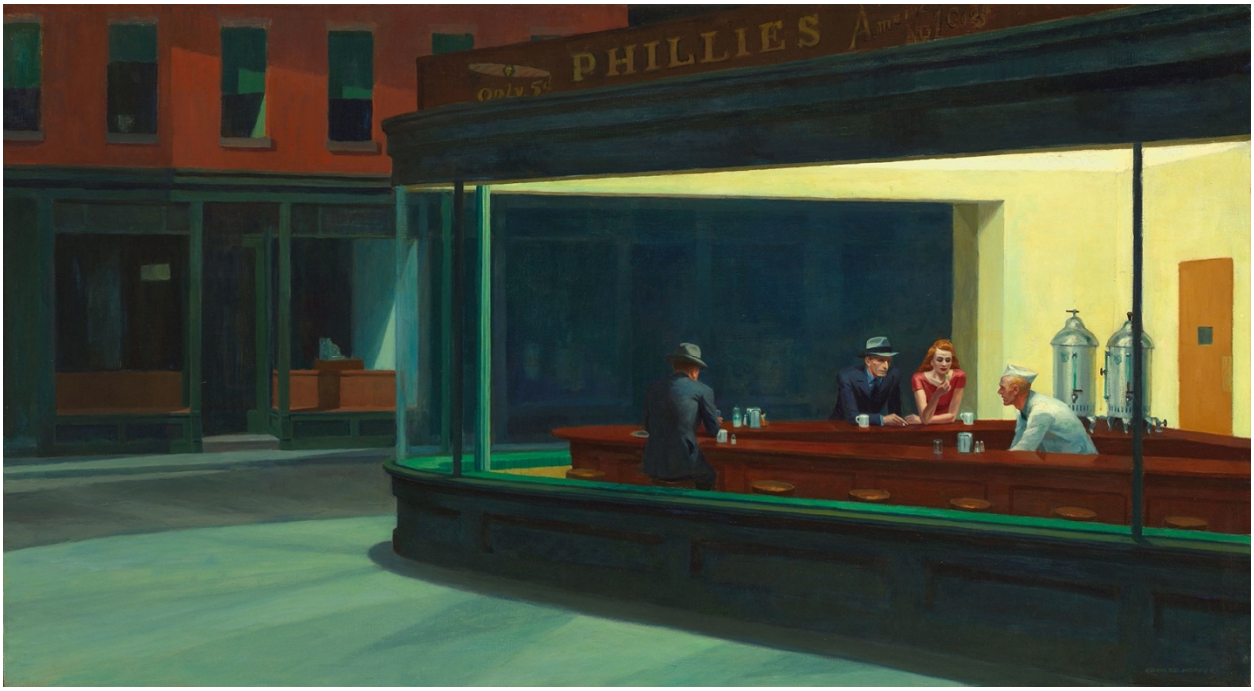
Reliability, constant work, and a good cup of coffee

Colm MacCárthaigh

One of my favorite paintings is "Nighthawks" by Edward Hopper. A few years ago, I was lucky enough to see it in person at the Art Institute of Chicago. The painting's scene is a well-lit glassed-in city diner, late at night. Three patrons sit with coffee, a man with his back to us at one counter, and a couple at the other. Behind the counter near the single man a white-coated server crouches, as if cleaning a coffee cup. On the right, behind the server loom two coffee urns, each as big as a trash can. Big enough to brew cups of coffee by the hundreds.

Coffee urns like that aren't unusual. You've probably seen some shiny steel ones at many catered events. Conference centers, weddings, movie sets. . . we even have urns like these in our kitchens at Amazon. Have you ever thought about why coffee urns are so big? Because they are always ready to dispense coffee, the large size has to do with constant work.



If you make coffee one cup at time, like a trained barista does, you can focus on crafting each cup, but you'll have a hard time scaling to make 100 cups. When a busy period comes, you're going to have long lines of people waiting for their coffee. Coffee urns, up to a limit, don't care how many people show up or when they do. They keep many cups of coffee warm no matter what. Whether there are just three late-night diners, or a rush of busy commuters in the morning, there'll be enough coffee. If we were modeling coffee urns in boring computing terminology, we could say that they have no scaling factor. They perform a constant amount of work no matter how many people want a coffee. They're O(1), not O(N), if you're into big-O notation, and who isn't.

Before I go on, let me address a couple of things that might have occurred to you. If you think about systems, and because you're reading this, you probably do, you might already be reaching for a "well, actually." First, if you empty the entire urn, you'll have to fill it again and people will have to wait, probably for a longer time. That's why I said "up to a limit" earlier. If

you've been to our annual AWS re:Invent conference in Las Vegas, you might have seen the hundreds of coffee urns that are used in the lunch room at the Sands Expo Convention Center. This scale is how you keep tens of thousands of attendees caffeinated.

Second, many coffee urns contain heating elements and thermostats, so as you take more coffee out of them, they actually perform a bit *less* work. There's just less coffee left to keep warm. So, during a morning rush the urns are actually more efficient. Becoming more efficient while experiencing peak stress is a great feature called *anti-fragility*. For now though, the big takeaway is that coffee urns, up to their limit, don't have to do any more work just because more people want coffee. Coffee urns are great role models. They're cheap, simple, dumb machines, and they are incredibly reliable. Plus, they keep the world turning. Bravo, humble coffee urn!

Computers: They do exactly as you tell them

Now, unlike making coffee by hand, one of the great things about computers is that everything is very repeatable, and you don't have to trade away quality for scale. Teach a computer how to perform something once, and it can do it again and again. Each time is exactly the same. There's still craft and a human touch, but the quality goes into how you teach computers to do things. If you skillfully teach it all of the parameters it needs to make a great cup of coffee, a computer will do it millions of times over.

Still, doing something millions of times takes more time than doing something thousands or hundreds of times. Ask a computer to add two plus two a million times. It'll get four every time, but it will take longer than if you only asked it to do it once. When we're operating highly reliable systems, variability is our biggest challenge. This is never truer than when we handle increases in load, state changes like reconfigurations, or when we respond to failures, like a power or network outage. Times of high stress on a system, with a lot of changes, are the worst times for things to get slower. Getting slower means queues get longer, just like they do in a barista-powered café. However, unlike a queue in a café, these system queues can set off a spiral of doom. As the system gets slower, clients retry, which makes the system slower still. This feeds itself.

Marc Brooker and David Yanacek have written in the Amazon Builders' Library about how to get timeouts and retries right to avoid this kind of storm. However, even when you get all of that right, slowdowns are still bad. Delay when responding to failures and faults means downtime.

This is why many of our most reliable systems use very simple, very dumb, very reliable constant work patterns. Just like coffee urns. These patterns have three key features. One, they don't scale up or slow down with load or stress. Two, they don't have *modes*, which means they do the same operations in all conditions. Three, if they have any variation, it's to do less work in times of stress so they can perform better when you need them most. There's that anti-fragility again.

Whenever I mention anti-fragility, someone reminds me that another example of an anti-fragile pattern is a cache. Caches improve response times, and they tend to improve those

response times even better under load. But most caches have *modes*. So, when a cache is empty, response times get much worse, and that can make the system unstable. Worse still, when a cache is rendered ineffective by too much load, it can cause a cascading failure where the source it was caching for now falls over from too much direct load. Caches appear to be anti-fragile at first, but most amplify fragility when over-stressed. Because this article isn't focused on caches, I won't say more here. However, if you want to learn more using caches, Matt Brinkley and Jas Chhabra have written in detail about what it takes to build a truly anti-fragile cache.

This article also isn't just about how to serve coffee at scale, it's about how we've applied constant work patterns at Amazon. I'm going to discuss two examples. Each example is simplified and abstracted a little from the real-world implementation, mainly to avoid getting into some mechanisms and proprietary technology that powers other features. Think of these examples as a distillation of the important aspects of the constant work approach.

Amazon Route 53 health checks and healthiness

It's hard to think of a more critical function than health checks. If an instance, server, or Availability Zone loses power or networking, health checks notice and ensure that requests and traffic are directed elsewhere. Health checks are integrated into the Amazon Route 53 DNS service, into Elastic Load Balancing load balancers, and other services. Here we cover how the Route 53 health checks work. They're the most critical of all. If DNS isn't sending traffic to healthy endpoints, there's no other opportunity to recover.

From a customer's perspective, Route 53 health checks work by associating a DNS name with two or more answers (like the IP addresses for a service's endpoints). The answers might be weighted, or they might be in a primary and secondary configuration, where one answer takes precedence as long as it's healthy. The health of an endpoint is determined by associating each potential answer with a health check. Health checks are created by configuring a target, usually the same IP address that's in the answer, such as a port, a protocol, timeouts, and so on. If you use Elastic Load Balancing, Amazon Relational Database Service, or any number of other AWS services that use Route 53 for high availability and failover, those services configure all of this in Route 53 on your behalf.

Route 53 has a fleet of health checkers, broadly distributed across many AWS Regions. There's a lot of redundancy. Every few seconds, tens of health checkers send requests to their targets and check the results. These health-check results are then sent to a smaller fleet of aggregators. It's at this point that some smart logic about health-check sensitivity is applied. Just because one of the ten in the latest round of health checks failed doesn't mean the target is unhealthy. Health checks can be subject to noise. The aggregators apply some conditioning. For example, we might only consider a target unhealthy if at least three individual health checks have failed. Customers can configure these options too, so the aggregators apply whatever logic a customer has configured for each of their targets.

So far, everything we've described lends itself to constant work. It doesn't matter if the targets are healthy or unhealthy, the health checkers and aggregators do the same work every time. Of course, customers might configure new health checks, against new targets, and each one adds slightly to the work that the health checkers and aggregators are doing. But we don't need to worry about that as much.

One reason why we don't worry about these new customer configurations is that our health checkers and aggregators use a cellular design. We've tested how many health checks each cell can sustain, and we always know where each health checking cell is relative to that limit. If the system starts approaching those limits, we add another health checking cell or aggregator cell, whichever is needed.

The next reason not to worry might be the best trick in this whole article. Even when there are only a few health checks active, the health checkers send a set of results to the aggregators that is sized to the maximum. For example, if only 10 health checks are configured on a particular health checker, it's still constantly sending out a set of (for example) 10,000 results, if that's how many health checks it could ultimately support. The other 9,990 entries are dummies. However, this ensures that the network load, as well as the work the aggregators are doing, won't increase as customers configure more health checks. That's a significant source of variance … gone.

What's most important is that even if a very large number of targets start failing their health checks all at once—say, for example, as the result of an Availability Zone losing power—it won't make any difference to the health checkers or aggregators. They do what they were already doing. In fact, the overall system might do a little less work. That's because some of the redundant health checkers might themselves be in the impacted Availability Zone.

So far so good. Route 53 can check the health of targets and aggregate those health check results using a constant work pattern. But that's not very useful on its own. We need to do something with those health check results. This is where things get interesting. It would be very natural to take our health check results and to turn them into DNS changes. We could compare the latest health check status to the previous one. If a status turns unhealthy, we'd create an API request to remove any associated answers from DNS. If a status turns healthy, we'd add it back. Or to avoid adding and removing records, we could support some kind of "is active" flag that could be set or unset on demand.

If you think of Route 53 as a sort of database, this appears to make sense, but that would be a mistake. First, a single health check might be associated with many DNS answers. The same IP address might appear many times for different DNS names. When a health check fails, making a change might mean updating one record, or hundreds. Next, in the unlikely event that an Availability Zone loses power, tens of thousands of health checks might start failing, all at the same time. There could be millions of DNS changes to make. That would take a while, and it's not a good way to respond to an event like a loss of power.

The Route 53 design is different. Every few seconds, the health check aggregators send a fixed-size table of health check statuses to the Route 53 DNS servers. When the DNS servers receive it, they store the table in memory, pretty much as-is. That's a constant work pattern. Every few seconds, receive a table, store it in memory. Why does Route 53 push the data to the DNS servers, rather than pull from them? That's because there are more DNS severs than there are health check aggregators. If you want to learn more about these design choices, check out Joe Magerramov's article on putting the smaller service in control.

Next, when a Route 53 DNS server gets a DNS query, it looks up all of the potential answers for a name. Then, at query time, it cross-references these answers with the relevant health check statuses from the in-memory table. If a potential answer's status is healthy, that answer is eligible for selection. What's more, even if the first answer it tried is healthy and eligible, the server checks the other potential answers anyway. This approach ensures that even if a status changes, the DNS server is still performing the same work that it was before. There's no increase in scan or retrieval time.

I like to think that the DNS servers simply don't care how many health checks are healthy or unhealthy, or how many suddenly change status, the code performs the very same actions. There's no new mode of operation here. We didn't make a large set of changes, nor did we pull a lever that activated some kind of "Availability Zone unreachable" mode. The only difference is the answers that Route 53 chooses as results. The same memory is accessed and the same amount of computer time is spent. That makes the process extremely reliable.

Amazon Simple Storage Service (S3) as a configuration loop

Another application that demands extreme reliability is the configuration of foundational components from AWS, such as Network Load Balancers. When a customer makes a change to their Network Load Balancer, such as adding a new instance or container as a target, it is often critical and urgent. The customer might be experiencing a flash crowd and needs to add capacity quickly. Under the hood, Network Load Balancers run on AWS Hyperplane, an internal service that is embedded in the Amazon Elastic Compute Cloud (EC2) network. AWS Hyperplane could handle configuration changes by using a workflow. So, whenever a customer makes a change, the change is turned into an event and inserted into a workflow that pushes that change out to all of the AWS Hyperplane nodes that need it. They can then ingest the change.

The problem with this approach is that when there are a large number of changes all at once, the system will very likely slow down. More changes mean more work. When systems slow down, customers naturally resort to trying again, which slows the system down even further. That isn't what we want.

The solution is surprisingly simple. Rather than generate events, AWS Hyperplane integrates customer changes into a configuration file that's stored in Amazon S3. This happens right when the customer makes the change. Then, rather than respond to a workflow, AWS Hyperplane nodes fetch this configuration from Amazon S3 every few seconds. The AWS Hyperplane nodes then process and load this configuration file. This happens even if nothing has changed. Even if the configuration is completely identical to what it was the last time, the nodes process and load the latest copy anyway. Effectively, the system is always processing and loading the maximum number of configuration changes. Whether one load balancer changed or hundreds, it behaves the same.

You can probably see this coming now, but the configuration is also sized to its maximum size right from the beginning. Even when we activate a new Region and there are only a handful of Network Load Balancers active, the configuration file is still as big as it will ever be. There are dummy configuration "slots" waiting to be filled with customer configuration. However, as far the workings of AWS Hyperplane are concerned, the configuration slots there nonetheless.

Because AWS Hyperplane is a highly redundant system, there is anti-fragility in this design. If AWS Hyperplane nodes are lost, the amount of work in the system goes down, not up. There are fewer requests to Amazon S3, instead of more attempts in a workflow.

Besides being simple and robust, this approach is very cost effective. Storing a file in Amazon S3 and fetching it over and over again in a loop, even from hundreds of machines, costs far less than the engineering time and opportunity cost spent building something more complex.

Constant work and self-healing

There's another interesting property of these constant-work designs that I haven't mentioned yet. The designs tend to be naturally self-healing and will automatically correct for a variety of problems without intervention. For example, let's say a configuration file was somehow corrupted while being applied. Perhaps it was mistakenly truncated by a network problem. This problem will be corrected by the next pass. Or say a DNS server missed an update entirely. It will get the next update, without building up any kind of backlog. Since a constant work system is constantly starting from a clean slate, it's always operating in "repair everything" mode.

In contrast, a workflow type system is usually *edge-triggered*, which means that changes in configuration or state are what kick off the occurrence of workflow actions. These changes first have to be detected, and then actions often have to occur in a perfect sequence to work. The system needs complex logic to handle cases where some actions don't succeed or need to be repaired because of transient corruption. The system is also prone to the build-up of backlogs. In other words, workflows aren't naturally self-healing, you have to make them self-healing.

Design and manageability

I wrote about big-O notation earlier, and how constant work systems are usually notated as O(1). Something important to remember is that O(1) doesn't mean that a process or algorithm only uses one operation. It means that it uses a constant number of operations regardless of the size of the input. The notation should really be O(C). Both our Network Load Balancer configuration system, and our Route 53 health check system are actually doing many thousands of operations for every "tick" or "cycle" that they iterate. But those operations don't change because the health check statuses did, or because of customer configurations. That's the point. They're like coffee urns, which hold hundreds of cups of coffee at a time no matter how many customers are looking for a cup.

In the physical world, constant work patterns usually come at the cost of waste. If you brew a whole coffee urn but only get a handful of coffee drinkers, you're going to be pouring coffee down the drain. You lose the energy it took to heat the coffee urn, the energy it took to sanitize and transport the water, and the coffee grounds. Now for coffee, those costs turn out to be small and very acceptable for a café or a caterer. There may even be more waste brewing one cup at a time because some economies of scale are lost.

For most configuration systems, or a propagation system like our health checks, this issue doesn't arise. The difference in energy cost between propagating one health check result and propagating 10,000 health check results is negligible. Because a constant work pattern doesn't

need separate retries and state machines, it can even save energy in comparison to a design that uses a workflow.

At the same time, there are cases where the constant work pattern doesn't fit quite as well. If you're running a large website that requires 100 web servers at peak, you could choose to always run 100 web servers. This certainly reduces a source of variance in the system, and is in the spirit of the constant work design pattern, but it's also wasteful. For web servers, scaling elastically can be a better fit because the savings are large. It's not unusual to require half as many web servers off peak time as during the peak. Because that scaling happens day in and day out, the overall system can still experience the dynamism regularly enough to shake out problems. The savings can be enjoyed by the customer and the planet.

The value of a simple design

I've used the word "simple" several times in this article. The designs I've covered, including coffee urns, don't have a lot of moving parts. That's a kind of simplicity, but it's not what I mean. Counting moving parts can be deceptive. A unicycle has fewer moving parts than a bicycle, but it's much harder to ride. That's not simpler. A good design has to handle many stresses and faults, and over enough time "survival of the fittest" tends to eliminate designs that have too many or too few moving parts or are not practical.

When I say a *simple* design, I mean a design that is easy to understand, use, and operate. If a design makes sense to a team that had nothing to do with its inception, that's a good sign. At AWS, we've re-used the constant work design pattern many times. You might be surprised how many configuration systems can be as simple as "apply a full configuration each time in a loop."