AWS Fargate with App::FargateStack

A Builder's Guide

By Rob Lauer

Version 1.0.34

What Is Fargate?

AWS Fargate is a serverless, pay-as-you-go compute engine that lets you run containers without managing servers or clusters. It sits at the heart of the AWS container ecosystem, offering a powerful middle ground between the raw control of EC2 instances and the high-level abstraction of services like AWS App Runner. Its unique and compelling power comes from this position: Fargate handles all the underlying infrastructure management—the patching, scaling, and securing of servers—so you can focus purely on your application, packaged as a container. It's the "just run my container" service that developers have always wanted, combining the simplicity of serverless with the power of container orchestration.

Think of AWS Fargate as a powerful set of Legos. You can build almost any containerized application with it, but first, you have to assemble all the foundational plumbing—the networking, roles, security groups, and service definitions. This initial setup involves a lot of friction and deep knowledge, which is often why architects look to other solutions.

What is App::FargateStack?

App::FargateStack is like an expert builder that assembles that standard plumbing for you, following best practices. It lets you get straight to your main goal: deploying your containerized application. When it's done, it hands you a transparent box where you can see all the gears and springs working together. There's no magic; it's all right there in front of you.

This guide is for anyone who wants to build on that solid foundation.

This book is for you if:

- You are an **Architect or DevOps Engineer** who understands the plumbing but wants to automate the tedious, repetitive assembly process with a tool that is both powerful and transparent.
- You are a **Developer** whose primary goal is to deploy a container. You don't necessarily need to know how every piece of the plumbing works on day one, but you value a tool that builds it correctly and doesn't hide the details, allowing you to learn and grow.
- You are **learning AWS Fargate** and want to see what a production-ready stack looks like "under the hood." This guide uses a real-world framework to teach you the components of a robust Fargate deployment.

The App::FargateStack Philosophy

This framework—and by extension, this guide—is not anti-shortcut. In fact, App::FargateStack is a powerful shortcut.

The distinction is the *kind* of shortcut it provides. It is a professional accelerator, not a fragile, opaque hack. It's for those who want a shortcut to a **correct and maintainable solution**, not just *any* solution that happens to work right now.

If you appreciate tools that automate complexity without sacrificing transparency, then you've come to the right place.

Welcome. Let's get building.

AWS Fargate with App::FargateStack **1**
A Builder's Guide 1
By Rob Lauer 1
Version 1.0.34What Is Fargate? 1
What is App::FargateStack? 2
The App::FargateStack Philosophy 2
Preface: Why Fargate Deserves a Second Look **6**
The Simplicity of Intent 6
Fargate vs. EC2: A Clear Choice 7
The Cost Argument: Pay for What You Use 8
Where App::FargateStack Fits In 9
Chapter 1: Introduction to App::FargateStack **10**
What is App::FargateStack? 10
The Core Philosophy: Configuration as State 10
The plan and apply Workflow 10
Chapter 2: Your First Stack - A Guided Tutorial **12**
Step 1: Creating the Configuration with create-stack 12
Step 2: Planning the Infrastructure with plan 12
Step 3: Provisioning the Stack with apply 14
Step 4: Launching and Verifying the Service 15
Chapter 3: The Configuration File in Detail **18**
Top-Level Configuration 18
The tasks Section 18
Chapter 4: Core Workload Patterns **21**
4.1 Understanding Task Types 21
4.2 Daemon Services 21
4.3 Tasks: Ad-Hoc and Scheduled Jobs 22
Ad-Hoc Jobs 23
Scheduled Jobs 23
4.4 HTTP & HTTPS Services 25
Architectural Flow 26
Chapter 5: Advanced Topics & Integrations **28**
5.1 Networking: VPC and Subnet Discovery 28
VPC Discovery 28
Subnet Categorization 28
5.2 Security: IAM, Security Groups, and Secrets 29
IAM Roles and Policies 29
Security Groups 29
Injecting Secrets from Secrets Manager 29

S3 Buckets 32
EFS File Systems 32
Automating Network Access with authorize_ingress 33
5.4 Messaging: SQS Queues 33
Custom Queue Attributes 34
5.5 Environment Variables & Secrets 35
The environment Block: For Non-Sensitive Configuration 35
The secrets Block: For Sensitive Credentials 35
Chapter 6: Command Reference **37**
Core Workflow Commands 37
plan 37
apply 37
Service & Task Lifecycle Commands 37
start-service 37
stop-service 37
redeploy 37
run-task 38
stop-task 38
State & Information Commands 38
status 38
list-tasks 38
logs 38
state 38
Configuration & Definition Commands 38
create-stack 39
register-task-definition 39
update-service 39
Chapter 7: Continue Your Fargate Journey **40**
Valuable Resources for Deeper Learning 40
Join the Community 40

Preface: Why Fargate Deserves a Second Look

In a world filled with complex container orchestration platforms, it's easy to overlook the elegant power of simpler solutions. AWS Fargate is often dismissed far too quickly, lost in the noise of the Kubernetes bandwagon. This guide—and the `App::FargateStack` framework it documents—is built on a simple premise: for the vast majority of containerized workloads, **Fargate is the smarter, faster, and more cost-effective choice**.

The primary motivation behind `App::FargateStack` was to make provisioning Fargate tasks simple, fast, and easy. By doing so, we hope to evangelize a versatile service that deserves a more prominent place in every cloud architect's toolbox.

The Simplicity of Intent

Imagine you want to run a background worker. With `App::FargateStack`, your entire initial thought process can be captured in a few lines of YAML:

```
app:
  name: my-stack
tasks:
  my-daemon:
    image: mycorp/worker:latest
    type: daemon
```

This is the core of your **intent**. From this simple declaration, a whole world of production-ready infrastructure can be planned and provisioned. This is the promise of `App::FargateStack` to let you focus on your container, not the fleet of servers beneath it.

Fargate vs. EC2: A Clear Choice

For decades, the virtual machine—the EC2 instance—has been the default unit of cloud computing. But for containerized applications, it introduces a layer of unnecessary operational overhead. You are responsible for provisioning, patching, scaling, and securing the host OS, all before you even run your first container.

Fargate eliminates this entire class of problems.

Feature	**AWS Fargate**	**EC2**
Provisioning	Fully managed—no server provisioning	Manual—you provision and manage EC2 instances
Scaling	Auto-scales per task definition	You manage autoscaling groups or scale manually
IAM Permissions	Per-task IAM roles	Instance-level IAM roles
Pricing Model	Pay per task vCPU and memory	Pay per instance uptime, regardless of usage
Maintenance	None—no OS patching or AMI management	You are responsible for updates and security patches

In short, Fargate excels in simplicity and security isolation. It is the ideal platform for modern, containerized microservices. You should only fall back to EC2 if you need deep, low-level control over the host environment, such as access to a GPU.

The Cost Argument: Pay for What You Use

The most compelling argument for Fargate comes down to cost, especially for intermittent workloads like scheduled jobs or services with variable traffic. With EC2, you pay for the instance to be "on," whether it's doing useful work or sitting idle. With Fargate, you pay only for the CPU and memory your task consumes, for the seconds it's running.

Consider a simple task that runs for one minute every 15 minutes:

Category	**Fargate (Intermittent)**	**EC2 t3.small (Always On)**
Total compute time	48 hours / month	720 hours / month
Monthly Estimate	**~$0.60**	**~$18.08**

For workloads that aren't running at 100% capacity 24/7, the cost savings are not just marginal—they are dramatic.

Where App::FargateStack Fits In

Once you've chosen Fargate, the next question is how to manage it. While you can use verbose tools like Terraform or CloudFormation, a new class of "infrastructure accelerators" has emerged to simplify the process. App::FargateStack is one such tool, but it occupies a unique space.

Feature	App::FargateStack	AWS Copilot CLI	AWS App Runner
Core Idea	Infrastructure accelerator	Developer-first CLI	Fully managed service
Abstraction	**Glass Box:** Automates visible resources	Higher Abstraction: Manages CFN stacks	**Black Box:** Hides all infrastructure
Workloads	Web, Daemons, Scheduled & Ad-hoc	Web, Backend, Scheduled	Web Apps & APIs only
Transparency	**Very High:** Updates YAML with live state	**High:** Resources are in your account	**Low:** You don't see the underlying parts

App::FargateStack is the ideal tool for teams who want powerful automation without sacrificing visibility. It automates the tedious parts of building a Fargate stack but keeps you in full control, with a transparent view of every resource it creates.

This guide will show you how to harness this power. By pairing the simplicity of Fargate with the intelligent automation of App::FargateStack, you can build and deploy sophisticated, production-grade applications faster and more efficiently than ever before.

Chapter 1: Introduction to App::FargateStack

Welcome to the engine room. The preface made the case for *why* you should build on AWS Fargate. This chapter introduces the tool that will help you do it: `App::FargateStack`.

What is App::FargateStack?

`App::FargateStack` is an opinionated, command-line framework that acts as an **infrastructure accelerator**. Its purpose is to automate the creation of all the foundational "plumbing" required to run containerized applications on Fargate.

It is not a replacement for understanding AWS, but rather a powerful tool that handles the repetitive, error-prone, and complex parts of infrastructure setup for you. It lets you focus on your application's intent—what you want to run—while it takes care of the implementation details, following production-ready best practices.

The Core Philosophy: Configuration as State

The most important concept to understand about `App::FargateStack` is its "configuration as state" philosophy. Your entire stack—every task, service, IAM role, and networking component—is defined in a single YAML file.

This file is not just a set of instructions; it is a living document that represents the **desired state** of your infrastructure. The framework's job is to make your AWS environment match what is described in that file.

This leads to a powerful and transparent workflow.

The `plan` and `apply` Workflow

`App::FargateStack` operates on a two-phase cycle that will be familiar to users of tools like Terraform:

1. **`plan` (The Dry Run)**: When you run the `plan` command, the framework reads your YAML file, inspects your live AWS environment, and performs a discovery process. It identifies what resources already exist and what needs to be created or updated to match your configuration. It then presents a clear blueprint of the changes it intends to make, without actually touching anything in your account.
2. **`apply` (The Execution)**: Once you've reviewed the plan and are confident in the changes, you run the `apply` command. The framework then executes the plan, making the necessary API calls to create, configure, and connect all the resources in your stack.

Crucially, after a successful `apply`, the framework updates your YAML file with the real-world details of the provisioned resources, such as ARNs, IDs, and auto-discovered network settings. Your configuration file becomes a perfect, up-to-date record of your live infrastructure. This idempotency means you can run `apply` over and over again, and the framework will only make the changes necessary to bring your stack into alignment with your configuration.

With this foundation in place, we are now ready to build our first stack.

Chapter 2: Your First Stack - A Guided Tutorial

The best way to understand App::FargateStack is to build something with it. In this chapter, we will walk through the entire lifecycle of a simple application stack, from generating the initial configuration to deploying a running service.

Our goal is to deploy a simple daemon service using a pre-existing container image named helloworld from our ECR repository.

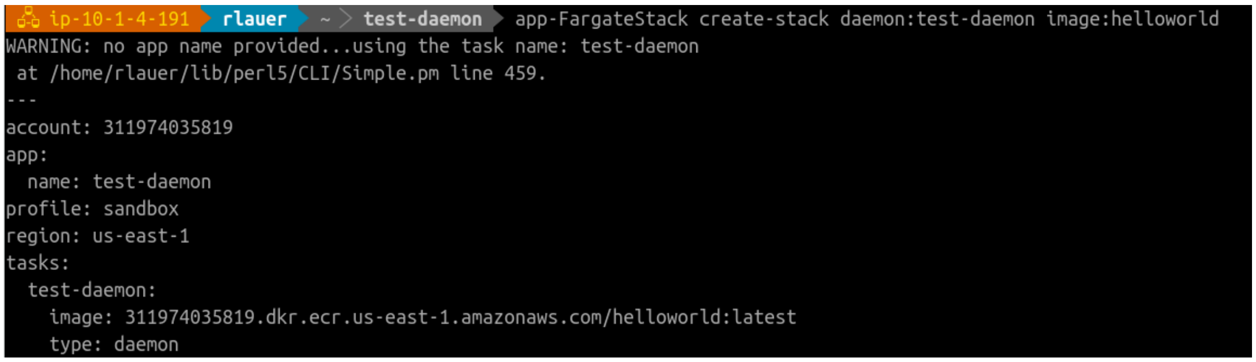
Step 1: Creating the Configuration with create-stack

We begin with the create-stack command. This is a powerful shortcut that generates a starter YAML configuration file based on a few key pieces of information: the service type, a name for it, and the container image.

The Command:

```
App::FargateStack create-stack daemon:test-daemon image:helloworld
```

The Output:



```
 ip-10-1-4-191  rlauer  ~  test-daemon  app-FargateStack create-stack daemon:test-daemon image:helloworld
WARNING: no app name provided...using the task name: test-daemon
 at /home/rlauer/lib/perl5/CLI/Simple.pm line 459.
---
account: 311974035819
app:
  name: test-daemon
profile: sandbox
region: us-east-1
tasks:
  test-daemon:
    image: 311974035819.dkr.ecr.us-east-1.amazonaws.com/helloworld:latest
    type: daemon
```

In this single step, the framework has already done several intelligent things:

- **Inferred the App Name:** It warned us that we didn't provide a top-level application name and defaulted to using our service name, test-daemon.
- **Discovered AWS Context:** It automatically determined our AWS Account ID (311974035819) and Region (us-east-1) from our active AWS profile.
- **Resolved the Image URI:** It took our shorthand image name, helloworld, verified that it exists in our ECR repository, and expanded it to its full, globally unique URI.

This command creates a test-daemon.yml file in our current directory, giving us a perfect starting point.

Step 2: Planning the Infrastructure with plan

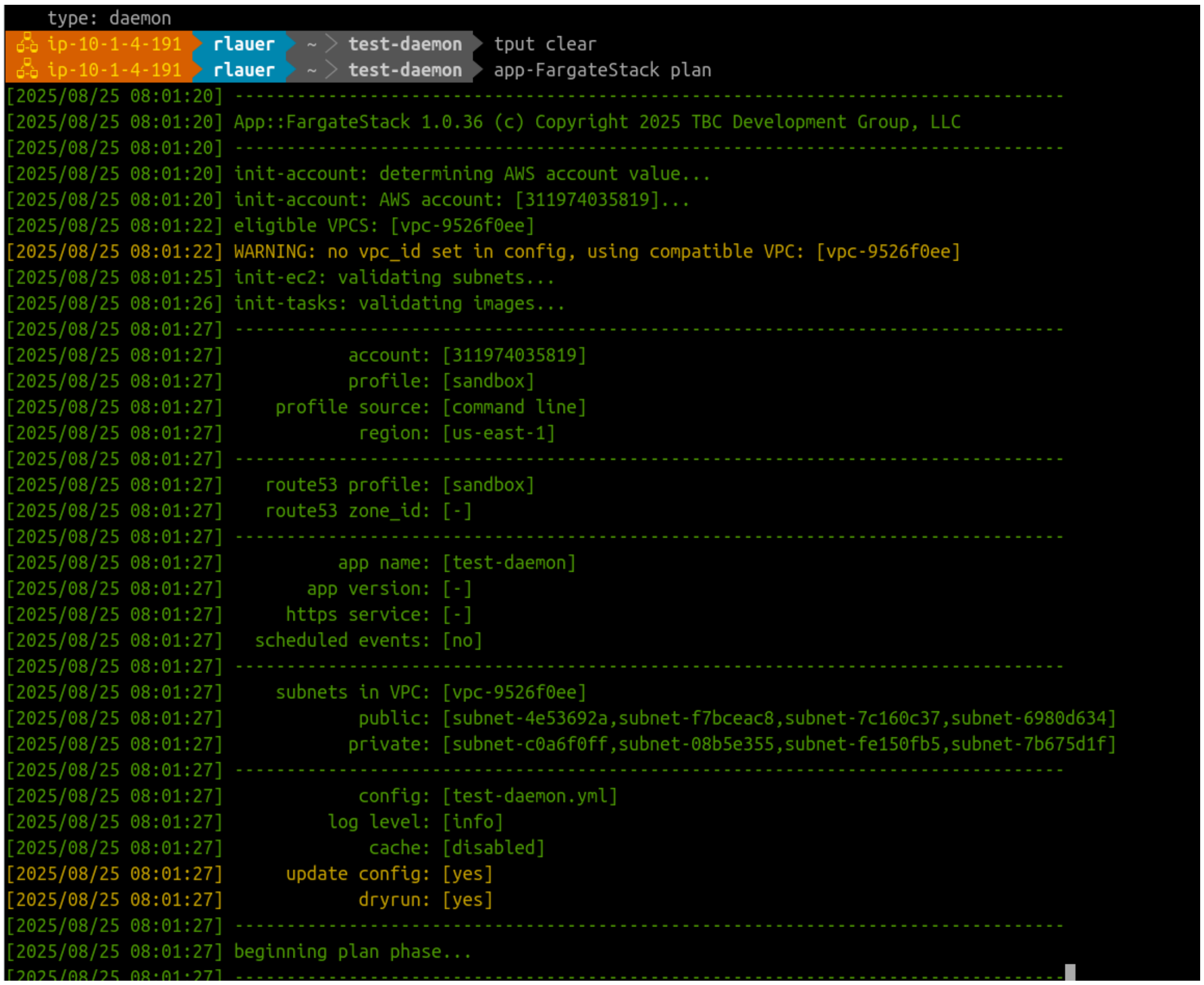
Before we create any resources, we'll perform a dry run using the plan command. This command inspects our AWS environment and shows us a complete blueprint of the stack it intends to build.

The Command:

```
App::FargateStack plan
```

(Note: We don't need to specify `-c test-daemon.yml` because the framework remembers the last configuration file we worked with.)

The Output (abbreviated):



```
    type: daemon
 ip-10-1-4-191  rlauer  ~  test-daemon  tput clear
 ip-10-1-4-191  rlauer  ~  test-daemon  app-FargateStack plan
[2025/08/25 08:01:20] --------------------------------------------------------------------------------
[2025/08/25 08:01:20] App::FargateStack 1.0.36 (c) Copyright 2025 TBC Development Group, LLC
[2025/08/25 08:01:20] --------------------------------------------------------------------------------
[2025/08/25 08:01:20] init-account: determining AWS account value...
[2025/08/25 08:01:20] init-account: AWS account: [311974035819]...
[2025/08/25 08:01:22] eligible VPCS: [vpc-9526f0ee]
[2025/08/25 08:01:22] WARNING: no vpc_id set in config, using compatible VPC: [vpc-9526f0ee]
[2025/08/25 08:01:25] init-ec2: validating subnets...
[2025/08/25 08:01:26] init-tasks: validating images...
[2025/08/25 08:01:27] --------------------------------------------------------------------------------
[2025/08/25 08:01:27]           account: [311974035819]
[2025/08/25 08:01:27]           profile: [sandbox]
[2025/08/25 08:01:27]    profile source: [command line]
[2025/08/25 08:01:27]            region: [us-east-1]
[2025/08/25 08:01:27] --------------------------------------------------------------------------------
[2025/08/25 08:01:27]   route53 profile: [sandbox]
[2025/08/25 08:01:27]   route53 zone_id: [-]
[2025/08/25 08:01:27] --------------------------------------------------------------------------------
[2025/08/25 08:01:27]          app name: [test-daemon]
[2025/08/25 08:01:27]       app version: [-]
[2025/08/25 08:01:27]     https service: [-]
[2025/08/25 08:01:27]  scheduled events: [no]
[2025/08/25 08:01:27] --------------------------------------------------------------------------------
[2025/08/25 08:01:27]    subnets in VPC: [vpc-9526f0ee]
[2025/08/25 08:01:27]            public: [subnet-4e53692a,subnet-f7bceac8,subnet-7c160c37,subnet-6980d634]
[2025/08/25 08:01:27]           private: [subnet-c0a6f0ff,subnet-08b5e355,subnet-fe150fb5,subnet-7b675d1f]
[2025/08/25 08:01:27] --------------------------------------------------------------------------------
[2025/08/25 08:01:27]            config: [test-daemon.yml]
[2025/08/25 08:01:27]         log level: [info]
[2025/08/25 08:01:27]             cache: [disabled]
[2025/08/25 08:01:27]     update config: [yes]
[2025/08/25 08:01:27]            dryrun: [yes]
[2025/08/25 08:01:27] --------------------------------------------------------------------------------
[2025/08/25 08:01:27] beginning plan phase...
```

The `plan` command gives us a wealth of information:

- **Discovery:** It found an eligible VPC and all the public and private subnets within it.
- **Blueprint:** It clearly states which resources (log group, IAM role, cluster, etc.) *will be*

created.
- **Summary:** The final "Required Resources" table is our guarantee—this is the complete list of what `apply` will build.

Our `test-daemon.yml` file has now been updated with all of this discovered information.

Step 3: Provisioning the Stack with `apply`

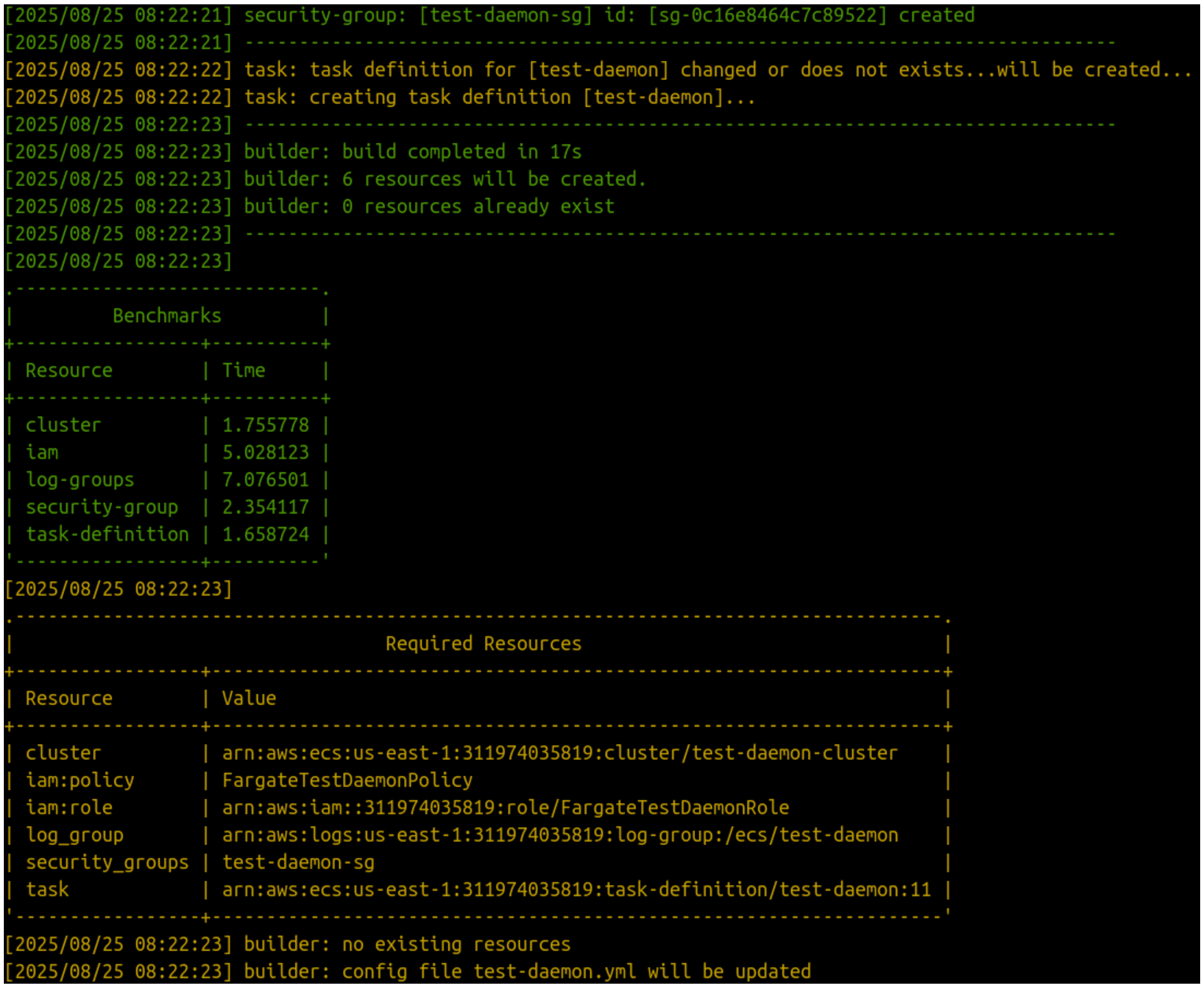
Now that we've reviewed the plan, we can provision the infrastructure using the `apply` command. This will execute the plan we just saw.

The Command:

```
App::FargateStack apply
```

The Output:

The output will look nearly identical to the `plan` command, but without the `(dryrun)` warnings. It will show each resource being created in real-time. The final "Required Resources" table will now be populated with the actual ARNs and IDs of the newly created resources.



```
[2025/08/25 08:22:21] security-group: [test-daemon-sg] id: [sg-0c16e8464c7c89522] created
[2025/08/25 08:22:21] --------------------------------------------------------------------------------
[2025/08/25 08:22:22] task: task definition for [test-daemon] changed or does not exists...will be created...
[2025/08/25 08:22:22] task: creating task definition [test-daemon]...
[2025/08/25 08:22:23] --------------------------------------------------------------------------------
[2025/08/25 08:22:23] builder: build completed in 17s
[2025/08/25 08:22:23] builder: 6 resources will be created.
[2025/08/25 08:22:23] builder: 0 resources already exist
[2025/08/25 08:22:23] --------------------------------------------------------------------------------
[2025/08/25 08:22:23]
.----------------------------.
|         Benchmarks         |
+-----------------+----------+
| Resource        | Time     |
+-----------------+----------+
| cluster         | 1.755778 |
| iam             | 5.028123 |
| log-groups      | 7.076501 |
| security-group  | 2.354117 |
| task-definition | 1.658724 |
'-----------------+----------'
[2025/08/25 08:22:23]
.------------------------------------------------------------------------------------.
|                                 Required Resources                                 |
+-----------------+------------------------------------------------------------------+
| Resource        | Value                                                            |
+-----------------+------------------------------------------------------------------+
| cluster         | arn:aws:ecs:us-east-1:311974035819:cluster/test-daemon-cluster   |
| iam:policy      | FargateTestDaemonPolicy                                          |
| iam:role        | arn:aws:iam::311974035819:role/FargateTestDaemonRole             |
| log_group       | arn:aws:logs:us-east-1:311974035819:log-group:/ecs/test-daemon   |
| security_groups | test-daemon-sg                                                   |
| task            | arn:aws:ecs:us-east-1:311974035819:task-definition/test-daemon:11 |
'-----------------+------------------------------------------------------------------'
[2025/08/25 08:22:23] builder: no existing resources
[2025/08/25 08:22:23] builder: config file test-daemon.yml will be updated
```

Our infrastructure is now live. All the foundational plumbing is in place.

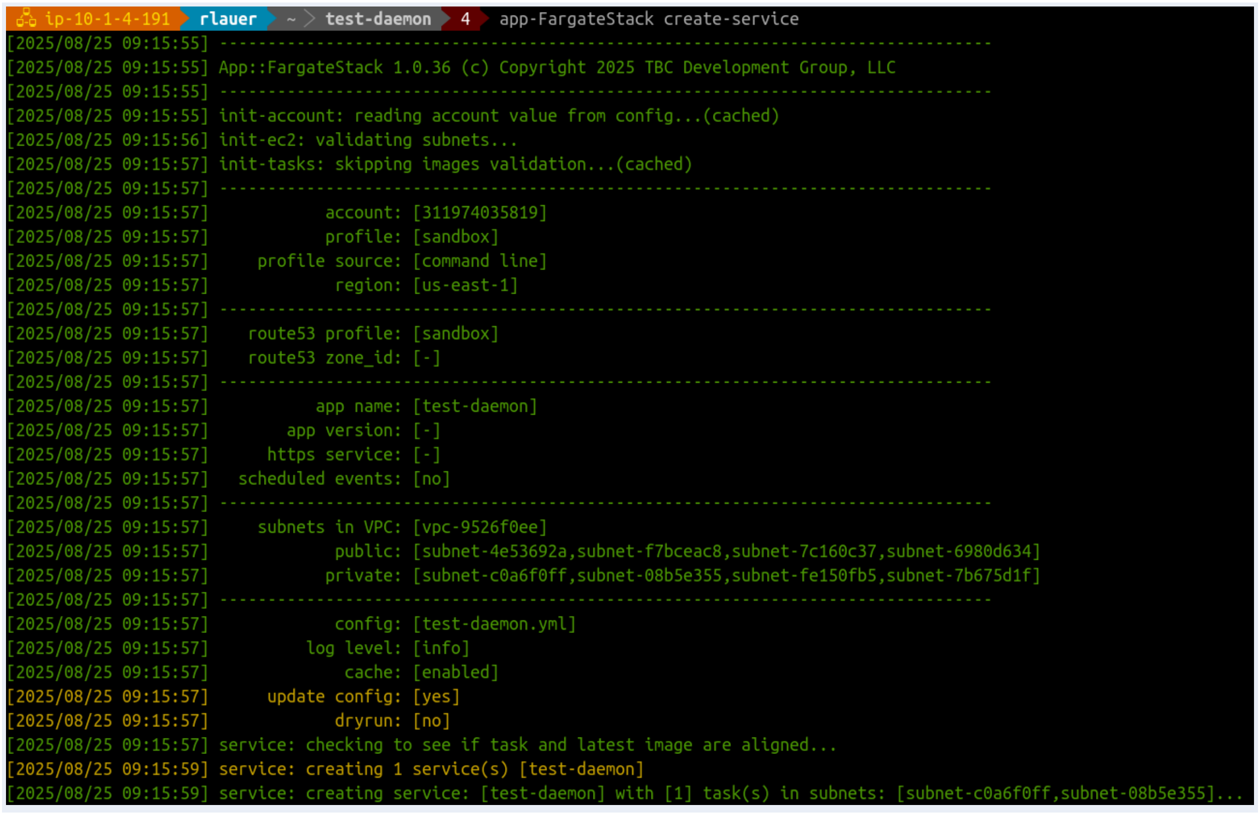
Step 4: Launching and Verifying the Service

The `apply` command creates the infrastructure, but it doesn't start the service. This is a deliberate design choice that separates infrastructure management from application runtime management.

To launch our daemon the first time, we use the `create-service` command.

The Command:

```
App::FargateStack create-service
```



```
ip-10-1-4-191  rlauer  ~  test-daemon  4  app-FargateStack create-service
[2025/08/25 09:15:55] --------------------------------------------------------------------------------
[2025/08/25 09:15:55] App::FargateStack 1.0.36 (c) Copyright 2025 TBC Development Group, LLC
[2025/08/25 09:15:55] --------------------------------------------------------------------------------
[2025/08/25 09:15:55] init-account: reading account value from config...(cached)
[2025/08/25 09:15:56] init-ec2: validating subnets...
[2025/08/25 09:15:57] init-tasks: skipping images validation...(cached)
[2025/08/25 09:15:57] --------------------------------------------------------------------------------
[2025/08/25 09:15:57]            account: [311974035819]
[2025/08/25 09:15:57]            profile: [sandbox]
[2025/08/25 09:15:57]     profile source: [command line]
[2025/08/25 09:15:57]             region: [us-east-1]
[2025/08/25 09:15:57] --------------------------------------------------------------------------------
[2025/08/25 09:15:57]    route53 profile: [sandbox]
[2025/08/25 09:15:57]    route53 zone_id: [-]
[2025/08/25 09:15:57] --------------------------------------------------------------------------------
[2025/08/25 09:15:57]           app name: [test-daemon]
[2025/08/25 09:15:57]        app version: [-]
[2025/08/25 09:15:57]      https service: [-]
[2025/08/25 09:15:57]   scheduled events: [no]
[2025/08/25 09:15:57] --------------------------------------------------------------------------------
[2025/08/25 09:15:57]     subnets in VPC: [vpc-9526f0ee]
[2025/08/25 09:15:57]             public: [subnet-4e53692a,subnet-f7bceac8,subnet-7c160c37,subnet-6980d634]
[2025/08/25 09:15:57]            private: [subnet-c0a6f0ff,subnet-08b5e355,subnet-fe150fb5,subnet-7b675d1f]
[2025/08/25 09:15:57] --------------------------------------------------------------------------------
[2025/08/25 09:15:57]             config: [test-daemon.yml]
[2025/08/25 09:15:57]          log level: [info]
[2025/08/25 09:15:57]              cache: [enabled]
[2025/08/25 09:15:57]      update config: [yes]
[2025/08/25 09:15:57]             dryrun: [no]
[2025/08/25 09:15:57] service: checking to see if task and latest image are aligned...
[2025/08/25 09:15:59] service: creating 1 service(s) [test-daemon]
[2025/08/25 09:15:59] service: creating service: [test-daemon] with [1] task(s) in subnets: [subnet-c0a6f0ff,subnet-08b5e355]...
```

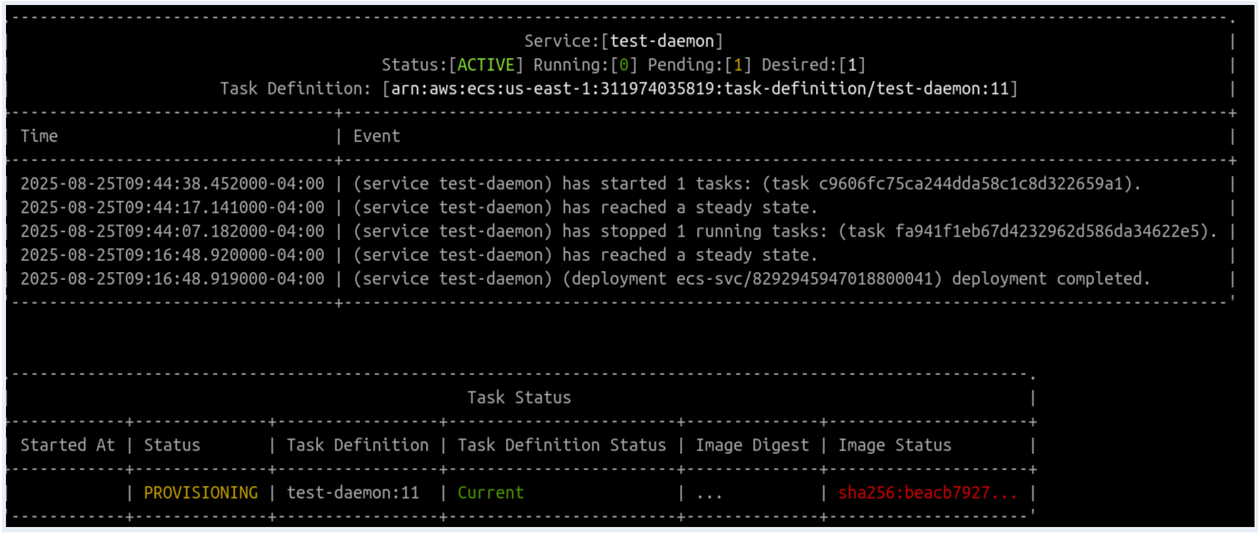
This tells ECS to launch one instance of our `test-daemon` task. Because it's a managed service, Fargate will ensure it stays running. If the task fails, Fargate will automatically restart it.

We can immediately check its status:

The Command:

```
App::FargateStack status
```

The Output:



```
---------------------------------------------------------------------------------------------------------------------------------.
|                                                     Service:[test-daemon]                                                      |
|                                       Status:[ACTIVE] Running:[0] Pending:[1] Desired:[1]                                      |
|                      Task Definition: [arn:aws:ecs:us-east-1:311974035819:task-definition/test-daemon:11]                      |
+----------------------------------+---------------------------------------------------------------------------------------------+
| Time                             | Event                                                                                       |
+----------------------------------+---------------------------------------------------------------------------------------------+
| 2025-08-25T09:44:38.452000-04:00 | (service test-daemon) has started 1 tasks: (task c9606fc75ca244dda58c1c8d322659a1).         |
| 2025-08-25T09:44:17.141000-04:00 | (service test-daemon) has reached a steady state.                                           |
| 2025-08-25T09:44:07.182000-04:00 | (service test-daemon) has stopped 1 running tasks: (task fa941f1eb67d4232962d586da34622e5). |
| 2025-08-25T09:16:48.920000-04:00 | (service test-daemon) has reached a steady state.                                           |
| 2025-08-25T09:16:48.919000-04:00 | (service test-daemon) (deployment ecs-svc/8292945947018800041) deployment completed.        |
-----------------------------------+---------------------------------------------------------------------------------------------'


------------------------------------------------------------------------------------------------------------.
|                                                Task Status                                                |
+------------+--------------+-----------------+------------------------+--------------+---------------------+
| Started At | Status       | Task Definition | Task Definition Status | Image Digest | Image Status        |
+------------+--------------+-----------------+------------------------+--------------+---------------------+
|            | PROVISIONING | test-daemon:11  | Current                | ...          | sha256:beacb7927... |
-------------+--------------+-----------------+------------------------+--------------+---------------------'
```

Success! In four commands, we have gone from a simple idea to a fully provisioned, managed, and running Fargate service.

Chapter 3: The Configuration File in Detail

At the heart of every App::FargateStack deployment is a single YAML file. This configuration file is the source of truth for your entire stack. While the framework can discover and populate many of its values for you, understanding the structure of this file is key to unlocking the full power of the tool.

This chapter provides a section-by-section breakdown of the fargate-stack.yml schema.

Top-Level Configuration

These keys define the overall context and properties of your application stack.

```
# Top-Level Keys
account: 311974035819
profile: sandbox
region: us-east-1
vpc_id: vpc-9526f0ee
app:
  name: test-daemon
domain: my-app.example.com # Required for HTTP/S services
route53:
  profile: dns-management-profile
  zone_id: Z0123456789ABCDEFGHIJ
```

- **account**, **profile**, **region**: Define the AWS account and region where the stack will be deployed. These are typically discovered and populated for you.
- **vpc_id**: The ID of the VPC to deploy into. If omitted, the framework will attempt to find a single, eligible VPC in your account.
- **app**: A namespace for application-level metadata.
  - **name** (Required): The global name for your application stack. This is used to derive default names for many resources (e.g., my-app-cluster, my-app-sg).
- **domain**: The fully qualified domain name for your service. This is **required** if you are deploying an http or https task.
- **route53**: Configuration for DNS management.
  - **profile**: Use this if your Route 53 hosted zones are in a different AWS account than your Fargate resources.
  - **zone_id**: The ID of the Route 53 hosted zone for your domain. If omitted, the framework will attempt to discover it.

The tasks Section

This is the most important section of the file. It is a map where each key is the name of a service or job you want to run, and the value is an object describing its configuration.

```
tasks:
  test-daemon:
    # Core Configuration
    type: daemon
    image: 311974035819.dkr.ecr.us-east-1.amazonaws.com/helloworld:latest

    # Resource Sizing
    size: medium
    cpu: 1024
    memory: 2048

    # Scheduling (for type: task)
    schedule: 'cron(0 18 * * ? *)'

    # Networking (for type: http/https)
    port: 8080

    # Environment & Secrets
    environment:
      LOG_LEVEL: info
    secrets:
      - /myapp/database/password:DB_PASSWORD

    # Storage
    efs:
      id: fs-12345678
      mount_point: /data
```

- **type** (Required): Defines the workload pattern. Valid values are:
  - daemon: A long-running background service.
  - task: A one-shot or scheduled job.
  - http / https A web service fronted by an Application Load Balancer.
- **image** (Required): The full URI of the Docker container image to run.
- **size, cpu, memory**: Defines the vCPU and memory allocated to the task. You can specify a predefined size (e.g., tiny, small, medium) or provide specific cpu and memory values.
- **schedule**: For tasks of type: task, this key turns it into a scheduled job. The value must be a valid AWS EventBridge cron() or rate() expression.
- **port**: For http/https services, this specifies the port your container listens on. Defaults

to `80`.

- **`environment`**: A map of key-value pairs to be injected into the container as environment variables. Use this for non-sensitive configuration.
- **`secrets`**: A list of secrets to be securely injected from AWS Secrets Manager.
- **`efs`**: Configuration for mounting an EFS file system into the container.

By combining these sections, you can define a complete, multi-service application stack in a single, readable file.

Chapter 4: Core Workload Patterns

While `App::FargateStack` can orchestrate a wide variety of AWS resources, its primary focus is on running your containerized applications. The framework supports several distinct workload patterns, each tailored to a specific use case. Your primary decision when defining a new service is choosing its `type`.

This chapter will explore each of the core patterns in detail, with configuration examples and explanations of the infrastructure that gets provisioned for each.

4.1 Understanding Task Types

The `type` key in your task configuration is the most important setting. It tells the framework what kind of application you intend to run and determines the entire set of AWS resources that will be built to support it.

There are four primary types:

- **`daemon`**: For long-running background services that should always be active. `App::FargateStack` will create an ECS Service to ensure the desired number of tasks is always running. If a task fails, ECS will automatically launch a replacement. This is ideal for message queue consumers, data processors, or any persistent background worker.
- **`task`**: For jobs that run to completion. This is used for two main scenarios:
  - **Ad-Hoc Jobs**: You can trigger these manually with the `run-task` command. They are perfect for database migrations, one-off administrative scripts, or debugging.
  - **Scheduled Jobs**: By adding a `schedule` key (e.g., `schedule: 'cron(0 18 * * ? *)'`), you transform the task into a recurring job managed by AWS EventBridge. This is ideal for nightly reports, batch processing, or any cron-like workflow.
- **`http`** & **`https`**: For web applications or APIs. These types tell the framework to provision a full, production-ready web stack, including:
  - An Application Load Balancer (ALB) to distribute traffic.
  - Target Groups and Listener Rules.
  - An ECS Service to manage the running tasks.
  - For `https` it will also provision an ACM certificate and create an alias record in Route 53 for your custom domain.

In the following sections, we will dive deep into each of these patterns.

4.2 Daemon Services

A daemon is a long-running background process that is not directly accessible from the internet. These are the workhorses of many application backends, responsible for tasks like processing messages from an SQS queue, handling data streams, or performing continuous calculations.

When you specify type: daemon, you are telling App::FargateStack that you need a resilient, highly-available service.

Minimal Configuration:

```
tasks:
  my-queue-processor:
    type: daemon
    image: my-app/queue-processor:1.2.3
```

What App::FargateStack Provisions:

Based on this simple configuration, the framework creates a robust environment for your daemon:

- **ECS Service**: This is the key component. An ECS Service is created to ensure that the desired number of tasks (by default, one) is always running. If your container crashes or the underlying Fargate infrastructure experiences an issue, the ECS scheduler will automatically launch a replacement task to maintain availability.
- **Dedicated IAM Role**: A role is created with the necessary permissions for the ECS agent to pull your image from ECR and write logs to CloudWatch. The policy will be automatically expanded if you configure other resources like SQS queues or S3 buckets.
- **CloudWatch Log Group**: All stdout and stderr from your container are automatically streamed to a dedicated log group, allowing for centralized logging and monitoring.

Lifecycle Management:

Because a daemon is a managed service, you interact with it using service-level commands:

- App::FargateStack start-service my-queue-processor [count]: Deploys and starts the service.
- App::FargateStack stop-service my-queue-processor: Stops the service by setting its desired count to zero.
- App::FargateStack status my-queue-processor: Checks the health and status of the running service.

4.3 Tasks: Ad-Hoc and Scheduled Jobs

The task type is the most versatile workload pattern. It represents a container that is designed to run for a period of time and then exit. Unlike a daemon, a task is not automatically restarted by an ECS service if it stops.

This pattern serves two distinct but related use cases: running a job on-demand, and running a job on a recurring schedule.

Ad-Hoc Jobs

An ad-hoc job is a task you trigger manually. This is the perfect pattern for administrative scripts, database migrations, data import/export routines, or any one-off process.

Minimal Configuration:

```
tasks:
  db-migration:
    type: task
    image: my-app/db-tools:latest
```

Lifecycle Management:

You interact with ad-hoc jobs using the `run-task` command:

- `App::FargateStack run-task db-migration`: Launches a single instance of the task. By default, the command will wait for the task to complete and stream its logs to your terminal, which is ideal for monitoring its progress.
- `App::FargateStack run-task db-migration --no-wait`: Launches the task and exits immediately, allowing the job to run in the background.

Scheduled Jobs

By adding a single `schedule` key to a task definition, you transform it from an ad-hoc job into a recurring, scheduled job.

Minimal Configuration:

```
tasks:
  nightly-report:
    type: task
    image: my-app/report-generator:1.0
    schedule: 'cron(0 2 * * ? *)' # Runs every day at 2:00 AM UTC
```

What `App::FargateStack` Provisions:

In addition to the standard resources (IAM Role, Log Group, etc.), specifying a `schedule` tells the framework to create and configure **AWS EventBridge** resources:

- **EventBridge Rule**: An EventBridge rule is created with the schedule you specified.
- **EventBridge Target**: The Fargate cluster and task definition are set as the target for

the rule. When the schedule is met, EventBridge invokes the Fargate `RunTask` API to launch your job.

- **Dedicated IAM Role for Events**: A separate, secure IAM role is created specifically for EventBridge, granting it the minimum necessary permissions to launch your task on your behalf.

Lifecycle Management:

While you can still run a scheduled job manually with `run-task`, you also gain commands to manage its schedule:

- `App::FargateStack disable-scheduled-task nightly-report`: Pauses the schedule without deleting any resources.
- `App::FargateStack enable-scheduled-task nightly-report`: Resumes a paused schedule.

4.4 HTTP & HTTPS Services

This is the most powerful and comprehensive workload pattern in `App::FargateStack`. When you need to expose your container to the world (or your internal network) as a web application or API, you will use the `http` or `https` type.

This tells the framework to provision a complete, production-ready, and load-balanced web stack.

Minimal Configuration:

```
app:
  name: my-web-app
domain: my-app.example.com # Required for http/https services
route53:
  zone_id: Z0123456789ABCDEFGHIJ # Required if not discoverable

tasks:
  apache-web-server:
    type: https
    image: my-company/web-app:production
```

What `App::FargateStack` Provisions:

This is where the framework's role as an "infrastructure accelerator" is most apparent. Based on this simple configuration, it provisions and connects an entire suite of AWS resources:

- **Application Load Balancer (ALB)**: An ALB is either discovered or created to act as the front door for your application. For `https` services, it will be an internet-facing ALB in public subnets. For `http` services, it will be an internal ALB for private traffic.
- **Target Group & Listener Rules**: The framework configures a target group to route traffic to your Fargate tasks and creates listener rules on the ALB (e.g., on port 443 for HTTPS) to direct traffic based on the hostname.
- **ECS Service**: Just like a `daemon`, an ECS Service is created to ensure your web server tasks are always running and are automatically replaced if they fail.
- **Security Groups**: Multiple security groups are created and configured to allow traffic to flow securely from the ALB to your Fargate tasks, while blocking all other access.
- **Route 53 Alias Record**: An `A` record is created in your hosted zone, pointing your custom domain (`my-app.example.com`) to the ALB.
- **ACM Certificate** (for `https` only): If you specify `type: https`, the framework will automatically provision a free, auto-renewing SSL/TLS certificate from AWS Certificate Manager and attach it to the ALB listener, enabling secure traffic.

Architectural Flow

The architecture follows a standard, secure pattern for web services on AWS.

```
                   Internet
                       |
                       V
+--------------------------------------------+
|            AWS Cloud                       |
|                                            |
|  +----------+      +------------------+    |
|  | Route 53 |----->| App Load Balancer|    |
|  | (DNS)    |      | (Public Subnet)  |    |
|  +----------+      +-------+----------+    |
|                              |             |
|                              V             |
|                      +---------------+     |
|                      | Target Group  |     |
|                      +-------+-------+     |
|                              |             |
|                              V             |
|  +---------------------------------------+ |
|  | Fargate Task 1 | Fargate Task 2 | ... | |
|  | (Private Subnet)                      | |
|  +---------------------------------------+ |
|                                            |
+--------------------------------------------+
```

1. **User Request & DNS**:
   - An **Internet Client** makes a request to your domain (e.g., my-app.example.com).
   - **Route 53** receives the request, looks up the A (Alias) record for your domain, and resolves it to the DNS name of the Application Load Balancer.
2. **Load Balancing & Security**:
   - The request hits the **Application Load Balancer (ALB)**, which lives in your VPC's **Public Subnets**.
   - The ALB's **Listener** on port 443 (HTTPS) receives the traffic.
   - An **ACM Certificate** attached to the listener terminates the SSL/TLS connection, decrypting the request.
   - A rule on the listener forwards the decrypted traffic to a specific Target Group.
3. **Application Tier**:
   - The **Target Group** routes the traffic to a healthy Fargate task. It knows the private IP addresses of your containers and continuously monitors their health.
   - Your **Fargate Tasks** (containers) run securely in the VPC's **Private Subnets**. They do not have public IP addresses and cannot be reached directly from the internet.

    - An **ECS Service** ensures that the desired number of tasks is always running. If a task fails, the service replaces it automatically.

Lifecycle Management:

Like daemons, HTTP/S services are managed with service-level commands:

- `App::FargateStack start-service apache-web-server [count]`: Deploys and starts the web service.
- `App::FargateStack stop-service apache-web-server`: Stops the service.
- `App::FargateStack status apache-web-server`: Checks the health and status of the running service and its tasks.

Chapter 5: Advanced Topics & Integrations

With a solid understanding of the core workload patterns, we can now explore the more advanced features of App::FargateStack. This chapter delves into the details of how the framework handles networking, security, storage, and messaging.

These are the components that elevate your application from a simple container into a fully-featured, production-grade stack. While the framework provides sensible defaults for all of these, understanding how to configure them is key to customizing your deployment for your specific needs.

5.1 Networking: VPC and Subnet Discovery

Proper networking is the foundation of any secure and scalable cloud application. App::FargateStack is designed to work with your existing network infrastructure, intelligently discovering and utilizing your VPCs and subnets.

VPC Discovery

If you do not specify a vpc_id in your configuration file, the framework will automatically scan your AWS account to find an eligible VPC. A VPC is considered "eligible" if it has the necessary components for Fargate tasks to function, namely:

- An **Internet Gateway (IGW)** to allow communication with the outside world.
- At least one **NAT Gateway**, which enables tasks in private subnets to initiate outbound connections (e.g., to pull container images or call external APIs).

If exactly one eligible VPC is found, App::FargateStack will use it by default and update your configuration file with its ID. If multiple eligible VPCs are found, the plan will fail with an error, prompting you to resolve the ambiguity by explicitly setting the vpc_id in your configuration.

Subnet Categorization

Once a VPC is selected, the framework inspects its route tables to categorize every subnet as either **public** or **private**.

- **Public Subnets**: Have a direct route to an Internet Gateway. Resources in a public subnet can be directly accessible from the internet.
- **Private Subnets**: Have a route to a NAT Gateway, but not an Internet Gateway. Resources in a private subnet can make outbound connections to the internet, but the internet cannot initiate connections to them.

This discovery process is crucial. The framework uses this information to follow best practices for task placement. For example, https services will have their load balancers placed in public subnets, while the Fargate tasks themselves are placed in private subnets for security. The discovered subnets are automatically added to your configuration file.

5.2 Security: IAM, Security Groups, and Secrets

`App::FargateStack` is built on the principle of "secure by default." It automates the creation of IAM roles, policies, and security groups to ensure your application adheres to the principle of least privilege.

IAM Roles and Policies

The framework creates a single IAM role for all tasks within a stack. This role is automatically granted the minimum permissions required for your tasks to function, based on the resources you define in your configuration file.

- **Automatic Policy Generation**: You do not need to write IAM policies by hand. If you define an SQS queue, the framework adds `sqs:*` permissions for that specific queue to the role. If you define an S3 bucket, it adds the appropriate S3 permissions. This dynamic policy generation ensures your tasks have exactly the permissions they need, and no more.
- **Trust Relationships**: The framework automatically configures the trust policy, allowing the ECS service to assume the role on your behalf. For scheduled jobs, a separate, even more restrictive role is created for EventBridge.

Security Groups

A dedicated security group is provisioned for your Fargate cluster. By default, this group has no inbound rules and allows all outbound traffic. Ingress rules are added automatically based on your task configuration. For example, if you create an `https` service, a rule is added to allow inbound traffic on the container port *only* from the Application Load Balancer's security group. This ensures that the only traffic reaching your tasks is the traffic that has been processed by the load balancer.

Injecting Secrets from Secrets Manager

Placing sensitive information like database passwords or API keys directly into your configuration file or environment variables is a security risk. `App::FargateStack` integrates directly with **AWS Secrets Manager** to solve this problem.

You can securely inject secrets into your container's environment by defining a `secrets` block in your task configuration.

Configuration:

```
tasks:
  my-worker:
    type: daemon
    image: my-app/worker:latest
    secrets:
      - /my-app/prod/database/password:DB_PASSWORD
```

How it Works:

For each entry, the framework will:

1. Look up the secret `/my-app/prod/database/password` in Secrets Manager.
2. Grant the task's IAM role permission to read this specific secret.
3. Inject the value of the secret into the container's environment as the `DB_PASSWORD` environment variable.

This process is secure because the secret's value is never written to the task definition or your configuration file. It is fetched securely by the ECS agent at runtime and passed directly to your container.

5.3 Storage: S3 Buckets and EFS Mounts

For many applications, ephemeral container storage isn't enough. App::FargateStack provides first-class integrations for two primary AWS storage services: Amazon S3 for object storage and Amazon EFS for shared file system storage.

S3 Buckets

You can define a single S3 bucket for your stack, which is useful for storing user uploads, logs, or application artifacts. The framework will either create a new bucket or use an existing one, and it will automatically configure the necessary IAM permissions.

Configuration:

```
bucket:
  name: my-app-data-bucket
  readonly: true # Optional: defaults to false
  paths:         # Optional: restricts access to specific prefixes
    - public/*
    - processed/*
```

- **name**: The globally unique name for your S3 bucket.
- **readonly**: If set to true, tasks will only be granted s3:GetObject and s3:ListBucket permissions. If false (the default), full read/write permissions are granted.
- **paths**: An optional list of key prefixes. If specified, the IAM policy will be scoped down to allow access *only* to objects within these prefixes.

EFS File Systems

When you need a persistent, shared file system that can be mounted by multiple tasks simultaneously, EFS is the solution. This is ideal for content management systems, shared data processing workloads, or any application that needs a traditional file system.

App::FargateStack does not provision the EFS file system itself, but it will validate that an existing file system is available and configure your tasks to use it.

Configuration (within a task definition):

```
tasks:
  my-cms-app:
    type: https
    image: my-company/cms:latest
    efs:
      id: fs-12345678 # The ID of your existing EFS file system
      mount_point: /var/www/html/uploads # Path inside the container
      path: /app-uploads # Path on the EFS volume
      readonly: false
      authorize_ingress: true # Optional: Automates security group rules
```

Based on this configuration, the framework will automatically:

1. **Update IAM Policies**: Grant the task's IAM role the necessary permissions to connect to and interact with the specified EFS file system.
2. **Configure Task Definitions**: Add the volume and mount point configuration to the ECS task definition, making the file system available inside your container.

Automating Network Access with `authorize_ingress`

For a Fargate task to successfully mount an EFS volume, the security groups must be configured to allow NFS traffic (on TCP port 2049) between them. This is a common point of failure, as a missing rule will cause the task to fail to start with a timeout error.

To solve this, `App::FargateStack` offers the `authorize_ingress` flag. When you set `authorize_ingress: true`, the framework will:

- **Discover** the security group(s) attached to the EFS mount targets.
- **Automatically add** an inbound rule to each EFS security group, allowing traffic on port 2049 specifically from the Fargate task's security group.
- **Automatically revoke** this rule when you delete the task, ensuring your security posture remains clean.

This opt-in feature automates a critical but error-prone step, turning a frustrating networking problem into a simple configuration flag.

5.4 Messaging: SQS Queues

For applications that rely on asynchronous processing, `App::FargateStack` provides built-in support for Amazon SQS. You can define a primary queue for your stack and an optional Dead

Letter Queue (DLQ) to handle failed messages.

Configuration:

```
queue:
  name: my-app-work-queue
  max_receive_count: 5 # This key automatically enables a DLQ

dlq:
  name: my-app-work-queue-dlq # Optional: defaults to <queue_name>-dlq
```

- **queue**: Defines the primary SQS queue.
  - **name**: The name of your queue.
  - **max_receive_count**: This is the magic key. If you define it, the framework will automatically provision a DLQ and configure the main queue's redrive policy to send messages there after the specified number of failed processing attempts.
- **dlq**: An optional block to customize the Dead Letter Queue. If omitted, a DLQ will still be created with a default name if `max_receive_count` is set.

As with other resources, the framework automatically updates the stack's IAM policy to grant tasks full access to the created queues.

Custom Queue Attributes

While the framework provides sensible defaults for queue attributes, you have full control to override them. You can specify any valid SQS queue attribute directly in the `queue` or `dlq` configuration blocks.

Example:

```
queue:
  name: fu-man-q
  visibility_timeout: 60
  delay_seconds: 5
  receive_message_wait_time_seconds: 20
  message_retention_period: 1209600 # 14 days
  maximum_message_size: 262144
  max_receive_count: 5
```

This gives you the flexibility to tune your queues for specific use cases, such as long polling or larger message sizes, without leaving the simple YAML configuration.

5.5 Environment Variables & Secrets

Injecting configuration into your container at runtime is a fundamental requirement for building flexible applications. `App::FargateStack` provides two distinct mechanisms for this, each designed for a specific type of data: the `environment` block for non-sensitive configuration, and the `secrets` block for sensitive data.

The `environment` Block: For Non-Sensitive Configuration

The `environment` block is a simple key-value map that injects standard environment variables into your container. This is the ideal place for non-sensitive, application-level configuration that you might want to change between environments.

Use Cases:

- Setting a `LOG_LEVEL` for your application.
- Specifying an `ENVIRONMENT` name like `development` or `production`.
- Passing feature flags or other runtime toggles.

Configuration (within a task definition):

```
tasks:
  my-worker:
    type: daemon
    image: my-app/worker:latest
    environment:
      LOG_LEVEL: info
      FEATURE_FLAG_X: true
```

Security Note: Values in the `environment` block are stored in plaintext in the ECS task definition. **Never** place passwords, API keys, or any other sensitive credentials in this section.

The `secrets` Block: For Sensitive Credentials

For sensitive data, `App::FargateStack` provides a secure integration with **AWS Secrets Manager**. This mechanism ensures that your secrets are never exposed in plaintext.

Configuration (within a task definition):

```
tasks:
  my-worker:
    type: daemon
    image: my-app/worker:latest
    secrets:
      - /my-app/prod/database/password:DB_PASSWORD
      - /my-app/prod/api-key:THIRD_PARTY_API_KEY
```

When you use the secrets block, the framework automatically updates the task's IAM role to grant read access to those specific secrets. At runtime, the ECS agent securely fetches the secret values and injects them into your container as environment variables, just like the environment block. The crucial difference is that the values themselves are never visible in any configuration file or task definition.

Chapter 6: Command Reference

This chapter provides a detailed reference for every command available in the `App::FargateStack` CLI. While the `plan` and `apply` commands are the core of the workflow, these utilities give you fine-grained control over the entire lifecycle of your application stack.

Core Workflow Commands

These are the primary commands you will use to manage your infrastructure.

`plan`

Usage: `App::FargateStack plan`

Performs a dry run. It reads your configuration file, inspects your live AWS environment, and produces a detailed report of the resources it will create or update. No changes are made to your AWS account. This command is essential for safely reviewing changes before they are executed.

`apply`

Usage: `App::FargateStack apply`

Provisions or updates your infrastructure to match the state defined in your configuration file. It executes the blueprint generated by the `plan` command. After a successful run, your YAML configuration file is updated with the ARNs and IDs of the provisioned resources.

Service & Task Lifecycle Commands

These commands are used to manage the runtime state of your applications.

`start-service`

Usage: `App::FargateStack start-service [service-name] [count]`

Starts an ECS service (for `daemon`, `http`, or `https` types). The `count` parameter specifies the desired number of running tasks (defaults to 1).

`stop-service`

Usage: `App::FargateStack stop-service [service-name]`

Stops a running service by setting its desired task count to 0. The underlying infrastructure is not removed.

`redeploy`

Usage: `App::FargateStack redeploy [service-name]`

Forces a new deployment of a running service. This is useful for forcing ECS to pull the latest version of a container image tag (e.g., `:latest`) without creating a new task definition revision.

`run-task`

Usage: `App::FargateStack run-task [task-name]`

Launches a one-shot, ad-hoc task (for `type: task`). By default, it waits for the task to complete and streams its logs. Use `--no-wait` to run the task in the background.

`stop-task`

Usage: `App::FargateStack stop-task [task-arn|task-id]`

Stops a specific, running Fargate task. You can get the task ID from the `list-tasks` command.

State & Information Commands

These commands help you inspect and manage your stack.

`status`

Usage: `App::FargateStack status [service-name]`

Displays the current status of a running service, including the desired and running task counts, and the most recent service events.

`list-tasks`

Usage: `App::FargateStack list-tasks [stoppped]`

Lists all currently running or stopped tasks in your cluster, showing their status, resource utilization, and start time.

`logs`

Usage: `App::FargateStack logs [task-name] [start-time] [end-time]`

Streams logs from CloudWatch for a specific task. You can specify a time range (e.g., `5m` for the last 5 minutes) and use `--log-wait` to tail the logs in real-time.

`state`

Usage: `App::FargateStack state [config-file.yml]`

Manages the default configuration file used by the framework. Running it without an argument shows the current defaults. Providing a filename sets it as the new default.

Configuration & Definition Commands

These commands are used for managing the underlying definitions of your services.

`create-stack`

Usage: `App::FargateStack create-stack [app-name] daemon:my-daemon image:my-image ...`

Generates a new `fargate-stack.yml` configuration file from a shorthand syntax. This is the fastest way to get started with a new project.

`register-task-definition`

Usage: `App::FargateStack register-task-definition [task-name]`

Forces the creation of a new revision of an ECS task definition. This is typically done automatically by `plan` and `apply`, but can be run manually if needed.

`update-service`

Usage: `App::FargateStack update-service [service-name]`

Updates an ECS service to use the latest registered task definition revision. This is the key command for deploying a new version of your application image.

Chapter 7: Continue Your Fargate Journey

You've reached the end of this guide, but you're at the beginning of a new way of building on AWS. You've seen how to take a containerized application and, with a few simple commands, deploy it onto a robust, secure, and scalable infrastructure. More importantly, you've seen that the magic behind it isn't magic at all—it's just intelligent, transparent automation. `App::FargateStack` is the expert builder that assembles the plumbing for you, but it always leaves the blueprints on the table for you to inspect, learn from, and modify.

The goal of this guide was to empower you to build on Fargate with confidence. The real journey begins now, as you take these patterns and apply them to your own unique challenges. As you do, you may want to dive deeper into the underlying AWS services that make all of this possible.

Valuable Resources for Deeper Learning

To continue your journey from a builder to a Fargate expert, here are some of the best resources available directly from AWS:

- **The Official AWS Fargate User Guide**: This is the definitive source of truth for all things Fargate. When you need to understand a specific feature or behavior in detail, this is the first place to look.
    - AWS Fargate User Guide
- **ECS Workshop for Fargate**: For hands-on learning that goes beyond this guide, the official ECS Workshop is an invaluable, self-paced lab that covers a wide range of Fargate scenarios.
    - ECS Workshop
- **AWS Containers Blog**: The official AWS blog is the best place to learn about new features, advanced architectural patterns, and best practices from the AWS product teams and expert solution architects.
    - AWS Containers Blog
- **AWS Whitepapers & Guides**: For deep architectural guidance, the whitepapers provide comprehensive and authoritative information on building well-architected systems.
    - AWS Whitepapers & Guides

Join the Community

`App::FargateStack` is an open-source project, and its future will be shaped by builders like you. If you have ideas for new features, find a bug, or want to contribute to the code or documentation, your participation is welcome.

- **Report Issues & Request Features**: App::FargateStack GitHub Issues

Thank you for joining us on this journey. Now, go build something amazing.