# **Implementing a pub/sub architecture with AWS Copilot**

*Rafael Mosca, Associate Solutions Architect, WWPS EMEA Central SA*

The [AWS Copilot CLI](https://aws.github.io/copilot-cli/) is a tool that since its [launch in 2020](https://aws.amazon.com/blogs/containers/introducing-aws-copilot/), developers have been using to build, manage, and operate Linux and Windows containers on [Amazon Elastic Container Service (Amazon ECS)](https://aws.amazon.com/ecs/), [AWS Fargate](https://aws.amazon.com/fargate/), and [AWS App Runner.](https://aws.amazon.com/apprunner/)

In this post, we walk you through how you can use the tool to easily implement a publisher service and subscriber “worker” services on Amazon ECS and AWS Fargate, that respectively publish and consume events in a [pub/sub architecture](https://aws.amazon.com/pub-sub-messaging/).

To illustrate this functionality we

## Overview of solution

Paragraph

Diagram as appropriate, centered

Light background

File size: 250 KB or smaller

Dimensions: 800x400px (no wider than 1000px)

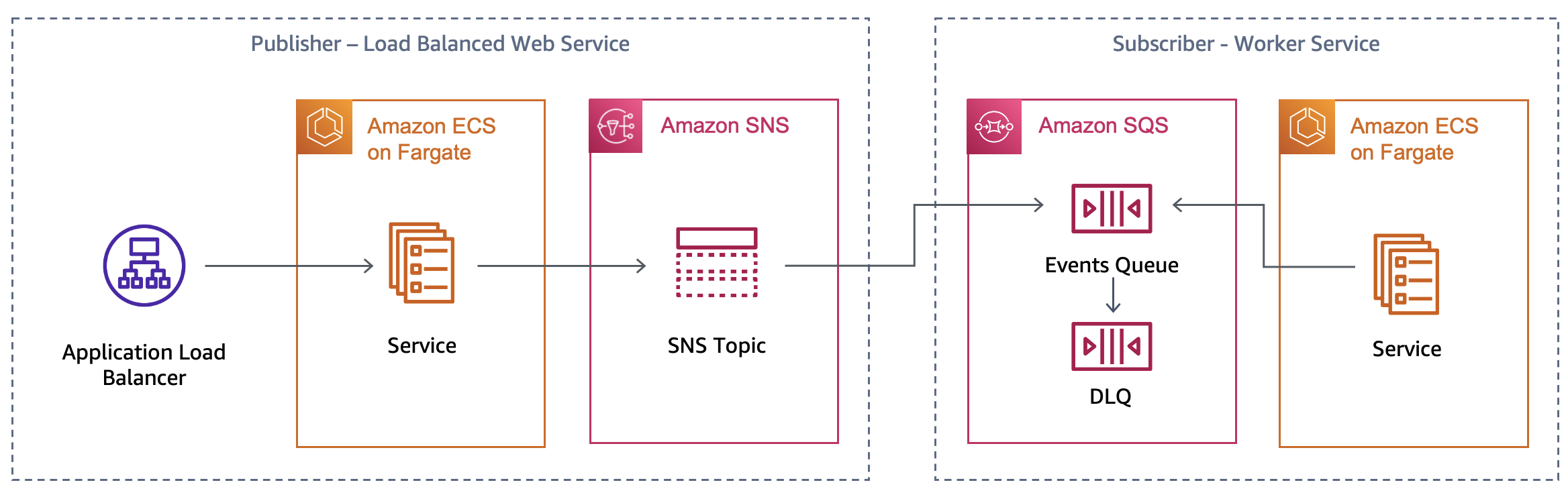
## Context

Publish/subscribe messaging, or pub/sub messaging is an asynchronous messaging pattern that decouples applications by enabling loosely coupled service-to-service communication where messages are exchanged without the need of knowing the identity of the sender or recipient. Senders (also called publishers) broadcast messages to message topics, whereas recipients (called subscribers) subscribe to different topics and receive only the messages published to the topics they are subscribed to and match their filtering policy.

Although it is possible for subscribers to be indefinitely listening to messages from a publisher, it is good practice to apply a pattern known as topic-queue chaining, which as its names indicates, chains an SNS Topic to SQS queues, this way, if a service runs into an exception or needs to undergo maintenance, the messages are persisted into a queue. This has also the advantage that the queue can act as a buffer which ultimately behaves as a load-balancer.

The AWS Copilot CLI allows you to easily implement a Pub/Sub architecture with the topic-queue chaining pattern by allowing you to publish messages to an [Amazon SNS topic](https://docs.aws.amazon.com/sns/latest/dg/sns-create-topic.html) by simply modifying a service manifest and allowing you to create worker services composed of one or more [Amazon SQS queues](https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/welcome.html) to process notifications published to the topics, [dead-letter queues](https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-dead-letter-queues.html) (DLQs) to handle failures and an Amazon ECS service running on AWS Fargate that is able to poll messages from the queue and asynchronously process the messages.

In this blog post we will implement the following architecture using the Load Balanced Web Service and Worker Service abstractions that AWS Copilot CLI provides us.



## Walkthrough

The steps we will follow will be the following:

1. We will clone the sample repository and explore the code.
2. We will create the environment where our microservices will live using the AWS Copilot CLI.
3. Create the publisher and an SNS topic.
4. Create the subscribers, their respective SQS queues, and subscriber policies.
5. Verify how the pub-sub architecture we implemented works.

### Prerequisites

For this walkthrough, you should have the following prerequisites:

* An [AWS account](https://aws.amazon.com/premiumsupport/knowledge-center/create-and-activate-aws-account/).
* Have the [AWS Copilot CLI](https://aws.github.io/copilot-cli/docs/getting-started/install/) installed.
* Properly configured [AWS Credentials](https://aws.github.io/copilot-cli/docs/credentials/) using the [AWS CLI](https://docs.aws.amazon.com/cli/latest/userguide/cli-configure-quickstart.html#cli-configure-quickstart-config) or with [environment variables](https://docs.aws.amazon.com/cli/latest/userguide/cli-configure-envvars.html).
* [Docker](https://www.docker.com/products/docker-desktop) is installed and up and running.

### Clone the sample repository

As a first step, position yourself on the directory where you want to clone the Github repository and perform a git clone.

git clone https://gitlab.aws.dev/rafams/copilot-pubsub.git

Observe the different directories. We have a folder for each service, one for the publisher and one for the two subscribers named invoicing and promotion. The folder structure should look like this:

copilot\_pubsub/

├─ subscribers/

│ ├─ invoicing/

│ │ └─ ...

│ ├─ promotion/

│ │ ├─ requirements.txt

│ │ ├─ promotion.py

│ │ └─ Dockerfile

├─ publisher/

│ └─ ...

### Create an application and environment

As a first thing, we are going to create a logical group of related services, environments, and pipelines we might create. In the AWS Copilot terminology this is called an *application*.

copilot app init pubsub

Once we execute that command, Copilot will use a folder named copilot to hold special YAML configuration files called *manifests* that will help us to easily deploy containerized applications on the AWS cloud.

Our next step is to create an environment for the application where we will deploy our services. With AWS Copilot it is possible to create different environments that logically isolate the deployments of our applications in a very easy way. A common use case is to have a test environment and a separate production environment where applications are deployed only when they have been validated on the test environment. For the scope of this walkthrough, we will only deploy the services to a testing environment named *test* that we create with the following command:

copilot env init \

--app pubsub \

--name test \

--region 'eu-west-1' \

--default-config

Once you press enter on the command, you will be asked to select AWS credentials that will be used to create the necessary infrastructure to host our services. Once the credentials have been selected, Copilot will start to create the resources on you behalf. This process may take a while so stretch a bit while this process is completed.

For every environment you create, AWS Copilot will create a separate networking stack and ECS cluster using the AWS Fargate compute engine.

### Create the publisher

Now that our environment has been deployed and we already have an ECS cluster where we can deploy our applications, we can go ahead and deploy our first microservice named publisher which will send messages to an SNS topic.

As a first thing, explore the directory publisher. You will see that inside there is a Python file that implements the logic and a Dockerfile that is used to build a container image with the code and dependencies.

publisher/

│ ├─ templates

│ │ ├─ index.html

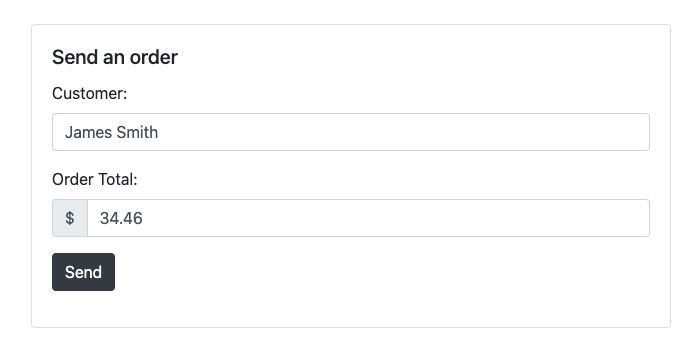
│ │ └─ order.html

│ ├─ requirements.txt

│ ├─ publisher.py

│ └─ Dockerfile

The code is very simple, we use Flask to create a small front-end shown in the image below:



The frontend provides a form with two fields, one for the customer name and another for the order amount. This is just a simple way to trigger the processing of a request. Every time the *Send* button is clicked, the microservice will process the form, save the data of the order to a database and then send message to an SNS topic so that processing can start asynchronously on the subscriber microservices.

To implement such a microservice we would need to create multiple infrastructure components and the process of doing so might be time consuming. In order to help us use our time on developing microservices rather than losing time understanding how to create the underlying infrastructure, AWS Copilot helps us on creating a Load Balancer, an ECR Repository, a Task Definition, an ECS task as well as resources such as SNS Topics and DynamoDB tables with some CLI commands and additional configuration through the Copilot YAML manifest files.

As a first thing we are going to leverage a Copilot pattern called "[Load Balanced Web Service](https://aws.github.io/copilot-cli/docs/concepts/services/#load-balanced-web-service)" and create an ECS service with an ALB that allows it to be publicly accessible. We can do so with the following command:

copilot svc init \

--app pubsub \

--svc-type "Load Balanced Web Service" \

--name "publisher" \

--port 5000 \

--dockerfile "publisher/Dockerfile"

When executed, this command will create a private Amazon ECR repository where the container image will be safely stored, and a manifest file under the copilot directory which includes configuration options for the service.

Remark: ensure the Docker daemon is running at this point or the command will fail.

Before we proceed to deploy the service, observe the manifest.yml file that has been generated in the directory copilot/publisher. Observe how the manifest file holds the configuration for your service, and you can modify the assigned CPU, memory, number of tasks among other things.

For the use case of this demo we need to add two extra resources: an SNS topic where the publisher will send messages to, and a database where the requests will be saved.

To create easily create an SNS Topic with the AWS Copilot CLI we can the following section to the manifest file created for the service.

publish:

topics:

- name: ordersTopic

For each topic we declare, Copilot will create a Standard SNS topic, will inject the ARN of the resource through an environment variable called COPILOT\_SNS\_TOPIC\_ARNS and will give the ECS task the appropriate permissions to publish messages to that topic. The environment variable has a JSON structure where the keys are topic names and each key has as value their respective Topic ARN, so in Python we access these dictionary-like structure by writing:

sns\_topics\_arn = json.loads(os.getenv("COPILOT\_SNS\_TOPIC\_ARNS"))

topic\_arn = sns\_topics\_arn['ordersTopic']

To add the database table we can execute the command:

copilot storage init \

--name ordersTable \

--storage-type DynamoDB \

--workload publisher \

--partition-key id:S \

--no-sort --no-lsi

This will create a file named addons/ordersTable.yml which contains configuration of the DynamoDB table that will be deployed using the AWS Copilot CLI.

Now that we have created all the needed resources we can go ahead and deploy them. The Docker Daemon will be used to build a container image that will then be uploaded to ECR and used as image for the ECS task.

copilot svc deploy --name publisher --env test

After the service creation process is completed, you will receive an ELB address though which you are able to access the service across the internet. Copy this URL as we will use this later.

### Create the first subscriber (invoicing)

In the previous step we have created the publisher that will send the message to the subscribers, now its time to deploy the first subscriber service. As a first thing we create the service by running the command:

copilot svc init \

--app pubsub \

--svc-type "Worker Service" \

--name invoicing \

--port 5000 \

--dockerfile "subscribers/invoicing/Dockerfile"

Note that here we’re using the Worker service abstraction provided by the AWS CLI which provisions an Amazon ECS service running on AWS Fargate, and an SQS queue that acts as a buffer and holds the messages.

When we run the command it is going to ask if we want to subscribe to our already existing SNS topic. We use the spacebar to select the topic and then press Enter.

[x] ordersTopic (publisher)

A new manifest file has been created for the service. Observe that there is now a section where the topics subscriptions have been added.

subscribe:

topics:

- name: ordersTopic

service: publisher

With this configuration, the Copilot CLI will inject an environment variable named COPILOT\_QUEUE\_URI that you can use to access the events received on the queue. There are some times in which there might be certain messages that are repeatedly read from the queue and the application is not able to process them, these messages should be rerouted to another queue usually called [Dead-Letter Queue (DLQ)](https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-dead-letter-queues.html) for manual examination. With AWS Copilot It is very easy to specify to create a DLQ and a [*redrive configuration*](https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-configure-dead-letter-queue-redrive.html). All we need to do is add the following section to the manifest file:

subscribe:

topics:

- name: ordersTopic

service: publisher

queue:

dead\_letter:

tries: 3

Once we have modified the manifest, we can deploy the subscriber service into Amazon ECS and AWS Fargate with the command:

copilot svc deploy --name invoicing --env test

### Create the second subscriber (promotion)

In the previous step we created a subscriber service that will process every message sent to the ordersTopic. However it is common that some microservices might not need to process every message they receive but only some messages which have some particular characteristics. To do so, many customers create a new topics or do some preprocessing on the consumer deciding whether the message should be processed or not, however this is not good practice. What we usually recommend is to use a native functionality within SNS which allows you to publish message attributes along the message content so that subscribers can then specify a [subscription filtering policy](https://docs.aws.amazon.com/sns/latest/dg/sns-subscription-filter-policies.html) which defines which kind of messages they want to receive. SNS will then only forward the messages which meet the constraints. With this solution you don’t need to create extra topics or do unnecessary preprocessing.

The service we will deploy in this step is named promotion and is only interested on orders which have an amount greater than $80. In such scenarios a 20% coupon code will be generated for the next purchase of the customer.

Similarly to the service we deployed before we create the service by running:

copilot svc init \

--app pubsub \

--svc-type "Worker Service" \

--name promotion \

--port 5000 \

--dockerfile "subscribers/promotion/Dockerfile"

However on the manifest file we will add a new section specifying the filter policy for the SNS topic subscription.

subscribe:

topics:

- name: ordersTopic

service: publisher

filter\_policy:

amount:

- numeric:

- ">="

- 80

queue:

dead\_letter:

tries: 3

Once we have modified the manifest, we can deploy the subscriber service into Amazon ECS and AWS Fargate with the command:

copilot svc deploy --name promotion --env test

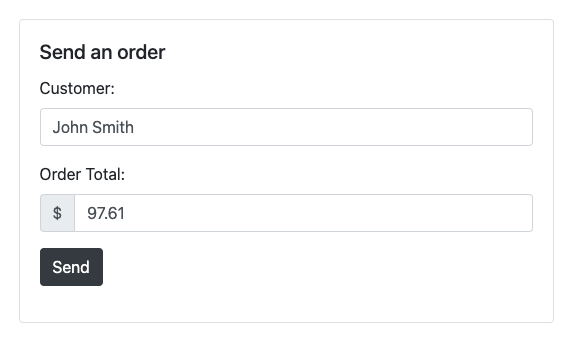
### Verify it works

Now that we have created all the microservices is time to test it works. Open your browser and go to the URL of the load balancer that you copied at the end of Step 3. In case you have lost this URL you can run:

copilot svc show --name publisher

and copy the value of the variable named COPILOT\_LB\_DNS.

Everytime you refresh the page a new name and order total will be generated, but you are able to modify the fields if you want.



To see what’s happening behind the scenes, open a terminal window and run the following command

copilot svc logs \

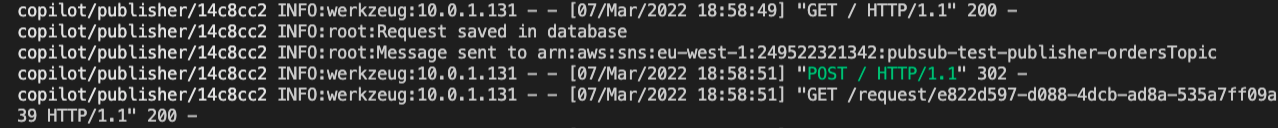
--name publisher \

--env test \

--follow \

--since 1s

This command will allow you to stream the logs out of your subscriber service so arrange your windows so that you can see the frontend and the terminal at the same time. You might not see anything at the beginning, but as soon as you press the send button you should see a similar output:



In order to see what’s happening on the other two services run a similar command for the subscriber services:

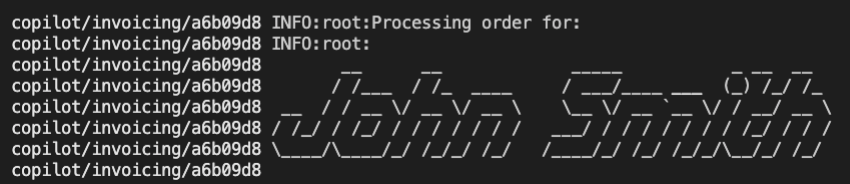
copilot svc logs \

--name invoicing \

--env test \

--follow \

--since 1s



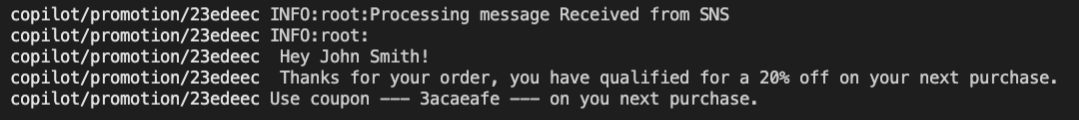
copilot svc logs \

--name promotion \

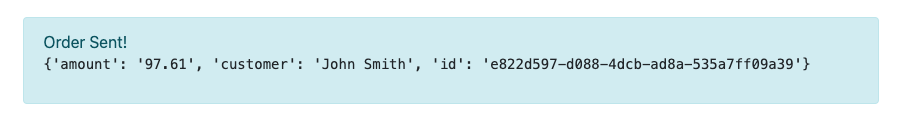
--env test \

--follow \

--since 1s



It is recommended that you have three terminal windows open at the same time so you get to see how the processing occurs and



### Cleaning up

To avoid incurring future charges, delete the resources. If you created everything correctly, you should be able to run the command

copilot app delete pubsub

and all the services and related infrastructure created for this demo will be deleted.

## Conclusion

Restate the post purpose and add next steps. The calls to action can include related content.

### [Optional] Author bio

|  |  |
| --- | --- |
| Photo | *Three sentences introducing the author’s AWS role, experience and interests, and a lighthearted personal note.* |

**Suggested tags:** [match these to the channel SEO strategy]