


Going serverless: How we migrated our customer websites to AWS Lambda

 blog.aiir.com/going-serverless-how-we-migrated-our-customer-websites-to-aws-lambda-387f169804d0

Today we're pleased to announce we've open sourced our PHP Lambda Layer, which will allow you to migrate a PHP website in to an AWS Lambda function easily. In this technical post we'll explore how and why we moved to this approach for all our customer websites.

Over the last couple of years the buzz around 'serverless' has grown at an impressive pace. Here at Aiir we've been increasingly using AWS' Lambda service to build serverless "microservices"—small highly focussed functions that typically respond to an incoming event, and are chained together to perform business functions. One of the key benefits of deploying a task as a Lambda function is that the entire lifecycle, including scaling, is managed by the service.

The Problem

Lambda functions have worked well for us so far. The added peace of mind of not having to peer in to our existing stack and think about where something should be placed or how we're going to build new VMs to accommodate an application has been a welcome relief. It's helped our small team put more focus on developing our products.

Meanwhile our Front End stack (the code and servers that run customer facing websites and mobile app APIs) have remained on a more traditional virtual server stack. This has worked well for us for many years, but with one infrequent headache; sudden traffic peaks and the constant review and refinement process around auto scaling.

We frequently found ourselves between two key issues, which are just as familiar to more traditional on-premise setups—make your rules too sensitive and you'll see random scale ups at inappropriate times that burn money, relax them too much and you'll react too slowly to spikes. Things got more complicated when a major release of our front end

code's runtime massively improved CPU performance but caused memory usage to become our new virtual instance bottleneck, a metric that's not trackable with standard EC2 auto scaling rules.

We ended up in a situation where even with all the tools at our disposal to autoscale our front end environment, it was typically reacting too slowly to increase load—taking anywhere up to five minutes to increase capacity enough to return operation to normal. But we knew this, and mitigated it by keeping a close eye on obvious traffic indicators (weather forecasts, wobbly health of prominent pop/rock stars...) and pre-empted these with preventative scale ups.

Creating a solution

During our last (successfully managed!) traffic event, an idea struck—could we bundle our front end application in to a Lambda function? The way functions are built and scaled is entirely different to the way virtual instances operate and, crucially, is a managed solution. But what would be involved?

At the time we first began exploring this the list of supported runtimes did not include PHP; the language our front end is written in. This didn't feel like the right time to propose a complete rewrite of one of our core system components (hell, when is?) so instead we looked at alternative options.

Behind the scenes, Lambda functions run inside containers. You can take a binary compiled against Amazon Linux and drop it in to your Lambda function distribution zip file and execute it without too much issue. This led to the idea of building a slimline handler function in one of the already supported languages and including a PHP executable. AWS API Gateway provided the necessary HTTP event to trigger the function, and with a custom domain parked on a stage you could successfully serve a website from a Lambda function. We quickly got a proof of concept working with this approach, and even [open sourced our initial Node.js-based PHP execution handler solution](#).

This approach worked reasonably well, the most significant issued we faced was the requirement to include a build of PHP in our codebase distribution every time we wished to roll an update. It was possible with some careful release pipeline tooling, but not the most elegant of solutions.

Then AWS:reInvent 2018 happened.

During the conference AWS announced two new features that could dramatically simplify our previous approach—layers and custom runtimes from Lambda. Essentially, these allowed us to split the runtime from the actual codebase in a layered fashion and, crucially, not introduce another language such as Node.js in to the mix. A layer in your stack could include an executable file, or ‘bootstrap’, to grab an incoming request, perform a task, and return a response.

We got off the ground with layers and custom runtimes quickly thanks to the excellent initial work of the team at Stackery and their example PHP layer. This easily demonstrated how to create a PHP-based bootstrap runtime layer, including the executable and PHP extensions, and build it in a way that the actual function distribution can be as simple as your PHP code, dependencies and ini file.

In testing this layer we hit some issues with its approach. Internally the Stackery version of the layer was using PHP’s internal development server to execute the code in a web server-like environment. Whilst it was functional, it hit issues with not being exactly the same as a typical PHP FPM environment, which caused issues with how some popular libraries handling routing and other common HTTP tasks.

Creating a solution (again, part 2)

We looked at how this could be resolved and revisited some of the findings from the previous Node.js implementation. By creating a variant of the runtime that used *php-cgi*, the executable used inside of PHP FPM, we could create an environment that directly mirrored that of our previous NGINX-based FPM stack. We soon had a working version of this approach and have continued to develop and refine this version.

Another important announcement from re:Invent was the ability for an Application Load Balancer (ALB) to directly invoke a Lambda function. This allowed us to replace API Gateway, which realistically provided little value to our stack as our application handles routing directly. Because ALB can inspect the incoming HTTP request (i.e. the A in ALB), you can vary which target is used to serve a request based on host. This provided an easy way to vary our production and staging environments by domain, without having to have a second ALB setup.

Our production and staging environment stem from the same configured Lambda function, making use of the built-in versioning and aliasing. Staging always follows the \$LATEST version, whereas production updates trigger a new version to be created and the Production alias is updated to point to it. We've now built this in to our release pipeline and all environments can be triggered via commits to our codebase's Git repo.

Finally, at the end of January we moved all of our customer website hosting to Lambda. So far we're extremely pleased with the move from instances to a serverless functions. The recent changes introduced by AWS made this much more viable and elegant. Seeing Lambda automatically scale up and down in a matter of seconds is a huge improvement for us over our previous setup where we often had to preempt certain spikes and always run with a costly additional overhead to ensure the reliability we strive to deliver.

We're always looking at ways to improve our product offering, both in terms of new features and in terms of refining the infrastructure and tooling we're using to deliver our products. This project is a great example of seizing a new technology to help us with those goals. If you run a PHP stack we'd love for you to take a look at our PHP Lambda Layer and get in touch or get involved with issues and pull requests.