dashbird

AWS Well-Architected Framework and serverless

API Gateway has high error percentage
Timeout in Lambda: users_get_prod
SQS queue laten… minutes

Table of Contents

Introduction to the Well-Architected Framework

The AWS Well-Architected Framework (WAF) is a collection of whitepapers that outline best practices for building software with AWS infrastructure. The framework is split into one main whitepaper that gives an overview and five more detailed whitepapers, called pillars. These pillars are **Operational Excellence** (OPS), **Security** (SEC), **Reliability** (REL), **Performance Efficiency** (PERF), and **Cost Optimization** (COST).

Each pillar contains questions you should be able to answer for your AWS based software. These answers relate to technical or organizational decisions that are not directly related to the features your software provides.

For example, when you build a blog, you want people to write articles and other people to read articles to implement features for these use-cases. But you also want your system to be safe, your servers to handle all the traffic, and have all this for a reasonable price.

You can find out more about how to save money on your Lambdas here.

The answers to these questions are sometimes as simple as using a special AWS service (technical). Sometimes, they require you to implement some kind of process in your company (organizational) because it can't be automated entirely.

The most crucial pillars are OPS and SEC. They should never be traded in to get more out of the other three pillars. This is because the answers to the OPS questions lead to the foundational setup needed to facilitate the answers to other pillars' questions with a sane amount of work. **And the savings on SEC could lead to breaches that can destroy your company overnight.**

The other three pillars REL, PERF, and COST, are a matter of business requirements. **Going too cheap may render your system unusable**, but going 100% on REL and PERF might be far too expensive to build a sustainable business. Depending on your budget, you have to weigh these pillars against each other to get the optimal solution.

Maybe it's okay for a customer to wait a bit longer for their results, and you invest more money into long-term data retention. Maybe nobody will use an expensive system that delivers sub-second results, but they will use one that takes a few seconds but is substantially cheaper.

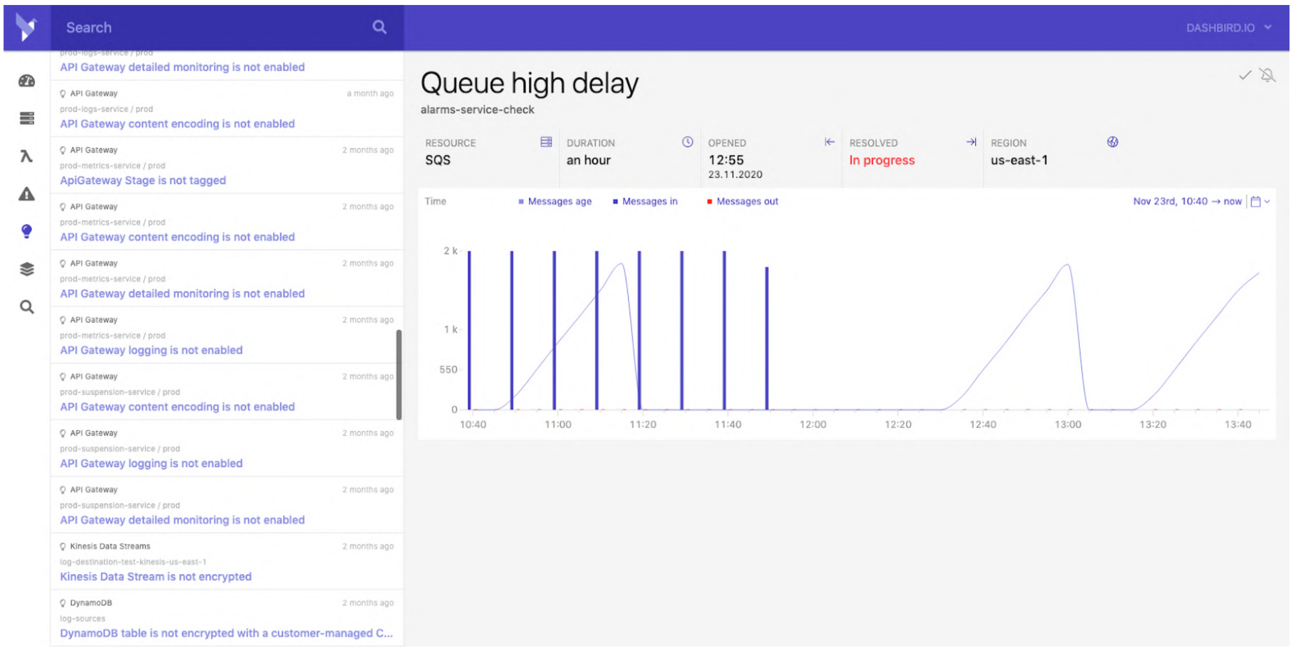
AWS also offers a free Well Architected tool and hands-on guides that help understand the WAF and answer its questions for your software. This way, you can train WAF principles even without your own software and evaluate architecture ideas you got for the software you want to or already have built.

There are additional, more specialized whitepapers, called **lenses**. These consolidate the general WAF pillars' ideas and specify their questions regarding a specific type of software. For example, serverless applications, machine learning, or IoT.

In this article, we will focus on the Serverless Application Lens, which "covers common serverless applications scenarios and identifies key elements to ensure that your workloads are architected according to best practices." A workload here is a **collection of software systems**, called components, that deliver business value.

A monolithic software system can be a workload, but a collection of different microservices can also form a workload.

Dashbird WAF Insights Features



prod-logs-service / prod
API Gateway detailed monitoring is not enabled

API Gateway — a month ago
prod-logs-service / prod
API Gateway content encoding is not enabled

API Gateway — 2 months ago
prod-metrics-service / prod
ApiGateway Stage is not tagged

API Gateway — 2 months ago
prod-metrics-service / prod
API Gateway content encoding is not enabled

API Gateway — 2 months ago
prod-metrics-service / prod
API Gateway detailed monitoring is not enabled

API Gateway — 2 months ago
prod-metrics-service / prod
API Gateway logging is not enabled

API Gateway — 2 months ago
prod-suspension-service / prod
API Gateway content encoding is not enabled

API Gateway — 2 months ago
prod-suspension-service / prod
API Gateway logging is not enabled

API Gateway — 2 months ago
prod-suspension-service / prod
API Gateway detailed monitoring is not enabled

Kinesis Data Streams — 2 months ago
log-destination-test-kinesis-us-east-1
Kinesis Data Stream is not encrypted

DynamoDB — 2 months ago
log-sources
DynamoDB table is not encrypted with a customer-managed C...

Queue high delay
alarms-service-check

RESOURCE	DURATION	OPENED	RESOLVED	REGION
SQS	an hour	12:55 23.11.2020	In progress	us-east-1

Dashbird features a continuous engine for detecting and enforcing Serverless Well-Architected practices. Dashbird is constantly running users' infrastructure data through its WAF rule engines detecting anomalies and weaknesses within the architecture, and notifying the users of critical errors and/or when improvements should be made.

This allows developers to build and operate complex Well-Architected serverless applications without compromising its reliability over time and also be able to layer on complexity later on.

The Security Pillar

As mentioned above, the SEC pillar is one of the two most important pillars, the other one being the OPS pillar. **You shouldn't trade this pillar in to save money or make a bit faster responses.** If a country you operate in requires you to use a minimal amount of security measures for a specific type of data or system, you can't cut corners.

This doesn't mean you have to encrypt all your data with a one time pad and mail people hard drives filled with encryption keys. It just means keeping your system at least as safe as required and going over that if you can justify it somehow without getting you and your customers into trouble.

After all, a security breach can take down your company in a blink of an eye, so it's worth staying safe and secure!

The SEC pillar itself is also subdivided into five parts; let's look into them:

1. Identity and Access Management

The first part is "identity and access management," which forms the core of the pillar. It could also be called authentication and authorization. If you want to keep your system safe, you need to know who (identity) is using it and what they are using (access).

2. Detective Tools

The second part is "detective controls." If you want to protect your system, you need a **way to monitor that security was breached**; otherwise, you don't know it's safe.

3. Infrastructure Protection

The third part is "infrastructure protection," which is about protecting networks and compute resources. In terms of serverless, most of this is done by AWS because the services you use don't expose networks or servers directly. **You still need to use the right access permissions**, but you usually don't need to think about kernel patches or TCP inspection.

4. Data Protection

Then there is "data protection." You don't want your data or the data of your users to get stolen. The most important step in protecting your data is to **classify your data by security needs** because encrypting everything as strong as possible is prohibitively expensive.

5. Incident Response

Last but not least, there is "incident response." After you did everything to protect your system, things can still go wrong, and often your whole reputation is staked on **how "you anticipate, respond to, and recover from incident[s]."**

SAL Questions of the Security Pillar

The SAL asks three questions about the security of your serverless applications. Let's look at how these can be answered.

SEC 1: How do you control access to your Serverless API?

You can build serverless APIs on AWS either with Amazon API Gateway for HTTP, WebSockets, and REST APIs or with AWS AppSync for GraphQL APIs.

To authorize internal (that means inside AWS and inside your account) services, you can use IAM roles. **For internal users, IAM users are the solution.**

If you need your customers or "end users" to access your APIs, you need Amazon Cognito User Pools. They help with sing ups and logins and even integrate with social providers to make the well known "Facebook Login" possible.

If you have to integrate with external services and know the IP ranges they are hosted on, you can configure resource policies. If you don't know the IP ranges, you should use temporary credentials to give access.

This is especially helpful when integrating with legacy auth services.

Finally, API Gateway Lambda authorizers allow you to implement any custom authentication workflow you may need.

SEC 2: How are you managing the security boundaries of your Serverless Application?

The security principle of least privilege also holds in serverless systems. Don't give everyone execution rights to your Lambda functions and only give minimal permissions to every Lambda function. Also, **don't reuse IAM permissions between multiple Lambda functions and forget to remove permissions again when they become obsolete.**

Your CI/CD pipeline should also include a vulnerability scan that checks your code and all of its dependencies for security issues right before you deploy it.

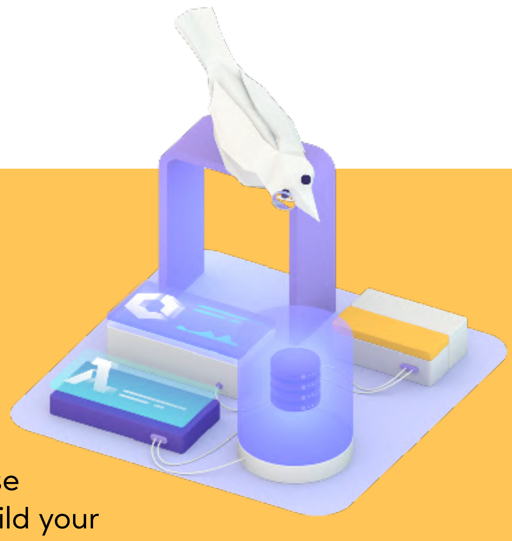
API Gateway and AppSync connections are encrypted in transit by default, but you should keep in mind that the URL might not be, so don't put private information into the path or query string. The AWS API services also support selective access logs. **They should be configured in a way that they don't log sensitive data.**

Encryption at rest should be enabled for DynamoDB and S3 if sensitive data is stored.

SEC 3: How do you implement Application Security in your workload?

You should always validate and sanitize all data from the outside and scan your code for vulnerabilities like you would with non-serverless systems, nothing special here.

And don't forget to store the credentials to third-party APIs in the AWS Secrets Manager, so they're always encrypted at rest.



Security Pillar summary

The WAF and the SAL are an exciting collection of whitepapers to read when building serverless applications. Not only when you design these systems but also after you release them to production. It's a dense material, and often some things get clearer after you build your application. It would help if you also looked into Well-Architected Tool, which is a bit more interactive in questioning you about your system, which can also help because sometimes we like to skip things we feel uncomfortable with.

This article talked a bit more about the SEC pillar and what it's all about. **Authenticate your users and services, encrypt private data, and respond accordingly when anything was breached.** Always keep the principle of least privilege in mind when you set permissions; this goes for everything, not just your public-facing API Gateway. Security has to be applied on all layers of your architecture; if your front door breaks, you don't want attackers to storm right down to the database.

The Operational Excellence Pillar

The OPS and the Security pillar (SEC) form the core of the AWS Well-Architected framework. **The OPS pillar is a catalyst for the other five pillars** because it's mostly about automation in the development and deployment process.

The basic idea of the OPS pillar is that the fewer tasks humans have to do in a software project, the fewer errors will happen, and **the faster you will be able to react and change things**. For example, if you have a static code analysis inside your continuous integration (CI) pipeline, many bugs can be found automatically. **This enhances security**. But if you don't have a CI pipeline, you would have to **manually run the analysis**, which could be forgotten or skipped.

Let's look into the four parts that make up the OPS pillar.

1. Organization

The people working on a software system need to understand why they do it. Why is it that this system is created, and what is their role in the whole picture? Who are the (internal/external) stakeholders? What do they want, what are the security threats, what does the law require them to do? **All these things have to be evaluated**; otherwise, it's like running around in the dark.

For small teams, this often leads to every single person in the team taking on multiple duties. Still, if the software has many parts that are all worked on by multiple teams, things need to be split up, so everyone knows their responsibility to work effectively. For this, the AWS Well-Architected Framework offers Operation Models, which explain how a system can be effectively split up between teams.

2. Prepare

First and foremost, **keep your team prepared** and **their skills up-to-date**. If the people working on the software aren't used to learn new best practices and experiment with new things, they will get into a rut. And getting out of habits is very expensive for companies. You need to take into account the **technical requirements of your system when you plan it**. You should plan the architecture with monitoring in mind to know what's happening before and after it was deployed.

Keep your team prepared and their skills up-to-date.

You should also **design the system with infrastructure as code** right from the start, so your developers can change everything with code, not just your actual software but also the services it runs on.

Being ready for failure is imperative too. If you can partially roll-out features and quickly roll back if something goes wrong, you are much more inclined to try new things to give an edge in the future.

3. Operate

To operate your system effectively, you need to **measure metrics on different abstraction levels**. For example, how good does the system itself perform? Can it handle enough requests? But also, how good do your teams perform? Can they deliver improvements in an efficient timeframe?

You need to define key performance indicators for your system's important metrics and your teams to give you a feeling about what they can do.

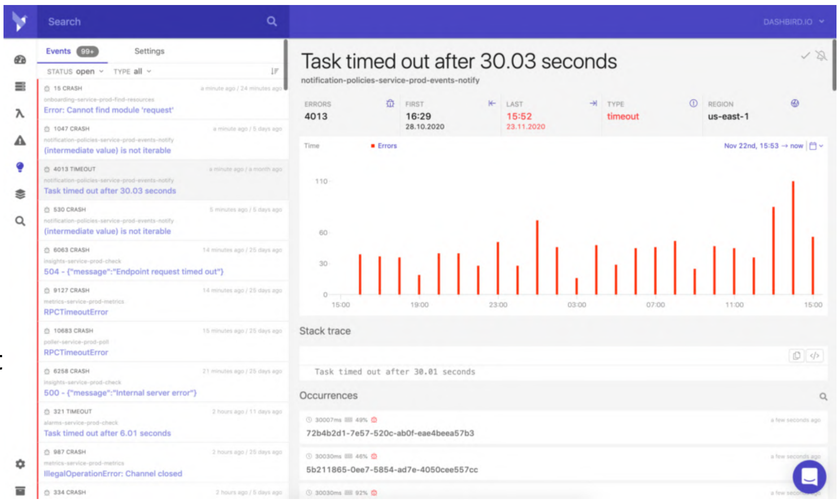
You should also be able to **react to planned or unplanned events** that change your system's conditions. Can your system handle more traffic you expect from an ad campaign? And more importantly, can it handle the increased traffic that hit you unexpectedly?

4. Evolve

You should **define processes that allow you to evaluate and apply changes to your system**. You want to improve on many levels, deliver new features, improve performance. If things go wrong, you also need a way to include changes that prevent known errors from happening again.

Stay aligned with the Operational Excellence pillar

To maintain operational excellence, it's important to know **exactly what's going on in your application at all times** and **constantly improve weak points in your application**. With one of our core features being built on top of the Operational Excellence principles, Dashbird helps serverless users do just that in order to stay aligned with the AWS Well-Architected Framework's best practices.



15 CRASH — a minute ago / 24 minutes ago
onboarding-service-prod-find-resources
Error: Cannot find module 'request'

1047 CRASH — a minute ago / 5 days ago
notification-policies-service-prod-events-notify
(intermediate value) is not iterable

4013 TIMEOUT — a minute ago / a month ago
notification-policies-service-prod-events-notify
Task timed out after 30.03 seconds

530 CRASH — 5 minutes ago / 5 days ago
notification-policies-service-prod-events-notify
(intermediate value) is not iterable

6063 CRASH — 14 minutes ago / 25 days ago
insights-service-prod-check
504 - {"message":"Endpoint request timed out"}

9127 CRASH — 14 minutes ago / 25 days ago
metrics-service-prod-metrics
RPCTimeoutError

10683 CRASH — 15 minutes ago / 25 days ago
poller-service-prod-poll
RPCTimeoutError

6258 CRASH — 21 minutes ago / 25 days ago
insights-service-prod-check
500 - {"message":"Internal server error"}

321 TIMEOUT — 2 hours ago / 11 days ago
alarms-service-prod-check
Task timed out after 6.01 seconds

987 CRASH — 2 hours ago / 25 days ago
metrics-service-prod-metrics
IllegalOperationError: Channel closed

334 CRASH — 2 hours ago / 5 days ago

Task timed out after 30.03 seconds
notification-policies-service-prod-events-notify

ERRORS	FIRST	LAST	TYPE	REGION
4013	16:29 28.10.2020	15:52 23.11.2020	timeout	us-east-1

Stack trace

```
Task timed out after 30.01 seconds
```

Occurrences

30007ms 49% — a few seconds ago
72b4b2d1-7e57-520c-ab0f-eae4beea57b3

30030ms 46% — a few seconds ago
5b211865-0ee7-5854-ad7e-4050cee557cc

30030ms 92% — a few seconds ago

As opposed to a traditional environment, a serverless infrastructure can encompass thousands of resources, using tens of different services, which makes monitoring and alerting much more complex and noisy.

Dashbird doesn't just show users raw data but **automatically translates all their data output into actionable Well-Architected insights** – including warnings and critical errors, which users will be alerted of immediately. This way, users can debug, optimize, and Well-Architect their serverless environments before and after deployment.

SAL Questions for the Operational Excellence Pillar

The SAL asks two questions about the operational excellence of your serverless applications. Let's look at how these can be answered.

OPS 1: How do you understand the health of your Serverless application?

When you build serverless applications, you end up with many managed services; **the serverless approach dials the microservice architecture to the max**. They either integrate directly or can be glued together with Lambda functions. In a complex application, this can get messy pretty quickly. You need to implement some kind of monitoring to get insights into all these moving parts.

A **centralized and structured way of logging is the way to go** here. If your authentication, data storage, and API are separate services, why not monitoring? The structure your logs follow should include some kind of tracing identifier. This way, all log entries that belong to one task handled by multiple services can be correlated in the end. If your computations 10 layers down fail, you know from what API request it came.

You also have to **define metrics that should be measured**. AWS services come with a bunch of predefined system metrics out-of-the-box that get logged to CloudWatch. Still, you should also think about business metrics (i.e., orders placed), UX metrics (i.e., time to check out), and operational metrics (i.e., open issues). It doesn't help that all your Lambda functions run at maximum efficiency if a user has to do 50 clicks to buy a pair of socks. If you don't define crucial metrics, how do you know when to alert someone in your team that things are going wrong?

OPS 1: Dashbird's answer

Observability and AWS Well-Architected insights!

Understanding how your application is running, gathering insights, and discovering opportunities for performance and cost optimization are keys.

Dashbird lets you drill into your data on:

- **Accounts and Microservices Level**, which gives you an instant understanding of trends for overall application health, the most concerning areas as well as cost and activity metrics.
- **Resource Level**, which lets you dig deeper into a specific resource and shows you anomalies, and past and present errors in order to improve and better align with best practices.
- **Execution Level**, which allows users to source full activity details like duration, memory usage, and start and end times for issues and optimization. Going deeper though, we can look at the profile of the execution; requests to other resources, how long it took, and its level of success. We can also detect retries and cold starts here.

For true operational excellence, **monitoring needs to be paired with a good serverless alerting strategy**. Failures and errors are inevitable and so reducing the time to discover and fix is imperative. Monitoring needs to be constant with preemptive checks continuously running for security, best practice, cost, and performance.

However, we also need to be able to filter log events for errors and failures; **this is the first step in the alerting strategy**. Read more about setting up a winning alerting strategy for operational excellence.

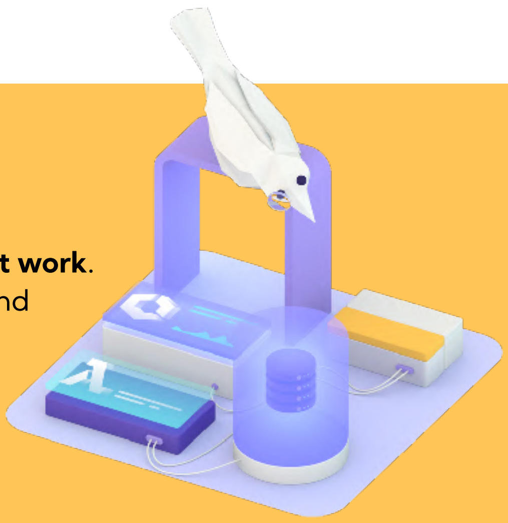
OPS 2: How do you approach application lifecycle management?

When you answered the last question sufficiently, you're able to start **implementing the actual system**. The imperative here is:

Always automate.

If you build a new subsystem, you should prototype it with infrastructure as code (IaC) in a separate AWS account. The IaC tool allows you to replicate your new system with a click in a production account later. **Using different accounts gives you more wiggle room with the limits on AWS account and limits the blast radius if something goes wrong**. IaC can also be versioned, which makes keeping track of changes quite a bit simpler.

With CodePipeline and CodeBuild AWS also offers their own CI/CD tools that can package, test, and deploy your systems directly into different AWS accounts.



Operational Excellency Pillar Summary

The OPS pillar is all about **enabling your team to do their best work**. It focuses on keeping your team members' skills up to date and everyone knowing what's expected of the system they build. This doesn't just mean they should know about the features they have to implement and how to think about automation and security.

You should always use IaC to manage your system so that you can experiment quickly in different accounts.

Don't forget about monitoring! Cloud services, and serverless services, in particular, often seem like a black box to many developers. "If I can't run and debug it on my machine, how should I debug it at all?" But with monitoring solutions like Dashbird, you can always stay on top of things, whether it's about bugs, changes in user engagement, or changes to your deployments.

The Reliability Pillar

Unlike the Operational Excellence (OPS) and Security (SEC) pillars, the REL pillar is **tradable**. You can trade its goals for getting more out of the remaining two pillars: the Cost Optimization (COST) and the Performance Efficiency (PERF) pillars.

Trading means that **you don't have to go all-in in every one of these pillars**. Maybe you want to save money, so you don't do global replications, which would make your system more reliable but also more expensive.

The same goes for the PERF pillar; maybe you want to be as reliable as possible, this can imply that you wait for eventually consistent data storage to do its thing before you respond to a client, which makes your system more reliable in terms of a crash, but also slower in terms of performance.

The three parts that make up the REL pillar:

1. Foundations

The foundation of the REL pillar is the **knowledge of quotas and constraints of the services you use**. If you make a system unreliable because of a bug, that's one thing, but if you didn't know that a service is eventually consistent, you have a greater problem. This is also true for forgetting that you can only send a specific amount of requests per time frame to a service.

Luckily, for AWS services, some tools can help with that. The AWS Service Quotas Console can give you **insights** about each AWS service and even **notify you** when your systems hit the limits of the services they're using. The AWS Trusted Advisor could also help to find out how much of a service you already used.

Managing Foundations

Dashbird integrates with the majority of the popular managed services in AWS to **provide alerting and warning notifications** for when the usage of a service reaches any sort of limits, such as **timeouts, throttling, out of memory,** and the like.

In addition, developers can implement **custom** alarms and **policies** for use cases **specific to their environment**. Moreover, the platform **visualizes the limits** of services to grasp the state of resource usage easily and **understand the capacity** and **long-term threats** to the system.

2. Change Management

Change management is the **anticipation of changes to your serverless system**. This means how customers are changing their system's usage patterns and how you change your system in terms of code.

Examples of this are **traffic spikes**, which are usually handled automatically by a serverless system because it can scale out automatically. Still, change management also includes **new features** you want to deploy or migrations when you change databases.

Staying on top of Change Management

Dashbird gives engineering teams **confidence and the ability to iterate quickly.** A large factor in this is the **reduction of the time it takes to detect and respond to incidents**.

Another topic that Dashbird helps with is getting **real-time visibility** into the inner workings of serverless applications. Developers can use this functionality to **monitor the service at critical times** and **measure the performance**, **cost**, and **quality impact** of system changes.

3. Failure Management

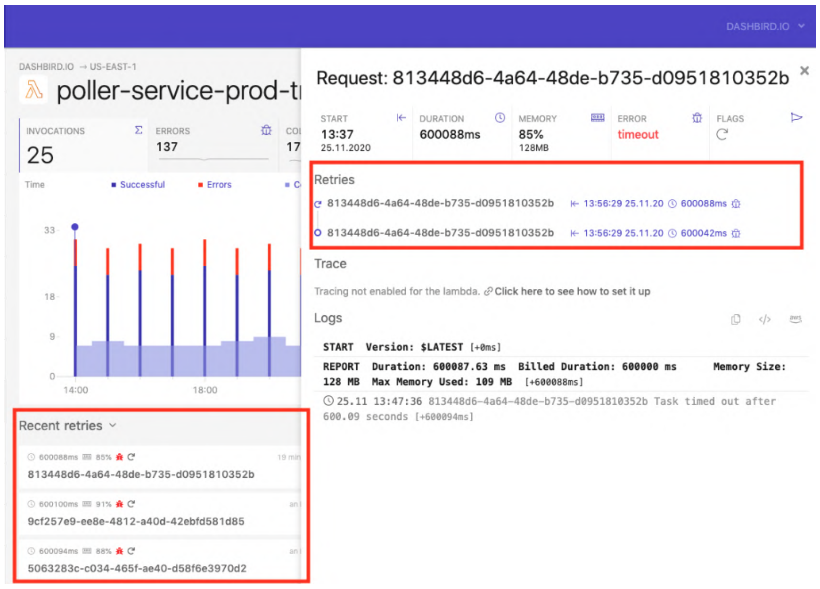
Failure management is about **what you do when things fail**, and they will fail because nothing is forever. Serverless services, especially managed ones, provide much of the failure management, for low-level issues, out-of-the-box, but this doesn't mean that everything will keep working indefinitely.

Serverless systems are often **event-based** and utilize **asynchronous communication** rather heavily. In essence, this means if you send a request to an API, it might not respond with the actual result but just tells you that it accepted your request and will now start to process it. Now, if something goes wrong along the way, **you have no direct way of finding out in the client that sent the request**.

To make sure nothing gets lost, you need to **keep track of your events**. Implement retry logic for your Lambda functions with dead-letter queues and log what went wrong.

Staying on top of failures

Dashbird helps you monitor SQS queues and provides functionality to set alarms for DLQs.



Request: 813448d6-4a64-48de-b735-d0951810352b

START	DURATION	MEMORY	ERROR
13:37 25.11.2020	600088ms	85% 128MB	timeout

Retries

813448d6-4a64-48de-b735-d0951810352b 13:56:29 25.11.20 600088ms
813448d6-4a64-48de-b735-d0951810352b 13:56:29 25.11.20 600042ms

Tracing not enabled for the lambda. Click here to see how to set it up

```
START  Version: $LATEST [+0ms]
REPORT  Duration: 600087.63 ms  Billed Duration: 600000 ms      Memory Size:
128 MB  Max Memory Used: 109 MB  [+600088ms]
25.11 13:47:36 813448d6-4a64-48de-b735-d0951810352b Task timed out after
600.09 seconds [+600094ms]
```

Recent retries

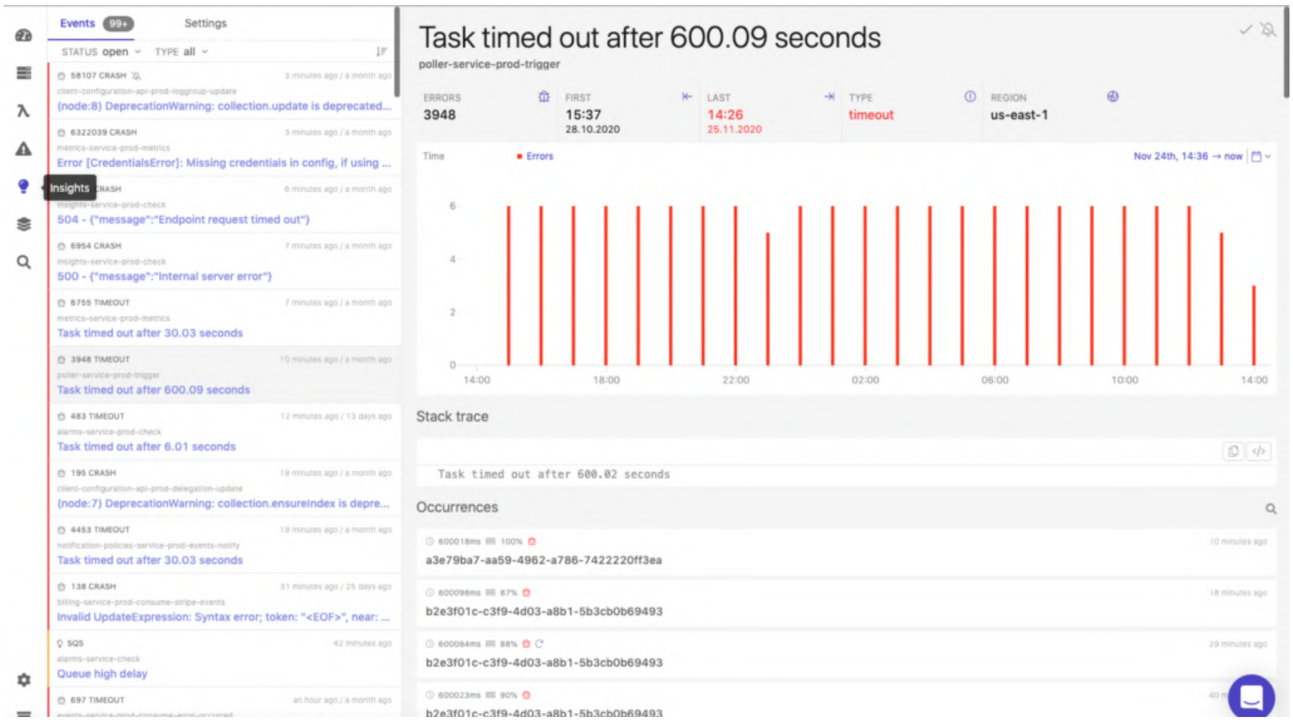
600088ms 85% — 19 min
813448d6-4a64-48de-b735-d0951810352b

600100ms 91% — an
9cf257e9-ee8e-4812-a40d-42ebfd581d85

600094ms 88% — an
5063283c-c034-465f-ae40-d58f6e3970d2

Maintaining reliability

A **serverless developer** needs a tool that **automatically monitors for known and unknown failures across all managed services**. Dashbird platform provides engineering organisations with **end-to-end visibility** into all monitoring data across cloud-native services (logs, metrics and traces in one place) combined with an **automatic failure detection** functionality, identifying know and unknown failures as soon as they happen.



```
58107 CRASH                                   3 minutes ago / a month ago
client-configuration-api-prod-loggroup-update
(node:8) DeprecationWarning: collection.update is deprecated...

6322039 CRASH                                 3 minutes ago / a month ago
metrics-service-prod-metrics
Error [CredentialsError]: Missing credentials in config, if using ...

CRASH                                         6 minutes ago / a month ago
insights-service-prod-check
504 - {"message":"Endpoint request timed out"}

6954 CRASH                                    7 minutes ago / a month ago
insights-service-prod-check
500 - {"message":"Internal server error"}

6755 TIMEOUT                                  7 minutes ago / a month ago
metrics-service-prod-metrics
Task timed out after 30.03 seconds

3948 TIMEOUT                                  10 minutes ago / a month ago
poller-service-prod-trigger
Task timed out after 600.09 seconds

483 TIMEOUT                                   12 minutes ago / 13 days ago
alarms-service-prod-check
Task timed out after 6.01 seconds

195 CRASH                                     19 minutes ago / a month ago
client-configuration-api-prod-delegation-update
(node:7) DeprecationWarning: collection.ensureIndex is depre...

4453 TIMEOUT                                  19 minutes ago / a month ago
notification-policies-service-prod-events-notify
Task timed out after 30.03 seconds

138 CRASH                                     31 minutes ago / 25 days ago
billing-service-prod-consume-stripe-events
Invalid UpdateExpression: Syntax error; token: "<EOF>", near: ...

SQS                                           42 minutes ago
alarms-service-check
Queue high delay

697 TIMEOUT                                   an hour ago / a month ago
events-service-prod-consume-error-occurred
```

Task timed out after 600.09 seconds
poller-service-prod-trigger

ERRORS	FIRST	LAST	TYPE	REGION
3948	15:37 28.10.2020	14:26 25.11.2020	timeout	us-east-1

```
Task timed out after 600.02 seconds
```

```
600018ms  100%                                10 minutes ago
a3e79ba7-aa59-4962-a786-7422220ff3ea

600098ms  87%                                 18 minutes ago
b2e3f01c-c3f9-4d03-a8b1-5b3cb0b69493

600084ms  88%                                 29 minutes ago
b2e3f01c-c3f9-4d03-a8b1-5b3cb0b69493

600023ms  90%                                 40
b2e3f01c-c3f9-4d03-a8b1-5b3cb0b69493
```

SAL Questions for the Reliability Pillar

There are two serverless related questions about the REL pillar in the SAL. Let's look into them.

REL 1: How are you regulating inbound request rates?

Your serverless applications will have some kind of entry point, a front door, so to say, where all external data comes into your system. AWS offers different services to facilitate this, one is API Gateway, and another one is AppSync.

These services, like all the other services you'll be using downstream, **have their limits**. It can lead to reliability issues if you rely on these limits alone. If your system gets sufficiently complex, **it's not easy to calculate what service will fold first**.

That's why you should **set up adequate throttling for API Gateway** and **AppSync**. These services also allow defining usage plans for issued API keys; that way, you can clearly communicate how much a customer can expect from your system.

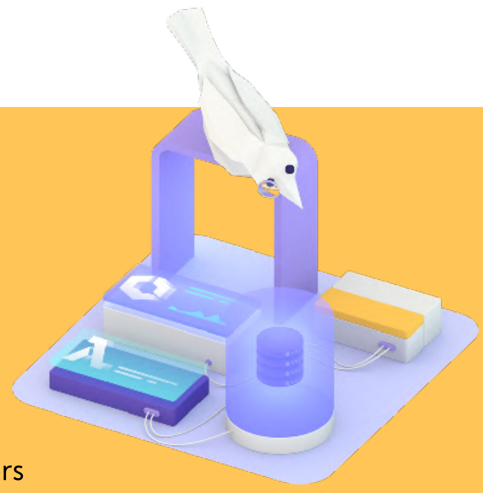
It's also crucial to use concurrency controls of Lambda because it can scale faster than most services. If you integrate with a non-serverless service and suddenly your Lambda function scales up to thousands of concurrent invocations, it will be like a distributed denial-of-service (DDoS) attack.

REL 2: How are you building resiliency into your serverless application?

The main lever for increasing resiliency is decoupling of logic and responsibility between resources and designing the system to handle failures on its own. In most use cases, as much as possible should be made asynchronous.

This is a great post outlining the design principles for building resilience into serverless applications.

In addition to system design, it's important to have tools and processes to measure and track system activity and to get notified on unexpected events in reasonable time windows. No system will be 100% resilient and have the ability to recover from any failure. Engineering teams building on serverless should be responsible for testing their system with different failure scenarios and make continuous improvements and modifications, constantly learn from past incidents and thrive to develop the most optimal processes and tooling to respond to incidents.



The Reliability Pillar Summary

The REL pillar is all about **designing your system in a way that won't break down**. Learn about the services quotas and limits. Sometimes a service sounds like just what you need before reading that it can't handle more than 1000 requests per second. Throttle your systems entry-points so clients can't overload downstream services and give customers clear answers on what they can expect from your system.

Also, **keep everything monitored.** The inherent asynchronicity of serverless systems makes them less straightforward to debug when something has gone wrong; this means **you need a way to get notified when things go out of bounds so you can react quickly**. This also means you need logging data to evaluate what has gone wrong after an incident.

The Cost Optimization Pillar

The COST pillar concerns itself with the **money you spend on your cloud infrastructure**. It's important to think about your system's cost because, in reality, the perfect system won't be used simply because it's too expensive.

That means you have to analyze your security, performance, and reliability requirements to design a system that might not be 100% effective. Still, it's as efficient as possible, making the costs of it bearable for day to day business.

This means the COST pillar is also **tradable for more performance and reliability**. Can your users wait a few hundred milliseconds longer for the result when they only have to pay a tenth of the price? Do your customers want to pay a fortune for the fact that your system is offline only two hours a year, or isn't this worth their money, and they are okay with one-hour downtime every three months?

The COST pillar consists of five parts. Let's look into them.

1. Practice Cloud Financial Management

The first step to cloud financial management is **establishing a cost optimization function** at your company—someone or a team that spends dedicated time **analyzing costs and proposing optimizations**.

Next, you should have financial and technical leads at your company work more closely together. Serverless technology enables engineers to make better predictions on **how much a customer's action will actually cost**, allowing financial personnel to make better pricing decisions.

Make sure your processes and company culture are aligned with costs when choosing AWS services; this way, an otherwise successful project won't fail in the end because it's too expensive to operate.

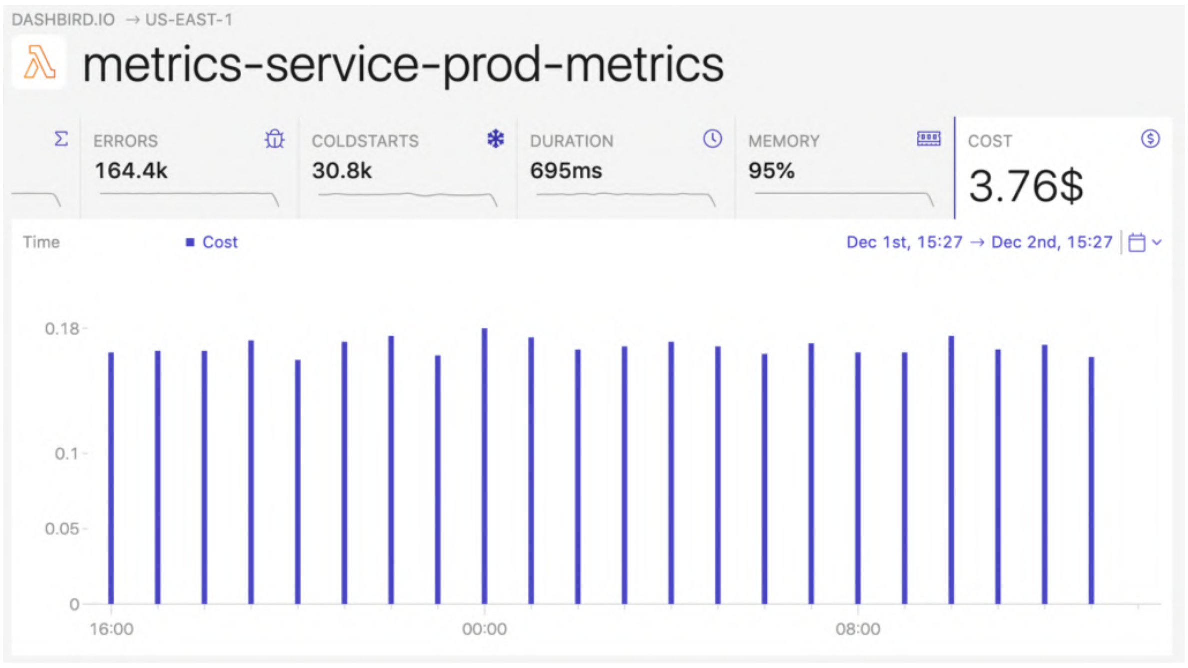
AWS also helps with tools that forecast the costs of your cloud infrastructure and you can use Dashbird's Lambda cost calculator if you're just starting out. Additionally, Dashbird has a built-in cost tracking feature helping you understand the per-resource cost and changes in costs across your serverless infrastructure.

2. Expenditure and Usage Awareness

It's crucial to get a feeling for **what you spent** and **what is actually used in your system**. Luckily this is often a rather easy mathematical function for serverless technology. But as your system grows, you have to **think about account structure**. At least have one management account with one member account. This way, you can **centralize the billing**, making it easier to find out what could be optimized.

Understanding the detailed level of cost across your AWS managed services stack is key to start identifying optimization opportunities and making changes to drive better unit economics for your business.

With Dashbird, an organization can understand what parts of the applications are costing the most and what exactly is driving the cost at a low level (such as latency of Lambda executions or overprovisioning of resources). After the cost breakdown has been identified, it is then easy to modify the application to be at the most optimal configuration between cost, performance, and reliability.



Optimization targets are also required **if you want to stay competitive**. Make it your team's goal to **improve efficiency at least every six months**. This can mean decommissioning resources that aren't used anymore, which **isn't much of a problem for serverless systems** because of the on-demand pricing.

Make it your team's goal to improve efficiency at least every six months.

But it also means you should be on the lookout for **new features that might lower your bill**. Either in terms of direct savings like lower invocation times or indirectly by freeing an employee from manual work.

Here's how Blow Ltd, UK's leading on-demand beauty services software, reduced their time to discovery from hours to seconds, freeing up their developers' time from debugging to focus on product development.

3. Cost-Effective Resources

Use the right service for a use-case and use the right types, sizes, and numbers of services to do so. Get your data to S3 Glacier if it isn't used anymore. Think about Lambda configurations and don't leave everything on default; **sometimes, more Lambda resources can lead to shorter invocation times that are cheaper in the end**.

Don't forget data transfer costs. Getting data out of AWS is especially costly; you need to include these costs when using third-party services that extract data out of your AWS infrastructure.

4. Manage Demand and Supplying Resources

You should **match up the demand and supply of your resources**, so you have enough if a spike hits and don't pay for things that aren't used right now. In a serverless system, this is mostly done automatically for you.

Sometimes you have to integrate with non-serverless technology. If your access patterns lead a non-serverless system to scale up, this takes some time, but it also takes some time to scale down again, which you have to pay for even if it's idle later.

5. Optimize Over Time

As already mentioned, you should always be on the lookout for optimization opportunities. Just yesterday (December 1st, 2020), AWS released millisecond-based billing for Lambda. Until then, running under 100ms wouldn't give you any cost savings, so you could save optimization time after you hit that mark. Was it enough to have an off shelve Python script? It could now be much cheaper to let an optimized Rust binary do the work.

It could now be much cheaper to let an optimized Rust binary do the work.

You should take some time to **review new services or new features of existing services** to see if they fit your use-cases even better. That way, no competitor can outrun you just by being cheaper.

SAL Questions for the Cost Optimization Pillar

Well, this time, there is just one question. This is because **serverless technology is priced on-demand**, which doesn't require you to think about what you do with resources you don't use.

COST 1: How do you optimize your costs?

The starting point for any optimisation journey is first getting a detailed understanding of the current situation and the cost structure of the application.

After understanding the situation and being able to derive insights from it, it's good to iterate until you find **what configuration is the cheapest that while still delivering the expected results**. Lambda is now billed by the millisecond, so getting an invocation time of 50ms could be worth your while.

Think about Lambda throttling and **parallel executions**, especially if you don't charge your customers on-demand yourself. If they pay a fixed amount of money per time frame, but you forget that they can scale up to thousands of parallel invocations, you have a serious problem.

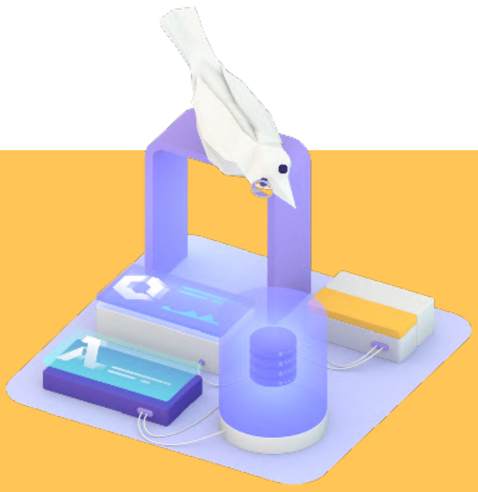
Tag your resources.

Tag your resources, so you know to which project they belong.

Keep logging costs with CloudWatch in mind. Keep your retention periods low if not required otherwise and only log what you need.

Last but not least, **integrate managed services directly**. You don't need a Lambda function to integrate a DynamoDB table with an API Gateway. This also applies to many other AWS services. If all your Lambda does is a transformation from one format to another, it could very well be that you can do the same with a VTL template, which doesn't have cold-starts or additional costs.

With Dashbird, your serverless infrastructure is constantly analyzed for optimization opportunities and opportunities for better decision-making are surfaced in real-time.



The Cost Optimization Pillar Summary

Serverless technology gives you a detailed insight into your costs and many ways to optimize inefficiencies away. You **only pay for what you use**, so if you can get away without a Lambda function and directly integrate two services, you don't have to pay for it.

But as with the other pillars, it's not just about technology. It's also about people. You have to **keep their skills up to date** and **foster a culture of curiosity and cost awareness**. If your team can learn new skills regularly, it can very well be that they get new ideas from the outside without even actively thinking about it. Use-cases that have been too costly before may now be very affordable.

The Performance Efficiency Pillar

Do you deliver the most efficient solution to your customers?

The PERF pillar is all about **using cloud resources efficiently**. This also includes efficient operation if the demand changes.

This is a recurring question you should ask yourself a few times a year. This means selecting the right services for your requirements and thinking about **what those requirements really are**.

If you don't communicate correctly with your customers, it could be that you want to deliver them **something they aren't interested in paying for**. For example, if you can make a service deliver results in under 100ms time, but your customers would be okay to get them in a few seconds, you can save on engineering time and resource cost that such a high-performance system would otherwise require.

The PERF, COST, and REL pillars can be traded for each other to get the best solutions for your business.

Let's look into the four parts that make up the PERF pillar, to understand how to achieve this.

1. Selection

There are many services in the AWS portfolio, so you should take some time to research before using one of them. A focus on serverless architecture removes a huge chunk of services from this list. But sometimes, it can be beneficial to use a non-serverless service for parts of your business. If you need real-time performance, an architecture based on Lambda functions may not cut it, and you have to look into containers.

These services also can be configured in different ways. Lambda alone comes in many sizes. So even if you only focus on serverless technology, think about how each of these services can be used most efficiently.

Maybe you need that DynamoDB Accelerator to meet your business requirement. Maybe, you need a global table to get latency down to an acceptable level. But maybe you don't and can focus your optimization efforts on different aspects of your system.

2. Review

Use infrastructure as code and automated performance tests. This way, you're able to **review new options faster** and see if they **improve efficiency in a meaningful way**. You need well-defined technical and business metrics to measure things. As I said, it's cool if you can deliver something in real-time, but if nobody wants to pay for it, **don't invest time into optimization here**.

Paying for it includes **all costs** here. Not just the operational costs but also the costs of implementing the solution. More often than not, the engineering building the solution can be the highest price point.

3. Monitoring

Always monitor your system; otherwise, you don't really know what it's doing. **This is true for every pillar**. If you build your system based on some assumptions, it could still fail to meet your goals in practice. After all, they were just assumptions.

You have to **gather production data about your efficiency** and try to use this real data for your architecture's next iterations. This might be a sobering experience, especially the first few times you do this, but remember, it's all about getting the right result. It doesn't help anyone but your ego if everyone just thinks you were right.

For small-scale serverless environments, AWS CloudWatch could easily do the job, as it provides just enough data for metrics for invocations, but doesn't have the ability to deep dive into retries, cold starts, memory usage, or cost.

Dashbird was **built to support and strengthen your serverless application** through a single pane on glass view, out-of-the-box error and warnings alerts, and actionable best practice recommendations, **no matter the size of the stack**.

Integrating with your AWS account, Dashbird **takes monitoring a step further** with its **AWS Well-Architected Insights engine** proactively assessing your infrastructure, alerting you of errors, and highlighting optimization opportunities, and thus, helping your environment to always stay aligned with the AWS Well-Architected Framework's pillars.

Dashbird takes monitoring a step further.

4. Trade-Offs

Sometimes you have to make trade-offs in your architecture. And with sometimes, I mean always. Jokes aside, this is what makes engineering work interesting. Maybe you can get away with eventual consistency and deliver much lower latency in turn. But maybe you run a bank, and making your customer's accounts eventually consistent isn't the best course of action here.

Maybe, you're tight on personnel and simply can't afford the complexity a multi-tier caching solution brings, even if the improved performance sounds awesome. **Simplicity also has a business value in the long run**. Always think about what you need and what you can pay for it regarding money, reliability, durability, and complexity.

Read more about how professional serverless teams manage software issues.

SAL Questions for the Performance Efficiency Pillar

Again, like with the COST pillar, there is just one SAL question for the PERF pillar.

PER 1: How have you optimized the performance of your serverless application?

There are a lot of ways to optimize a serverless application. In our experience, the majority of opportunities come from using the **right mix of services** and **best practices** and **avoiding waste through bad architectural design decisions**.

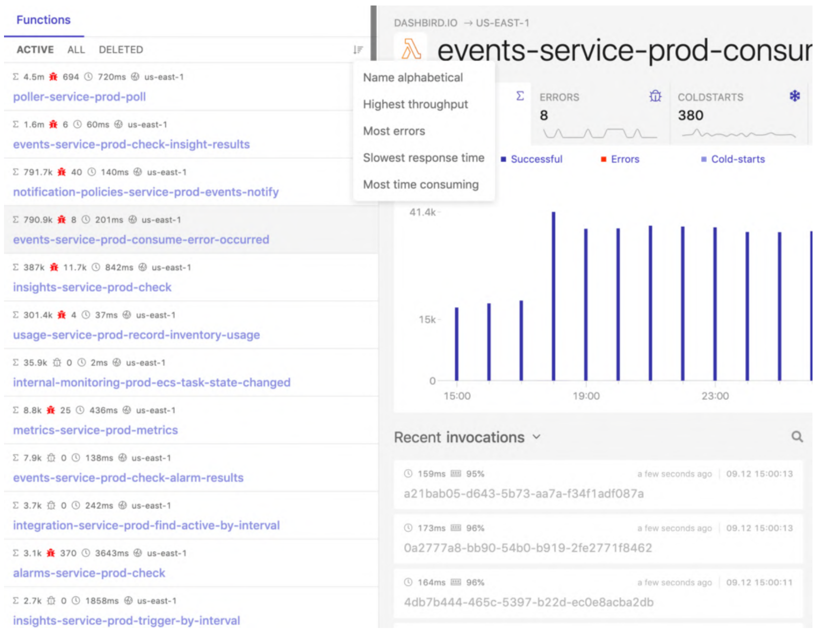
For example, using the right databases and downstream services, not waiting in code, and instead of using Step-Functions to orchestration logic, selecting between API Gateway and ALB for APIS and such. Often, the majority of your serverless cost and performance overhead is accumulated in services other than Lambda functions.

An example of optimizing DynamoDB: depending on how predictable your traffic is, **on-demand or provisioned capacity can be the right choice for your DynamoDB tables**. The DynamoDB Accelerator could also be a way to get out that last bit of performance.

Optimising Lambda functions

If you are optimizing Lambda functions, there are multiple methods to identify the biggest opportunities for increased efficiency. The first thing we recommend doing is **finding the highest latency and highest volume Lambdas**, that affect the user experience the most and focus on them.

Dashbird provides a breakdown of all the functions and their performance in a single pane of class and you can **order your functions based on the highest throughput and lowest response times**.



Σ 4.5m 694 720ms us-east-1
poller-service-prod-poll

Σ 1.6m 6 60ms us-east-1
events-service-prod-check-insight-results

Σ 791.7k 40 140ms us-east-1
notification-policies-service-prod-events-notify

Σ 790.9k 8 201ms us-east-1
events-service-prod-consume-error-occurred

Σ 387k 11.7k 842ms us-east-1
insights-service-prod-check

Σ 301.4k 4 37ms us-east-1
usage-service-prod-record-inventory-usage

Σ 35.9k 0 2ms us-east-1
internal-monitoring-prod-ecs-task-state-changed

Σ 8.8k 25 436ms us-east-1
metrics-service-prod-metrics

Σ 7.9k 0 138ms us-east-1
events-service-prod-check-alarm-results

Σ 3.7k 0 242ms us-east-1
integration-service-prod-find-active-by-interval

Σ 3.1k 370 3643ms us-east-1
alarms-service-prod-check

Σ 2.7k 0 1858ms us-east-1
insights-service-prod-trigger-by-interval

159ms 95% a few seconds ago | 09.12 15:00:13
a21bab05-d643-5b73-aa7a-f34f1adf087a

173ms 96% a few seconds ago | 09.12 15:00:13
0a2777a8-bb90-54b0-b919-2fe2771f8462

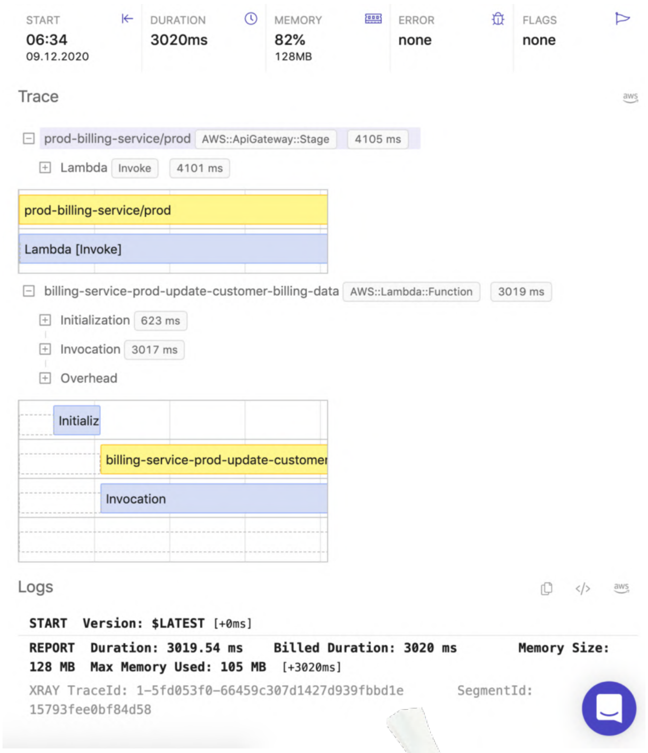
164ms 96% a few seconds ago | 09.12 15:00:11
4db7b444-465c-5397-b22d-ec0e8acba2db

When you have identified a problem in any one of your functions, the next step is to **dive deeper and understand where you're struggling to perform**. Is the function execution very **processing-heavy** and **provisioning more memory** would help speed up the execution? Or maybe it's **waiting** most of its execution time **after a third-party service**, and adding more resources does nothing to speed up the performance?

To understand the breakdown of an individual invocation, Dashbird integrates with X-ray and **shows the list and duration of each individual activity executed within the function**.

Understanding the situation within the execution enables developers to take further action and to either experiment with memory provisioning of an application or **make changes to the services that the function is communicating with** to optimize the performance or cost a specific function.

On top of helping you optimize your functions, Dashbird **analyses the metrics and configurations of other managed services** typically used in a serverless architecture and **offers actionable recommendations** on provisioning, configuration, helping you make **informed decisions about your serverless stack**. Dashbird also **surfaces slow and increased-delay situation** across databases, API Gateways, SQS queues, Kinesis stream, and others.



START 06:34 09.12.2020 | DURATION 3020ms | MEMORY 82% 128MB | ERROR none | FLAGS none

Trace

prod-billing-service/prod AWS::ApiGateway::Stage 4105 ms
Lambda Invoke 4101 ms

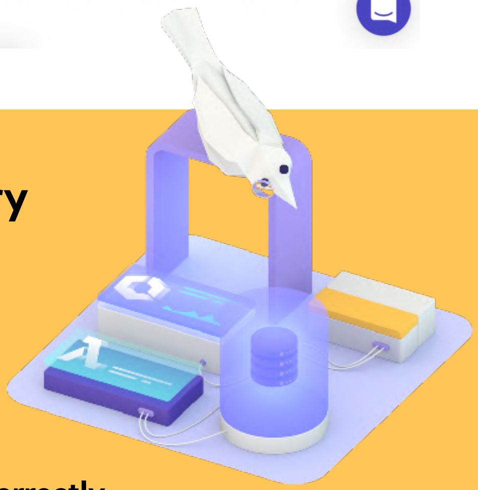
billing-service-prod-update-customer-billing-data AWS::Lambda::Function 3019 ms
Initialization 623 ms
Invocation 3017 ms
Overhead

prod-billing-service/prod
Lambda [Invoke]
Initializ
billing-service-prod-update-customer
Invocation

Logs

```
START  Version: $LATEST [+0ms]
REPORT  Duration: 3019.54 ms    Billed Duration: 3020 ms       Memory Size:
128 MB  Max Memory Used: 105 MB  [+3020ms]
XRAY TraceId: 1-5fd053f0-66459c307d1427d939fbbd1e      SegmentId:
15793fee0bf84d58
```

The Performance Excellency Pillar Summary



The PERF pillar helps you focus on **using your resources most efficiently**. Serverless technology often includes optimizations mentioned in that pillar implicitly, but it's not just about choosing the right services.

The PERF pillar is also about **configuring the services you use correctly**. Think about the memory size of your Lambda functions. Think about the deployment configuration of your API Gateway. And also, think about the capacity of your DynamoDB tables.

Make sure you **define the right metrics when monitoring your system**; this way, you can optimize the parts that matter. Also, review new releases every now and then. AWS releases new services every year and also updates its existing ones. **Sometimes your systems just get faster or cheaper without you doing anything**. For example, during re:Invent 2020, AWS announced strong consistency for S3 without any more cost or configuration, and millisecond-based pricing for AWS Lambda; your existing apps profit from that without you doing anything. But sometimes, you need to explicitly configure a service differently to get the benefits, like with the new 10GB memory size of Lambda.

And finally, remember that everything has a trade-off. Lambda can scale to thousands of parallel invocations in a matter of seconds, but it could be that your system needs more than 15 minutes of invocation times to do its work.

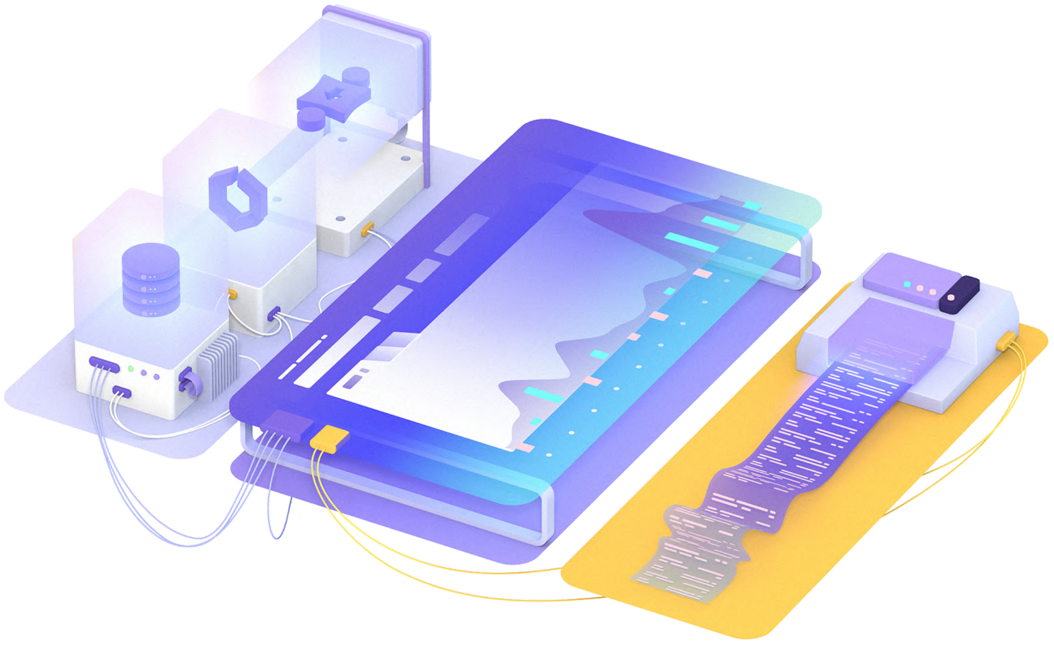
Wrapping up the Well-Architected Framework

The WAF whitepapers are a good source of ideas to start improving the systems you build on AWS. They are very dense, so you might not understand everything right from the start, but you can always revisit them if you have the time or need to solve technical issues.Sometimes the examples from the whitepapers only make sense if you tried to build a system by yourself.

Or you can skip all that and let Dashbird's Well Architected actionable insights, warnings and errors engine tell you exactly what's going on in your system, what needs improvement, and where exactly, what's about to break and what's broken. Find out more or book a call – we love taking serverless and Well Architected.

If you're building your architecture based on serverless technology, you will follow most of the advice given in the WAF pillars implicitly, and if not, the SAL focuses on the things that serverless technology might not solve for you out-of-the-box.

If you're still curious to learn more about the WAF, how it came about and some more best practices for each of the five pillars, you can watch our recent session with Tim Robinson (AWS) here.



dashbird.io