AWS Compute Blog

Building well-architected serverless applications: Optimizing application performance – part 2

by Julian Wood | on 24 AUG 2021 | in Amazon API Gateway, Amazon CloudFront, Amazon DynamoDB, Amazon Simple Queue Service (SQS), Amazon VPC, AWS Lambda, AWS Step Functions, Kinesis Data Streams, Serverless | Permalink | Share

This series of blog posts uses the AWS Well-Architected Tool with the Serverless Lens to help customers build and operate applications using best practices. In each post, I address the serverless-specific questions identified by the Serverless Lens along with the recommended best practices. See the introduction post for a table of contents and explanation of the example application.

PERF 1. Optimizing your serverless application's performance

This post continues part 1 of this security question. Previously, I cover measuring and optimizing function startup time. I explain cold and warm starts and how to reuse the Lambda execution environment to improve performance. I show a number of ways to analyze and optimize the initialization startup time. I explain how only importing necessary libraries and dependencies increases application performance.

Good practice: Design your function to take advantage of concurrency via asynchronous and stream-based invocations

AWS Lambda functions can be invoked synchronously and asynchronously.

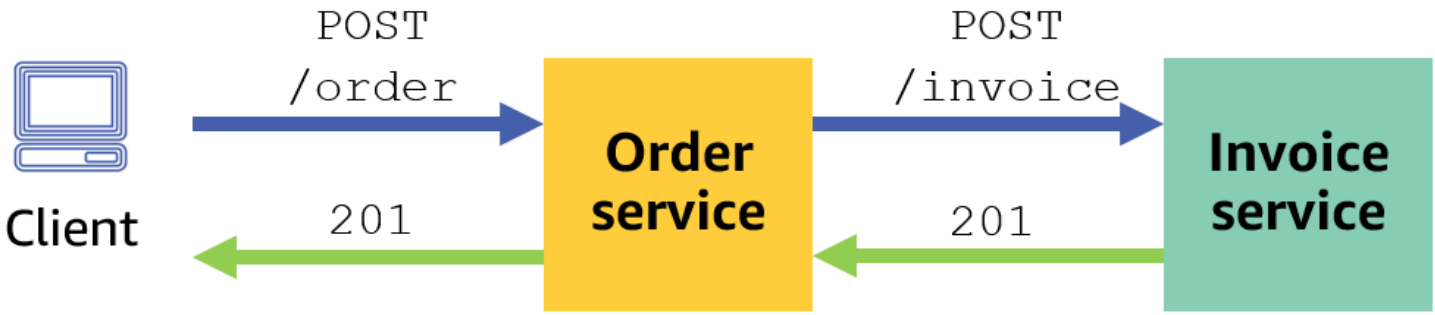
Favor asynchronous over synchronous request-response processing.

Consider using asynchronous event processing rather than synchronous request-response processing. You can use asynchronous processing to aggregate queues, streams, or events for more efficient processing time per invocation. This reduces wait times and latency from requesting apps and functions.

When you invoke a Lambda function with a synchronous invocation, you wait for the function to process the event and return a response.

Synchronous invocation

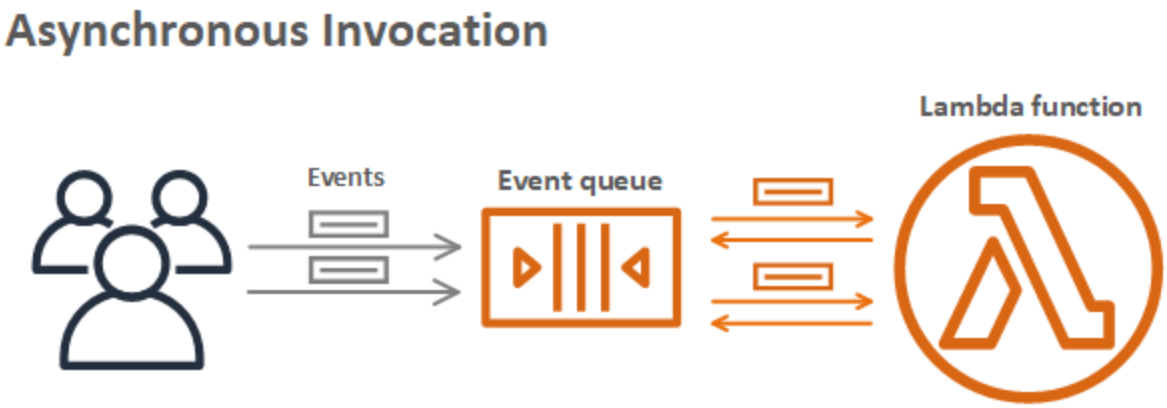
As synchronous processing involves a request-response pattern, the client caller also needs to wait for a response from a downstream service. If the downstream service then needs to call another service, you end up chaining calls that can impact service reliability, in addition to response times. For example, this POST /order request must wait for the response to the POST /invoice request before responding to the client caller.



Example synchronous processing

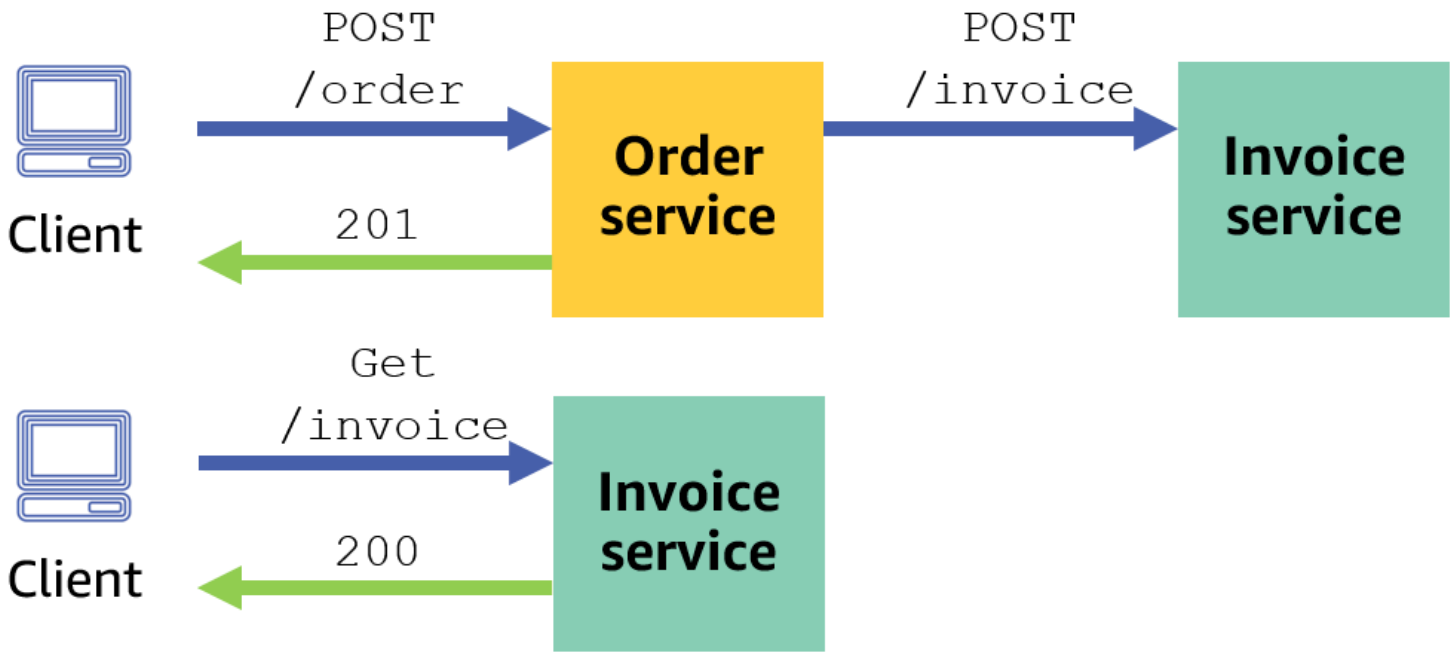
The more services you integrate, the longer the response time, and you can no longer sustain complex workflows using synchronous transactions.

Asynchronous processing allows you to decouple the request-response using events without waiting for a response from the function code. This allows you to perform background processing without requiring the client to wait for a response, improving client performance. You pass the event to an internal Lambda queue for processing and Lambda handles the rest. An external process, separate from the function, manages polling and retries. Using this asynchronous approach can also make it easier to handle unpredictable traffic with significant volumes.



Asynchronous invocation

For example, the client makes a POST /order request to the order service. The order service accepts the request and returns that it has been received, without waiting for the invoice service. The order service then makes an asynchronous POST /invoice request to the invoice service, which can then process independently of the order service. If the client must receive data from the invoice service, it can handle this separately via a GET /invoice request.



Example asynchronous processing

You can configure Lambda to send records of asynchronous invocations to another destination service. This helps you to troubleshoot your invocations. You can also send messages or events that can't be processed correctly into a dedicated Amazon Simple Queue Service (SQS) dead-letter queue for investigation.

You can add triggers to a function to process data automatically. For more information on which processing model Lambda uses for triggers, see "Using AWS Lambda with other services".

Asynchronous workflows handle a variety of use cases including data Ingestion, ETL operations, and order/request fulfillment. In these use-cases, data is processed as it arrives and is retrieved as it changes. For example asynchronous patterns, see "Serverless Data Processing" and "Serverless Event Submission with Status Updates".

For more information on Lambda synchronous and asynchronous invocations, see the AWS re:Invent presentation "Optimizing your serverless applications".

Tune batch size, batch window, and compress payloads for high throughput

When using Lambda to process records using Amazon Kinesis Data Streams or SQS, there are a number of tuning parameters to consider for performance.

You can configure a *batch window* to buffer messages or records for up to 5 minutes. You can set a limit of the maximum number of records Lambda can process by setting a *batch size.* Your Lambda function is invoked whichever comes first.

For high volume SQS standard queue throughput, Lambda can process up to 1000 concurrent batches of records per second. For more information, see "Using AWS Lambda with Amazon SQS".

For high volume Kinesis Data Streams throughput, there are a number of options. Configure the `ParallelizationFactor` setting to process one shard of a Kinesis Data Stream with more than one Lambda invocation simultaneously. Lambda can process up to 10 batches in each shard. For more information, see "New AWS Lambda scaling controls for Kinesis and DynamoDB event sources." You can also add more shards to your data stream to increase the speed at which your function can process records. This increases the function concurrency at the expense of ordering per shard. For more details on using Kinesis and Lambda, see "Monitoring and troubleshooting serverless data analytics applications".

Kinesis enhanced fan-out can maximize throughput by dedicating a 2 MB/second input/output channel per second per consumer instead of 2 MB per shard. For more information, see "Increasing stream processing performance with Enhanced Fan-Out and Lambda".

Kinesis stream producers can also compress records. This is at the expense of additional CPU cycles for decompressing the records in your Lambda function code.

Required practice: Measure, evaluate, and select optimal capacity units

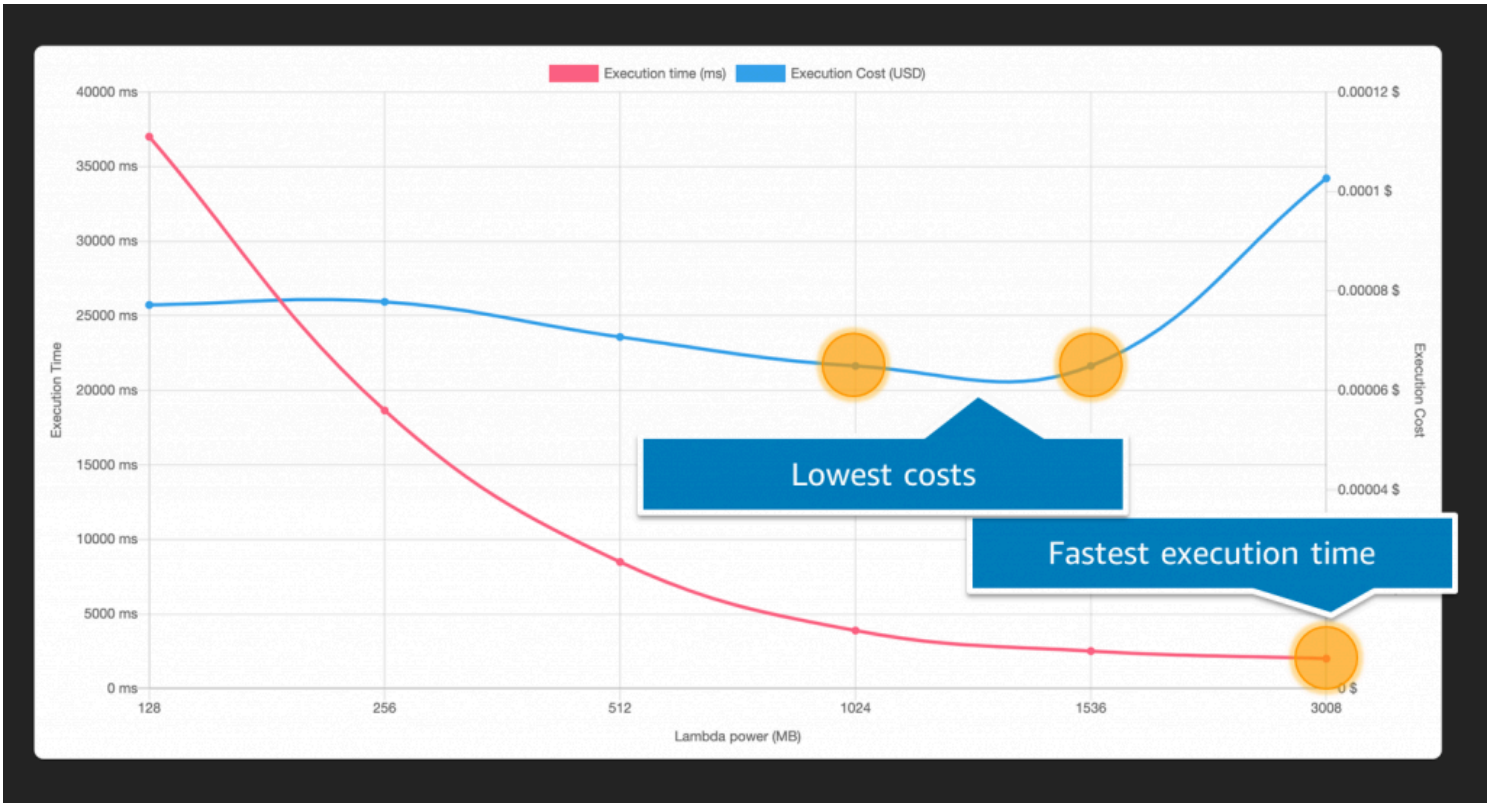
Capacity units are a unit of consumption for a service. They can include function memory size, number of stream shards, number of database reads/writes, request units, or type of API endpoint. Measure, evaluate and select capacity units to enable optimal configuration of performance, throughput, and cost.

Identify and implement optimal capacity units.

For Lambda functions, memory is the capacity unit for controlling the performance of a function. You can configure the amount of memory allocated to a Lambda function, between 128 MB and 10,240 MB. The amount of memory also determines the amount of virtual CPU available to a function. Adding more memory proportionally increases the amount of CPU, increasing the overall computational power available. If a function is CPU-, network- or memory-bound, then changing the memory setting can dramatically improve its performance.

Choosing the memory allocated to Lambda functions is an optimization process that balances performance (duration) and cost. You can manually run tests on functions by selecting different memory allocations and measuring the time taken to complete. Alternatively, use the AWS Lambda Power Tuning tool to automate the process.

The tool allows you to systematically test different memory size configurations and depending on your performance strategy – cost, performance, balanced – it identifies what is the most optimum memory size to use. For more information, see "Operating Lambda: Performance optimization – Part 2".



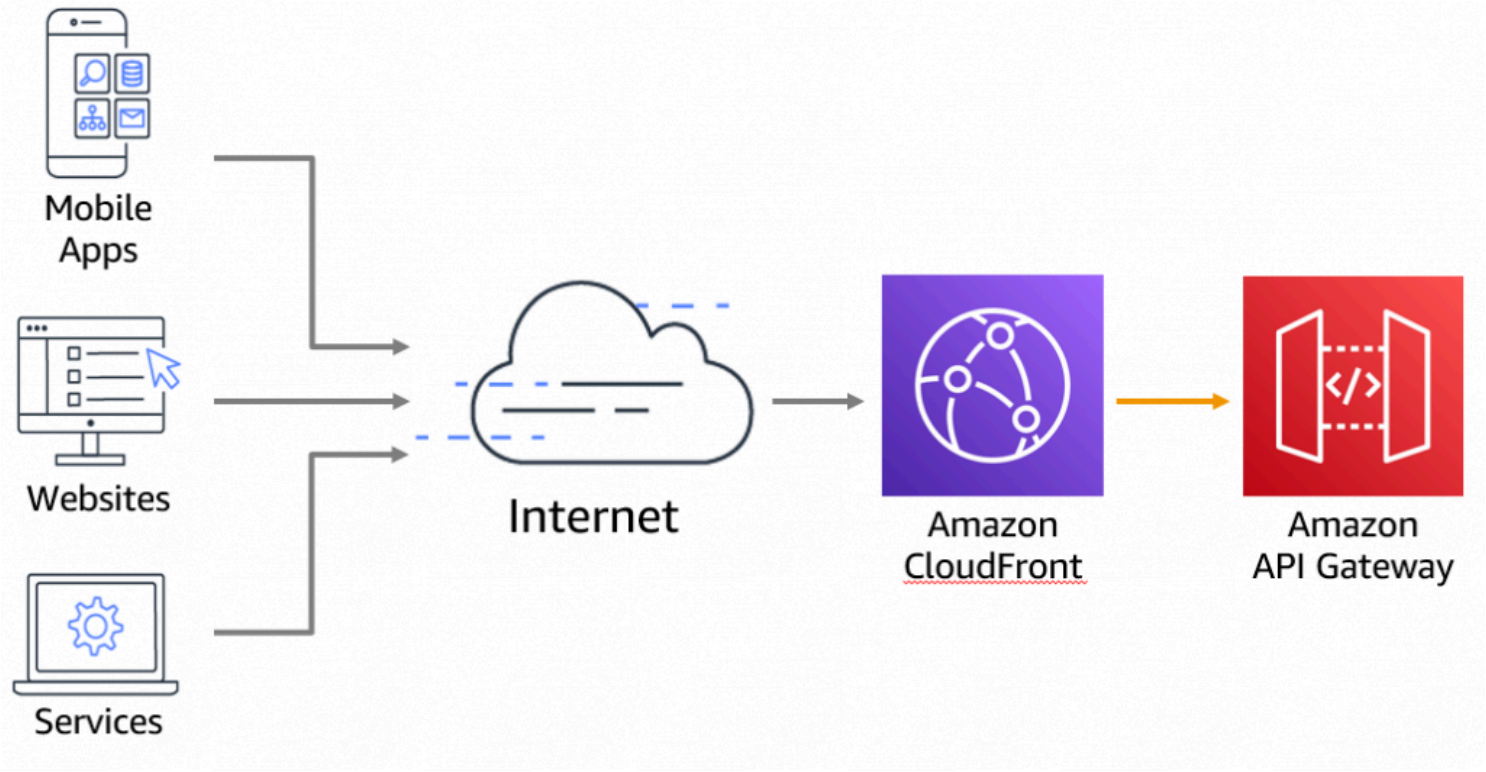
AWS Lambda Power Tuning report

Amazon DynamoDB manages table processing throughput using read and write capacity units. There are two different capacity modes, on-demand and provisioned.

On-demand capacity mode supports up to 40K read/write request units per second. This is recommended for unpredictable application traffic and new tables with unknown workloads. For higher and predictable throughputs, provisioned capacity mode along with DynamoDB auto scaling is recommended. For more information, see "Read/Write Capacity Mode".

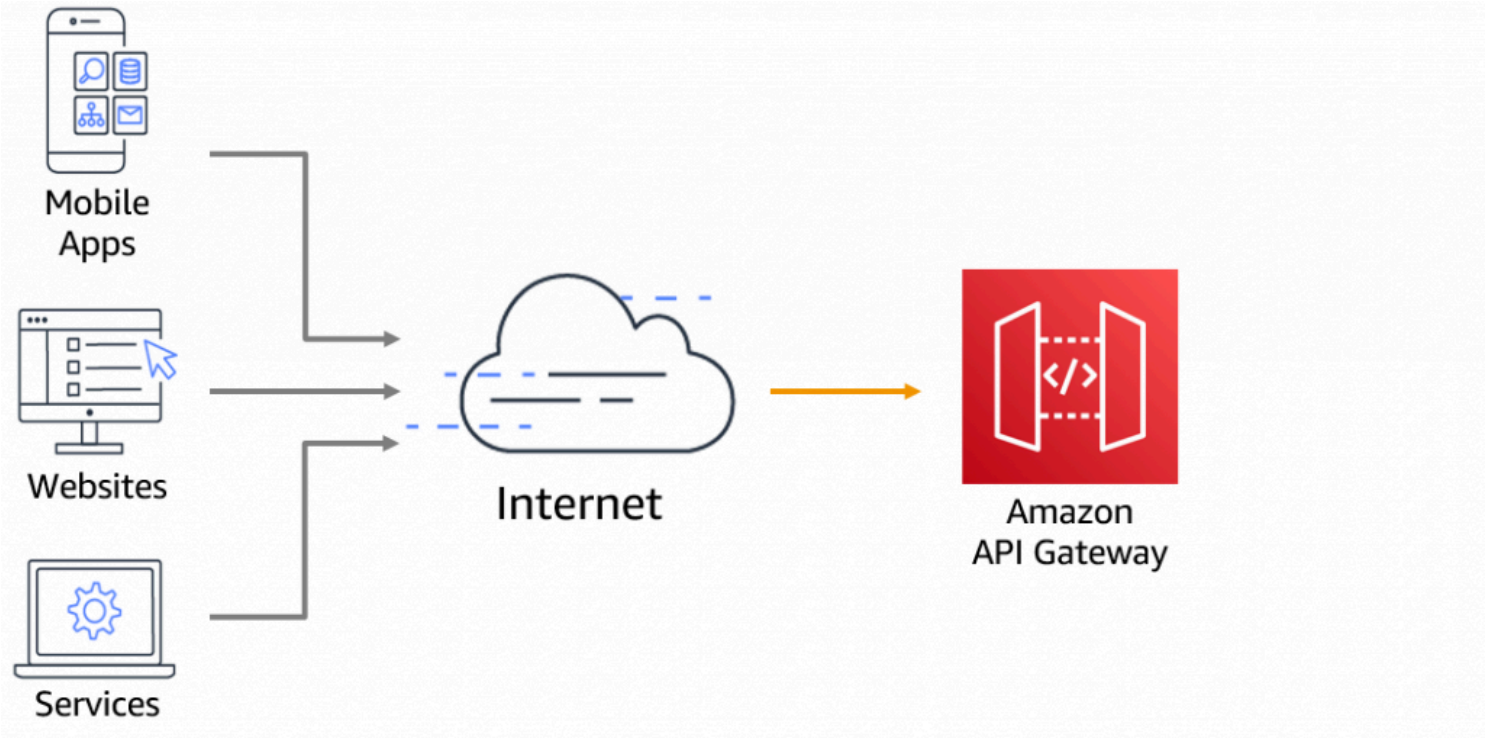
For high throughput Amazon Kinesis Data Streams with multiple consumers, consider using enhanced fan-out for dedicated 2 MB/second throughput per consumer. When possible, use Kinesis Producer Library and Kinesis Client Library for effective record aggregation and de-aggregation.

Amazon API Gateway supports multiple endpoint types. Edge-optimized APIs provide a fully managed Amazon CloudFront distribution. These are better for geographically distributed clients. API requests are routed to the nearest CloudFront Point of Presence (POP), which typically improves connection time.



Edge-optimized API Gateway deployment

Regional API endpoints are intended when clients are in the same Region. This helps you to reduce request latency and allows you to add your own content delivery network if necessary.



Regional endpoint API Gateway deployment

Private API endpoints are API endpoints that can only be accessed from your Amazon Virtual Private Cloud (VPC) using an interface VPC endpoint. For more information, see "Creating a private API in Amazon API Gateway".

For more information on endpoint types, see "Choose an endpoint type to set up for an API Gateway API". For more general information on API Gateway, see the AWS re:Invent presentation "I didn't know Amazon API Gateway could do that".

AWS Step Functions has two workflow types, standard and express. Standard Workflows have exactly once workflow execution and can run for up to one year. Express Workflows have at-least-once workflow execution and can run for up to five minutes. Consider the per-second rates you require for both execution start rate and the state transition rate. For more information, see "Standard vs. Express Workflows".

Performance load testing is recommended at both sustained and burst rates to evaluate the effect of tuning capacity units. Use Amazon CloudWatch service dashboards to analyze key performance metrics including load testing results. I cover performance testing in more detail in "Regulating inbound request rates – part 1".

For general serverless optimization information, see the AWS re:Invent presentation "Serverless at scale: Design patterns and optimizations".

Conclusion

Evaluate and optimize your serverless application's performance based on access patterns, scaling mechanisms, and native integrations. You can improve your overall experience and make more efficient use of the platform in terms of both value and resources.

This post continues from part 1 and looks at designing your function to take advantage of concurrency via asynchronous and stream-based invocations. I cover measuring, evaluating, and selecting optimal capacity units.

This well-architected question continues in part 3 where I look at integrating with managed services directly over functions when possible. I cover optimizing access patterns and applying caching where applicable.

For more serverless learning resources, visit Serverless Land.
TAGS: serverless, well-architected