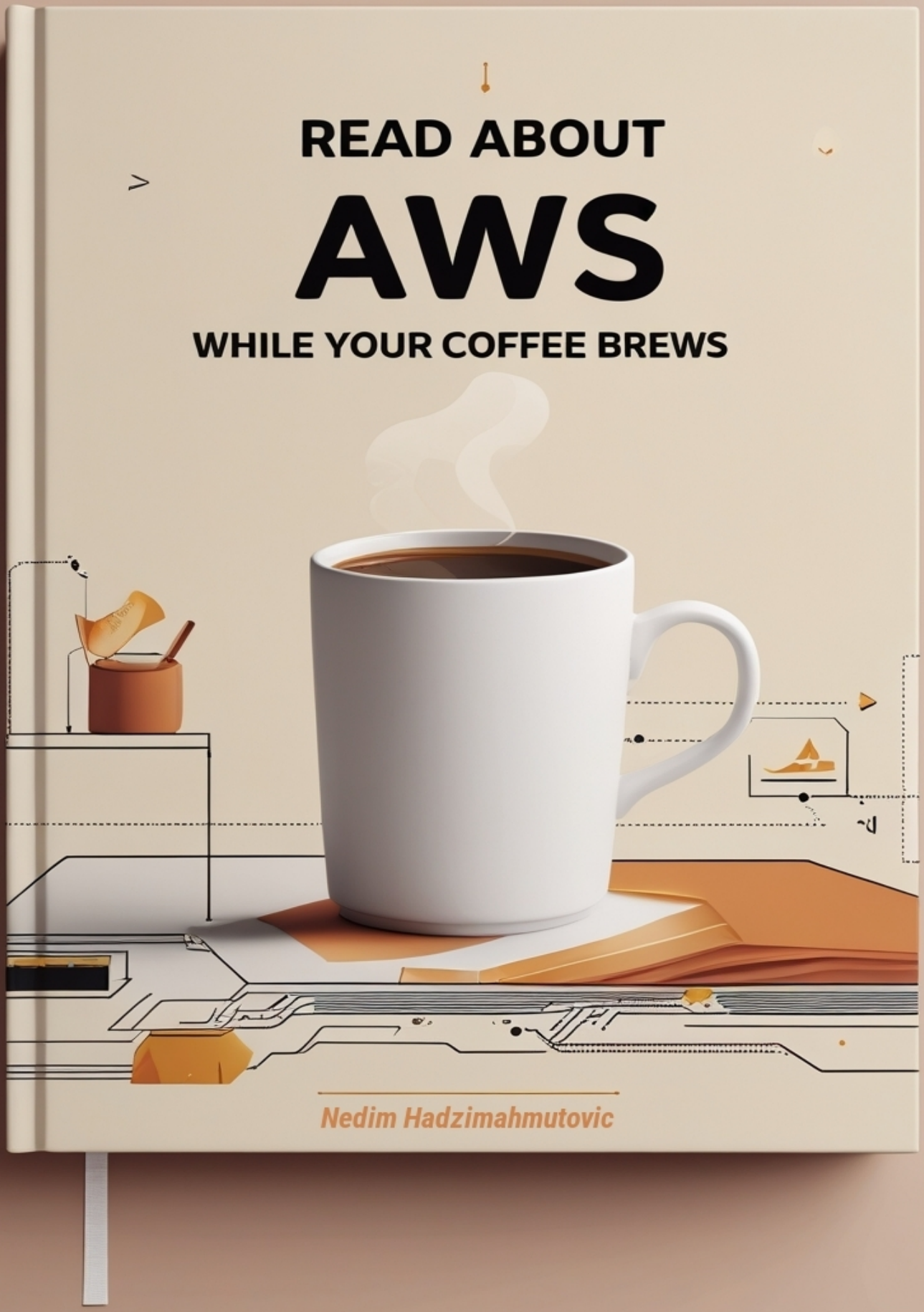
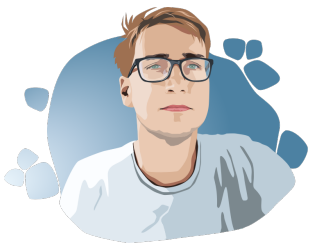

READ ABOUT
AWS
WHILE YOUR COFFEE BREWS
Nedim Hadzimahmutovic

Contents

The Author



With a career continuing nearly two decades, Nedim has reached expertise in Infrastructure Automation. He is an expert in all things DevOps-related. He specializes in the AWS platform, Linux, Terraform, and scripting in Ruby or Python.

Contact

If you'd like to contact me, you can use the links below.

- ***Linkedin***
- ***Gumroad***
- ***X***

Enjoy learning about AWS Cloud and Linux!

Best, Nedim.

Acknowledgments

Thanks to the amazing ClearView.team for supporting my work.

Copyright

Prerequisites

It is required that you setup your AWS credentials and configure `awscli` by yourself. This is covered by AWS very well in **this article**.

The most Flexible AWS Savings Plan

> Date: Mar 11, 2024

AWS provides several ways to reduce your cost but no other program is as flexible as the Compute Saving Plan. This article covers what you need to know to successfully and efficiently use the AWS savings program.

What are Savings Plans

In essence, all AWS cost-saving plans offer cost reductions compared to On-Demand prices, in exchange for a long-term commitment. That commitment is 1 or 3 years long and is an hourly spend commitment type.

AWS provides three types of Savings Plans:

- Compute Savings Plans,
- EC2 Instance Savings Plans, and
- Amazon SageMaker Savings Plans.

> The focus of this article is on the most flexible type of savings plan which is the Compute Savings Plan.

Research, Research, Research

The first step we need to take is to see if there are any Recommendations. This page might tell you what AWS recommends to purchase. Not everyone gets this kind of suggestion as they are based on your usage.

Follow the next example to understand how to purchase a Compute Savings Plan.

Get a Discount on a Single t4g.micro EC2 Instance

This is a straightforward example. In this example, the commitment is for 1 year in Europe (Ireland), the eu-west-1 region.

A very important note is to know that discounts vary from Region to Region so do your research beforehand.

Step 1: Check Your Bill

Find your bill in the AWS console. You can see something like this.

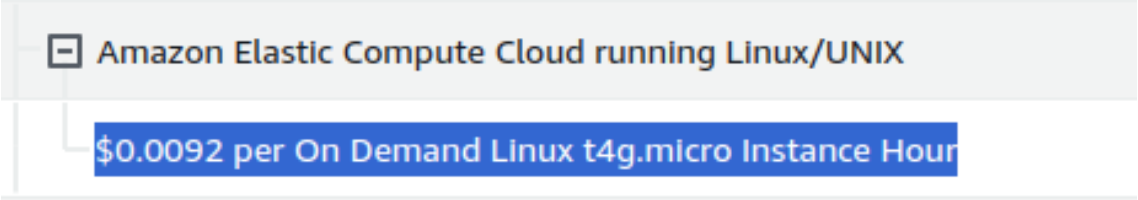


Figure 1: Image showing the AWS Bill.

We identified that we are paying On-Demand prices for a t4g.micro EC2 instance.

Step 2: Identify the Savings Rate in Your Region

Find out in which region your EC2 instance is hosted. After that access the table at Savings Plan Compute Pricing page. That page will show you a table like the one below.

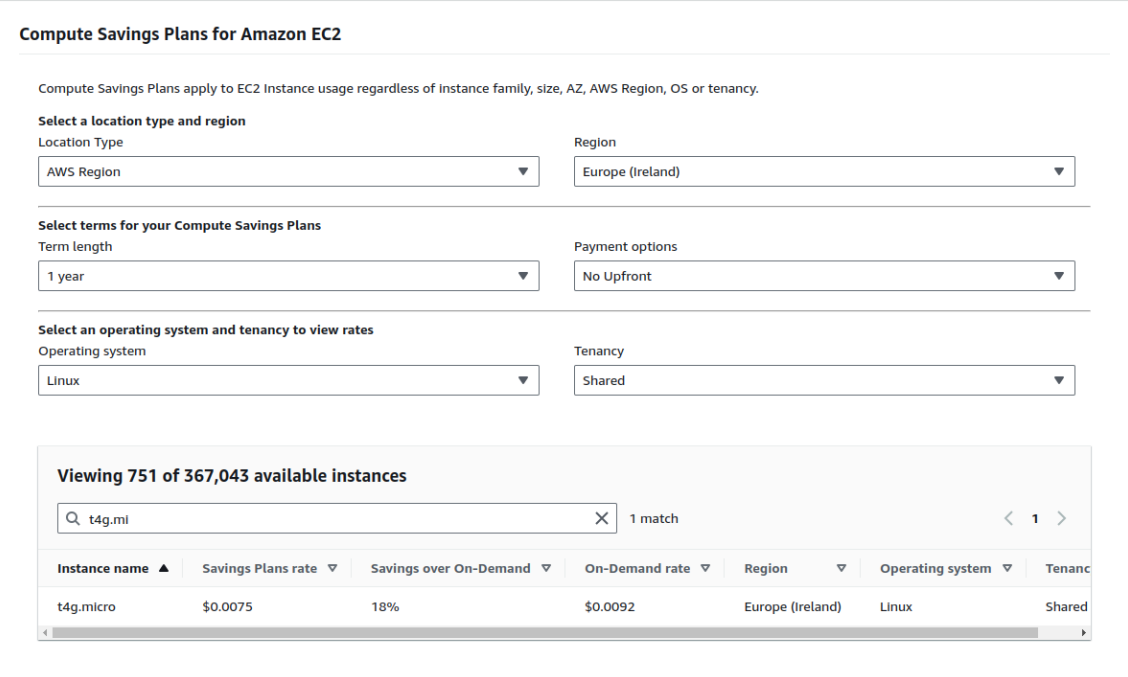


Figure 2: Image showing the AWS Savings Plan Compute Pricing Page screenshot.

The goal is to get a discount on the usage of a t4g.micro EC2 instance.

- **On-Demand rate:** $0.0092 * 720 hours (hours in a 30-day month) = $6.624
- **Savings Plans rate:** $0.0075 * 720 hours (hours in a 30-day month) = $5.62500
- ***Savings over On-Demand — 18%***

> If you commit to a 3-year term the savings will be 42% instead of 18% for the Ireland Region.

Step 3: Purchase a Plan

Therefore, the purchase plan will look like this.

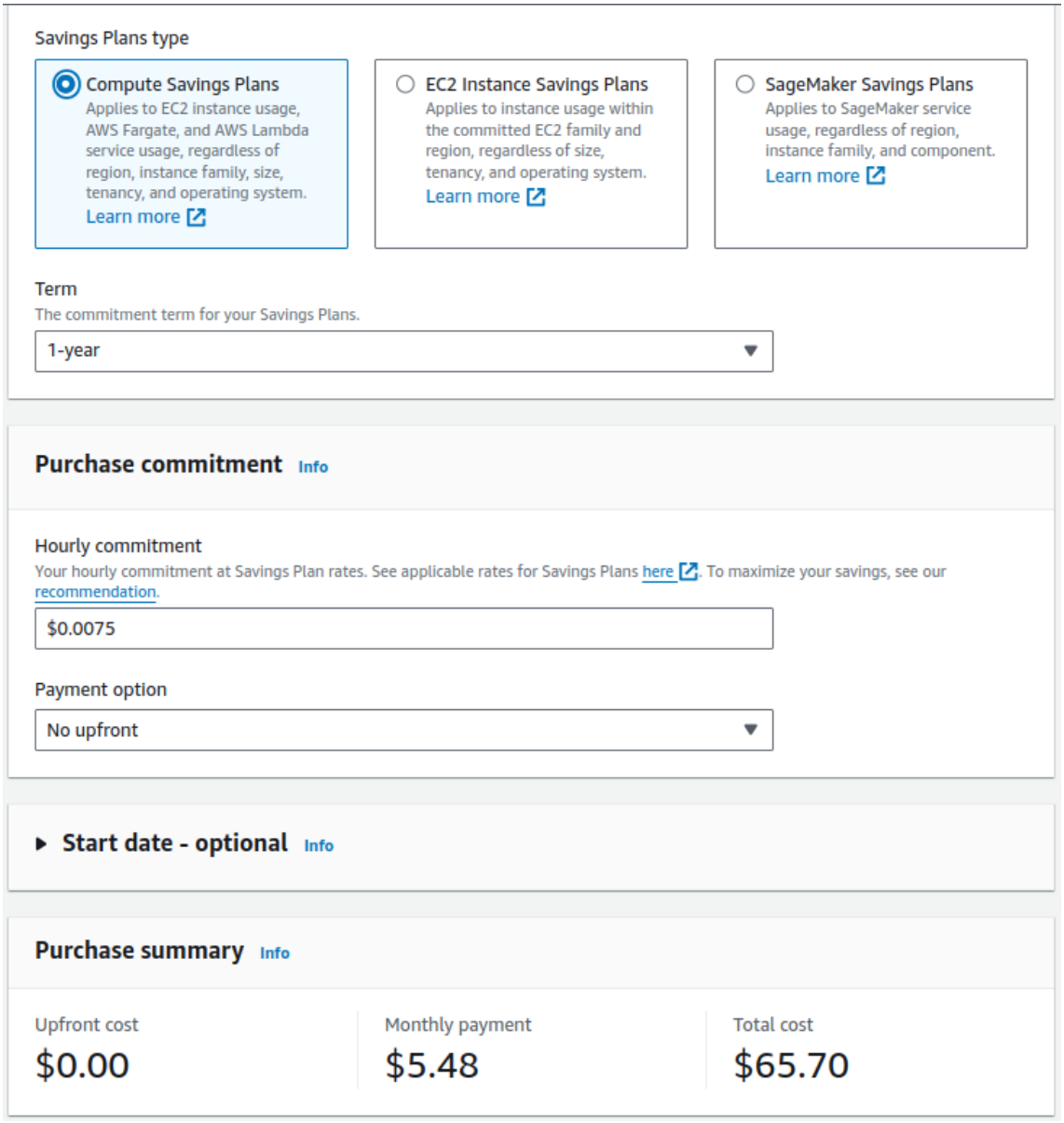


Figure 3: Image showing the AWS Savings Plan Purchase screenshot.

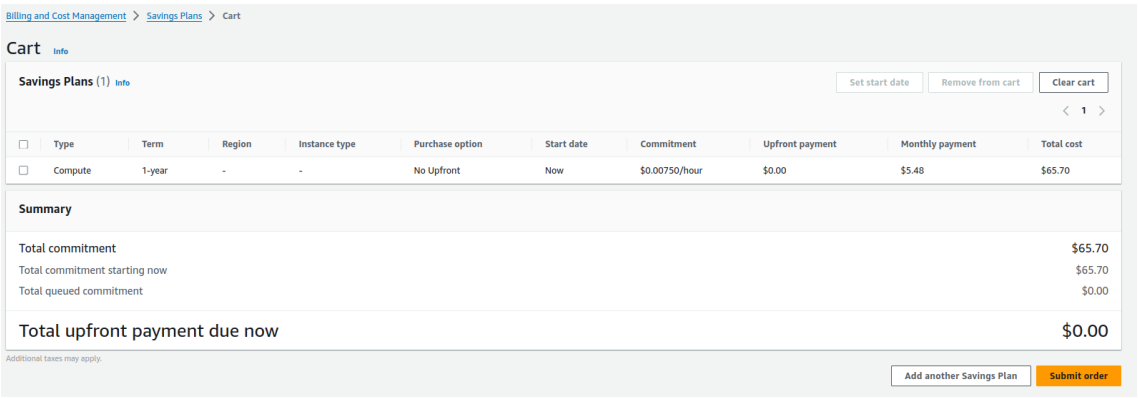


Figure 4: Image showing the AWS Savings Plan Cart screenshot.

Important notes:

- Do not purchase large plans and over-commit.
- Purchase small plans and then monitor the coverage.
- If you need more coverage at a later period in time you can purchase additional savings plans.

> The hourly commitment rate is the one after the discount — the Savings Plans Rate.

Reports Analysis

Utilization Report

Below you can see a sample utilization report for a single day after the savings plan has been purchased.

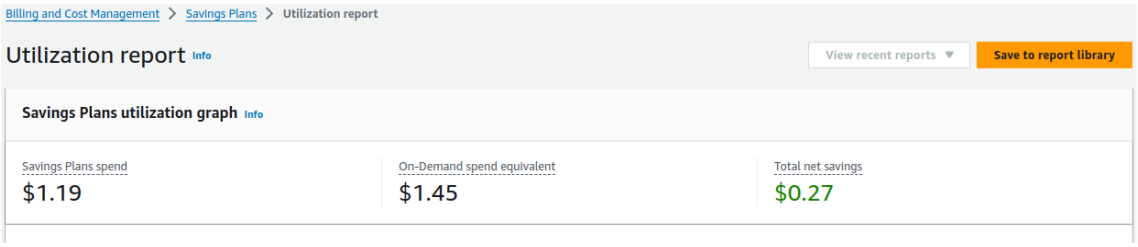


Figure 5: Image showing the AWS Utilization report screenshot.

Coverage Reports

Under the *"Billing and Cost Management / Savings Plan"* menu, you can find the Coverage report. Here you can get information about how much of your spending has been covered by a savings plan during a certain period.

You get information such as:

- Average coverage,
- Potential monthly savings vs On-Demand, and
- On-demand spending not covered.

Before Creating a Savings Plan

This is an example of a Coverage report showing no active Savings plan. It has yet to be activated.

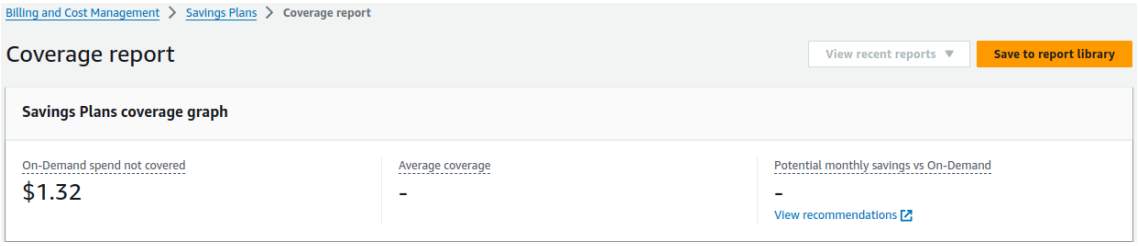


Figure 6: Image showing the AWS Coverage report screenshot.

After Creating a Savings Plan

This is an example of a Coverage report that shows an active Savings plan with a coverage of 43%

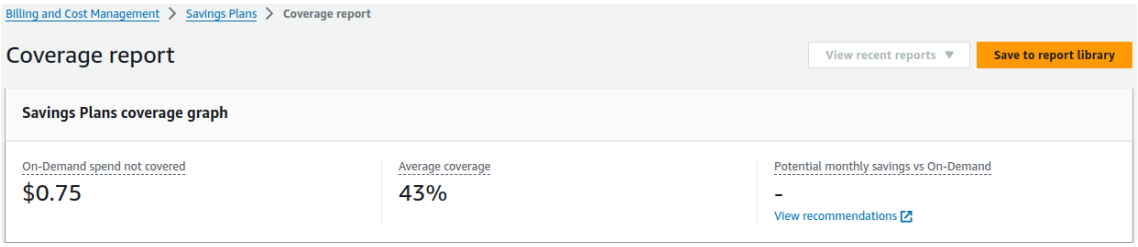


Figure 7: Image showing the AWS Coverage report screenshot.

It is very difficult to get 100% coverage so start with 50% and iterate.

Do Your Research First

As described above Compute Savings Plan represents a flexible savings plan that supports multiple regions, and services and you commit to a dollar amount instead of an instance type or size. But there are dangers as if you calculate wrong then you are stuck paying that committed dollar amount.

If you purchase Savings Plans you commit to a certain dollar amount on a period and you can not cancel. Therefore be careful and do your research first before committing to any kind of Savings Plan.

AWS Graviton Processors: The Future of Cloud Computing

> Date: Feb 3, 2024

The world of cloud computing is changing fast, that is a fact. Some changes have a more significant impact, as was the case with the rise of AWS Graviton processors.

The whole tech industry is experiencing the growing adoption of ARM architecture, so why should it not be the case with the Cloud as well?

This article focuses on the impact of AWS Graviton on the Cloud offering, the shift towards the ARM architecture, and how to help clients jump on the *Graviton* train and save on costs up to 20%.

Introduction

- AWS Graviton processors are becoming popular because of their superior performance and energy efficiency in cloud environments which reduces your carbon footprint,
- Adapting the ARM architecture requires data migration or possible code improvements before benefiting from the superior improvements in both cost and performance.

The Rise of AWS Graviton

Understanding AWS Graviton Processors

AWS Graviton processors are powered by ARM architecture, enabling performance and cost optimizations, which represent a significant advancement in cloud computing, compared to traditional x86 processors.

Graviton processors are optimized for Cloud workloads with the following benefits:

- Large L1 and L2 caches for every virtual central processing unit (vCPU), meaning that most of your workload never burdens RAM,
- Every virtual CPU is a physical core, meaning more isolation between virtual CPUs,
- Cores are connected in a fast mesh with ~2TB/s of bisection bandwidth, which allows applications to move very quickly from core to core,
- Graviton's RAM architecture means you don't need to worry about application memory allocation, or which cores are running the application.

Generations of AWS Graviton

Graviton processors come in various generations with each generation offering various advances in processing power and efficiency.

Graviton

The first generation ARM architecture-based, known as A1 type, Graviton-powered EC2 instances were launched in 2018. Not every region is still providing this first-generation instance type. Use the following `awscli` command to check if a region supports the A1 instance type.

```
aws ec2 describe-instance-type-offerings \
    --location-type "availability-zone" \
    --filters Name=location,Values=us-east-2a \
    --region us-east-2 \
    --query "InstanceTypeOfferings[*].[InstanceType]" \
    --output text \
    --profile default | sort | grep a1
```

This processor features:

- 64-bit ARM Neoverse cores,
- The instances are up to 40% less expensive than the same number of vCPUs and DRAM available in other instance types.

Graviton 2

Launched in 2019 it represents the second generation of AWS Graviton processes. Graviton2-based instance types offer up to 40% better price performance than fifth-generation instances.

Graviton 2 Instance types

We now have 12 instance families (M6g, M6gd, C6g, C6gd, C6gn, R6g, R6gd, T4g, X2gd, Im4gn, Is4gen, and G5g) that are powered by AWS Graviton2 processors that provide significant price performance benefits for a wide range of workloads.

Graviton 3

Launched in 2022, it is the third and latest generation that is generally available. When compared to AWS Graviton2 processors it provides:

- up to 25% better computing performance,
- up to 2x better floating-point performance,
- up to 2x faster crypto performance, and
- up to 3x better ML performance,
- support for `bfloat16`,
- and features the latest DDR5 memory, which provides 50% more memory bandwidth compared to DDR4.
- up to 60% less energy usage for the same performance than comparable EC2 instances.

Graviton 3 instance types

All 7th generation instance types containing “g” means it is based on AWS Graviton 3, the silicon designed by AWS. Examples:

- C7g, C7gd, C7gn: the “C” instance family is designed for compute-intensive workloads,
- M7g, M7gd: “G” instance family is designed for general-purpose workloads with balanced computing, memory, and networking,
- R7g, R7gd: The “R” instance family is designed for memory-intensive workloads.

Graviton 4

Launched in late November 2023 Graviton 4 represents the fourth generation and it is still in preview only, with general availability planned in Q1 of 2024. When compared to Graviton3 processors it provides:

- up to 30% better computing performance,
- 50% more cores,
- 75% more memory bandwidth.

Graviton4 will be available in memory-optimized Amazon EC2 R8g instances, which are currently in preview only. R8g instances offer larger instance sizes with up to 3x more vCPUs and 3x more memory than current generation R7g instances. To learn more about Graviton4-based R8g instances, visit this link.

Switch your AWS Services to Graviton

Here we will describe which services we have switched over to AWS Graviton. Some services can be switched with a few clicks, while others require a full data migration following best practices.

AWS EC2

Following is a tutorial on how we switched EC2 instances. Please note that t4g is run on Graviton 2. Migrating our NodeJScode to be compatible with ARM architecture meant upgrading a few packages only.

Do not forget to set your `awscli` profile as in all of these examples the default one is used.

Find ARM LTS Ubuntu Images

For the latest LTS which is 22.04 LTS (Jammy Jellyfish), currently.

```
aws ec2 describe-images \
    --filters "Name=name,Values=ubuntu/images/hvm-ssd/ubuntu-jammy-22.04-arm64-server-*"
    ↪  "Name=architecture,Values=arm64" \
    --query "sort_by(Images, &CreationDate)[].[Name, ImageId, Architecture]"  \
    --output text \
    --region us-east-1 \
    --profile=default
```

This command will show you the IDs of the images that support the ARM architecture. Choose one and set it at the next step.

Set AMI ID

As the AMI ID has been found in the previous command, we need to save it to a variable.

```
AMI_ID=ami-02ddaf75821f25213
```

List EC2 Key Pairs

The following commands will show keys that were added previously. If you get an empty result you need to add your public SSH key.

```
aws ec2 describe-key-pairs \
  --region us-east-1 \
  --profile=default
```

Find the Availability Zone

The zone is directly connected to the subnet.

```
aws ec2 describe-instance-type-offerings \
--location-type availability-zone \
    --filters Name=instance-type,Values=t4g.small \
    --region us-east-1 \
    --profile=default
```

::note This step is very important as you need to find the what availability zone supports your instance size. :::

Find the Subnet and Security Group

We need to find the correct Subnet and SG.

List Security Groups

```
aws ec2 describe-security-groups \
    --query "SecurityGroups[*].{Name:GroupName,ID:GroupId}" \
    --region us-east-1 \
    --profile=default
```

List subnets

```
aws ec2 describe-subnets \
    --region us-east-1 \
    --profile=default
```

Find the correct subnet that is used by your old EC2 instance.

Create the EC2 Instance

Please note that the following command is an example only and that you need to change the values for:

- the key name,

- the subnet ID,
- the security group ID,
- set your tags,
- use your profile name.

```
aws ec2 run-instances --image-id $AMI_ID --instance-type t4g.small  \
    --key-name nedim \
    --subnet-id subnet-00000000000000 \
    --security-group-ids sg-00000000000000000 \
    --associate-public-ip-address \
    --tag-specifications 'ResourceType=instance,Tags=[{Key=Name,Value=ubuntu-test-graviton}]' \
    --region us-east-1 \
    --profile=default
```

Prepare the image

Connect to the VM, and run the system upgrade.

New Server Configuration with Ansible

Since we used Ansible to set up the old server the process of setting up an identically configured server was a piece of cake. All we had to do was change the hostname in the host inventory file add the new server and rerun the playbooks.

Volume size
Make sure the new disk volume is identical to the size of the old server's disk volume as we will copy over data from the old server to the new one.

Data Migration

Using *RSYNC*, copy over the data from the old server to the new Graviton-based one, using the SSH protocol.

```
rsync -avz -e ssh /var/www/ root@new-server.compute-1.amazonaws.com:/var/www/
```

This data migration does not guarantee that the code will work 100%, you will need to deploy and build the code again to be sure it works as expected. In our case, we needed to upgrade several NodeJS packages.

AWS RDS

With RDS instances no migration is necessary and you can take the easy way of just modifying the instance class of your instances if your database engine version supports it. Determine if the current database version meets the minimum required version for moving to Graviton2.

Versions supporting Graviton:

- MySQL: 8.0.17 and higher,
- PostgreSQL: 12.3, 13 and higher,
- MariaDB: 10.4.13, 10.5 and higher.

If your RDS instance isn't at a version supported by Graviton2, you need to upgrade to a supported version.

AWS ElastiCache for Redis

Migrating Amazon Elasticache for Redis and Memcached to Graviton2-based instances offers a smooth transition, only a change of instance type is required, with a performance improvement of up to 45%, all at a 20% lower cost compared to similar x86-based instances.

The Beginning of a New Age in Cloud Computing

In conclusion, the rise of AWS Graviton and ARM architecture represents a change in cloud computing that impacts the whole IT industry. These technologies bring together improved performance, cost-effectiveness, and energy efficiency. As more organizations adopt these technologies, we can expect a future where the flexibility of ARM architecture and the efficiency of Graviton processors set the standard for cloud infrastructure. Therefore, do not miss the chance to make the switch and be part of the new standard.

AWS AppRunner: The Smart Way to Deploy Containers on The Cloud

> Date: Jan 10, 2024

If you tried to deploy containers on the cloud you have experienced anything but an easy process. It has been easy to deploy containers on your local machine but deploying them on the Cloud has not been an easy process.

That was true until AWS AppRunner was introduced back in 2021.

Great solution for startups

We have found that *AppRunner* can be of very benefit to startup projects as it is cost-effective because of its on-demand pricing model and it is a fully managed service.

This will result in less time spent on setting up the infrastructure which saves us time and money.

An attractive pricing model

What is great about *AppRunner* is that when the container is in an idle state you *only pay for the memory provisioned* in each container instance. When a container is in the idle state it is considered to be a Provisioned container instance. Once the container starts to use CPU resources, it will be considered an Active container instance and you will be charged for the CPU resources used.

> As you can pause and resume the service you can as well additionally save on costs.

In summary

- Provisioned container instances - charged for memory ($0.007 / GB-hour),
- Active container instances - charged for both memory and CPU resources used ($0.064 / vCPU-hour, $0.007 / GB-hour),
- Paused instances- CPU and memory are not charged.

What can AppRunner do?

Since AWS App Runner is a fully managed service it will:

- automatically deploy web applications by detecting a new container image that has been pushed to the container registry,
- automatically scale and load balance to meet the traffic needs.

> Automatic deployment, scaling, and load balancing make it a desirable technology.

Easily Managed by Terraform

By adding Terraform to the mix we can provision this technology even faster. Later in this article, we will provide examples of code on how to set up AppRunner with Terraform.

History

Back when it was introduced it was still an early tech and still not developed and mature but in the last two years, it has grown significantly. Here are some improvements that we have witnessed while using the tech:

- Added support for more regions across the world,
- Reduced duration for deploying applications using container images, which is about a 30–40% reduction in deployment time depending on the container image size,
- Increased instance startup time from one to a maximum of five minutes which enables slower instances that need more time of startup to be used,
- Added dual stack support for incoming traffic through public endpoints, both IPv4 and IPv6 endpoints, simultaneously,
- Reduced the time taken for image-based service deployment,
- Added support for immediate deployment failure if App Runner couldn’t pull an image.

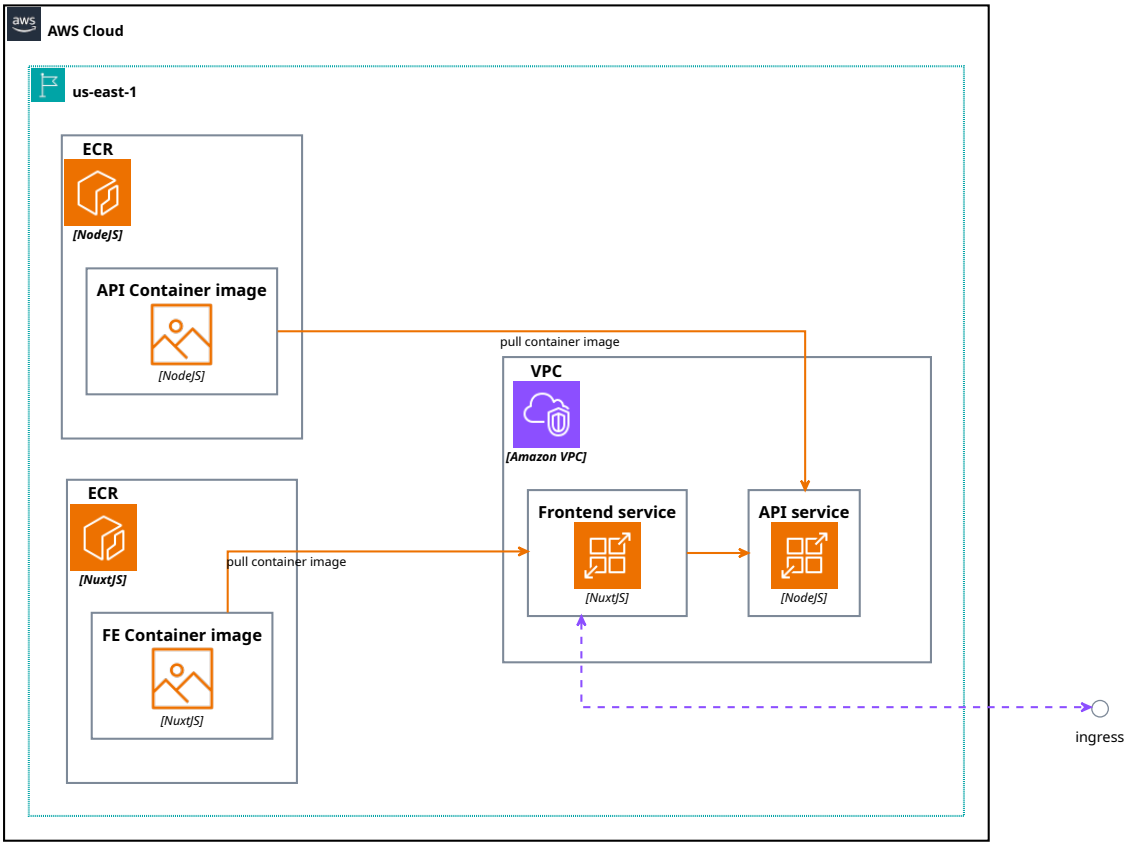
Provisioning with Terraform

In this demonstration, we will provision AWS services with Terraform as follows:

- ECR for hosting container images,
- NodeJS API service on AppRunner,
- NuxtJS FrontEnd service on AppRunner.

Local variables are defined at the top of the .tf file to be straightforward about what values should be changed depending on the project.

In the following diagram, it is clear that the publicly exposed service is the FrontEnd service, which then connects to the API service. To point and connect one service to the other environment variables are used.



Provision ECR

This is an example of how to provision the ECR for API.

```
resource "aws_ecr_repository" "my_ecr_api" {
  name                 = "my_ecr/my_api"
  image_tag_mutability = "MUTABLE"
  image_scanning_configuration {
    scan_on_push = true
  }
}

resource "aws_ecr_lifecycle_policy" "my_ecr_api" {
  repository = aws_ecr_repository.my_ecr_api.name

  policy = jsonencode({
    rules = [{
      rulePriority = 1
      description  = "keep last 10 images"
      action = {
        type = "expire"
      }
      selection = {
        tagStatus   = "any"
        countType   = "imageCountMoreThan"
        countNumber = 10
      }
    }]
  })
}

output "my_ecr_api_registry_id" {
  value       = aws_ecr_repository.my_ecr_api.registry_id
  description = ""
}
output "my_ecr_api_repository_url" {
  value       = aws_ecr_repository.my_ecr_api.repository_url
  description = ""
}
```

Provision of the API Service

In this example, the health check is performed against a custom endpoint available on the `/api/healtcheck` URL, but that might be different in your case.

To be safe and if you do not have a health check endpoint yet just use `/`.

If you choose to use HTTP protocol for the health check please **make sure your endpoint is returning HTTP status code 200**.

This instance is provisioned with the following configuration:

- 1024 CPU units,
- 2048 MB of memory,
- The health check endpoint is located at `/api/healthcheck` ,
- Container port running on 3000.

Provision the API service with Terraform

This is an example of how to provision the API service on the AWS AppRunner Container service with Terraform.

- *First we define the local variables.*

```
locals {
  api_port                  = 3000
  api_domain                = "api.mydomain.team"
  api_git_repo              = "my/my_api"
  api_development_git_branch = "dev"
  api_ecr_image_development  = "${aws_ecr_repository.my_api_ecr.repository_url}:dev"
}
```

- *Next we define the AppRunner service.*

```
resource "aws_apprunner_service" "my_api" {

  depends_on = [
    aws_ecr_repository.my_api_ecr,
    aws_iam_role.my_app_runner_roles,
    aws_apprunner_vpc_connector.my_vpc_connector
  ]

  service_name = "my_api"

  source_configuration {
    authentication_configuration {
      access_role_arn = aws_iam_role.my_app_runner_roles.arn
    }

    image_repository {
      image_identifier      = local.api_ecr_image_development
      image_repository_type = "ECR"
      image_configuration {
        port = 3000
        runtime_environment_variables = {
          # Server Port
          PORT = "3000"
          #
          NODE_ENV       = "development"
          LOG_LEVEL      = "debug"
          PORT           = local.api_port
          CONTAINER_PORT = local.api_port
          HOST_PORT      = local.api_port
        }
        runtime_environment_secrets = {
        }
      }
    }
    auto_deployments_enabled = true
  }

  health_check_configuration {
    path                = "/api/healthcheck"
    healthy_threshold   = 1
    interval            = 5
    protocol            = "HTTP"
    timeout             = 20
    unhealthy_threshold = 20
  }

  instance_configuration {
    cpu    = "2048"
    memory = "4096"
  }

  network_configuration {
    egress_configuration {
      egress_type       = "VPC"
      vpc_connector_arn = aws_apprunner_vpc_connector.my_vpc_connector.arn
    }
  }

  tags = {
    Name = "my-my_api-apprunner-service"
  }
}
```

- *Last step is to define the output of the AppRunner service domain to the screen.*

```
output "my_api_apprunner_my_api_service_url" {
  value       = aws_apprunner_service.my_api.service_url
  description = ""
}
```

Provision of the FrontEnd service

This instance is provisioned with the following configuration:

- 1024 CPU units,
- 2048 MB of memory,
- The health check endpoint is located at `/ping`.

Provision the FrontEnd service with Terraform

This is an example of how to provision the API service on the AWS AppRunner Container service with Terraform.

- *First we define the local variables.*

```
locals {
  my_fe_port                  = 3001
  my_fe_domain                = "frontend.my-domain.org"
  my_fe_apprunner_domain      = "members.development.my-domain.org"
  my_fe_ecr_image_development  = "${aws_ecr_repository.my_fe_ecr.repository_url}:dev"
  my_fe_git_repo              = "clearview/my-members-fe"
  my_fe_development_git_branch = "dev"
}
```

- *Next we define the AppRunner service.*

```
resource "aws_apprunner_service" "my_fe" {
  depends_on = [
    aws_ecr_repository.my_fe_ecr
  ]

  service_name = "my_fe"

  source_configuration {
    authentication_configuration {
      access_role_arn = aws_iam_role.my_app_runner_roles.arn
    }

    image_repository {
      image_identifier      = local.my_fe_ecr_image_development
      image_repository_type = "ECR"
      image_configuration {
        port          = local.my_fe_port
        start_command = "yarn dev"
        runtime_environment_variables = {
          HOST           = "0.0.0.0"
          HOSTNAME       = "0.0.0.0"
          NITRO_HOST     = "0.0.0.0"
          ENV            = "development"
          NODE_ENV       = "development"
          PORT           = local.my_fe_port
          NITRO_PORT     = local.my_fe_port
          CONTAINER_PORT = local.my_fe_port
          HOST_PORT      = local.my_fe_port
          # API
          PROXY_TARGET      = "https://${local.api_domain}"
          NUXT_API_BASE_URL = "https://${local.api_domain}"
          # Nuxt
          NUXT_PUBLIC_DEV      = true
          NUXT_PUBLIC_DEBUG    = true
          NUXT_PUBLIC_SITE_URL = "http://localhost:${local.my_fe_port}"
          NUXT_PORT            = local.my_fe_port
          NUXT_HOST            = "0.0.0.0"
          # Debug
          LOG_LEVEL = "debug"
        }
      }
    }
    auto_deployments_enabled = true
  }

  health_check_configuration {
    path                = "/ping"
    healthy_threshold   = 1
    interval            = 20
    protocol            = "HTTP"
    timeout             = 19
    unhealthy_threshold = 20
  }

  instance_configuration {
    cpu    = "1024"
    memory = "2048"
  }

  network_configuration {
    egress_configuration {
      egress_type       = "VPC"
      vpc_connector_arn = aws_apprunner_vpc_connector.my_vpc_connector.arn
    }

    ingress_configuration {
      is_publicly_accessible = true
    }

  }

  tags = {
    Name = "my-my_fe-apprunner-service"
  }
}
```

- *Last step is to define the output of the AppRunner service domain to the screen.*

```
output "my_fe_apprunner_my_fe_service_url" {
  value       = aws_apprunner_service.my_fe.service_url
  description = ""
}
```

Conclusion

In this article, we covered AppRunner from its humble beginnings to the powerful technology it has proven to be today. We provided useful diagrams and Terraform code for easy provisioning of service.

Monitor and Control Amazon Cloud Costs with Terraform

> Date: Jan 3, 2024

If you wonder why most startups fail it is simple, they run out of money and there have been many horror stories about crazy big cloud bills.

When we put a new client on the Cloud, we know they will get a bill eventually. It's our responsibility to make sure that the bill will not be a big and sudden surprise.

Setting up the budgets

The first step is to set budgets and alerts.

We usually set:

- The Account wide budget,
- The Most used service budget,
- The Most important tag budget.

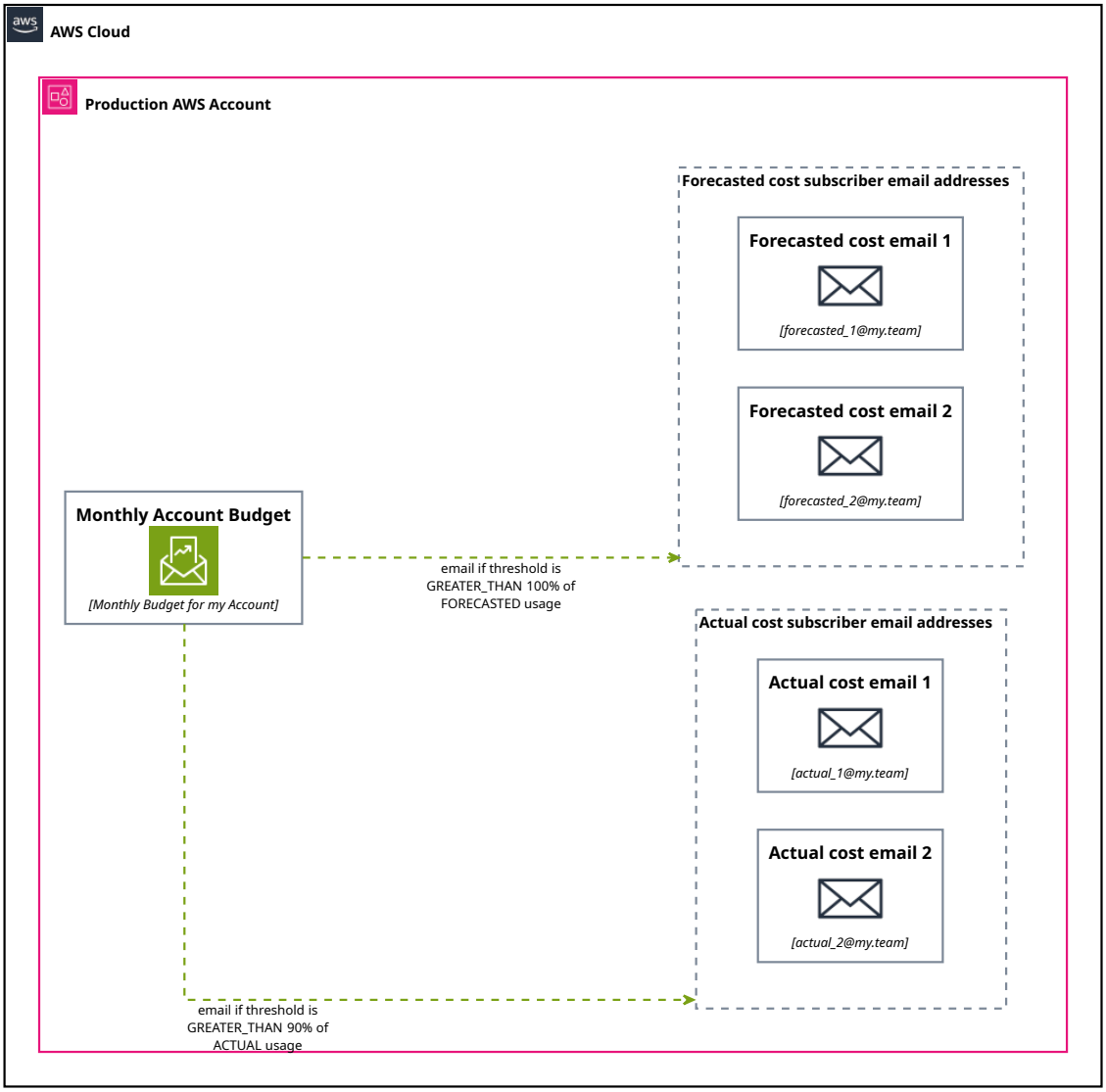
Every budget sends out an email notification when the budget cost threshold is reached, in these two cases:

- **The ACTUAL cost:** which will be triggered once that amount has already been spent,
- **The FORECASTED cost:** notification will be triggered earlier as the cost is based on prediction based on your past usage.

As we are a tech company the budgets are set using Terraform and in the next section, you can find diagrams and code samples that explain how budgets and notifications work.

Monthly Account Budget

This budget tracks account-wide costs.



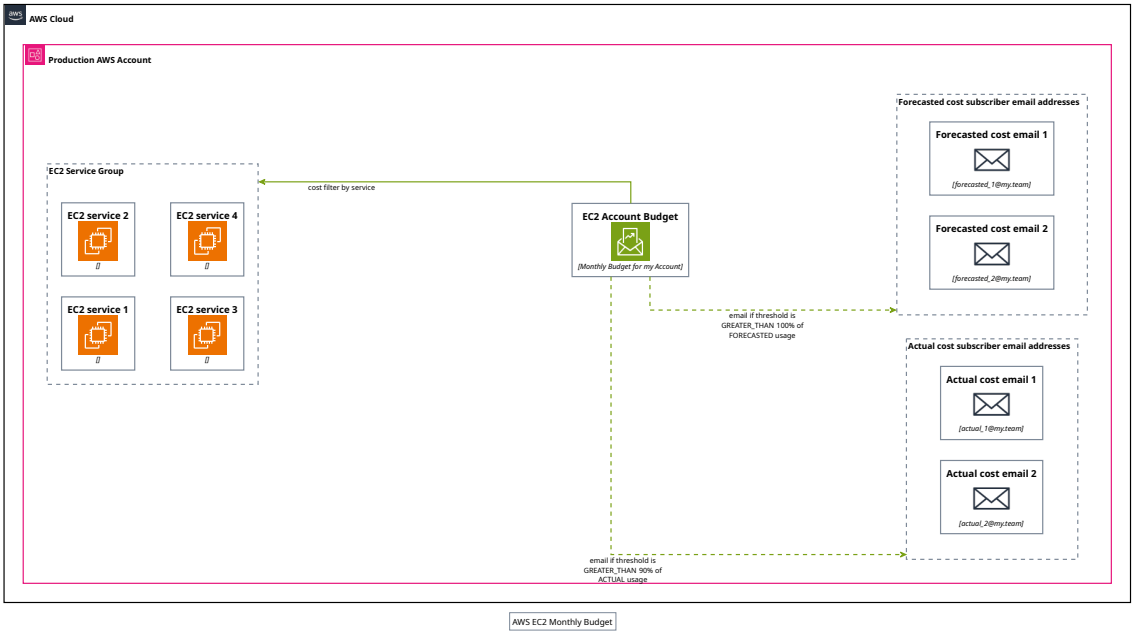
AWS Monthly Account Budget

```
resource "aws_budgets_budget" "monthly_account_budget" {
  name         = "Monthly Budget for my Account"
  budget_type  = "COST"
  limit_amount = "500"
  limit_unit   = "USD"
  time_unit    = "MONTHLY"

  notification {
    comparison_operator        = "GREATER_THAN"
    threshold                  = 90
    threshold_type             = "PERCENTAGE"
    notification_type          = "ACTUAL"
    subscriber_email_addresses = ["email@my.team"]
  }

  notification {
    comparison_operator        = "GREATER_THAN"
    threshold                  = 100
    threshold_type             = "PERCENTAGE"
    notification_type          = "FORECASTED"
    subscriber_email_addresses = ["email@my.team"]
  }
}
```

Budgets by Service

EC2 Monthly Budget

This budget will track costs for EC2 services only.

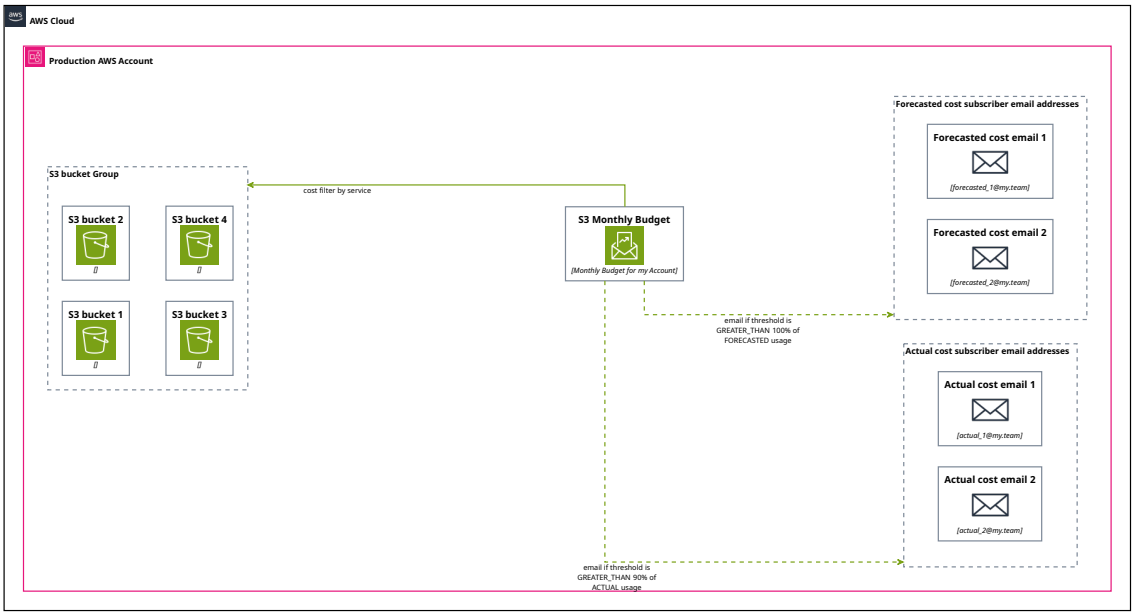


AWS EC2 Monthly Budget

```
resource "aws_budgets_budget" "ec2_monthly_budget" {
  name         = "My EC2 Monthly Budget"
  budget_type  = "COST"
  limit_amount = "400"
  limit_unit   = "USD"
  time_unit    = "MONTHLY"

  cost_filter {
    name = "Service"
    values = [
      "Amazon Elastic Compute Cloud - Compute",
    ]
  }

  notification {
    comparison_operator        = "GREATER_THAN"
    threshold                  = 90
    threshold_type             = "PERCENTAGE"
    notification_type          = "ACTUAL"
    subscriber_email_addresses = ["email@my.team"]
  }

  notification {
    comparison_operator        = "GREATER_THAN"
    threshold                  = 100
    threshold_type             = "PERCENTAGE"
    notification_type          = "FORECASTED"
    subscriber_email_addresses = ["email@my.team"]
  }
}
```

S3 Monthly Budget

This budget will track costs for S3 buckets only.

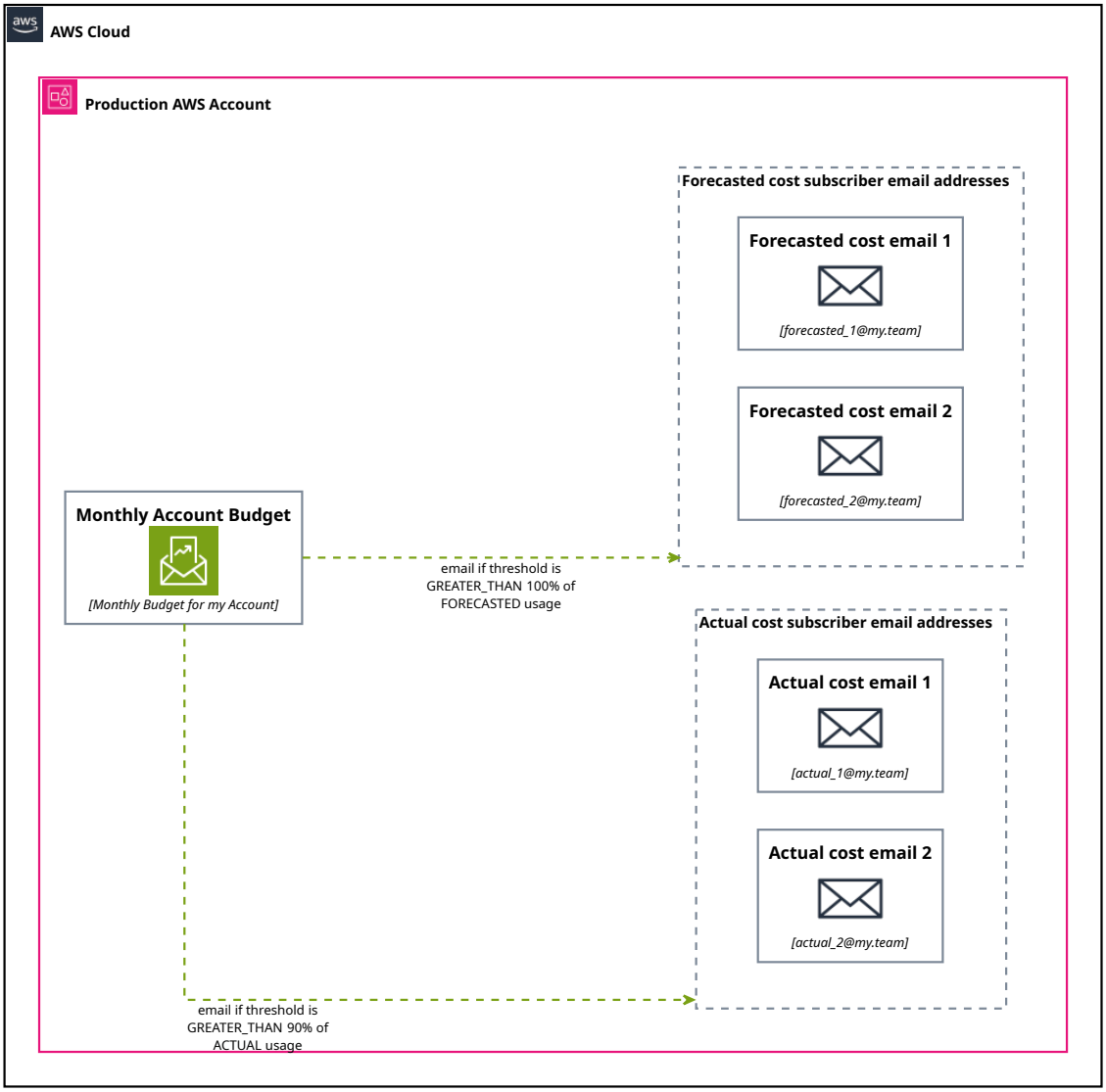


AWS S3 Monthly Budget

```
resource "aws_budgets_budget" "s3_monthly_budget" {
  name         = "My S3 Monthly Budget"
  budget_type  = "COST"
  limit_amount = "100"
  limit_unit   = "USD"
  time_unit    = "MONTHLY"

  cost_filter {
    name = "Service"
    values = [
      "Amazon Simple Storage Service",
    ]
  }

  notification {
    comparison_operator        = "GREATER_THAN"
    threshold                  = 90
    threshold_type             = "PERCENTAGE"
    notification_type          = "ACTUAL"
    subscriber_email_addresses = ["email@my.team"]
  }

  notification {
    comparison_operator        = "GREATER_THAN"
    threshold                  = 110
    threshold_type             = "PERCENTAGE"
    notification_type          = "FORECASTED"
    subscriber_email_addresses = ["email@my.team"]
  }
}
```

AppRunner Monthly Budget

This budget will track costs for AppRunner services only.



AWS Monthly Account Budget

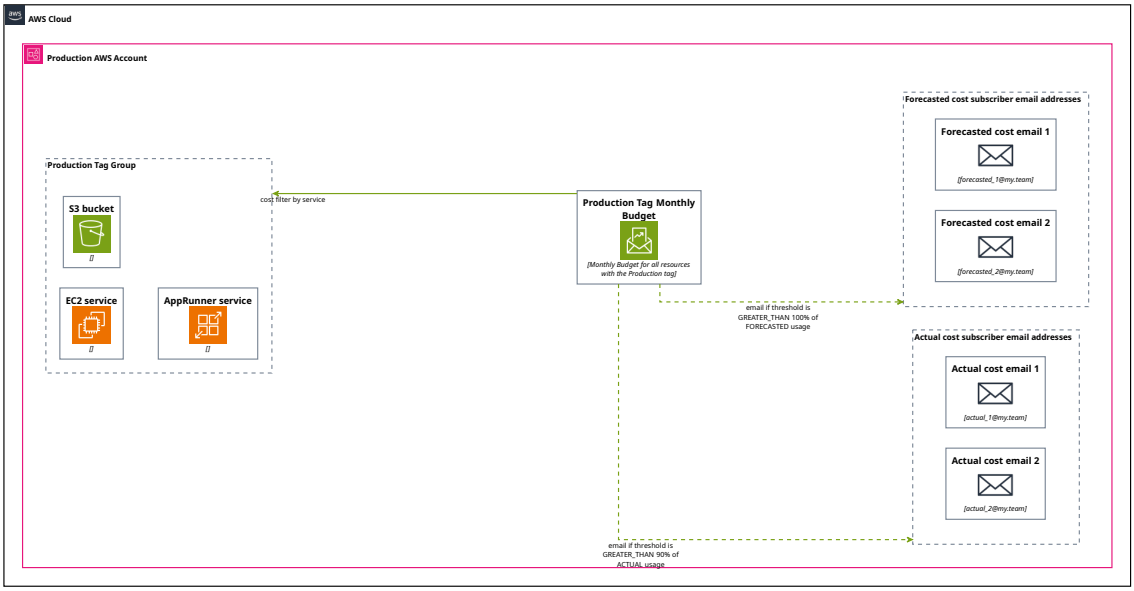
```
resource "aws_budgets_budget" "apprunner_monthly_budget" {
  name         = "My AppRunner Monthly Budget"
  budget_type  = "COST"
  limit_amount = "100"
  limit_unit   = "USD"
  time_unit    = "MONTHLY"

  cost_filter {
    name = "Service"
    values = [
      "AWS App Runner",
    ]
  }

  notification {
    comparison_operator        = "GREATER_THAN"
    threshold                  = 90
    threshold_type             = "PERCENTAGE"
    notification_type          = "ACTUAL"
    subscriber_email_addresses = ["email@my.team"]
  }

  notification {
    comparison_operator        = "GREATER_THAN"
    threshold                  = 110
    threshold_type             = "PERCENTAGE"
    notification_type          = "FORECASTED"
    subscriber_email_addresses = ["email@my.team"]
  }
}
```

Budgets by Tags

Here we will demonstrate how to create budgets based on tag filtering.

Production Tag Monthly Budget

This budget will group multiple types of service based on the tag, in this case, the production tag is used.



AWS Production Tag Monthly Budget

```
resource "aws_budgets_budget" "my_production_monthly_budget" {
  name         = "My Production Monthly Budget"
  budget_type  = "COST"
  limit_amount = "400"
  limit_unit   = "USD"
  time_unit    = "MONTHLY"

  cost_filter {
    name = "TagKeyValue"
    values = [
      "environment$Production",
    ]
  }

  notification {
    comparison_operator = "GREATER_THAN"
    threshold           = 90
    threshold_type      = "PERCENTAGE"
    notification_type   = "ACTUAL"
    subscriber_email_addresses = [
      "email1@my.team",
      "email2@my.team"
    ]
  }

  notification {
    comparison_operator = "GREATER_THAN"
    threshold           = 110
    threshold_type      = "PERCENTAGE"
    notification_type   = "FORECASTED"
    subscriber_email_addresses = [
      "email1@my.team",
      "email2@my.team"
    ]
  }
}
```

Conclusion

This article explained the importance of setting a cloud budget and then demonstrated how to do it programmatically. Not only that, we went a step further and created nice diagrams for each budget so that it is easier for every reader to imagine the setup.

We hope this article finds its way and helps you avoid unnecessary cloud costs.

DNS stand-alone subdomain on AWS Route53 with Terraform

> Date: 2024-03-19

Often I find myself in a situation where I need to create a root DNS zone which is a subdomain. This can be the case when the actual root zone is hosted on a different account.

> This is a common scenario when it is a requirement to host production and staging environments in different AWS accounts.

Therefore we create the staging subdomain in the staging account and create a record in the production account that points to the staging subdomain's Name Servers.

The Staging Account

Creating the Subdomain in Route53

The first step is to create the staging subdomain in Terraform,

```
resource "aws_route53_zone" "staging_my_domain_org" {
  name     = "staging.my-domain.org"
  tags     = {}
  tags_all = {}
}
```

Output the Staging Name Servers

Following Terraform code will output the Name Servers of the newly created subdomain

```
output "staging_my_domain_org_nameservers" {
  value = aws_route53_zone.staging_my_domain_org.name_servers
}
```

Get the Staging Name Servers

Run Terraform and apply the changes. After the records have been created the Name Servers of the newly created subdomain will be output on the screen. This is an example below.

```
staging_my_domain_org_nameservers = tolist([
  "ns-1177.awsdns-11.org",
  "ns-18652.awsdns-40.co.uk",
  "ns-3610.awsdns-45.com",
  "ns-9538.awsdns-53.net",
])
```

The Production Account

Now that we have the Name Servers we need to point the subdomain to them but this time on the production account.

Create the NS Record

We are gonna create a `staging.my-domain.org` NS record under the `my-domain.org` zone. We will do this using the web console as in my case that zone is not managed by Terraform.

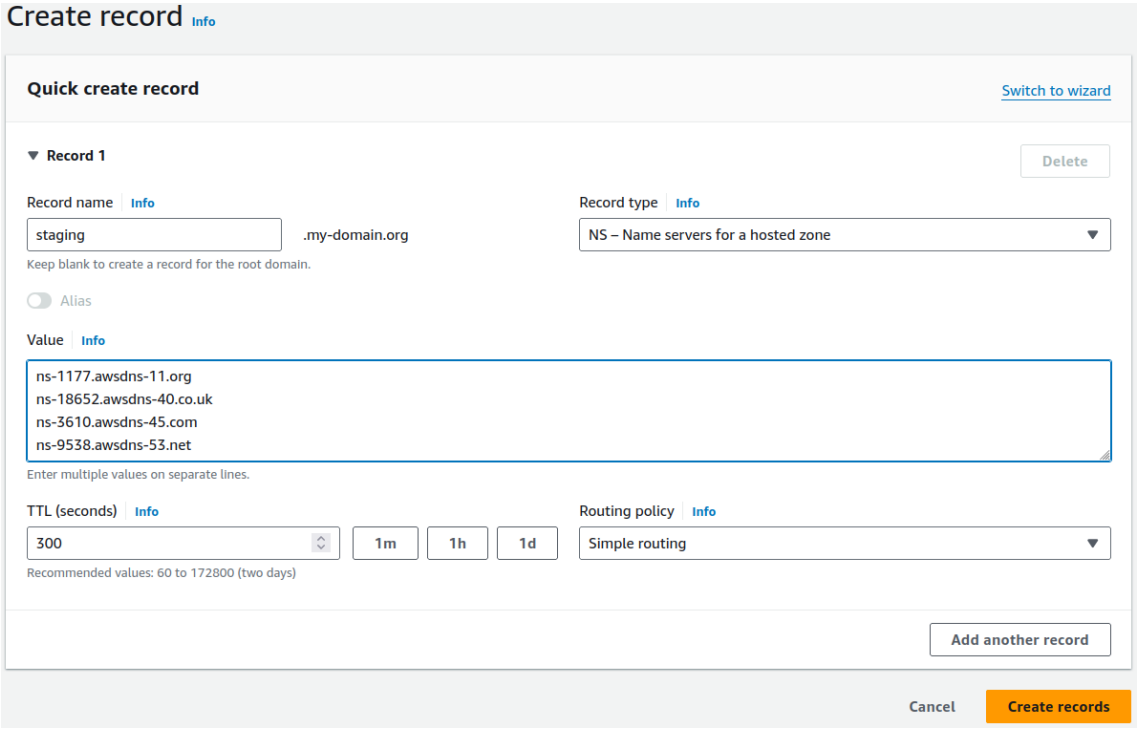


Figure 1: AWS DNS

Click `Create records` and you are done.

Conclusion

This is a valuable techie tip that served me well many times. I hope this simple approach serves your well.

Developer Read-Only AWS IAM Group with Terraform

> Date: 2024-03-11

One of the first things I do when setting up a Cloud account on AWS for any client is to set up an `IAM` group for developers.

> Most of the time the Developers just need a read-only group that will enable them to access resources but not modify anything.

Therefore they need to view logs and easily debug while being restricted not to doing something stupid and accidentally deleting a resource.

In the next part of this article we will:

- Create a new IAM group named *developer_read_only_group*,
- Attach few read only policies that are already provided and manages by AWS,
- Create a new custom policy that will expand the AWS read only policies,
- Attach the custom policy to the group.

Define the Group

In this step, we define the `IAM` group.

```
resource "aws_iam_group" "developer_read_only_group" {
  name = "developer_read_only_group"
}
```

Attach AWS Managed Policies

In this step, we attach AWS-managed read-only policies to the previously created `IAM` group.

```
data "aws_iam_policy" "developer_read_only_group_policy" {
  for_each = toset(
    [
      "AmazonCloudWatchEvidentlyReadOnlyAccess",
      "AmazonRDSReadOnlyAccess",
      "AWSAppRunnerReadOnlyAccess",
      "AmazonS3ReadOnlyAccess",
      "CloudFrontReadOnlyAccess",
      "AWSCodeBuildReadOnlyAccess",
      "AWSCodePipeline_ReadOnlyAccess",
      "AmazonEC2ReadOnlyAccess",
      "IAMReadOnlyAccess"
    ]
  )
  name = each.value
}

resource "aws_iam_group_policy_attachment" "developer_read_only_group_policy" {
  for_each   = data.aws_iam_policy.developer_read_only_group_policy
  group      = aws_iam_group.developer_read_only_group.name
  policy_arn = each.value.arn
```

Custom Policy

In this step, we create a custom `IAM` policy and attach it to the `IAM` group. This is where you can add custom permissions to the policies that are not covered by the AWS-managed ones.

```
data "aws_iam_policy_document" "custom_general_developer_read_only" {
  statement {
    effect = "Allow"
    actions = [
    "s3:ListAllMyBuckets",
      "s3:ListBucketMultipartUploads",
      "s3:GetAccountPublicAccessBlock",
      "s3:GetBucketPublicAccessBlock",
      "s3:GetBucketPolicyStatus",
      "s3:GetBucketAcl",
      "s3:ListAccessPoints"
    ]
    resources = [
      "*"
    ]
  }

  statement {
    effect = "Allow"
    actions = [
      "acm:ListCertificates",
      "cloudfront:GetDistribution",
      "cloudfront:GetStreamingDistribution",
      "cloudfront:GetDistributionConfig",
      "cloudfront:ListDistributions",
      "cloudfront:ListCloudFrontOriginAccessIdentities",
      "cloudfront:CreateInvalidation",
      "cloudfront:GetInvalidation",
      "cloudfront:ListInvalidations"
    ]
    resources = [
      "*"
    ]
  }

  statement {
    effect = "Allow"
    actions = [
    "logs:DescribeLogGroups",
    "logs:FilterLogEvents",
      "logs:DescribeLogGroups"
      "logs:DescribeLogGroups",
      "logs:FilterLogEvents"
    ]
    resources = [
      "*"
    ]
  }

  statement {
    effect = "Allow"
    actions = [
      "ecs:ListClusters",
      "ecs:Describe*",
      "ecs:List*",
      "ecs:ListContainerInstances",
      "ecs:ListTasks",
      "ecs:DescribeClusters",
      "ecs:DescribeClusters"
    ]
    resources = [
      "*"
    ]
  }

  statement {
    effect = "Allow"
    actions = [
      "iam:ListAttachedRolePolicies",
      "iam:ListRoles",
      "iam:ListGroups",
      "iam:ListUsers",
      "iam:GetPolicy",
      "iam:GetPolicyVersion",
      "iam:GetRole"
    ]
    resources = [
      "*"
    ]
  }

}
```

In this step, we attach the custom policy to the group.

```
resource "aws_iam_policy" "custom_general_developer_read_only" {
  name        = "custom_general_developer_read_only"
  path        = "/"
  description = "Allow "
  policy      = data.aws_iam_policy_document.custom_general_developer_read_only.json
}
```

In this step, we attach the custom policy to the `developer_read_only_group` group.

```
resource "aws_iam_group_policy_attachment" "custom_general_developer_read_only" {
  group      = aws_iam_group.developer_read_only_group.name
  policy_arn = aws_iam_policy.custom_general_developer_read_only.arn
}
```

User Management

Now that we have all `IAM` groups and policies in place we can focus on creating a user and assigning groups to the users.

```
# Create a User
resource "aws_iam_user" "user_name_created_with_terraform" {
  name = "user_name_created_with_terraform"
}

# Group Membership
resource "aws_iam_user_group_membership" "user_name_created_with_terraform" {
  user = aws_iam_user.user_name_created_with_terraform.name

  groups = [
    aws_iam_group.developer_read_only_group.name
  ]
}
```

Attach the AWS Change Password Policy

In this step, we attach the AWS managed change password policy. This will permit the user to change their password.

```
resource "aws_iam_policy_attachment" "iam_user_change_password_policy_attach" {
  name = "iam_user_change_password_policy_attach"
  users = [
    "aws_username_not_added_by_terraform",
    aws_iam_user.user_name_created_with_terraform.name,
  ]
  policy_arn = "arn:aws:iam::aws:policy/IAMUserChangePassword"
}
```

Conclusion

In this article, we covered how to create a read-only `IAM` group with AWS-managed and custom policies, and user management. This is

very useful for every Cloud Engineer.

Deploying PHP-App-as-a-Container Services in Amazon Lightsail with GitHub Actions

> Date: Jun 21, 2021

AWS Lightsail containers service is a new and easy way to deploy containers. With Lightsail, you don't need to worry about registries, services, health checks, hostnames, etc. — as everything is built-in.

The app

In this example, we will be deploying a dummy PHP app and running two containers:

- NginX container
- PHP container

I even provided a dummy composer file so that we can install the dependencies.

The NginX Container

This Dockerfile copies preconfigured NginX configuration files and sets the usual stuff, such as:

- the webroot,
- enables gzip,
- increases the size limits so that we do not get the usual timeouts,
- And, of course, the most important thing is to pass the php files to the PHP container (which listens on port 9001).

For more detailed information, please see the linked file below.

```
FROM nginx as nginx

WORKDIR /app

COPY infra/nginx/nginx.conf /etc/nginx
COPY infra/nginx/default.conf /etc/nginx/conf.d
COPY infra/nginx/gzip.conf /etc/nginx/conf.d
COPY infra/nginx/proxy_params.conf /etc/nginx/conf.d
COPY infra/nginx/size_limits.conf /etc/nginx/conf.d
COPY infra/nginx/timeouts.conf /etc/nginx/conf.d

COPY . .
```

The PHP Container

This Dockerfile does the following:

- uses the latest Composer Docker image to copy the composer binary to the PHP image,
- installs the PHP version 8 and the required extensions,
- sets the usual parameters in php.ini,
- shows the installed php modules during build (great for debugging),
- changes the PHP-FPM port from the standard 9000 to 9001,
- runs the composer install command,
- uses a custom entry point script, which is great for error debugging as you can later add a custom command (which will be run on container startup).

```
FROM composer:latest as composer

FROM php:8.0-fpm-alpine as php_install
RUN apk update && apk add mysql-client icu make icu-dev g++
COPY --from=composer /usr/bin/composer /usr/bin/composer
RUN docker-php-ext-install intl mysqli pdo_mysql && \
    docker-php-ext-enable intl mysqli pdo_mysql && \
    php -m && \
    pecl list-packages

FROM php_install as php_setup
RUN mv "$PHP_INI_DIR/php.ini-development" "$PHP_INI_DIR/php.ini"
RUN sed -e 's/max_execution_time = 30/max_execution_time = 90/' -i "$PHP_INI_DIR/php.ini" && \
    sed -e 's/memory_limit = 128M/memory_limit = 136M/' -i "$PHP_INI_DIR/php.ini" && \
    sed -e 's/post_max_size = 8M/post_max_size = 16M/' -i "$PHP_INI_DIR/php.ini" && \
    sed -e 's/upload_max_filesize = 2M/upload_max_filesize = 22M/' -i "$PHP_INI_DIR/php.ini"  && \
    sed -i 's/9000/9001/' /usr/local/etc/php-fpm.d/*

RUN cat /usr/local/etc/php/php.ini | grep ^[^\;] && \
    cat /usr/local/etc/php/php.ini | grep error

WORKDIR /app

COPY . .

RUN composer install && \
    chown -R www-data:www-data /app

COPY infra/docker-php-entrypoint.sh /entrypoint.sh
RUN chmod +x /entrypoint.sh
EXPOSE 9001
ENTRYPOINT ["/entrypoint.sh"]
CMD ["php-fpm"]
```

Lightsail setup

Create the service via `awscli`.

We must choose:

- the name of the service,
- the power of the service (the memory and the vCPU),
- the scale of the service (the number of compute nodes on which we run the containers).

In this example, we will add a service named my-lightsail-container-service, with the power of nano, and we will scale it to one compute node instance.

Sometimes the deployment can fail because of the lack of resources; in this case, up the power of the service.

```
aws lightsail create-container-service \
– service-name my-lightsail-container-service \
– power nano \
– scale 1 \
– output yaml
```

You will need to add the service name as a Github secret for the deployment via Github actions to work.

Let’s check:

- if the service has been created
- its state
- the public endpoint.

```
aws lightsail \
get-container-services \
– output yaml
```

You need to have `awscli` installed and the credentials on your local comp for the above commands to work.

GitHub Actions setup

Secrets setup

The first step is to set up the Github secrets for AWS authentication and to set the container service name that will be created on AWS Lightsail.

To do so, create the following secrets, as seen in the picture below:

- AWS_ACCESS_KEY_ID
- AWS_SECRET_ACCESS_KEY
- AWS_REGION
- SERVICE_NAME

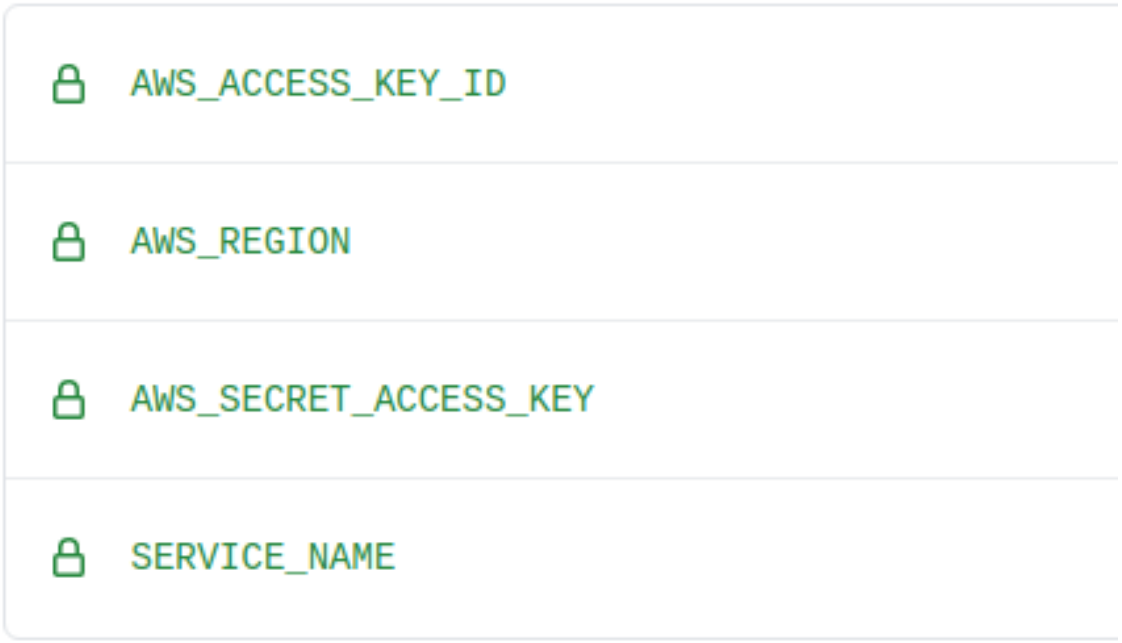


Figure 1: Image showing Secrets setup.

The workflow

The GitHub actions consist of two files:

- The CI/CD workflow file which runs the action,
- the container deployment template.

The CI/CD Workflow file

These are the steps in the workflow:

- Upgraded the `awscli` on the runner and install the `lightsailctl`,
- Configure the AWS Credentials so that we can deploy,
- Build Docker PHP Image,
- Build Docker NginX Image,

- Lists the Docker images on the runner,
- Push the Docker PHP Image,
- Push the Docker NginX Image,
- Get the Docker Images from Lightsail,
- Get the Latest NginX Docker Images from Lightsail,
- Get the Latest PHP Docker Images from Lightsail,
- Get the latest PHP Docker Image from Lightsail and save it to the LATEST_PHP_LIGHTSAIL_DOCKER_IMAGE variable,
- Get the latest NGINX Docker Image from Lightsail and save it to the LATEST_NGINX_LIGHTSAIL_DOCKER_IMAGE,
- Create container_with_image.yml and populate it with the values from LATEST_PHP_LIGHTSAIL_DOCKER_IMAGE and LATEST_NGINX_LIGHTSAIL_DOCKER_IMAGE,
- Deploy to Lightsail,
- Debug PHP container logs,
- Debug NginX container logs.

```
name: Dev - Build and deploy Docker to AWS Lightsail Container Service

on:
  push:
   branches:
     - main

jobs:
  build_and_deploy_to_lightsail_container_service:
    runs-on: ubuntu-latest

    env:
      SERVICE_NAME: ${{ secrets.SERVICE_NAME }}
      DATABASE_URL: ${{ secrets.DEV_DATABASE_URL }}

    steps:

    - uses: actions/checkout@master

    - name: Upgrade AWS CLI version and setup lightsailctl
      run: |
       aws --version
       curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
       unzip awscliv2.zip
       sudo ./aws/install --bin-dir /usr/local/bin --install-dir /usr/local/aws-cli --update
       which aws
       aws --version
       sudo curl "https://s3.us-west-2.amazonaws.com/lightsailctl/latest/linux-amd64/lightsailctl" -o
       ↪  "/usr/local/bin/lightsailctl"
       sudo chmod +x /usr/local/bin/lightsailctl

    - name: Configure AWS Credentials
      uses: aws-actions/configure-aws-credentials@v1
      with:
        aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
        aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
        aws-region: ${{ secrets.AWS_REGION }}

    - name: Build Docker PHP Image
      run: >
        docker build .
        --file ./infra/Dockerfile.php
        --tag php:latest

    - name: Build Docker NginX Image
      run: >
        docker build .
        --file ./infra/Dockerfile.nginx
        --tag nginx:latest

    - name: List the Docker Images and Tags
```

```
      run: |
        docker images list

    - name: Push the Docker PHP Image
      run: >
        aws lightsail push-container-image
        --service-name ${{ env.SERVICE_NAME }}
        --label php
        --image php:latest

    - name: Push the Docker NginX Image
      run: >
        aws lightsail push-container-image
        --service-name ${{ env.SERVICE_NAME }}
        --label nginx
        --image nginx:latest

    - name: Get the Docker Images from Lightsail
      run: >
        aws lightsail get-container-images
        --service-name ${{ env.SERVICE_NAME }}

    - name: Get the Latest NginX Docker Images from Lightsail
      run: >
        aws lightsail get-container-images
        --service-name ${{ env.SERVICE_NAME }} |
        jq --arg NGINX_DOCKER_IMAGE "$NGINX_DOCKER_IMAGE" '.containerImages[] |
        select(.image | startswith($NGINX_DOCKER_IMAGE))'
      env:
        NGINX_DOCKER_IMAGE: ":${{env.SERVICE_NAME}}.nginx"

    - name: Get the Latest PHP Docker Images from Lightsail
      run: >
        aws lightsail get-container-images --service-name ${{ env.SERVICE_NAME }} |
        jq --arg PHP_DOCKER_IMAGE "$PHP_DOCKER_IMAGE" '.containerImages[] |
        select(.image | startswith($PHP_DOCKER_IMAGE))'
      env:
        PHP_DOCKER_IMAGE: ":${{env.SERVICE_NAME}}.php"

    - name: Get the latest PHP Docker Image from Lightsail
      run: >
        echo "LATEST_PHP_LIGHTSAIL_DOCKER_IMAGE=$(aws lightsail get-container-images --service-name ${{
        ↪  env.SERVICE_NAME }} | jq -r '.containerImages | map(select(.image | contains ("php") )) |
        ↪  .[0].image')" >> $GITHUB_ENV

    - name: Get the latest NGINX Docker Image from Lightsail
      run: >
        echo "LATEST_NGINX_LIGHTSAIL_DOCKER_IMAGE=$(aws lightsail get-container-images --service-name ${{
        ↪  env.SERVICE_NAME }} | jq -r '.containerImages | map(select(.image | contains ("nginx") )) |
        ↪  .[0].image')" >> $GITHUB_ENV

    - name: Test value of LATEST_PHP_LIGHTSAIL_DOCKER_IMAGE and LATEST_NGINX_LIGHTSAIL_DOCKER_IMAGE var
      run: |
        echo $LATEST_NGINX_LIGHTSAIL_DOCKER_IMAGE
        echo $LATEST_PHP_LIGHTSAIL_DOCKER_IMAGE

    - name: Create container_with_image.yml
      run: |
        cat .github/workflows/container.yml.tpl | envsubst > container_with_image.yml
        cat container_with_image.yml

    - name: Deploy to Lightsail
      run: >
        aws lightsail create-container-service-deployment
        --cli-input-yaml file://container_with_image.yml
        --output yaml

    - name: Debug PHP container logs
      run: >
        aws lightsail get-container-log
        --service-name ${{ env.SERVICE_NAME }}
        --container-name php
        --output yaml

    - name: Debug NginX container logs
      run: >
        aws lightsail get-container-log
        --service-name ${{ env.SERVICE_NAME }}
        --container-name php
        --output yaml
```

The Container Deployment file

This file defines the result that is parsed by AWS Lightsail. It took all the steps above just to get here; we have a final version of the deployment file.

This file defines the following:

- The service name,
- Containers with the latest image version,
- The environment variables for each container,
- Which container will be used as the public endpoint,
- The health check.

```
serviceName: ${SERVICE_NAME}
containers:
  nginx:
    command: []
    environment:
      test: test
    image: ${LATEST_NGINX_LIGHTSAIL_DOCKER_IMAGE}
    ports:
      "80": HTTP
  php:
    command: []
    environment:
      APP_NAME: "My App Name"
      DEBUG: "true"
      DATABASE_URL: "${DATABASE_URL}"
    image: ${LATEST_PHP_LIGHTSAIL_DOCKER_IMAGE}
publicEndpoint:
  containerName: nginx
  containerPort: 80
  healthCheck:
    healthyThreshold: 2
    intervalSeconds: 20
    path: /index.php
    successCodes: 200-499
    timeoutSeconds: 4
    unhealthyThreshold: 2
```

Conclusion

In this article, we demonstrated in a detailed way how to deploy a PHP app hosted on the popular GitHub platform, using Github actions, to AWS Lightsail Container Service — and how to debug the deployed containers. We hope you will have fun working with this new tech from AWS, which brings containers one step closer to a wider audience.

Support

Please support my other work.

Master Linux Permissions and File Types While Your Coffee Brews

This book is available on: ***GumRoad***.

About The Book

In Linux, everything you interact with, such as files, folders, and even devices like your keyboard or mouse, is considered a `file`. This might seem unusual the first time you start working on Linux, but it's a core concept that makes Linux incredibly flexible and powerful.

This book, will tech you about the different types of files, from regular files to special files that represent hardware or are used for communication. It also teaches about links, which are shortcuts to files, and how to identify them.

One of the most important aspects of Linux is security. File permissions determine who can access and modify your files. This is crucial in a multi-user environment, where you usually share a computer with other people. This book will tech you how to use commands like `ls` and `chmod` to view and change these permissions, giving you control over your data.

Figure 1: Master Linux Permissions and File Types Book

Special permissions, like the `Sticky bit`, `SUID`, and `SGID`, provide extra layers of security. For example, the `Sticky bit` can prevent other users from deleting files in shared directories.

Understanding file types and permissions is a must for any Linux user. It allows you to manage your system efficiently and protect your data.