AWS Compute Blog

Building well-architected serverless applications: Optimizing application performance – part 3

by Julian Wood | on 31 AUG 2021 | in Amazon API Gateway, Amazon CloudFront, Amazon DynamoDB, Amazon DynamoDB Accelerator (DAX), Amazon OpenSearch Service, AWS AppSync, AWS Batch, Kinesis Data Firehose, Serverless | Permalink | Share

February 12, 2024*: Amazon Kinesis Data Firehose has been renamed to Amazon Data Firehose. Read the AWS What's New post to learn more.*

September 8, 2021*: Amazon Elasticsearch Service has been renamed to Amazon OpenSearch Service. See details.*

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 2 of this security question. Previously, I look at designing your function to take advantage of concurrency via asynchronous and stream-based invocations. I cover measuring, evaluating, and selecting optimal capacity units.

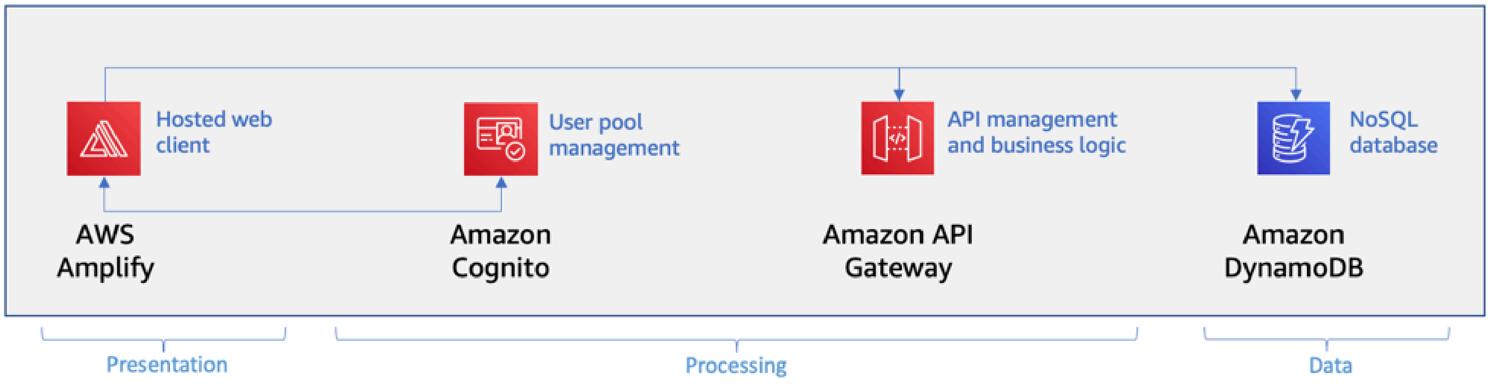
Best practice: Integrate with managed services directly over functions when possible

Consider using native integrations between managed services as opposed to AWS Lambda functions when no custom logic or data transformation is required. This can enable optimal performance, requires less resources to manage, and increases security. There are also a number of AWS application integration services that enable communication between decoupled components with microservices.

Use native cloud services integration

When using Amazon API Gateway APIs, you can use the `AWS` integration type to connect to other AWS services natively. With this integration type, API Gateway uses Apache Velocity Template Language (VTL) and HTTPS to directly integrate with other AWS services.

Timeouts and errors must be managed by the API consumer. For more information on using VTL, see "Amazon API Gateway Apache Velocity Template Reference". For an example application that uses API Gateway to read and write directly to/from Amazon DynamoDB, see "Building a serverless URL shortener app without AWS Lambda".

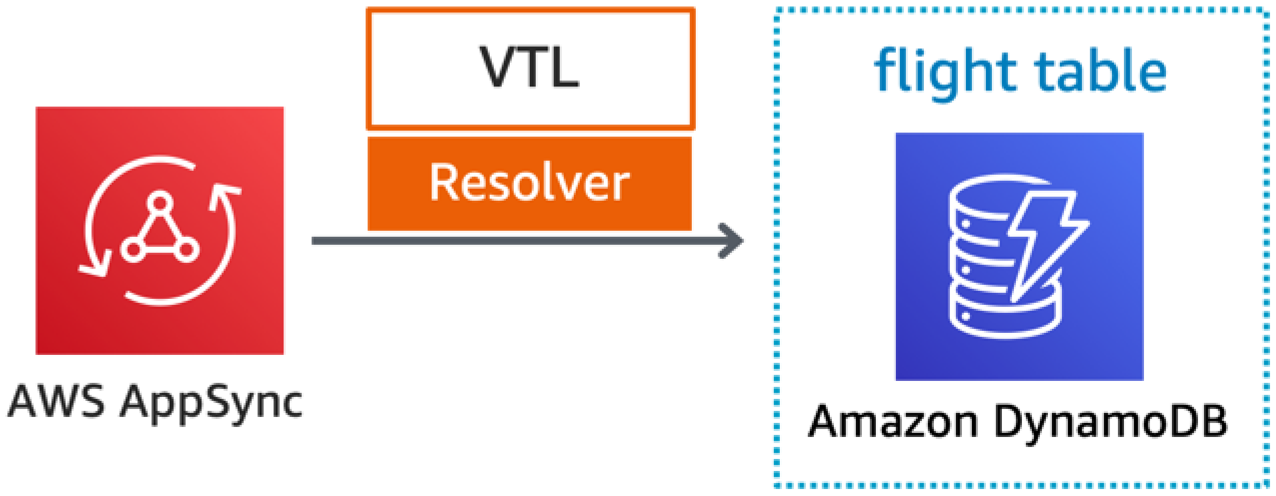


API Gateway direct service integration

There is also a tutorial available, Build an API Gateway REST API with AWS integration.

When using AWS AppSync, you can use VTL, direct integration with Amazon Aurora, Amazon Elasticsearch Service, and any publicly available HTTP endpoint. AWS AppSync can use multiple integration types and can maximize throughput at the data field level. For example, you can run full-text searches on the *orderDescription* field against Elasticsearch while fetching the remaining data from DynamoDB. For more information, see the AWS AppSync resolver tutorials.

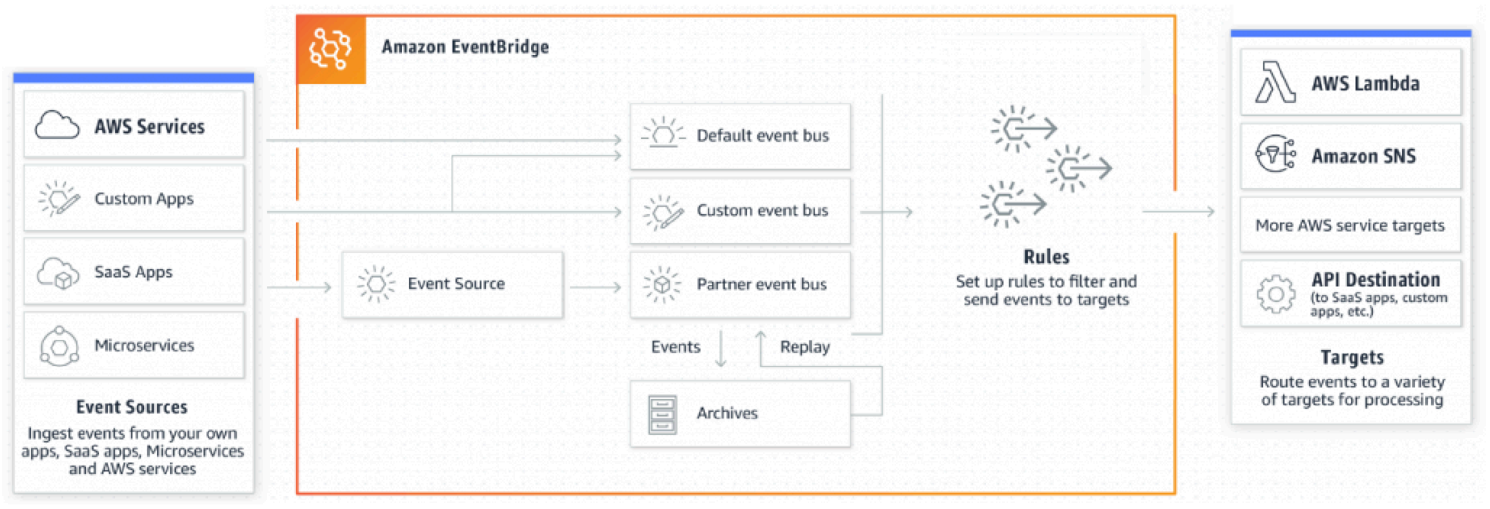
In the serverless airline example used in this series, the catalog service uses AWS AppSync to provide a GraphQL API for searching flights. AWS AppSync uses DynamoDB as a database, and all compute logic is contained in the Apache Velocity Template (VTL).



Serverless airline catalog service using VTL

AWS Step Functions integrates with multiple AWS services using service Integrations. For example, this allows you to fetch and put data into DynamoDB, or run an AWS Batch job. You can also publish messages to Amazon Simple Notification Service (SNS) topics, and send messages to Amazon Simple Queue Service (SQS) queues. For more details on the available integrations, see "Using AWS Step Functions with other services".

Using Amazon EventBridge, you can connect your applications with data from a variety of sources. You can connect to various AWS services natively, and act as an event bus across multiple AWS accounts to ease integration. You can also use the API destination feature to route events to services outside of AWS. EventBridge handles the authentication, retries, and throughput. For more details on available EventBridge targets, see the documentation.



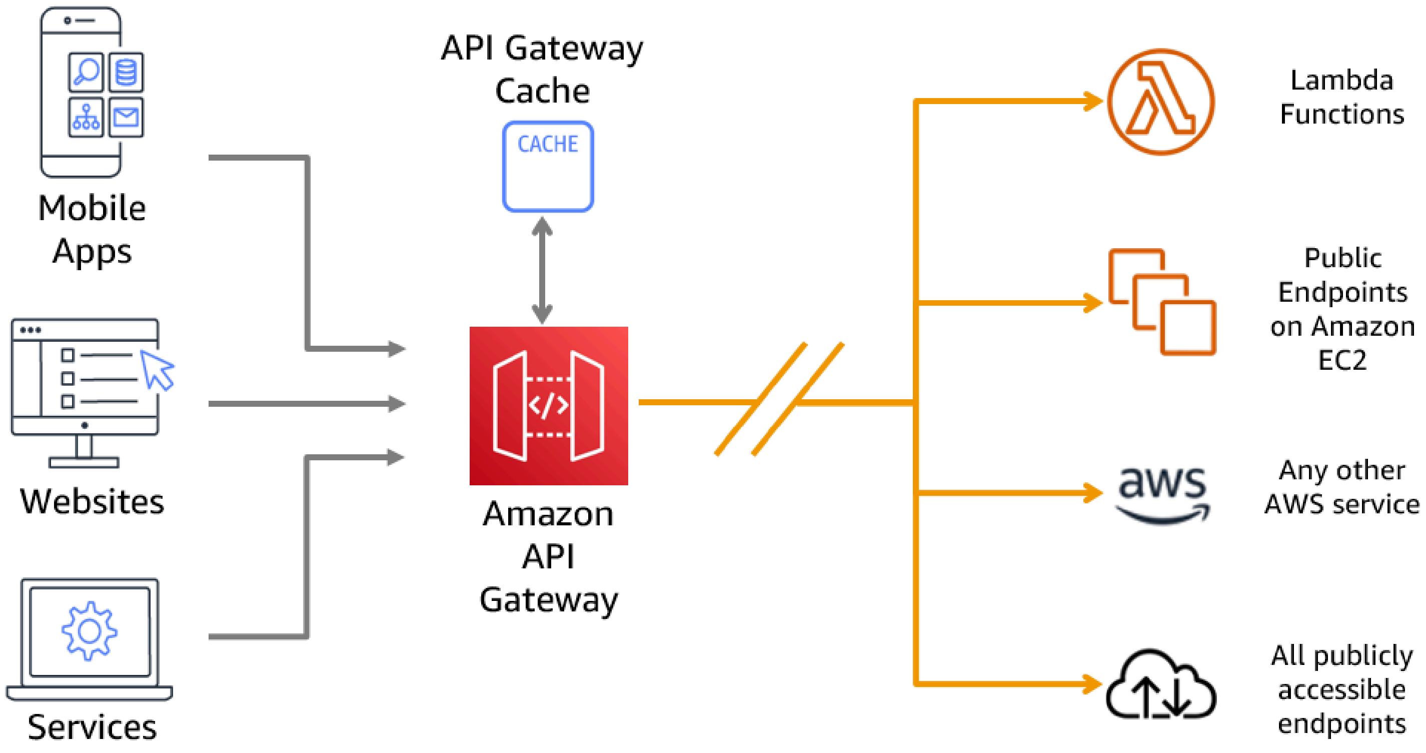
Amazon EventBridge

Good practice: Optimize access patterns and apply caching where applicable

Consider caching when clients may not require up to date data. Optimize access patterns to only fetch data that is necessary to end users. This improves the overall responsiveness of your workload and makes more efficient use of compute and data resources across components.

Implement caching for suitable access patterns

For REST APIs, you can use API Gateway caching to reduce the number of calls made to your endpoint and also improve the latency of requests to your API. When you enable caching for a stage or method, API Gateway caches responses for a specified time-to-live (TTL) period. API Gateway then responds to the request by looking up the endpoint response from the cache, instead of making a request to your endpoint.



API Gateway caching

For more information, see "Enabling API caching to enhance responsiveness".

For geographically distributed clients, Amazon CloudFront or your third-party CDN can cache results at the edge and further reducing network round-trip latency.

For GraphQL APIs, AWS AppSync provides built-in server-side caching at the API level. This reduces the need to access data sources directly by making data available in a high-speed in-memory cache. This improves performance and decreases latency. For queries with common arguments or a restricted set of arguments, you can also enable caching at the resolver level to improve overall responsiveness. For more information, see "Improving GraphQL API performance and consistency with AWS AppSync Caching".

When using databases, cache results and only connect to and fetch data when needed. This reduces the load on the downstream database and improves performance. Include a caching expiration mechanism to prevent serving stale records. For more information on caching implementation patterns and considerations, see "Caching Best Practices".

For DynamoDB, you can enable caching with Amazon DynamoDB Accelerator (DAX). DAX enables you to benefit from fast in-memory read performance in microseconds, rather than milliseconds. DAX is suitable for use cases that may not require strongly consistent reads. Some examples include real-time bidding, social gaming, and trading applications. For more information, read "Use cases for DAX".

For general caching purposes, Amazon ElastiCache provides a distributed in-memory data store or cache environment. ElastiCache supports a variety of caching patterns through key-value stores using the Redis and Memcache engines. Define what is safe to cache, even when using popular caching patterns like lazy caching or write-through. Set a TTL and eviction policy that fits your baseline performance and access patterns. This ensures that you don't serve stale records or cache data that should have a strongly consistent read. For more information on ElastiCache caching and time-to-live strategies, see the documentation.

For additional serverless caching suggestions, see the AWS Serverless Hero blog post "All you need to know about caching for serverless applications".

Reduce overfetching and underfetching

Over-fetching is when a client downloads too much data from a database or endpoint. This results in data in the response that you don't use. Under-fetching is not having enough data in the response. The client then needs to make additional requests to receive the data. Overfetching and underfetching can both affect performance.

To fetch a collection of items from a DynamoDB table, you can perform a query or a scan. A scan operation always scans the entire table or secondary index. It then filters out values to provide the result you want, essentially adding the extra step of removing data from the result set. A query operation finds items directly based on primary key values.

For faster response times, design your tables and indexes so that your applications can use query instead of scan. Use both Global Secondary Index (GSI) in addition to composite sort keys to help you query hierarchical relationships in your data. For more information, see "Best Practices for Querying and Scanning Data".

Consider GraphQL and AWS AppSync for interactive web applications, mobile, real-time, or for use cases where data drives the user interface. AWS AppSync provides data fetching flexibility, which allows your client to query only for the data it needs, in the format it needs it. Ensure you do not make too many nested queries where a long response may result in timeouts. GraphQL helps you adapt access patterns as your workload evolves. This makes it more flexible as it allows you to move to purpose-built databases if necessary.

Compress payload and data storage

Some AWS services allow you to compress the payload or compress data storage. This can improve performance by sending and receiving less data, and can save on data storage, which can also reduce costs.

If your content supports deflate, gzip or identity content encoding, API Gateway allows your client to call your API with compressed payloads. By default, API Gateway supports decompression of the method request payload. However, you must configure your API to enable compression of the method response payload. Compression in API Gateway and decompression in the client might increase overall latency and require more computing times. Run test cases against your API to determine an optimal value. For more information, see "Enabling payload compression for an API".

Amazon Kinesis Data Firehose supports compressing streaming data using gzip, snappy, or zip. This minimizes the amount of storage used at the destination. The Amazon Kinesis Data Firehose FAQs has more information on compression. Kinesis Data Firehose also supports converting your streaming data from JSON to Apache Parquet or Apache ORC before storing the data in Amazon S3. Parquet and ORC are columnar data formats that save space and enable faster queries compared to row-oriented formats like JSON.

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.

In part 1, 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 explain how only importing necessary libraries and dependencies increases application performance.

In part 2, I look at designing your function to take advantage of concurrency via asynchronous and stream-based invocations. I cover measuring, evaluating, and selecting optimal capacity units.

In this post, I look at integrating with managed services directly over functions when possible. I cover optimizing access patterns and applying caching where applicable.

In the next post in the series, I cover the cost optimization pillar from the Well-Architected Serverless Lens.

For more serverless learning resources, visit Serverless Land.

TAGS: serverless, well-architected