SRE Best Practices for Capacity Management

LUIS QUESADA TORRES AND DOUG COLISH

Luis Quesada Torres is a Site Reliability Engineer and Manager at Google, where he is responsible for keeping Google Cloud's Artificial Intelligence products running reliably and efficiently. In his spare time, Luis jumps from hobby to hobby: he composes and produces music across multiple genres, he skateboards, and he speaks Spanish, English, German, Swiss German, and Esperanto. Soon Japanese as well. luis@google.com

Doug Colish is a Technical Writer at Google in NYC supporting Site Reliability Engineering (SRE) teams. He contributed to several chapters of Google's "Building Secure and Reliable Systems" book. Doug has over three decades of system engineering experience specializing in UNIX and security. His hobbies include detailing and modifying cars, attending concerts, and watching and discussing great movies. dcolish@google.com

As an SRE, you're responsible for determining the initial resource requirements of your service and ensuring your service behaves reasonably even in the face of unexpected demand. *Capacity management* is the process of ensuring you have the appropriate amount of resources for your service to be scalable, efficient, and reliable. User-facing and company internal services must accommodate both expected and unexpected growth. We define utilization as the percentage of a resource that is being used. It's difficult to determine initial resource utilization and predict future needs. We present ways to estimate utilization and identify blind spots, and we discuss the benefits of building in redundancy to avoid failures. You'll use this information to design your architecture such that increasing the resource allocation for each component of the service effectively increases the capacity of the entire service linearly.

Principles of Capacity Management

A *service*, in the context of this article, is defined as the set of all of the binaries (service stack) that provides a set of functions.

Successful capacity management entails allocating resources from two complex points of view: *resource provisioning*, which provides the initial capacity to run the service now, and *capacity planning*, which safeguards the reliability of the service into the future.

At its core, capacity management must follow three basic principles in order to keep a service scalable, usable, and manageable:

- **Services must use their resources efficiently**. Large services that require a considerable amount of resources are expensive to deploy and maintain.
- **Services must run reliably**. Limiting resource capacity to improve service efficiency can put the service at risk of malfunctioning and suffering user-facing outages. There is a tradeoff between service efficiency and reliability.
- **Service growth must be anticipated**. Adding resources to a service can take a long time and has real world limitations around deployment. This may involve buying and deploying new equipment or building new datacenters. It may also require increasing capacity for other software systems and infrastructure that are dependencies of the service.

Complexities of Capacity Management

A large service is a complex living organism whose behavior is unexpected at times. You need to consider several areas when making engineering decisions that could potentially alter the service's scope:

Service performance. Understand how different components of the service perform under load.

Service failure modes. Consider the known failure modes and how the service behaves when subjected to them. Also, consider how the service might behave when subjected to unknown failure modes. Be prepared by generating a list of possible bottlenecks and service dependencies you may encounter.

Demand. Determine the expected user count and traffic, where the user base is located, and the usage patterns.

Organic growth. Estimate how demand may grow over time.

Inorganic growth. Keep in mind the long-term resource impact of adding new features or of the service becoming more successful than expected.

Scaling. Understand how the service scales when increasing resource allocations.

Market analysis. Estimate how market changes affect your ability to acquire additional resources. Research new technologies that can improve the performance, reliability, or efficiency of the service and the cost of implementing them. Investigate how quickly you can adopt new technologies, such as replacing HDDs with SSDs.

The goal of capacity management is controlling uncertainty. In the midst of the unknown, the service must be available now and continue to run in the future. A challenging but rewarding and delicate balance of tradeoffs is in play: efficiency vs. reliability, accuracy vs. complexity, and effort vs. benefit.

Use data to drive capacity decisions. You'll still make unavoidable mistakes, and you'll have fires to put out, often in creative ways. But the end result is a reliable business-critical service.

Resource provisioning addresses the tactical question, "How do I keep the service running right now?" while *capacity planning* addresses the strategic question, "How do I keep the service running for the foreseeable future?"

The following sections discuss these topics in detail.

Resource Provisioning

Our discussions focus on a serving system, that is, a service that responds to user requests by looking up some data. However, you can apply these principles equally to a data storage service, data transformation service, and most other things you can do with a computer.

Resource provisioning involves figuring out the *target utilization* of resources a service needs and allocating those resources. Target utilization is defined as the highest possible utilization for a specific *resource class* that allows the service to function reliably. A resource class refers to a specific type of computing asset. CPU, RAM, storage, etc. are resource classes.

To provision resources for your service, use demand signals as inputs and create the production layout with concrete resource allocations as output, as shown in Figure 1. Services often use several resource classes.

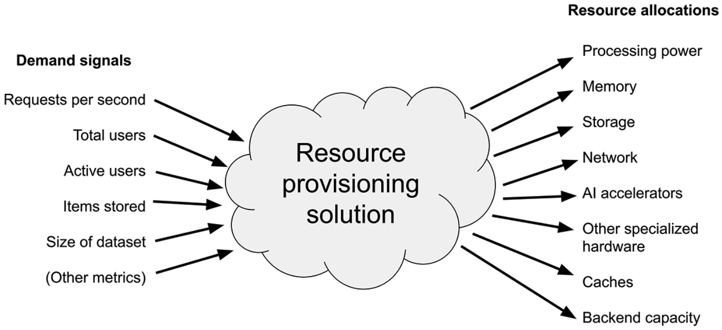


Figure 1: Demand signals and resource allocations of a resource provisioning solution

The Impact of Resource Shortages

A shortage of resources can make the service fail differently, depending on the resource class.

When resources become a bottleneck in the service's *critical path*, users experience increased latency. In a worst-case scenario, the bottleneck causes requests to backlog, resulting in ever-increasing latency and, eventually, the timeout of queued requests. Without a mitigation plan in place, the service fails to process requests and suffers an outage. The outage continues until the incoming traffic drops off, allowing the service to catch up, or until the service is restarted.

Resources that are often in the critical path include:

- Processing power
- Network
- Storage throughput

When resources become a bottleneck in the *non-critical path*, the service suffers delays in some of its non-time critical functions, such as maintenance or asynchronous processing. If these tasks are delayed long enough, they could impact service performance, features, data integrity, and even cause an outage in extreme cases.

When a service runs out of storage, writes fail. Even certain reads may fail if they are dependent on writes: for example, if the service or storage solution stores Paxos state to do consistent reads, or if the storage solution keeps track of all accessed data and the time it was accessed.

When other resources such as memory or network sockets are low, a service may crash, restart, or hang. A service with low resources may start to thrash from garbage-collection or misbehave in other ways. These failures decrease the service's capacity and can trigger cascading failure scenarios requiring human interaction to resolve.

For mitigation strategies, see the Decrease the Impact of Outages section below.

Estimating Utilization

Because of their different nature, resource usage and target utilization are different for every service and for each resource class. In order to estimate the target utilization for a specific service, each of the following aspects need to be considered.

Peak Usage

A service's peak usage is simply the highest usage rate over a given time period and depends on the nature of the service and the user base. The early hours of a business-related service may drive the weekday peaks. Social-related services peak late in the afternoon, at night, during weekends, or coinciding with social events such as concerts, sporting events, etc. When an unexpected event happens, usage can drop or soar. A global service's user base is spread across different countries and time zones, forming a more complex daily traffic pattern.

Assuming non-constant load, resource utilization shouldn't surpass 100% of the service's allocated resources during peak traffic. By not using all of its resources, the service has sufficient capacity to serve the peak and is not overprovisioned in any wasteful way.

Maximum Peak Utilization

Even at peak, it's a bad idea to run the service at 100% utilization. Some software, languages, or platforms will misbehave or garbage-collection thrash before CPU use even reaches 100%. A service will crash with an *out of memory* (OOM) error if a component reaches 100% memory utilization.

Fine-tuning your monitoring sufficiently to capture the precise resource utilization in small enough time frames (microseconds or even seconds) is tedious. Thus, it's difficult to determine the resource usage peak for low-latency applications.

Redundancy

Issues with rollouts, hardware, software, or even planned maintenance can cause the components of a service to fail or restart. This can result in a failure as small as a single binary instance crashing or as large as the whole service going offline.

Redundancy is a system design principle that includes duplicated components that are active only when they replace other components that failed. The degree of redundancy is denoted by *N+x*, where *N* is the total number of active components, and *x* is the number of backup components. Thus, *N*+3 indicates that three system components can fail because there are three duplicated components to replace them. Meanwhile, the service remains completely functional, regardless of the total number of components (*N*).

Redundancy can be applied within regions or across regions. A region is an independent failure domain located in a physical site different from other regions so that network issues or natural disasters do not impact more than one region at the same time.

Redundancy within Regions

Redundancy within a region is fairly trivial to achieve.

Within a region, you want to provide protection against failed binaries or physical machines. Typically, you can simply add extra instances of the service binaries running per region, with a load-balancing solution to redirect traffic if binaries or machines are down. The required extent of redundancy is tied to the infrastructure's service level agreement (SLA). Specifically, the SLA accounts for the total number of machines that can be in a failed state simultaneously and the speed in which new instances of binaries can be restarted on new machines.

Understand that redundancy within the region won't protect your service at all from failures that take out the whole region (power, network, natural disaster, etc.).

Redundancy across Regions

Redundancy across regions is far more complex.

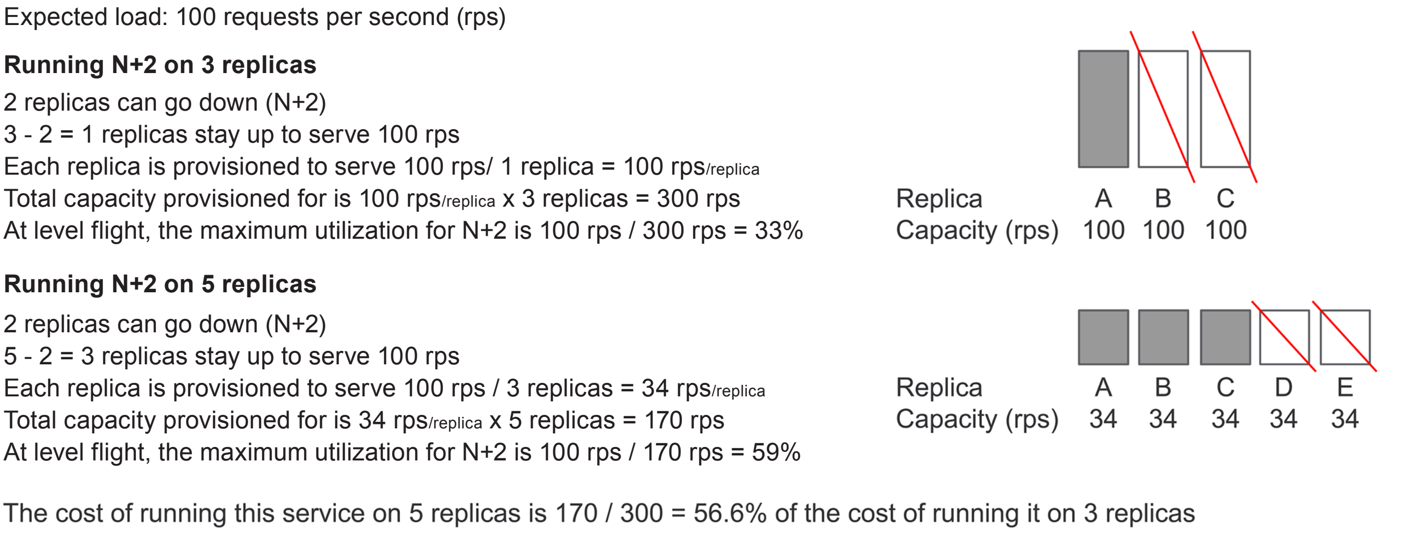
Across regions, you'll need protection from total region outages. By deploying replicas, or full copies of the service stack in several regions, you can implement redundancy across regions to accommodate your service's load at peak. Note, each replica must have enough capacity to serve all of the expected load when any number of replicas are down based on your declared redundancy. As stated above, regardless of the number of replicas (*N*), the degree of regional redundancy of the service is defined as follows:

- **N+0**: when the service is up and running, but cannot tolerate any region going down
- **N+1**: when the service can withstand a single region going down
- **N+2**: when it can still serve with two regions down
- etc.

While some of this redundancy involves capacity, it's also about the service architecture itself. For example, consistent storage services often require that a majority of replicas are up and running to ensure that writes aren't rolled back.

Provisioning a service for N+2 has a positive effect on reliability: maintenance can be planned for an entire region at once, but lowers redundancy to N+1 during the maintenance. The service can still tolerate an unplanned incident in another region. This lowers the redundancy to N+0, but does not cause an outage. Note that failing over to another region may have effects on visible latency.

With N+0 redundancy and no tolerance for further failure, your priority is to mitigate or resolve the unplanned incident as fast as possible. One option is to complete or revert the planned maintenance work to bring the service back to N+1. Otherwise, any other region suffering an incident could cause a user-facing outage.



Expected load: 100 requests per second (rps)

Running N+2 on 3 replicas

2 replicas can go down (N+2)
3 - 2 = 1 replicas stay up to serve 100 rps
Each replica is provisioned to serve 100 rps/ 1 replica = 100 rps/replica
Total capacity provisioned for is 100 rps/replica x 3 replicas = 300 rps
At level flight, the maximum utilization for N+2 is 100 rps / 300 rps = 33%

Replica	A	B	C
Capacity (rps)	100	100	100

Running N+2 on 5 replicas

2 replicas can go down (N+2)
5 - 2 = 3 replicas stay up to serve 100 rps
Each replica is provisioned to serve 100 rps / 3 replicas = 34 rps/replica
Total capacity provisioned for is 34 rps/replica x 5 replicas = 170 rps
At level flight, the maximum utilization for N+2 is 100 rps / 170 rps = 59%

Replica	A	B	C	D	E
Capacity (rps)	34	34	34	34	34

The cost of running this service on 5 replicas is 170 / 300 = 56.6% of the cost of running it on 3 replicas

Figure 2: Example comparison of the cost of resource provisioning a service with three and five replicas

The Cost of Redundancy

The more regions a service operates in, the lower the cost of running any level of redundancy. Consider the service described in Figure 2. It needs to run with N+2 redundancy. In the first setup, it is running three replicas (N=3), and in the second setup, it is running five (N=5). Both configurations have two spare replicas (+2) and thus can withstand two replicas failing.

Next, examine the five-replica setup. Its replicas are smaller in size, and even when two replicas fail and both spare replicas are in use, there are still three active replicas to share the load. This results in the five-replica N+2 setup costing 56.6% of the three-replica service using the same degree of redundancy. See the calculations provided in Figure 2.

Homogeneous and Heterogeneous Services

It's easier to implement redundancy for services with homogeneously sized replicas than those services with heterogeneously sized replicas.

Your service must be provisioned to handle failures in the largest region. If regions have different capacities (i.e., heterogeneous), the capacities needed to withstand the unavailability of the other largest regions are different in each region. The result is that your smaller regions require more resources, and your overall required resources to serve the same load are higher.

Replicated and Distributed Traffic

Provisioning for redundancy also depends on the characteristics of the service's traffic.

Stateless services, such as web servers that handle user requests, receive traffic that is distributed among replicas. Requests that read from storage services can also be distributed across replicas in different regions. Provisioning these for N+1 or N+2 is trivial and follows the logic from the previous example.

Services that handle requests replicated across regions, such as writes, behave differently. Each write to an entity needs to be eventually written to every single replica to keep your service's data consistent across replicas.

When a replica becomes unavailable, replicated write requests do not cause additional load to the replicas that remain up. However, there is a cost incurred when the unavailable replica comes back online. This replica needs to catch up with outstanding writes that were missed during its downtime. This operation increases its load. The replicas that remain running provide the data needed to sync the recovering replica, increasing the load on all replicas during recovery. Ideally, this is capped to avoid hurting low-latency traffic across the entire set of replicas.

Each service and each component can receive a different proportion of replicated and distributed traffic, which need to be factored in when resource provisioning.

Latency-Insensitive Processes

A service typically has latency-insensitive processes such as batch jobs, asynchronous requests, maintenance, and experiments.

However, these processes put additional strain on the service while it handles the production load, which is latency-sensitive. The service thus requires additional resources to accommodate a higher peak, increasing its cost.

You can minimize the extra cost of latency-insensitive requests by assigning them lower priorities or by scheduling them during low-load periods in order to decrease the overall peak. Note, both of these solutions need to be properly tested and carefully deployed to prevent service interruptions.

Additional Resources for the Unknown

The last aspect to consider is the unknown factor. There are many good reasons to throw in additional resources when provisioning a service: for example, the performance regression of an underlying library supported by another team or when implementing a team-external requirement such as encrypting all RPCs.

Spare capacity can keep the service performing as expected, in regards to latency and errors, if anything goes wrong. However, keep in mind that this decision can be expensive, so make sure that the tradeoff in reliability, predictability, and scaling is worth the cost.

Capacity Planning

While *resource provisioning* refers to the process of determining the correct amount of resources to keep your service running right now, *capacity planning* entails forecasting demand to guarantee resource supply in the future.

Overview of Capacity Planning

Like resource provisioning, capacity planning is an attempt to determine the amount of each computing asset (resource class) you need to sustain the service. However, it involves making those determinations at multiple points in time: for example, your resource needs in three months, six months, or a year.

For an existing service, capacity planning uses historic demand to forecast growth to build on top of resource provisioning for your service's maximum peak utilization, redundancy requirements, latency-insensitive processes, and the unknown factor. Generally, you'll want to add to this forecast any planned new consumers of your resources, including new services, marketing campaigns, new features, etc.

You'll need different amounts of each individual resource class for each component in your service. Take RAM, for example. A web server may need a lot of RAM, whereas a proxy server may need very little. To determine the various values of a single resource when you are planning for future capacity, take into account the following:

- The number of different components (database, proxy, application) in your service
- The number of instances of each component (1 database, 2 proxy, 2 application)
- The regions your service runs in (i.e., across-region N+1 or N+2)
- The number of data points you need for your forecast

While this is a simple example of a complex formula, a single resource class like RAM may require you to think in terms of the following:

(# of different components) × (# of instances of each component) × (# of regions) × (# of datapoints) × (other contributing factors)

As you can see, when you consider all resource classes for all server types in all regions and add in redundancy, the number of capacity values that you must determine grows exponentially.

Forecasting Resources

Capacity planning is an extremely complex process as there are myriad factors at play, and each can change independently. Expanding on the high level overview above, consider the following when forecasting:

Resource Classes by Component

In addition to determining the total number of components, you must also consider the various resource classes that each one utilizes: RAM, CPU, storage, network, etc. One component may use one set of resource classes, and others may have a very different set. If your service consists of many components, the set of resource classes that you must track quickly increases.

Multiple Regions

If you are required to run in many regions around the world, you can imagine how forecasting a single resource class such as CPU for various machines (web, database server, application, proxy, etc.) is made even more difficult. Add in all of the other resources classes for all machines, redundancy across all regions over a given period of time (six months or a year from now) to start your planning.

Service Demand

Demand depends on the success and adoption rate of the new service and is only known after the service is launched. You must update forecasts over time and correct long-term predictions. Understand you are preparing for a sudden unplanned load increase that can cause an outage if ignored.

Other unexpected events like natural disasters, network interrupts, or power outages can drastically alter your traffic patterns. Even planned situations such as social events or the beginning or end of holidays can affect your service in unexpected ways. It's challenging to extrapolate the changing impact of such events year to year as new features are launched or the user base varies.

Changes in user distribution in different time zones also have service implications. Traffic may appear more or less spread out across the day, unexpectedly raising and lowering peak demand.

Growth

Growth depends on the success of your service. It may take some time (and marketing campaigns) for users to learn about your service and take interest, and the interest may grow slowly or sharply over time. Other services on the Internet can have a dependency on yours, and their success or failure can directly affect your service. A successful external service can increase traffic to you, and vice versa.

There may be social, economic, political, or other factors that may increase or decrease your user traffic. You have to determine your growth rate and take this into account for your capacity planning sessions.

Forecasting Example

To illustrate the multitude of potential separate resource class values you, as the service owner, must try to predict correctly, let's use a very simple example:

Resource Classes for a Two-Component Service

Suppose you have a small service such as a social media application. It consists of two machines, a web server and a database. The web server uses CPU and RAM, and the database uses CPU, RAM, HDD storage, HDD throughput, and SSD storage. This is a total of six unique resource class values to define. Note, this is far short of a complete set of values in a real-world application.

By having three replicas, you now have 18 values to define. If you are forecasting quarterly for 12 months, that number jumps to 72 (four quarters per year × 18).

Trends That Impact Your Service

You've learned that your social media service is affected by seasonal trends. You have an increase in traffic at the beginning of the holiday season (Nov–Dec), another during spring break, and one more at the start of summer. Your forecasting cannot be just a linear increase in resources, you must account for the spikes during peak times of the year.

You may also experience similar trends with peaks during the month for batch-processing tasks such as data cleanup or database compaction. The load may be different each month, or even each week, further complicating your ability to estimate resource utilization accurately.

Best Practices

We present several best practices for capacity management to help you anticipate common problems and pitfalls.

Load Testing

Run a small replica of the service at target utilization and above, and exercise failover, cache failures, rollouts, etc. Assess how the service reacts to and recovers from overload, and empirically validate that the resource allocation is adequate to serve a defined load. Be careful when extrapolating estimates from your data. If a binary instance allocated with one CPU can serve 100 requests per second, it's generally OK to assume that two binary instances, each with one CPU, can serve 200 requests per second in total. It is not OK to assume that a binary instance with two CPUs allocated can serve 200 requests per second. There may be bottlenecks other than processing power.

Holistically Evaluate the Capacity

While you should add extra capacity for the unknown, avoid stacking too many resources and inadvertently overprovisioning the service. However, provide enough spare resources so the service can withstand issues. This can buy some extra time to secure resources in case the service is more successful than was expected and was provisioned for.

Decrease the Impact of Outages

It's possible to prepare the service so that outages have a lower impact when it runs out of resources. Suggested preventative measures include:

- **Graceful degradation**. The service disables some non-critical features to relieve resource usage when it's overwhelmed.
- **Denial-of-Service (DoS) attack protection**. Provided in case the increased traffic comes from an ill-intentioned party.
- **Effective timeouts**. Requests eventually time out, and the service drops the requests without wasting further resources on them.
- **Load shedding**. The service quickly rejects requests when it's overwhelmed, allowing a routing layer above to retry the requests or make them fail fast. This avoids the issues of a service falling behind and wasting efforts on requests that are going to time out anyway.

Quota Management and Throttling

Deploying a quota system helps limit the throughput between your service and the back end, providing isolation from other services using that same back end. Whenever a service sends more requests than expected and reaches the quota limit, the back end throttles the services rather than overloading itself and impacting other services using that back end.

Monitoring

The relevant metrics gathered from monitoring your service provide data to guide resource provisioning and capacity planning decisions. Using our sample service above as a model, the following are very useful:

Load metrics

- Incoming requests per second
- Latency-insensitive load
- Number of active users
- Number of total users

Resource metrics

- Resource allocations
- Actual resource usage
- Quota usage
- How many requests are throttled

Performance metrics

- Latency
- Errors

High-level health metrics (that can help filter out other tainted metrics data)

- When the service was impacted by an outage
- When the service was undergoing maintenance

Alerting

Use alerts for resource provisioning and capacity planning to prevent outages. Some examples of useful alerts are those that trigger when the service is not at the intended redundancy level and is therefore underprovisioned, alerts that indicate the service lacks future resources according to forecasts, current performance issues, etc.

Resource Pooling

Pooling is the grouping of resources so that several services share them rather than providing separate allocations per service. Pooling is often used to decrease planning complexity and to reduce resource fragmentation, hence, improving the efficiency of a service. When you implement this strategy, planning for large services is still detailed and precise. However, small services use a pool of resources that is provisioned for as a single entity, approximately and conservatively. This decreases the effort on capacity planning at the expense of isolation.

General SRE Best Practices

Follow the basic SRE principles that you would for any service. For example, store the capacity state as a configuration in a version control system and require peer reviews for any changes. Automate enforcement, roll out all changes gradually, constantly monitor your service, and be ready to roll back if needed.

In the event of a failure or other issue, exercise blameless postmortems to honestly learn from the mistakes, and commit to improving the system to avoid repeating them.

Evaluating a Service for Capacity

When evaluating capacity for a new or existing service, we recommend determining its resource requirements by following these steps:

Hardware	**Specs**
Processors	CPU type and count (cores)
Graphics Processing Units	GPU type and count
Storage	HDD (hard drives) and SSD (solid state disk): • Amount of storage (TB) • Bandwidth • IOPS
Network	Intra datacenter, inter datacenter, ISP access: • Latencies • Bandwidth
Back Ends	Services and capacity needed
Other	AI accelerators, other special hardware

Table 1: Resource assessment template

1. Estimate the resources needed to serve the expected load. Use the template in Table 1 and fill it in with the expected service demand for the different resource classes.
2. Calculate and factor in the target utilization of the different components of the service. You may need to perform load testing to assess:
   - Peak usage
   - Maximum peak utilization
   - Redundancy
   - Latency-insensitive processes
   - Spare resources for the unknowns
3. Consider aspects such as:
   - Priority
   - Region
   - Service components
   - Specific points in time and time into the future (monthly, quarterly, for six months, a year, etc.)
4. Perform forecasting, considering whether you need to plan for capacity per:
   - Priority
   - Region
   - Service components
   - Number of points in time per year

5. Continue to learn about capacity management:
    - Watch the video "Complexities of Capacity Management for Distributed Services" for an extended tech talk on the topic [1].
    - Read the *;login:* article "Capacity Planning" [2].
    - Review the "Software Engineering in SRE," "Managing Critical State," and "Reliable Product Launches at Scale" chapters of Google's *Site Reliability Engineering* [3].

Conclusion

In this article we discussed the components and complexities of capacity management. We separated the topic into two parts: *resource provisioning*, which addresses the tactical question, "How do I keep the service running right now?" and *capacity planning*, which addresses the strategic question, "How do I keep the service running for the foreseeable future?" Answering these questions is not a trivial task, and each requires reviewing different aspects of your service.

When provisioning resources, examine the various demand signals (input) and their effect on the resource allocations (output). It helps to understand the expected peak demands the service may face and the amount of redundancy you're required to build into it. Have you considered the impact of resource shortages and vendor supply?

Capacity planning forces you to attempt to predict what the service and, more importantly, its load look like in the ever-changing future. You have to fully understand your service to do this—for example, you need to identify the peak cycles and when they occur, determine the number of locations you must run in and the varying capabilities of each, and anticipate the natural, social, and even legal events that might impact your service. When it's time to add more capacity, do you have the approvals or funds to accommodate the growth?

While the many best practices we presented are all important, following solid SRE tenets helps simplify capacity management: perform proper load testing, implement extensive monitoring and alerting, use source control systems, understand the strengths and weaknesses of your service, develop a capacity plan, and be prepared to anticipate growth and scale when needed.

Acknowledgments

The authors are grateful for the suggestions from JC van Winkel, Michal Kottman, Grant Bachman, Todd Underwood, Betsy Beyer, and Salim Virji.

References

[1] L. Quesada Torres, "Complexities of Capacity Management for Distributed Services," Google Tech Talk: https://www.youtube.com/watch?v=pOo0oKNM9I8.

[2] D. Hixson and K. Guliani, "Capacity Planning," *;login:*, vol. 40, no. 1 (February 2015): https://www.usenix.org/system/files/login/articles/login_feb15_07_hixson.pdf.

[3] B. Beyer, C. Jones, N. R. Murphy, and J. Petoff, eds., *Site Reliability Engineering*, Chapters 18, 23, and 27: https://landing.google.com/sre/sre-book/toc/index.html.