corebytes
a fintech blog from coremont

coremont

Scalable Compute

By Tim Roberts, Software Engineer

March 2024

Scalable Compute

"Hi can I please borrow 20,000 CPU with 30 TB memory?"

"Sure, when do you need them?"

"Less than one second from now please."

"How long would you like them for?"

"Five seconds and I am only paying for five seconds of use."

"No problem."

Public cloud creates some options that would have seemed outlandish not long ago. However, when conversations like the one above are possible, they can also create subsequent obstacles. This is a case study of how Coremont tackled those challenges to deliver the most efficient use of compute resource.

Our demand for compute resource is driven by user interaction that results in an embarrassingly parallel CPU-bound workload: somewhat predictable use but with large unpredictable spikes.

To meet this demand, the options considered were:

- Run idle compute to ensure ample capacity. Computationally great. Financially dreadful.
- Run requests in batches or queues to maximize utilisation. No wasted resource. Avoids scale-up time to retrieve and prepare anything common between requests e.g. the app retrieved data. But the results will be out of date (therefore worthless) before you even begin.
- Autoscale virtual machines based on load. If the entire request aims to take under five seconds then the delay of starting a new VM (let alone bootstrapping the orchestration, runtime, app) makes this untenable.
- Function-as-a-service. Sounds good tell me more.

What we wanted was some combination of the above:

- Enough compute available to never have to wait.
- A non-eye-watering bill.
- Optimisation to reduce cold start delays from pulling platform, runtime, our app, starting process, loading unchanged data etc.

AWS Lambda allowed us to achieve this but we faced a few challenges in implementation.

Cannot select the underlying hardware.

Public cloud providers offer a wide range of virtual machine families/types/series/generations with specific underlying hardware depending on the desired hardware vs. cost.

AWS Lambda allows requesting vCPU and GB memory as required but not the underlying hardware. This created two problems:

1. **Cannot benchmark our workload across available options and select the fastest.**

   Our function executed approximately 50% slower on Lambda than on Amazon's "Compute Optimized" EC2 instances.

   To counter this, we could now make use of the scalability of Lambda. Make our "embarrassingly parallel" request run even more parallel. Our new parallelism limit is the AWS Lambda account concurrency limit which is much higher than what was available before moving to Lambda.

2. **Do not know what hardware our function will execute on.**

   This makes bin packing difficult since the size of the bin may vary.

   For example, in the few weeks our function executed on:

CPU	% of invocations
Intel(R) Xeon(R) Processor @ 2.50GHz	92.95%
Intel(R) Xeon(R) Processor @ 3.00GHz	4.46%
Intel(R) Xeon(R) Platinum 8252C CPU @ 3.80GHz	1.48%
Intel(R) Xeon(R) Processor @ 2.90GHz	1.10%

This CPU variance can make our estimated calculation time less accurate. The user must wait for the slowest request so we do not want to risk one invocation delaying the entire response.

Our solution was to build our bin packing estimates from the most commonly hit CPU. We capture the CPU used on Lambda (using the X86Base.Cpuid(Int32, Int32) Method), so we can factor this in to our bin packing.

Lambda executes our function on the slower CPU the majority of the time. By basing our estimates on this it means the ~7% of the requests that run on faster CPUs are an acceptable inefficiency, and do not affect the user response time.

Connecting to VPC resources can cause network bottlenecks at scale.

Our Lambda function must access other private services contained within our VPC. This is supported by AWS but we hit some unexpected issues with the Lambda Hyperplane ENIs.

Hyperplane Elastic Network Interfaces (ENI) are part of the managed network your traffic requires to go from your function in Lambda's VPC to your own VPC.

Our Lambda function reads approximately 20MB of data from S3 on a cold start (first invocation). The 20MB is split across approximately 100 keys read in parallel. In a non-VPC Lambda function this will hit S3's wide and mighty public surface. Reading 20MB over 100 keys from several thousand parallel Lambda invocations wouldn't even register as a blip for S3. However, in a VPC Lambda function this S3 traffic all goes via Hyperplane ENIs. (specifically: our S3 request in a VPC Lambda function goes via a hyperplane ENI to our VPC. Then to our S3 VPC endpoint. Then to S3.)

When invoking our max number of parallel requests, we would experience S3 timeouts on some invocations. After some AWS VPC Flow Logs analysis, we determined that the Hyperplane ENIs were the bottleneck.

Hyperplane ENIs do autoscale once 65,000 connections are reached. However, the connections must be sustained for some time (minutes, not seconds). This meant Hyperplane ENI autoscaling was not fast enough to deal with our aggressive S3 read during a spike.

We needed more Hyperplane ENIs.

Our solution was to take advantage of the logic that determines when a Hyperplane ENI is created.

The documentation states "Lambda creates a Hyperplane ENI when you define a unique subnet plus security group combination for a VPC-enabled function in an account"

We tried splitting our VPC into many smaller subnets and configuring our function to use them all. This successfully created more Hyperplane ENIs which reduced the number of S3 timeouts. However, we soon hit the max number of subnets (16) that can be allocated to a Lambda function.

Our additional workaround was to create multiple copies of the same Lambda function and allocate each of them a subset of our large array of subnets.

After experimentation we ended up with:

- our-function-1 (uses subnets 1,2,3,4,5,6)
- our-function-2 (uses subnets 7,8,9,10,11,12)
- ... etc.

And we changed our code that invoked Lambda to simply round-robin which function it requests.

This created the additional Hyperplane ENIs we needed and fixed our Hyperplane ENI bottleneck completely. Turned out what we had was a "Hyperplane ENIs per Lambda function" problem.

With the problems above all resolved we have been able to take full advantage of the scale that AWS Lambda offers.

About Coremont

Coremont provides market-leading portfolio management technology and data across all investment management and trading functions, including risk and quantitative analysis, operations, treasury, accounting and regulatory reporting. Clients include hedge fund managers and institutional asset managers, spanning four continents.

The cornerstone of Coremont's offering is Clarion, cloud-based portfolio management technology, which is powered by Coremont's proprietary pricing models and analytics. Available through UI or API, Clarion helps clients navigate the daily demands of trading and risk management, with advanced capabilities in derivatives.

Coremont LLP was spun out from Brevan Howard in early 2018, with offices in London, Connecticut and Hong Kong. For more information, visit Coremont's website at **www.coremont.com**

If you have any questions or would like more information about Coremont and our flagship portfolio management system, Clarion, please email us at enquiries@coremont.com