a common practice used by most (if not all) regulated financial systems [1]. It is often used to keep accounts balanced and as an error detection tool. Implementation of such a system requires several key details:

- Each entry added is immutable.
- Every entry added to one account, requires a corresponding entry in the other account.
- Independent tracking of charges (debits) and payments (credits).

Double-entry systems are often implemented as a ledger [15]. This allows the system to track transactions for a given account. In this solution, we augment the traditional approach to double-entry bookkeeping with hierarchy. We do this to model the real-world exchange of resources between groups. Consider Figure 2.

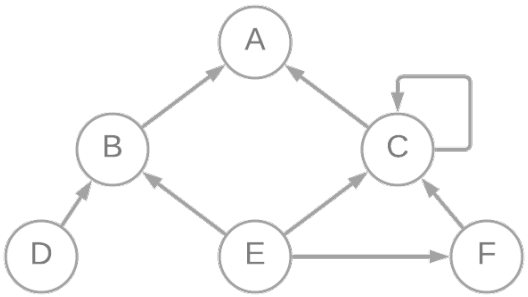

Figure 2: Exchange of service graph

This graph shows how projects can build upon one another to deliver value to the business. For example, E uses services provided by B, C, and F. At the end of the day, we want to attribute E the resources that they consume.

Implementation

When it comes to implementation, I've found there are two schools of thought.

The first approach is to use real world dollars. There can be some challenge in this approach. If you work for an international organization, you'll need to choose a base currency. Then, you'll need to consider things like exchange rates. In many ways, this information is valuable for product owners. It communicates how much their system costs and helps them determine a price for their service. In the wrong hands, this information is dangerous.

The second approach is to use a basis point type of system. This approach allows you to represent the underlying usage as a percentage. One problem with this approach is that when you scale a cluster up, usage goes down. This can be confusing if you haven't deployed changes recently, but see a shift in your usage.

At the end of the day, we need a combination of approaches. We can use basis points to apportion out usage at each tier. When displaying or analyzing information, we can provide a currency value. The currency value could be synthetic, an estimate, or a real world value. This approach allows for the most accurate representation of a given products cost.

Apportioning Usage

Apportioning usage is the process of dividing a pool of resources amongst its consumers. In this solution, we do this using basis points. Basis points represent fractional percentages. They are often represented as one hundredth of a percent. We use basis points as it helps address a couple of challenges.

1. Normalize measurements to a common unit
2. Better handle cases of over-commit resources

To determine basis points for a process, we need to know what the total available resources are. The total (T) will be the larger of what is *available* (common case) and what is *reserved* (over-commit). We apply the process to each resource we want to apportion out. The equations below compute the total availability of a given resource.

$$T_{reserved} = \sum_{W}^{workloads} max(W_{reserved}, W_{usage})$$

$$T = max(T_{available}, T_{reserved})$$

Once you've computed the total amount of a resource, you can distribute basis points. The equation below calculate basis points for a single resource for a single workload.

$$basispoints = \frac{max(reserved, usage)}{T} * 100$$

At the end, we have a set of basis points for each resource we want to charge for. We need to convert this set of basis points to a single basis point. A simple approach would be to take the average. Taking an average of percentages requires some care [10, 13]. In this case, we can take steps to ensure taking an average of percentages is safe and representative of the underlying usage.

We should keep in mind that these basis points are being distributed at each node in our graph. This ensures that we know the full scope of data at the time we calculate percentages. This also ensures that the sample size of the data is the same for each resource we calculate. This means that we compute basis points from a constant or a computed value from the same record set. Because of this, we can safely leverage an average of basis points. Let us consider the following allocations.

- **10000** basis points from GPU reservations
- **1000** basis points from CPU reservations
- **3000** basis points from memory reservations

The above workload averages out with **4667** basis points. Suppose this workload ran in a shared cluster with one other workload. That workload claims the remaining resources (0 GPU, 9000 CPU, 7000 memory) which averages **5333** basis points. Together, they sum to **10000** or 100%. This will allow us to annotate every edge with a single weight.

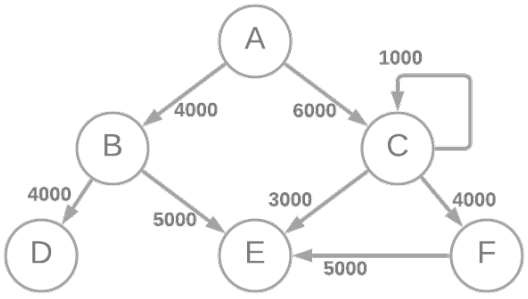

Figure 3: Weighted exchange of service graph

In a properly weighted graph, the sum of all weights for a node should be less than or equal to **10000**.

$$10000 \leq \sum_{edge}^{node} edge_{weight}$$

Apportioning Cost

As said earlier, apportioning usage is not enough. There needs to be able to be a monetary value. Now, this monetary value could be a real value, like a bill. Or, it could be a synthetic value that someone made up.

It works by attaching a cost to a root workload in the graph. From there, we can derive all subsequent charges using common, well known algorithms.

Path Enumeration

In most cases, we want to know the cost of a single product. Path enumeration provides a list of all paths between a pair of vertices in a graph [6]. We can apply this same concept using a single starting vertex and a direction to search. The following is a path enumeration for A in Figure 3.

1. A, B, D
2. A, B, E
3. A, C, E
4. A, C, F, E

Similarly, we can apply this in the opposite direction. The following is an inverse path enumeration for E.

1. E, B, A
2. E, C, A
3. E, F, C, A

This process is key to attributing cost.

Computing Cost

Using the enumerated paths, we can compute the total cost for a given workload. We compute costs using the weights on the edge of the graph in combination with root costs. The equations are as follows.

$$\forall_{w}^{workloads}$$

$$cost(w) = \sum_{p}^{paths} cost(p)$$

$$cost(p) = root(p) \times \prod_{edge}^{p} (weight/10000)$$

$$root(p) = p[len(p) - 1]_{cost}$$

To help explain, let's consider D. It has a single path, D, B, A. Figure 4 highlights this path in blue.

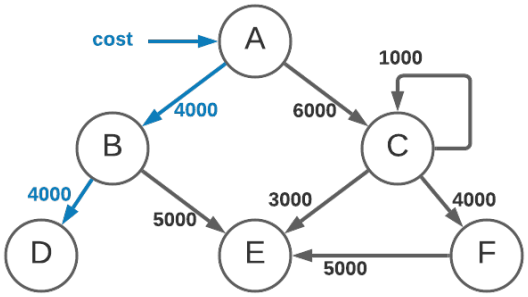

Figure 4: p1 for D

Using the information in the graph and our equations, we can compute a cost for D.

$$cost(p_1) = A_{cost} \times 0.4 \times 0.4$$

$$cost(D) = A_{cost} \times 0.16$$

If A costs **100 USD**, then D costs **16 USD**. Similarly, we can consider a more complicated example. E has three paths. They are documented in the section on Path Enumeration and highlighted in Figures 5, 6, and 7.

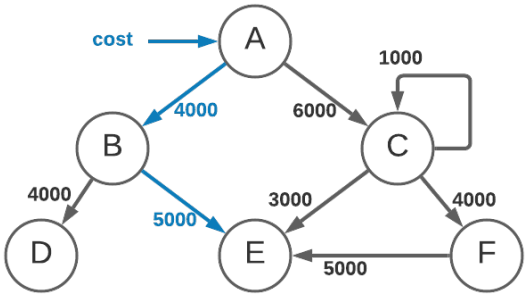

Figure 5: p1 for E

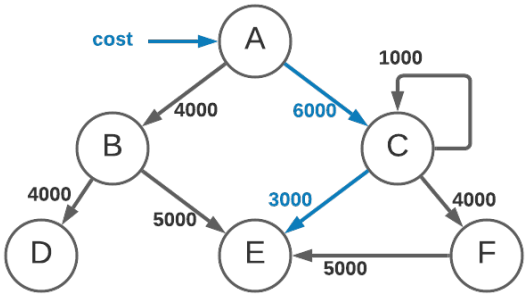

Figure 6: p2 for E

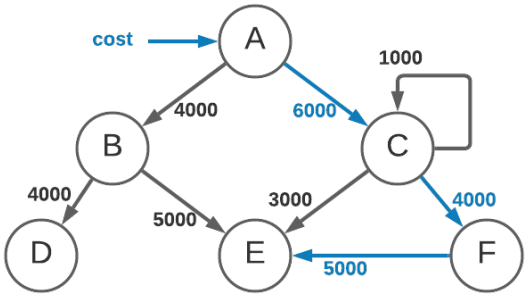

Figure 7: p3 for E

While more complicated, the process is the same.

$$cost(p_1) = A_{cost} \times 0.4 \times 0.5$$
$$cost(p_2) = A_{cost} \times 0.6 \times 0.3$$
$$cost(p_3) = A_{cost} \times 0.6 \times 0.4 \times 0.5$$

$$cost(E) = A_{cost} \times (0.2 + 0.18 + 0.12)$$
$$cost(E) = A_{cost} \times 0.5$$

If A costs **100 USD**, then E costs **50 USD**.

Reconciling Cost

As we are dealing with a financial system, we should ensure we apportion cost correctly. In accounting, we use reconciliation to ensure that money leaving an account matches the actual money spent. We can use the same process to balance our computed costs against the provided costs. Table 1 demonstrates how we balance and reconcile accounts.

w	Spend	Attributed	Balance
A	x	10000	0
B	$x \times .4$	9000	$x \times 0.04$
C	$x \times .6$	7000	$x \times 0.18$
D	$x \times .16$	0	$x \times 0.16$
E	$x \times .5$	0	$x \times 0.50$
F	$x \times .24$	5000	$x \times 0.12$
			$x \times 1.00$

Table 1: Reconciliation

We compute this using the current spend and the current attributed amount. The attributed amount shouldn't factor in anything attributed back to your own team. Using the following equation, we compute the remaining balance for each item.

$$balance(w) = spend \times \left(1 - \frac{attributed}{10000}\right)$$

To help solidify this, let us consider the example from earlier. If A cost **100 USD**, the balance for each would be as follows.

w	Balance	USD
A	0	$0
B	$x \times 0.04$	$4
C	$x \times 0.18$	$18
D	$x \times 0.16$	$16
E	$x \times 0.50$	$50
F	$x \times 0.12$	$12
	$x \times 1.00$	$100

Table 2: Balance sheet

This shows that this approach to cost attribution balances.

The Ledger

Accountants leverage ledgers to track transactions between two parties. Consider Table 2 and Table 3. They detail the fields and data types of the ledger. Each entry in the ledger represents a workload specified by a URN.

Field	Explanation
datetime	Time of transaction
payer	Who received services (from)
payee	Who provides services (to)
type	A debit or credit
usage	The usage
urn	A uniform resource name
detail	A break down of charges
labels	Organization specific metadata

Table 3: Fields

Field	Type
datetime	datetime, int64
payer	string
payee	string
type	enum(debit, credit)
usage	int32
urn	string
detail	map[string]int32
labels	map[string]string

Table 4: Data types

Usage can store basis points or a monetary value as shown in Table 5. Storing data as basis points make it quick and easy to record allocations over time. We can later compute a monetary ledger using the basis point one and a set of bills.

Payee	Payer	Usage
A	B	4000
A	C	6000
B	D	4000
B	E	5000
C	C	1000
C	E	3000
C	F	4000
F	E	5000

Table 5: Example basis point ledger

Parting Thoughts

Multiple Roots

Throughout this document, we've demonstrated the process using a single product with a cost. In practice, this graph is often more complex than that. Figure 8 demonstrates how cost might come from multiple roots.

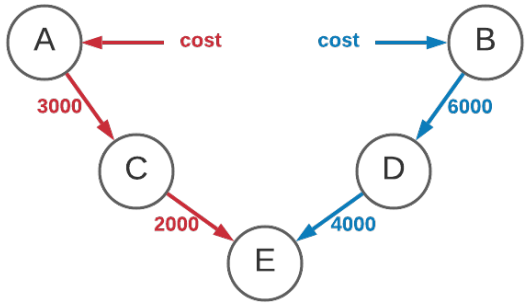

Figure 8: Multi-cost example

Regardless, the process is still the same.

$$cost(p_1) = A_{cost} \times 0.3 \times 0.2$$
$$cost(p_2) = B_{cost} \times 0.6 \times 0.4$$

$$cost(E) = (A_{cost} \times 0.06) + (B_{cost} \times 0.24)$$

Importance of Modeling

Data modeling plays a major role in proper representation of resource consumption. While it's easy to apportion resources at a cluster level, it often misses nuances. For example, CPU heavy nodes or GPU nodes cost more than common compute. As a result, resources may be weighted allowing preferential distribution of cost. With proper data modeling you ensure the cost of those nodes are apportioned properly.

Data Storage

The ledger can take many shapes and forms. Storage should use a consistent system. Eventually consistent systems can leave you in troublesome positions when reconciling cost. Indexing labels on the ledger allows for the categorization of edge data. The enabled further analysis and grouping into hierarchies, such as your organization structure.

Equitable Distribution

In some cases, products may not distribute all their resources amongst their consumers. This often leaves them with an overhead that they are responsible for. In this case, groups may choose to redistribute their overhead amongst their consumers. This process is often referred to as equitable distribution. The following equation computes a redistribution value to distribute amongst your customers.

$$redistribution = \frac{10000 - billable}{consumers}$$

References

[1] bench.co. *A Relatively Painless Guide to Double-Entry Accounting*. URL: https://bench.co/blog/accounting/double-entry-accounting/. (accessed: 2020-09-22).

[2] cloudtamer.io. *cloudtamer.io*. URL: https://www.cloudtamer.io/. (accessed: 2020-09-22).

[3] Apache Software Foundation. *Apache Hadoop*. URL: https://hadoop.apache.org/. (accessed: 2020-09-23).

[4] Apache Software Foundation. *Apache Mesos*. URL: https://mesos.apache.org/. (accessed: 2020-09-23).

[5] The Linux Foundation. *Kubernetes*. URL: https://kubernetes.io/. (accessed: 2020-09-23).

[6] Roberto Grossi. "Enumeration of Paths, Cycles, and Spanning Trees". In: *Encyclopedia of Algorithms*. Ed. by Ming-Yang Kao. New York, NY: Springer New York, 2016, pp. 640–645. ISBN: 978-1-4939-2864-4. DOI: 10.1007/978-1-4939-2864-4_728. URL: https://doi.org/10.1007/978-1-4939-2864-4_728.

[7] The PostgreSQL Global Development Group. *PostgreSQL*. URL: https://www.postgresql.org/. (accessed: 2020-09-23).

[8] IETF. *Uniform Resource Names (URNs)*. URL: https://tools.ietf.org/html/rfc8141. (accessed: 2020-09-22).

[9] IETF. *URN Syntax*. URL: https://tools.ietf.org/html/rfc2141. (accessed: 2020-09-22).

[10] Indeed.com. *How to Calculate Average Percentage*. URL: https://www.indeed.com/career-advice/career-development/how-to-calculate-average-percentage. (accessed: 2020-09-16).

[11] kubecost. *kubecost*. URL: https://kubecost.com/. (accessed: 2020-09-22).

[12] Oracle. *MySQL*. URL: https://www.mysql.com/. (accessed: 2020-09-23).

[13] Roberto Reif. *Why you should be careful when averaging percentages*. URL: https://www.robertoreif.com/blog/2018/1/7/why-you-should-be-careful-when-averaging-percentages. (accessed: 2020-09-16).

[14] Amazon Web Services. *Amazon Resource Names (ARNs)*. URL: https://docs.aws.amazon.com/general/latest/gr/aws-arns-and-namespaces.html. (accessed: 2020-09-23).

[15] Wikipedia. *Double-entry bookkeeping*. URL: https://en.wikipedia.org/wiki/Double-entry_bookkeeping. (accessed: 2020-09-22).