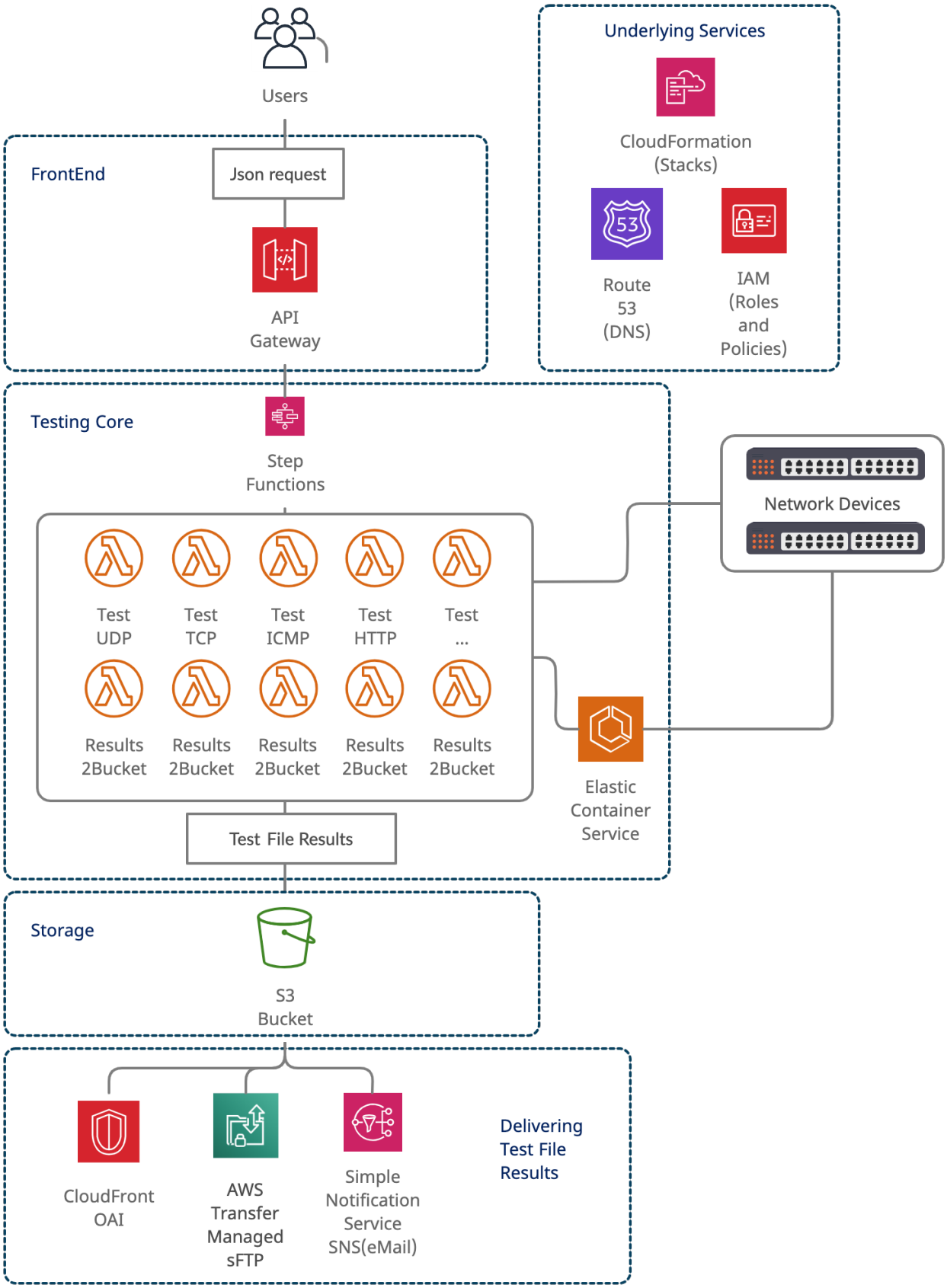
High level diagram

Here is the high level schema of the entire solution



Implementation

Frontend:

- **API Gateway + Web Application Firewall (AWS WAF):**
  - Managing the API request from customers
  - AWS WAS enabled protecting APIs from attacks
  - Calling the step Function Machine and passing them all the necessary parameters

Testing Core / Backend

Name	Technology	Function	Interacts with
StepFunctionMachine	StepFunctions	Orchestrating the Lambda execution and passing the parameters	API Gateway, Lambda functions
Lambda Functions	Lambda	Running Network test on isolated containers (Custom docker and Lambda managed), and uploading result files to	Amazon Elastic Container Service , S3 Bucket
TesterRegistry	Amazon Elastic Container Service	Storing custom Docker images and Run custom Docker containers	Lambda functions, S3 Bucket
Network Devices	-	The devices from the customer to test	

Testing Core / Lambda functions

Please find a description of the implemented lambda functions:

	Technology	Language	Image
TestUDP	Custom Docker images hosted on Amazon ECR Registry	Python 3.8	net-tester-python
TestTCP	Lambda	Python 3.8	net-tester-python
TestICMP	Custom Docker images hosted on Amazon ECR Registry	Python 3.8	Lambda
TestHTTP	Lambda	Python 3.8	Lambda
Upload Results	Lambda	Python 3.8	Lambda

Additional notes:

- Due to network tests, it makes sense to add the option of custom Docker images. I think makes sense due more than probably some network tools are needed, such as nmap, apache-tools, netcat, wireshark, etc…

- Ensured same python version running on lambda than in custom Docker container
- The function uploading results file to bucked function is splitted into a specific lambda function for better fine granting permissions and reduce exposition/attach surface

Storage

- **Amazon S3 Bucket:**
  - Hosting all the resulting files from the executed tests
  - Interacts with containers (Custom docker and Lambda managed), Lambda Functions and delivering services (CloudFront OAI, Amazon Transfer sFTP and Simple Notification Service)

Delivering services

- **CloudFront OAI:** Making HTTP and API accessible test result files
- **Amazon Transfer sFTP:** Storing and delivering the result files
- **Simple Notification Service:** Sending notification and eMails with the result files

Additional notes:

- All listed services interact with Lambda Functions and S3 Bucket

Underlying services

- **CloudFormation:** creating and keeping the services stacks
- **IAM:** enforcing roles and policies in all the other services
- **Route53:** managing custom DNS entries references for all the services

Additional notes:

- All listed services interact with the other services due to permissions, roles and Stack-Managed.

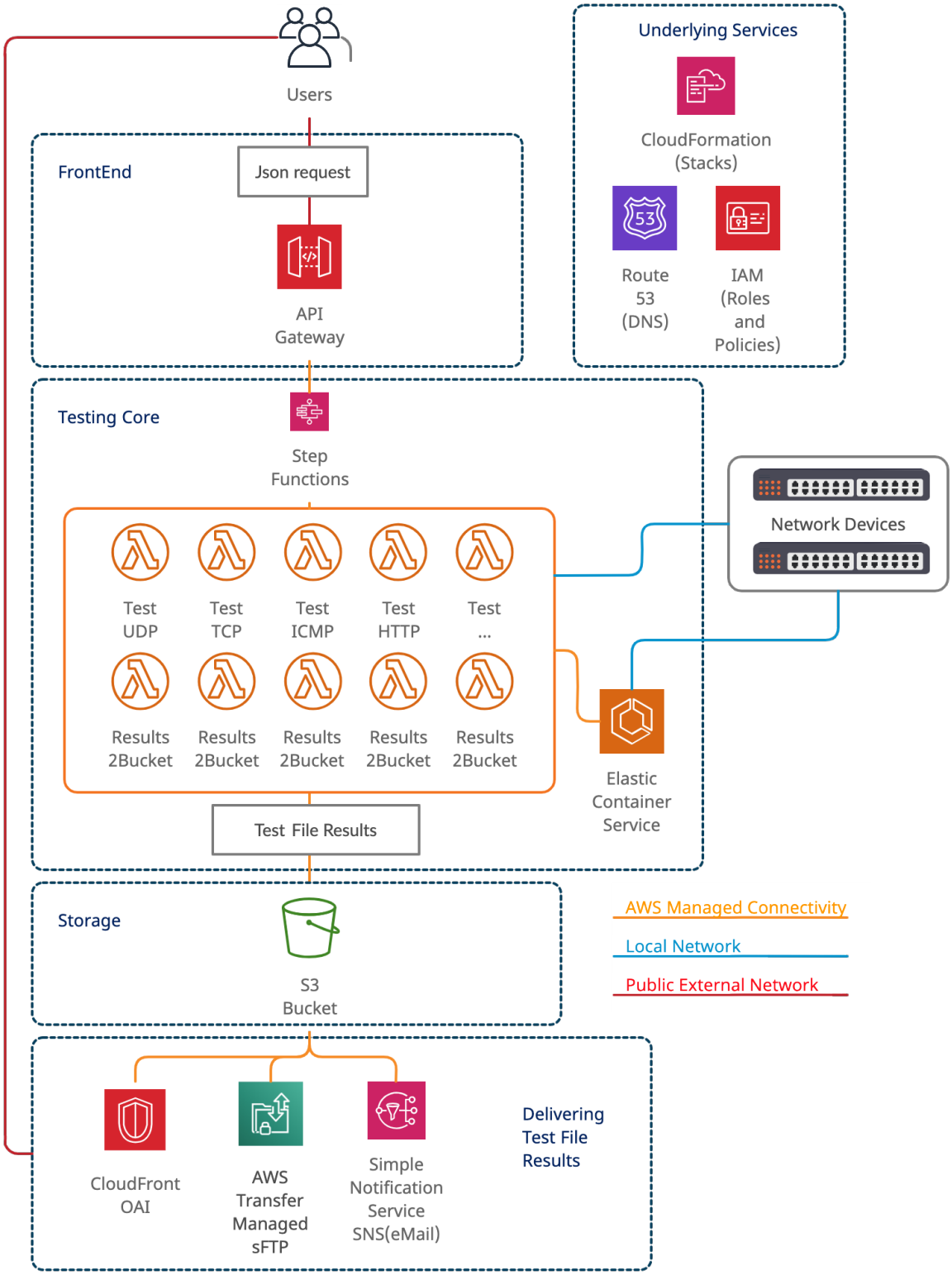
Connectivity

The solution will have 3 types of networks:

- **Public/external networks** (in red on diagram): Exposing the API gateway for API request and CloudFrontOAI offering result files.
- **Internal network** (in blue on Diagram): connecting Network Devices with Elastic Container Service and Lambda functions
- **AWS Managed network** (orange in Diagram): connecting all other devices.

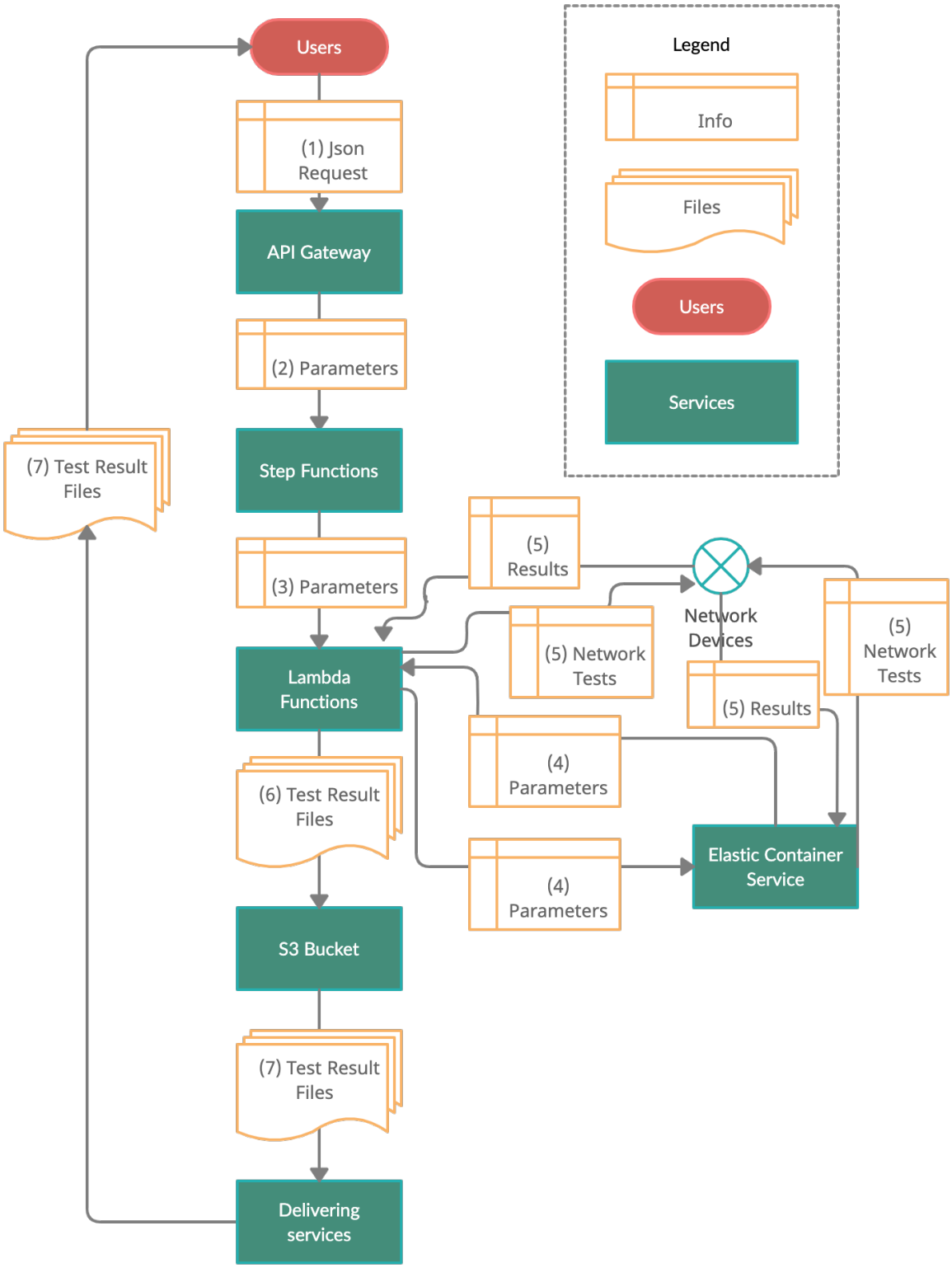
Note:

- I'm supposing direct connectivity between AWS Lamba and Customer devices, avoinding references to additional network services like VPC.


Users
FrontEnd
Json request
API Gateway
Underlying Services
CloudFormation (Stacks)
Route 53 (DNS)
IAM (Roles and Policies)
Testing Core
Step Functions
Test UDP
Test TCP
Test ICMP
Test HTTP
Test ...
Results 2Bucket
Results 2Bucket
Results 2Bucket
Results 2Bucket
Results 2Bucket
Test File Results
Network Devices
Elastic Container Service
Storage
S3 Bucket
AWS Managed Connectivity
Local Network
Public External Network
CloudFront OAI
AWS Transfer Managed sFTP
Simple Notification Service SNS(eMail)
Delivering Test File Results

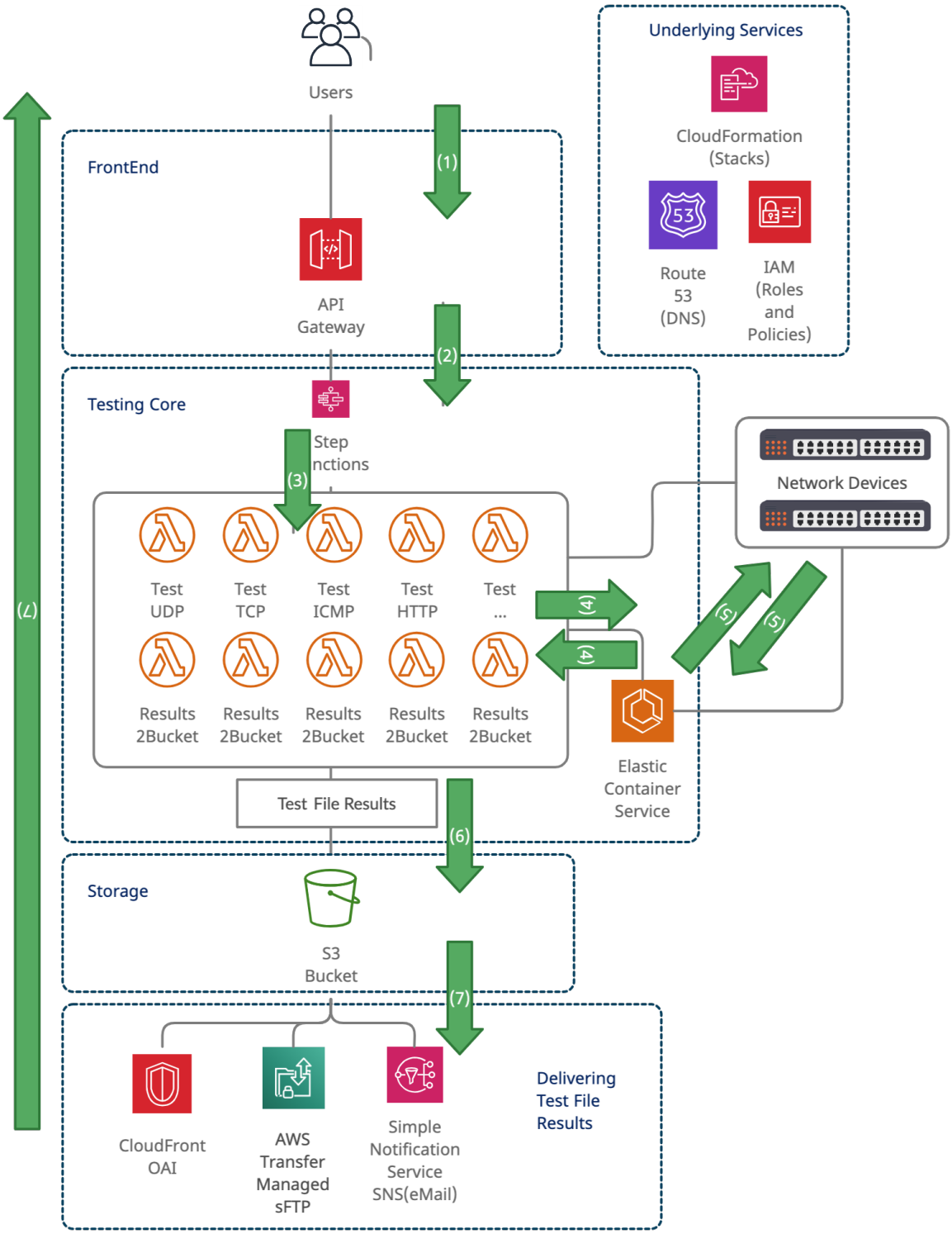
Dataflow diagram

Please find the schema of the data flow

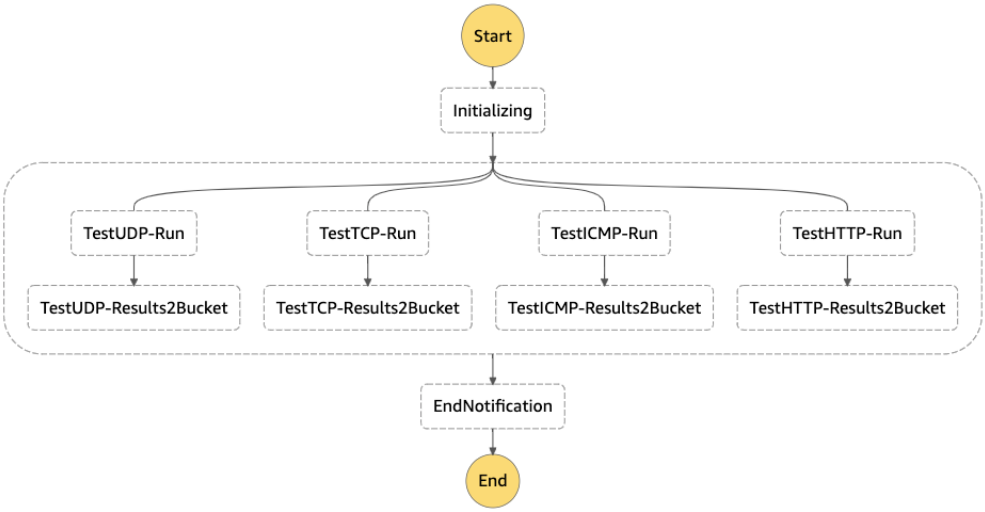


Workflow diagram

Workflow inside the solution will step as follows (maked in green on the diagram)



1. The user calls the Frontend (API Gateway) using a Json request with some specific parameters (DeviceID, TestID, Delivery method, etc…)
2. API gateway manages the Json parameters and triggers function step machine with these parameters
3. The function step machine calls each individual lambda function in parallel with the specific parameters (Device ID, test, etc…) in parallel



4. Lambda functions starts the containers (custom Docker form ECS or Lambda-managed)
5. Containers run the test against the customer Network Devices
6. Containers generates a result file and it uploads the results on the S3 bucket
7. Once all the tests are finished, results are accessible from the bucket on the requested way (sFTP, HTTP, email, etc…)

Code files

Please find a short reference of each code file

Notes:

- Individual commands each file is on the readme and the 00-DeployingSolution.sh script.
- Deployment of some services has been splitted on different files and CloudFormation Stacks to simplify test/debugging.
- Some names differ from the final solution, and some changes are missing, and will be listed on the next .

Filename	Filetype	Description
10-Stack-ECRRegistry.yaml	CloudFormation Stack Template	Contains the definition of the Registry hosting custom Docker images for testing
Docker	Dockerfile	Necessary files to create the custom docker image net-tester-python
21-Stack-BucketResults_sFTP.yaml	CloudFormation Stack Template	Contains the definition of: -The Bucket that will host the test results -policies and roles -SFTP Server -sFTP Server test “demouser”

22-Policy-Bucket-makePublic.json	Bucker policy	Policy for the bucket making some files public based on a tag and on a customer trusted IP range
LambdaCustomDocker	sam and config files	Deploying Lambda functions linked to Custom and lambda manage containers
40-StepFunctions-Pipeline.json	Step Functions definition	Setting the pipeline that it runs all the steps, only one container per test branch
40-StepFunctions-Pipeline_Results2Buckethierarchy	Step Functions definition	Setting the pipeline that it runs all the steps, splitting store on another lambda function
41-LambdaFunction-StepStates.py	Python	Python function (included on custom docker files as well) running the test and uploading files to Results Bucked
Readme.md	Readme markdown file	Details about how to deploy the solution from the code files. Some command line steps, like docker image generation or sns set up are detailed there

Additional Notes:

- Some values (like arn references, Images URS or bucket names) are not parameterized and maybe needs to be manually updated
- Public setting for Bucket and Mae public policy cames from some testing and workaround I would like to keep. Final approach will use CloudFront OAI

Pending to be implemented / differences from the attached code

The following features are pending to be implemented from the attached code

- API Gateway and AWS WAF
- Cloud OAI
- Split Lambda functions (results2bucked from the test ones). StepFunctions machine pipeline is done.

Future improvements and possible changes

Please find some design decisions taken:

- **Code changes:** set up an automatic trigger for image regeneration (CI/CD) on each test code/Dockerfile update, updating Docker “image-name:latest” tag and regenerating the associated lambda functions. This is not actually but could be easily integrated with AWS CodeBuild.
  https://aws.amazon.com/codebuild/features/?nc=sn&loc=2

- **Filename:** To avoid possible overrides, depending of the number of concurrent customers using the platform, maybe makes sense to insert on the filename additional unique Id's like "context.aws_request_id" (Id from each lambda call) or Docker container container_id
  https://docs.aws.amazon.com/lambda/latest/dg/python-context.html
- **HTTP Access:** switch CloudFront OAI for PreSigned URLs. I preferred CloudFront OAI to preserve more control and information about the published content.
  https://docs.aws.amazon.com/AmazonS3/latest/dev/ShareObjectPreSignedURL.html
- **Lambda Functions:** switch StepFunctions for Amazon SNS to notify/run Lambda Functions. I preferred Step Functions in order to be able to provide retry, timeout and exception catch on failed test, and gather logging.
  https://docs.aws.amazon.com/sns/latest/dg/welcome.html
  https://docs.aws.amazon.com/step-functions/latest/dg/concepts-error-handling.html
- **Quota per user:** maybe for security reasons, billing or usage requirements these need to be implemented.
  https://docs.aws.amazon.com/general/latest/gr/aws-service-information.html
- **Delegated/self managed customer permissions:** depending of the customer specific, makes sense to delegate some roles or access.
  https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles_create_policy-examples.html

I personally discarded the following options to not be part of the requirements:

- **Custom TestLoader:** an interface/API where some advanced users could load a json file with the all the test details (Dockerfile/docker image, code to run, etc…) and then that automatically generates the test (Image, Lambda function, expands API, Setup Functions, generated necessary roles, etc…). My main concern has been security issues (function security issues, managing Docker attestator, exposure, security surface for some roles, and maintaining and managing roles for functions generating automatically other functions on the fly, etc…
- **S3 Bucket:** archive/delete passed results to a different storage class/bucked using a Lyfecycle policies
  https://docs.aws.amazon.com/AmazonS3/latest/user-guide/create-lifecycle.html
- **Provide IAM, Directory and API Gateway integration**

Design approach

The described design takes advantage of the cloud and tries use only necessary services using standardized tools with a minimal operational cost as all of them are self-managed and payed-per-use on AWS. I avoided always-running tools suck a Kubernetes clusters or custom virtual machine using alternatives such as ECR.
Every test is end-to-end automatic, due all the tests are already deployed, containers start and stop automatically and the generated results are send automatically. A long timestamp with the TestID and the Device on the filename ensures integrity and no conflicts on similar tests.
Test performance has been optimized in 4 points:

- Containerized functions using the minim resources (avoiding heavy virtual machines) with the minimal image and using lambda when it's possible
- Minimum overhead: taking advantage of containerization avoiding no needed libraries on functions or an entire VM to start up
- Parallel execution: Isolation between test avoiding blockers
- Network optimization: all services running on the same cloud zone, with local connectivity with Network devices

Thanks to CloudFormation, all the elements defined on the stack are consistent. On top of that, retries and timeouts are set on each test branch in order to assure a successful behavior. An API versioning (/api/v1/old_call, /api/v2/newcall) it ensures old/legacy API calls still running until they are properly deprecated.
Using Amazon S3, so all the information is redounded with more than 99% of assurance.
I also proposed some additional changes on the previous section, like code changes integration with AWS CodeBuild, although from my point of view is not strictly necessary, could improve redeploys on a CI/CD workflow.