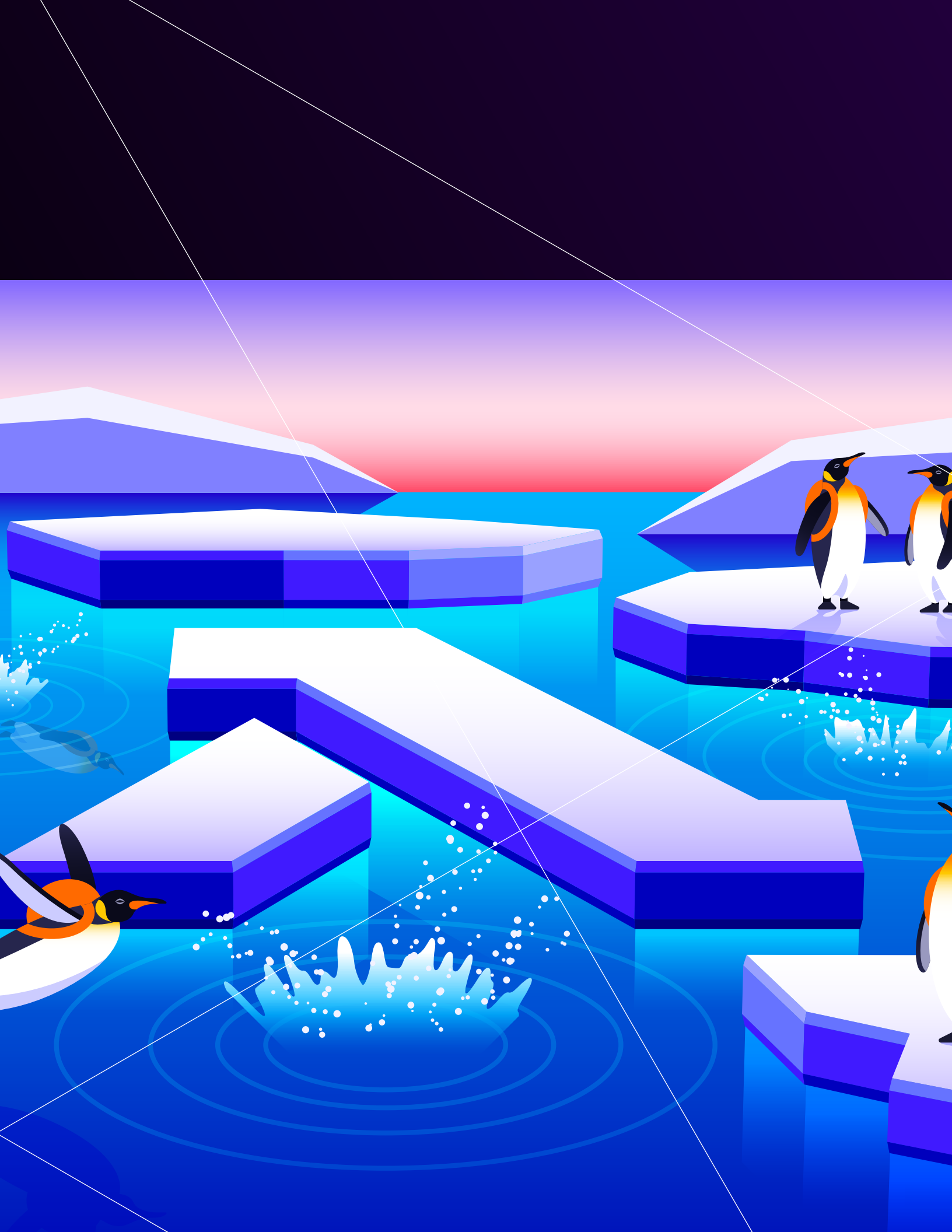
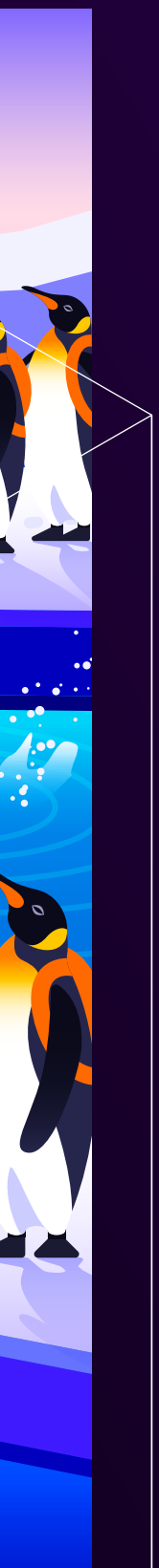
Cloud-scale monitoring with AWS and Datadog

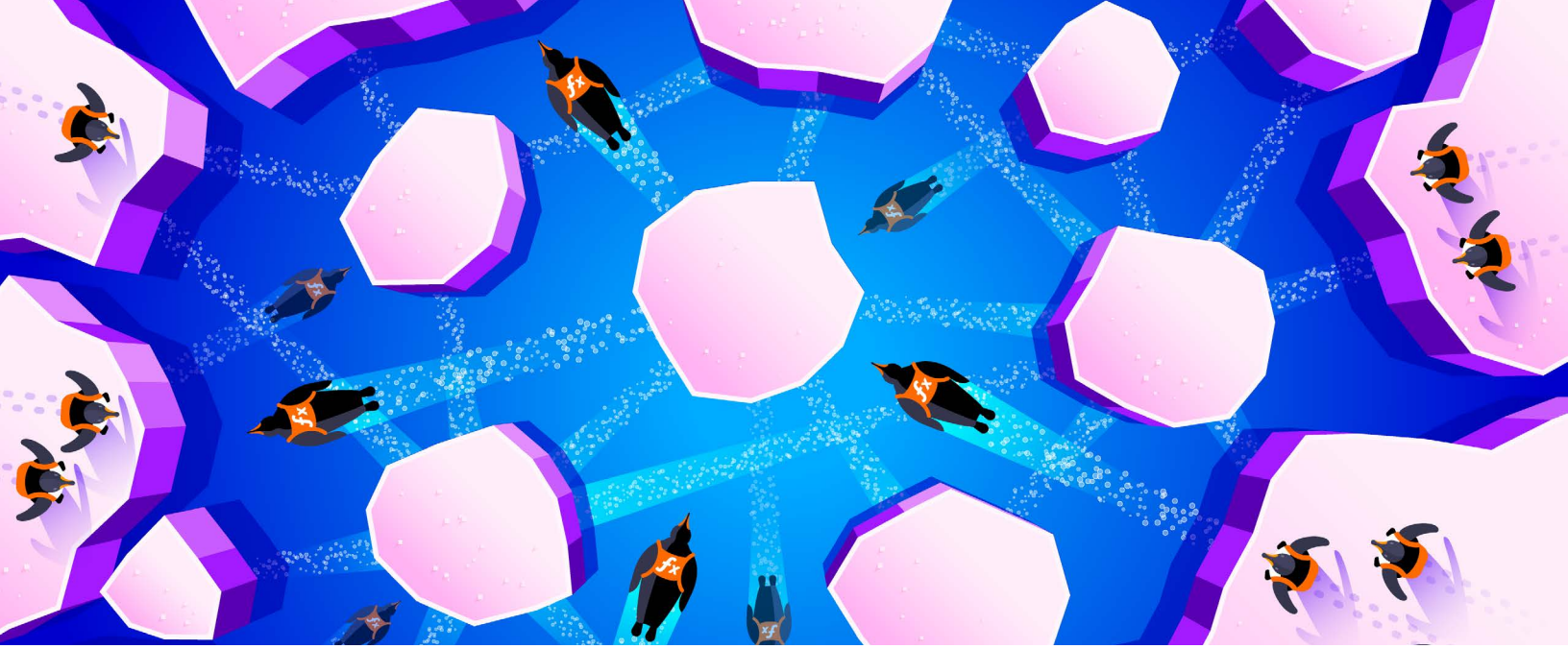
Serverless applications in AWS

aws | DATADOG



Serverless applications in AWS





Monitoring serverless applications in AWS

Adopting serverless architecture enables you to shift the responsibility of traditional application operations, such as provisioning and managing servers, from your team to the cloud. The cloud provider becomes responsible for managing infrastructure resources and using them to deploy serverless code as needed. Leveraging serverless for your applications is cost effective as you no longer need to pay for or manage always-on application resources (e.g., server capacity, network, security patches); you are only charged for the resources you use for your functions. Serverless architectures can also easily scale to meet demand.

Running workloads using serverless architectures introduces new challenges for monitoring. Serverless applications are fundamentally different from ones running on traditional hardware. For one, you won't be able to collect typical system metrics. But it is still critical to collect observability data in order to monitor the performance of your serverless functions. For example, your AWS Lambda functions are controlled by concurrency limits and allocated memory. If you exceed these limits, your functions will timeout or be killed by the runtime. We'll walk through some of these scenarios as well as some key metrics for monitoring AWS's platform of serverless tools, including:

- AWS Lambda
- AWS Fargate
- Amazon API Gateway
- AWS Step Functions

We'll also look at how Datadog enables you to monitor your entire serverless architecture in one place and provides deeper insights into the performance of AWS's serverless tools.

AWS Lambda

AWS Lambda is a compute service at the center of the AWS serverless platform that deploys your serverless code as **functions**. Functions are event driven and can be triggered by events such as message queues, file uploads, HTTP requests, and cron jobs. You can trigger Lambda functions using events from other AWS services, such as API calls from Amazon API Gateway or changes to a DynamoDB table. When a service invokes a function for the first time, it initializes the function's runtime and **handler** method. AWS Lambda supports a range of runtimes, so you can write functions in the programming language of your choice (e.g., Go, Python, Node.js) and execute them within the same environment.

HOW TO MONITOR AWS LAMBDA

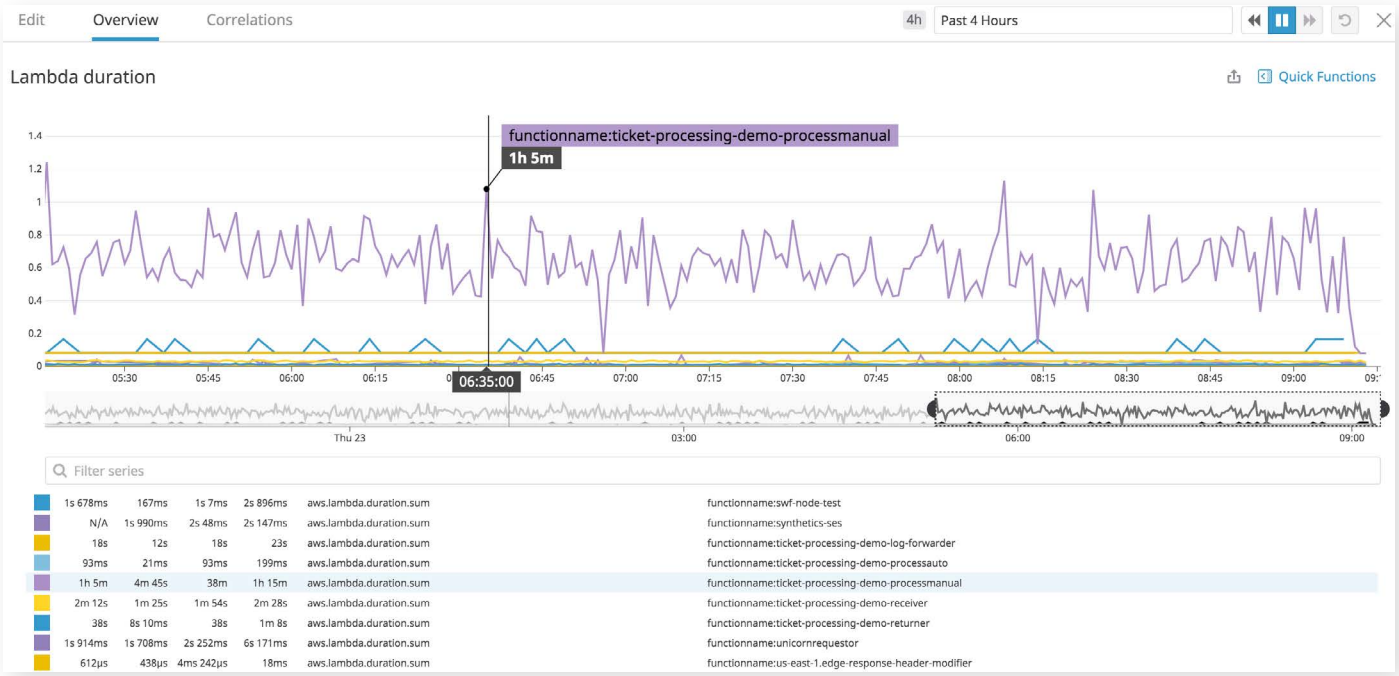
Since AWS Lambda manages infrastructure resources for you, you won't be able to capture typical system metrics such as CPU usage. Instead, Lambda reports the performance and efficiency of your functions as it runs them, so monitoring your functions involves tracking function utilization, invocations, and concurrency (including provisioned concurrency).

KEY FUNCTION PERFORMANCE AND UTILIZATION METRICS

Lambda automatically tracks the amount of time that a function is in use (performance) and how much memory a function uses during an invocation (utilization). Monitoring this data can help you optimize your functions and manage costs. Function utilization metrics are included in your CloudWatch logs, as seen in the example log below:

```
REPORT RequestId: f1d3fc9a-4875-4c34-b280-a5fae40abcf9    Duration: 72.51 ms    Billed Duration: 100 ms    Memory Size: 128 MB Max Memory Used: 58 MB Init Duration: 2.04 ms
```

Duration and billed duration: Monitoring a function's execution time (i.e., it's duration), can help you determine which functions can (or should) be optimized. Slow code execution could be the result of cold starts—an initial delay in response time for an inactive Lambda function—overly complex code, or network latency if your function relies on third-party or other AWS services. Lambda limits a function's total execution time to 15 minutes before it will terminate it and throw a timeout error, so monitoring duration helps you see when you are about to reach this threshold.



The *billed* duration measures the execution time rounded up to the nearest 100 ms. Billed duration is the basis for AWS's Lambda pricing, along with the function's memory size, which we will talk about next.

You can compare a function's duration with its billed duration to see if you can decrease execution time and lower costs. For instance, let's look at this function's log:

```
REPORT RequestId: f1d3fc9a-4875-4c34-b280-a5fae40abcf9    Duration: 102.25
ms    Billed Duration: 200 ms   Memory Size: 128 MB Max Memory Used: 120
MB    Init Duration: 2.04 ms
```

The function's duration was 102 ms, but what you will pay for is based on the 200 ms billed duration. If you notice the duration is consistent (e.g., around 102 ms), you may be able to add more memory in order to decrease the duration and the billed duration. For example, if you increase your function's memory from 128 MB to 192 MB and the duration drops to 98 ms, your billed duration would then be 100 ms. This means you would be charged less because you are in the 100 ms block instead of the 200 ms block for billed duration. Though we used a simple example, monitoring these two metrics is important for understanding the costs of your functions, especially if you are managing large volumes of requests across hundreds of functions.

Memory size and max memory used: A function's duration and billed duration are partially affected by how much memory it has—slower execution times may be a result of not having enough memory to process requests. Or, you may allocate more memory than your function needs. Both scenarios affect costs, so tracking memory usage can help you strike a balance between processing power and execution time. You can allot memory for your function, which Lambda logs refer to as its memory size, within AWS Lambda quotas.

You can compare a function's memory usage with its allocated memory in your CloudWatch logs, as seen below:

```
REPORT RequestId: f1d3fc9a-4875-4c34-b280-a5fae40abcf9    Duration: 102.25
ms    Billed Duration: 200 ms   Memory Size: 512 MB Max Memory Used: 58 MB
Init Duration: 2.04 ms
```

You can see that the function uses (`Max Memory Used`) only a fraction of its allocated memory. If this happens consistently, you may want to adjust the function's memory size to reduce costs. On the other hand, if a function's memory usage is consistently reaching its memory size then it doesn't have enough memory to process incoming requests, increasing execution times.

KEY FUNCTION INVOCATION METRICS

You can invoke a Lambda function in one of three ways: synchronously, asynchronously, or via an event source mapping.

Synchronous services create the event, which Lambda passes directly to a function and waits for the function to return a response before passing the result back to the service. This is useful if you need the results of a function before moving on to the next step in the application workflow. If an error occurs, the AWS service that originally sent Lambda the event will retry the invocation.

Asynchronous invocation means that, when a service invokes a function, it immediately hands off the invocation event for Lambda to add to a queue. As soon as the service receives a response that the event was added to the queue successfully, it moves on to the next request. Asynchronous invocations can help decrease wait times for a service because it doesn't need to wait for Lambda to finish processing a request If a function returns an error—as a result of a timeout or an issue in the function code—Lambda will retry processing the event up to two times before discarding it. Lambda may also return events to the queue—and throw an error—if the function doesn't have enough concurrency to process them.

You can also use event source mapping to link an event source, such as Amazon Kinesis or DynamoDB streams, to a Lambda function. Mappings are resources that configure data streams or queues to serve as a trigger for a Lambda function. For example, when you map a Kinesis data stream to a function, the Lambda runtime will read batches of events, or records, from the shards (i.e., the sequences of records) in a stream and then send those batches to the function for processing. By default, if a function returns an error and cannot process a batch, it will retry the batch until it is successful or the records in the batch expire (for a data stream). To ensure a function doesn't stall when processing records, you can configure the number of retry attempts, the maximum age of a record in a batch, and the size of a batch when you create the event source mapping.

Each of these invocation methods results in some different metrics to monitor in addition to standard metrics that apply to all invocation types, such as invocation count and iterator age.

Invocations: Monitoring invocations can help you understand application activity and how your functions are performing overall. Anomalous changes in invocation counts could indicate either an issue with a function's code or a connected AWS service. For example, an outage for a function's downstream service could force multiple retries, increasing the function's invocation count.

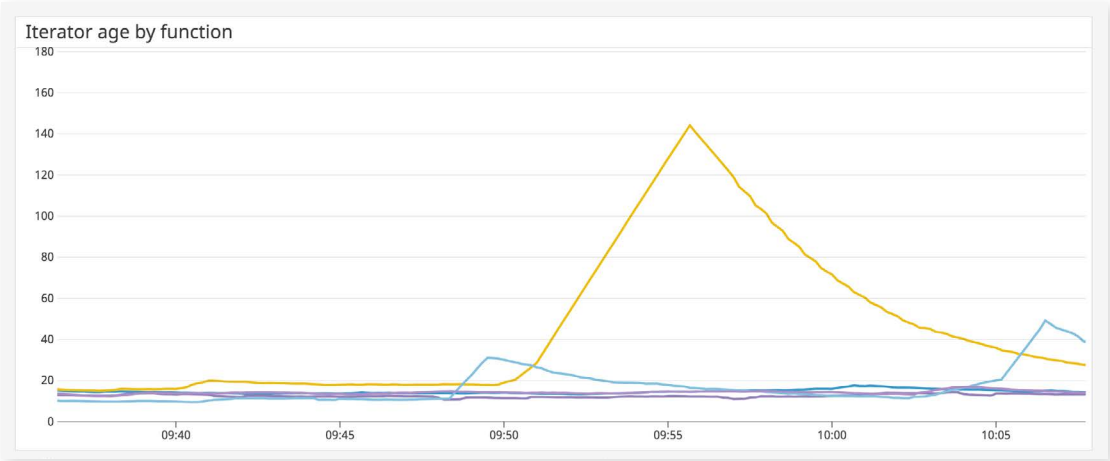
Additionally, if your functions are located in multiple regions, you can use the invocation count to determine if functions are running efficiently, with minimal latency. For example, you can quickly see which functions are invoked most frequently in which region and evaluate if you need to move them to another region or availability zone or modify load balancing in order to improve latency. Services like Lambda@Edge can improve latency by automatically running your code in regions that are closer to your customers.

Iterator age: Lambda emits the iterator age metric for stream-based invocations. The iterator age is the time between when the last record in a batch was written to a stream (e.g., Kinesis, DynamoDB) and when Lambda received the batch, letting you know if the amount of data that is being written to a stream is too much for a function to accept for processing.

There are a few scenarios that could increase the iterator age:

- a high execution duration for a function
- not enough shards in a stream
- invocation errors
- insufficient batch size

If you see the iterator age increase, it could mean the function is taking too long to process a batch of data and your application is building a large backlog of unprocessed events.



To decrease the iterator age, you need to decrease the time it takes for a Lambda function to process records in a batch. Long durations could result from not having enough memory for the function to operate efficiently. You can allocate more memory to the function or find ways to optimize your function code.

Adjusting a stream's batch size, which determines the maximum number of records that can be batched for processing, can also help decrease the iterator age in some cases. If a batch consists of mostly calls to simply trigger another downstream service, increasing the batch size allows functions to process more records in a single invocation, increasing throughput. However, if a batch contains records that require additional processing, then you may need to reduce the batch size to avoid stalled shards.

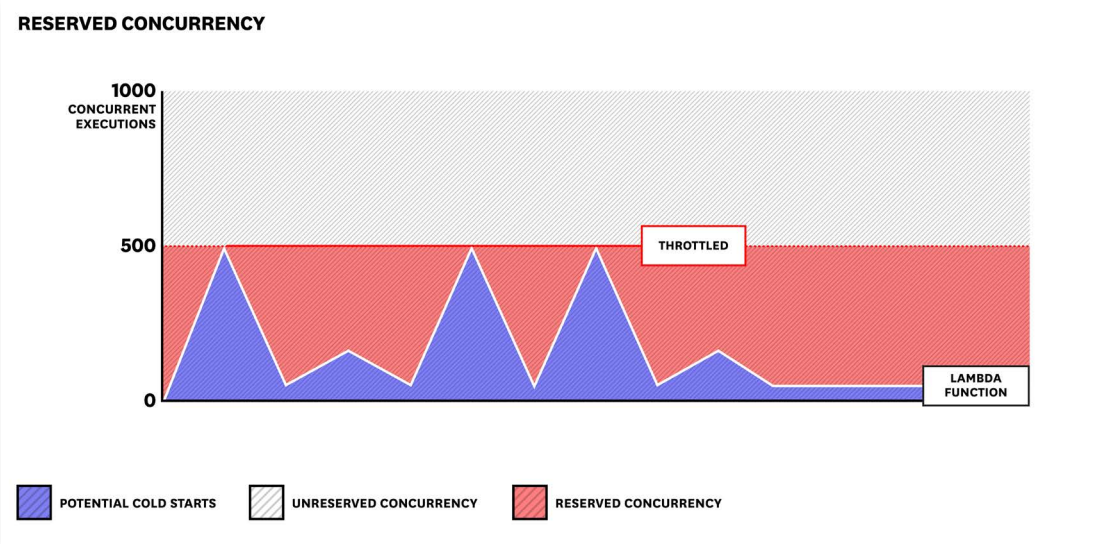
Another key component for monitoring a function's iterator age is tracking invocation errors—which you can view in Lambda's logs. Invocation errors can affect the time it takes for a function to process an event. If a batch of records consistently generates an error, the function cannot continue to the next batch, which increases the iterator age. Invocation errors may indicate issues with a stream accessing the function (e.g., incorrect permissions) or exceeding Lambda's concurrent execution limit.

MONITORING FUNCTION CONCURRENCY

A function's concurrency is a measure of how many invocations that function can handle at one time. When a service invokes a function for the first time, the Lambda runtime creates a new instance of the function to process an event. If the service invokes a function while it is still processing an event, Lambda creates another instance. This cycle continues until there are enough function instances to serve incoming requests, or a function reaches its concurrency limit and is throttled.

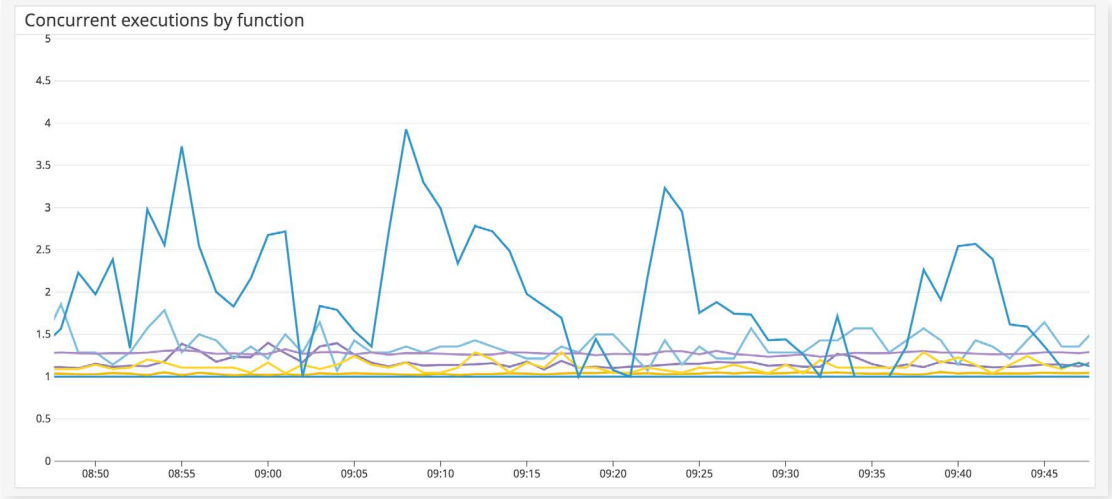
By default, Lambda provides an initial pool of 1,000 concurrent executions per region, which are shared by all of your functions in that region. You can increase the per-region limit by submitting a request to AWS support. Lambda also requires the per-region concurrency pool to always have at least 100 available concurrent executions for all of your functions at all times. Monitoring concurrency along with a function's invocation count can help you manage overprovisioned functions and scale your functions to support the flow of application traffic. A burst of new invocations, for instance, could throttle your function if it does not have enough concurrency to process incoming traffic.

Lambda automatically scales function instances based on the number of incoming requests, though there is a limit on how many instances can be created during an initial burst. Once that limit is reached (between 500 and 3,000 instances, depending on your region), functions will scale at a rate of 500 instances per minute until they exhaust all available concurrency. This can come from the per-region concurrent executions limit or a function's **reserved concurrency**, which is the portion of the available pool of concurrent executions that you allocate to one or more functions. You can configure reserved concurrency to ensure that functions have enough concurrency to scale—or that they don't scale out of control and hog the concurrency pool.



Reserving concurrency for a function is useful if you know that function regularly requires more concurrency than others. You can also reserve concurrency to ensure that a function doesn't process too many requests and overwhelm a downstream service. Note that if a function uses all of its reserved concurrency, it will not access additional concurrency from the unreserved pool. Make sure you only reserve concurrency for your function(s) if it does not affect the performance of your other functions, as doing so will reduce the size of the available concurrency pool.

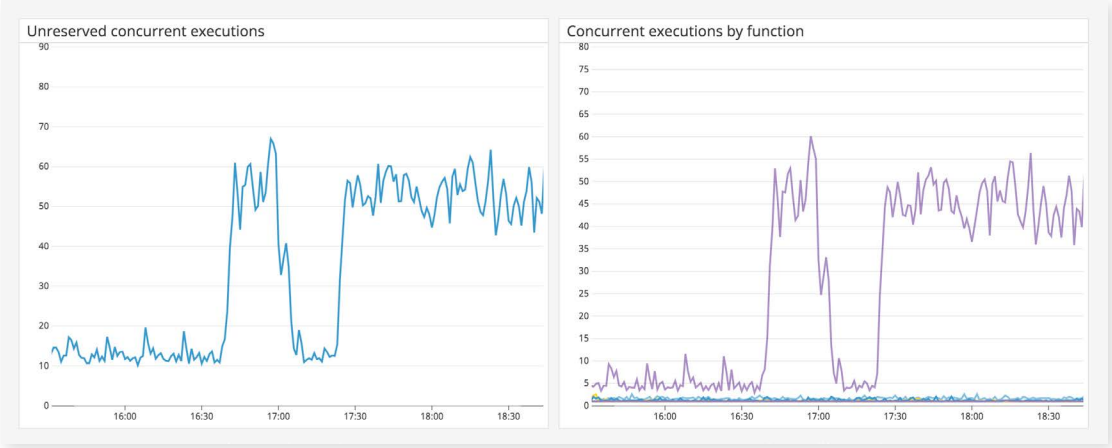
Concurrent executions: In order to monitor concurrency, Lambda emits the concurrent executions metric. This metric allows you to track when functions are using up all of the concurrency in the pool.



In the example above, you can see a spike in executions for a specific function. As mentioned previously, you can limit concurrent executions for a function by reserving concurrency from the common execution pool. This can be useful if you need to ensure that a function doesn't process too many requests simultaneously. However, keep in mind that Lambda will throttle the function if it uses all of its reserved concurrency.

Unreserved concurrent executions: Unreserved executions are equivalent to the total number of available concurrent executions for your account, minus any reserved concurrency. You can compare the unreserved concurrent executions metric with the concurrent executions metric to monitor which functions are exhausting the remaining concurrency pool during heavier workloads.

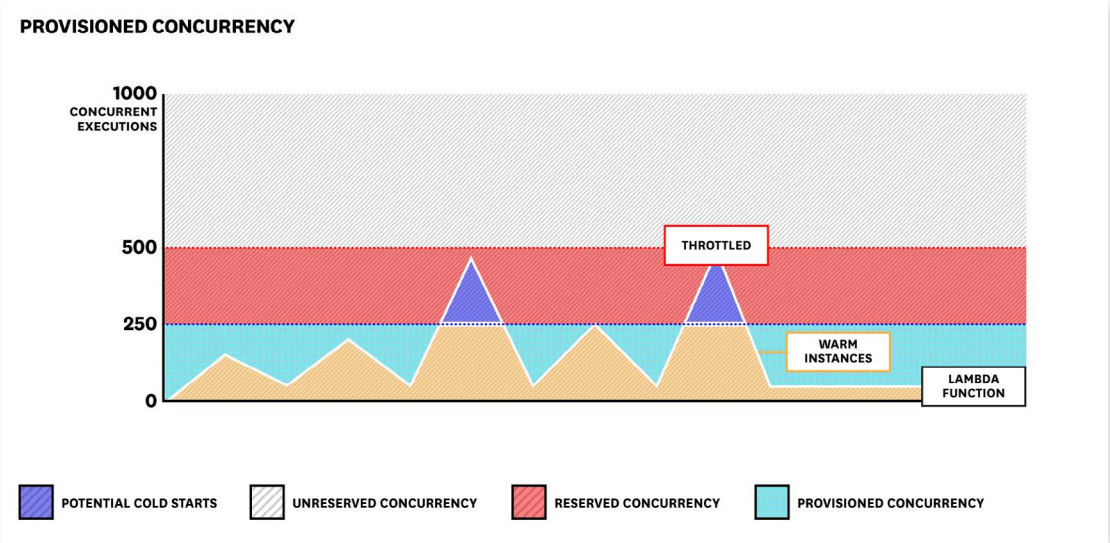
The graphs above show a spike in unreserved concurrency and one function using most of the available concurrency. This could be due to an upstream service sending too many requests to the function.



MONITORING PROVISIONED CONCURRENCY

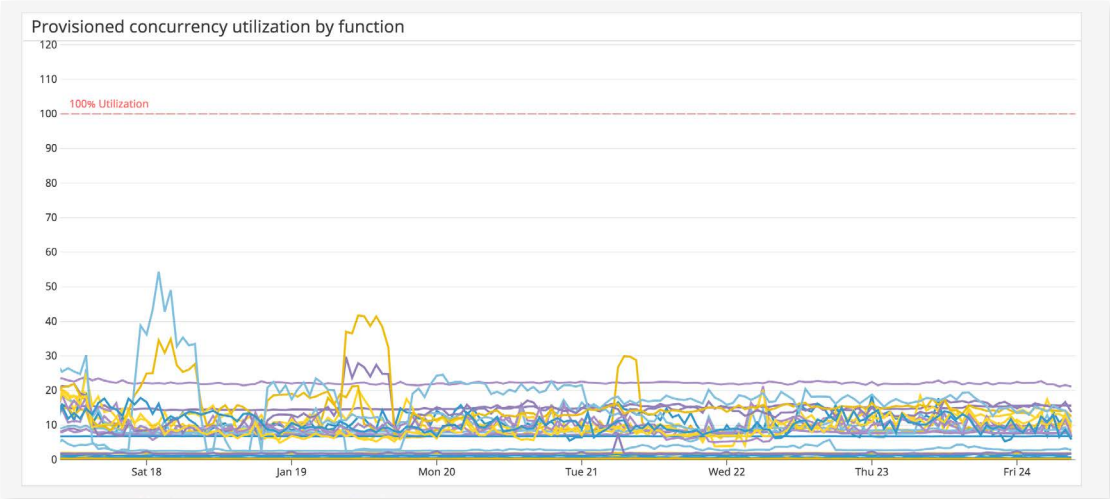
Since Lambda only runs your function code when needed, you may notice additional latency (cold starts) if your functions haven't been used in a while. This is because Lambda needs to initialize a new container and provision packaged dependencies for any inactive functions. Each initialization can add several seconds of latency to function execution. Lambda will keep containers alive for approximately 45 minutes, though that time may vary depending on your region or if you are using VPCs.

If a function has a long startup time (e.g., it has a large number of dependencies), requests may experience higher latency—especially if Lambda needs to initialize new instances to support a burst of requests. You can mitigate this by using provisioned concurrency, which automatically keeps function instances pre-initialized so that they'll be ready to quickly process requests.



Allocating a sufficient level of provisioned concurrency (e.g., the number of warm instances) for a function helps reduce the likelihood that it will encounter cold starts, which can be critical for applications that experience bursts in traffic during specific times of the day (e.g., a food delivery application). You can manage provisioned concurrency with Application Auto Scaling, which enables you to automatically adjust concurrency based on a scaling schedule or utilization to prepare for incoming traffic. Keep in mind that provisioned concurrency comes out of your account's regional concurrency pool and uses a different pricing model.

Provisioned concurrency utilization: One key metric for monitoring the efficiency of a function's provisioned concurrency is provisioned concurrency utilization. A function that is using up all of its available provisioned concurrency—its utilization threshold—may need additional concurrency. Or, if utilization is consistently low, you may have overprovisioned a function. You can disable or reduce provisioned concurrency for that function to manage costs.



Amazon API Gateway

Amazon API Gateway is a service that enables you to create and publish APIs for your applications, without the need for managing dedicated API servers. In the serverless ecosystem, API Gateway can route HTTP requests to the appropriate function, cluster, or instance for processing, as well as provide authentication and authorization layers when connected to other AWS services like Amazon Cognito.

HOW TO MONITOR AMAZON API GATEWAY

The functions that rely on your APIs will fail if an API fails or experiences a significant increase in latency. Monitoring the performance of your APIs—particularly by looking at error and latency metrics—is an important part of ensuring your serverless application is available for customers.

KEY API GATEWAY METRICS

5xx errors: When a client sends a request to an API endpoint, the endpoint returns an HTTP response code to indicate if the request was successful or not. A 200 HTTP response, for example, indicates that the request was received while a 5xx response is an indicator of a server-side error. 503 (Service Unavailable) is a common server error for API Gateway. These errors are typically caused by misconfigured gateways (e.g., referencing a function that no longer exists) or if there are too many requests to process.

Integration latency and latency: Amazon CloudWatch provides two latency metrics for API Gateway. The integration latency metric measures the time it takes for a function to return a response to API Gateway after it submits a request and can help you monitor the responsiveness of your functions. The latency metric measures the end-to-end responsiveness of your API calls—the time it takes for the API Gateway to return a response after it receives a request from the client.

An increase in integration latency could be a result of "cold requests"—where the API Gateway tries to invoke an idle function—or of the API Gateway making a call to a function or other resource that is in a different AWS region. The end-to-end responsiveness of your API calls (e.g., latency) can be affected by general latency in AWS services, so it's important to compare both integration latency and latency metrics to pinpoint the source of the issue.

MONITORING API GATEWAY LOGS

API latency and error metrics only tell a part of the story when debugging issues with your APIs. You also need to understand the reason behind any anomalous behavior. You can use API Gateway's execution and access logs to get more context around API activity by logging API calls in CloudWatch. Execution logs provide more details about the types of calls that were made, including information about errors, request or response parameters, and payloads. Access logs show who accessed your API and how they accessed it.

For example, monitoring API Gateway logs can help you determine if an increase in API errors is caused by insufficient permissions for a client accessing a specific API, or a downstream Lambda function returning a malformed response. API Gateway logs provide the context needed for troubleshooting issues with your APIs, so you have end-to-end visibility into each API endpoint misconfigured IAM role.

AWS Step Functions

AWS Step Functions is a service that enables you to break down the components of a distributed application into individual state machines, so you can easily manage and visualize the steps within application workloads. You can use Step Functions to perform tasks such as managing registration steps for your application, merging data from multiple systems into a single format for processing, or incorporating a manual approval process into a purchasing system.

HOW TO MONITOR AWS STEP FUNCTIONS

Amazon CloudWatch provides several different metrics for monitoring Step Functions, including metrics for tracking state machine executions. Monitoring executions can help ensure that you do not hit AWS quotas or risk running state machines indefinitely.

KEY AWS STEP FUNCTIONS METRICS

Execution time: State machines can run for one year before timing out, but they can trigger a new execution before the current execution terminates, potentially resulting in an indefinite execution loop. Execution time measures how long a state machine has been running, so you can identify ones that are running for too long and troubleshoot.

Failed executions: Failed executions is another key metric for monitoring the health of your serverless applications. Correlating it with metrics from AWS Lambda can help you pinpoint the cause of a breakdown between your Step Function and Lambda function. For example, if you are seeing an equal increase in both failed state machine executions and Lambda errors, then the issue could be related to a specific Lambda function. If the number of Lambda errors is low, then the cause of the execution errors could be a misconfigured IAM role.

AWS Fargate

AWS Fargate is another tool for building serverless architectures. Fargate is integrated with both Amazon Elastic Container Service (ECS) and Amazon Elastic Kubernetes Service (EKS), enabling you to run containers without having to manually provision servers, or Amazon EC2 instances. Fargate provides more CPU and RAM capacity for instances than Lambda, with no runtime limits.

To launch applications with Fargate on ECS, you define tasks that automatically provision networking, IAM policies, CPU, and memory requirements. Fargate then launches containers based on that configuration to run your applications. For EKS, you create a Fargate profile that determines which Kubernetes pods should run. AWS will automatically provision them with Fargate compute resources that best matches a pod's resource requirements.

HOW TO MONITOR AWS FARGATE

KEY AWS FARGATE METRICS

Since Fargate can be used on both EKS and ECS, we'll look at some key metrics for monitoring Fargate performance on both platforms.

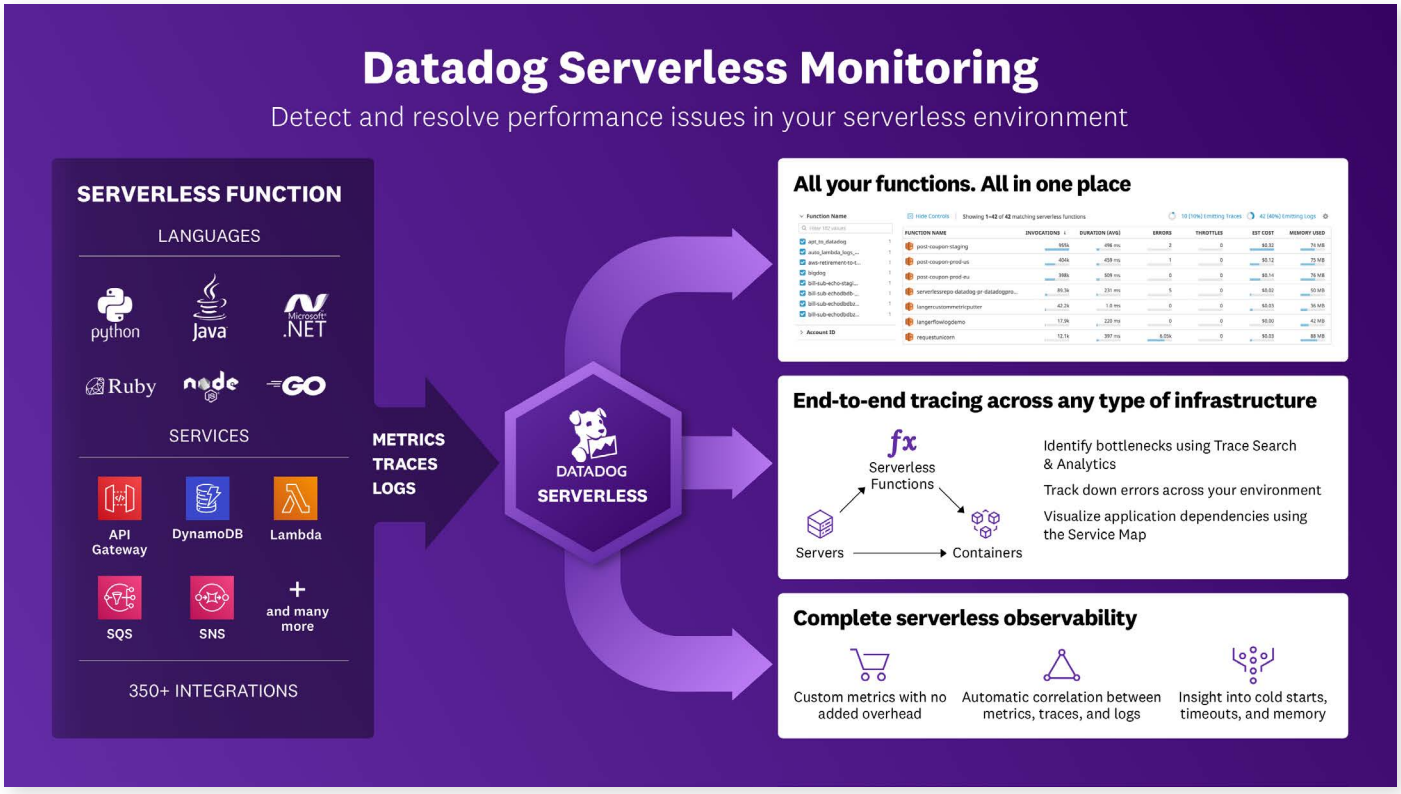
Memory and CPU utilization: AWS pricing is based on a task's or pod's configured CPU and memory resources, so memory and CPU utilization are critical metrics for ensuring that you have not over- or under-provisioned your containers. You may be spending more than what is necessary if you have allocated more memory for a task than what it typically uses. For example, if you dedicate 8 GB of memory to a task that only uses 2 GB, then you can reduce provisioned memory to save costs without much risk of affecting performance. Conversely, your EKS pods could be terminated if their memory usage exceeds their configured limits.

Monitoring CPU utilization for EKS pods and ECS clusters can also help you determine when compute resources are close to throttling. If they are close to hitting their CPU thresholds, you can scale your deployments to ensure they are able to accommodate the load.

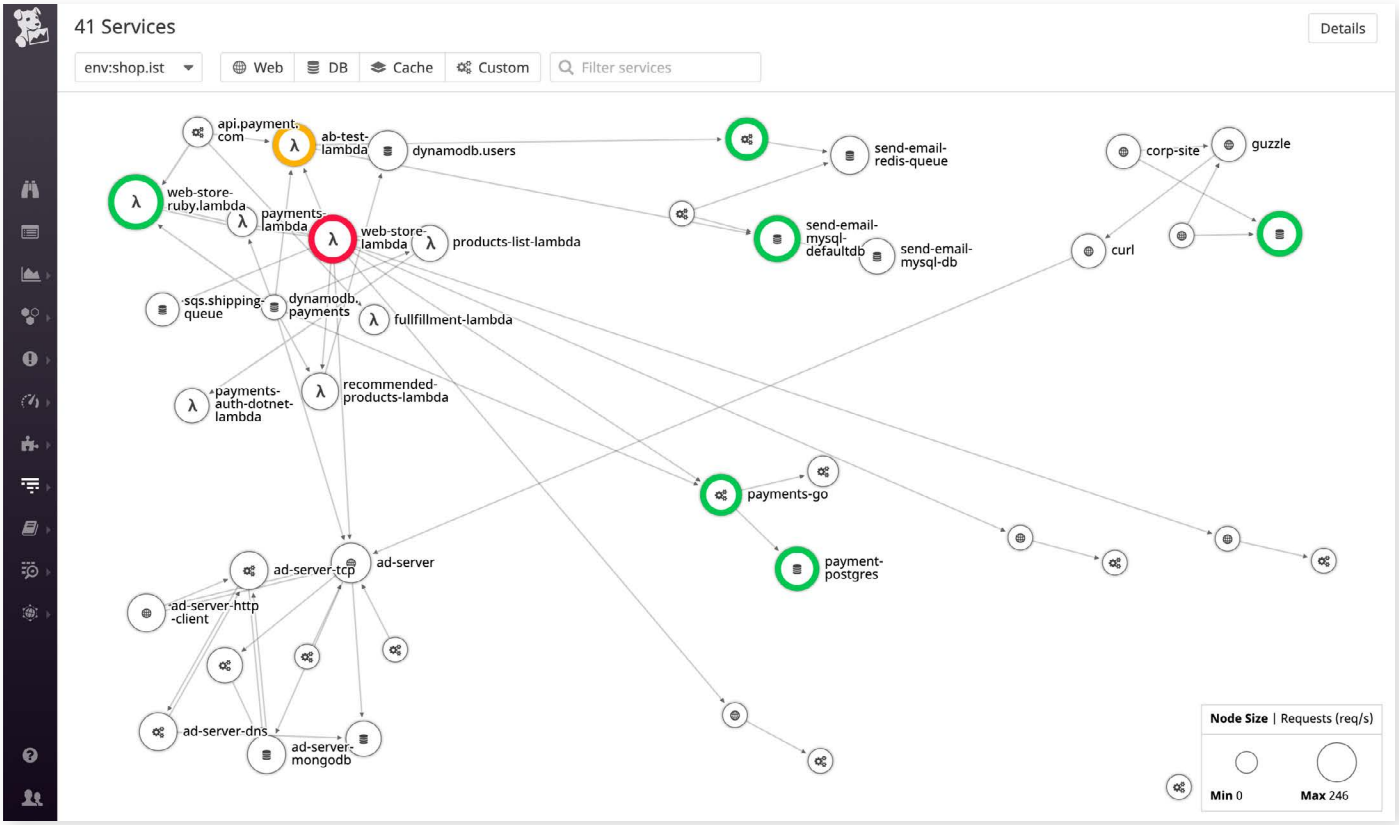
Monitoring your AWS serverless platform with Datadog

Serverless applications introduce a new set of challenges for monitoring. You should be mindful of how incoming requests and other services interact with your functions, as well as how resilient your functions are to high demand or errors. For example, an influx of new requests could cause AWS to throttle your function if it does not have enough concurrency to process that traffic. Or, errors from an upstream service could stop your function code from executing.

To effectively monitor serverless applications, you need visibility into your entire serverless architecture so that you can understand how your functions and other AWS services are interoperating. Datadog provides full visibility into the state of your serverless applications in one place. You can identify service bottlenecks with end-to-end tracing, track custom metrics, correlate data, and get insight into function performance.



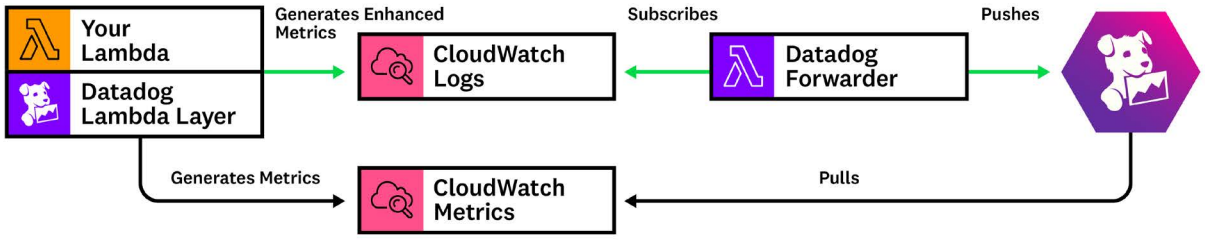
At a high level, Datadog provides a built-in AWS integration that collects CloudWatch data from all of your AWS services, including those used for your serverless applications (e.g., Lambda, Fargate, API Gateway, Step Functions). With Datadog's Service Map, you can visualize all your serverless components in one place and understand the flow of traffic across upstream and downstream dependencies in your environment.



For deeper insights into your serverless functions, Datadog uses a dedicated Lambda Layer and Forwarder to collect observability data from your functions. Datadog's Lambda Layer runs as a part of each function's runtime and works with the Datadog Forwarder Lambda function to generate enhanced metrics at a higher granularity than standard CloudWatch metrics. Data collected with the Lambda Layer complements the metrics, logs, and other traces that you are already collecting via Datadog's out-of-the-box AWS integrations.

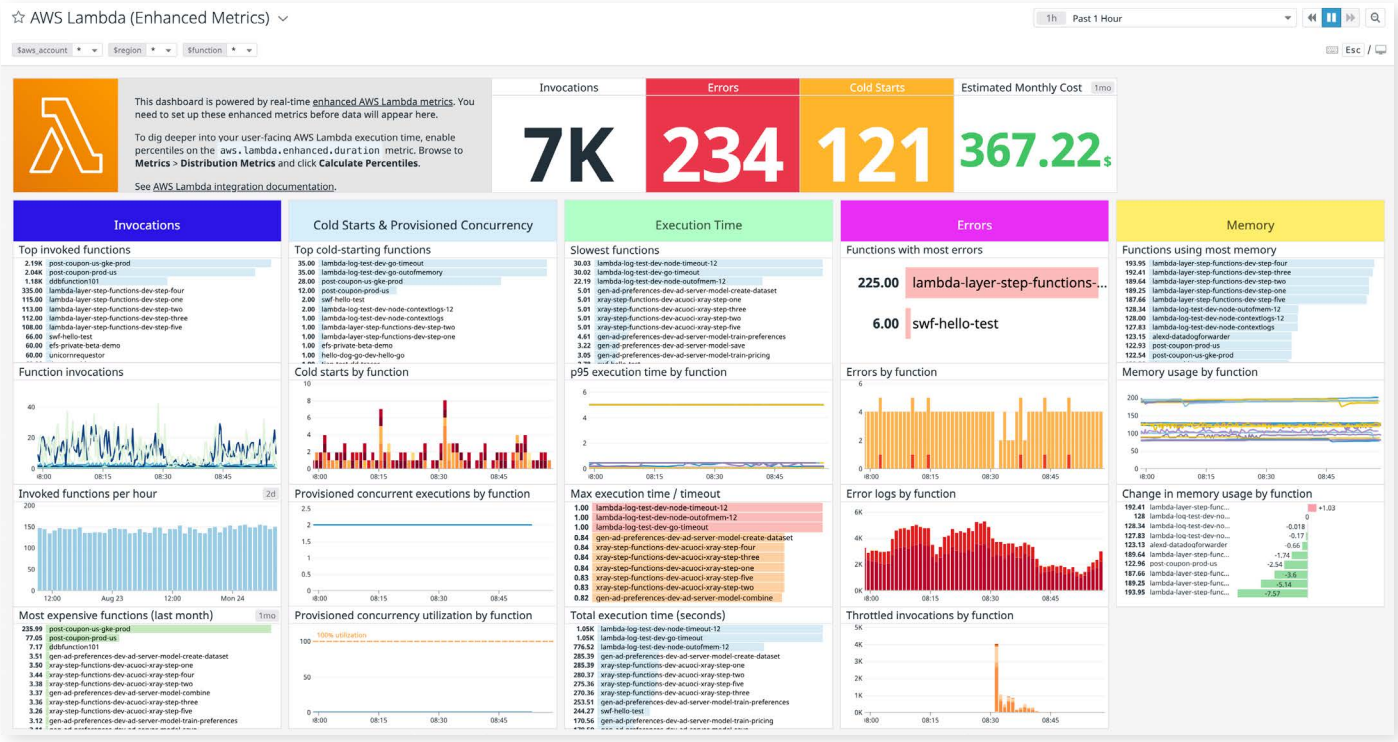
VISUALIZE YOUR SERVERLESS METRICS

COLLECTING AWS LAMBDA METRICS



Datadog uses the Lambda Layer and Forwarder to collect custom metrics and generate real-time enhanced metrics, which are metrics, such as billed duration, timeouts, and estimated cost, that Datadog automatically extracts from your AWS function logs. The Lambda Layer can also send custom metrics (synchronously or asynchronously), giving you additional insights into use cases that are unique to your application workflows, such as a user logging into your application, purchasing an item, or updating a user profile. Sending metrics asynchronously is recommended because it does not add any overhead to your code, making it an ideal solution for functions that power performance-critical tasks for your applications.

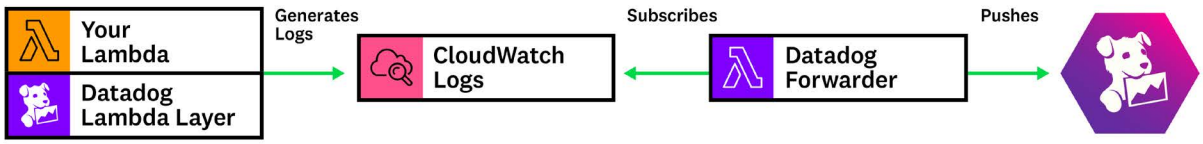
Datadog provides out-of-the-box integration dashboards for your AWS infrastructure, including dashboards for Lambda, Step Functions, and Fargate, giving you a high-level overview of how your serverless applications are performing.



For example, with the dashboard above, you can easily track cold starts, errors, and memory usage for all of your Lambda functions. You can also customize your dashboards to include function logs and trace data, as well as metrics from any of your other services for easy correlation.

SEARCH AND ANALYZE SERVERLESS LOGS IN ONE PLACE

LOGGING WITH AWS LAMBDA

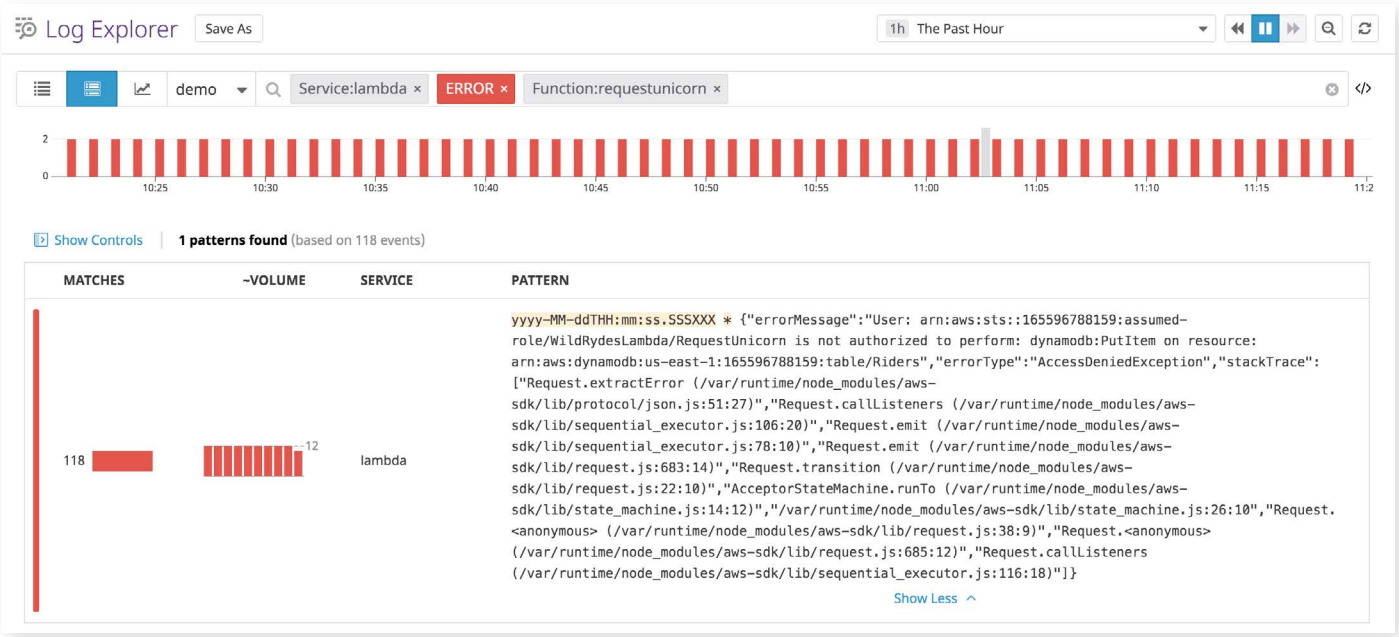


Datadog's Lambda Layer automatically forwards CloudWatch logs to the Datadog Forwarder, which then pushes them to Datadog. The Forwarder can send logs and other telemetry to your account, such as Amazon S3 events and Amazon Kinesis data stream events. Deploying the Forwarder via CloudFormation is recommended as AWS will then automatically create the Lambda function with the appropriate role, add Datadog's Lambda Layer, and create relevant tags like `functionname`, `region`, and `account_id`, which you can then use in Datadog to search your logs.

Because the Forwarder is a Lambda function, it relies on triggers to execute. You can let Datadog automatically set these triggers up for you, or you can manually set them up to forward data as soon as they are added to S3 buckets or CloudWatch log groups. Once configured, Datadog's Lambda Forwarder will begin sending logs from Lambda (and any other AWS services you've configured) to your Datadog account.

Lambda functions generate a large volume of logs, making it difficult to pinpoint issues during an incident or simply to monitor the current state of your functions. You can use Datadog's Log Patterns to help you surface interesting trends in your logs.

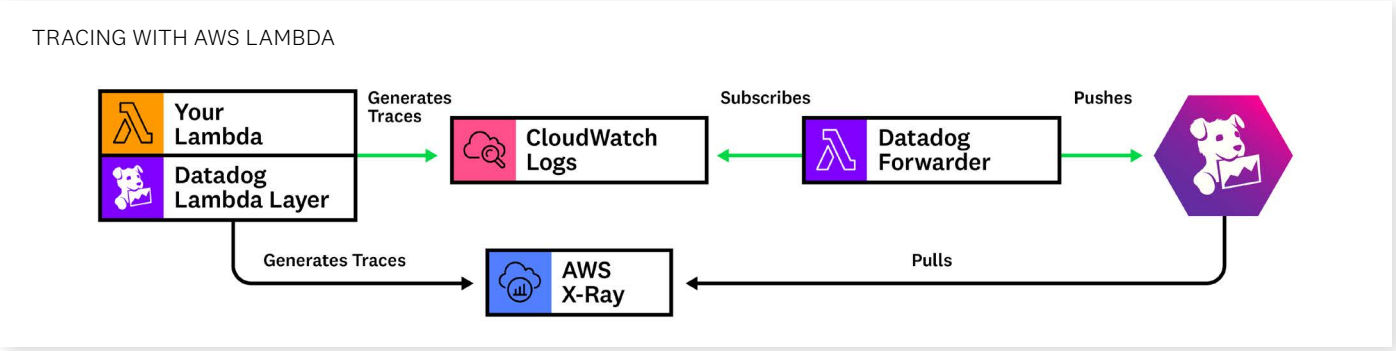
For example, if you notice a spike in Lambda errors on your dashboard, you can use Log Patterns to quickly search for the most common types of errors. In the example below, you can see a cluster of function logs recording an `AccessDeniedException` permissions error. The logs provide a stack trace so you can troubleshoot further.



ROUTE DATADOG LOG ALERTS TO AMAZON EVENTBRIDGE

Datadog also enables you to create alerts from your logs so you can be notified of issues and have the ability to automate workflows for managing your functions through our Amazon EventBridge integration. For example, if a Lambda function triggers a Datadog log alert for out-of-memory errors, you can use EventBridge to automatically increase memory for that function. This enables you to streamline your remediation pipelines so you can ensure applications keep running.

EXPLORE TRACE DATA WITH DATADOG APM

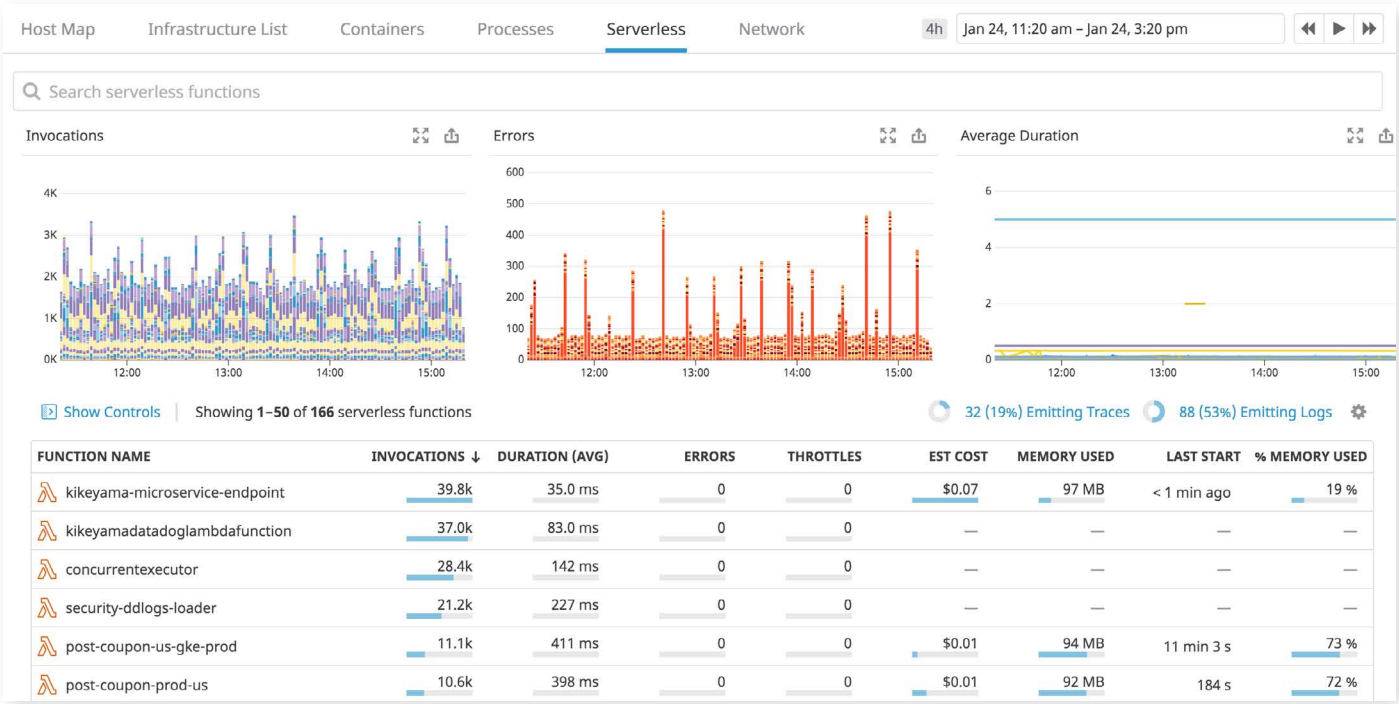


The Datadog Lambda Layer automatically propagates trace headers across services, providing end-to-end distributed tracing for your serverless applications. Datadog APM provides tracing libraries that you can use with the Lambda Layer in order to natively trace request traffic across your serverless architecture. Traces are sent asynchronously, so they don't add any latency overhead to your serverless applications.

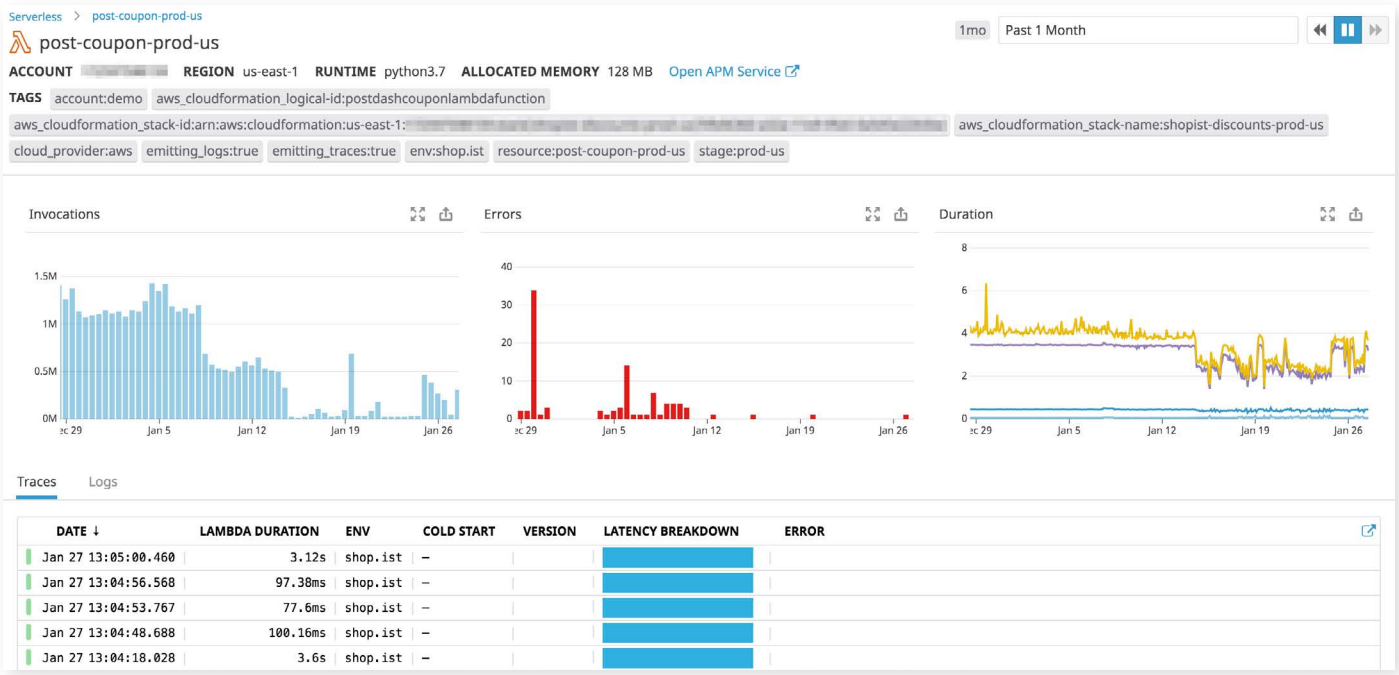
Datadog also provides integrations for other services you may use with your serverless applications, such as AWS Fargate, Amazon API Gateway, Amazon SNS, and Amazon SQS. This ensures that you get visibility into every layer of your serverless architecture. With these integrations enabled, you can drill down to specific functions that are generating errors or cold starts to optimize their performance. AWS charges based on the time it takes for a function to execute, how much memory is allocated to each function, and the number of requests for your function. This means that your costs could quickly increase if, for instance, a high-volume function makes a call to an API Gateway service experiencing network outages and has to wait for a response.

With tracing, you can map upstream and downstream dependencies such as API Gateway and trace requests across your entire stack to pinpoint any latency bottlenecks. And through the Datadog Forwarder, you can also analyze serverless logs to quickly identify the types of errors your functions generate.

To start analyzing trace data from your serverless functions, you can use Datadog's Serverless view. This view gives a comprehensive look at all of your functions and includes key metrics such as invocation count and memory usage. You can search for a specific function or view performance metrics across all your functions. You can also sort your functions in the Serverless view by specific metrics to help surface functions that use the most memory, or that are invoked the most, as seen in the example below.

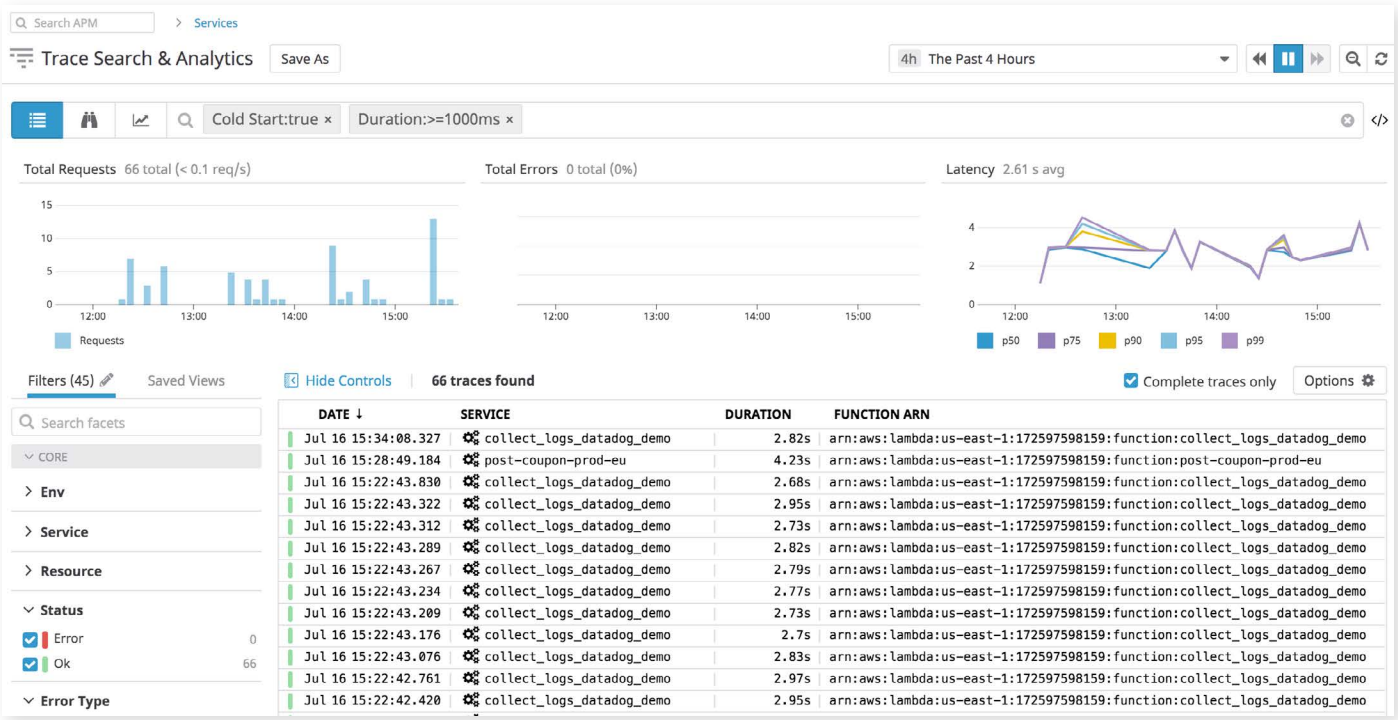


When you click on a function, you will see all of its associated traces and logs, as well as a detailed breakdown of information for each invocation such as duration, related error messages, and whether the function experienced a cold start during the invocation.



API latency and cold starts are two common issues with serverless functions, both of which can significantly increase a function's execution time. Cold starts typically occur when functions scale behind the scenes to handle more requests. API latency could be a result of network or other service outages. Datadog enables you to proactively monitor latency and cold starts for all your functions.

For example, you can create an anomaly alert to notify you when a function experiences unusual latency. From the triggered alert, you can pivot to traces and logs to determine if the latency was caused by cold starts or degradation in an API service dependency. Datadog also automatically detects when cold starts occur and applies a `cold_start` tag to your traces, so you can easily surface functions that are experiencing cold starts to troubleshoot further.



If a function's increased execution time is a result of too many cold starts, you can configure Lambda to reduce initialization latency by using provisioned concurrency. Latency from an API service, on the other hand, may be a result of cross-region calls. In that case, you may need to colocate your application resources into the same AWS region.

FULL VISIBILITY INTO YOUR SERVERLESS ECOSYSTEM

We've looked at some key metrics for monitoring your serverless applications as well as how to troubleshoot common serverless problems. AWS provides a comprehensive suite of tools that enable you to focus more on building scalable services and less on provisioning or managing infrastructure resources.

Datadog offers deep visibility into these serverless applications, enabling you to easily collect observability data through built-in service integrations, a dedicated Lambda Layer, and an easy-to-install Forwarder. You can visualize your serverless metrics with integration dashboards, sift through logs with Datadog's Log Explorer, and analyze distributed request traces with Datadog APM.

Further reading

You can learn more about monitoring your serverless applications by checking out the following guides:

- AWS Lambda monitoring series
- Monitoring Amazon EKS on AWS Fargate
- Monitoring AWS Step Functions

Or, if you would like to put these monitoring strategies in action for your serverless environments, you can sign up for a full-featured Datadog trial at www.datadog.com.


aws
DATADOG