SRE Challenge Journal

Reading this journal

My thought stream appear in blue bubbles.

> This is an example of a thought.

Decisions I make appear in green bubbles.

> This is an example of a decision.

Timekeeping entries appear in yellow bubbles.

> Total machine time is 0 hours, 0 minutes.

Receiving the challenge

> 2022-11-2 @ 9:30am – Looked at the challenge PDF for the first time.

> I noticed there is some emphasis placed on high availability and auto scaling the application but the backend mentioned it uses a in-memory database. It seems unlikely the application will behave correctly if auto scaled if it isn't stateless.

> I sent an email to Sang-Woo asking about this point and seeking clarification on the scope of the exercise.

> I estimate I should have time to complete the exercise before Saturday 2022-11-05.

> Sang-Woo confirmed that adjusting the application code isn't in scope. I will therefore build in auto scaling without considering application code changes in my solution.

Initial thoughts and planning

2022-11-2 @ 1:45pm – Starting journal and capturing my distilled thoughts from this morning.

I will need to get the application cloned and running on my machine in the first instance.

I will use my personal AWS cloud account as a deploy target during this exercise.

I will use Terraform to declare the infrastructure as code. CDK might be a better choice, as we are specifically targeting AWS right now, but it means putting all our eggs in AWS and that makes me a little uncomfortable. While the platform agnosticism of Terraform is a bit overstated it will at least mean some of our constants will be platform agnostic.

I will get started by cloning the code from the ClearPoint repository and pushing it to my own repo. (I will deliberately not fork the repo as I don't want to undermine ClearPoint's assessment by having a direct link on GitHub).

Total machine time is 0 hours, 15 minutes.

Setting up a development environment

I need to get a few things in order on my development workstation because I recently formatted and I don't have a lot of things installed that I will need.

Setting up version control

2022-11-2 @ 2:00pm – Cloning repository and pushing a copy to my own personal GitHub to work against.

2022-11-2 @ 2:10pm – New repository is created.

Total machine time is 0 hours, 25 minutes.

Setting up docker

2022-11-2 @ 2:15pm – Installing `docker-compose` and `docker` on my workstation.

While it isn't strictly necessary to continue, next I want to get Docker and Docker Compose running on my workstation (I haven't installed them on this machine since its latest re-format) and ultimately get the application running locally.

The default package managed version of docker-compose isn't high enough to run the project (its only version 1.25.0 and I need 3.9.x). That is what I get for running a Debian based Linux distribution I guess. I will probably need to add some Docker maintained software repository to get a later version of compose.

Found a handy guide that only needed some simple changes to match my distribution to work.

2022-11-2 @ 2:30pm – Managed to get docker installed and have a version of compose that can run the project.

Total machine time is 0 hours, 45 minutes.

Running things locally

I will now try start the app up locally.

The build time for the images is a few minutes. I will want to keep that in mind if I end up doing the stretch objective of building the code integration and deployment pipelines.

I enjoy the humour in the app. “Oh snap! You got an error!” made me laugh.

2022-11-2 @ 2:45 Compose-based local environment is now running and I can access the app locally. I think I am all set to get started proper.

Total machine time is 1 hours, 0 minutes.

Setting up Terraform

Now I will set up Terraform and connect to my AWS account from my workstation.

2022-11-2 @ 4:30pm – Setting up Terraform.

Like with Docker, if I want a modern version of Terraform I am going to need to set up a Hashicorp managed package repository and install it from there.

The Terraform website actually provides commands to add the Hashicorp managed package repository to Debian, so I can just follow their commands here https://www.terraform.io/downloads

Turns out it wasn’t quite that simple. I needed to swap out the `$(lsb_release -cs)` that incorrectly populated because I am not running a standard Debian or Ubuntu install.

2022-11-2 @ 4:45pm – I have Terraform 1.3.3 installed and working now.

Total machine time is 1 hours, 15 minutes.

Making a change to AWS via Terraform

2022-11-2 @ 4:45pm – I am thinking about how to proceed.

I want to set up connectivity to AWS to show it works. I know I will need to generate an access key before Terraform will be able to apply to the account. If I start defining some simple resources then I can use the terraform cli to test if I have things configured right.

I will start by defining the ECS cluster that I am sure will be part of my solution in Terraform.

I will add Terraform code into an `infrastructure` directory in the existing version controlled repository.

I am not likely to be doing anything very original at this stage so I expect to take most of my code from examples online, at least early on.

I figure my solution is going to essentially follow the pattern in this Hashicorp example using ecs and an application load balanacer, so I will poach my starting code from there.

https://github.com/hashicorp/terraform-provider-aws/tree/main/examples/ecs-alb

Getting Terraform to create an ECS cluster

2022-11-2 @ 5:05pm – I am copying over enough code from the ECS example from Hashicorp to create an ECS cluster.

I have copied over the main.tf from the ECS example and now I am stripping out everything except the cluster resource.

A `terraform validate` shows the AWS region variable isn't specified because I haven't copied that over.

I will grab variables.tf from the example code too.

The default region is `us-west-2`. That isn't ideal for us.

I will change the default region to `ap-southeast-2` (Sydney) because that makes for sense over here.

A `terraform validate` passes now so `terraform apply` should help us tease out the AWS account key, profile etc. we need next.

As expected, we get an error from `terraform apply` saying we need to configure a credential source. I need to set this up now.

DONE: I will create a new AWS account for this exercise and generate an API key for programmatic access. I will use the root user for this, but we really should create a role for infrastructure management at a later point.

I have created an AWS profile called `SREAssessment.Infrustructure` that I will now configure in Terraform.

I have create a Terraform variable for the AWS profile to use. This feels like the best compromise to keep secrets out of version control, specify sensible defaults and allow other developers to override the profile.

I accidentally forgot to change the default region from the Hashicorp example (even thought I said I would) and I was confused for a moment why my `terraform apply` worked but I saw no cluster in the admin console. It is so easy to accidentally generate resources and not know where they are.

2022-11-2 @ 5:45pm – It works, I have successfully generated an ECS cluster.

Total machine time is 2 hours, 15 minutes.

Getting the Hashicorp “Ghost” example running in AWS

The Hashicorp ECS example isn’t exactly what I am looking for (DONE: I am going to modify it to run via Fargate instead of EC2 for example). But it seems like a good foundation and it lets me defer dealing with some of the other parts of the challenge till after I have a working set of containers.

I will copy over the rest of the example Terraform from Hashicorp and get their copy of Ghost running with it.

2022-11-3 @ 9:45am – Copying the rest of the Terraform example over and getting it to run.

I have had to copy over configuration for files I know I don’t actually need for my solution. I am not expecting to have EC2 instances as part of my solution. I think it should be straightforward to simplify things once I have something working.

Ah. Hit something I should have anticipated. I need to provide a number of configuration items for accessing the EC2 instances directly that I don’t need. It might be easier to swap to Fargate sooner rather than later.

Change of plans. I am going to strip out the EC2 instances that I don’t need upfront.

I can let `terraform validate` steer me to some degree as I remove things I don’t need, but I need to be very wary of leaving resources I don’t need defined.

2022-11-3 @ 10:15am – Trimmed things down to not use EC2 and instead use Fargate managed instances. Going to try and apply it now.

Encountered a problem. We need to adjust the task definition to launch on Fargate.

Looks like the task defintion resource needed `requires_compatibilities = ["FARGATE"]`. I am not certain what that does.

Another problem. Fargate network needs to be `awsvpc`. I need to look into this. Perhaps copying this example wasn’t such a great idea!

While reviewing the code, I noticed there is a security group we no longer need. I will remove it.

I hope everything runs once I have got past the errors. I have stripped so much of the example code away to run on Fargate now. It would be unfortunate if I encounter some deal breaker than means I have to abandon the example code.

I needed to make some tweaks to log configuration. We need to specify a log stream prefix when using Fargate.

Creating the load balancer via Terraform takes about two minutes, and other resources are not created till after it. This makes the feedback loop for fixing problems longer than would be ideal.

We need to add a role for the Fargate tasks to run as now that we have moved away from EC2. I will create this role now, I will use the AWS managed role because we have no special permissions for these tasks.

Another error, we need to define the instance size details for Fargate. These were implied by the EC2 instance previously. I will add the cpu and memory keys.

I just noticed there are a bunch of unused variables in the Terraform now. I will clean them out.

Now our target group is incompatible with our tasks. I need to dig into how to resolve this. Networking is a less familiar area of AWS for me.

It appears we were getting a default target group that targeted instances, not IP addresses. Because Fargate will add tasks to a target group based on their IP, we need to explicitly set the `target_type = “IP”` when using Fargate.

A follow on to the above is that we now need to provide a subnet to the ECS service to allocate IP addresses from.

It turns out we don’t need to specify an ECS service role because we are using simplified networking now. I can remove the explicit service role.

Success! A clean `terraform apply`. Time to check if it is reachable.

Got a bit of a scare when I saw the 503 unreachable. But the task is still starting. It seems to be taking a long time to start. Nothing shows in the logs yet. I will be patient a bit longer.

Got an error from the task that it could not pull the image. This seems surprising, I will confirm if I can pull it myself locally.

Things work for me locally. Comments online suggest this is a problem where I haven't given ECS enough permissions to pull the image from the public internet. I will need to resolve this after lunch.

2022-11-3 @ 12:00am – Terraform is now doing its part and things are running on Fargate.

Total machine time is 4 hours, 30 minutes.

2022-11-3 @ 1:45pm – Had some inspiration during lunch. Going to try adding public ips to the tasks.

DEFERRED: I am currently counting on my instances having direct access to the public internet, but there won't be a long term reason for this to happen so I should undo this later.

Got a bit further. Now I have a problem with Ghost specifically. It looks like it wants a database. I don't want to spend too much time on this.

I am going to try run it in development mode using environment variables.

Finally I reached a point where I can start focusing on running a copy of the provided application.

Next steps will be defining a container registry, adding the docker images for the todo list app that we really want to host to a container registry and then adjusting the task definition to load that app instead of ghost.

2022-11-3 @ 2:15pm – Ghost is running and accessible now.

Total machine time is 5 hours, 0 minutes.

Ghost isn’t stable

2022-11-3 @ 3:15pm – Came back to find that ghost had stopped and was no longer reachable.

It appears to be due to load balancer health check failures.

I will swap out the image for an Apache image instead.

2022-11-3 @ 3:30pm – Apache is stable. There does not appear to be anything wrong with the infrastructure.

Total machine time is 5 hours, 15 minutes.

Getting our build artifacts to AWS

Now we have a web service hosted in AWS we need to swap it out for the app we really want to run. For that we need to get our artifacts into AWS. There are a few things we could do here, but given our artifacts for our development environment are docker images, we can use an Elastic Container Registry to hold the images. This is also a really nice boundary between the build and deploy stages and we can use the container registry to drive automation later.

I will use Terraform to define an Elastic Container Registry next.

Adding ECR to our infrastructure

2022-11-3 @ 4:30pm – Adding an ECR registry to our Terraform.

I will start by adding the resource and making sure it is created. I know we will need to grant permission to the execute role before we can pull from it later.

This part was super easy, there isn't a lot of complexity to the container registry and I was able to follow the documentation easily.

2022-11-3 @ 4:45pm – I have an empty container registry created.

Total machine time is 5 hours, 30 minutes.

Pushing httpd into the container registry

Now we want to start loading our images from our container registry instead of the public internet. I am going to push a copy of the httpd image into the new registry we created next.

2022-11-3 @ 4:45pm – Pushing httpd image into ECR

I need to install the AWS cli in order to get a temporary login to ECR.

Somehow I haven't needed the AWS cli yet on my workstation. Because I manually add keys to my AWS credentials file and that is what Terraform is using.

DONE: I realise at this point we need a front and back repository. I need to rename this one once we get this working.

Got the image pushed. This all feels uncomfortably manual. We will fix this with the integration and deployment automation if we get that far.

2022-11-3 @ 5:00pm – Image pushed. It appears in ECR now.

Total machine time is 5 hours, 45 minutes.

Pull the image from our container registry when starting tasks

Now we need to adjust our task definition file to fetch the image from our container registry. The task execution role will need to be granted permission to pull from the registry for this to work.

2022-11-3 @ 5:00pm – Adjusting the task definition and task execution role to pull from ECR.

I renamed the ECR repositories to match our eventual targets for the todo app.

Curiously, I found that the connection to ECR goes over the public internet, it does not appear to be possible to pull the image privately from ECR. This held me up from a bit while I was experimenting removing the public IP.

2022-11-3 @ 5:45pm – I have the `httpd` image masquerading as the clearpoint todo app frontend.

Total machine time is 6 hours, 30 minutes.

Swap the httpd image for the real todo frontend

The way should be paved for us to push the todo app frontend into the container registry and force a redeploy of the task to render the frontend app. For now, we only support one container, we will change this soon.

I will now replace the httpd image in ECR with the locally built frontend image from my workstation and force a re-deploy of the service.

2022-11-3 @ 5:45pm – Deploying the todo app frontend.

I removed some leftover environment variables from ghost that I forgot about.

2022-11-3 @ 6:00pm – The todo frontend is now available. Everything works as expected.

2022-11-3 @ 6:15pm – Tidied up some old references to example and httpd.

Total machine time is 7 hours, 0 minutes.

DONE: I noticed the frontend Dockerfile consumes a `.env` file, while building the image, to set the API URL. This might get in the way when we try to host both containers from the same domain. I will later need to clarify if environment variables I declare at build time will override the `.env` file. If not, I need to find another way to suppress the development configuration without breaking the development environment.

Getting the todo app to run

The next major milestone is going to be wiring the backend and frontend together on the same domain. There are a few steps left to get there. I need to push the backend artifact to its container repository, I need to create a second service for the

backend artifact (this will probably involve some refactoring of Terraform code to avoid duplication as well) and finally I need to have the application load balancer send API traffic to the tasks for the backend service.

I will define the new service first, cause that is the highest value change to make.

2022-11-4 @ 10:00am – Declaring the backend service in Terraform.

If I do this naively, I am going to add a bunch of duplication to my Terraform. Given there is little difference between how I am running these tasks. I really want to call the same module twice to create the backend and frontend services.

I will extract the common parts of the current, frontend, infrastructure to a separate module.

I will keep the container registries defined separately. In my experience it makes sense to keep interfacing infrastructure separate. In this case our container registry would be the interface between our future build and deployment pipelines, so it makes sense to define them away from the running application infrastructure components.

2022-11-4 @ 10:45am – I have successfully extracted a common `simple-ecs-service` module and everything still creates and runs correctly.

Now I will use this module to define the backend service and attach it to the same cluster.

I renamed a bunch of things in the module that were clearpoint, todo or frontend specific.

I got a name clash on my task execution role. These don't actually need to be unique, but roles are cheap and having the flexibility for them to be different by service later is worth preserving. I have added the application name and service name to the role to resolve the problem.

AWS will not create the backend service until I wired up the target group to the load balancer. I didn't relalise I couldn't create an orphaned target group.

I will have to define the path based routing to the API now so that the target group is attached to the load balancer.

I now have two services in my cluster. But I get cross origin request errors in the browser and its clear that the frontend is trying to send to localhost still. We need to get the frontend to call the backend properly.

Another problem is that I need to push the build image into the container registry. There is no backend image currently.

I will try setting the environment variable I noticed earlier to see if that resolves the cross origin errors. This gets baked in at build time, so I will also have to push a new image and restart the services.

I will also push the backend image while I am doing this to resolve the missing image for that service.

The backend service definitely works. That is a good sign, it means the load balancer is properly configured.

The frontend stopped throwing cross origin errors, which is also good news. Unfortunately I missed the `/api` from the environment variable when I replaced it, so the app didn't work. I need to build it again.

Everything works now! Yay. There are quite a few compromises to overcome and improvements still to make though.

2022-11-4 @ 11:45am – The app works in AWS now.

Total machine time is 8 hours, 45 minutes.

Documenting and describing the current behaviour

This journal has my stream of consciousness and the code in GitHub captures the work I have delivered so far but there is a pretty big gap for somebody picking things up and using it. I need to document how things are set up and how to use it before I make further improvements.

I will start with updating the README.md in the project to add instructions on how to create everything.

2022-11-4 @ 1:45pm – Updating README.md with instructions on how to get things running.

2022-11-4 @ 2:30pm – The README.md now includes instructions on how to set things up.

Documenting the API URL behaviour prompted me to fix up the way we provide the URL during the development experience vs the build for AWS. I added a build argument to the frontend Dockerfile to populate the `REACT_APP_TODO_APP_API_ENDPOINT` environment variable.

A nice bonus is that now we don't actually bake the domain into to the frontend for an AWS build. This means the artifact could be re-used between deployments. That would be key for any meaningful testing that tested the interaction between services.

2022-11-4 @ 3:45pm – The README.md now includes instructions on how to run terraform and push images to the container registries.

Total machine time is 10 hours, 45 minutes.

Summary of TODOs from this journal

I have a number of TODOs I left in this journal. I am going to summarise them here and come up with a plan for each.

2022-11-7 @ 10:00am – Summarising remaining TODO items I noted earlier in the journal.

- Stop using the root account to create resources.
- Remove public IP addresses from tasks

The tasks currently need public IP addresses in order to pull images on startup. My understanding is that it would be best to remove this and set up network address translation instead. However, I am going to leave things as is, given that isn't the primary focus of the exercise.

I have been creating all the infrastructure using the root user so far. I think it is important for account security and to follow the principal of least privilege here and stop doing this. I will address this now.

2022-11-7 @ 10:15am – TODO summary complete and actions decided.

Total machine time is 11 hours, 0 minutes.

Replacing the root user account with an infrastructure user

It isn’t responsible to use the root user for an AWS account or even to issue access keys for that user. I will now create a user account for myself and use that to perform infrastructure operations instead.

There is a tricky clash of principals here. All infrastructure, including IAM users, should be defined by applying infrastructure as code. However, we should never issue access keys for the root user to allow programmatic access via that account. This bootstrapping problem doesn’t have a clear answer.

I will be creating an additional user manually, as I feel that not issuing keys for the root user is more important here. This is my typical behaviour for personal projects though I accept that choice may be different in a workplace.

I could create a group to manage the permissions for my user, if this were a multi-person project this would be practically essential.

It would also be a better again to create a role and grant resource creation and teardown permissions. I could then assume that role via a profile to run Terraform code and my user could have read-only permissions normally. I am going to avoid that additional complexity for now as it isn’t the focus of this exercise.

My new user will have full Administrator access permissions as my goal is simply to avoid using the root user.

2022-11-7 @ 10:15am – Creating a new IAM user for myself.

I have successfully created the user, now I will replace my access for the `SREAssessment.Infrustructure` profile and confirm I can destroy and recreate all the infrastructure with this other account.

That worked great. I will now revoke the old access key for the root user so there is no way those keys could be used again.

I updated the README.md to discourage use of the root user for running Terraform.

2022-11-7 @ 10:45am – I have successfully migrated to an IAM user to build and destroy infrastructure.

Total machine time is 11 hours, 30 minutes.

Reviewing application resiliency

One of the key points of the exercise is to include auto scaling and be highly available.

High availability is going to have a long tail of potential improvements so its hard to know where to draw the line. However, I want to make sure we are starting at least two tasks and the tasks are started in different availability zones.

I definitely haven't enabled auto scaling for the tasks. I will enable this too.

Ensure we are running services across multiple availability zones

2022-11-7 @ 10:45am – Reviewing availability zone configuration.

We have subnets created across two availability zones and Fargate will automatically distribute tasks between the subnets available.

However, the current configuration only creates one task at a time. If there was an outage in the availability zone one of our tasks was running in OR if one of the tasks exited unexpectedly (perhaps due to failing healthchecks) then the service would be unavailable for a time.

I will address this by configuring at least two tasks to be started in each service.

2022-11-7 @ 11:00am – Adjusting Terraform to require two tasks per service.

Success. The Elastic Container Service has now spun up second tasks for the front and back ends and each is running against a different subnet and thus a different availability zone. The application should now be resilient to failure of a single task.

As I suspected. Adding additional backend tasks causes the application to behave unexpectedly on reload.

This is because the application is stateful, the in-memory database is unique to the instance that receives the request. Until this limitation is overcome, the application cannot run correctly while being highly available.

2022-11-7 @ 11:15am – Application is now configured to be highly available.

Total machine time is 12 hours, 0 minutes.

Enable auto scaling of services

To ensure we can keep up with any load thrown at the service we can enable auto scaling. The Elastic Container Service will then add additional tasks to manage spikes in load that occur.

Auto scaling can lead to spikes in AWS cost too, so it will be important to make the scaling values configurable in Terraform, along with some sensible defaults.

2022-11-7 @ 11:15am – Enabling auto scaling.

Without profiling the applications it is hard to know what to set as auto scaling thresholds. Ideally this would be informed by load testing.

As load testing isn't the focus of this exercise I will just choose an arbitrary CPU threshold of 80% load.

2022-11-7 @ 12:00am – Auto scaling on CPU load is now enabled.

Total machine time is 12 hours, 45 minutes.

I have ticked all the required boxes for the exercise now. I need to add a diagram of my solution.

I would also like to add Continuous Deployment driven off the container registry before I submit.

Add a diagram showing the infrastructure

I suspect I can find something built by somebody else as I am not using a very unique design.

2022-11-7 @ 12:00am – Looking for an existing diagram that matches my solution.

I could not trivially find a diagram I thought matched what I designed. I will build one myself.

2022-11-7 @ 12:30pm – I have created a diagram using an online tool.

Total machine time is 13 hours, 15 minutes.

Adding a Continuous Deployment pipeline

I want to have a go at the stretch objective of creating a Continuous Deployment pipeline because I think it is important to build automated deployment for applications alongside the infrastructure decisions.

I also think Continuous Integration of code is essential too, but it doesn't have quite the same connection to infrastructure. In fact, I don't even think AWS has appropriate tools for CI in the first place.

For that reason I don't think building a CI pipeline is really as meaningful for this assessment.

I will build the automated deployment to run from the container registries.

2022-11-7 @ 3:45pm – Enabling Continuous Deployment driven by `latest` tags in ECR

Our task definition files are already set to pull to the `latest` tagged images from our container registries. This means all we really need to do to enable continuous deployment is to monitor the container registry for changes and then, when changes are observed, we force a deployment of fresh tasks in ECS.

CodePipeline in AWS can do this for us, we just need to add and configure the resources to do it.

I had to change the application name. I was including underscores and then using the name to derive an S3 bucket name. Bucket names cannot include underscores. Replaced them with dashes.

Caught a snag. I need find a way to create an `imagedefintions.json` for the ECS deployment. I just want to use the latest tag, but CodePipeline wants me to be more specific than that.

This stack overflow [https://stackoverflow.com/questions/58849736/did-not-find-the-image-definition-file-imagedefinitions-json] has a neat CodeBuild script we can place in between ECR and ECS to convert the `imageDetail.json` that ECR provides to the `imageDefinitons.json` file that ECS requires. I will try adding this stage.

It looks like if I set up a full Blue/Green deployment using CodeDeploy this would all pipe together nicely. That will probably be my fallback plan.

Change of plans. Adding a CodeBuild stage just to return a static file seems excessive. Instead I will create an S3 bucket and upload the `imageDefinitions.json` in. We can do all this with Terraform.

It appears we have to enable versioning of the S3 bucket to use it as a CodePipeline source.

Got things defined now. I need to run things to see what permissions I need to add to the pipeline role. I suspect I need ECR and ECS permissions to be added.

As expected. I need to add ECR permissions to the pipeline role to be able to monitor for new images.

Got stuck for a long time till I found some documentation saying I needed to zip the imagedefintions.json file. Now I need to work out how to get Terraform to do that.

Success. Next error is with permissions to interact with ECS. Feels like its getting close.

I ended up adding the managed policy `AmazonECS_FullAccess` which seems like more permissions than we should need, but more restrictive policy just gave me an

unhelpful “The provided role does not have sufficient permissions to access ECS” error.

2022-11-7 @ 7:00pm – The continous deployment pipeline is successfully updating tasks when run manually.

This seemed to have worked! I now have a continuous deployment pipeline. I need to confirm it properly triggers when I push a fresh image.

I will make a change to the frontend and confirm it works.

2022-11-7 @ 7:00pm – Doesn’t look like it is working. I will need to investigate later.

Total machine time is 16 hours, 45 minutes.

Fixing the bug with the pipeline triggering

2022-11-7 @ 9:15pm – Taking a look into why a push to ECR doesn’t trigger the pipeline to run.

It appears there is a limitation with the Terraform module that it does not create the CloudWatch event to monitor ECR when we define the action for CodePipeline. We need to create an event ourselves to trigger it.

I found some example code I will try out. https://github.com/hashicorp/terraform-provider-aws/issues/7012#issuecomment-834697378

2022-11-7 @ 10:00pm – I have the CloudWatch Event configured to monitor ECR for pushes.

I will now try making another change to the frontend and pushing a new image to ECR.

2022-11-7 @ 10:00pm – The continuous deployment pipeline triggers when a new image is pushed to the container registry.

Total machine time is 17 hours, 30 minutes.

Conclusion

Summary

I think I am pretty happy with the result. There are some clear things I would add next, but they are not in the scope of the assessment. I will capture some of my ideas for next steps tomorrow and submit the solution.

It took me much longer than I had anticipated spending, but I really enjoyed building some of these fundamental pieces I have taken for granted on more established projects in the past.

2022-11-8 @ 9:45am – Capturing final thoughts.

Known limitations and outstanding problems

There seems to be some persistent problem with the backend service. Tasks will keep dropping out of target groups due to failing health checks. When I did a brief poke around, it looked like it might be connected to missing HTTPS configuration.

This would need further investigation, but it didn't feel like it was part of the scope of the exercise.

The backend service is stateful. It has a in-memory database so each instance has its own version of the truth of the todo list. This means the service doesn't scale properly. When you make a request you might hit a different service than the one you sent your last todo item to. This causes problems when you reload the frontend cause it seems like your todo items disappear.

I got told not to worry about this when I asked about it.

Room for improvement

Public IP addresses shouldn't be assigned to tasks. They should just have private addresses and get their access to ECR via a NAT resource instead.

I probably should have done this, and its a key thing to change next if more effort went on this.

The Terraform could use more tidying up. My focus was on removing duplication, but there is some separation of concerns refactoring, in particular with the Continuous Deployment resources conflated with the `simple-esc-service` module. I felt this building up as I ingested more variables in the module.

The entire module could also be parameterised to make it easy to compose together this service as part of a wider solution later.

This would be a high value, low effort change to make and there are probably other opportunities for improvement in there too.

We could add a smoke test during the continuous delivery pipeline that tested the overall behaviour of the application. This would act as a final safety net to help inform us if anything had gone wrong with a deployment and needed urgent attention.

We could also further improve the deployment with blue/green to run the smoke test making the new environment live. This would further improve resiliency.

The frontend is currently running in ECS, but its just static assets so it should run out of an S3 bucket with a content delivery layer in front instead.

I made a deliberate decision here that my build artifacts were docker images, the images built and used in the `docker compose` development environment. There is

a tradeoff here between similarness to development and performance/resiliency that could be improved.

Continuous Integration of code and build artifacts should be added. This is absolutely essential to healthy software delivery and is a must have for any reasonably sized project.

I think the tools AWS provides, e.g. CodeCommit, CodeBuild and CodePipeline are not fit for this purpose, so I haven't explored this during this exercise. CI should be about quickly getting information about test and build results and should be well integrated with the version control system (in my opinion) to provide contextual feedback and AWS tools are best in class at this.

2022-11-7 @ 10:15am – Final thoughts captured.

Total machine time is 18 hours, 0 minutes.