ON THE RELEVANCE OF SERVERLESS CLOUD COMPUTING TO TV SERVICE OPERATIONS

M. Rizzo

BT Group plc, UK

ABSTRACT

Recent years have seen the emergence of *Serverless* computing, a model in which engineers deploy software into the cloud, but leave it to the cloud provider to provision and manage underlying servers. This enables delivery teams to focus on achieving business objectives with fewer distractions. Through use of a fine-grain Pay-As-You-Go (PAYG) model, it also enables better matching of capacity costs to real demand.

In this paper we explain why the *Serverless* model is particularly attractive for TV services, and how it can reduce operational overheads and lower barriers to entry. We start by covering the key concepts underpinning *Serverless*, with specific reference to compute, database and media encoding services. We then show why TV is well-placed to benefit, basing this on the demand patterns that TV places on IT resources and drawing on the concept of *elasticity of capacity*. For illustrative purposes the paper features a specific Pay-Per-View (PPV) use-case.

INTRODUCTION

Cloud computing has grown rapidly over the past decade, with public cloud providers such as AWS, Azure, and GoogleCloud firmly establishing themselves as viable infrastructure platforms in multiple industries. Numerous TV service vendors and operators e.g. Izrailevsky (1), have embraced cloud with great success, enabling them to deliver highly scalable and available solutions with global reach, and to manage costs by paying only for resources they consume on a PAYG basis.

With the concept of Infrastructure-as-a-Service (IaaS) firmly established in the industry, cloud providers are increasingly promoting a *Serverless* approach which, at its simplest level, means that cloud customers do not need to be aware of the servers that underpin their services. In this paper we explain why *Serverless* is an attractive paradigm for many TV use cases, and how it has the potential to transform the industry by lowering barriers to entry and operational overheads. We start by covering the key principles and benefits of *Serverless*, with specific reference to compute, database and media services. We then explain why the demand patterns that TV services place on IT resources often make them well-placed to benefit from *Serverless*, drawing on the concept of *Elasticity of Capacity*. Following an illustration based on a hypothetical PPV scenario, we conclude with an assessment of current limitations, possible future evolution, and relevance to the industry.

SERVERLESS

Cloud providers broadly offer services at three levels of abstraction (see Figure 1).

Earlier cloud offerings focussed on providing raw compute services, complemented by a small number of managed services, such as object storage and message queueing (see Barr, J. (2) for an AWS perspective). While such offerings deliver many benefits, they require considerable effort to maintain because the cloud customer retains responsibility for server configuration and software i.e. essentially *unmanaged servers*.

Recognising this, cloud providers continued to innovate to reduce maintenance overhead.

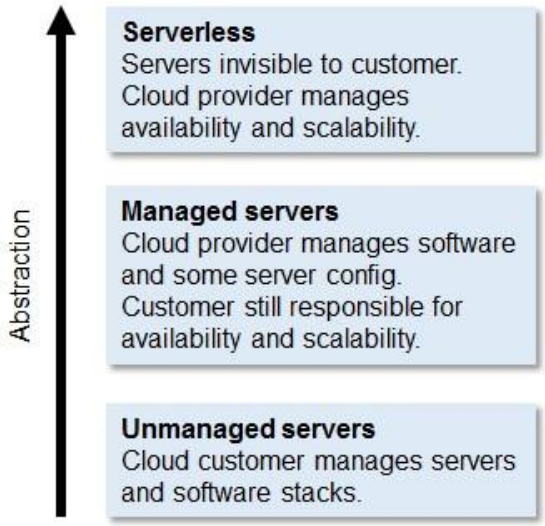

Figure 1 – Increasing levels of machine (VM)

They added and supported pre-built virtual images for popular software abstraction for cloud services stacks, and introduced services to alleviate the burden associated with managing servers for common needs such as relational databases and web applications. However these still require the cloud customer to be aware of the underlying servers, and to manage them for availability and scalability.

The next logical step for public cloud providers was to offer an even higher level of abstraction to customers wherein underlying servers are no longer visible to the cloud customer. This gave rise to the *Serverless* abstraction, in which scaling and high availability are handled by the cloud provider for a wide range of services including core services like compute, database, and storage, as well as specialist services like media processing and machine learning.

Benefits of Serverless

Eliminating the need to manage servers has many benefits: services are highly available and scalable by default, and with no additional effort. Consequently, engineering teams can spend a greater proportion of their time focussing on the business problem they need to solve, and operations teams can handle more services with less resource. AWS Lambda[1] is an example of a *serverless* compute service wherein engineers only need to supply the code for a function that is to be executed in response to a specified event. The Lambda service automatically provisions resource and distributes load across multiple AWS availability zones (AZs) to meet demand and provide high availability.

[1] The paper makes numerous references to AWS services and concepts. This is strictly for illustrative purposes, and the general principles are equally applicable to other public cloud providers. The reader is referred to the AWS White Paper (3) for an overview of AWS, as well as the AWS web-site at aws.amazon.com.

Serverless also removes the need for accurate demand forecasting, and helps to improve cost optimisation. Crucially the PAYG model is applied at a service feature level, enabling fine grain control over spend e.g. with Lambda, the customer is only billed for execution time (measured in tenths of a second) with no charge for idle time. As we shall see, this can have a very significant impact on total cost-to-serve under certain demand profiles.

THE NATURE OF DEMAND FOR IT RESOURCES IN TV

TV has numerous use cases that exhibit extreme peak-to-trough demand patterns, often with infrequent high peaks and long periods of low activity. A common example is that of a popular live event on a sport channel, where viewing can easily multiply by one or two orders of magnitude in the space of a few minutes. Another example would be the concentration of channel joins around programme boundaries, further exacerbated by automatic PVR joins which are synced to a common clock.

These patterns may not present much of an issue in the traditional broadcast world. But in the worlds of IPTV and OTT it is an entirely different story because the solutions that underpin them e.g. for content discovery and playback, rely heavily on individual interactions between end-user clients and back-end systems occurring at the time of use.

Such patterns are not limited to consumption of linear TV. VoD viewing can vary hugely by time of day, resulting in significant load variations on content discovery and playback. PPV events drive large infrequent peaks on authentication and purchasing components. And at an operational level, requests for adding new content to a VoD catalogue can often arrive in waves that coincide with the start of a new content contract, leading to big variations in capacity required for transcoding and packaging media.

It is easy to see how the combination of inflexible infrastructure management with these demand patterns leads to poor resource utilisation and a high cost-to-utilisation ratio. In the extreme, if an operator (i) dimensions their capacity to meet peak demand, (ii) provides additional redundant capacity to meet availability targets, and (iii) segregates resources for different solution components, then this results in long periods of significant unused capacity. When extrapolated to cover a few hundred components, replicated in multiple environments (test, reference, production etc) this turns into a very expensive inefficiency, which is then further compounded by inaccuracies in demand forecasting.

There are three pre-requisites that need to be met in order to eliminate such inefficiency. First it must be possible to flex the capacity associated with a particular component – which in turn means that the component must be horizontally scalable. Second it must be possible for resources freed up by one component to be re-used by another. And third it must be possible to scale in and out automatically in response to changes in demand.

Cloud computing offers a number of different approaches to address these pre-requisites. The next section introduces the concept of *Elasticity of Capacity* and defines a framework that can be used to assess the relative strengths of different approaches in this regard.

ELASTICITY OF CAPACITY

We define *elasticity of capacity* as *the extent to which paid-for*[2] *capacity matches demand*. Figure 2 illustrates this concept using three different scenarios. Scenario (a) denotes a perfectly inelastic system i.e. where the elasticity of capacity is 0, which corresponds to a fixed capacity system that often exceeds demand and may sometimes fall short. Scenario (c) depicts a perfectly elastic system i.e. elasticity of capacity is 1, wherein the capacity curve moves in perfect unison with the demand curve. Such a system would be ideal, but very difficult to achieve in practice! Scenario (b) depicts a more realistic "moderately elastic" system where capacity adapts to demand, but does so in coarse steps and timed to occur before/after an increase/decrease in resource is required.

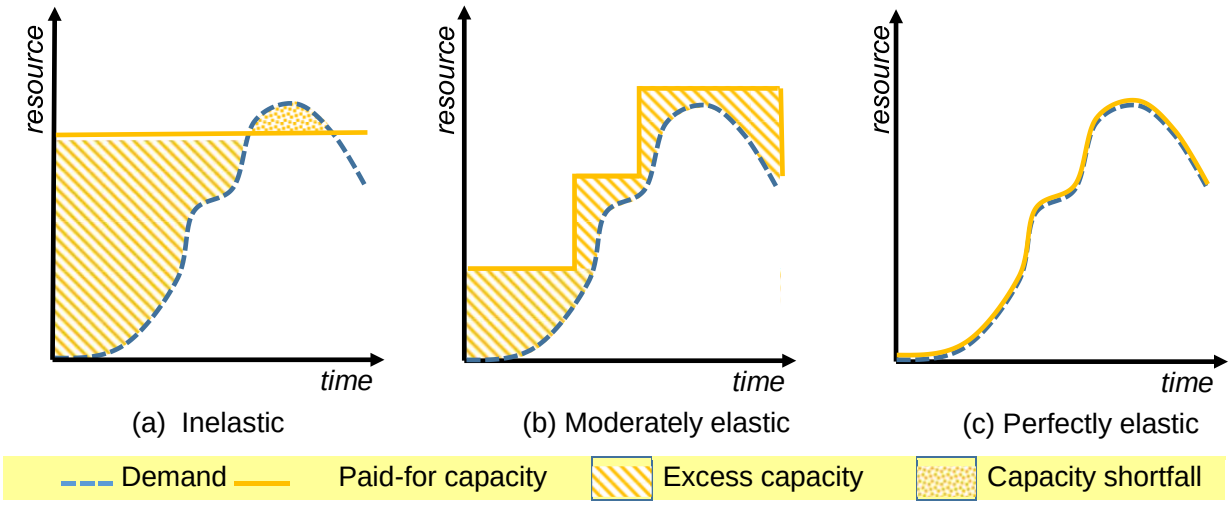

Figure 2 – Elasticity of Capacity

The extent to which paid-for capacity matches demand is determined by two factors, namely *granularity of resource adaptation* and *timeliness of resource adaptation*.

Granularity of resource adaptation refers to the smallest size of resource unit that can be added or subtracted to capacity in a single step i.e. the smallest amount by which the *y*axis value can increase or decrease at a point in time. For example it might be a VM instance, or a unit of throughput to a database.

Timeliness of resource adaptation is a measure of how early or late the resource is increased or decreased in relation to the change in demand e.g. in the case of a VM resource, time needs to be allowed for the VM to be initialised before it can be considered ready to receive requests, and time is subsequently needed to tear it down when no longer needed. Timeliness of adaptation can also be impacted by the granularity of time units in which resource is billed e.g. if a resource is paid for by the hour, then tearing that resource down after 20 minutes results in an extra 40 minutes of deviation from the demand curve.

[2] It is important to focus on "paid-for" capacity (as opposed to provisioned capacity) as ultimately this is what counts. Capacity which is de-provisioned but billed for (e.g. because billing occurs on hourly boundaries) offers no elasticity benefit to the operator.

It is important to recognise that while scalability is a pre-requisite for elasticity, they are not the same thing. Scalability is about a system's potential to be enlarged to accommodate growth in demand, but it does not mandate that this be realised in an elastic manner e.g. a scalable system might allow for the addition of more fixed capacity in advance of an anticipated increase in peak demand.

Ranking compute by elasticity

Having established this framework we can rank the elasticity of capacity for different compute options[3]. Figure 3 illustrates this using a selection of AWS compute services.

In the bottom left corner we have the inelastic fixed capacity option i.e. a fixed number of EC2 instances.

Option 2 adds auto-scaling, which enables instances to automatically be added and subtracted as required to meet demand. Here the granularity of resource adaptation is effectively determined by the instance type. The "smaller" the instance type, the more granular the adaptation, and the more elastic the system. This should not be taken to mean that smaller instance types are always preferable. As we shall see, higher elasticity does not necessarily result in the cheapest overall solution, and this very much depends on the demand profile.

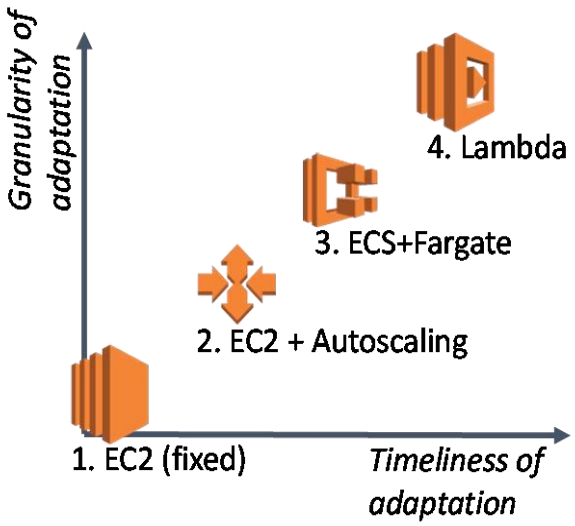

Figure 3 – Ranking of AWS Compute services

Option 3 is AWS Fargate, a service that can be used to host containerized applications. Here the granularity of resource adaptation is a *task*, with per-second billing (minimum 1 minute) applied on the basis of vCPU and memory resources allocated, making it finer grain than auto-scaling. Timeliness of adaptation is also better, as containers can be instantiated a lot more quickly than new server instances. Fargate is almost serverless in that it eliminates the need to manage cluster servers, but it still requires operators to explicitly configure scaling and placement policies for containers.

Finally Option 4 is AWS Lambda, a service wherein resource is provisioned to execute a single function, with billing applied in units of 10ths of a second on the basis of memory allocated. A key improvement here is that idle time is substantially reduced: while an EC2 instance or Fargate container might spend significant billable time waiting for a request to react to, this is not the case with Lambda.

We now consider the implications of this ranking with reference to a specific use case.

[3] While the focus for this paper is on compute services it is worth noting that the same framework can also be applied to databases, storage services, and media services e.g. in AWS we can use it to establish that S3 is more elastic than an object-store based on EBS, or that DynamoDB is more elastic than RDS.

ILLUSTRATION: PPV PURCHASE

This illustration is based around a PPV use case, wherein a TV operator is introducing a service that will make 10 live events available on a PPV basis each year. The purchase window for each event opens 30 days ahead of the event and lasts until the end of the event. It is forecast that 1 million customers will purchase a ticket to each event, with a peak purchase rate of 10 transactions per second (tps) over most of the purchase window, but with significantly higher peak demand in the hours leading up to the event estimated at 4,000 tps. It is further assumed that there is no more than one event per day, and that the service needs to be highly available at all times.

The focus for this illustration is on the compute aspects of the PPV purchase API offered to end customers. On receipt of a purchase request (via an API Gateway) this performs some checks and then creates a ledger entry (in a database table) which in turn feeds an event into an existing entitlement engine (which is outside the scope of this analysis). Figure 4 shows a simplified view of the common elements of the architecture, with the *Compute* block acting as a placeholder for the various compute options under consideration.

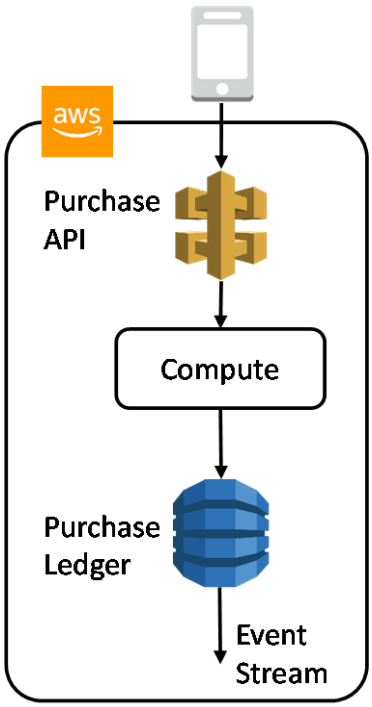

Figure 4 – PPV Purchase API

Comparison of compute options

We now consider the practical implications of "filling" the compute block using each of the AWS-based options we identified in the previous section. For each option we estimate fixed and variable cloud cost estimates (Table 1) based on AWS North Virginia pricing, and consider the relationship between cost and demand. We also compare operational overheads in each case.

In **Option 1 (EC2 fixed capacity)** we provision capacity to meet peak demand at all times, including redundant capacity to meet availability targets. If we assume that capacity will be spread across three AZs, and that we should tolerate a loss of one AZ, then each AZ should have enough capacity to take 50% of the total peak load i.e. 2,000 tps. We also allow for some margin of safety in the event that demand exceeds the forecast estimate.

Further design considerations here include selection of EC2 instance type/size to use[4], the number of instances per AZ, and the type and amount of disk storage required. These would ideally be optimised through repeated experimentation and fine tuning. For this example we assume (based on knowledge from similar use cases) that a single *t2.xlarge* with 300GB general purpose SSD will be used in each AZ. Finally an Elastic Load Balancer would be required to distribute load across the three AZs.

[4] AWS provides a range of instance families optimised for different use cases.

In addition to cloud charges, there are server maintenance overheads e.g. applying patches to OS and software, restarting hosts that have been degraded etc. It is also necessary to monitor utilisation over time, and to regularly re-assess the instance type and configuration; in the event that demand exceeds capacity then excess requests will not be serviced, whilst conversely excess capacity is still chargeable.

	EC2 fixed	Autoscale	Fargate	Lambda
Fixed	$6,200	$2,550	$1,330	$0
Variable	Negligible	< $50	$370	$35

Table 1 – PPV Cost Estimate Comparison

With **Option 2 (EC2 + autoscaling)** we still need to allow for loss of an AZ, but do not need to provision full capacity up front. Rather we start with a minimal configuration to handle off-peak load and then use an auto-scaling policy (based on a suitable utilisation metric) to add instances as demand increases. Moreover the upper capacity limit can be set higher than forecast demand to accommodate unforeseen excess demand.

Again design choices need to be made in relation to instance type and auto-scaling policy configuration. One immediate problem is that, for most AWS instance families, the smallest instance type is actually quite large! The only family with smaller instance types is the “t2” burstable family. These need time to accumulate CPU credits before they can burst to support higher loads (or can be set up to burst-on-demand, but with higher charges) – which is not a good fit with auto-scaling adding fresh instances!

One option here is to create a mix of instance types in two separate auto-scaling groups under a common load balancer. For this exercise we assume a setup in which there is always one *t2.medium* per AZ to cater for off-peak load, and then add *m5.large* instances (in groups of three to maintain HA) as demand ramps up. Similar operational overheads as for option 1 apply.

Perhaps surprisingly the variable cost element here is still quite small. The reason for this is that scaling out is only required for a few hours per event, with the system running at baseline configuration for more than 99% of the time overall. The main benefit that autoscaling brings to this case is simply a reduction of the baseline fixed cost[5].

With **Option 3 (Fargate)** we are no longer concerned with instance types, and only need to specify how much vCPU and memory to allocate to the task that will host our container. We also need to configure task placement and scaling policies to scale in/out the number of containers across multiple AZs as demand varies. The granularity of resource adaptation is much finer than for EC2, and timeliness of adaptation is better. This enables us to use an even lower baseline configuration, resulting in a more significant variable element. Operational overhead is lower because there is no need to patch servers or replace AMIs.

Finally, with **Option 4 (Lambda)** we simply deploy the function code for our Purchase API, specifying a memory size as the only parameter. Lambda automatically takes care of spreading load across multiple AZs, and scaling underlying capacity as required. Based on

[5] It should be emphasised that this result is specific to the PPV use case under consideration. The variable cost element of auto-scaling could be more significant in other use cases.

experimentation we have ascertained that, with a 1GB memory allocation, the function would execute in under 200ms in the vast majority of cases. At the time of writing, Lambda is priced at $0.20 per million calls plus $0.00001667 per GB-second, giving rise to a total cost estimate of just $35 which is entirely linearly proportional to demand.

In this example we assumed that the use of three AZs within a single region provided sufficient redundancy to deliver the required availability. If required, even higher availability can be achieved using multiple regions. Whilst this adds complexity from a data consistency perspective, it is worth noting that from a compute perspective Lambda would add minimal cost, because all of the cost is variable and linked to actual execution time. In contrast server-based solution costs would double for an active-active setup.

CONCLUSION

We have seen how *Serverless* reduces cloud costs in extreme peak-to-trough demand cases, eliminates overheads associated with managing servers, and makes it easier to maintain high scalability and availability. We have also highlighted some TV application areas that can benefit, and illustrated the potential of *Serverless* using a PPV case.

Given the benefits, it is tempting to argue for use of *Serverless* in any situation. However, high elasticity does not always translate into lowest cost. The PPV case, while reflective of a real situation, was deliberately chosen to demonstrate the effectiveness of *Serverless* with extreme peak-trough demand profiles. But applications that consistently place an even load on resources (with little or no idle time) are probably better off with less elastic services because the unit price per resource/time is typically lower. Rather than adopt a *Serverless*-first approach, it is therefore better to consider each use case on its own merits. A simple cost model will often suffice to make this judgment call.

Another consideration is that *Serverless* solutions are not infinitely scalable. There may be limits on how far they can scale out, or the rate at which they can scale out. In this respect it would be useful if cloud providers enabled operators to reserve capacity in advance of an anticipated spike. An interesting (if complex) alternative might be hybrid deployment, wherein a solution could switch between *Serverless* functions and containers to handle significant shifts in demand profile. This could help to overcome any *Serverless* capacity limits whilst achieving better cost optimisation for sustained peak loads.

A potential limitation for some cases is that *Serverless* implementation constraints can reduce flexibility for engineers e.g. Lambda only supports specific versions of a number of programming languages, and limits the time available for the function to execute.

For new operators, *Serverless* helps to lower barriers to entry by eliminating the need for upfront capital, and linking cost to actual service consumption. It also enables engineers to focus more time on core business problems. We have focussed primarily on compute in this paper, but the opportunities are much broader e.g. *Serverless* transcoding is billed on length of jobs as opposed to servers, databases are billed on storage and throughput etc. See Sbarski (4) for an example of how a start-up used *Serverless* to build a video-based learning business.

For existing operators, *Serverless* can lower costs and overheads significantly, but only if existing services are re-architected to benefit. A simple lift-and-shift approach for migration

to the cloud will not suffice. This means that deployment design needs to be factored in early, and that cloud skills are needed across all functions.

REFERENCES

1. Izrailevsky, Y., 2016. Completing the Netflix Cloud Migration, Netflix Media Center. https://media.netflix.com/en/company-blog/completing-the-netflix-cloud-migration
2. Barr, J., 2009. The AWS Blog: The First Five years, AWS News Blog. https://aws.amazon.com/blogs/aws/aws-blog-the-first-five-years/
3. AWS White Paper, 2017. Overview of Amazon Web Services. https://docs.aws.amazon.com/aws-technical-content/latest/aws-overview/aws-overview.pdf
4. Sbarski, P., 2017. Serverless Architectures on AWS. Manning Publications Co.