Chapter 25. Compute as a Service

Written by Onufry Wojtaszczyk

Edited by Lisa Carey

> *I don't try to understand computers. I try to understand the programs.*
>
> —Barbara Liskov

After doing the hard work of writing code, you need some hardware to run it. Thus, you go to buy or rent that hardware. This, in essence, is *Compute as a Service* (CaaS), in which "Compute" is shorthand for the computing power needed to actually run your programs.

This chapter is about how this simple concept—just give me the hardware to run my stuff[1]—maps into a system that will survive and scale as your organization evolves and grows. It is somewhat long because the topic is complex, and divided into four sections:

- "Taming the Compute Environment" covers how Google arrived at its solution for this problem and explains some of the key concepts of CaaS.
- "Writing Software for Managed Compute" shows how a managed compute solution affects how engineers write software. We believe that the "cattle, not pets"/flexible scheduling model has been fundamental to Google's success in the past 15 years and is an important tool in a software engineer's toolbox.
- "CaaS Over Time and Scale" goes deeper into a few lessons Google learned about how various choices about a compute architecture play out as the organization grows and evolves.
- Finally, "Choosing a Compute Service" is dedicated primarily to those engineers who will make a decision about what compute service to use in their organization.

Taming the Compute Environment

Google's internal Borg system[2] was a precursor for many of today's CaaS architectures (like Kubernetes or Mesos). To better understand how the particular aspects of such a service answer the needs of a growing and evolving organization, we'll trace the evolution of Borg and the efforts of Google engineers to tame the compute environment.

Automation of Toil

Imagine being a student at the university around the turn of the century. If you wanted to deploy some new, beautiful code, you'd SFTP the code onto one of the machines in the university's computer lab, SSH into the machine, compile and run the code. This is a tempting solution in its simplicity, but it runs into considerable issues over time and at scale. However, because that's roughly what many projects begin with, multiple organizations end up with processes that are somewhat streamlined evolutions of this system, at least for some tasks—the number of machines grows (so you SFTP and SSH into many of them), but the underlying technology remains. For example, in 2002, Jeff Dean, one of Google's most senior engineers, wrote the following about running an automated data-processing task as a part of the release process:

> *[Running the task] is a logistical, time-consuming nightmare. It currently requires getting a list of 50+ machines, starting up a process on each of these 50+ machines, and monitoring its progress on each of the 50+ machines. There is no support for automatically migrating the computation to another machine if one of the machines dies, and monitoring the progress of the jobs is done in an ad hoc manner [...] Furthermore, since processes can interfere with each other, there is a complicated, human-implemented "sign up" file to throttle the use of machines, which results in less-than-optimal scheduling, and increased contention for the scarce machine resources*

This was an early trigger in Google's efforts to tame the compute environment, which explains well how the naive solution becomes unmaintainable at larger scale.

Simple automations

There are simple things that an organization can do to mitigate some of the pain. The process of deploying a binary onto each of the 50+ machines and starting it there can easily be automated through a shell script, and then—if this is to be a reusable solution—through a more robust piece of code in an easier-to-maintain language that will perform the deployment in parallel (especially since the "50+" is likely to grow over time).

More interestingly, the monitoring of each machine can also be automated. Initially, the person in charge of the process would like to know (and be able to intervene) if something went wrong with one of the replicas. This means exporting some monitoring metrics (like "the process is alive" and "number of documents processed") from the process—by having it write to a shared storage, or call out to a monitoring service, where they can see anomalies at a glance. Current open source solutions in that space are, for instance, setting up a dashboard in a monitoring tool like Graphana or Prometheus.

If an anomaly is detected, the usual mitigation strategy is to SSH into the machine, kill the process (if it's still alive), and start it again. This is tedious, possibly error prone (be sure you connect to the right machine, and be sure to kill the right process), and could be automated:

- Instead of manually monitoring for failures, one can use an agent on the machine that detects anomalies (like "the process did not report it's alive for the past five minutes" or "the process did not process any documents over the past 10 minutes"), and kills the process if an anomaly is detected.
- Instead of logging in to the machine to start the process again after death, it might be enough to wrap the whole execution in a "`while true; do run && break; done`" shell script.

The cloud world equivalent is setting an autohealing policy (to kill and re-create a VM or container after it fails a health check).

These relatively simple improvements address a part of Jeff Dean's problem described earlier, but not all of it; human-implemented throttling, and moving to a new machine, require more involved solutions.

Automated scheduling

The natural next step is to automate machine assignment. This requires the first real "service" that will eventually grow into "Compute as a Service." That is, to automate scheduling, we need a central service that knows the complete list of machines available to it and can—on demand—pick a number of unoccupied machines and automatically deploy your binary to those machines. This eliminates the need for a hand-maintained "sign-up" file, instead delegating the maintenance of the list of machines to computers. This system is strongly reminiscent of earlier time-sharing architectures.

A natural extension of this idea is to combine this scheduling with reaction to machine failure. By scanning machine logs for expressions that signal bad health (e.g., mass disk read errors), we can identify machines that are broken, signal (to humans) the need to repair such machines, and avoid scheduling any work onto those machines in the meantime. Extending the elimination of toil further, automation can try some fixes first before involving a human, like rebooting the machine, with the hope that whatever was wrong goes away, or running an automated disk scan.

One last complaint from Jeff's quote is the need for a human to migrate the computation to another machine if the machine it's running on breaks. The solution here is simple: because we already have scheduling automation and the capability to detect that a machine is broken, we can simply have the scheduler allocate a new machine and restart the work on this new machine, abandoning the old one. The signal to do this might come from the machine introspection daemon or from monitoring of the individual process.

All of these improvements systematically deal with the growing scale of the organization. When the fleet was a single machine, SFTP and SSH were perfect solutions, but at the scale of hundreds or thousands of machines, automation needs to take over. The quote we started from came from a 2002 design document for the "Global WorkQueue," an early CaaS internal solution for some workloads at Google.

Containerization and Multitenancy

So far, we implicitly assumed a one-to-one mapping between machines and the programs running on them. This is highly inefficient in terms of computing resource (RAM, CPU) consumption, in many ways:

- It's very likely to have many more different types of jobs (with different resource requirements) than types of machines (with different resource availability), so many jobs will need to use the same machine type (which will need to be provisioned for the largest of them).
- Machines take a long time to deploy, whereas program resource needs grow over time. If obtaining new, larger machines takes your organization months, you need to also make them large enough to accommodate expected growth of resource needs over the time needed to provision new ones, which leads to waste, as new machines are not utilized to their full capacity.[3]
- Even when the new machines arrive, you still have the old ones (and it's likely wasteful to throw them away), and so you must manage a heterogeneous fleet that does not adapt itself to your needs.

The natural solution is to specify, for each program, its resource requirements (in terms of CPU, RAM, disk space), and then ask the scheduler to bin-pack replicas of the program onto the available pool of machines.

My neighbor's dog barks in my RAM

The aforementioned solution works perfectly if everybody plays nicely. However, if I specify in my configuration that each replica of my data-processing pipeline will consume one CPU and 200 MB of RAM, and then—due to a bug, or organic growth—it starts consuming more, the machines it gets scheduled onto will run out of resources. In the CPU case, this will cause neighboring serving jobs to experience latency blips; in the RAM case, it will either cause out-of-memory kills by the kernel or horrible latency due to disk swap.[4]

Two programs on the same computer can interact badly in other ways as well. Many programs will want their dependencies installed on a machine, in some specific version—and these might collide with the version requirements of some other program. A program might expect certain system-wide resources (think about `/tmp`) to be available for its own exclusive use. Security is an issue—a program might be handling sensitive data and needs to be sure that other programs on the same machine cannot access it.

Thus, a multitenant compute service must provide a degree of *isolation,* a guarantee of some sort that a process will be able to safely proceed without being disturbed by the other tenants of the machine.

A classical solution to isolation is the use of virtual machines (VMs). These, however, come with significant overhead[5] in terms of resource usage (they need the resources to run a full operating system inside) and startup time (again, they need to boot up a full operating system). This makes them a less-than-perfect solution for batch job containerization for which small resource footprints and short runtimes are expected. This led Google's engineers designing Borg in 2003 to look to different solutions, ending up with *containers*—a lightweight mechanism based on cgroups (contributed by Google engineers into the Linux kernel in 2007) and chroot jails, bind mounts and/or union/overlay filesystems for filesystem isolation. Open source container implementations include Docker and LMCTFY.

Over time and with the evolution of the organization, more and more potential isolation failures are discovered. To give a specific example, in 2011, engineers working on Borg discovered that the exhaustion of the process ID space (which was set by default to 32,000 PIDs) was becoming an isolation failure, and limits on the total number of processes/threads a single replica can spawn had to be introduced. We look at this example in more detail later in this chapter.

Rightsizing and autoscaling

The Borg of 2006 scheduled work based on the parameters provided by the engineer in the configuration, such as the number of replicas and the resource requirements.

Looking at the problem from a distance, the idea of asking humans to determine the resource requirement numbers is somewhat flawed: these are not numbers that humans interact with daily. And so, these configuration parameters become themselves, over time, a source of inefficiency. Engineers need to spend time determining them upon initial service launch, and as your organization accumulates more and more services, the cost to determine them scales up. Moreover, as time passes, the program evolves (likely grows), but the configuration parameters do not keep up. This ends in an outage—where it turns out that over time the new releases had resource requirements that ate into the slack left for unexpected spikes or outages, and when such a spike or outage actually occurs, the slack remaining turns out to be insufficient.

The natural solution is to automate the setting of these parameters. Unfortunately, this proves surprisingly tricky to do well. As an example,

Google has only recently reached a point at which more than half of the resource usage over the whole Borg fleet is determined by rightsizing automation. That said, even though it is only half of the usage, it is a larger fraction of configurations, which means that the majority of engineers do not need to concern themselves with the tedious and error-prone burden of sizing their containers. We view this as a successful application of the idea that "easy things should be easy, and complex things should be possible"—just because some fraction of Borg workloads is too complex to be properly managed by rightsizing doesn't mean there isn't great value in handling the easy cases.

Summary

As your organization grows and your products become more popular, you will grow in all of these axes:

- Number of different applications to be managed
- Number of copies of an application that needs to run
- The size of the largest application

To effectively manage scale, automation is needed that will enable you to address all these growth axes. You should, over time, expect the automation itself to become more involved, both to handle new types of requirements (for instance, scheduling for GPUs and TPUs is a major change in Borg that happened over the past 10 years) and increased scale. Actions that, at a smaller scale, could be manual, will need to be automated to avoid a collapse of the organization under the load.

One example—a transition that Google is still in the process of figuring out—is automating the management of our *datacenters*. Ten years ago, each datacenter was a separate entity. We manually managed them. Turning a datacenter up was an involved manual process, requiring a specialized skill set, that took weeks (from the moment when all the machines are ready) and was inherently risky. However, the growth of the number of datacenters Google manages meant that we moved toward a model in which turning up a datacenter is an automated process that does not require human intervention.

Writing Software for Managed Compute

The move from a world of hand-managed lists of machines to the automated scheduling and rightsizing made management of the fleet much easier for Google, but it also took profound changes to the way we write and think about software.

Architecting for Failure

Imagine an engineer is to process a batch of one million documents and validate their correctness. If processing a single document takes one second, the entire job would take one machine roughly 12 days—which is probably too long. So, we shard the work across 200 machines, which reduces the runtime to a much more manageable 100 minutes.

As discussed in "Automated scheduling", in the Borg world, the scheduler can unilaterally kill one of the 200 workers and move it to a different machine.[6] The "move it to a different machine" part implies that a new instance of your worker can be stamped out automatically, without the need for a human to SSH into the machine and tune some environment variables or install packages.

The move from "the engineer has to manually monitor each of the 100 tasks and attend to them if broken" to "if something goes wrong with one of the tasks, the system is architected so that the load is picked up by others, while the automated scheduler kills it and reinstantiates it on a new machine" has been described many years later through the analogy of "pets versus cattle."[7]

If your server is a pet, when it's broken, a human comes to look at it (usually in a panic), understand what went wrong, and hopefully nurse it back to health. It's difficult to replace. If your servers are cattle, you name them replica001 to replica100, and if one fails, automation will remove it and provision a new one in its place. The distinguishing characteristic of "cattle" is that it's easy to stamp out a new instance of the job in question—it doesn't require manual setup and can be done fully automatically. This allows for the self-healing property described earlier—in the case of a failure, automation can take over and replace the unhealthy job with a new, healthy one without human intervention. Note that although the original metaphor spoke of servers (VMs), the same applies to containers: if you can stamp out a new version of

the container from an image without human intervention, your automation will be able to autoheal your service when required.

If your servers are pets, your maintenance burden will grow linearly, or even superlinearly, with the size of your fleet, and that's a burden that no organization should accept lightly. On the other hand, if your servers are cattle, your system will be able to return to a stable state after a failure, and you will not need to spend your weekend nursing a pet server or container back to health.

Having your VMs or containers be cattle is not enough to guarantee that your system will behave well in the face of failure, though. With 200 machines, one of the replicas being killed by Borg is quite likely to happen, possibly more than once, and each time it extends the overall duration by 50 minutes (or however much processing time was lost). To deal with this gracefully, the architecture of the processing needs to be different: instead of statically assigning the work, we instead divide the entire set of one million documents into, say, 1,000 chunks of 1,000 documents each. Whenever a worker is finished with a particular chunk, it reports the results, and picks up another. This means that we lose at most one chunk of work on a worker failure, in the case when the worker dies after finishing the chunk, but before reporting it. This, fortunately, fits very well with the data-processing architecture that was Google's standard at that time: work isn't assigned equally to the set of workers at the start of the computation; it's dynamically assigned during the overall processing in order to account for workers that fail.

Similarly, for systems serving user traffic, you would ideally want a container being rescheduled not resulting in errors being served to your users. The Borg scheduler, when it plans to reschedule a container for maintenance reasons, signals its intent to the container to give it notice ahead of time. The container can react to this by refusing new requests while still having the time to finish the requests it has ongoing. This, in turn, requires the load-balancer system to understand the "I cannot accept new requests" response (and redirect traffic to other replicas).

To summarize: treating your containers or servers as cattle means that your service can get back to a healthy state automatically, but additional effort is needed to make sure that it can function smoothly while experiencing a moderate rate of failures.

Batch Versus Serving

The Global WorkQueue (which we described in the first section of this chapter) addressed the problem of what Google engineers call "batch jobs"—programs that are expected to complete some specific task (like data processing) and that run to completion. Canonical examples of batch jobs would be logs analysis or machine learning model learning. Batch jobs stood in contrast to "serving jobs"—programs that are expected to run indefinitely and serve incoming requests, the canonical example being the job that served actual user search queries from the prebuilt index.

These two types of jobs have (typically) different characteristics,[8] in particular:

- Batch jobs are primarily interested in throughput of processing. Serving jobs care about latency of serving a single request.
- Batch jobs are short lived (minutes, or at most hours). Serving jobs are typically long lived (by default only restarted with new releases).
- Because they're long lived, serving jobs are more likely to have longer startup times.

So far, most of our examples were about batch jobs. As we have seen, to adapt a batch job to survive failures, we need to make sure that work is spread into small chunks and assigned dynamically to workers. The canonical framework for doing this at Google was MapReduce,[9] later replaced by Flume.[10]

Serving jobs are, in many ways, more naturally suited to failure resistance than batch jobs. Their work is naturally chunked into small pieces (individual user requests) that are assigned dynamically to workers—the strategy of handling a large stream of requests through load balancing across a cluster of servers has been used since the early days of serving internet traffic.

However, there are also multiple serving applications that do not naturally fit that pattern. The canonical example would be any server that you intuitively describe as a "leader" of a particular system. Such a server will typically maintain the state of the system (in memory or on its local filesystem), and if the machine it is running on goes down, a newly created instance will typically be unable to re-create the system's state. Another example is when you have large amounts of data to serve—more than fits on one machine—and so you decide to shard the data among, for instance, 100 servers, each holding

1% of the data, and handling requests for that part of the data. This is similar to statically assigning work to batch job workers; if one of the servers goes down, you (temporarily) lose the ability to serve a part of your data. A final example is if your server is known to other parts of your system by its hostname. In that case, regardless of how your server is structured, if this specific host loses network connectivity, other parts of your system will be unable to contact it.[11]

Managing State

One common theme in the previous description focused on *state* as a source of issues when trying to treat jobs like cattle.[12] Whenever you replace one of your cattle jobs, you lose all the in-process state (as well as everything that was on local storage, if the job is moved to a different machine). This means that the in-process state should be treated as transient, whereas "real storage" needs to occur elsewhere.

The simplest way of dealing with this is extracting all storage to an external storage system. This means that anything that should survive past the scope of serving a single request (in the serving job case) or processing one chunk of data (in the batch case) needs to be stored off machine, in durable, persistent storage. If all your local state is immutable, making your application failure resistant should be relatively painless.

Unfortunately, most applications are not that simple. One natural question that might come to mind is, "How are these durable, persistent storage solutions implemented—are *they* cattle?" The answer should be "yes." Persistent state can be managed by cattle through state replication. On a different level, RAID arrays are an analogous concept; we treat disks as transient (accept the fact one of them can be gone) while still maintaining state. In the servers world, this might be realized through multiple replicas holding a single piece of data and synchronizing to make sure every piece of data is replicated a sufficient number of times (usually 3 to 5). Note that setting this up correctly is difficult (some way of consensus handling is needed to deal with writes), and so Google developed a number of specialized storage solutions[13] that were enablers for most applications adopting a model where all state is transient.

Other types of local storage that cattle can use covers "re-creatable" data that is held locally to improve serving latency. Caching is the most obvious example here: a cache is nothing more than transient local storage that holds

state in a transient location, but banks on the state not going away all the time, which allows for better performance characteristics on average. A key lesson for Google production infrastructure has been to provision the cache to meet your latency goals, but provision the core application for the total load. This has allowed us to avoid outages when the cache layer was lost because the noncached path was provisioned to handle the total load (although with higher latency). However, there is a clear trade-off here: how much to spend on the redundancy to mitigate the risk of an outage when cache capacity is lost.

In a similar vein to caching, data might be pulled in from external storage to local in the warm-up of an application, in order to improve request serving latency.

One more case of using local storage—this time in case of data that's written more than read—is batching writes. This is a common strategy for monitoring data (think, for instance, about gathering CPU utilization statistics from the fleet for the purposes of guiding the autoscaling system), but it can be used anywhere where it is acceptable for a fraction of data to perish, either because we do not need 100% data coverage (this is the monitoring case), or because the data that perishes can be re-created (this is the case of a batch job that processes data in chunks, and writes some output for each chunk). Note that in many cases, even if a particular calculation has to take a long time, it can be split into smaller time windows by periodic checkpointing of state to persistent storage.

Connecting to a Service

As mentioned earlier, if anything in the system has the name of the host on which your program runs hardcoded (or even provided as a configuration parameter at startup), your program replicas are not cattle. However, to connect to your application, another application does need to get your address from somewhere. Where?

The answer is to have an extra layer of indirection; that is, other applications refer to your application by some identifier that is durable across restarts of the specific "backend" instances. That identifier can be resolved by another system that the scheduler writes to when it places your application on a particular machine. Now, to avoid distributed storage lookups on the critical path of making a request to your application, clients will likely look up the address that your app can be found on, and set up a connection, at startup

time, and monitor it in the background. This is generally called *service discovery*, and many compute offerings have built-in or modular solutions. Most such solutions also include some form of load balancing, which reduces coupling to specific backends even more.

A repercussion of this model is that you will likely need to repeat your requests in some cases, because the server you are talking to might be taken down before it manages to answer.[14] Retrying requests is standard practice for network communication (e.g., mobile app to a server) because of network issues, but it might be less intuitive for things like a server communicating with its database. This makes it important to design the API of your servers in a way that handles such failures gracefully. For mutating requests, dealing with repeated requests is tricky. The property you want to guarantee is some variant of *idempotency*—that the result of issuing a request twice is the same as issuing it once. One useful tool to help with idempotency is client-assigned identifiers: if you are creating something (e.g., an order to deliver a pizza to a specific address), the order is assigned some identifier by the client; and if an order with that identifier was already recorded, the server assumes it's a repeated request and reports success (it might also validate that the parameters of the order match).

One more surprising thing that we saw happen is that sometimes the scheduler loses contact with a particular machine due to some network problem. It then decides that all of the work there is lost and reschedules it onto other machines—and then the machine comes back! Now we have two programs on two different machines, both thinking they are "replica072." The way for them to disambiguate is to check which one of them is referred to by the address resolution system (and the other one should terminate itself or be terminated); but it also is one more case for idempotency: two replicas performing the same work and serving the same role are another potential source of request duplication.

One-Off Code

Most of the previous discussion focused on production-quality jobs, either those serving user traffic, or data-processing pipelines producing production data. However, the life of a software engineer also involves running one-off analyses, exploratory prototypes, custom data-processing pipelines, and more. These need compute resources.

Often, the engineer's workstation is a satisfactory solution to the need for compute resources. If one wants to, say, automate the skimming through the 1 GB of logs that a service produced over the last day to check whether a suspicious line A always occurs before the error line B, they can just download the logs, write a short Python script, and let it run for a minute or two.

But if they want to automate the skimming through 1 TB of logs that service produced over the last year (for a similar purpose), waiting for roughly a day for the results to come in is likely not acceptable. A compute service that allows the engineer to just run the analysis on a distributed environment in several minutes (utilizing a few hundred cores) means the difference between having the analysis now and having it tomorrow. For tasks that require iteration—for example, if I will need to refine the query after seeing the results—the difference may be between having it done in a day and not having it done at all.

One concern that arises at times with this approach is that allowing engineers to just run one-off jobs on the distributed environment risks them wasting resources. This is, of course, a trade-off, but one that should be made consciously. It's very unlikely that the cost of processing that the engineer runs is going to be more expensive than the engineer's time spent on writing the processing code. The exact trade-off values differ depending on an organization's compute environment and how much it pays its engineers, but it's unlikely that a thousand core hours costs anything close to a day of engineering work. Compute resources, in that respect, are similar to markers, which we discussed in the opening of the book; there is a small savings opportunity for the company in instituting a process to acquire more compute resources, but this process is likely to cost much more in lost engineering opportunity and time than it saves.

That said, compute resources differ from markers in that it's easy to take way too many by accident. Although it's unlikely someone will carry off a thousand markers, it's totally possible someone will accidentally write a program that occupies a thousand machines without noticing.[15] The natural solution to this is instituting quotas for resource usage by individual engineers. An alternative used by Google is to observe that because we're running low-priority batch workloads effectively for free (see the section on multitenancy later on), we can provide engineers with almost unlimited quota for low-priority batch, which is good enough for most one-off engineering tasks.

CaaS Over Time and Scale

We talked above how CaaS evolved at Google and the basic parts needed to make it happen—how the simple mission of "just give me resources to run my stuff" translates to an actual architecture like Borg. Several aspects of how a CaaS architecture affects the life of software across time and scale deserve a closer look.

Containers as an Abstraction

Containers, as we described them earlier, were shown primarily as an isolation mechanism, a way to enable multitenancy, while minimizing the interference between different tasks sharing a single machine. That was the initial motivation, at least in Google. But containers turned out to also serve a very important role in abstracting away the compute environment.

A container provides an abstraction boundary between the deployed software and the actual machine it's running on. This means that as—over time—the machine changes, it is only the container software (presumably managed by a single team) that has to be adapted, whereas the application software (managed by each individual team, as the organization grows) can remain unchanged.

Let's discuss two examples of how a containerized abstraction allows an organization to manage change.

A *filesystem abstraction* provides a way to incorporate software that was not written in the company without the need to manage custom machine configurations. This might be open source software an organization runs in its datacenter, or acquisitions that it wants to onboard onto its CaaS. Without a filesystem abstraction, onboarding a binary that expects a different filesystem layout (e.g., expecting a helper binary at */bin/foo/bar*) would require either modifying the base layout of all machines in the fleet, or fragmenting the fleet, or modifying the software (which might be difficult, or even impossible due to licence considerations).

Even though these solutions might be feasible if importing an external piece of software is something that happens once in a lifetime, it is not a sustainable

solution if importing software becomes a common (or even only-somewhat-rare) practice.

A filesystem abstraction of some sort also helps with dependency management because it allows the software to predeclare and prepackage the dependencies (e.g., specific versions of libraries) that the software needs to run. Depending on the software installed on the machine presents a leaky abstraction that forces everybody to use the same version of precompiled libraries and makes upgrading any component very difficult, if not impossible.

A container also provides a simple way to manage *named resources* on the machine. The canonical example is network ports; other named resources include specialized targets; for example, GPUs and other accelerators.

Google initially did not include network ports as a part of the container abstraction, and so binaries had to search for unused ports themselves. As a result, the `PickUnusedPortOrDie` function has more than 20,000 usages in the Google C++ codebase. Docker, which was built after Linux namespaces were introduced, uses namespaces to provide containers with a virtual-private NIC, which means that applications can listen on any port they want. The Docker networking stack then maps a port on the machine to the in-container port. Kubernetes, which was originally built on top of Docker, goes one step further and requires the network implementation to treat containers ("pods" in Kubernetes parlance) as "real" IP addresses, available from the host network. Now every app can listen on any port they want without fear of conflicts.

These improvements are particularly important when dealing with software not designed to run on the particular compute stack. Although many popular open source programs have configuration parameters for which port to use, there is no consistency between them for how to configure this.

Containers and implicit dependencies

As with any abstraction, Hyrum's Law of implicit dependencies applies to the container abstraction. It probably applies *even more than usual*, both because of the huge number of users (at Google, all production software and much else will run on Borg) and because the users do not feel that they are using an API when using things like the filesystem (and are even less likely to think whether this API is stable, versioned, etc.).

To illustrate, let's return to the example of process ID space exhaustion that Borg experienced in 2011. You might wonder why the process IDs are exhaustible. Are they not simply integer IDs that can be assigned from the 32-bit or 64-bit space? In Linux, they are in practice assigned in the range [0,..., PID_MAX - 1], where PID_MAX defaults to 32,000. PID_MAX, however, can be raised through a simple configuration change (to a considerably higher limit). Problem solved?

Well, no. By Hyrum's Law, the fact that the PIDs that processes running on Borg got were limited to the 0...32,000 range became an implicit API guarantee that people started depending on; for instance, log storage processes depended on the fact that the PID can be stored in five digits, and broke for six-digit PIDs, because record names exceeded the maximum allowed length. Dealing with the problem became a lengthy, two-phase project. First, a temporary upper bound on the number of PIDs a single container can use (so that a single thread-leaking job cannot render the whole machine unusable). Second, splitting the PID space for threads and processes. (Because it turned out very few users depended on the 32,000 guarantee for the PIDs assigned to threads, as opposed to processes. So, we could increase the limit for threads and keep it at 32,000 for processes.) Phase three would be to introduce PID namespaces to Borg, giving each container its own complete PID space. Predictably (Hyrum's Law again), a multitude of systems ended up assuming that the triple {hostname, timestamp, pid} uniquely identifies a process, which would break if PID namespaces were introduced. The effort to identify all these places and fix them (and backport any relevant data) is still ongoing eight years later.

The point here is not that you should run your containers in PID namespaces. Although it's a good idea, it's not the interesting lesson here. When Borg's containers were built, PID namespaces did not exist; and even if they did, it's unreasonable to expect engineers designing Borg in 2003 to recognize the value of introducing them. Even now there are certainly resources on a machine that are not sufficiently isolated, which will probably cause problems one day. This underlines the challenges of designing a container system that will prove maintainable over time and thus the value of using a container system developed and used by a broader community, where these types of issues have already occurred for others and the lessons learned have been incorporated.

One Service to Rule Them All

As discussed earlier, the original WorkQueue design was targeted at only some batch jobs, which ended up all sharing a pool of machines managed by the WorkQueue, and a different architecture was used for serving jobs, with each particular serving job running in its own, dedicated pool of machines. The open source equivalent would be running a separate Kubernetes cluster for each type of workload (plus one pool for all the batch jobs).

In 2003, the Borg project was started, aiming (and eventually succeeding at) building a compute service that assimilates these disparate pools into one large pool. Borg's pool covered both serving and batch jobs and became the only pool in any datacenter (the equivalent would be running a single large Kubernetes cluster for all workloads in each geographical location). There are two significant efficiency gains here worth discussing.

The first one is that serving machines became cattle (the way the Borg design doc put it: "*Machines are anonymous:* programs don't care which machine they run on as long as it has the right characteristics"). If every team managing a serving job must manage their own pool of machines (their own cluster), the same organizational overhead of maintaining and administering that pool is applied to every one of these teams. As time passes, the management practices of these pools will diverge over time, making company-wide changes (like moving to a new server architecture, or switching datacenters) more and more complex. A unified management infrastructure—that is, a *common* compute service for all the workloads in the organization—allows Google to avoid this linear scaling factor; there aren't *n* different management practices for the physical machines in the fleet, there's just Borg.[16]

The second one is more subtle and might not be applicable to every organization, but it was very relevant to Google. The distinct needs of batch and serving jobs turn out to be complementary. Serving jobs usually need to be overprovisioned because they need to have capacity to serve user traffic without significant latency decreases, even in the case of a usage spike or partial infrastructure outage. This means that a machine running only serving jobs will be underutilized. It's tempting to try to take advantage of that slack by overcommitting the machine, but that defeats the purpose of the slack in the first place, because if the spike/outage does happen, the resources we need will not be available.

However, this reasoning applies only to serving jobs! If we have a number of serving jobs on a machine and these jobs are requesting RAM and CPU that sum up to the total size of the machine, no more serving jobs can be put in there, even if real utilization of resources is only 30% of capacity. But we *can* (and, in Borg, will) put batch jobs in the spare 70%, with the policy that if any of the serving jobs need the memory or CPU, we will reclaim it from the batch jobs (by freezing them in the case of CPU or killing in the case of RAM). Because the batch jobs are interested in throughput (measured in aggregate across hundreds of workers, not for individual tasks) and their individual replicas are cattle anyway, they will be more than happy to soak up this spare capacity of serving jobs.

Depending on the shape of the workloads in a given pool of machines, this means that either all of the batch workload is effectively running on free resources (because we are paying for them in the slack of serving jobs anyway) or all the serving workload is effectively paying for only what they use, not for the slack capacity they need for failure resistance (because the batch jobs are running in that slack). In Google's case, most of the time, it turns out we run batch effectively for free.

Multitenancy for serving jobs

Earlier, we discussed a number of requirements that a compute service must satisfy to be suitable for running serving jobs. As previously discussed, there are multiple advantages to having the serving jobs be managed by a common compute solution, but this also comes with challenges. One particular requirement worth repeating is a discovery service, discussed in "Connecting to a Service". There are a number of other requirements that are new when we want to extend the scope of a managed compute solution to serving tasks, for example:

- Rescheduling of jobs needs to be throttled: although it's probably acceptable to kill and restart 50% of a batch job's replicas (because it will cause only a temporary blip in processing, and what we really care about is throughput), it's unlikely to be acceptable to kill and restart 50% of a serving job's replicas (because the remaining jobs are likely too few to be able to serve user traffic while waiting for the restarted jobs to come back up again).
- A batch job can usually be killed without warning. What we lose is some of the already performed processing, which can be redone. When a serving

job is killed without warning, we likely risk some user-facing traffic returning errors or (at best) having increased latency; it is preferable to give several seconds of warning ahead of time so that the job can finish serving requests it has in flight and not accept new ones.

For the aforementioned efficiency reasons, Borg covers both batch and serving jobs, but multiple compute offerings split the two concepts—typically, a shared pool of machines for batch jobs, and dedicated, stable pools of machines for serving jobs. Regardless of whether the same compute architecture is used for both types of jobs, however, both groups benefit from being treated like cattle.

Submitted Configuration

The Borg scheduler receives the configuration of a replicated service or batch job to run in the cell as the contents of a Remote Procedure Call (RPC). It's possible for the operator of the service to manage it by using a command-line interface (CLI) that sends those RPCs, and have the parameters to the CLI stored in shared documentation, or in their head.

Depending on documentation and tribal knowledge over code submitted to a repository is rarely a good idea in general because both documentation and tribal knowledge have a tendency to deteriorate over time (see Chapter 3). However, the next natural step in the evolution—wrapping the execution of the CLI in a locally developed script—is still inferior to using a dedicated configuration language to specify the configuration of your service.

Over time, the runtime presence of a logical service will typically grow beyond a single set of replicated containers in one datacenter across many axes:

- It will spread its presence across multiple datacenters (both for user affinity and failure resistance).
- It will fork into having staging and development environments in addition to the production environment/configuration.
- It will accrue additional replicated containers of different types in the form of attached services, like a memcached accompanying the service.

Management of the service is much simplified if this complex setup can be expressed in a standardized configuration language that allows easy

expression of standard operations (like “update my service to the new version of the binary, but taking down no more than 5% of capacity at any given time”).

A standardized configuration language provides standard configuration that other teams can easily include in their service definition. As usual, we emphasize the value of such standard configuration over time and scale. If every team writes a different snippet of custom code to stand up their memcached service, it becomes very difficult to perform organization-wide tasks like swapping out to a new memcache implementation (e.g., for performance or licencing reasons) or to push a security update to all the memcache deployments. Also note that such a standardized configuration language is a requirement for automation in deployment (see Chapter 24).

Choosing a Compute Service

It’s unlikely any organization will go down the path that Google went, building its own compute architecture from scratch. These days, modern compute offerings are available both in the open source world (like Kubernetes or Mesos, or, at a different level of abstraction, OpenWhisk or Knative), or as public cloud managed offerings (again, at different levels of complexity, from things like Google Cloud Platform’s Managed Instance Groups or Amazon Web Services Elastic Compute Cloud [Amazon EC2] autoscaling; to managed containers similar to Borg, like Microsoft Azure Kubernetes Service [AKS] or Google Kubernetes Engine [GKE]; to a serverless offering like AWS Lambda or Google’s Cloud Functions).

However, most organizations will *choose* a compute service, just as Google did internally. Note that a compute infrastructure has a high lock-in factor. One reason for that is because code will be written in a way that takes advantage of all the properties of the system (Hyrum’s Law); thus, for instance, if you choose a VM-based offering, teams will tweak their particular VM images; and if you choose a specific container-based solution, teams will call out to the APIs of the cluster manager. If your architecture allows code to treat VMs (or containers) as pets, teams will do so, and then a move to a solution that depends on them being treated like cattle (or even different forms of pets) will be difficult.

To show how even the smallest details of a compute solution can end up locked in, consider how Borg runs the command that the user provided in the configuration. In most cases, the command will be the execution of a binary (possibly followed by a number of arguments). However, for convenience, the authors of Borg also included the possibility of passing in a shell script; for example, `while true; do ./my_binary; done`.[17] However, whereas a binary execution can be done through a simple fork-and-exec (which is what Borg does), the shell script needs to be run by a shell like Bash. So, Borg actually executed `/usr/bin/bash -c $USER_COMMAND`, which works in the case of a simple binary execution as well.

At some point, the Borg team realized that at Google's scale, the resources—mostly memory—consumed by this Bash wrapper are non-negligible, and decided to move over to using a more lightweight shell: ash. So, the team made a change to the process runner code to run `/usr/bin/ash -c $USER_COMMAND` instead.

You would think that this is not a risky change; after all, we control the environment, we know that both of these binaries exist, and so there should be no way this doesn't work. In reality, the way this didn't work is that the Borg engineers were not the first to notice the extra memory overhead of running Bash. Some teams were creative in their desire to limit memory usage and replaced (in their custom filesystem overlay) the Bash command with a custom-written piece of "execute the second argument" code. These teams, of course, were very aware of their memory usage, and so when the Borg team changed the process runner to use ash (which was not overwritten by the custom code), their memory usage increased (because it started including ash usage instead of the custom code usage), and this caused alerts, rolling back the change, and a certain amount of unhappiness.

Another reason that a compute service choice is difficult to change over time is that any compute service choice will eventually become surrounded by a large ecosystem of helper services—tools for logging, monitoring, debugging, alerting, visualization, on-the-fly analysis, configuration languages and meta-languages, user interfaces, and more. These tools would need to be rewritten as a part of a compute service change, and even understanding and enumerating those tools is likely to be a challenge for a medium or large organization.

Thus, the choice of a compute architecture is important. As with most software engineering choices, this one involves trade-offs. Let's discuss a few.

Centralization Versus Customization

From the point of view of management overhead of the compute stack (and also from the point of view of resource efficiency), the best an organization can do is adopt a single CaaS solution to manage its entire fleet of machines and use only the tools available there for everybody. This ensures that as the organization grows, the cost of managing the fleet remains manageable. This path is basically what Google has done with Borg.

Need for customization

However, a growing organization will have increasingly diverse needs. For instance, when Google launched the Google Compute Engine (the "VM as a Service" public cloud offering) in 2012, the VMs, just as most everything else at Google, were managed by Borg. This means that each VM was running in a separate container controlled by Borg. However, the "cattle" approach to task management did not suit Cloud's workloads, because each particular container was actually a VM that some particular user was running, and Cloud's users did not, typically, treat the VMs as cattle.[18]

Reconciling this difference required considerable work on both sides. The Cloud organization made sure to support live migration of VMs; that is, the ability to take a VM running on one machine, spin up a copy of that VM on another machine, bring the copy to be a perfect image, and finally redirect all traffic to the copy, without causing a noticeable period when service is unavailable.[19] Borg, on the other hand, had to be adapted to avoid at-will killing of containers containing VMs (to provide the time to migrate the VM's contents to the new machine), and also, given that the whole migration process is more expensive, Borg's scheduling algorithms were adapted to optimize for decreasing the risk of rescheduling being needed.[20] Of course, these modifications were rolled out only for the machines running the cloud workloads, leading to a (small, but still noticeable) bifurcation of Google's internal compute offering.

A different example—but one that also leads to a bifurcation—comes from Search. Around 2011, one of the replicated containers serving Google Search web traffic had a giant index built up on local disks, storing the less-often-

accessed part of the Google index of the web (the more common queries were served by in-memory caches from other containers). Building up this index on a particular machine required the capacity of multiple hard drives and took several hours to fill in the data. However, at the time, Borg assumed that if any of the disks that a particular container had data on had gone bad, the container will be unable to continue, and needs to be rescheduled to a different machine. This combination (along with the relatively high failure rate of spinning disks, compared to other hardware) caused severe availability problems; containers were taken down all the time and then took forever to start up again. To address this, Borg had to add the capability for a container to deal with disk failure by itself, opting out of Borg's default treatment; while the Search team had to adapt the process to continue operation with partial data loss.

Multiple other bifurcations, covering areas like filesystem shape, filesystem access, memory control, allocation and access, CPU/memory locality, special hardware, special scheduling constraints, and more, caused the API surface of Borg to become large and unwieldy, and the intersection of behaviors became difficult to predict, and even more difficult to test. Nobody really knew whether the expected thing happened if a container requested *both* the special Cloud treatment for eviction *and* the custom Search treatment for disk failure (and in many cases, it was not even obvious what "expected" means).

After 2012, the Borg team devoted significant time to cleaning up the API of Borg. It discovered some of the functionalities Borg offered were no longer used at all.[21] The more concerning group of functionalities were those that were used by multiple containers, but it was unclear whether intentionally—the process of copying the configuration files between projects led to proliferation of usage of features that were originally intended for power users only. Whitelisting was introduced for certain features to limit their spread and clearly mark them as poweruser–only. However, the cleanup is still ongoing, and some changes (like using labels for identifying groups of containers) are still not fully done.[22]

As usual with trade-offs, although there are ways to invest effort and get some of the benefits of customization while not suffering the worst downsides (like the aforementioned whitelisting for power functionality), in the end there are hard choices to be made. These choices usually take the form of multiple small questions: do we accept expanding the explicit (or worse, implicit) API surface to accommodate a particular user of our infrastructure, or do we significantly inconvenience that user, but maintain higher coherence?

Level of Abstraction: Serverless

The description of taming the compute environment by Google can easily be read as a tale of increasing and improving abstraction—the more advanced versions of Borg took care of more management responsibilities and isolated the container more from the underlying environment. It's easy to get the impression this is a simple story: more abstraction is good; less abstraction is bad.

Of course, it is not that simple. The landscape here is complex, with multiple offerings. In "Taming the Compute Environment", we discussed the progression from dealing with pets running on bare-metal machines (either owned by your organization or rented from a colocation center) to managing containers as cattle. In between, as an alternative path, are VM-based offerings in which VMs can progress from being a more flexible substitute for bare metal (in Infrastructure as a Service offerings like Google Compute Engine [GCE] or Amazon EC2) to heavier substitutes for containers (with autoscaling, rightsizing, and other management tools).

In Google's experience, the choice of managing cattle (and not pets) is the solution to managing at scale. To reiterate, if each of your teams will need just one pet machine in each of your datacenters, your management costs will rise superlinearly with your organization's growth (because both the number of teams *and* the number of datacenters a team occupies are likely to grow). And after the choice to manage cattle is made, containers are a natural choice for management; they are lighter weight (implying smaller resource overheads and startup times) and configurable enough that should you need to provide specialized hardware access to a specific type of workload, you can (if you so choose) allow punching a hole through easily.

The advantage of VMs as cattle lies primarily in the ability to bring our own operating system, which matters if your workloads require a diverse set of operating systems to run. Multiple organizations will also have preexisting experience in managing VMs, and preexisting configurations and workloads based on VMs, and so might choose to use VMs instead of containers to ease migration costs.

What is serverless?

An even higher level of abstraction is *serverless* offerings.[23] Assume that an organization is serving web content and is using (or willing to adopt) a common server framework for handling the HTTP requests and serving responses. The key defining trait of a framework is the inversion of control—so, the user will only be responsible for writing an "Action" or "Handler" of some sort—a function in the chosen language that takes the request parameters and returns the response.

In the Borg world, the way you run this code is that you stand up a replicated container, each replica containing a server consisting of framework code and your functions. If traffic increases, you will handle this by scaling up (adding replicas or expanding into new datacenters). If traffic decreases, you will scale down. Note that a minimal presence (Google usually assumes at least three replicas in each datacenter a server is running in) is required.

However, if multiple different teams are using the same framework, a different approach is possible: instead of just making the machines multitenant, we can also make the framework servers themselves multitenant. In this approach, we end up running a larger number of framework servers, dynamically load/unload the action code on different servers as needed, and dynamically direct requests to those servers that have the relevant action code loaded. Individual teams no longer run servers, hence "serverless."

Most discussions of serverless frameworks compare them to the "VMs as pets" model. In this context, the serverless concept is a true revolution, as it brings in all of the benefits of cattle management—autoscaling, lower overhead, lack of explicit provisioning of servers. However, as described earlier, the move to a shared, multitenant, cattle-based model should already be a goal for an organization planning to scale; and so the natural comparison point for serverless architectures should be "persistent containers" architecture like Borg, Kubernetes, or Mesosphere.

Pros and cons

First note that a serverless architecture requires your code to be *truly stateless*; it's unlikely we will be able to run your users' VMs or implement Spanner inside the serverless architecture. All the ways of managing local state (except not using it) that we talked about earlier do not apply. In the containerized

world, you might spend a few seconds or minutes at startup setting up connections to other services, populating caches from cold storage, and so on, and you expect that in the typical case you will be given a grace period before termination. In a serverless model, there is no local state that is really persisted across requests; everything that you want to use, you should set up in request-scope.

In practice, most organizations have needs that cannot be served by truly stateless workloads. This can either lead to depending on specific solutions (either home grown or third party) for specific problems (like a managed database solution, which is a frequent companion to a public cloud serverless offering) or to having two solutions: a container-based one and a serverless one. It's worth mentioning that many or most serverless frameworks are built on top of other compute layers: AppEngine runs on Borg, Knative runs on Kubernetes, Lambda runs on Amazon EC2.

The managed serverless model is attractive for *adaptable scaling* of the resource cost, especially at the low-traffic end. In, say, Kubernetes, your replicated container cannot scale down to zero containers (because the assumption is that spinning up both a container and a node is too slow to be done at request serving time). This means that there is a minimum cost of just having an application available in the persistent cluster model. On the other hand, a serverless application can easily scale down to zero; and so the cost of just owning it scales with the traffic.

At the very high-traffic end, you will necessarily be limited by the underlying infrastructure, regardless of the compute solution. If your application needs to use 100,000 cores to serve its traffic, there needs to be 100,000 physical cores available in whatever physical equipment is backing the infrastructure you are using. At the somewhat lower end, where your application does have enough traffic to keep multiple servers busy but not enough to present problems to the infrastructure provider, both the persistent container solution and the serverless solution can scale to handle it, although the scaling of the serverless solution will be more reactive and more granular than that of the persistent container one.

Finally, adopting a serverless solution implies a certain loss of control over your environment. On some level, this is a good thing: having control means having to exercise it, and that means management overhead. But, of course,

this also means that if you need some extra functionality that's not available in the framework you use, it will become a problem for you.

To take one specific instance of that, the Google Code Jam team (running a programming contest for thousands of participants, with a frontend running on Google AppEngine) had a custom-made script to hit the contest webpage with an artificial traffic spike several minutes before the contest start, in order to warm up enough instances of the app to serve the actual traffic that happened when the contest started. This worked, but it's the sort of hand-tweaking (and also hacking) that one would hope to get away from by choosing a serverless solution.

The trade-off

Google's choice in this trade-off was not to invest heavily into serverless solutions. Google's persistent containers solution, Borg, is advanced enough to offer most of the serverless benefits (like autoscaling, various frameworks for different types of applications, deployment tools, unified logging and monitoring tools, and more). The one thing missing is the more aggressive scaling (in particular, the ability to scale down to zero), but the vast majority of Google's resource footprint comes from high-traffic services, and so it's comparably cheap to overprovision the small services. At the same time, Google runs multiple applications that would not work in the "truly stateless" world, from GCE, through home-grown database systems like BigQuery or Spanner, to servers that take a long time to populate the cache, like the aforementioned long-tail search serving jobs. Thus, the benefits of having one common unified architecture for all of these things outweigh the potential gains for having a separate serverless stack for a part of a part of the workloads.

However, Google's choice is not necessarily the correct choice for every organization: other organizations have successfully built out on mixed container/serverless architectures, or on purely serverless architectures utilizing third-party solutions for storage.

The main pull of serverless, however, comes not in the case of a large organization making the choice, but in the case of a smaller organization or team; in that case, the comparison is inherently unfair. The serverless model, though being more restrictive, allows the infrastructure vendor to pick up a much larger share of the overall management overhead and thus *decrease the*

management overhead for the users. Running the code of one team on a shared serverless architecture, like AWS Lambda or Google's Cloud Run, is significantly simpler (and cheaper) than setting up a cluster to run the code on a managed container service like GKE or AKS if the cluster is not being shared among many teams. If your team wants to reap the benefits of a managed compute offering but your larger organization is unwilling or unable to move to a persistent containers-based solution, a serverless offering by one of the public cloud providers is likely to be attractive to you because the cost (in resources and management) of a shared cluster amortizes well only if the cluster is truly shared (between multiple teams in the organization).

Note, however, that as your organization grows and adoption of managed technologies spreads, you are likely to outgrow the constraints of a purely serverless solution. This makes solutions where a break-out path exists (like from KNative to Kubernetes) attractive given that they provide a natural path to a unified compute architecture like Google's, should your organization decide to go down that path.

Public Versus Private

Back when Google was starting, the CaaS offerings were primarily homegrown; if you wanted one, you built it. Your only choice in the public-versus-private space was between owning the machines and renting them, but all the management of your fleet was up to you.

In the age of public cloud, there are cheaper options, but there are also more choices, and an organization will have to make them.

An organization using a public cloud is effectively outsourcing (a part of) the management overhead to a public cloud provider. For many organizations, this is an attractive proposition—they can focus on providing value in their specific area of expertise and do not need to grow significant infrastructure expertise. Although the cloud providers (of course) charge more than the bare cost of the metal to recoup the management expenses, they have the expertise already built up, and they are sharing it across multiple customers.

Additionally, a public cloud is a way to scale the infrastructure more easily. As the level of abstraction grows—from colocations, through buying VM time, up to managed containers and serverless offerings—the ease of scaling up increases—from having to sign a rental agreement for colocation space,

through the need to run a CLI to get a few more VMs, up to autoscaling tools for which your resource footprint changes automatically with the traffic you receive. Especially for young organizations or products, predicting resource requirements is challenging, and so the advantages of not having to provision resources up front are significant.

One significant concern when choosing a cloud provider is the fear of lock-in—the provider might suddenly increase their prices or maybe just fail, leaving an organization in a very difficult position. One of the first serverless offering providers, Zimki, a Platform as a Service environment for running JavaScript, shut down in 2007 with three months' notice.

A partial mitigation for this is to use public cloud solutions that run using an open source architecture (like Kubernetes). This is intended to make sure that a migration path exists, even if the particular infrastructure provider becomes unacceptable for some reason. Although this mitigates a significant part of the risk, it is not a perfect strategy. Because of Hyrum's Law, it's difficult to guarantee no parts that are specific to a given provider will be used.

Two extensions of that strategy are possible. One is to use a lower-level public cloud solution (like Amazon EC2) and run a higher-level open source solution (like OpenWhisk or KNative) on top of it. This tries to ensure that if you want to migrate out, you can take whatever tweaks you did to the higher-level solution, tooling you built on top of it, and implicit dependencies you have along with you. The other is to run multicloud; that is, to use managed services based on the same open source solutions from two or more different cloud providers (say, GKE and AKS for Kubernetes). This provides an even easier path for migration out of one of them, and also makes it more difficult to depend on specific implementation details available in one one of them.

One more related strategy—less for managing lock-in, and more for managing migration—is to run in a hybrid cloud; that is, have a part of your overall workload on your private infrastructure, and part of it run on a public cloud provider. One of the ways this can be used is to use the public cloud as a way to deal with overflow. An organization can run most of its typical workload on a private cloud, but in case of resource shortage, scale some of the workloads out to a public cloud. Again, to make this work effectively, the same open source compute infrastructure solution needs to be used in both spaces.

Both multicloud and hybrid cloud strategies require the multiple environments to be connected well, through direct network connectivity between machines in different environments and common APIs that are available in both.

Conclusion

Over the course of building, refining, and running its compute infrastructure, Google learned the value of a well-designed, common compute infrastructure. Having a single infrastructure for the entire organization (e.g., one or a small number of shared Kubernetes clusters per region) provides significant efficiency gains in management and resource costs and allows the development of shared tooling on top of that infrastructure. In the building of such an architecture, containers are a key tool to allow sharing a physical (or virtual) machine between different tasks (leading to resource efficiency) as well as to provide an abstraction layer between the application and the operating system that provides resilience over time.

Utilizing a container-based architecture well requires designing applications to use the “cattle” model: engineering your application to consist of nodes that can be easily and automatically replaced allows scaling to thousands of instances. Writing software to be compatible with that model requires different thought patterns; for example, treating all local storage (including disk) as ephemeral and avoiding hardcoding hostnames.

That said, although Google has, overall, been both satisfied and successful with its choice of architecture, other organizations will choose from a wide range of compute services—from the “pets” model of hand-managed VMs or machines, through “cattle” replicated containers, to the abstract “serverless” model, all available in managed and open source flavors; your choice is a complex trade-off of many factors.

TL;DRs

- Scale requires a common infrastructure for running workloads in production.
- A compute solution can provide a standardized, stable abstraction and environment for software.

- Software needs to be adapted to a distributed, managed compute environment.
- The compute solution for an organization should be chosen thoughtfully to provide appropriate levels of abstraction.

1 Disclaimer: for some applications, the "hardware to run it" is the hardware of your customers (think, for example, of a shrink-wrapped game you bought a decade ago). This presents very different challenges that we do not cover in this chapter.

2 Abhishek Verma, Luis Pedrosa, Madhukar R Korupolu, David Oppenheimer, Eric Tune, and John Wilkes, "Large-scale cluster management at Google with Borg," *EuroSys*, Article No.: 18 (April 2015): 1–17.

3 Note that this and the next point apply less if your organization is renting machines from a public cloud provider.

4 Google has chosen, long ago, that the latency degradation due to disk swap is so horrible that an out-of-memory kill and a migration to a different machine is universally preferable—so in Google's case, it's always an out-of-memory kill.

5 Although a considerable amount of research is going into decreasing this overhead, it will never be as low as a process running natively.

6 The scheduler does not do this arbitrarily, but for concrete reasons (like the need to update the kernel, or a disk going bad on the machine, or a reshuffle to make the overall distribution of workloads in the datacenter bin-packed better). However, the point of having a compute service is that as a software author, I should neither know nor care why regarding the reasons this might happen.

7 The "pets versus cattle" metaphor is attributed to Bill Baker by Randy Bias and it's become extremely popular as a way to describe the "replicated software unit" concept. As an analogy, it can also be used to describe concepts other than servers; for example, see Chapter 22.

8 Like all categorizations, this one isn't perfect; there are types of programs that don't fit neatly into any of the categories, or that possess characteristics typical of both serving and batch jobs. However, like most useful categorizations, it still captures a distinction present in many real-life cases.

9 See Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," 6th Symposium on Operating System Design and Implementation (OSDI), 2004.

10 Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert Henry, Robert Bradshaw, and Nathan Weizenbaum, “Flume-Java: Easy, Efficient Data-Parallel Pipelines,” ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2010.

11 See also Atul Adya et al. “Auto-sharding for datacenter applications,” OSDI, 2019; and Atul Adya, Daniel Myers, Henry Qin, and Robert Grandl, “Fast key-value stores: An idea whose time has come and gone,” HotOS XVII, 2019.

12 Note that, besides distributed state, there are other requirements to setting up an effective “servers as cattle” solution, like discovery and load-balancing systems (so that your application, which moves around the datacenter, can be accessed effectively). Because this book is less about building a full CaaS infrastructure and more about how such an infrastructure relates to the art of software engineering, we won’t go into more detail here.

13 See, for example, Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, “The Google File System,” Proceedings of the 19th ACM Symposium on Operating Systems, 2003; Fay Chang et al., “Bigtable: A Distributed Storage System for Structured Data,” 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI); or James C. Corbett et al., “Spanner: Google’s Globally Distributed Database,” OSDI, 2012.

14 Note that retries need to be implemented correctly—with backoff, graceful degradation and tools to avoid cascading failures like jitter. Thus, this should likely be a part of Remote Procedure Call library, instead of implemented by hand by each developer. See, for example, Chapter 22: Addressing Cascading Failures in the SRE book.

15 This has happened multiple times at Google; for instance, because of someone leaving load-testing infrastructure occupying a thousand Google Compute Engine VMs running when they went on vacation, or because a new employee was debugging a master binary on their workstation without realizing it was spawning 8,000 full-machine workers in the background.

16 As in any complex system, there are exceptions. Not all machines owned by Google are Borg-managed, and not every datacenter is covered by a single Borg cell. But the majority of engineers work in an environment in which they don’t touch non-Borg machines, or nonstandard cells.

17 This particular command is actively harmful under Borg because it prevents Borg’s mechanisms for dealing with failure from kicking in. However, more complex wrappers that echo parts of the environment to logging, for example, are still in use to help debug startup problems.

18 My mail server is not interchangeable with your graphics rendering job, even if both of those tasks are running in the same form of VM.

19 This is not the only motivation for making user VMs possible to live migrate; it also offers considerable user-facing benefits because it means the host operating system can be patched and the host hardware updated without disrupting the VM. The alternative (used by other major cloud vendors) is to deliver “maintenance event notices,” which mean the VM can be, for example, rebooted or stopped and later started up by the cloud provider.

20 This is particularly relevant given that not all customer VMs are opted into live migration; for some workloads even the short period of degraded performance during the migration is unacceptable. These customers will receive maintenance event notices, and Borg will avoid evicting the containers with those VMs unless strictly necessary.

21 A good reminder that monitoring and tracking the usage of your features is valuable over time.

22 This means that Kubernetes, which benefited from the experience of cleaning up Borg but was not hampered by a broad existing userbase to begin with, was significantly more modern in quite a few aspects (like its treatment of labels) from the beginning. That said, Kubernetes suffers some of the same issues now that it has broad adoption across a variety of types of applications.

23 FaaS (Function as a Service) and PaaS (Platform as a Service) are related terms to serverless. There are differences between the three terms, but there are more similarities, and the boundaries are somewhat blurred.