Using dependency isolation to contain concurrency overload

Algorithms imitating life

When I was in high school, I worked summers as a bank teller. My job was to handle transactions—typically deposits and withdrawals—for customers who came into the bank. Each teller had a computer and cash drawer and helped one customer at a time. When all the tellers were busy with customers, any new customer had to wait in line.

Nobody enjoyed standing in line, so tellers tried to serve customers quickly to avoid building up a queue. Unfortunately, two conditions caused queue depth to increase: The first occurred when there was a surge of customers—typically around lunch time or after work, when people ran errands. The second occurred when something else would cause us to take longer to process a transaction, such as a network outage or receiving large or complex transactions at the end of the day from store owners who were depositing a day's worth of payments from their customers. Of course, the bank did its best to avoid extreme queue situations by staffing up for peak traffic and making sure its network was reliable. If the network did go down, tellers could process some transactions offline, like check deposits, and synchronize account balances after the network came back. However, there was always a possibility that the bank could become very busy, and the line could grow. When this happened, customers could get upset about standing in line for a long time, or they might even leave without completing their transaction because the line was too long.

When I started learning about algorithms in college, I thought about this experience at the bank in terms of queuing and thread pools. The queue was the line that customers stood in, and the tellers represented threads that were each pulling work from the queue. Now that I've been working on services at Amazon, I look back and draw comparisons to the applications I've worked on here. When we think about building resiliency into our services, it helps me to remember how we handled the overwhelmed bank branch in the real world. At Amazon, we use techniques like load shedding and rate limiting to handle spikes in request volumes for handling situations like the first overload condition in the bank (the traffic surge). But the second case (the increase in latency) requires other techniques and tools because the extra work doesn't originate from the service's callers. Instead, the extra work in the system comes from latency—our code doing more work or the dependencies that our systems take on taking longer to respond.

In distributed systems, queue times can increase abruptly when there's an increase in request processing latency. Latency can increase for plenty of reasons that are outside of an application's control. For example, say an application is talking to a database. The database could slow down because it's performing more expensive queries than normal, or some background job is putting extra load on it. There is little that an application can do if the slowdown is out of its control. However, the application can make sure that its APIs that don't depend on the database or requests that could have been served out of a cache still work even when the dependency is slow. At Amazon, we call this technique *dependency isolation*.

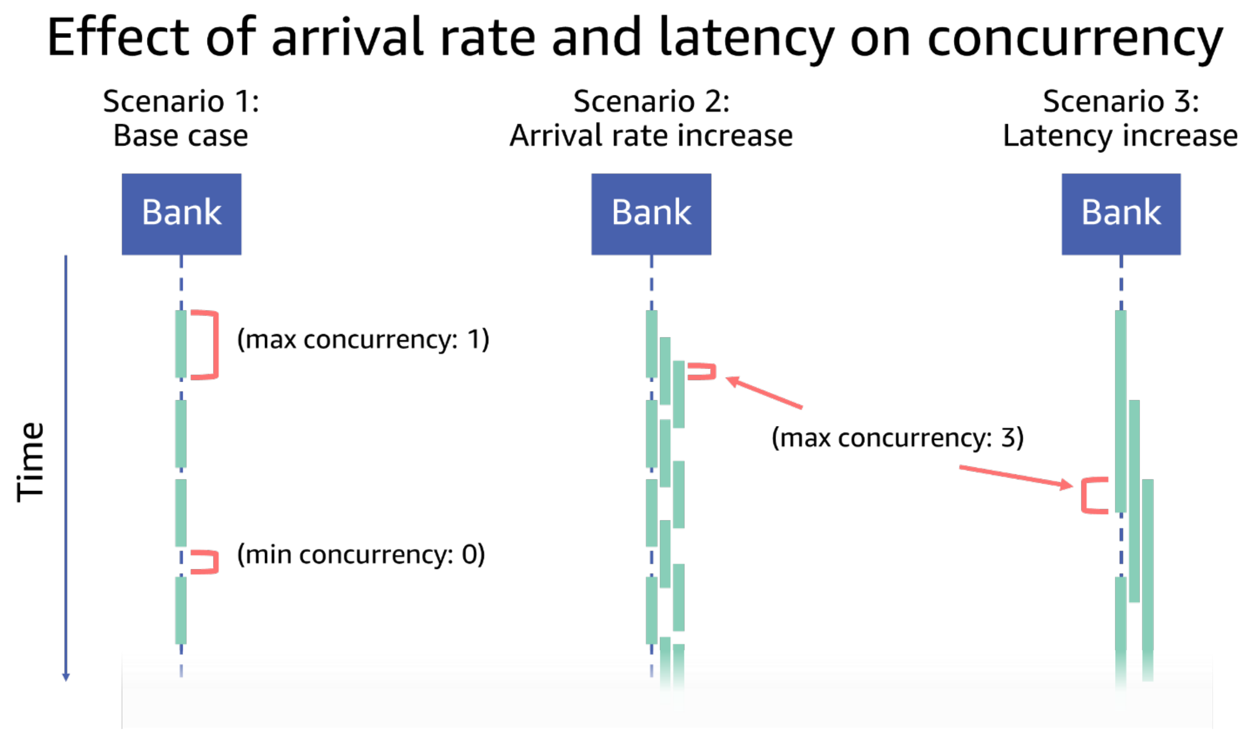
In this article, I examine the approaches we take at Amazon to prevent latency in a dependency from affecting the functionality that doesn't require that dependency. I start by illustrating some concurrency theory, and how increases in concurrency can be a problem for software systems. Then I explore how we use circuit breakers called *bulkheads* to help systems degrade gracefully in the face of an increase in concurrency. I move on to share some of the edge cases and practical considerations around using bulkheads with partitioned data stores, caches, and sections of business logic that have variable latency. Finally, I show you a few examples of how some AWS services, such as AWS Lambda, implement bulkheads to isolate internal dependencies behind the scenes.

Thinking in terms of concurrency

We refer to the number of requests that a service is working on at a point in time as the service's *concurrency*. When a request is being actively worked on by a server, it consumes resources like CPU for parsing or performing authentication. But even when the request is idle, such as when it's waiting for a remote database query to return, it still consumes some resources on the server, such as memory, sockets, file descriptors, or threads. Therefore, when we think about the capacity of a server or other limits in our software, we have to think about its limits that are proportional to concurrency.

There are two factors that increase the concurrency of a service. First, if the incoming request rate increases, the service's concurrency increases. Second, if the incoming request rate stays the same, but processing time for each request increases, the concurrency also increases.

This relationship between request rate and latency is described in Little's Law, which states that concurrency is equal to the arrival rate times the latency. One way to analogize this to the real world is to picture the number of customers who are inside the bank, which is illustrated in the following drawing. Scenario 1 shows at most one customer in the bank at a time. Scenario 2 shows an increase by three times in the customer arrival rate, with each customer spending the same amount of time in the bank as those in Scenario 1. Scenario 3 shows customers arriving at the same rate as those in Scenario 1 but spending three times as much time in the bank. Both Scenario 2 and Scenario 3 show that concurrency has increased three times as compared to Scenario 1.


Effect of arrival rate and latency on concurrency
Scenario 1: Base case
Scenario 2: Arrival rate increase
Scenario 3: Latency increase
Bank
Bank
Bank
Time
(max concurrency: 1)
(min concurrency: 0)
(max concurrency: 3)

Concurrency is affected by latency, and one common source of an application's latency at Amazon comes from calls to its remote dependencies. *Dependencies* are remote service calls the application makes to other microservices, databases, caches, disks, or things it talks to over a network. Remote dependencies can become slow for many reasons, from packet loss somewhere in the network, to servers in the dependency becoming overloaded on their own. On the other hand, locally executing code is typically easier to control with predictable latency, by doing things like intentionally limiting the input size and complexity of inputs. As a remote dependency slows down, the concurrency on the calling service increases. And because concurrency is a limited resource on a server, this means that when a dependency slows down, it can cause a server to breach its limits.

Different ways to run out of concurrency

When a server takes on more work than it can handle, it can behave in unpredictable and suboptimal ways. In cases of extreme overload, it can become too slow to respond, resulting in perceived unavailability from clients that are timing out. At Amazon we use many techniques to prevent servers from becoming overloaded, as described in Amazon Builders' Library articles, Using load shedding to avoid overload and Fairness in multi-tenant systems. However, these articles mainly focus on overloads that are driven by increases in request rate, rather than protecting against overloads from concurrency driven by latency in a dependency.

Overloads due to increases in latency-driven concurrency need to be protected against too. Just as the physical bank where I worked had limits on the amount of standing room there was in line, concurrency drives up usage of system resources, and it's possible to "run out of room" by reaching application, operating system, and server limits. Some examples of ways that an application or server can run out of resources include:

- Holding onto memory about requests. Applications hold onto memory about a request when it is being processed, even if the request is paused and waiting on a dependency. If a server runs out of memory, the operating system might choose unpredictable processes to kill in order to meet new allocation demand. That selected process could be the server process itself and cause a service to lose capacity, or it could be a support process of some kind, which makes it harder to operate the service or results in a gray failure if the process were needed for the service to operate normally.

- Consuming file descriptors. When an application opens a file to read from, or a socket to use to talk to a remote dependency, it consumes a file descriptor. Operating systems protect themselves from being overwhelmed with keeping track of too many file descriptors by limiting each process to a certain limit.
- Reaching maximum connection pool configuration. Applications often communicate with dependencies through connection pools, such as database connection pools and HTTP client connection pools. These pools are often limited with some configured maximum value. If the latency to that dependency increases, more connections in the pool are utilized concurrently, and the maximum can be reached, resulting in errors.
- Filling up thread pools. Applications use pools of threads to process work. If the code running in a thread makes a blocking call to a dependency, then the thread is unusable for making progress on other work until the remote call returns or times out. Thread pools are often configured with an upper bound in terms of number of threads, so increased latency in a dependency can result in thread pools filling up. Non-blocking programming models can help avoid this particular bottleneck by making I/O operations run asynchronously, thus freeing up threads for other work. However, even these applications can run out of concurrency by shifting the bottleneck from threads to the other kinds of constrained resources listed here.
- Increasing queue depth. Applications use queues to buffer incoming work and to exchange work between internal parts of an application. If the time to process each item in the queue increases, the depth of that queue increases. If the queue grows unbounded, it can cause the server to run out of memory. Even if bounded in size, queues can still cause problems. If a queue is consumed in a first-in-first-out algorithm, then by the time work is pulled off the queue, it might be too late for the work to be completed in time, but the application will work on it anyway and waste resources on the server and in any dependencies it calls.

Implementing an overall concurrency limit

With the variety of failure modes and the difficulty in predicting their effects on a system, it's important to use bulkheads to limit the amount of overall concurrency in the system to prevent overload. Implementing an overall concurrency limit is relatively straightforward. When a service receives a request, it can count "up" on a variable, and when it completes the request, it can count "down" on that variable. If its value exceeds some limit, the server can simply reject new work until the measured concurrency dips below the limit.

Although a concurrency limiter will prevent overload driven by an increase in latency, it can still be improved upon to help a service degrade more gracefully. Services often expose multiple APIs, and a particular dependency might only be needed for a subset of its APIs. So if a dependency becomes slow, we want to ensure that only the functionality of the service that depends on that API is affected by the concurrency limiter, and the other functionality that doesn't need the dependency keeps working.

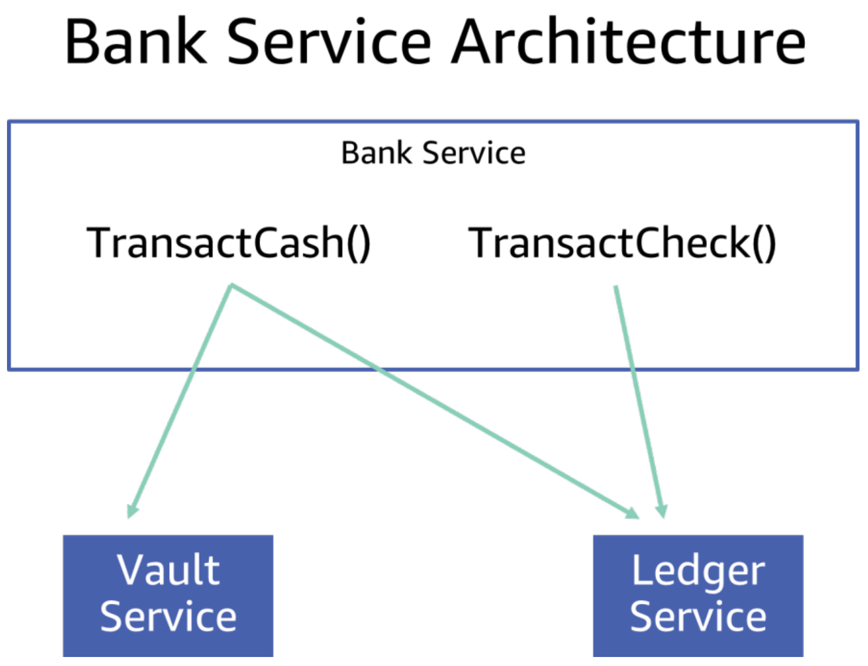
Using limits to control the blast radius

One simple approach for isolating dependencies from each other is to limit the dependencies' concurrency separately by creating a different bulkhead for each dependency. To give you an analogy of how this would work, let's go back to the bank to talk about a real-world scenario.

As a bank teller, I'd handle two types of transactions: cash or check. Check deposits or withdrawals were quick because these transactions only involved validation and data entry. But cash deposits took longer. Cash transactions required very careful counting of cash and coins. Sometimes they required asking a supervisor to go to the vault and either retrieve or store larger amounts of money than I could hold in in my drawer (we'll call this a *cash miss*). When the bank was busy and the line backed up, the people who had a transaction that they thought would be quick to process, like depositing a check, were the most upset. Sometimes a customer with a large cash transaction would let other people behind them in line go first, which was very considerate. Sometimes the group of tellers would ad-hoc organize into two sets—an "express" set for quick, check-only transactions, and an "everything" set for handling any type of slower transaction. This kept the average queue time down, and it also reduced the total number customers in the bank at any time. The following illustration shows a bank with a slow line for cash transactions and a second express line for check transactions.

In extreme cases, all cash transactions started taking a long time while the supervisor went to and from the vault. So even with a separate express line for check transactions, eventually the bank filled up with customers waiting for cash transactions. So to keep this "overload" from happening, the bank could simply limit the length of the line for the slower cash transactions, ensuring there was always room in the bank for check deposits.

The technique from the real world maps well to the virtual world. In a threaded server where different sets of APIs have different dependencies, this could be as simple as using a different thread pool for each group of APIs. For example, let's say that the bank I worked at were a service called `BankService`. The service has two APIs – `TransactCheck` and `TransactCash`. The `TransactCheck` API depends on the ledger, and does simple bookkeeping, while the `TransactCash` API depends on the same ledger, but also on the `VaultService`. The following diagram shows how `TransactCheck` and `TransactCash` work using this architecture.



To protect the dependencies from each other, let's say we implement the service using two thread pools, each with its own maximum size. Even if the `VaultService` gets slow, only the `TransactCash` thread pool fills up, and the service keeps serving `TransactCheck` calls without any degradation. To the end users of this service, the result is positive. Instead of the whole bank service being affected when the `VaultService` is slow, customers can still conduct check transactions.

This approach works in a non-blocking server implementation as well. But instead of having a separate thread pool for each API, we maintain a numeric variable to count the concurrency of each API. If a request comes in and the server is already working on the maximum number of configured concurrency for that API, the request is rejected.

Although we describe both of these approaches in terms of hard-coded limits, we find that it's important to be able to reconfigure this kind of load shedding configuration at runtime instead of being configurable only at system startup. To implement this in a threaded server, rather than attempting to reconfigure thread pool sizes, we often implement the threaded server with a single thread pool and use the same simple variable-based concurrency checking technique as the non-blocking approach.

Hard vs. soft allocation of concurrency limits

When it comes to allocating concurrency limits to each API or dependency, we use a variety of techniques: hard-allocate, soft-allocate, or dynamically allocate.

The simplest option is to hard-allocate the available concurrency to each API. Going back to the `BankService` example, let's say we perform a load test against the service and find that it can handle 50 concurrent requests, regardless of the mix of the two APIs. Because we can't predict which API will be called more than the other, we set the concurrency limit of each to 25.

Unfortunately, hard allocation makes full resource utilization tricky. If our clients end up calling the `TransactCheck` API far more often than the `TransactCash` API, we start returning errors, even though the server utilization is low. Clients can change their access pattern unpredictably, making it impossible to pick a "perfect" allocation up-front.

Another option is to soft-allocate the available concurrency to each API. Instead of hard allocating 25 concurrent requests to each API, we allocate 35 concurrent requests to each API and limit the service to handle 50 concurrent requests as a whole. Now if the traffic distribution is different than we expected, there is some room for one API to dip into the other API's concurrency. Unfortunately, this approach has a downside as well: if one API's dependency becomes slow, now that API can hog 35 threads, leaving only 15 for the other API. If the other API were still being called at a high enough rate, the service could end up still running out of capacity.

A more complicated approach is to attempt to balance the "hard" and "soft" approaches by permitting one API to dip into the shared buffer for a short duration, but over time limiting it to a smaller value. This approach absorbs bursts of traffic, and protects against a long-duration slow dependency, but has the same problem with sustained traffic as the hard-allocation approach does, if that extra concurrency was actually being helpful before it was ratcheted back. This approach is complex, and it's difficult to prove its efficacy in every workload the service will see in production.

A comparable way to frame these approaches is to categorize work as "droppable" or "non-droppable." Ideally of course a service doesn't drop any requests, but at a moment in time where a service is out of capacity, it has to make a decision. At some high threshold, at say 90% of the service's maximum concurrency, the service begins rejecting "droppable" work and only accepts work that is "non-droppable." This prioritization can be extended to involve more categories, each with its own "droppable" threshold.

There are plenty of variations on these themes of resource allocation, sharing, and borrowing. As the approach becomes more complicated, it also becomes more difficult to test, instrument, and monitor to ensure that it works when needed, and that it doesn't have unforeseen downsides. Because more complex approaches tend to be harder to test and reason about, we strive to use the simplest approach possible, and we add more complexity only after we prove we need it through testing or operational experience.

Timeouts are not enough

Timeouts are one tool we use for limiting the impact of a slow dependency. When the service makes remote calls to a dependency, we configure our client libraries to use a maximum amount of time to wait for the service to return. If that timer elapses, the service might even retry in case it succeeds the next time around, so it returns success to the caller with a perceived latency increase instead of a failure. Unfortunately, a retry ties up that concurrency for even longer on our service, which could cause the service to run out of concurrency if it happens frequently enough. To avoid retries from becoming a problem, we use algorithms to automatically turn off retries when the dependency is consistently failing. Our approach to timeouts and retries is described in the Amazon Builders' Library article, Timeouts, retries, and backoff with jitter.

Although it's important to set retries to reasonable thresholds, tuning timeouts and retry policies is unlikely to prevent services from running out of concurrency. If the typical latency for a remote call is 10 milliseconds, and the timeout is set to 100 milliseconds or 1 second, then a latency increase in a dependency could still be a 10 times or 100 times increase in concurrency, which could likely overload the service.

Instead of focusing on timeouts as a concurrency protection mechanism, we instead configure timeouts based on other considerations, such as aligning the timeout configuration of a dependency we call with the timeout that our own caller is using, to avoid wasting work when our caller stops waiting for a response. To contain the concurrency-related blast radius from a pathologically slow dependency, we focus on getting the right concurrency bulkheads in place around each client.

It's not always the dependency's fault

So far, we've described dependency isolation in terms of dependencies slowing down. However, at Amazon we've seen that a service can slow down, and sometimes hit concurrency limits, even when its dependencies are fast.

For example, a dependency might start returning errors. Maybe these are server-side errors, or maybe the dependency is throttling the service's requests because it's being called too often. In either case, sometimes client libraries or SDKs will automatically retry. After all, retries can paper over transient errors and improve your service's overall availability. But the problem with a retry is that it increases the application's latency for that codepath. This problem is greatly amplified by backoff between retries, in which the client will intentionally sleep in between attempts. So although the actual latency of the dependency might be fine, the application slows down dramatically because of the way it interacts with that dependency.

Here's another example of a slowdown that occurs with the workload the service sends to a dependency: Let's say our `BankService` has a new API called `ListTransactions` that returns up to 100 transactions to its client. When it queries its transactions database, let's assume it returns paginated results, 10 at a time. Let's say that typical queries are for "today's transactions," and they return only four transactions, on average. This means that the service typically makes only one call to the database every time a client calls the `ListTransactions` API. But if clients start performing queries over longer time ranges that would return 100 transactions, the service would need to make 10 back-to-back calls to the database. The actual remote call latency to the database dependency stays the same, but the business logic that interacted with the dependency takes longer.

In both cases, the business logic exhibits dangerous modal behavior: In the first situation, the code behaved one way, but then in the second situation, the code became more expensive. These modes make operating a service difficult and should be avoided as much as possible. The good news is that the concurrency-limiting strategies we described for isolating dependencies still work, since we isolate the concurrency of each API altogether. The point is that you need to guard against the concurrency of your code that uses the dependency, not just the concurrency spent making remote calls. One technique we use at Amazon for designing resilient systems is to design around this kind of modal behavior by performing constant work on every request, as described in the Amazon Builders' Library article, Reliability, constant work, and a good cup of coffee.

How micro of a microservice?

There's another way to deal with the issue of dependency isolation that's more structural: Split the service into separate services. This approach is common at Amazon because a service-oriented architecture is deeply embedded in our DNA. However, it can still be difficult to decide how many smaller services to split a larger service into. It requires weighing factors like operational overhead, scaling, deploying, and overall system complexity. However, we find that dependency isolation is a useful lens to look through. If a service has multiple modes of operating—like serving different APIs with different sets of dependencies—then we can make our overall systems more reliable by operating them independently as separate services.

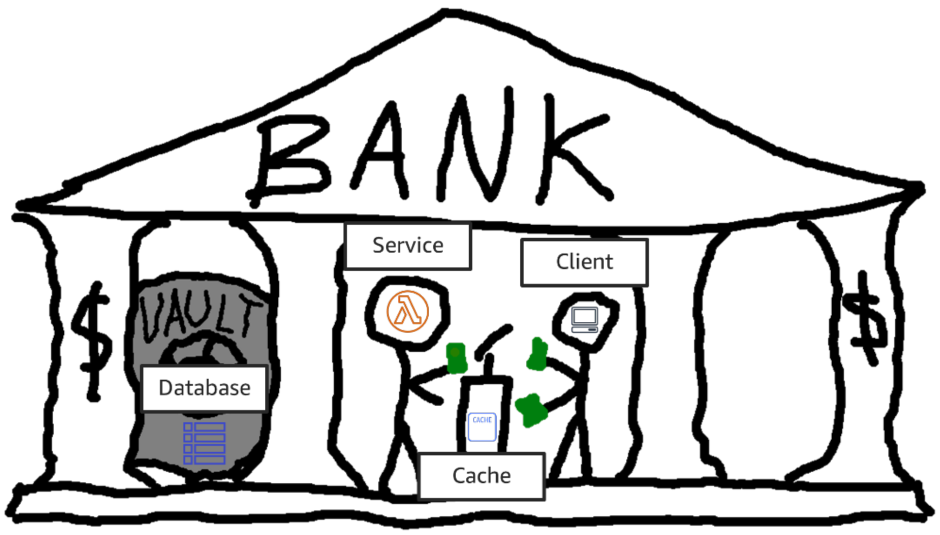
Taking this a step further, when we implement our services using AWS Lambda, we can configure each of our APIs to execute as a different Lambda function. A feature in AWS Lambda, called *per-function-concurrency,* lets developers reserve a maximum amount of concurrency for each Lambda function. That way, because functions have different dependencies and different failure modes or traffic spikes, they're isolated from affecting one another. In essence, we've taken our experience with dependency isolation, and baked it into AWS Lambda to make it easier for developers to implement.

Fortunately, breaking a service into multiple services for dependency isolation has gotten easier over time. And it can be done without complicating things for the clients of the service. Amazon proxy services like Application Load Balancer and Amazon API Gateway enable a service owner to present a single endpoint to its callers, but route different workloads to different services behind the scenes.

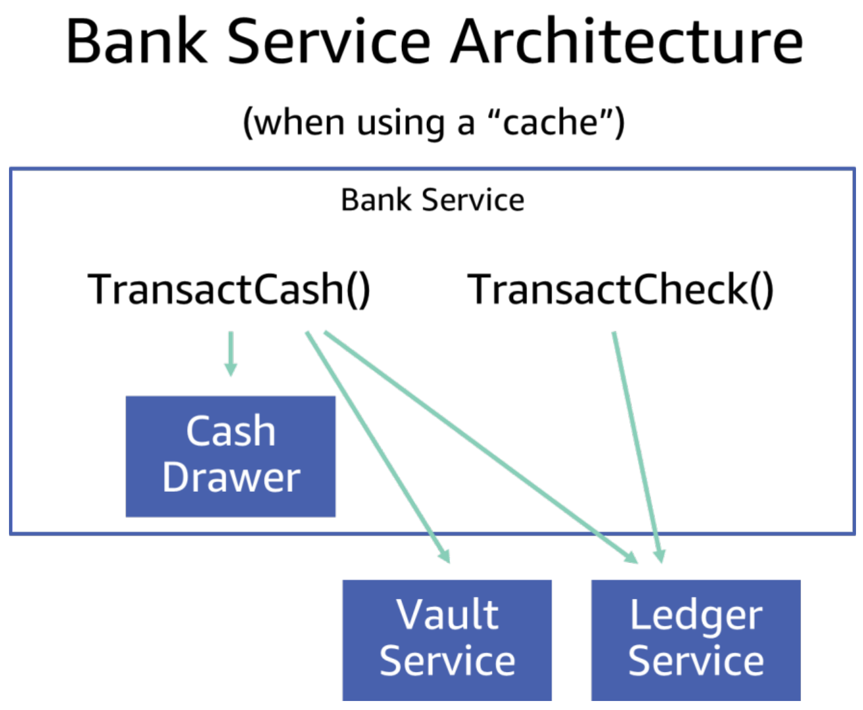
Dependency isolation and caching

When I worked at the bank, not every transaction involving cash required me to ask my supervisor to go to the vault. Instead, each teller had their own cash drawer for keeping small amounts of cash for accepting small deposits and taking out small withdrawals. If my cash drawer had a balance that was too high, I'd ask my supervisor to store the excess in the vault, and if a customer wanted a large withdrawal, I'd ask the supervisor to get more money out of the vault.

In a way, this model loosely maps to a service architecture that uses a database (the vault) and a cache (the *cash* drawer—pun actually intended, even though at Amazon we don't tend to assume anything about durability in a cache). When we use this "database with a cache" architecture at Amazon, we're careful to assess the risks of modal behavior, as described in the Amazon Builders' Library article, Caching challenges and strategies. The following illustration shows the analogy: a bank teller (the service) uses the cash drawer (the cache) instead of the vault in an interaction with a customer (client).



In terms of our pretend `BankService`, we can update the architecture to incorporate the cash drawer, as the following diagram illustrates. Now, before `BankService` calls the `VaultService`, it checks its local cache drawer to see if it had enough money there, and if so, skips the call to the vault.



Now the single `TransactCash` API has three dependencies: its cache, and the `VaultService` and `LedgerService` like before. A service that uses a cache also operates under two modes; cache hits are faster and cheaper to serve than cache misses. If the `VaultService` becomes slow or unavailable, the `BankService` can still answer client requests if it has everything it needs in the cache. However if we forget to isolate the service's concurrency for performing for `VaultService` calls from the concurrency it uses for serving cache hits, it could run out of overall concurrency and fail both types of requests. So, instead, we limit the concurrency of calls to `VaultService`, therefore leaving capacity available for serving requests out of the cache and retaining some availability even when a different dependency is having problems.

Circuit breakers and partitioning

The types of dependency isolation discussed in this article are forms of circuit breakers. (Some open-source libraries like Resiliency4j refer to this specific kind of concurrency isolator as a *bulkhead*.) *Circuit breakers* are logic that prevents one workload, mode of operating, or dependency from affecting an entire application. They trigger when things aren't "working well" to fast-fail new requests instead of accepting them, in hopes that by shedding load, the system recovers.

The circuit breaker techniques described so far have been designed to isolate each remote dependency separately so that functionality unrelated to the slow dependency continues to work. However, some dependencies are composed of infrastructure or constructs that are designed to fail independently, such as a partitioned database, or a dependency with modes of its own (like serving results from its own cache instead of its database).

Amazon services are built to be horizontally scalable. To deal with state in a scalable way, they often use a horizontally scalable, partitioned data store, such as Amazon DynamoDB. These data stores have partitions that operate independently, with independent limits. If requests exceed the limits of a single partition (say by driving more than 3,000 reads / second to it – the limit at the time of writing), requests will begin to be rejected to that partition, but requests to other partitions will be unaffected. So with 10 partitions, a single partition overload lowers the availability to 90%, instead of to 0% if there had been only one partition.

If our application were to wrap calls to DynamoDB in a single circuit breaker, then if a circuit breaker sees many throttling errors talking to a single partition, the circuit breaker might erroneously conclude that the entire table was having an issue, leading the circuit breaker to proactively fail future calls that would have been served by other partitions just fine.

There are some tough trade-offs that we have to make when implementing circuit breakers for a dependency that is a partitioned data store. With a partitioned data store, we often don't know which partition a request will be routed to. Therefore, the client has to treat the entire partitioned service as a single dependency, and set the circuit-breaker logic at that level, rather than at a fine-grained partition level. However, partitioned data stores like DynamoDB expose some higher-level grouping concepts, like table, with independent limits and sets of partitions, so circuit breakers can be applied at a table level rather than at a service level, to achieve some more fine-grained specificity.

When we build systems that interact with a partitioned dependency where the partitions are transparent, like a distributed Memcached or Redis cluster, we implement a circuit breaker that is partition-aware and avoid this whole problem. Otherwise, we implement some heuristics to detect certain cases where requests are repeatedly going to the same partition. For example, we can feed the partition key into a heavy hitters algorithm to notice when we are accessing the same partition at a high rate. It won't catch all cases, but it can be a helpful "hot key" detection heuristic.

When we use a bulkhead to guard calls to an opaquely partitioned dependency, we have a false positive problem. That is, the bulkhead will cause requests to fast-fail that would otherwise have succeeded because the requests would have landed on a healthy partition. However, even an imprecise circuit breaker still can be valuable. If we had left out the bulkhead completely, an overloaded partition would cause the whole service to run out of concurrency, and requests that didn't even need to talk to that dependency would have failed as well. But this discussion shows that we've learned the importance of not making circuit breakers as closely aligned to the infrastructure that they are guarding against as possible to avoid amplifying the blast radius of a small outage within a dependency and turning it into a perceived outage across the whole dependency.

Circuit breakers and operational visibility

When we use circuit breaking techniques like this at Amazon, we focus heavily on the operational visibility we need to ensure that the circuit breaker is working correctly. This allows us to see how close to various limits the service is over time, to alarm when limits are too close, and to distinguish failures caused by the circuit breaker tripping versus a timeout or failure talking to the dependency. To accomplish this, we include instrumentation throughout the code to measure the concurrency for each circuit breaker at the start of each request. That way we can plot statistics like the median and maximum concurrency over time. This is also useful when introducing the circuit breaker in the first place. By deploying it in "instrumentation only" mode, we can ensure we have the right configuration that won't cause an unintended background failure rate. After we have the configuration right, we can flip it into "enforcement mode." For an in-depth description of how we instrument our applications, check out the Amazon Builders' Library article, Instrumenting distributed systems for operational visibility.

Dependency isolation in AWS Lambda

AWS Lambda provides a real-world example of dependency isolation. AWS Lambda is a serverless service that executes your code without the need to provision or manage servers. To set context for how Lambda achieves dependency isolation, let's first understand the Lambda architecture and follow a request through it.

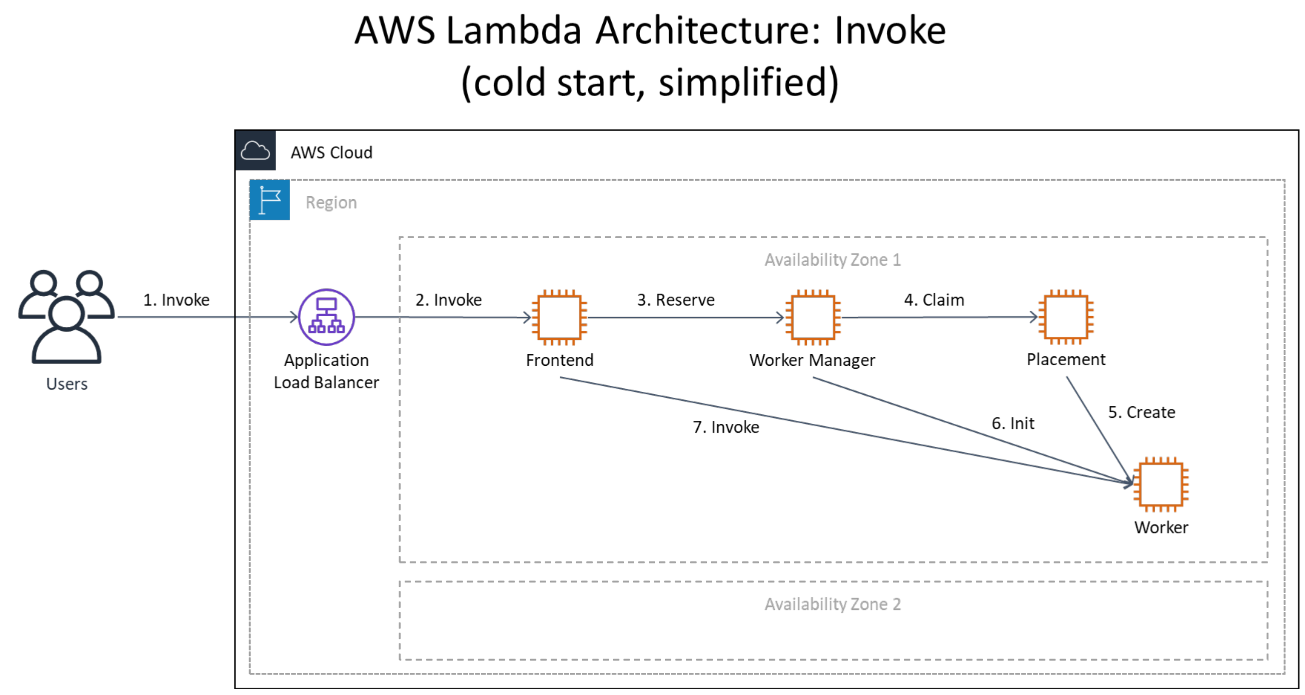
When you call Lambda, you're talking to the Lambda frontend invoke service. The frontend invoke service performs authentication and request validation, and it orchestrates the execution of your function in the Lambda backend.

AWS Lambda runs your code in [execution environments]() on instances called *workers*, which it creates on the fly as needed. Spinning up execution environments is fast, but not instantaneous, so Lambda does its best to re-use each execution environment for a while before tearing them down after they've been idle for a period of time.

The *worker manager* is in charge of keeping track of which execution environments already exist for each function. When you invoke your function on Lambda, the frontend invoke service asks the worker manager to look up the location of a suitable execution environment.

However, there might not always be an idle execution environment for your function already running on a worker. In this case, the worker manager asks the placement service to figure out the ideal worker on which to run your new execution environment.

The following diagram shows what happens behind the scenes when you ask Lambda to invoke a new function in the case where Lambda needs to create a new execution environment. (Note that this is a simplified view of the Lambda architecture used to illustrate the relevant interactions.)



The example of dependency isolation in Lambda is within the worker manager service. Recall that if the worker manager has an idle execution environment for that function, it marks it as busy and returns it. If it doesn't have an idle execution environment, it calls the placement service for a new one to be created. This introduces two modes of operation for worker manager. It responds right away when it already has an execution environment for your function (the *warm invoke* path) and it takes slightly longer to respond when it needs to ask the placement service for a new execution environment.

Worker manager doesn't know whether or not it's going to call the placement service, so it has one thread pool for handling all of its requests. Because worker manager operates in the two modes described earlier, it protects one mode from affecting the other. For example, it limits the number of threads that can be concurrently waiting for placement service to respond. That way, if placement service were to become slow, worker manager still has plenty of threads to handle the warm invoke path and route invoke requests to the execution environments that are already running. Of course we put a great deal of engineering effort into continuously improving the performance and resiliency of placement service, but we still find that it's important for clients of services, like worker manager, to isolate their different workloads to limit the blast radius of their dependencies—so that only functionality in a service requiring a critical dependency can be affected by an issue with that critical dependency.

Concurrency isolation in Amazon EC2

Concurrency isolation is also something that the Amazon Elastic Compute Cloud (Amazon EC2) service teams think about a lot. AWS Regions are composed of multiple isolated Availability Zones that are designed to be independent. Each Availability Zone is separated by a meaningful physical distance from other zones to avoid correlated failure scenarios due to environmental hazards like fires, floods, and tornadoes. Each Availability Zone has independent physical infrastructure: dedicated connections to utility power, standalone backup power sources, independent mechanical services, and independent network connectivity within and beyond the Availability Zone.

When your instance interacts with the Amazon EC2 control plane (the part that launches instances or returns details about your running instances), it is talking to an endpoint for the Region that your instance lives in, even though your instance lives in a particular Availability Zone within that Region. This is so you can describe your instances in a Region by providing their instance ID, without having to remember which Availability Zone you launched it into.

After the Amazon EC2 regional control plane performs the minimal logic to validate, authenticate, authorize, and figure out the relevant Availability Zone for your request, it forwards your request to a separate service that handles requests only for instances that live in that Availability Zone. This way most of what happens behind the scenes in the Amazon EC2 control plane logic and dependencies is zonal, sharing fate with the Availability Zone infrastructure it runs in. This means that if Amazon EC2 has trouble launching instances in one Availability Zone, you might not be able to launch or describe instances in that Availability Zone, but you can launch or describe instances in the healthy Availability Zones. However there is still one Region-wide endpoint for Amazon EC2. Amazon EC2 has gone to great lengths to make this Region-wide endpoint operate smoothly in the face of impact to a particular Availability Zone. Ultimately this comes down to having strict resource isolation, including concurrency and thread pool isolation in its frontend Regional endpoint, using separate resources to talk to other zonal services in each Availability Zone. More about how we built the Amazon EC2 control plane around Availability Zones is described in the Amazon Builders' Library articles, Static stability using Availability Zones and Minimizing correlated failures in distributed systems.

Conclusion

In a service-oriented architecture like ours at Amazon, each service is likely to have multiple dependencies. If one of a service's dependencies becomes slow, that service also becomes slow, driving up the amount of concurrent work in the service. Because that service has limits on how much concurrency it can handle (threads, memory, sockets), a slow dependency can have an impact on the entire service—affecting even the APIs in the service that don't share that dependency.

At Amazon, our service-oriented architecture is many layers deep. A service might make a synchronous call to another service, which makes a synchronous call to another service, and so forth. In this model, a significant latency increase in a downstream service is increasingly difficult for an upstream service to deal with the further it is away from the source of the slowdown. And it's harder to deal with a slowdown than it is to deal with other modes like fast failures. With fast failures, upstream applications can use techniques like stale caching to mask problems in dependencies. Latency complicates this by making error handling more difficult, and it consumes resources in client applications too. This is why we prefer to implement services that fail fast, rather than trying to queue up work and introduce a modal behavior of high latency.

Fortunately, we have found ways to contain the impact of a slow dependency to only the APIs that depend on it. We do this by separating thread pools or using simple counting variables to reject excess work for each API or dependency. If we are concerned enough about the impact of one API on another, we will even split the APIs into separate services.

More generally, services might have multiple “modes”: different APIs with different dependencies, different codepaths depending on input, different request or response sizes, and different caches. Modal behavior in systems is hard to reason about, and it should be minimized to make systems easier to test and operate. However, we also find that we need to balance this with other practical implementation and operational considerations, so systems often have some variety of modes.

When our systems have these modes, we isolate the resources that we use to interact with each dependency so that a slowdown in one doesn’t starve the application from some concurrency-related resource. As I said at the outset of this article: We try to prevent a slow single dependency from affecting the functionality that doesn’t require that dependency.

And finally, we find that if we don’t regularly test our dependency-isolation circuit breakers, then we should assume they don’t work. These measures are easy to get wrong and hard to prove without thorough testing, using tools like the AWS Fault Injection Simulator. Test!