Voice Foundry Technical Demonstration Assessment

Author: Oliver Hanson-Bragg

Date: 2021-02-28

Contents

[Requirements 3](#_Toc65438872)

[Foreword 3](#_Toc65438873)

[Deployment Guidelines 4](#_Toc65438874)

[Prerequisites 4](#_Toc65438875)

[Instructions 5](#_Toc65438876)

[Solution Overview 8](#_Toc65438877)

[Infrastructure Overview Diagram 9](#_Toc65438878)

[Solution Component Description 9](#_Toc65438879)

[Back-end 9](#_Toc65438880)

[Front-end 10](#_Toc65438881)

[API Gateway 10](#_Toc65438882)

[Solution Implementation Rationale 10](#_Toc65438883)

[Development Struggles and Problems Overcome 11](#_Toc65438884)

[Generating Vanity Numbers 11](#_Toc65438885)

[Custom Resource 12](#_Toc65438886)

[Create\_contact\_flow 12](#_Toc65438887)

[Front-end Development 13](#_Toc65438888)

[Bad Practice / Shortcuts 13](#_Toc65438889)

[Additions / Improvements Given Extra Time 14](#_Toc65438890)

[Productionisation 16](#_Toc65438891)

# Requirements

Deliverables:

1. Git repo containing all code and documentation
2. **BONUS**: A working AWS Connect phone number to test in your environment

Exercise:

1. Create a Lambda function that converts phone numbers to vanity numbers and save the best five resulting vanity numbers and the caller’s number in a DynamoDB table. “Best” is defined as you see fit – explain your thoughts.
2. Create an Amazon Connect contact flow that looks at the caller’s phone number and says the three vanity possibilities that come back from the Lambda function.
3. Create a custom resource for CloudFormation that will allow you to publish the contact flow to a Connect instance with the appropriate Lambda ARN already in the contact flow so that there is no manual configuration in the flow.
4. Built a deployment package with AWS SAM, AWS CDK, or CloudFormation to allow a user, or assignment reviewer, to deploy your Lambda, custom resource, and anything else you’d like to add into their own AWS Account/Amazon Connect Instance.
5. **SUPER BONUS**: Create a web app that displays the vanity numbers from the last five callers
6. Writing and documentation:
   1. Record your reasons for:
      1. implementing the solution in the way that you did
      2. struggles you faced
      3. problems that you overcame
   2. What shortcuts did you take that would be a bad practice in production?
   3. What would you have done with more time?
   4. What other considerations would you make before making our top app into something that would be ready for high volumes of traffic, potential attacks from bad folks, etc.?
   5. Please include an architectural diagram

# Foreword

I would like to thank Dom and Lorne for the opportunity to undertake this assessment. I feel like the assessment has provided a great challenge and lends itself to the wide range of ‘disciplines’ and therefore is excellent at highlighting a candidates strengths and weaknesses in each area. As a result, the assessment did a really good job of highlighting where I have some noticeable weaknesses: specifically, what I refer to as ‘proper dev stuff’ (i.e. algorithms and *actual* programming – not just simple CRUD), and web development. Regardless of the continuation of my application to Voice Foundry, there are a few elements which I will definitely be investigating further and taking back to work for incorporation into our own AWS practice: namely custom resources, and SAM.

On the other hand, whilst not utilising AWS SAM or CDK as would be ideal, I did feel much more at home with the CloudFormation and general deployment and configuration of the AWS resources and enjoyed building this solution.

On the whole I feel like I successfully addressed most of the requirements, except the premise around which the entire solution revolves – the generation of vanity words from a telephone. I’m still not sure whether I have fundamentally misunderstood the requirement or whether I’m much less clever than I thought, because building such a function / solution (that is reliable) appears verydifficult to me… and I can’t help but think most places with them started with the word they wanted and then worked backwards to the number, not the other way around.

# Deployment Guidelines

This section is intended to provide the reader with instructions on how to deploy the deployment package provided.

In general, the project is laid out as below:

root

aws-resources

front-end

back-end

api-gateway

src

lambda

connect

api

misc

web

css

js

The root directory contains .bat files which utilise AWS CLI commands to package and deploy CloudFormation templates and in some cases use other service’s CLI commands to take actions. The execution of these .bat files is how the resources are deployed.

The deployment is split into three ‘modules’:

1. Back-end
2. Front-end
3. API Gateway

The reason for this is that in a production environment these can be classified as three separates ‘products’ and therefore are separated to each other to prevent unnecessary regression testing. For example, when you change the front-end you don’t risk breaking the back end and API Gateway, and vice versa.

The CloudFormation templates for which are found in the aws-resources directory. The python function code for the Lambda functions is found in the src/lambda directory, and finally, the code utilised for the website is found in src/web.

As a result, the deployment is in three stages later described.

## Prerequisites

1. AWS CLI configured to deploy resources into the target account with the ability to create the following Services:
   1. API Gateway
   2. Lambda Function
   3. IAM Role
   4. Log Group
   5. DynamoDB Table
   6. CloudFormation Custom Resource
   7. S3 Bucket
   8. CloudFront
2. An existing AWS Connect Instance in the same region as to which you wish to deploy these resources

If IAM credentials are not located directly in the target account and a tool such as awsume is not used to manage profiles, then it may be necessary to modify the xxx\_deployment.bat files found in the root folder to include “--profile “ in each of the AWS CLI commands.

## Instructions

1. Note the following information to be passed as command line arguments:
   1. Region (e.g. eu-west-2)
   2. Environment (e.g. dev/test/prod)
   3. AWS Connect Instance Arn (e.g. “arn:aws:connect:eu-west-2:589882802822:instance/f5310adc-eab7-444f-9ab6-c7a77a389353”)
2. Open PowerShell or CMD in the projects root directory and run the following commands:

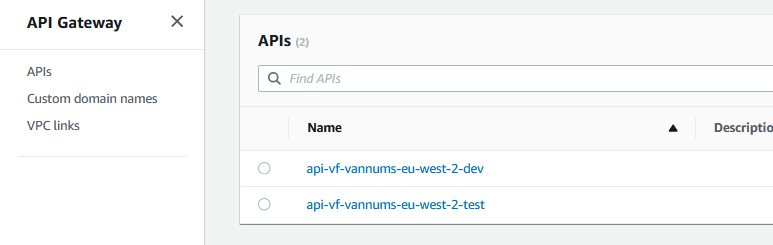
“./deploy\_backend.bat {region} {environment} {connectArn}”

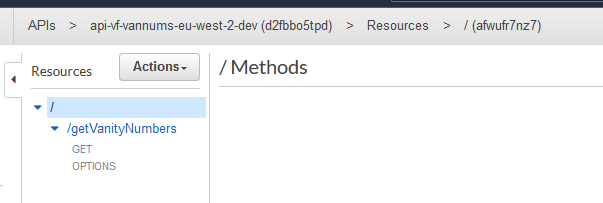
Note: The values in curly brackets should be replaced appropriately by the values identified in step 1.

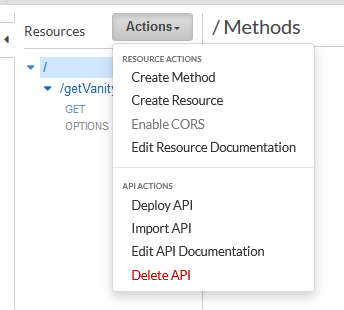
1. Following successful deployment of deploy\_backend.bat, run command:

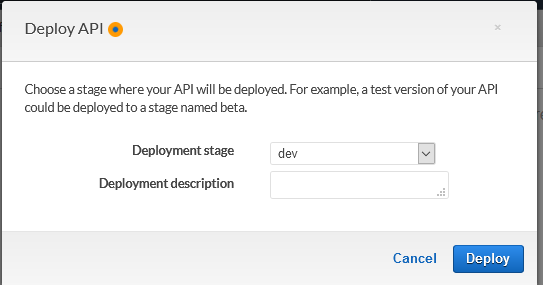
“./deploy\_apigateway.bat {region} {environment}”

1. Following successful deployment of deploy\_apigateway.bat: navigate to the API Gateway service in the target region in AWS Management Console. Identify and open the API which has been deployed, and from the *Actions* drop-down, select *Deploy API*. From the Deployment stage drop down, select the only value, and press *Deploy*.

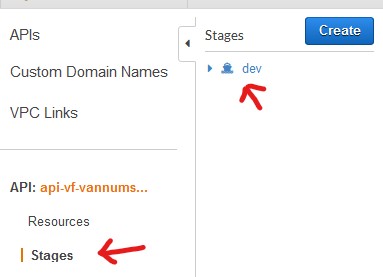








1. After the API has been deployed, navigate to the *Stages* tab on the left and select the Deployment stage chosen in the drop-down box in the previous step:



1. Take note of the Invoke URL, e.g. https://d2fbbo5tpd.execute-api.eu-west-2.amazonaws.com/dev
2. Open file src/web/js/index.js and on line 5 replace API\_GATEWAY\_URL\_GOES\_HERE/getVanityNumbers with the Invoke URL from the previous step, such that the resulting string value for the url variable is for example: https://d2fbbo5tpd.execute-api.eu-west-2.amazonaws.com/dev/getVanityNumbers

Save changes.

1. Open file deploy\_cloudfront.bat and on line 30 replace “ENV\_HERE” with your environment name, e.g. dev/test/prod, and REGION\_HERE with your region, e.g. eu-west-2. As a result, your String should look like:

*“aws s3 cp ./../../src/web s3://s3-vf-vannums-****dev****-****eu-west-2****-webapp --recursive”*

Save changes.

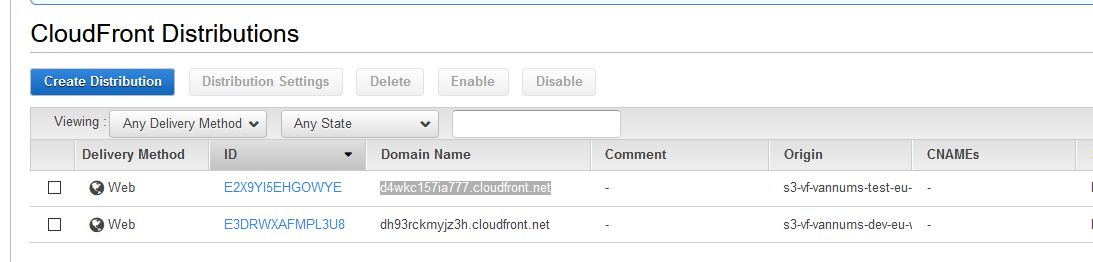
1. Return to PowerShell or CMD, run command:

./deploy\_cloudfront.bat {region} {environment}

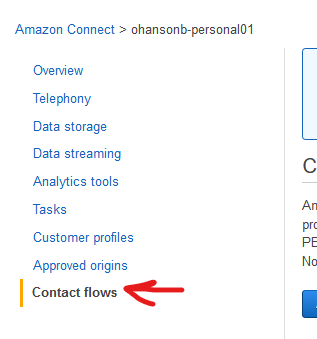
1. Navigate to CloudFront and identify the distribution that has been created upon successful deployment in the previous step. The URL for to access the website will be:

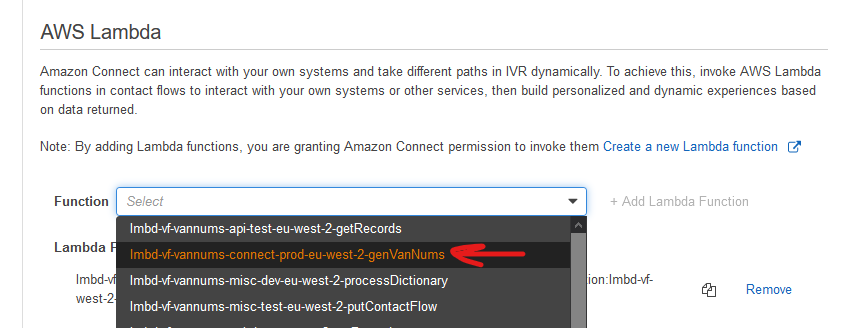
“https://” + Distribution Domain Name, e.g. <https://d4wkc157ia777.cloudfront.net>

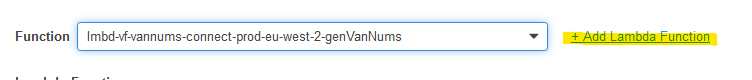
Note: The appropriate distribution for each ‘environment’ can be identified by the Origin (S3 bucket) name. Additionally, note that it may take several hours for the website to become available.



1. Open your AWS Connect Instance in your target region
2. In the Instance overview settings, navigate to the *Contact flows* section and within the AWS Lambda section select Lambda function with name “lmbd-vfn-vannums-connect-{environment}-{region}-genVanNums”, and press “+ Add Lambda Function”.







1. Finally, in your AWS Connect Instance claim / allocate a telephone number to the Contact Flow which has been created. Wait 30-60 seconds, and call to test.

# Solution Overview

As put forth in the requirements, the solution herein described includes a deployment package which deploys a series of AWS resources which are used with AWS Connect to take a callers telephone number, generate several words from their telephone number (vanity numbers), read several of these back to the caller, and then store these in a DynamoDB table. Further, a CloudFront distribution and API Gateway have been included to provide a basic web front-end which displays the details of the five most recent callers to the AWS Connect contact flow.

Due to a lack of previous experience with AWS SAM or CDK, the deployment package is comprised of three .bat files in turn to deploy the constituent solution components, from a ‘local’ desktop environment:

1. Connect flow and backend
2. CloudFront distribute and web code
3. API Gateway.

I have tried to provide as many of the resources as necessary to fully deploy the application within the deployment package, including the use of AWS CLI commands within the .bat deployment files to automate some deployment tasks. However, due to the time restricted nature of the assessment this has not been 100% possible.

In the event of a failure to deploy or for other reasons (such as that CloudFront takes several hours to sort itself out) the below details the telephone number and website URL the deployment which I have created in my own AWS account:

**Telephone number:** +44 20 3467 2906 **Website URL:** <https://dh93rckmyjz3h.cloudfront.net/>

Whilst using Jira for issue management at work, for personal projects and individual work I use Trello as a Kanban style system for tracking work. As such, please find attached the Trello board that I used for this project. This board highlights the work items complete, those that I still thought were required, and those that I thought were improvements or nice to have:

<https://trello.com/b/JyZwnxfg/voice-foundry-assessment>

## Infrastructure Overview Diagram

Diagram

Description automatically generated



## Solution Component Description

As previously stated, the solution is generally split into three constituent components:

1. Connect flow and back-end deployment
2. CloudFront distribution and website
3. API Gateway

The purpose of this section is to briefly describe the CloudFormation templates / resources involved with each part.

### Back-end

The backend deployment deploys two CloudFormation templates:

1. aws-resources/backend/Deployment\_bucket.yml
2. aws-resources/backend/Backend-deployment.yml

The first template deploys and S3 bucket into which the deployment packages of all following deployments can be created. As such, the deployment of the back end is a requirement for the other two components.

The second CloudFormation template deploys the following resources:

* DynamoDB table
* Lambda function for generating vanity numbers from Connect flow
* Lambda function for processing a dictionary file (unused)
* Lambda function to put the contact flow into the target AWS Connect Instance
* A Custom Resource which triggers the putContactFlow function

### Front-end

The front-end deployment deploys CloudFormation template aws-resources/frontend/cloudfront-deployment.yml.

This CloudFormation template includes an S3 bucket to serve as the repository of HTML, JavaScript, and CSS code used by the CloudFront distribution (and other associated resources) which are also created by this template.

### API Gateway

The API Gateway deployment deploys CloudFormation template aws-resources/api-gateway/apigateway-deployment.yml.

This template launches an API Gateway resource with a single method named /getVanityNumbers with GET and OPTIONS methods. The Lambda function invoked by this method is also included.

# Solution Implementation Rationale

My experience with AWS, AWS Connect, and programming in general is restricted to what has been learnt over the past two to two and a half years through the introduction of AWS technologies to my company. In other words, we (me, my current employer, etc.) don’t have any previous AWS or programming experience to rely on and therefore by and large the approach that I have taken is one that we have ‘developed’ over a period of time and has been found to work for us. In summary, the solution has been deployed in the way that it has because this is the way that I currently know how to deploy such solutions.

This is all to say that I am quite aware that compared to SAM and CDK the method of using .bat files is probably somewhat ‘janky’, and from my first interview is clearly dissimilar to how resources and projects are deployed within Voice Foundry. However, I do quite like deploying resources in this way for a few reasons:

1. There are minimal pre-requisites (more or less just the AWS CLI)
2. You stay quite close to the bone in terms of abstraction (or lack thereof), which in my view makes teaching non-developers a bit more straightforward (which is a common thing for me)
3. Another key reason that we take this approach is because ultimately the deployment package can then be quite easily converted into a CodeBuild project such that it is deployable across multiple environments on an as-needs basis by relatively non-technical staff at the click of just a few buttons (again, a common thing for us).

In regard to the lower-level aspects of the implementation, there has been a necessary balance between ‘proof of concept’ and ‘production’ level configuration given the time allotted and my own personal expertise. In later sections I will cover some of these.

# Development Struggles and Problems Overcome

Mostly due to lack of experience there have been a few areas where significantly more time was spent than expected. In most cases however, I have overcome or worked around the issues that I have found.

The key areas where I struggled that I think are worth shouting out are:

1. By and large the entire concept of vanity numbers and the realistic feasibility of turning phone numbers into valid English words. This is the key main area where I have obviously not delivered against the specification.
2. Custom Resource and cfnresponse (or lack of) causing the CloudFormation deployment to hang for an hour on creation and deletion
3. AWS Connect’s create\_contact\_flow API call
4. Front-end development in general

## Generating Vanity Numbers

I will be honest that when I read the first bullet point of the exercise, I scratched my head and went off to Google for a bit before sending an email to Rachael.

As a telephony engineer the only concept of vanity numbers that I have previously are with non-geographic telephone numbers, where some companies request ‘pretty’ numbers such as for example 0800 00 1066 (for Hastings Direct) or numbers that are easily memorable or otherwise jingle-friendly. Converting a mobile number to such a number didn’t make much sense at all to me.

The only alternative and the option that I took as my working assumption as the correct choice was the more American usage of vanity numbers, or phone words, where a common example would be 1-800-FLOWERS; where the characters in the word FLOWERS represent a single press of the corresponding digit on an ‘old-style’ telephone keypad (e.g. 1-800-3569377). However, this did admittedly fill be with a little bit of dread initially. Conceptually I could grasp at least one way to approach this problem (the way that I chose in the end but didn’t complete) but on Googling for ideas and speaking with my partner (who works for a consultancy doing data science and ‘AI/ML’) there was talk of graph theory and all sorts of things which are far beyond my capability to either intelligently grasp or reproduce (especially in the time I had available).

As such, the general idea was that I would take a fixed number of digits from the end of the caller’s number (six seemed reasonable and comparable to the FLOWERS example). Using mapping of digits to possible letters I would then generate all possible character permutations. Then, using a dictionary file containing approx. 400k English words, I would essentially compare each of the possible permutations to the dictionary to check whether any of my possible permutations were valid English words. If so, great – fingers crossed that there are four more. If there weren’t any then my idea was to reduce the six-character word to two three-character words, or a two-character word and a four-character word, etc.

However, a quick sanity check if this idea threw a few spanners:

* What to do with 0s and 1s
* Using my own number as an example (ends 707934), the likelihood of finding *any* valid words with those characters seemed slim to none.

However, for lack of any better ideas this is the idea that I went with.

The logic itself to derive all permutations of the characters proved too difficult / time consuming and so eventually I resorted to co-opting some code from StackOverflow. Further, due to time constraint I put the dictionary comparison to the end of the list and I never got around to it unfortunately (there are some elements in the deployment which allude to this but are ultimately unused).

In the end, I ended up with a solution that:

1. Generates all permutations based on the number / letter mapping
2. Takes five of those at random
3. Converts zeroes to ‘o’ and ones to ‘i’
4. Selects the first three for playback to the customer

As expected, this by and large results in character combinations that are most definitely *not* real words. However, you do sometimes get lucky with some and Polly pronounces them in such a way that *sounds* like a word or two.

## Custom Resource

The second area where I faced issues was with implementing a Custom Resource. These resources are some that I have come across before and was aware of their ability to invoke Lambda functions within a CloudFormation deployment but one that I have never actually developed or use.

The issue that I primarily faced was the fact that the resource always seemed to hang or failed deployment, creation, or deletion. Originally, I thought it was because of an issue with the deployment itself and so spent two or three hours waiting for the stack to stabilise and then delete.

In the end it turned out that it was because I wasn’t using cfnresponse to ‘talk back’ to CloudFormation and give status updates. However, on reading it seems like cfnresponse is only included in Python when you use zipped code, which I wasn’t. As such, I copied the cfnresponse.send function from the AWS documentation and included this in my Lambda code, which seemed to do the trick. However, I’m sure that there will be a better way to manage this and it’s mostly down to the ‘weird’ deployment method I’ve chosen.

## Create\_contact\_flow

The third major struggle I had was probably the most frustrating to get to the bottom of, but mercifully was the easiest to fix. Maybe it’s my reading, but the documentation for the Connect API is completely inadequate for explaining how to use this API action.

Looking at the documentation it clearly states that the content property should be a string. “Aha, obvious!” I thought… After skipping into Connect and exporting my contact flow to JSON, and a good couple of hours of bashing my head against escaping quotes and double-quotes and generally going round and round in circles later, I stumbled across the update\_contact\_flow\_content action which signposts the reader to “Example contact flow in Amazon Connect Flow language” where it is quite clearly stated that the recommendation is to use the DescribeContactFlow API to get the *actual* content for this API call property.

## Front-end Development

Probably needless to say after you’ve looked at the website, but a front-end developer I am not. Fortunately, in my team at work we do have a couple of ‘professional’ front-end devs and so I’ve had the good fortune to be able to largely rely on them for this. As a result, though, the front-end that I’ve come up with is very… basic. It does however do the job (although whether it does it well, I’m sure is debatable. With web development I mostly don’t even know what I don’t know in terms of the actual writing of code and best practice).

# Bad Practice / Shortcuts

Due to time constraints there were a few little shortcuts that I took, or things that I thought could be better – or as I prefer to think of them – features that didn’t get implemented.

* In the.bat files used to deploy resources there are several relatively hard coded values, such as for example the name of the S3 bucket to which resources should be uploaded. This could have mostly been avoided by demanding extra command line arguments but given that the S3 bucket would be created once and then sit there unchanged for most of eternity it seems more convenient to hard code it as much as possible – however obviously if you do change the bucket name then you need to remember to change your deployment scripts.

Further, in the case of the CloudFront .bat specifically the “aws cp…” command is very much so hardcoded, and needs to be manually changed depending on the environment into which you’re deploying… For some reason the aws cp command didn’t much seem to like taking a variable in the command arguments in the same way as the other commands did.

* Speaking of the deploy\_cloudfront.bat file, the “aws cp” command to get the contents of the src/web/ directory into the S3 bucket isn’t something that I would normally do or necessarily recommend either, because there are often files in this directory containing keys and such that are only appropriate to be visible in development environments (not to mention dev vs test vs prod configuration values). Further, putting your code into the bucket this way prevents the ability to run a proper development and deployment pipeline which would ideally include phase-gates including testing and manual interventions at certain stages if required.
* Something that you would do well to evaluate down the line would be the resource utilisation of the resource such as Lambda functions and DynamoDB tables. Off the top of my head I think Lambda function allocated memory is at its lowest by default but if not then it would potentially be worth assessing this over time and reducing if possible, to save cost. Similarly, the DynamoDB tables are set to ON\_DEMAND pricing. With very low throughput such as in a case as this, that’s the cheapest option. However, in a production environment depending on the number of reads and writes it may be cheaper to provision a specific amount of capacity.
* API Gateway does not have any authorisation on it. Typically, in a production environment (at least the ones that I work in) any websites that we build will be protected by some authentication (most commonly Cognito). One of the benefits of this is that API Gateway allows the use of a Cognito authoriser to ensure that only active Cognito token holders can access the API. Despite being highly scalable, a lack of authorisation could be abused by demanding a large number of read requests from DynamoDB, therefore incurring significant cost (or if provisioned capacity was used then this could cause genuine users’ requests to be throttled). Alternatively, I imagine you could use IAM authorisation for your API and failing that you could maybe(?) use CORS config on your API Gateway to restrict API access to only your website, whereas I have allowed all for CORS.
* Ditto, the front-end doesn’t have authorisation on it. Depends on the workload as to whether or not required.
* Additionally, with more time it would have been nice to be able to claim a domain for the website for a more ‘user-friendly’ URL, which then gives the ability to claim a certificate in ACM which enforces TLSv1.2 and v1.3
* One that may or may not be accepted as a bad practice is that I tend to use a lot of “try … catch” blocks which catch *all* exceptions. I’m generally aware that this is not overly useful, but due to my lack of experience I don’t really know a better alternative at this time (beyond catching all exceptions individually which seems overly verbose to me), although I am trying to get a bit more specific with for example the “except KeyError” blocks when retrieving environment variables and data from the event object.
* Lack of unit / integration testing. Similarly, to the try / catch, this is something that I know that I don’t know. These are not included simply because I don’t know how to implement them (although something that we’re working on internally and are going to start trying to learn and get better at!).
* Inflexible website JavaScript. Partly again due to lack of time and partly due to not knowing a much better way of managing it, the API URL is hardcoded into index.js. This means that whenever you deploy to an environment you either need to manually amend the file to include the appropriate URL or you have to maintain multiple copies of the file – one for each environment, or move the URLs and other environment-specific data into a config file of sorts. However, still managing to get the correct config file into the correct build is something that I haven’t figured out a better way of doing yet.
* Data retention policies. The DynamoDB table is storing PII and therefore I definitely should have included a TTL attribute to manage expiration of the data after a sensible period of time. I did try to include lifecycle policies on the S3 buckets, but I’m aware that I also missed CloudWatch Log Stream retentions on most Lambda functions.

# Additions / Improvements Given Extra Time

There are many and there’s a lot of cross-over with the above section about bad practice. Some of these include:

* Adding the dictionary comparison to my vanity number generator to try and maximise the generation of valid English words and automating that deployment (and potentially processing the dictionary file to pull out the 6, 4, 3, and 2 letter words to reduce overall file size).
* Authorisation on the website (because why not?)
* Authentication on the API Gateway
* Generally putting correct any lapses in data retention policies (DDB TTL, S3 lifecycle policies, CloudWatch Log Stream retention)
* Support for pagination on the Scan operation for the getRecords API call
* Generally better error handling throughout (in Lambda functions, within the contact flow, and within the front-end)
* Added a Lambda permission to the generateVanityNumbers Lambda function invoked within Connect to prevent the assessor / user from having to manually allow Connect to invoke the function
* Create a single unifying .bat file which allowed command line arguments to be set and then subsequently trigger the other .bat files based on a ‘module’ argument. This would allow the entire solution to be deployed through a single CLI command e.g.:

*./deploy.bat {region} {env} {instanceArn} {module}*

*..*

*If module == all; then*

*//deploy back-end*

*//deploy front-end*

*//deploy api*

*Else if module == back-end; then*

*//deploy back-end*

*Else If module == front-end; then*

*//deploy front-end*

*Else If module == etc.*

* Would have liked to add support into the contact flow for cancelling of user input when calling anonymously (and associated error handling and parsing in the Lambda function)
* Would have liked to have spent the time to bring the SSML generation out of Connect and into the Lambda function to better control the response played back within Connect (and simplify the configuration of the Connect play prompt block), and use more SSML tags to improve the way in which the results were read back to the caller.
* I didn’t like the way that I implemented the Lambda function to put the contact flow into Connect. Keeping the JSON document for the flow as a single line string and then using ‘“..” + LambdaArnVariable + “..”’ felt really bad, but this seemed like the easier / fastest thing to do. I would have liked to find / figure a better way of doing this.
* Would have liked to spend more time finding a better solution for cfnresponse, if one exists.
* One of the things that I *really* hoped I had time for (and started but found it was too late / taking too long) was trying to use SAM to deploy the API Gateway. Using native CloudFormation and a Swagger template like I have done is really verbose and I find *really* difficult to get correct when doing it from scratch, and from a quick scan of the documentation it looks like using SAM for an API Gateway in comparison would be a dream. However, as I say, it was taking me too long to get up to speed with SAM so I ditched it in favour of something I knew I could get running quickly (because I was doing this on Saturday evening or Sunday morning).

# Productionisation

In general, a lot of the things that I would talk about in this section I think I have already spoken about in regard to where I have taken shortcuts or ran out of time.

There are however a few dedicated comments to make on the subject of system productionisation… Which are conveniently articulated in the 5 Pillars of the AWS Well-Architected Framework:

1. Consideration must be given to the deployment and ongoing extension and maintenance of the deployment. As a result, in order to be a production ready deployment I would insist on the improvement of the deployment itself – streamlining and removing the hard-coding, and generally making amendments or additions that would maximise the possibility of a successful deployment (i.e. making the deployment as simple and modular as possible) and minimise the risk and impact in the event that it all goes wrong (i.e. testing code, version control, etc.)
2. Security: Given the two primary ‘vectors’ of attack within our solution there are a few things that can be done or considered:
   1. For the IVR: Obviously, there’s not *too* much that can be done here in terms of pro-active prevention or indeed of ways of attacking your solution. In my view the biggest key is having the ability to detect that something is happening, and in that respect having alerting on CloudWatch metrics for Concurrent Calls is key. If a malicious actor flooded your contact centre with calls and breached your concurrency quota then that obviously affects all of your other DDIs and queues, therefore the ability to alert on high concurrency is valuable. Following this, the ability to block certain numbers within the flow (via an attribute branch) might also be valuable.
   2. For the website and API: in production you would obviously consider the user-base of the website. If not an actually public website, then you could look to limit access to the website itself full stop through a service such as WAF & Shield which allows IP based or geographic (I think off the top of my head?) blocking; then, authentication on the website itself *and* the API (as discussed) would also be effective (and also potentially in the Lambda function if possible).

In terms of general AWS, having a properly configured CloudTrail application is obviously important as well to keep records of activities, and enabling encryption wherever possible is also super low overhead and also important. Retention policies are also likewise important if for no reason beyond they limit the amount of data available to be stolen or interfered with.

I’ll call it out again but not having sensitive config values in your website is also important.

1. Reliability: There’s not too much here to talk in terms of maintenance as all services in this solution are serverless.

However, this doesn’t mean we’re completely hands-free: limits on the maximum number of concurrent Lambda invocations, and DynamoDB read / write capacity are things which should be considered (although in my experience the limits are so high that for most applications it isn’t necessary). If dedicated capacity is provisioned for the DynamoDB tables then auto-scaling might be useful to allow for bursts in traffic.

1. Performance Optimisation: Not something that I feel I’m qualified to speak about or recommend. Obviously however I do appreciate the benefits (reduced Lambda function invocation time, reduced cost of each service, etc.). In this particular exercise I can’t think of massive areas for there to be performance improvement except for perhaps coming up with a more efficient way of executing the generateVanityNumbers function.
2. Cost Optimisation: Touched on before but worth mentioning again, services such as Lambda are so cheap that on the scale I’m used to (i.e. largely within Contact Flows and a little bit in ‘internally’ used web applications) there’s not too much to worry about. DynamoDB on the other hand can be relatively expensive if miss-sized, which is the reason that I tend to go with ON\_DEMAND pricing, however there is obviously a point where it tilts in the other direction. Therefore, the key is to either go ON\_DEMAND or over-provision and then later assess and re-do if required.

As a general rule of thumb, we find Connect to be 95%+ of the end-of-month bill and therefore the best of use time is finding ways to reduce average handling time, queue time, and repeat callers (although none of these are necessarily an issue in this application). In services where data storage is a factor such as DynamoDB and S3, then retention policies again come in useful for reducing the amount of data stored.