CO$T-EFFECTIVE CLOUD APPLICATION ARCHITECTURES

Sumit Nigam
Consultant Engineer
EMC Corporation
sumit_only@yahoo.com
# Table of Contents

The architecture of limits ........................................................................................................... 3

Cost and metering ...................................................................................................................... 5

- Pricing policies ....................................................................................................................... 6
- Cost of a transaction ............................................................................................................... 6
- Leaky Abstractions ................................................................................................................. 8

Monolithic and distributed architectures .................................................................................. 10

- For the N\textsuperscript{th} time .................................................................................................. 12

Incentives and strategies ........................................................................................................... 14

- Cache It = Cash It! ............................................................................................................... 15
- Data transfer .......................................................................................................................... 16
- ‘Compressing’ the bills ......................................................................................................... 18
- Statelessness ......................................................................................................................... 19
- Lambda functions .................................................................................................................. 20
- Memory leak = revenue leak ............................................................................................... 22
- S‘low’ clients = ‘low’ cash ..................................................................................................... 25
- Blocked resources = blocked revenue ................................................................................ 26
  - Actor model ....................................................................................................................... 27
  - Exploiting parallelism ....................................................................................................... 28

Hybrid Cloud – Another dimension to cost effectiveness ...................................................... 30

- EMC Hybrid cloud ............................................................................................................... 31
- Workload partitioning ......................................................................................................... 31

The container and VM debate ................................................................................................. 35

Projecting with data ................................................................................................................ 37

  - Back-of-the-envelope calculations .................................................................................. 39

Annotation ............................................................................................................................... 40

Conclusion ............................................................................................................................... 42

References ............................................................................................................................... 44

Disclaimer: The views, processes or methodologies published in this article are those of the author. They do not necessarily reflect EMC Corporation’s views, processes or methodologies.
The architecture of limits

Thomas Jefferson, the 3rd President of United States famously said¹ – “Never spend your money before you have it”. This should resonate with how traditionally enterprises had to put an upfront investment in their infrastructures even if they were unable to leverage it immediately. Further, enterprises had to continue investing in infrastructure in keeping up with projected demands. Once those demands were over the infrastructure would lie idle having incurred costs even before any substantial returns. Allow me the liberty of coining this investment as “Jefferson effect”, although no such term exists today.

It is precisely this “Jefferson effect” that cloud computing addresses with its pay-as-you-go model. The idea is to treat computing itself as a utility. The billing is also modeled to take into account only the procured computing units as it is with other utilities such as electricity.

One of the strengths of cloud computing is its elasticity property. So, what exactly is elasticity? Yes, that ‘something’ which we were taught in Physics. Or was it in History? Never mind. Elasticity in a cloud platform is the ability to not only spawn new infrastructure on demand but also shrink it as and when the need for it reduces.

However, this article is not about benefits of a cloud platform. These, and other benefits of cloud computing are reasonably well documented and understood². This article is also not about how cloud providers stack up the entire solution and draw up their reference architectures. Readers are directed to learn more³ about them if needed. Isn’t it interesting how more involved concepts are left out of technical papers asking readers to read about them? But then that is the norm, so we will also follow the same.

No matter whether we are building a video-sharing application on cloud or a healthcare application, it is well understood that good application architecture benefits from some core principles. Ability to create an effective architecture is no mean feat. Rarer still, is the ability to build application architecture for the cloud platform. I’d term this as rare because most of the time an existing application needs to be migrated to cloud and that is a tremendous challenge for any technical architect.

Would there be anything more challenging than these? What about effectiveness of this application architecture from cost point of view?

Forrester projects⁴ that 45% of total infrastructure technology (IT) spending will be on cloud services by 2020. The verdict has been out for long. Cloud applications are here to stay.
Controlling the application cost is the technical architect’s (architect) responsibility. In this article, we will distill those principles through which an architect should reduce the cost of operating their application on cloud. This is taking the challenge to its ultimate limit. Thus, this article is about the architecture of such limits.
Cost and metering

Before we set out to optimize our costs of operating on a public cloud, we must understand what constitutes our costs. Stated in a different way we should understand what we are being metered for by the cloud provider. This restating, of course, does not change anything but makes us believe that we at least understand the problem better. Answering this question, however, is important because once we have an idea of what contributes to our costs, we can attempt to derive a cost effective architecture.

In any public cloud, the main factors contributing to costs include compute, storage, and bandwidth usage. Then it follows that we must leverage those architectural principles which impact these considerably. Amazon, Google, Microsoft, Rackspace, and others are some key players in this space. Google published an interesting study\(^5\) which showed that cloud pricing is not following Moore’s Law\(^6\). Pricing for public cloud has fallen 6-8% annually ever since they became prevalent but the hardware itself has reduced by almost 30% annually in accordance with Moore’s Law. This sizeable parity between the two is responsible for continuous reduction in cloud prices. The pressure to retain customers has also led to multiple price wars over the years. The intensity of the same can be gauged from the fact that Amazon itself has lowered prices close to 50 times in last 5 years or so\(^7\). Makes one wonder how much profit margins they had before and if that is not the case, then how are they making money now. We will let their shareholders worry about that.

All cloud providers have an on-demand metering model as the most fundamental pricing strategy. Besides these, all of them offer pricing models such as tiered pricing, reserve instance(s) pricing and customizations to those models as well. The idea is to allow customers as much flexibility in choice as possible. The very basic tenets of cloud computing are based on the concept of flexibility.

Compute is usually charged per-instance hour consumed from the time it is launched to the time it is terminated or released. This category of metering falls under on-demand instance pricing. The storage and network bandwidth are charged on a per gigabyte basis. Tiered pricing helps mostly with storage and network wherein as consumption increases, the metering for those resources tends to decrease. Reserving the infrastructure pricing requires customers to block the resource upfront and make a one-time fee. It may be noted that bandwidth is charged only for data transfer out of the cloud and not for data transfer into it. Making a note of such common practices is useful for any architect.
Pricing policies
What should ideally contribute to pricing?

1. The losses that cloud providers have incurred in previous year.
2. The quantum of business they lost to competitors.
3. The value they have promised to their shareholders but failed to deliver.

Luckily, those are still not a part of pricing policies of any provider; at least not on paper! So, let’s get real.

While compute, storage, and network are the fundamental building blocks of all cloud pricing models, there are other equally important aspects to factor in. These include (but not limited to):

1. License costs are always passed on to users (as expected).
2. Cloud providers have a notion of database service tiers. As an example, Microsoft Azure cloud defines tiers for database service. Different tiers support varied levels of concurrent transactions. Of course, the costs rise as better tiers are chosen.
3. Type of compute instance chosen along with memory configurations.
4. Different hypertext transfer protocol (HTTP) requests such as GET, PUT, COPY, etc. are charged differently.
5. Backups can be charged on gigabyte basis too.
6. Load balancers which are used to distribute traffic are usually charged based on the quantum of data they process as well as their run times. Providers usually allow for a certain number of load balancing rules. For example, Google Compute Engine (GCE) allows up to 5 such configured rules. Beyond that, every rule is charged extra.
7. Most clouds provide tools to monitor applications. Some amount of basic monitoring is usually free. Detailed monitoring using those tools is charged.
8. Number of requests made to some underlying service such as database or message queue service.
9. Number of objects stored in some underlying service such as active directory.

Cost of a transaction
The complaints from enterprises that there are hidden costs to hosting applications on cloud have refused to die down. Cloud providers would instead blame the enterprises for the lack of planning. They would like to clarify that their cost calculators are all available online and that
their pricing policies are well documented and unambiguous. That would sound like taking the customers to task by indicating that there’s none as blind as those who will not see.

Whatever the truth behind this debate, one thing is clear; developing a good sense of what component is contributing to how much cost can be very insightful. This can be done in terms of total queries each service is contributing to (if cost was from database perspective) or the number of requests hitting a service over a period of time or amount of data transferred out from the service, etc.

A good metric to measure is cost per transaction. Simply put, a transaction is a distinct piece of operation. A transaction could be uploading of some media file or streaming some content. It could be a simple database call or a call which spans multiple services. It should be translated in terms of doing something valuable for the end users.

Since, we’ve seen that compute, storage, and network constitute some of the key parameters based on which a user is charged, it makes sense to develop the cost model based on how each transaction impacts these. Overall, there are models and theories which help ascertain these costs and they can be quite involved at times. A very simple measure to put in place is to monitor overall latency that each transaction introduces. Costs are usually directly proportional to run time of a compute instance. In a given application flow, which spans multiple components on the cloud with each component comprising multiple transactions, capturing latency introduced by each service can itself lead to insightful discoveries. As an example, the component which is contributing to maximum latency is also in all likelihood consuming maximum compute time.

This analysis can be very useful when releasing a new version of the service. It can help ascertain whether we have added to “cost” of the service. This is extremely valuable data because services which contribute more to cost can be made to run “when needed”. This is taking advantage of elasticity feature of the cloud we touched upon earlier.

Keeping track of such metrics also helps correct decisions about choice of say, wrong storage model or wrong compute instance type on cloud. Operational expenses can go completely haywire if wrong decisions are not corrected soon.

Cloud applications require some mindset change on the part of architects. It is very convenient (and is the norm) to host the application on some hardware and completely forget about it. It is
important to keep a close (and not closed) eye on the costs of deploying the application on the cloud.

**Leaky Abstractions**

Abstractions are a way of hiding complexities of underlying implementation. They are a wonderful concept. Instead of dealing with low-level details, abstractions make it possible to deal with building blocks. Abstractions are quite prevalent in software.

Unfortunately, abstractions cannot truly insulate the user from underlying complexities. For example, network file system (NFS) is an abstraction which allows programs to treat files located remotely as if they were present locally. Programming languages provide application programming interfaces (API) which let us treat a file across the network using same programming constructs as a file present on local disk. As an example, Java programming language provides file handling APIs which deal with both local and remote files through same methods. One such method to deal with renaming or moving a file is `renameTo()` method in its package java.io File class. This method succeeds in renaming the file on local disk but can fail for files on NFS with multiple clients getting successful responses to their concurrent rename operations. This means the underlying implementation of NFS protocol leaks into high level Java program and this method failure needs to be handled specifically in the program.

This abstraction is further leaky in the sense that when a network connection becomes slow or goes down, the file is no longer available. The underlying network has made its presence felt! This leaky abstraction now puts a penalty in the form of code to be written to work around these issues.

Cloud computing suffers from leaky abstractions too. On a cloud, every resource is abstracted out and cloud users get a unit of compute or network or storage and are also charged accordingly. However, leaky abstractions have two main implications here. The first one has to do with architecture itself and the second one is about cost. Let us discuss them briefly.

An aspect which affects applications hosted on cloud is the ‘noisy neighbor’ problem. Cloud is based on the notion of multi-tenancy which essentially means that the same underlying infrastructure is shared between multiple cloud users. Although cloud providers attempt to isolate these physical resources through a virtual layer; these abstractions do not always help. As an example, a tenant of the cloud can hog a lot of input/ output (I/O) resources impacting other tenants sharing the same underlying storage. This is a case of leaky cloud abstraction.
The implication of leaky abstraction on costs is more implicit. As discussed, noisy neighbor problem can lead to slowdown of other tenant applications in the vicinity. This in turn can lead them to automatically spawn new instances because of some underperforming nodes and hence, incurring extra costs. A way out may be to purchase instances in surplus. That can work, except that it is usually costly and inefficient.

Active monitoring can somewhat alleviate the problem. The idea is to terminate the underperforming instance and spawn a new one. Reserving a higher capability cloud instance addresses noisy neighbor problem too.

An important first step to take when moving to cloud is to create a virtual local area network (VLAN). This ensures a good isolation of not only data packets between VLANs but also of safety from another VLAN whose security may have been compromised.
Monolithic and distributed architectures

Eric Raymond in his influential book The Cathedral & the Bazaar introduced cathedral and bazaar as two metaphors of software development. The cathedral involves a huge amount of time to build. Once complete, it stays unchanged. The bazaar, by contrast, is modified and enhanced regularly by the people working in it. Raymond used the bazaar metaphor for open source development.

In fact, these metaphors are so powerful that they can easily be extended to how earlier applications were built compared to now. Until a few years back, most of the applications were monolithic in nature and once developed were difficult to modify. This should resonate with Raymond’s Cathedral metaphor.

A lot of present day applications are complex and require adapting them to ever-changing requirements. This is the bazaar metaphor. A lot of complex applications are, by nature, also distributed. The idea behind such architectures is to decompose huge monolithic applications into smaller and cohesive services which evolve over time.

What really is a service? The answer to this question can be found in any dictionary indicating that it is a useless question. One can also argue that service can take different meanings depending on the context. Armed with this new insight, let us state the question again. What
really is a service in the context of distributed architectures? Now, we are making some progress. Services are application programs which provide a particular function through a well-defined interface. They could be an operating system background process or inventory management component. They may or may not be exposed over web. Fundamental to building services is the separation between its interface and the concrete implementation.

![Distributed architecture built with loosely coupled services](image)

**Fig 2:** Distributed architecture built with loosely coupled services

What is the right way to build architectures? Should we build a monolithic application or lean towards distributed architecture? Depends on how we are feeling on that particular day! On a serious note, as with other things in life, the answer to this question is “it depends”. Distributed applications are difficult to architect and even more difficult to troubleshoot. Monolithic architectures are still used widely and will continue to remain so. They may not be the best way according to some, but they’ve been a success.

Most of the successful architectures have usually had modest beginnings and those can be traced back to some form of monolithic application architecture. As the need arose, changes
were done to them and they made way for distributed applications. With the advent of cloud platforms, those distributed applications were modified to adapt to cloud.

**For the N\textsuperscript{th} time**

Principles used to build applications on cloud naturally derive from those of distributed computing. It is a good starting point to review some of these key principles. Many books\textsuperscript{13} dedicate a number of pages glorifying them. Without these principles, we probably won’t fail miserably but we are also guaranteed not to succeed fabulously. Like that vintage wine which does not fail to delight, we too will repeat them for the n\textsuperscript{th} time.

1. **Design for failure** – One of the key guiding principles for building distributed architectures is to design for failure. This implies that one must assume that compute, network, and storage will all fail and factor this into the architecture. Disks will get full, networks will get flaky, dependent services will crash and hence, an architecture must take them into account.

2. **Loosely coupled services** – Principles of service oriented architecture\textsuperscript{14} (SOA) help build application architecture as services which do not get tightly bound to one another. The key advantages include ability to scale services independently, ability to contain faults within a service better, and flexibility to plug-and-play a concrete service with another which provides the same functions as the one being replaced.

3. **Elasticity** – One of the key tenets behind cloud application architectures is their ability to take advantage of auto-scaling feature of the cloud. For this, the service should allow for launching it easily on new instances and terminating those instances which are no longer required.

4. **Graceful degradation** – It is important to understand the most critical business aspects of the application and ensure that they continue to operate despite temporary failures. It is okay to shut down services which are not as critical as others as far as business criticality is concerned. Shutting down lesser critical services can alleviate the problem by preventing the fault from propagating further into business critical services. As an example, in an online stock trading application it is acceptable to not show users their order history compared to not allowing the user to trade at all.

5. **Automate** – Humans can make mistakes. Hence, their intervention should only be required during emergencies. It is best to automate as much as possible. Installation of services should be simple and well documented. Automated scripts which are well tested go a long way in ensuring smooth releases and fault prevention.
6. **Plan backup and restoration appropriately** – Applications hosted on cloud require backup, archival, and restoration capabilities provided by cloud providers. Planning geo-redundant architecture goes a long way in securing application from a business continuity perspective.

7. **Monitor proactively and extensively** – Monitoring and alerting should be completely non-intrusive to core application functionality. However, they should be exhaustive enough to give complete visibility into health of the services.

In next few pages we will leverage some of these principles to become cost effective. Of course, not all of them minimize costs. As an example, security on cloud has always been a major concern and continues to remain so. However, it has no direct bearing on cost effectiveness of architecture so we do not touch upon this subject.
Incentives and strategies

We now have a good background on what contributes to cost on cloud as well as some guiding principles of distributed computing. We can attempt to build some good cost effective measures into our architectures.

Cloud scale economics requires us to have a firm grasp on the incentives that we are working towards. To me, *Freakonomics* is undoubtedly the best book when it comes to exploring the beauty of incentives – “Any incentive is inherently a trade-off; the trick is to balance the extremes”. Sometime we get so obsessed about the incentives that we end up paying more than what we may have really gained. Donald Knuth’s words – “Premature optimization is the root of all evil” should resonate well here.

Pareto’s 80/20 rule gives us initial guidance. That is, applications spend 80% of their time executing only 20% of their code. It follows that our focus should ideally be in knowing that 20% and optimizing it for cost effectiveness. This gives us a good boundary for our target incentives. Relying upon the words of wise men always pays, so let us also make a note of it. Of course, Pareto’s rule does not work if our application is coded horribly, to begin with. In such cases, the best bet is to indeed call friendly neighborhood Spiderman!

An important point to note is that license costs are to be borne by users. Cloud providers allow users to bring their existing licenses and use them on cloud. This should definitely be the first choice if the user has licenses handy. Because license costs are passed on to the tenants of the cloud, it makes sense to consider cost effective alternative of enterprise software. As an example, Ubuntu Linux is free and if an application does not need RedHat Enterprise Linux (RHEL) specifically, then Ubuntu should be considered.

These may seem obvious but sometime the obvious also fails to penetrate the thick skull. This also happens when existing applications are migrated to cloud. Enterprises try to do a 1:1 mapping of what they were running their applications on in their data centers to what they would need on cloud. In doing so, the cost inefficiencies also move to cloud.

Two terms have gained substantial popularity in software these days. The first one is Greenfield project. These are a category of applications which lack any constraint imposed upon them by any prior work. Hence, they offer a free hand to architects to put in any strategy they deem fit. On the contrary, a Brownfield project is one which refers to developing new software in conjunction with existing legacy systems. This makes it trickier to evolve them and even contain...
costs. The guiding principles of distributed computing and SOA discussed previously can benefit a Brownfield application as well.

**Cache It = Cash It!**

The advantages of caching are well known from reducing latency point of view. A caching solution can also minimize a bill significantly. To understand this better, we need to recall that cloud providers charge us based on number of requests made to an underlying service. Using a caching layer can help alleviate not only load on the underlying services but also reduce the number of calls to them. As an example, if caching can reduce the number of calls to a database by 30%, those savings will immediately reflect in our next bill. Consider this for millions of transactions over a period of time and the savings become substantial.

The same is also true of other services such as storage service. Most of the storage services will also be accessed using representational state transfer (REST) style HTTP calls. Those calls are all charged by numbers.

There is an indirect benefit too. Caching helps reduce latency and if we have opted for a pay-as-you-go model, our compute will be taxed less and will result in faster termination of operations. This in itself can result in savings due to presence of a caching layer. It is a good idea to cache results of complex operations.

Caching has another advantage in terms of cost effectiveness. We discussed that cloud providers do not charge for inbound data transfers. However, they do charge for outbound transfers. This has implications for our cost effectiveness if the application is deployed in two geographically distinct cloud locations. In cases where data needs to be transferred from one cloud location to another because our application demands it, we will be charged for data transferred out of one location. Sounds like 'hidden cost'?

In such cases, deploying a caching solution at either site can tremendously reduce bandwidth costs. Add to this the costs reduced due to fewer cross-site requests and savings can begin to accrue quickly.
Similarly, putting a caching layer in front of a web server can cache a lot of static content such as JavaScript, images, and HTML pages.

Using a computational cache can alleviate load on compute nodes and reduce the amount of time taken to calculate an expensive operation. Reduction in total latency has a direct impact on amount of time we will need a compute node for and time is money, literally.

It is very important to keep track of memory footprint of caches. Cache solutions also bring added challenge to handle stale entries. Unless we like to live dangerously, it is important to implement an appropriate eviction policy in cache design. This is important to keep cache size in check as well as in ensuring that items which are needed often are not evicted.

**Data transfer**

Cloud providers charge based on outbound data. It follows that we must try to minimize data being sent out from our application. A caching solution in front of our web servers cannot help here. We are being charged for number of requests that hit the load balancer. Plus, we are paying for compute resource in direct proportion to its usage. This is assuming a pay-as-you-go pricing model.

There are a couple of cost effective measures which can help here. HTTP provides ‘Expires’ header which can be embedded in the response for components contained in the HTML page. When a browser program encounters this header it saves the value of this header internally. The value indicates how far in time does the component expire and hence, should be rendered again. Between now and expiry time, the browser is free to use its internally saved content.
This reduces one call per request for the page where this image is embedded. Imagine a scenario where this page is requested by millions of users in a given day. We have just reduced the amount of data transfer out of the cloud. The same header can be applied to cascading style sheets (CSS) as well as JavaScript. This is one of the simplest cost effective techniques. Steve Souders makes a very compelling case with his path-breaking book18 “High Performance Web Sites” where he proves that 80-90% of response time for any given web request is spent in making HTTP requests for all components contained in the HTML page. These include images, scripts, style sheets, etc. Using techniques to reduce the number of roundtrips between the client browser and cloud hosted application can have a tremendous impact on the overall bill.

Similarly, another useful header in HTTP is “Last-Modified”. This header allows browser to determine whether the component received by it previously (possibly days or months back) is modified from that time or not. If not, browser would not make an additional request to the application for the same.

We may have only looked at this strategy from a public cloud point of view so far. How about a hybrid cloud? When someone asks a good question like this, a good strategy is to shift the goalpost by posing a return question because that is easier than answering. So, the return question is what exactly is a hybrid cloud? Wikipedia defines hybrid cloud19 as an integrated cloud service which is a composition of more than one cloud (public and private, for instance) that remain distinct entities but are bound together, offering the benefits of multiple deployment models.

Let us answer the original question by considering a popular use case. Enterprises may host their web tier within their public cloud but store all data on a private cloud. This has an advantage in terms of bandwidth usage that data has to be transferred into public cloud and hence, does not incur any cost. If latencies are acceptable between the two clouds for such a data transfer, then this is a good strategy to start with. Eventually, some data can be moved over to public cloud. Keeping dynamic data closer to compute reduces round trip latencies.
I once reviewed an application which did not filter data enough at database end before returning the same to clients. The queries were of the format given below.

```
SELECT * from <table> where <field> = <value>
```

The query fetched all the columns of the table satisfying the condition and those were returned to the browser. The client was not even interested in more than 1/5th of returned data and discarded it. Such inefficient piece of code are a strict no-no in any setup, more so in a cloud as this outbound data transfer adds up to our bills quickly.

Database queries definitely require very close review when an application is migrated to cloud. Database indexes consume space too. It can be useful to review if a composite index can serve better than having individual indexes.

**‘Compressing’ the bills**

Network bandwidth consumption can grow very quickly as any anyone downloading movies would testify. Getting serious, our strategies should be around minimizing footprint of data moving out from cloud.

Compression of outbound data can give us substantial reduction in our bills. Compressing data before storing it can be useful because as we reduce storage footprint, we also reduce our costs. Compression can be computationally intensive and hence, we may end up exhausting more CPU cycles. A trade-off needs to be worked out because we pay for compute resources instead.

Data being backed up can be compressed for sure. Recall that we also pay for using backup services by gigabytes per month as well as per some quantum of requests per month. Here, reducing backup storage footprint by compressing the same can be a good cost saving.

We discussed earlier how data transfer across two geographically located sites within a cloud can add up to bandwidth usage and how caching can help there. Further gains can be achieved by compressing data before relaying it.

Coupled with caching, this strategy will reduce outbound bandwidth usage considerably.

Compression can also be applied at HTML level. HTTP header ‘Content-Encoding’ is useful in indicating to the browser that the contents are compressed.
It is definitely useful to consider compressing any text response being sent from the application. These include extensible markup language (XML) documents or JavaScript object notation (JSON) documents. Compressing images does not help as they are already in a compressed form. Various compression algorithms exist and most of them trade off performance for size and vice-versa. Potential gains can be up to 65-70% of size reduction easily. This is a very significant gain in terms of costs for outbound transfers.

**Statelessness**

One of the main design principles followed by distributed architectures is to make services stateless. Ascetics believe that statelessness is a way to nirvana. Some less fortunate ones spend a lifetime in attaining statelessness and still do not achieve anything! We do not have that liberty at cloud scale. We need to attain it much more easily and quickly. That is because statelessness has a direct bearing on scaling applications and has an impact on costs. Let us understand statelessness next.

We discussed how elasticity property of cloud is so advantageous when it comes to scaling up. It is also equally important to take advantage of it when scaling down. One of the main impediments to scaling down services is holding user specific state information within those services. When a service holds user specific information it becomes ‘bound’ to the user in a way. This holding state on behalf of the user is termed as session stickiness. Expanding it further, consider what would happen if the next request from the user went to another node altogether which hosts the same service but does not contain user state information?

How does this affect cost? Elasticity implies the ability to not only spawn new service instances when needed but also give up resources when not needed. Giving up instances when they are not needed is very important for cost savings. But giving up an instance is not easy when it retains state on behalf of the users.

Can we suddenly take down an instance if it was having active sessions and data pertaining to those requests only available locally on disk or in memory? We can, of course, if we intend to frustrate users of our application by knocking them out in the middle of their transactions. Frustrating others is still a favorite pastime of many. Bottom line, we cannot dispose of instances at will, if we hold user specific information inside those instances.
Also consider what happens when we have an application hosted on two separate data centers. How can a transaction which has some state stored in the first data center be handled in the other data center without replication of stored state data? Replication would require data to be sent from the first data center into the second, adding to our bandwidth costs. Besides bandwidth costs, we need to also store this data on the second site as well. That just added to our monthly bill. Imagine this scenario for millions of transactions over a period of time.

![Diagram](image)

Figure 4: Keeping state locally impedes both scalability and cost effectiveness

An ideal cloud application should continue to perform even if an instance crashed or was decommissioned. This requires no user state (session data) being stored local to service. A good strategy is to have a separate tier (built upon EMC Pivotal GemFire, for instance) which all services have access to and store information on. Then instances can be reduced as and when demand reduces without impacting any active user.

Another effective strategy could be to push some of this data to clients using technologies such as AngularJS or HTML5. Of course, pushing state to client over internet may not be a great idea when state data is huge.

Ascetics are right. Statelessness is a way to nirvana at cloud scale too.

**Lambda functions**

Mathematics has certain applications such as equation solving or computing integral calculus which require functions to be passed as argument to some algorithm. Such functions can very
well be expressed with lambda calculus. Lambda calculus was introduced by Alonzo Church as a formal system for expressing computations based on function abstraction. The strength of lambda calculus lies in the fact that all computable functions can be stated and computed using it.

One very powerful use of lambda functions is in implementing callbacks. This makes them perfectly suited to follow event-driven programming models. It allows for executing a lambda function in response to some external event. For example, one could create lambda function to be launched and executed when a new file is uploaded or a new entry is made in a database or some output from a connected device is available, etc.

Figure 5: Cost-effectiveness of lambda functions on cloud

There is no need to write polling loops or other compute-intensive code to monitor for changes. Instead, the lambda function will be executed in response to an event as callback logic.

Stateless lambda functions allow scaling them linearly. This allows responding to incoming events as quickly as possible by launching as many copies of the function as needed.

Cloud providers such as Amazon have started supporting user provided lambda functions. Users can upload their lambda functions as callbacks for relevant events which various Amazon Web Services (AWS) platform services generate. The key benefit gets clearer when we consider that these lambda functions lie passive until the events to trigger them are not
generated. This ensures that compute instance hosting the lambda function is launched only on a need basis. Once the lambda function completes its execution, the instance is terminated. Clearly, this is a very cost effective measure. There is no need to over-provision compute resources for executing lambda functions which would be the case with polling programs. Polling requires looping through logic to determine whether there is a change that the application is interested in.

Lambda functions on cloud will become important for building very reliable, very concurrent, and very cost-effective applications.

One aspect that makes the lambda functions so compelling on cloud is the speed with which the function can be launched and executed. This makes them a good choice to be run over virtualized containers versus VMs. We will look into containers and VMs in a section devoted to them, later.

**Memory leak = revenue leak**

Memory leaks are always dreadful. They have the potential to bring down the entire application. In a distributed environment, as one service instance leaking memory goes offline, other instances end up sharing that load. In a typical round-robin load balancer set up, this load would be divided equally. However, with increased load the remaining service instances tend to lose memory even faster. As those service instances become unresponsive, new ones can be spawned but a huge number of those requests will be reissued by the users.

Of course, a good design should ensure that reissuing those requests would not cause any side effect. This is achieved by making operations idempotent. In computing, an idempotent operation is one that can be repeated multiple times with same inputs yielding same results. Even with such a good design, those requests would lead to increased bills due to them landing on our applications more than once.

Ideally, there will be an upper limit set on how many instances we are willing to spawn. However, once that limit is exhausted we are again looking at service-wide outage and that too, at the cost of having spawned numerous new instances. Bottom line, memory leak is revenue leak.

A good way to insulate entire applications from an underlying service level fault is to prevent its propagation. As the memory-leaking service becomes unavailable, it impacts all those services which are dependent on it. A good way to tackle these kind of issues it to protect other services.
One way to do the same is to build flows as asynchronous message exchanges. This has a direct bearing on scalability of the application besides impacting the cloud costs. The idea is to introduce a message queuing service in between the services. Services communicate with each other by sending messages to this queuing service which in turn delivers them to the services which have subscribed for those messages. That way the services do not have direct dependency on each other.

Another effective way is graceful degradation of the application. This usually requires some bookkeeping in the form of number of faults a service returns or the time that it takes for a service to return the response. The idea is to build a mechanism which decouples the originating service from another service if the other one faults more than the allowed number of times in a given interval or its average latency exceeds a threshold within a given timeframe. In such cases, messages are queued and delivered to the target service later. How to build such a mechanism?

As always, we can either write an intelligent algorithm to do the decoupling automatically or possibly, even do that manually. Doing it manually would require an army of support staff just for that. We could do that too, except that it seems costly and our paper is about cost effectiveness. So, let’s rule that out.
Our choice then, is an intelligent decoupling algorithm. The ‘circuit-breaker’ is a simple and effective algorithm to achieve graceful degradation. It has three states which translate into ‘circuit-open’, ‘circuit-close’, and ‘circuit-half-open’.

During the closed state, messages flow through unhindered. However, as responses to those messages are received they are recorded for bookkeeping. Responses may indicate failures or be used to capture time being taken to process them. For example, if the number of errors increases beyond a threshold within a given timeframe, the circuit-breaker is transitioned into open state. In such a state, messages can only be queued to be retried later. After a certain time elapses the state is transitioned into half-open state. In such a state, some messages are trickled through and responses recorded. If the response continues to indicate threshold being violated, the state shifts to open state again, else the circuit-breaker closes and allows smooth passage of messages across to the target service.

![Circuit-breaker state transitions](image)

Figure 7: Circuit-breaker state transitions

Sometimes, the target service performs an important function and cannot really be decoupled completely. In such a case, it is a good idea to keep some tailor-made response ready. As an example, in case of online shopping application if the recommendation service fails to respond it is better to show some standard recommendations rather than fail the order placing service. The
idea behind graceful degradation is simple. Identify the most important business functions and isolate them from failures of other underlying services. The idea should be to fail-fast rather than hang the invoking service by putting some time-outs when communicating with any other service.

**S'low’ clients = ‘low’ cash**
Imagine a similar scenario like the previous one where instead of leaking memory, the underlying service is strapped for resources. Take an example of slow clients. What happens when a client connects with our service and sends the transfer control protocol (TCP) SYN signal and upon receiving a SYN signal from our end, never completes the three-way handshake by issuing the ACK? In such a case, we just lost one connection waiting for either client to send the ACK or close the socket or network level timeout to kick in. This is exactly one of the weaknesses exploited by distributed denial of service (DDoS) attacks.

However, it is even possible to run into this scenario with valid customers of our application. What happens when a large number of our clients are running very slow connections? What if a number of them are using a very slow WiFi connection? Very soon all our front-end connections can get blocked leading to a situation similar to a DDoS attack. The only difference is that these are legitimate users.

As more and more front-end services choke, the elasticity will kick in by spawning more and more of those services. This is a situation similar to memory leak. Of course, we should terminate choked instances and spawn new ones. However, the price we are paying here is not necessarily with number of instances spawned but also due to number of requests which are being reissued by clients because they have hit one of the choked service instances. We've just allowed slow clients to drain out a good amount of our cash.

If this issue were coupled with poor design of maintaining session data within each service node, we will not be able to easily terminate these instances after spawning them as well. Slow clients seem to pose a tricky problem. Who said that users are always good for application?

Our architecture can benefit significantly if we position a reverse buffering proxy in between our application and the clients. What exactly is that?

A proxy sits in between the clients and the application. It intercepts all requests and can be configured to buffer entire request from client before opening a dedicated connection with back-end service. This has a great advantage. The back-end service is protected against two
important aspects. One, the slow clients are prevented from opening and blocking a dedicated connection with back-end service and those clients can be timed out at proxy end itself in case they fail to send data within a stipulated timeframe. Two, during times of traffic bursts, the proxy can provide surge protection to backend services.

A very popular buffering reverse-proxy is nginx\textsuperscript{20}. Nginx buffers responses from service before relaying them to the clients. This is very useful when the clients are slow. Nginx stores the response in its internal buffers and waits for full response to be received from the backend service. On the reverse path, it also insulates clients from slow backend services.

Nginx is designed to handle a huge number of connections concurrently. This is due to it using multiplexing and event-driven notifications extensively. Connections to nginx are handled by highly efficient and non-blocking, single threaded worker processes. A single worker is capable of handling thousands of concurrent connections per second.

**Blocked resources = blocked revenue**

Thus far, we have avoided discussion on scalability to be able to focus more on cost aspects. Scalability is usually directly proportional to cost savings but that is not always the case. Sometimes, truly scale-out architecture may not necessarily be cost-effective. An example is that of an application which is truly scale-out but fails to shrink its service instances when not needed. Another case is that of an application which is horizontally scalable but probably uses a large number of instances due to inefficient data structures and algorithms used to build it.
In recent years, the onus has been on building applications with truly non-blocking constructs. This has a direct bearing on being able to reduce the number of instances needed to serve the same number of clients. Non-blocking paradigm can be used almost anywhere we have shared resources. Shared resources sooner or later lead to contention and that limits scalability. Wherever we have contention, we need to use some mechanism to lock the shared resources for exclusive use. This leads to blocking other requestors. Any kind of blocking that we introduce eventually leads to either dissatisfied users or spawning of more service instances to tackle the traffic load. Shared database connection pools are another example of the same. Shared resources can be a free ticket to hell on a very busy day!

One way to address the problem is to provide independent paths of execution. For example, if a service communicates with multiple other services then usually a thread pool is deployed to facilitate the same. Imagine now, that one of the backend services becomes slow. This would eventually lead to all threads in the pool getting blocked or affected. This also limits communications with other healthy services because all threads in the pool are blocked. Providing independent thread pools to communicate with each downstream service contains the fault to that pool and keeps paths to other services healthy.

**Actor model**

At a finer level, a key concern with multi-threaded applications is that they require synchronizing on some shared-memory. Modeling the flows with asynchronous messages is always beneficial. This is not only a great idea at architecture level, but the same concept is behind a lot of computer languages which support actor-based programming. In such executions, concurrency is modeled with asynchronous actors instead of shared memory. Actors are concurrent, thread-like entities. Instead of locking on a shared resource, the actors communicate by sending messages to each other. Synchronization is achieved because a message can be received only after it has been sent.

An actor has an associated work-queue which is similar to mailbox for receiving its messages. Sending a message to another actor is as simple as just putting it in the mailbox of that actor. The benefit is that ‘posting’ the message to other’s queue does not block the sender. Similarly, the receiver is not interrupted when it gets any number of new messages in their mailbox queue.

Actors usually iterate reading messages from their mailboxes and doing work in response. Take the example of incrementing a counter using both thread-based model as well as actor-based model. In a multithreaded application, each thread usually tries to increment the counter by
grabbing an exclusive lock which prevents others from corrupting this variable. This ensures that the value changed by the lock-holding thread is committed to main memory for other threads to utilize. This has a major drawback that all other threads are now blocked waiting to acquire the same lock. While they are blocked they are not able to do anything useful.

Contrast this with the actor-based model. There can be a single actor responsible for incrementing the variable value. All other actors who want to increment the value put a message, i.e. ‘increment’ into the mailbox of this actor. Since putting the message does not require acquisition of any lock; the actors do not block each other. They are free to continue with their work. The actor responsible for incrementing runs a loop and performs increments without blocking as there are no contenders.

**Exploiting parallelism**

Lock-free implementations have a simple goal. Minimize contention between threads and avoid context switches. That would keep them all busy doing things and not wasting time waiting for contention to resolve. It is estimated\(^1\) that within a few years, processors could have 4,096 cores, server CPUs could have 512 cores, and desktop chips could have up to 128 cores.

The cloud providers will be quick enough to deploy these to benefit the end users. It then follows that as hardware becomes more and more efficient; software has to use more and more parallelism to keep that hardware busy.

The number of cores will increase exponentially over the next 10 years. The ability to take advantage of multicore processors will be important to reduce an application’s hardware footprint as we move forward. As algorithms operate more in parallel they lead to quicker termination of the instances and hence, reduction of overall expenses.

Most programming languages have taken steps to help build parallelism into applications. For example, Java 7 introduced Fork-Join\(^2\) framework. As the name suggests, this framework is designed for work which can be represented in smaller pieces and executed recursively. The splitting of a larger task into subtasks continues until the subtask cannot be divided any further.

One may not fail to notice the divide and conquer paradigm at its heart. This way, all processor cores are kept busy by worker threads. This is a very effective way to keep cloud compute resources fully utilized instead of spawning more of them because the program can use available processors optimally.
Even if a worker completes its task ahead of others it resorts to ‘work stealing’ which amounts to taking work of other workers who are still busy. This approach scales linearly with number of CPU cores. On a lighter note, work stealing should appear very counter-intuitive to many of us. Who likes ‘stealing’ other’s work just because they have some free time at hand?
Hybrid Cloud – Another dimension to cost effectiveness

An architect’s job is not only about creating cost effective software architecture. The architect must also have a deep understanding of hardware and organize the architecture around taking best advantage of what is available versus what requires a new investment.

We’ve briefly touched upon hybrid cloud in earlier sections. Enterprises would like to retain their legacy infrastructure and continue to leverage it. A hybrid cloud therefore, is a very cost effective model if both existing and new investments can be made to operate in tandem. It allows choice between public and private clouds to divide their workloads.

It is easy to see that clouds will increasingly become hybrid. This puts additional responsibility on architects. They need to develop cost effective architecture which can leverage both aspects of hybrid cloud and deliver value. This will require careful study of architectural pieces and choose the right cloud for them to run. Eventually, the architecture gains from both public and private clouds.

Hybrid clouds are all about freedom. Joe Tucci, EMC Chairman and Chief Executive Officer recently stated[^24] – “We believe there's a huge opportunity in private clouds, huge opportunity in public cloud, but what customers really want is hybrid cloud.”

Because we have a penchant for believing survey results, I steal some of those from Gartner[^25], which projects that 50% of large enterprises will use hybrid cloud by end of 2017.

Hybrid clouds require answering some questions which broadly consist of:

1. How will some instances of the same service coexist on public cloud as well as on premise private cloud?
2. How would the communication be handled between the two?
3. How will public cloud-hosted services access data resident in private cloud and vice-versa?
4. How will data copies across the two clouds be kept in sync?

These questions will be discussed in a section on workload partitioning. The strategies which we discussed in preceding sections apply to hybrid clouds as well. It should easily follow that cost effective architectures on hybrid cloud are a perfect winning strategy.
EMC Hybrid cloud
As organizations adopt hybrid cloud there will be a need to manage those cloud environments, broker them, and ensure SLAs are met. An added dimension of cost effectiveness comes from the ability to set up infrastructure and be live quickly. Time-to-market is still the key to cost effectiveness.

EMC has been at the forefront in building a fully engineered Enterprise Hybrid Cloud Solution (EHC). EHC is disruptive in how it accelerates time-to-value by delivering it in as few as 28 days. EHC is built with the solutions offered by popular hypervisors and shipped in packages of 500, 5,000, and 10,000 VMs. VMware hypervisor is supported already; while the Microsoft- and Open Stack-based solutions are targeted for early 2015. All three hypervisor solutions consist of EMC flagship storage virtualization product ViPR. ViPR software-defined storage provides a multi-vendor management layer with cloud and big data capabilities.

At EMC World 2014, EHC was configured from scratch within 48 hours! EMC recommends using their converged-infrastructure Vblock for hybrid cloud. However, it supports EHC on other infrastructures as well. This includes the customer's servers and other network equipment and storage arrays from EMC and its partner providers.

A detailed description of EHC is beyond the scope of this article. However, readers are encouraged to study its reference architecture in depth here.

Workload partitioning
Hybrid clouds are important from a cost effectiveness point of view. There are two parts to leveraging hybrid clouds. The first is about how to allow seamless workload movements between public and private clouds. The second is about what workloads should be stationed where.

There are different ways to enable workload movement from a private on premise cloud to public clouds of choice. This includes making the internet protocol (IP) and media access control (MAC) addresses of the public cloud appear as if they are a part of the internal network. Virtual extensible local area network (VXLAN), developed by Cisco provides the ability to overlay a Layer 2 (L2) network over a Layer 3 (L3) network. An overlay network is a virtual network built over L2 and L3. This eliminates the need to update IPs and MACs to gain VM mobility across clouds. VXLAN can extend enterprise L2 networks to public cloud provider networks and make them appear as a single seamless network.
Another way is to create a virtual private network (VPN) tunnel between private and public networks.

This capability allows the services to run without any change when moved to public clouds. Most cloud providers have direct connect features to allow dedicated links from hosted data centers that are in close proximity to public clouds. The ‘intelligence’ to move workloads dynamically between a private and public cloud goes by the term ‘cloud bursting’.

What is the right way to partition the architectural components between the private and public cloud? If we were lazy, we could toss a coin to decide which service runs where. Didn’t some wise man say – “laziness is happiness”? Agreed, but laziness here can be misconstrued as lack of knowledge, so we need to do better.

One interesting way to divide workload between on-premise private cloud and off-premise public cloud is to consider some kind of sandboxing. The idea is to determine based on some criteria whether or not on-demand public cloud resources need to be spawned.

Consider an example of media sharing applications where the size of media can differ vastly. A simple rule can be set wherein messages greater than a certain size are handled by off premise service instance. This not only segregates traffic better but also insulates the service instances from fluctuations in payload size. A similar rule can be applied to payloads which may require some special algorithms known to be CPU intensive. For example, content transcoding engines which adapt content by either reducing its size or converting it into another format are excellent candidates to be offloaded to a public cloud-hosted service. Similarly, not-so-critical workloads can be offloaded. Of course, the same applies when we use public cloud as primary location and use a private cloud component to offload or sandbox traffic.

A common deployment strategy is to let private cloud handle all incoming traffic and spawn extra instances of some backend service on a public cloud as and when the need for it arises. Such use cases do not require any extra support of global load balancing between private and public clouds.

The advantage of spawning service instances on the fly offer great flexibility to handle spikes in traffic around important events such as launch of a new product, new campaign, or annual events such as Christmas or New Year. Of course, all these are possible only if the architecture scales horizontally. The ability to give up public cloud resources when the demand for them
ces requires statelessness of services and we have already devoted a section to its importance.

It is more cost effective to store data closer to service which needs it. Not only does that reduce latencies, it may also save on bandwidth costs. Sometimes, decisions need to be based on sensitivity of data or regulations around the same. In fact, protection of sensitive data is definitely going to be one of the main reasons for enterprises to retain that data within their private data centers.

In cases where the service is hosted on private cloud, making its database locally resident makes sense. But what if we suddenly need to spawn more instances of this service on public cloud or what if there are other services which use same database? In such cases, replicating entire data across two sites and keeping them in sync is not only unreliable but also costly. A good strategy would be to deploy some data on both sites. Portions of data which are required by services running on cloud can be kept there. Caching would also be a great help. Eventually, some amount of data synchronization across sites may be needed as well.
In our previous example, we considered creating more instances of a backend service which does not really require load balancing client requests between private and public clouds. There can be cases when a client-facing service needs to be dynamically spawned on public cloud due to a spike in traffic. Global load balancing capabilities provided by products such as F5 by Cisco are needed in such cases to facilitate client connection load balancing between public and private cloud services.

Many vendors provide tools to make hybrid cloud computing easier. We discussed earlier how the VMs retain the same IP and MAC addresses across clouds. VMware vCloud Connector31 is one such tool which facilitates this. It also facilitates copying templates and VMs between clouds. This helps bring existing workloads to public cloud. Built-in compression, path-optimized data transfer, and checkpoint-restart makes transferring workloads between connected clouds faster and more reliable. Using some measurable indicators, the VMware Orchestrator32 then spins additional VMs on public cloud.

A challenge with migrating workloads is in ensuring consistency in environments across secondary and primary sites. Industry hardened tools such as Chef and Puppet enable this. Cloudify33 is another tool which aids a continuous delivery pipeline by standardizing deployment, orchestration, and build processes to create smoother transitions between different sites.
The container and VM debate

Virtualization is undoubtedly one of the chief reasons behind why cloud computing is so successful. Stated in simplest terms, virtualization is the art of hosting one operating system on top of another. Hypervisor-based virtualization penetrated the enterprise deeply and although container virtualization was also invented around the same time, it did not make enough inroads. This is despite a company like Google using it for years within its own data centers.

Let us understand key differences between the two technologies to understand how this debate relates to cost effectiveness. We may also be able to understand what makes containers a natural fit for launching lambda functions which we discussed earlier.

Hypervisor-based virtualization emulates hardware with the help of operating system (OS) and then spins VMs with their individual guest OS on top of this virtual hardware. Container virtualization derives from OS-level virtualization rather than at hardware level. So, one key difference between container- and hypervisor-based virtualization is that containers share the same OS and cannot have their own OS. This has an advantage that they are leaner than hypervisor counterparts.

Hypervisor-based VMs can take full advantage due to hardware isolation in the sense that the VMs cannot interfere with each other. Containers do not have that isolation yet and hence, provide a bigger attack surface to exploit. VMs on the other hand exist as a set of file definitions and policies. They provide logical separation around server resources to set them apart from other VMs. They have a small attack surface. The host’s hypervisor does its work with just 30 or 40 commands communicated directly to the hardware.

Depending on the requirements, containers are extremely compelling technology and provide great cost benefits by reducing hardware footprint. This is so, because individual containers do not run their individual OS. A lot of user applications on cloud VM consume resources (CPU, memory, or disk) much lower than what the guest OS consumes. Further, a lot of VMs remain underutilized mostly because they are fixed-size compared to auto-scaling containers. This sub-optimal usage of VMs adds up to the costs. In contrast, containers can be spawned as and when needed and that too, very quickly.

The number of containers possible on a server can go up to hundreds. By contrast, the same server may be able to spawn VMs on the order of tens only.
Not having to run OS of their own provides another advantage that the time to start them up can be as low as one-tenth of a second. A VM usually takes up to 30 seconds to start up. Experts are still debating the effectiveness of containers as compared to VMs. Their final word of advice is still not out. The problem with expert advice is that sometimes only experts understand it. Instead of posting my own opinion, I’d follow the age old suggestion that it is better to be quiet and thought a fool than speak up and clear all doubt!

It is a good idea to make a note of architectural pieces which can be spawned on the fly, rather than being put on a dedicated VM. Plus, keeping track of resource utilization to know which services can be taken off VM and launched on containers can also yield cost benefits.

To summarize, containers are cost effective and quick to start and replicate. Containers have some restrictions which we have highlighted above. They are scoped to an application. Plus, the user has to be willing to stick to one platform to provide shared OS. Containers should be a natural fit for short-lived execution tasks because they are so quick to launch and dispose. Security will be one of the key reasons behind users relying upon VMs until containers catch up on that front.
Projecting with data

Sometimes, making projections is important to understand expenses which we might incur in addition to what we have budgeted for. We could use one of the following methods:

1. Use some interesting math and make accurate projections
2. Make some simplified assumptions, do back-of-the-envelope calculations, and come up with fairly accurate projections
3. Rely on luck and do some guess work

I was once faced with a situation that the service was unable to handle requests containing payload greater than a certain threshold if more than a particular number of such requests hit the same service instance concurrently. The service was developed using Java language and a couple of such requests hitting the same service concurrently would end up shooting its garbage collection activity and slowing down the service.

Consider having hosted this application on a hybrid cloud wherein the private cloud is primary location and public cloud is used more as a spill-over location. In such a scenario, if we have limited capacity in private cloud, we may spawn additional instances on public cloud to handle the additional flow. Estimating how often we may spawn such service instances can help us be aware of costs we may incur and with what probability.

Assume that out of millions of messages inbound to our services there are around 270 such requests per service instance in a given hour that have payloads greater than a threshold. We have to determine the chances of at least 5 such requests landing on the same service within the same minute. If they are spread out during the hour, our application will handle them fine but when they start landing pretty close within a given minute, we run into trouble.

Even in absence of data points, we must be able to perform some quick back-of-the-envelope calculations. In our case we have good data points, so we may be able to perform accurate calculations. Both of these would give us estimates better than a plain guess. Could we also make a guess?

Of course, we can make a guess. This can work but is not guaranteed. Plus, we have a word of caution from the world’s best private detective Sherlock Holmes – “I never guess. It is a shocking habit,—destructive to the logical faculty.” There is a warning of potential brain damage!

Thus, we will take his advice here and not make a guess. Can we do any better?

Novel ‘The Sign of Four’ by Sir Arthur Conan Doyle
Chapter 1, Page 93: “I never guess. It is a shocking habit — destructive to the logical faculty.”
Turns out our problem can be modeled well with Poisson distribution. Suppose we can expect an independent event to occur \( \lambda \) number of times over a specified interval, then probability of exactly ‘x’ occurrences is given as:

\[ f(x; \lambda) = \frac{\lambda^x e^{-\lambda}}{x!} \]

Here, \( e \) is Euler’s constant with value equal to 2.718281 and denominator performs mathematical factorial function on \( x \). The value of \( \lambda = 270 \) per hour translates into 270/60 per minute which gives \( \lambda = 4.5 \) and \( x \geq 5 \) because we want to find chances of getting at least 5 such messages. So, the probability of having at least 5 such messages in a given minute can be computed as given below.

\[
P((x : x \geq 5) = 1 - [P(x : x = 0) + P(x : x = 1) + P(x : x = 2) + P(x : x = 3) + P(x : x = 4)]
\]

Here, \( P \) represents the probability. We’ve used a basic property here that probability of at least 5 such occurrences can be calculated by summing up probability values below 5 and subtracting their sum from 1. Feeding the values in Poisson distribution:

\[
P(x : x = 0) = (4.5^0 \times e^{-4.5})/0! = 0.011
\]

\[
P(x : x = 1) = (4.5^1 \times e^{-4.5})/1! = 0.049
\]

\[
P(x : x = 2) = (4.5^2 \times e^{-4.5})/2! = 0.112
\]

\[
P(x : x = 3) = (4.5^3 \times e^{-4.5})/3! = 0.168
\]

\[
P(x : x = 4) = (4.5^4 \times e^{-4.5})/4! = 0.189
\]

Hence, \( P(x : x \geq 2) = 1 - (0.011 + 0.049 + 0.112 + 0.168 + 0.189) = 0.467 \).

This means there is 0.467 probability of getting 5 or more such messages in a given minute. Put another way, we have an almost 50% chance of slowing down and spawning a new service.
instance which would add to our costs if that were spawned on public cloud. Making such a projection is only possible if we keep track of data relevant to making calculations.

**Back-of-the-envelope calculations**

Even in the absence of precise data points like in our case above, we must be able to perform some back-of-the-envelope calculations using our understanding of the system. Consider a case where we want to find the latency that a service introduces in a flow. In the section on cost of a transaction we discussed how latency directly translates to cost (especially, in the parlance of utility computing).

Assume that some service (say, service A) deploys a *circuit-breaker* (which we introduced earlier) while communicating with another service, B. Further assume that service B can hold and process around 500 messages in a healthy condition. Beyond that its response time is unacceptable and causes opening of *circuit-breaker* at service A. Let's take average processing time by service B to be roughly 5 seconds per message.

Little’s Law\(^3\) states that the average number of entities in a given system is equal to product of average time spent by them in the system (internal queuing and processing time) and average rate at which they leave the system. For Little’s Law to work, it is important that entities be independent of each other. This happens to be the case here because messages arriving at service B are not interdependent.

Feeding our data into this equation gives us: \(500 = 5 \times \text{message rate (out)}\). This gives us \(\text{message rate (out)} = 100\) messages per second.

To do back-of-the-envelope calculations, some simplifying assumptions have to be made. First, let us assume that all the instances of service B are handling identical load at any point in time. Another simplifying assumption is that there is no further blocking or latency introduced by service B at its incoming interface. So, message rate (in) = message rate (out).

Now, if service A instance has around 1000 messages ready to send to service B, the last few messages have to be ready to wait up to \(1000/100 = 10\) seconds to get into service B and then a further 5 seconds to get processed. If we need to reduce this further, we need to add more instances of service B. If we have an idea of cost per service, it becomes easy to see how much that will add to our bill.
**Annotation**

Before we close this article, it may be useful to list some additional strategies to minimize cloud costs. Some of these may no longer remain relevant in the future as cloud providers change their strategies often. However, they are very useful today so let us get some understanding.

All cloud providers have a notion of instance type. Depending upon the configuration, they usually fall under large, medium, and small categories. Their pricing also varies accordingly. Most people migrate to cloud and choose a safe strategy by putting their workloads on medium instance type to begin with. It is important to keep their utilization under watch. Surveys show that many customers can move their workloads to small instance types as current trends[^38] do not show more than 15-20% utilization. Monitoring status of VMs is important from a cost effectiveness point of view. As an example, Microsoft Azure bills customer for allocated virtual cores for VMs showing status as “Stopped Allocated”. It is only when status is “Stopped Deallocated” that the customer is not charged.

We discussed the concept of ‘Reserved Instances’ earlier. The idea is to block a certain amount of compute upfront. They are usually good for high availability system requirements and more cost-effective in the long run as further rates on them are discounted.

Amazon also has a notion of ‘Spot Instances’ wherein customers can bid for Amazon’s unutilized capacity. They can offer very good cost advantages because bid price can usually be lower than pricing for on-demand instances. Spot instances are more likely to fail because they are terminated as soon as spot price falls below ours or as and when the pool is shrunk. A way around these issues is to follow one of the key rules of distributed computing discussed earlier, which is to design for failure. Secondly, it is a good idea to spread deployment across bid instances as well as on others such as reserved or on-demand. It is also a good idea to spread instances across different availability zones (AZ). Finally, ‘Spot Instances’ take much longer to spawn (order of minutes) them.

It may be important to understand how load balancers such as Google Compute Engine’s load balancer or Amazon’s Route 53 or Azure Traffic Manager affect billing. They are usually charged according to number of DNS (Domain Name System) queries processed by them. DNS load balancers distribute incoming traffic between hosted services across the same or different data centers.
It is an old optimization technique to increase time-to-live (TTL) parameter in the original DNS response by the server. Increasing this value will cause other recursive DNS servers to cache those values. When a DNS query is initiated by the user, it will be satisfied by one of those cached responses in recursive DNS servers. These queries will not hit the cloud DNS server and hence, reduce the number of requests processed by them. This in turn, reduces the cost.

In Azure Traffic Manager DNS servers, the default TTL is 300 seconds. This can be increased to lessen the number of requests hitting it. This is a very simple way of reducing cost because those DNS responses are cached by recursive DNS servers.

Increasing TTL has a downside as well. In the event of failure of a load balanced service, the users will continue to be directed to faulted service. Hence, a very large TTL should be avoided too.
Conclusion

Cost effectiveness of an application on cloud is a neglected field. Many architects do not make a cost effective strategy for their applications on cloud. Elasticity in cloud is a wonderful concept. However, limiting auto-scaling beyond a threshold should be considered and instead, graceful degradation of application should be designed into the architecture.

All cloud providers publish their pricing details and it is important for an architect to have a deep understanding of the same. It pays to have an idea of how much a transaction is costing. Taken further, an architect must develop a sense of how much each service contributes to overall costs. Choice of wise design philosophies goes a long way in building cost effective application architecture. A good architecture should save money because every penny saved adds to revenue. Overspending is revenue leak, moreso when it stems due to inefficiencies in the underlying architecture. That is mainly because someone did not think through the cost effectiveness of the architecture.

Throughout this article we have discussed design strategies which are useful in building cost effective architectures. Despite their effectiveness, some restraint must be applied in implementing them. Sometimes we go overboard in optimizing the architectures to the point of diminishing returns. Repeating Donald Knuth’s golden words – “Premature optimization is root of all evil” – hold true. Or simply stated, as the classic Lost Horizon\(^9\) by James Hilton mentioned something to the same effect\(^1\) – “Laziness in doing stupid things can be a great virtue”. Yes, architects can afford to be lazy too!

Besides software architecture strategies, it is important to understand how hybrid cloud approaches cost effectiveness from the other end of the spectrum. Using hybrid cloud requires careful planning in terms of workload division.

Any architect should be able to make some projections using simple mathematics. Where data is not immediately available, simplified assumptions should be made to perform some back-of-the-envelope calculations. Guesswork should be avoided at all costs and instead, data must be relied upon. It’s like the popular quote\(^40\) – “In God we trust; all others must bring data”.

Building distributed architectures is not easy. Migrating application to cloud is even tougher. Add to that the pressures to contain costs and we are really living on the edge. But then these are precisely the challenges that any seasoned architect should seek. The reward that follows every relentless pursuit makes it worthy of all effort.

Chapter 8, Page 155: “Laziness in doing stupid things can be a great virtue.”
There are those who have already perfected the art of cost effectiveness. The question for the rest of us then, is – are we really building cost effective cloud application architectures?
References

5. Public cloud costs over the years - [https://cloud.google.com/pricing/philosophy/](https://cloud.google.com/pricing/philosophy/)
12. The Cathedral & the Bazaar by Eric S. Raymond - O’Reilly Media, 1999
16. The Fallacy of Premature Optimization - [http://ubiquity.acm.org/article.cfm?id=1513451](http://ubiquity.acm.org/article.cfm?id=1513451)
17. Pareto’s 80/20 principle - [http://en.wikipedia.org/wiki/Pareto_principle#In_software](http://en.wikipedia.org/wiki/Pareto_principle#In_software)
22. Oracle Java Fork-Join framework -
   http://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html
24. Joe Tucci, EMC CEO on hybrid cloud -
27. EMC ViPR - www.emc.com/vipr
30. EMC Hybrid cloud reference architecture - https://community.emc.com/docs/DOC-38025
32. VMware vCenter Orchestrator -
   https://www.vmware.com/support/pubs/vco_plugins_pubs.html
38. Current utility trends in cloud instances -
39. Lost Horizon by James Hilton – Pocket Books Publications, 1933
EMC believes the information in this publication is accurate as of its publication date. The information is subject to change without notice.

THE INFORMATION IN THIS PUBLICATION IS PROVIDED “AS IS.” EMC CORPORATION MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WITH RESPECT TO THE INFORMATION IN THIS PUBLICATION, AND SPECIFICALLY DISCLAIMS IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Use, copying, and distribution of any EMC software described in this publication requires an applicable software license.