Tidy Cloud AWS

Last Updated: November 29, 2022

search...

How to become an infrastructure-as-code ninja, using AWS CDK -part 8

ERIK LUNDEVALL-ZARA OCTOBER 3, 2022 ARTICLE

In this article, we are going to wrap up this article series on using AWS CDK for managing infrastructure as code. We will cover two topics in this article:

- Monitoring infrastructure

By the end of this article, you will know more about setting up monitoring and alarms for your infrastructure, and you will know more about finding useful libraries, not only from the core AWS CDK team.

The infrastructure we will monitor is based on what has been defined from previous articles in this series, so please have a look at earlier articles to see what the infrastructure we will manage is.

This article series uses Typescript as an example language. However, there are repositories with example code for multiple languages.

- https://github.com/cloudgnosis/iac-ninja-aws-cdk-ts
- https://github.com/cloudgnosis/iac-ninja-aws-cdk-python
- https://github.com/cloudgnosis/iac-ninja-aws-cdk-go

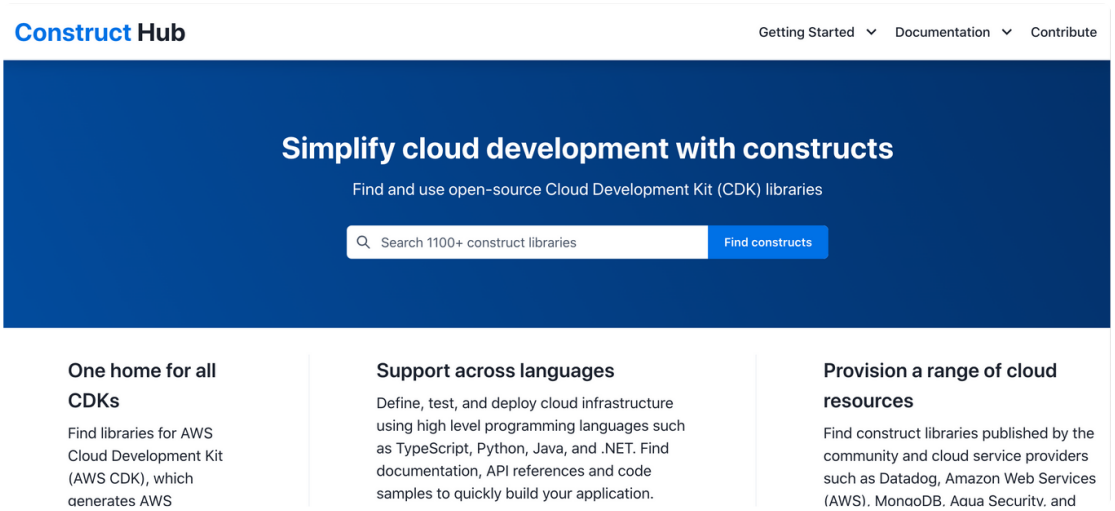
The repositories will contain all the code examples from the articles series, implemented in different languages with the AWS CDK. You can view the code from specific parts and stages in the series by checking out the code tagged with a specific part and step in that part. See the README file in each repository for more details.

Note: *At the time of this writing, the code from this article will only be available for Typescript and Python. This is because of the 3rd party library we will use is not yet published for Go. When the library will become available for Go, the example code above will be updated accordingly.*

Finding and using 3rd party libraries for AWS CDK

Currently, one of the best places to find libraries and resources for use with AWS CDK, or any other CDK (CDKTF, CDK8s) is Construct Hub.

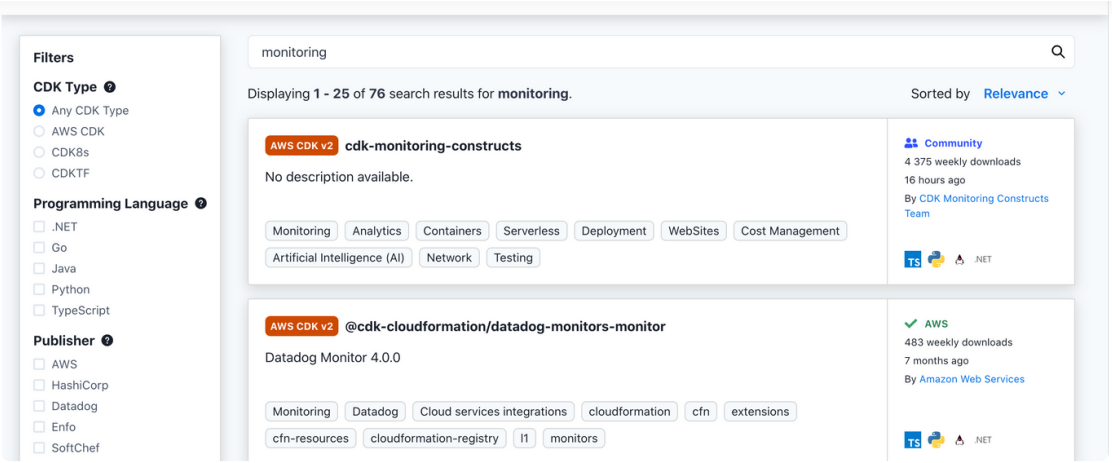
generated constructs for public CloudFormation registry modules.



Since we are going to set up some monitoring for our infrastructure, let us search for **monitoring** and see what we get…

The first one in the search result is a library called **cdk-monitoring-constructs**, which sounds pretty promising. Other results include something for *DataDog*. There are many other hits for the term *monitoring* as well. However, for now, let us look at *cdk-monitoring-constructs*.

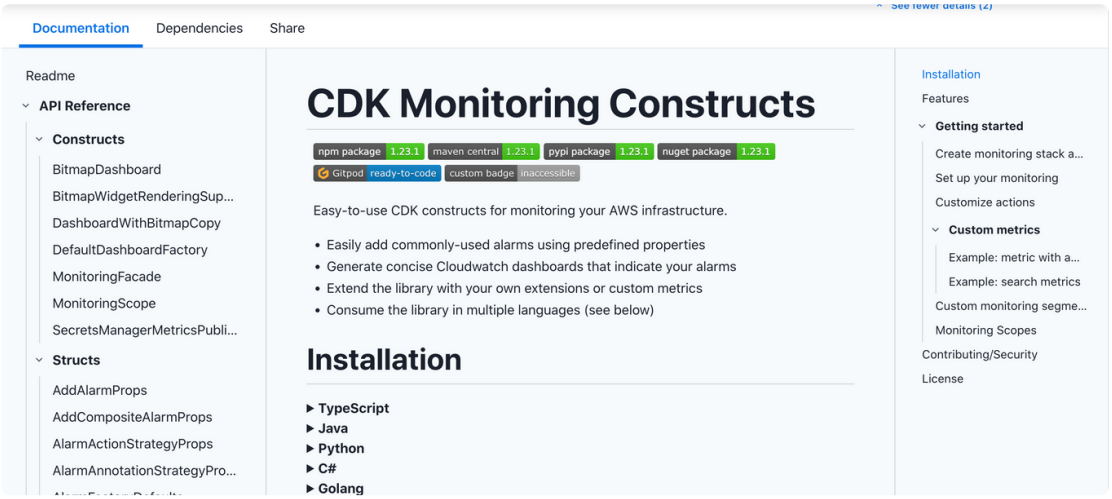
Tidy Cloud AWS



We can see from the symbols that this library is available for Typescript, Python, Java and .NET (C#). No Go though (yet).

We can click on the entry to get into the documentation, read about installation and how to use this construct library. Good stuff!

This will be the starting point for our work to set up some monitoring for our solution.

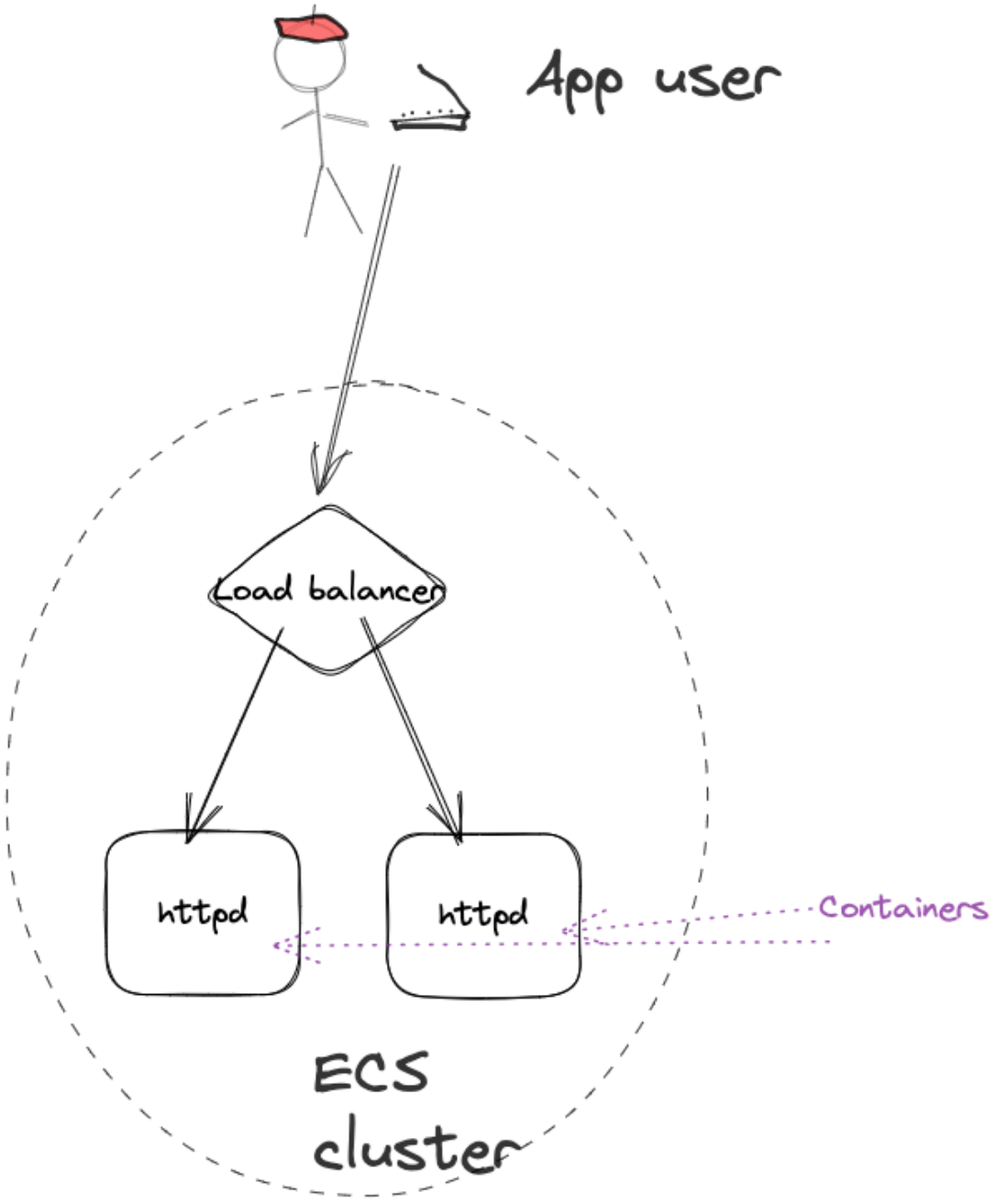


Set up our solution monitoring

Tidy Cloud AWS

If you have read the previous articles in this series, you know that we have defined an infrastructure running a containerized application in an AWS ECS cluster, behind a load balancer and with auto-scaling set up, to increase or decrease the number of container instances based on various performance characteristics (CPU and memory).

The application we have used is the *Apache httpd web server*.



- `bin/my-container-infrastructure.ts` - the main program
- `lib/containers/container-management.ts` - support functions for container infrastructure
- `test/containers/container-management.test.ts` - test code for support functions

The current code can be seen here, and is also available in the GitHub repositories mentioned at the beginning of this article.

bin/my-container-infrastructure.ts

```
import { App, Stack } from 'aws-cdk-lib';
import { IVpc, Vpc } from 'aws-cdk-lib/aws-ec2';
import {
  addCluster,
  addLoadBalancedService,
  addTaskDefinitionWithContainer,
  ContainerConfig,
  setServiceScaling,
  TaskConfig
} from '../lib/containers/container-management';

const app = new App();
const stack = new Stack(app, 'my-container-infrastr
  env: {
    account: process.env.CDK_DEFAULT_ACCOUNT,
    region: process.env.CDK_DEFAULT_REGION,
  },
});

let vpc: IVpc;
```

```
if (vpcName) {
  vpc = Vpc.fromLookup(stack, 'vpc', {
    vpcName,
  });
} else {
  vpc = new Vpc(stack, 'vpc', {
    vpcName: 'my-vpc',
    natGateways: 1,
    maxAzs: 2,
  });
}

const id = 'my-test-cluster';
const cluster = addCluster(stack, id, vpc);

const taskConfig: TaskConfig = { cpu: 512, memoryLi
const containerConfig: ContainerConfig = { dockerHu
const taskdef = addTaskDefinitionWithContainer(stac
const service = addLoadBalancedService(stack, `serv
setServiceScaling(service.service, {
  minCount: 1,
  maxCount: 4,
  scaleCpuTarget: { percent: 50 },
  scaleMemoryTarget: { percent: 70 },
});
```

lib/containers/container-management.ts

```
import { CfnOutput } from 'aws-cdk-lib';
import { IVpc, Peer, Port, SecurityGroup } from 'av
import { Cluster, ContainerImage, FargateService, F
import { ApplicationLoadBalancedFargateService } fr
import { RetentionDays } from 'aws-cdk-lib/aws-logs
import { Construct } from 'constructs';
```


```
            vpc,
        });
    }

    export interface TaskConfig {
        readonly cpu: 256 | 512 | 1024 | 2048 | 4096;
        readonly memoryLimitMB: number;
        readonly family: string;
    }

    export interface ContainerConfig {
        readonly dockerHubImage: string;
        readonly tcpPorts: number[];
    }

    export const addTaskDefinitionWithContainer =
    function(scope: Construct, id: string, taskConfig:
        const taskdef = new FargateTaskDefinition(scope
            cpu: taskConfig.cpu,
            memoryLimitMiB: taskConfig.memoryLimitMB,
            family: taskConfig.family,
        });

        const image = ContainerImage.fromRegistry(conta
        const logdriver = LogDriver.awsLogs({
            streamPrefix: taskConfig.family,
            logRetention: RetentionDays.ONE_DAY,
        });
        const containerDef = taskdef.addContainer(`cont
        for (const port of containerConfig.tcpPorts) {
            containerDef.addPortMappings({ containerPor
        }

        return taskdef;
    };
```

```
            id: string,
            cluster: Cluster,
            taskDef: FargateTaskDefinition,
            port: number,
            desiredCount: number,
            publicEndpoint?: boolean,
            serviceName?: string): ApplicationLoadBala
        // const sg = new SecurityGroup(scope, `${id}-s
        //     description: `Security group for service
        //     vpc: cluster.vpc,
        // });
        // sg.addIngressRule(Peer.anyIpv4(), Port.tcp(p

        const service = new ApplicationLoadBalancedFarg
            cluster,
            taskDefinition: taskDef,
            desiredCount,
            serviceName,
            //securityGroups: [sg],
            circuitBreaker: {
                rollback: true,
            },
            publicLoadBalancer: publicEndpoint,
            listenerPort: port,
        });

        return service;
    };


    export interface ScalingThreshold {
        percent: number;
    }
    export interface ServiceScalingConfig {
        minCount: number;
        maxCount: number;
```

```
}


export const setServiceScaling = function(service:
    const scaling = service.autoScaleTaskCount({
        maxCapacity: config.maxCount,
        minCapacity: config.minCount,
    });

    scaling.scaleOnCpuUtilization('CpuScaling', {
        targetUtilizationPercent: config.scaleCpuTa
    });

    scaling.scaleOnMemoryUtilization('MemoryScaling
        targetUtilizationPercent: config.scaleMemor
    });

}
```

test/containers/container-management.test.ts

```
import { Stack } from 'aws-cdk-lib';
import { Vpc } from 'aws-cdk-lib/aws-ec2';
import { Capture, Match, Template } from 'aws-cdk-l
import {
    addCluster,
    addLoadBalancedService,
    addTaskDefinitionWithContainer,
    ContainerConfig,
    setServiceScaling,
    TaskConfig
} from '../../lib/containers/container-management';
import { Cluster, TaskDefinition } from 'aws-cdk-li
```


```
    const stack = new Stack();
    const vpc = new Vpc(stack, 'vpc');

    // Test code
    const cluster = addCluster(stack, 'test-cluster

    // Check result
    const template = Template.fromStack(stack);

    template.resourceCountIs('AWS::ECS::Cluster', 1

    expect(cluster.vpc).toEqual(vpc);
});


test('ECS Fargate task definition defined', () => {
    // Test setup
    const stack = new Stack();
    const cpuval = 512;
    const memval = 1024;
    const familyval = 'test';
    const taskCfg: TaskConfig = { cpu: cpuval, memo
    const imageName = 'httpd';
    const containerCfg: ContainerConfig = { dockerH

    // Test code
    const taskdef = addTaskDefinitionWithContainer(

    // Check result
    const template = Template.fromStack(stack);

    expect(taskdef.isFargateCompatible).toBeTruthy(
    expect(stack.node.children.includes(taskdef)).t

    template.resourceCountIs('AWS::ECS::TaskDefinit
    template.hasResourceProperties('AWS::ECS::TaskD
```

```
            Memory: memval.toString(),
            Family: familyval,
        });

    });

    test('Container definition added to task definitior
        // Test setup
        const stack = new Stack();
        const cpuval = 512;
        const memval = 1024;
        const familyval = 'test';
        const taskCfg: TaskConfig = { cpu: cpuval, memc
        const imageName = 'httpd';
        const containerCfg: ContainerConfig = { dockerH

        // Test code
        const taskdef = addTaskDefinitionWithContainer(

        // Check result
        const template = Template.fromStack(stack);
        const containerDef = taskdef.defaultContainer;

        expect(taskdef.defaultContainer).toBeDefined();
        expect(containerDef?.imageName).toEqual(imageNa
        template.hasResourceProperties('AWS::ECS::TaskD
            ContainerDefinitions: Match.arrayWith([
                Match.objectLike({
                    Image: imageName,
                }),
            ]),
        });
    });

    describe('Test service creation options', () => {
        let stack: Stack;
```

```
beforeEach(() => {
    // Test setup
    stack = new Stack();
    const vpc = new Vpc(stack, 'vpc');
    cluster = addCluster(stack, 'test-cluster',

    const cpuval = 512;
    const memval = 1024;
    const familyval = 'test';
    const taskCfg: TaskConfig = { cpu: cpuval,
    const imageName = 'httpd';
    const containerCfg: ContainerConfig = { doc
    taskdef = addTaskDefinitionWithContainer(st
});

test('Fargate load-balanced service created, wi
    const port = 80;
    const desiredCount = 1;

    // Test code
    const service = addLoadBalancedService(stac

    // Check result
    const sgCapture = new Capture();
    const template = Template.fromStack(stack);

    expect(service.cluster).toEqual(cluster);
    expect(service.taskDefinition).toEqual(task

    template.resourceCountIs('AWS::ECS::Service
    template.hasResourceProperties('AWS::ECS::S
        DesiredCount: desiredCount,
        LaunchType: 'FARGATE',
        NetworkConfiguration: Match.objectLike(
            AwsvpcConfiguration: Match.objectLi
```

```
                }),
            }),
        });

        template.resourceCountIs('AWS::ElasticLoadB
        template.hasResourceProperties('AWS::Elasti
            Type: 'application',
            Scheme: 'internet-facing',
        });

        //template.resourceCountIs('AWS::EC2::Secur
        template.hasResourceProperties('AWS::EC2::S
            SecurityGroupIngress: Match.arrayWith([
                Match.objectLike({
                    CidrIp: '0.0.0.0/0',
                    FromPort: port,
                    IpProtocol: 'tcp',
                }),
            ]),
        });
    });

    test('Fargate load-balanced service created, wi
        const port = 80;
        const desiredCount = 1;

        // Test code
        const service = addLoadBalancedService(stac

        // Check result
        const template = Template.fromStack(stack);

        template.resourceCountIs('AWS::ElasticLoadB
        template.hasResourceProperties('AWS::Elasti
            Type: 'application',
            Scheme: 'internal',
```

```
test('Scaling settings of load balancer', () =>
    const port = 80;
    const desiredCount = 2;
    const service = addLoadBalancedService(stac

    // Test code
    const config = {
        minCount: 1,
        maxCount: 5,
        scaleCpuTarget: { percent: 50 },
        scaleMemoryTarget: { percent: 50 },
    };

    setServiceScaling(service.service, config);


    // Check result
    const scaleResource = new Capture();
    const template = Template.fromStack(stack);
    template.resourceCountIs('AWS::ApplicationA
    template.hasResourceProperties('AWS::Applic
        MaxCapacity: config.maxCount,
        MinCapacity: config.minCount,
        ResourceId: scaleResource,
        ScalableDimension: 'ecs:service:Desirec
        ServiceNamespace: 'ecs',
    });

    template.resourceCountIs('AWS::ApplicationA
    template.hasResourceProperties('AWS::Applic
        PolicyType: 'TargetTrackingScaling',
        TargetTrackingScalingPolicyConfiguratic
            PredefinedMetricSpecification: Matc
                PredefinedMetricType: 'ECSServi
            }),
```

```
            });

            template.hasResourceProperties('AWS::Applic
                PolicyType: 'TargetTrackingScaling',
                TargetTrackingScalingPolicyConfiguratic
                    PredefinedMetricSpecification: Matc
                        PredefinedMetricType: 'ECSServi
                    }),
                    TargetValue: config.scaleMemoryTarg
                }),
            });
        });

    });
```

Where to start with the monitoring?

So where do we start? We have found a monitoring library for AWS CDK that may make our lives easier perhaps, but we do not know that much about it yet.

If we have some monitoring, we want to see how our solution is doing, based on some kind of metrics, and we may want alerts when things may go bad. It would be good with some visualisation of this.

You may already have some corporate solution you want to hook up your monitoring to, but for this article series, we are just going to stick within the AWS services. One option available to us in this case is **to set up monitoring dashboards in CloudWatch**.

Thus, our infrastructure code should set up a dashboard at least, and then we have to sort out what we may want to put on that dashboard and how.

Let us explore that! ***We will start by writing a test****.*

Tidy Cloud AWS

Our initial idea here is that if we should add monitoring to our solution, we should also have at least one dashboard that can visualize information for us. We may change our minds later about this, but it is a starting point.

For this purpose, we will create a file to include tests for our monitoring - `test/monitoring.test.ts` . We add a test for a function called **initMonitoring**, which we can apply to our stack, and provide a configuration input. This should return data or handle that we can use for handling the monitoring we want to add. At the very least, we should have an empty CloudWatch dashboard set up. It seems appropriate to include the dashboard name then in the config.

How do we know if we will have a dashboard? The documentation for **cdk-monitoring-constructs** is not entirely clear from initial view, but it seems it may add a dashboard implicitly. If we look at the CloudFormation documentation, we also see that there is an **AWS::CloudWatch::Dashboard** resource.

So our test can make a call to an *initMonitoring* function and this should cause that a CloudWatch dashboard being added to the generated CloudFormation. Let us also now initially log what the generated CloudFormation will look like to see what we actually get.

```
import { Stack } from 'aws-cdk-lib';
import { Template } from 'aws-cdk-lib/assertions';
import { initMonitoring, MonitoringConfig } from '.

test('Init monitoring of stack, with only defaults'
    const stack = new Stack();

    const monitoringConfig: MonitoringConfig = {
        dashboardName: 'test-monitoring',
    }
    const monitoring = initMonitoring(stack, monitc
```

Tidy Cloud AWS

```
        console.log(JSON.stringify(template.toJSON(), r
        template.resourceCountIs('AWS::CloudWatch::Dash
        template.hasResourceProperties('AWS::CloudWatch
            DashboardName: monitoringConfig.dashboardNa
        });
});
```

For implementing *initMonitoring*, we can try to use the **MonitoringFacade** from cdk-monitoring-constructs. The function can simply return a structure which includes the MonitoringFacade object, which seems suitable for our purposes.

```
import { Construct } from 'constructs';
import { MonitoringFacade } from 'cdk-monitoring-co

export interface MonitoringConfig {
    readonly dashboardName: string;
}

export interface MonitoringContext {
    readonly handler: MonitoringFacade;
}

export const initMonitoring = function(scope: Const

    return {
        handler: new MonitoringFacade(scope, confic
    }
}
```

Running the test now when the code compiles, we can see both from our test and the log output, that our guess was correct, *there will be a dashboard created when we create the MonitoringFacade*!

We can add a call in our main program also to initialize the monitoring.

```
const monitoring = initMonitoring(stack, {
  dashboardName: 'monitoring',
});

monitoring.handler.addMediumHeader('Test App monitc
```

You can deploy the updated AWS CDK code if you want and verify that there will be an actual dashboard created. It would not contain anything, though.

Add monitoring of actual resources

Our next step is to add some actual monitoring of resources. We can look at what **cdk-monitoring-constructs** provides for us. From the documentation, we can see that are many functions available from the *MonitoringFacade*, that start with **monitor** in the name and refer to different resources.

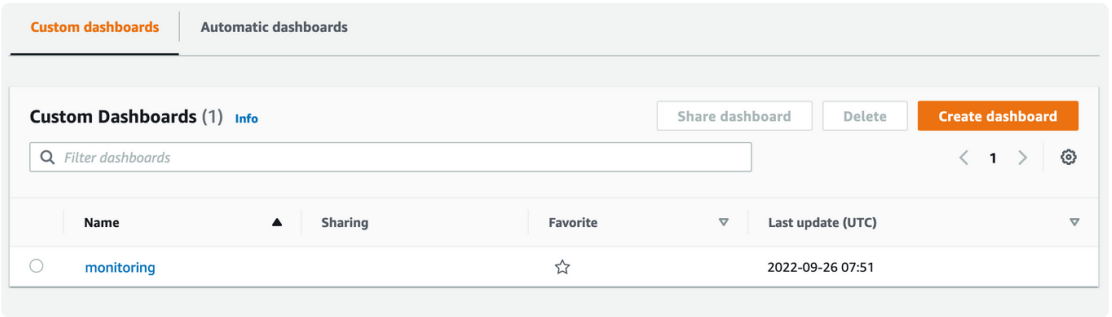
Since our solution sets up a Fargate Service in an ECS cluster with an application load balancer in front of the containers we run, the functions **monitorFargateApplicationLoadBalancer** and **monitorFargateService** seems relevant in this case. There is also a more generic **monitorScope**, which could also apply. Reading the somewhat sparse docs a bit more, *monitorFargateService* may be more appropriate if you use the AWS CDK **ApplicationLoadBalancedFargateService**, which is what we use in our solution.

So let us add monitoring using this function! There is only one required property for configuring monitoring, which is the *ApplicationLoadBalancedFargateService* object we have created. An optional field we may want to add also is the **humanReadableName** property as well.

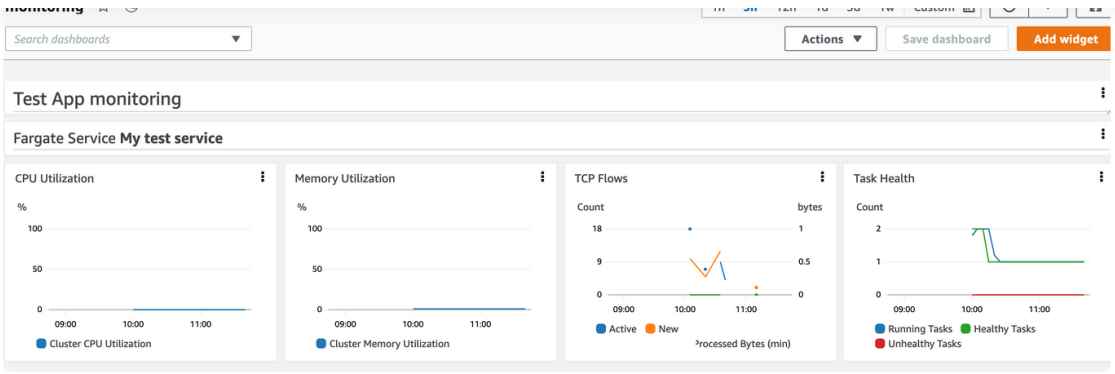
Tidy Cloud AWS

```
const service = addLoadBalancedService(stack, `serv
setServiceScaling(service.service, {
  minCount: 1,
  maxCount: 4,
  scaleCpuTarget: { percent: 50 },
  scaleMemoryTarget: { percent: 70 },
});

const monitoring = initMonitoring(stack, {
  dashboardName: 'monitoring',
});

monitoring.handler.addMediumHeader('Test App monitc
monitoring.handler.monitorFargateService({
  fargateService: service,
  humanReadableName: 'My test service',
});
```

After deployment, we can check out in CloudWatch to see what has been added.



Tidy Cloud AWS



We have a few widgets here for CPU and memory, TCP traffic and task health. That is a good start! The task health view actually shows a property of our configuration. We have set the desired task count for our service to 2, but we have also set up our auto-scaling to have a minimum task count of 1 and a maximum of 4. So the service initially started with 2 tasks running and then scaled down to a single task.

Note: *I also tried to use the* ***monitorFargateApplicationLoadBalancer*** *function, and the result was the same dashboard. The difference was the parameters that you were required to provide in the call.*

Setting up alarms - what do we need?

Now that we have some visuals in dashboard for our monitoring, let us try to set up some kind of alarm as well. For easy testing, let us set some alarm on the number of tasks running in our solution.

Reading the docs for *cdk-monitoring-constructs*, there is an option for us to add an alarm for **running task count**, e.g. if the number of running tasks go below a certain threshold for some time.

Reading through the docs, one can see that for this type of alarm to work, we need to enable **container insights** on the ECS Cluster. By default, this is disabled. The aws-ecs Cluster construct in AWS CDK allows us to set this properly, but does not allow us to check the state

Also, if we set an alarm, we need to send a notification about the alarm somewhere. One common approach is to send notifications on an *SNS topic*. Thus, we need to make sure we have an SNS topic that alarms will go to.

So we have 3 things to develop here that we can immediately think of:

- The running task alarm itself
- The SNS topic to send alarms to
- Enabling container insights on the ECS Cluster

Container insights should be in place before the alarm itself, so let us start there. The SNS topic does not have to be in place before the alarm, since notification is optional and SNS is not the only way to send notifications.

Enabling container insights

Let us start by adding a new test for creating an ECS cluster. Right now we have the **addCluster** function, which we pass in a construct scope, and id and a VPC. If we are going to pass in more properties to this function, we can add more function parameters. We can also use the same pattern as CDK constructs and pass in a set of properties as a single parameter.

I like the latter better, because it becomes more clear what each input is with named properties - at least in Typescript.

So let us refactor the current **addCluster** function to pass in a set of properties as a single parameter, and then add a new property to enable *container insights*.

```
        readonly vpc: IVpc;
        readonly enableContainerInsights?: true;
    }

    export const addCluster = function(scope: Construct
        return new Cluster(scope, id, {
            vpc: config.vpc,
            containerInsights: config.enableContainerIr
        });
    }
```

In our new test to check the container insights checking, we test the generated CloudFormation for the setting.

```
    test('Check that container insights will be enablec
        // Test setup
        const stack = new Stack();
        const vpc = new Vpc(stack, 'vpc');
        const config: ClusterConfig = {
            vpc,
            enableContainerInsights: true
        };

        // Test code
        const cluster = addCluster(stack, 'test-cluster

        // Check result
        const template = Template.fromStack(stack);
        template.hasResourceProperties('AWS::ECS::Clust
            ClusterSettings: Match.arrayWith([
                Match.objectEquals({
                    Name: 'containerInsights',
                    Value: 'enabled',
                }),
            ]),
        });
```

We can re-deploy the cluster with the new setting if we want.

Adding an alarm

Next step is to add an alarm. Here, the logic lives with the **cdk-monitoring-constructs** library itself. *So there is not much point in making sure the alarm is there, if we are just using the functions in the library itself.*

The alarm to set up for testing this feature will be *to trigger an alarm if the number of running tasks is less than 2 for 10 minutes or more*.

The alarm configuration will use 5-minute periods and trigger an alarm if 2 evaluation periods have passed and 2 data points fulfill the condition for the alarm. Why not just a single 10-minute period? The reason here is that sometimes data may be delayed or simply missing in CloudWatch. *To avoid false positives, we set the evaluation to include multiple periods and data points.*

We can provide alarm information in our call to **monitorFargateService**, which we set up for an alarm if the number of tasks go below 2. We know this will happen since our minimal task count in the auto-scaling is set to 1.

We add the alarm to our code and redeploy to see what the effect is on our monitoring deployment. The dashboard has an update, and there is a new alarm in place. We can see that the alarm has no action associated with it.

```
monitoring.handler.addMediumHeader('Test App monitc
monitoring.handler.monitorFargateService({
  fargateService: service,
  humanReadableName: 'My test service',
```

```
        maxRunningTasks: 2,
        comparisonOperatorOverride: ComparisonOperatc
        evaluationPeriods: 2,
        datapointsToAlarm: 2,
        period: Duration.minutes(5),
      }
    }
  });
```

Thus, the next step for us is to associate the alarm with an SNS topic.

Add an alarm notification topic

Our next consideration is how this SNS topic should be added, and how to test that.

We can add an alarm action for each alarm we define, which includes an SNS topic. However, it may be cumbersome to add this for every single alarm we define - especially if we decide to use the same topic for all or most alarms.

Fortunately, **cdk-monitoring-constructs** allows us to define *a default action for alarms*, when we create the *MonitoringFacade*. In that way, we can define the topic once only, and in one place.

Let us add an optional property to the configuration passed to **initMonitoring**, which is an SNS topic construct, and set that up as the default action. We can refactor this function to include a topic that will be set as a default action.

But how do we test this? Unfortunately, there is not any easy test, since we cannot directly extract that information from the created MonitoringFacade object. We essentially would need to create an alarm on some resource that we also created and then examine the created alarm if it has an action which includes the default SNS topic

Technically, we can certainly build such a test to check that this is generated properly. But then we also would mainly test the **cdk-monitoring-constructs** library, and not that much of our own code. That is wasteful. And possibly also brittle, since we cannot be 100% sure that our test code would work if the underlying implementation changed.

So we will relax on the test coverage here a bit for now.

```
test('Init monitoring of stack, with SNS topic for
    const stack = new Stack();
    const vpc = new Vpc(stack, 'vpc');
    const cluster = addCluster(stack, 'test-cluster
    const alarmTopic = new Topic(stack, 'alarm-topi

    const dashboardName = 'test-monitoring';
    const monitoringConfig: MonitoringConfig = {
        dashboardName,
        defaultAlarmTopic: alarmTopic,
    };

    const monitoring = initMonitoring(stack, monitc

    expect(monitoring.defaultAlarmTopic).toEqual(al
});
```

Add an alarm notification

We can deploy the infrastructure updates and see that we have alarm information in place as well now, which has an action to send to our SNS topic.

```
const alarmTopic = new Topic(stack, 'alarm-topic',
  displayName: 'Alarm topic',
```

```
  dashboardName: 'monitoring',
  defaultAlarmTopic: alarmTopic,
});
```

If you want to check that the notification is sent via SNS, you can add an email subscriber to the topic and check that way.

```
const alarmEmail = 'hello@example.com';
alarmTopic.addSubscription(new EmailSubscription(al
```

Alarm severity and category

When we send the alarm to SNS, we have nothing right now to show the severity of the alarm, nor any categorization of the alarm beside the name of the alarm.

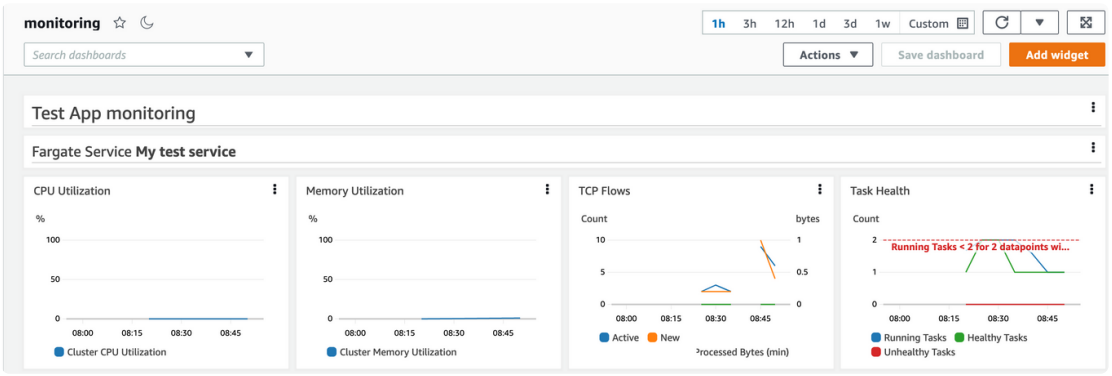
This is often handled by external solutions. It is also possible to use AWS services for this, like AWS Systems Manager OpsCenter. We can add some code to include sending alarm info to OpsCenter also, besides the SNS topic, with an override on the default alarm strategy on our alarm.

Deploying this code will allow the alarm to be visible at the OpsCenter dashboard as well!

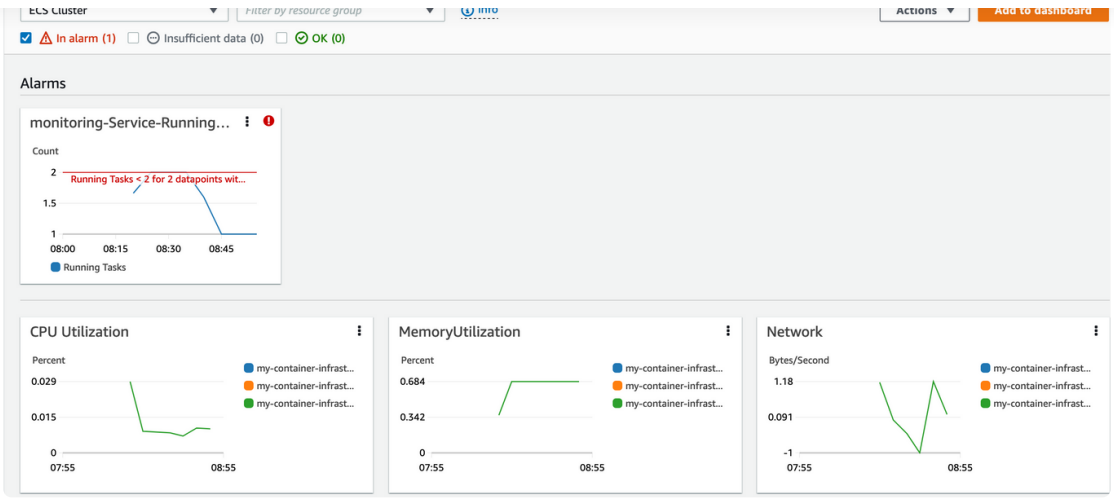
```
const alarmActions: IAlarmActionStrategy[] = [
  new OpsItemAlarmActionStrategy(OpsItemSeverity.ME
];
if (monitoring.defaultAlarmTopic) {
  alarmActions.push(new SnsAlarmActionStrategy({
    onAlarmTopic: monitoring.defaultAlarmTopic,
    onOkTopic: monitoring.defaultAlarmTopic,
  }));
}
```

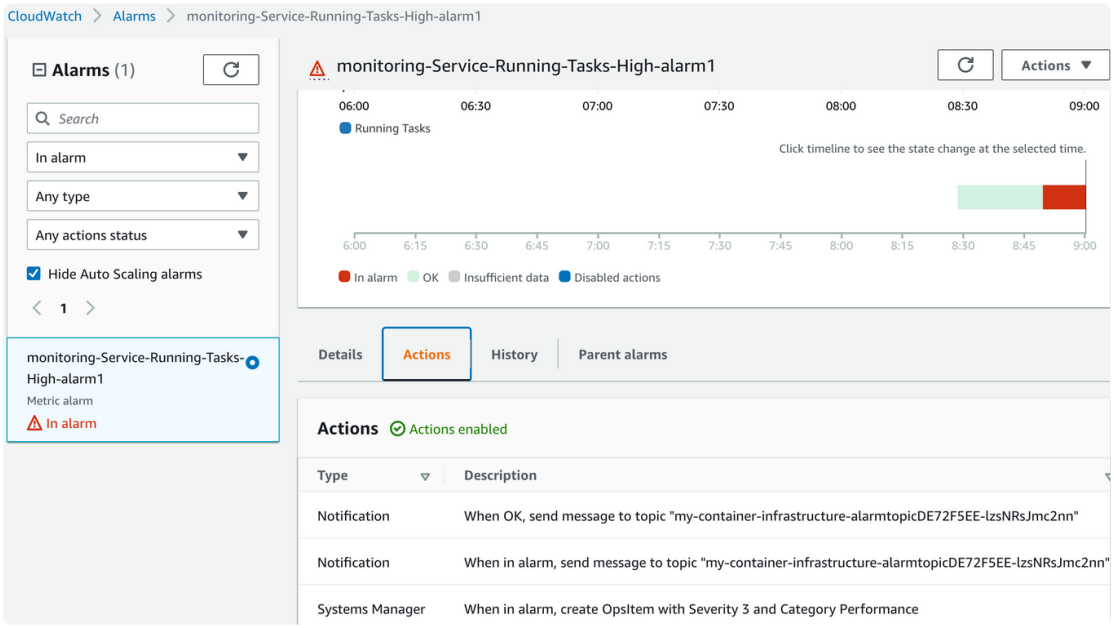
Tidy Cloud AWS

```
monitoring.handler.monitorFargateService({
  fargateService: service,
  humanReadableName: 'My test service',

  addRunningTaskCountAlarm: {
    alarm1: {
      maxRunningTasks: 2,
      comparisonOperatorOverride: ComparisonOperatc
      evaluationPeriods: 2,
      datapointsToAlarm: 2,
      period: Duration.minutes(5),
      actionOverride: new MultipleAlarmActionStrate
    }
  }
});
```

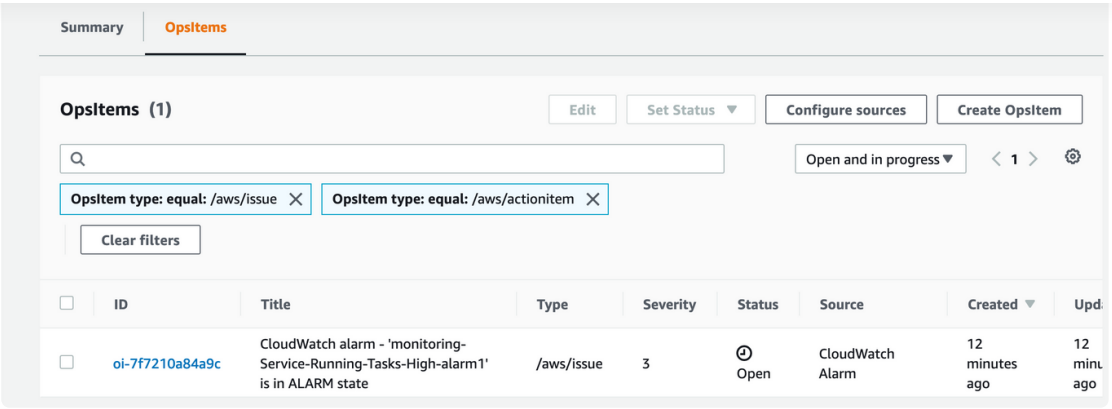


Tidy Cloud AWS





Tidy Cloud AWS



Summary and final words

In this article, we took an add-on library for AWS CDK to facilitate monitoring of our solution infrastructure. With that, we set up a dashboard with a few widgets for monitoring visualisation.

We also added an alarm with notification via SNS topic and to AWS Systems Manager OpsCenter.

We have kept the solution small and simple in this article series, and kept all infrastructure in the same stack. In a real-world setting, we may have multiple stacks, each dedicated to a specific group of resources.

We would also likely add some automation for the provisioning of the infrastructure. This is, however, beyond this article series.

This is the final part of this article series. However, it is not the end of this material. It will be refactored into a new form and with more material to come.

I hope you have enjoyed this article series, and it has provided some value to you. If it has, I would be happy to know more! If it has not, I would be happy to know about that as well! We need feedback to improve.

Tidy Cloud AWS

/Erik

• • •

Appendix: Code re-cap

Our final code will now look like this:

bin/my-container-infrastructure.ts

```TypeScript
import { App, Duration, Stack } from 'aws-cdk-lib';
import { ComparisonOperator } from 'aws-cdk-lib/aws
import { OpsItemCategory, OpsItemSeverity } from 'a
import { IVpc, Vpc } from 'aws-cdk-lib/aws-ec2';
import { Topic } from 'aws-cdk-lib/aws-sns';
import { EmailSubscription } from 'aws-cdk-lib/aws-
import { IAlarmActionStrategy, MultipleAlarmActionS
import {
  addCluster,
  addLoadBalancedService,
  addTaskDefinitionWithContainer,
  ClusterConfig,
  ContainerConfig,
  setServiceScaling,
  TaskConfig
} from '../lib/containers/container-management';
import { initMonitoring, MonitoringConfig } from '.

const app = new App();
const stack = new Stack(app, 'my-container-infrastr
  env: {
    account: process.env.CDK_DEFAULT_ACCOUNT,
    region: process.env.CDK_DEFAULT_REGION,
  },
});
```

Tidy Cloud AWS

```
let vpcName = app.node.tryGetContext('vpcname');
if (vpcName) {
  vpc = Vpc.fromLookup(stack, 'vpc', {
    vpcName,
  });
} else {
  vpc = new Vpc(stack, 'vpc', {
    vpcName: 'my-vpc',
    natGateways: 1,
    maxAzs: 2,
  });
}


const id = 'my-test-cluster';
const clusterConfig: ClusterConfig = { vpc, enableC
const cluster = addCluster(stack, id, clusterConfig

const taskConfig: TaskConfig = { cpu: 512, memoryLi
const containerConfig: ContainerConfig = { dockerHu
const taskdef = addTaskDefinitionWithContainer(stac
const service = addLoadBalancedService(stack, `serv
setServiceScaling(service.service, {
  minCount: 1,
  maxCount: 4,
  scaleCpuTarget: { percent: 50 },
  scaleMemoryTarget: { percent: 70 },
});

const alarmTopic = new Topic(stack, 'alarm-topic',
  displayName: 'Alarm topic',
});
const monitoring = initMonitoring(stack, {
  dashboardName: 'monitoring',
  defaultAlarmTopic: alarmTopic,
```

Tidy Cloud AWS

```
const alarmActions: IAlarmActionStrategy[] = [
  new OpsItemAlarmActionStrategy(OpsItemSeverity.ME
];
if (monitoring.defaultAlarmTopic) {
  alarmActions.push(new SnsAlarmActionStrategy({
    onAlarmTopic: monitoring.defaultAlarmTopic,
    onOkTopic: monitoring.defaultAlarmTopic,
  }));
}

monitoring.handler.addMediumHeader('Test App monitc
monitoring.handler.monitorFargateService({
  fargateService: service,
  humanReadableName: 'My test service',

  addRunningTaskCountAlarm: {
    alarm1: {
      maxRunningTasks: 2,
      comparisonOperatorOverride: ComparisonOperatc
      evaluationPeriods: 2,
      datapointsToAlarm: 2,
      period: Duration.minutes(5),
      actionOverride: new MultipleAlarmActionStrate
    }
  }
});

const alarmEmail = 'hello@example.com';
alarmTopic.addSubscription(new EmailSubscription(al
```

lib/containers/container-management.ts

```
import { CfnOutput } from 'aws-cdk-lib';
import { IVpc, Peer, Port, SecurityGroup } from 'av
import { Cluster, ContainerImage, FargateService, F
```

Tidy Cloud AWS

```
import { Construct } from 'constructs';

export interface ClusterConfig {
    readonly vpc: IVpc;
    readonly enableContainerInsights?: boolean;
}

export const addCluster = function(scope: Construct
    return new Cluster(scope, id, {
        vpc: config.vpc,
        containerInsights: config.enableContainerIn
    });
}

export interface TaskConfig {
    readonly cpu: 256 | 512 | 1024 | 2048 | 4096;
    readonly memoryLimitMB: number;
    readonly family: string;
}

export interface ContainerConfig {
    readonly dockerHubImage: string;
    readonly tcpPorts: number[];
}

export const addTaskDefinitionWithContainer =
function(scope: Construct, id: string, taskConfig:
    const taskdef = new FargateTaskDefinition(scope
        cpu: taskConfig.cpu,
        memoryLimitMiB: taskConfig.memoryLimitMB,
        family: taskConfig.family,
    });

    const image = ContainerImage.fromRegistry(conta
    const logdriver = LogDriver.awsLogs({
        streamPrefix: taskConfig.family,
```

Tidy Cloud AWS

```
        const containerDef = taskdef.addContainer(`cont
        for (const port of containerConfig.tcpPorts) {
            containerDef.addPortMappings({ containerPor
        }

        return taskdef;
    };

    export const addLoadBalancedService =
    function(scope: Construct,
             id: string,
             cluster: Cluster,
             taskDef: FargateTaskDefinition,
             port: number,
             desiredCount: number,
             publicEndpoint?: boolean,
             serviceName?: string): ApplicationLoadBala
        // const sg = new SecurityGroup(scope, `${id}-s
        //     description: `Security group for service
        //     vpc: cluster.vpc,
        // });
        // sg.addIngressRule(Peer.anyIpv4(), Port.tcp(p

        const service = new ApplicationLoadBalancedFarg
            cluster,
            taskDefinition: taskDef,
            desiredCount,
            serviceName,
            //securityGroups: [sg],
            circuitBreaker: {
                rollback: true,
            },
            publicLoadBalancer: publicEndpoint,
            listenerPort: port,
        });
```

```
export interface ScalingThreshold {
    percent: number;
}
export interface ServiceScalingConfig {
    minCount: number;
    maxCount: number;
    scaleCpuTarget: ScalingThreshold;
    scaleMemoryTarget: ScalingThreshold;

}


export const setServiceScaling = function(service:
    const scaling = service.autoScaleTaskCount({
        maxCapacity: config.maxCount,
        minCapacity: config.minCount,
    });

    scaling.scaleOnCpuUtilization('CpuScaling', {
        targetUtilizationPercent: config.scaleCpuTa
    });

    scaling.scaleOnMemoryUtilization('MemoryScaling
        targetUtilizationPercent: config.scaleMemor
    });

}
```

lib/monitoring/index.ts

```
import { Construct } from 'constructs';
import {
    IAlarmActionStrategy,
```

```
    SnsAlarmActionStrategy
} from 'cdk-monitoring-constructs';
import { ITopic } from 'aws-cdk-lib/aws-sns';

export interface MonitoringConfig {
    readonly dashboardName: string;
    readonly defaultAlarmNamePrefix?: string;
    readonly defaultAlarmTopic?: ITopic;
}

export interface MonitoringContext {
    readonly handler: MonitoringFacade;
    readonly defaultAlarmTopic?: ITopic;
    readonly defaultAlarmNamePrefix?: string;
}

export const initMonitoring = function(scope: Const

    let snsAlarmStrategy: IAlarmActionStrategy = ne
    if (config.defaultAlarmTopic) {
        snsAlarmStrategy = new SnsAlarmActionStrate
    }
    const defaultAlarmNamePrefix = config.defaultAl
    return {
        handler: new MonitoringFacade(scope, confic
            alarmFactoryDefaults: {
                actionsEnabled: true,
                action: snsAlarmStrategy,
                alarmNamePrefix: defaultAlarmNamePr
            },
        }),
        defaultAlarmTopic: config.defaultAlarmTopic
        defaultAlarmNamePrefix,
    }
}
```

Tidy Cloud AWS

```
import { Stack } from 'aws-cdk-lib';
import { Vpc } from 'aws-cdk-lib/aws-ec2';
import { Capture, Match, Template } from 'aws-cdk-l
import {
    addCluster,
    addLoadBalancedService,
    addTaskDefinitionWithContainer,
    ClusterConfig,
    ContainerConfig,
    setServiceScaling,
    TaskConfig
} from '../../lib/containers/container-management';
import { Cluster, TaskDefinition } from 'aws-cdk-li

test('ECS cluster is defined with existing vpc', ()
    // Test setup
    const stack = new Stack();
    const vpc = new Vpc(stack, 'vpc');

    // Test code
    const cluster = addCluster(stack, 'test-cluster

    // Check result
    const template = Template.fromStack(stack);

    template.resourceCountIs('AWS::ECS::Cluster', 1

    expect(cluster.vpc).toEqual(vpc);
});

test('Check that container insights will be enablec
    // Test setup
    const stack = new Stack();
    const vpc = new Vpc(stack, 'vpc');
    const config: ClusterConfig = {
        vpc,
```

```
        // Test code
        const cluster = addCluster(stack, 'test-cluster

        // Check result
        const template = Template.fromStack(stack);
        template.hasResourceProperties('AWS::ECS::Clust
            ClusterSettings: Match.arrayWith([
                Match.objectEquals({
                    Name: 'containerInsights',
                    Value: 'enabled',
                }),
            ]),
        });
    });


    test('ECS Fargate task definition defined', () => {
        // Test setup
        const stack = new Stack();
        const cpuval = 512;
        const memval = 1024;
        const familyval = 'test';
        const taskCfg: TaskConfig = { cpu: cpuval, memc
        const imageName = 'httpd';
        const containerCfg: ContainerConfig = { dockerH

        // Test code
        const taskdef = addTaskDefinitionWithContainer(

        // Check result
        const template = Template.fromStack(stack);

        expect(taskdef.isFargateCompatible).toBeTruthy(
        expect(stack.node.children.includes(taskdef)).t
```

Tidy Cloud AWS

```
            RequiresCompatibilities: [ 'FARGATE' ],
            Cpu: cpuval.toString(),
            Memory: memval.toString(),
            Family: familyval,
        });

    });

    test('Container definition added to task definitior
        // Test setup
        const stack = new Stack();
        const cpuval = 512;
        const memval = 1024;
        const familyval = 'test';
        const taskCfg: TaskConfig = { cpu: cpuval, memc
        const imageName = 'httpd';
        const containerCfg: ContainerConfig = { dockerH

        // Test code
        const taskdef = addTaskDefinitionWithContainer(

        // Check result
        const template = Template.fromStack(stack);
        const containerDef = taskdef.defaultContainer;

        expect(taskdef.defaultContainer).toBeDefined();
        expect(containerDef?.imageName).toEqual(imageNa
        template.hasResourceProperties('AWS::ECS::TaskD
            ContainerDefinitions: Match.arrayWith([
                Match.objectLike({
                    Image: imageName,
                }),
            ]),
        });
    });
```

```
let cluster: Cluster;
let taskdef: TaskDefinition;

beforeEach(() => {
    // Test setup
    stack = new Stack();
    const vpc = new Vpc(stack, 'vpc');
    cluster = addCluster(stack, 'test-cluster',

    const cpuval = 512;
    const memval = 1024;
    const familyval = 'test';
    const taskCfg: TaskConfig = { cpu: cpuval,
    const imageName = 'httpd';
    const containerCfg: ContainerConfig = { doc
    taskdef = addTaskDefinitionWithContainer(st
});

test('Fargate load-balanced service created, wi
    const port = 80;
    const desiredCount = 1;

    // Test code
    const service = addLoadBalancedService(stac

    // Check result
    const sgCapture = new Capture();
    const template = Template.fromStack(stack);

    expect(service.cluster).toEqual(cluster);
    expect(service.taskDefinition).toEqual(task

    template.resourceCountIs('AWS::ECS::Service
    template.hasResourceProperties('AWS::ECS::S
        DesiredCount: desiredCount,
        LaunchType: 'FARGATE',
```

```
                    AssignPublicIp: 'DISABLED',
                    SecurityGroups: Match.arrayWith(
                }),
            }),
        });

        template.resourceCountIs('AWS::ElasticLoadB
        template.hasResourceProperties('AWS::Elasti
            Type: 'application',
            Scheme: 'internet-facing',
        });

        //template.resourceCountIs('AWS::EC2::Secur
        template.hasResourceProperties('AWS::EC2::S
            SecurityGroupIngress: Match.arrayWith([
                Match.objectLike({
                    CidrIp: '0.0.0.0/0',
                    FromPort: port,
                    IpProtocol: 'tcp',
                }),
            ]),
        });
    });

    test('Fargate load-balanced service created, wi
        const port = 80;
        const desiredCount = 1;

        // Test code
        const service = addLoadBalancedService(stac

        // Check result
        const template = Template.fromStack(stack);

        template.resourceCountIs('AWS::ElasticLoadB
        template.hasResourceProperties('AWS::Elasti
```

```
    });
});

test('Scaling settings of load balancer', () =>
    const port = 80;
    const desiredCount = 2;
    const service = addLoadBalancedService(stac

    // Test code
    const config = {
        minCount: 1,
        maxCount: 5,
        scaleCpuTarget: { percent: 50 },
        scaleMemoryTarget: { percent: 50 },
    };

    setServiceScaling(service.service, config);


    // Check result
    const scaleResource = new Capture();
    const template = Template.fromStack(stack);
    template.resourceCountIs('AWS::ApplicationA
    template.hasResourceProperties('AWS::Applic
        MaxCapacity: config.maxCount,
        MinCapacity: config.minCount,
        ResourceId: scaleResource,
        ScalableDimension: 'ecs:service:Desirec
        ServiceNamespace: 'ecs',
    });

    template.resourceCountIs('AWS::ApplicationA
    template.hasResourceProperties('AWS::Applic
        PolicyType: 'TargetTrackingScaling',
        TargetTrackingScalingPolicyConfiguratic
            PredefinedMetricSpecification: Matc
```

```
                    TargetValue: config.scaleCpuTarget.
                }),
            });

            template.hasResourceProperties('AWS::Applic
                PolicyType: 'TargetTrackingScaling',
                TargetTrackingScalingPolicyConfiguratic
                    PredefinedMetricSpecification: Matc
                        PredefinedMetricType: 'ECSServi
                    }),
                    TargetValue: config.scaleMemoryTarg
                }),
            });
        });

    });
```

test/monitoring.test.ts

```
    import { Stack } from 'aws-cdk-lib';
    import { Template } from 'aws-cdk-lib/assertions';
    import { Vpc } from 'aws-cdk-lib/aws-ec2';
    import { Topic } from 'aws-cdk-lib/aws-sns';
    import { initMonitoring, MonitoringConfig } from '.
    import { addCluster } from '../lib/containers/conta

    test('Init monitoring of stack, with only defaults'
        const stack = new Stack();

        const monitoringConfig: MonitoringConfig = {
            dashboardName: 'test-monitoring',
        }
        const monitoring = initMonitoring(stack, monitc

        const template = Template.fromStack(stack);
```

```
        template.hasResourceProperties('AWS::CloudWatch
            DashboardName: monitoringConfig.dashboardNa
        });
    });

    test('Init monitoring of stack, with SNS topic for
        const stack = new Stack();
        const vpc = new Vpc(stack, 'vpc');
        const cluster = addCluster(stack, 'test-cluster
        const alarmTopic = new Topic(stack, 'alarm-topi

        const dashboardName = 'test-monitoring';
        const monitoringConfig: MonitoringConfig = {
            dashboardName,
            defaultAlarmTopic: alarmTopic,
        };

        const monitoring = initMonitoring(stack, monitc

        expect(monitoring.defaultAlarmTopic).toEqual(al
        expect(monitoring.defaultAlarmNamePrefix).toEqu
    });

    test('Init monitoring of stack, with SNS topic for
        const stack = new Stack();
        const vpc = new Vpc(stack, 'vpc');
        const cluster = addCluster(stack, 'test-cluster
        const alarmTopic = new Topic(stack, 'alarm-topi

        const dashboardName = 'test-monitoring';
        const alarmPrefix = 'my-prefix';
        const monitoringConfig: MonitoringConfig = {
            dashboardName,
            defaultAlarmTopic: alarmTopic,
            defaultAlarmNamePrefix: alarmPrefix,
        };
```

```
        expect(monitoring.defaultAlarmTopic).toEqual(al
        expect(monitoring.defaultAlarmNamePrefix).toEqu
    });
```

THIS ARTICLE WAS UPDATED ON OCTOBER 4, 2022

Erik Lundevall-Zara

Erik Lundevall-Zara has worked for more than 25 years in the IT industry, as well as writing articles for magazines and teaching computer science topics in corporate and university settings. In 2012 he took his first steps into cloud computing with AWS, and since 2015 worked full-time with making cloud solutions. Erik has a passion for learning and sharing knowledge, which this site is an expression of.

Tidy Cloud AWS

PREVIOUS POST

Tidy Cloud AWS issue #31 - Terraform, CDKTF, interesting AWS blog posts, picking infrastructure as code language

NEXT POST

Tidy Cloud AWS

Tidy Cloud AWS issue #32 - Continuous delivery, Pulumi, Terraform, AWS CDK ninjas and CloudFormation

Related posts

Tidy Cloud AWS

How to become an infrastructure-as-code ninja, using AWS CDK - part 4

DECEMBER 6, 2021

How to become an infrastructure-as-code ninja, using AWS CDK - part 3

NOVEMBER 22, 2021

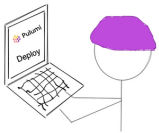
How to become an infrastructure-as-code ninja, using AWS CDK - part 2

NOVEMBER 15, 2021

Featured

From AWS ClickOps to infrastructure-as-software - importing with Pulumi

NOVEMBER 29, 2022

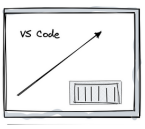


Testing Pulumi Deployments - deployments mini-challenge

NOVEMBER 16, 2022

How to Go with Pulumi YAML

NOVEMBER 7, 2022



Boost productivity with Visual Studio Code dev containers

OCTOBER 31, 2022

Tidy Cloud AWS

OCTOBER 25, 2022

How to become an infrastructure-as-code ninja, using AWS CDK -part 8

OCTOBER 3, 2022

How to pick an infrastructure as code language

SEPTEMBER 15, 2022

How to become an infrastructure-as-code ninja, using AWS CDK - part 7

MAY 29, 2022

Authors

Erik Lundevall-Zara

POST: 65

Want more AWS cloud articles?

Every second Thursday I send out the **Tidy Cloud AWS bulletin** with tips, learnings and news to help you on your Cloud journey in AWS.

Email Address

SOUNDS GOOD, I CAN TRY THAT!

We won't send you spam and respect your privacy. Unsubscribe at any time.

Tags

AppRunner	**(2)**
Article	**(27)**
Automation	**(2)**
AWS	**(50)**
blog	**(2)**