The Scale to Zero Problem

Matt Butcher

APRIL 04, 2022

Cloud is expensive. One reason for this is that a typical application instance sits idly waiting for inbound requests. Yet even though it is idle, it is incurring cloud charges. What if there was a way to scale down your application when it was not being used? What if it could be scaled to the point where no instance were even running? This is the idea behind “scale-to-zero” designs. In the Kubernetes ecosystem, a number of projects attempted to implement scale-to-zero. When we built our WebAssembly project, we built it to scale to zero by default.

Microservices, Statelessness, and Scaling

A central tenant of the microservice architecture is

GitHub

Twitter

RSS Feed

that over time, no service should rely on information stored in-memory.

Several years ago, monolithic applications were designed differently. Developers assumed (and rightly so) that the application itself would run for days, months, or even years before being interrupted. It was safe to store some critical pieces of information in memory and rely upon it as the authoritative copy.

The container-oriented microservice architecture upended this design principle. Instead of single instances of monolithic applications, the microservice architecture has championed using multiple disposable instances of many small services. Together, these many small services form a single application. Central to this architecture is the notion of statelessness: No single instance of a service should contain or be treated as an authoritative source of state information. That information should all be written to a data storage facility somewhere.

Because no individual microservice instance held state information, instances could be created and destroyed with regularity. As an application got more popular, more instances could be added to handle the load. And when popularity decreased, the application could be scaled back down. Excess instances would be terminated. This model saved system resources and ideally cut down on cloud cost as well.

COMMUNITY POSTS

Why the Bytecode Alliance is important to the Wasm ecosystem

Come join the Fermyon Discord community!

Wasm.Builders: The Site for Learning WebAssembly

But there was a natural lower bound to scaling. The general consensus was that every service should have at least two or three instances running at a given time. Scaling down any further would mean that when a user requested the application, it would not be there. It was prudent, therefore, to keep at least a couple of instances running. (Scaling down to one was considered dangerous, because if anything went wrong with that one instance, it could take quite a while to bring a replacement online.)

Keeping around two-to-three copies of every microservice in an application, though, can be expensive. Even if an instance is doing nothing, it still consumes memory and CPU. And in the public cloud that translates directly to billable compute time. This cost is relevant to both small companies that need to keep close leash on their cloud spend and also very large applications where ten or more microservices, each running two to three replicas, can start to inflate the cloud bill.

In the Kubernetes world, several different attempts were made to solve this problem, all under the name "scale-to-zero"

Scale-to-Zero

When load drops to zero, when there are no inbound requests for an application, the typical microservice

architecture would still keep a few copies of each microservice running.

In the scale-to-zero model, instead of keeping a couple copies of each microservice running, a piece of software is inserted between inbound requests and the microservice. This piece of software is responsible for tracking (and predicting) traffic and managing the number of microservice instances accordingly. Through one algorithm or another, it should be able to scale the microservice instances all the way down to the point where no instances are running, but then have the microservices ready to go when new requests come in.

A common misconception about Functions-as-a-Service tools, such as Lambda or Azure Functions, is that they function as scale-to-zero services. This is not quite accurate. FaaS services tend to keep “pre-warmed instances” available for execution, and just perform last-second bootstrapping to take those pre-warmed instances and execute the user’s function. This model is often reflected in FaaS pricing. Users share the cost of the pre-warmed instances.

While a number of scale-to-zero extensions have been written for Kubernetes, each comes with substantial trade-offs, and some even require that microservices be written in a particular way to work with these services. In our view, the reason such tools have not taken off is that they are difficult, unpredictable, and often introduce a new point of failure.

If we take a step back, though, we can see that the problem is not rooted in the service offerings. It's rooted in a fundamental design characteristic of container-based microservices.

Containers, Long-lived Services, and Startup Time

Containers provide an environment in which programs can be run. And they provide tools for connecting networks and file systems and so on. But at the end of the day, every web service that runs in a container must have the following features:

- It must act as a long-running process that stays active
- It must act as a server that can receive many requests and handle each one

To that end, a developer who writes a microservice starts by creating a web server.

When the container executes, it wires up storage, networking, and other facilities, then starts the process in the container, and runs until that process exits. The process is, in all senses of the word, a server. Even if that server is stateless, it must be available and listening for inbound connections.

Servers take a while to start. And this problem is amplified when the environment in which the server runs also has to be set up. Before a container can start, its sandbox must be constructed, its filesystem fetched and loaded, and any additional services must be started. On average, this takes around 12 seconds. Even highly optimized containers still take over a second. When the expectation is that a web request will be processed in under 100 milliseconds, no scale-to-zero solution can include a one second startup time.

Maybe an assumption about microservices can be adjusted, and by doing that, we could achieve the performance profile necessary to scale to zero.

WebAssembly Can Scale to Zero

If you are reading this on the Fermyon.com website, you are already experiencing scale-to-zero in WebAssembly. All of Fermyon.com is served by a scale-to-zero framework for WebAssembly. We only run a WebAssembly module when we are handling requests.

And the way we achieve this is to drop the assumption that *every microservice must run its own HTTP server*.

Fermyon.com has a long-running Web server that listens for incoming requests, and then loads the appropriate WebAssembly module needed for that request. That module handles the request and then shuts down. So our WebAssembly modules are only running when they are working. The memory, CPU, and other resources that are allocated to the module are freed and can be assigned to other instances as they handle their requests. This is an important part of what we call microservices v2.

Take this website as an example. Fermyon.com is powered by three different WebAssembly modules. One is a static file server that serves out images, stylesheets, and other assets. Another is a simple Go service for serving favicons. The third is Fermyon's new Content Management System (CMS) called Bartholomew. It loads content and generates the appropriate HTML.

Because WebAssembly is quick to load and execute, Fermyon.com can typically start up an instance of the CMS, handle the request, and shut the instance back down in around 35ms. And even better, since the HTTP server, TLS, and all of that is handled by the framework, when we wrote our CMS we didn't have to start by setting up a webserver. We just wrote the

HTTP request/response handler. So not only is it faster to execute, but it is much easier for us to write and maintain.

> A fresh page load of Fermyon.com may result in starting, executing, and shuting down 30+ concurrent WebAssembly processes. Yet the result is a page that loads so fast that it scores a 99% on Google's page speed ranking.

Behind the scenes at Fermyon, we opted not to operate a costly Kubernetes cluster. Instead, our solution runs on a Nomad cluster. Each worker in our cluster runs just a single HTTP listener (a Spin instance) that then handles requests to all our modules. A single Traefik proxy sits at the ingress to our cluster and balances the load across our workers. But at any given time, there are only as many WebAssembly modules running as there are requests coming in. If one request comes in at a time, only one module will be executed. If 200 requests occur concurrently, 200 instances will be run. Should we hit a moment where there are no requests across any services on our cluster, there will be no WebAssembly executing. Just the HTTP listeners ready to start up new modules on demand.

Because we don't need our WebAssembly modules to run for long periods of time, we can achieve much higher density in our services. That is, we could run more WebAssembly modules per compute unit. (See our earlier post comparing WebAssembly to Docker Containers.) However, we opted to take the opposite approach, which has been a big cost-saver: We run our workers on cheap compute units. For us, our cluster runs on three `t2.small` instances of AWS compute. Comparing this to a baseline install of a Kubernetes cluster, it is clear that we are immediately saving quite a bit of money. (We're working on a cost comparison post that we'll publish in the future.)

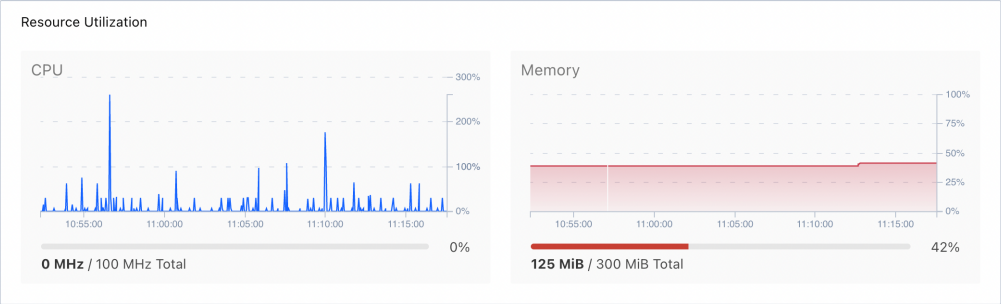
Scale-to-Zero In Action

Not long ago, we had a very busy day. Our traffic jumped to ten times its normal number of page requests. At a few moments, we were seeing over 2,500 requests coming in at a time (90 concurrent users each fetching around 30 resources).

In our model, we run three workers, each on a small-sized VM with 3 CPUs and 300Mb of memory per worker. As the name indicates, these specs are modest for cloud computing. When Spin and Wagi first start up, they pre-parse the WebAssembly modules and keep them cached in memory to reduce the startup time.

When a request is received, the WebAsembly module is loaded from the in-memory cache, executed, and then shut down and cleaned up. Given this behavior, what we should see is “spiky” CPU usage, where each request contributes to a spike.

Here’s what our Spin documents site looked like during its busiest day:



On the left, you can see the jagged shape of the CPU utilization, each spike representing a group of requests coming in. Even under load an order of magnitude higher than we usually see, CPU never maxes out (and consequently, we don’t believe our site visitors ever saw any CPU-based latency). Meanwhile, the memory graph on the left shows that a fairly stable 125M of memory was used throughout the server’s lifecycle. While we’re sure that if we really zoomed in, we’d see small spikes matching the CPU graph, the execution of the WebAssembly modules simply does not consume enough memory to surface these differences on the chart at this level.

In the course of writing this article we discovered an issue where we stored duplicate copies of some data. Consequently, we believe the memory consumption should be around 40Mb less than what is shown here.

The primary point here is that the WebAssembly system in Spin and Wagi is efficient enough that we can squeeze tremendous performance out of VMs that, for example, are too small to run even an empty Kubernetes node.

Conclusion

Scale-to-zero is a strategy that Kubernetes devops have been pursuing for a long time. And with good reason! Success means money saved. But when we started building the Fermyon Platform, we added this ability from the beginning, which means not only cost savings, but an improved developer experience, and a long-term easier-to-operate platform. We routinely hear that Kubernetes is too hard to operate, and scale-to-zero is but one reason why. Our goal is to build a platform that is easy for developers and operators alike. And if it so happens to save you money along the way, all the better!