# EinSight Multi-API Architecture Design Doc

## Glossary/Quick Links

|  |  |
| --- | --- |
| **Term** | **Definition/Link** |
| SIM | <https://issues.amazon.com/issues/spade-task-587> |
| Meeting Minutes | [EinSight 2.0 Backend Upgrade - Q4](https://quip-amazon.com/IMjaAUZxZBAo) |
| Cloudfront | CDN to serve static and dynamic content to clients. For EinSight it's essentially the initial entry point for the domain that is used to access EinSight |
| CDK | AWS Library that lets you create AWS resources using code rather than manually creating in AWS console/UI. This library converts your CDK code into Cloudformation templates (see below). |
| Cloudformation | Used to automatically manage AWS resources based on Cloudformation template. Cloudformation handles deployments of code and automatically creates/updates/deletes AWS resources |
| API Gateway | One way of creating and hosting APIs. Handles routing/load balancing |
| Application Load Balancer | Similar to API Gateway but with less convenient integrations |

## Problem Statement

*EinSight lacks a scalable backend architecture for defining additional APIs and API endpoints to add functionality for new and existing applications.*

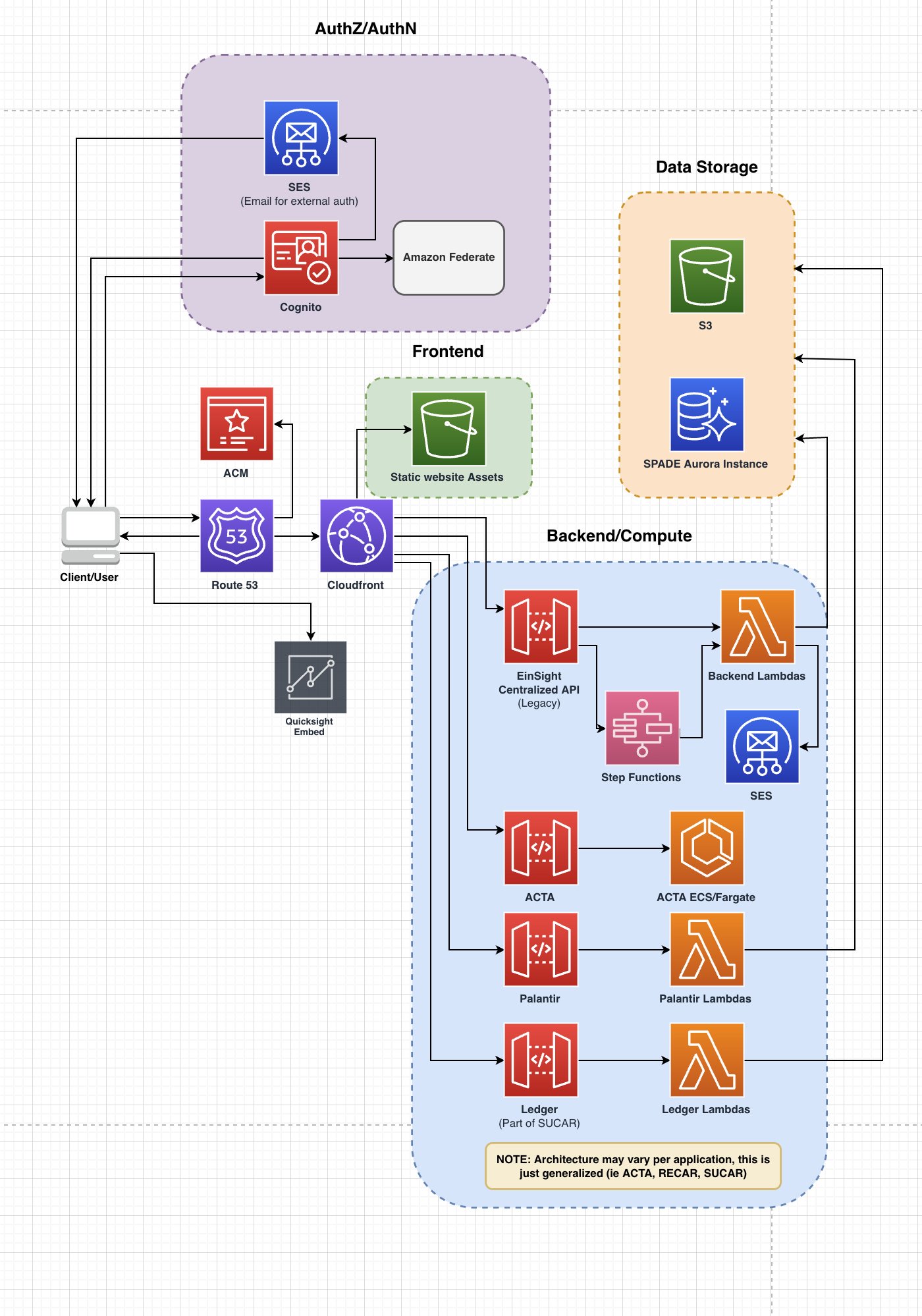
## Introduction

EinSight is a platform that hosts and ties in tools/functionalities across GES, and most of EinSight's technical architecture was also similar, tying everything to a single resource, in this case, one single API Gateway. While we should still strive to deliver a single, unified experience to our customers, from the technical side we are running into issues/blockers using a single API gateway.

We are starting to reach AWS Cloudformation limits for how many API endpoints we can have in a single API Gateway (about 500), which for EinSight will make developing new tools and adding features to existing tools difficult, if not impossible in certain cases. In addition, maintaining a single API Gateway endpoint also hosts other potential issues, one good example being blast radius, where changes or issues with one tool could affect every tool in EinSight.

The most significant risk currently is that every time a Cloudformation resource limit is reached, this blocks all other changes/deployments to the stack being deployed. Since a majority of AWS resources are hosted in a single stack and the backend pipelines deploy multiple stacks at once, a single stack with a deployment issue will stop all code changes from being deployed due to Cloudformation rollbacks. This will be disruptive for new feature development and bug fixes to the platform, and will essentially freeze the platform from any changes.

## Current State



[Design Inspector Link](https://design-inspector.a2z.com/#IEinSight Design Architecture)

The current EinSight architecture consists of a Cloudfront Distrubution that serves both the static content for the frontend and the dynamic content for the backend API.

EinSight originally started out with a single API gateway (EinSight Centralized API) that would be used for all tools. The endpoints in this API are often split by application, though a lot of the applications use common APIs methods (get, update, delete) that are used across multiple tools.

There are the usual issues of this approach, including blast radius, lack of proper regression testing of common API endpoints, etc. But the main issue is that Cloudformation only allows 500 resources per stack, and API Gateway only allows 300 endpoints per API, so we cannot continue adding new endpoints without hitting a soft limit.

The immediate solution has been to create independent API gateways and add them as an origin in the Cloudfront distribution, solving a lot of the issues above. However this approach is a band-aid fix that will only support up to 24 independent APIs due to Cloudfront’s 25 origin limit. Note that one origin is reserved for the static website S3 bucket.

## Requirements

### Functional

* Resolve resource limits in Cloudformation and AWS services (500 resources per stack, 300 endpoints per API gateway)
* System should be able to scale to a much larger set of APIs to scale with the incoming EinSight tools and services
  + What happens if we have 25 APIs, 50 APIs, etc
* Frontend system should be able to know how to reach the backend API endpoint in case API urls change
  + Either maintain mapping of API gateway URLs or use custom domains
* System must be extensible and configurable in CDK so new APIs/products can easily add their API gateway and make it callable from the frontend client
  + This saves the work of manually creating and configuring an API gateway resource

### Non-Functional

* Latency should not be dramatically altered
* System should have capacity to expand to other regions (namely EU regions)

## High-level Design (WIP)

Currently leaning towards Independent API gateways: [Independent API Gateways](https://quip-amazon.com/favkAtUdnxwm#temp:C:INf0babb3170f7e478abf3e8d35f) but need more discussions on how to manage and maintain AWS accounts among a smaller team.

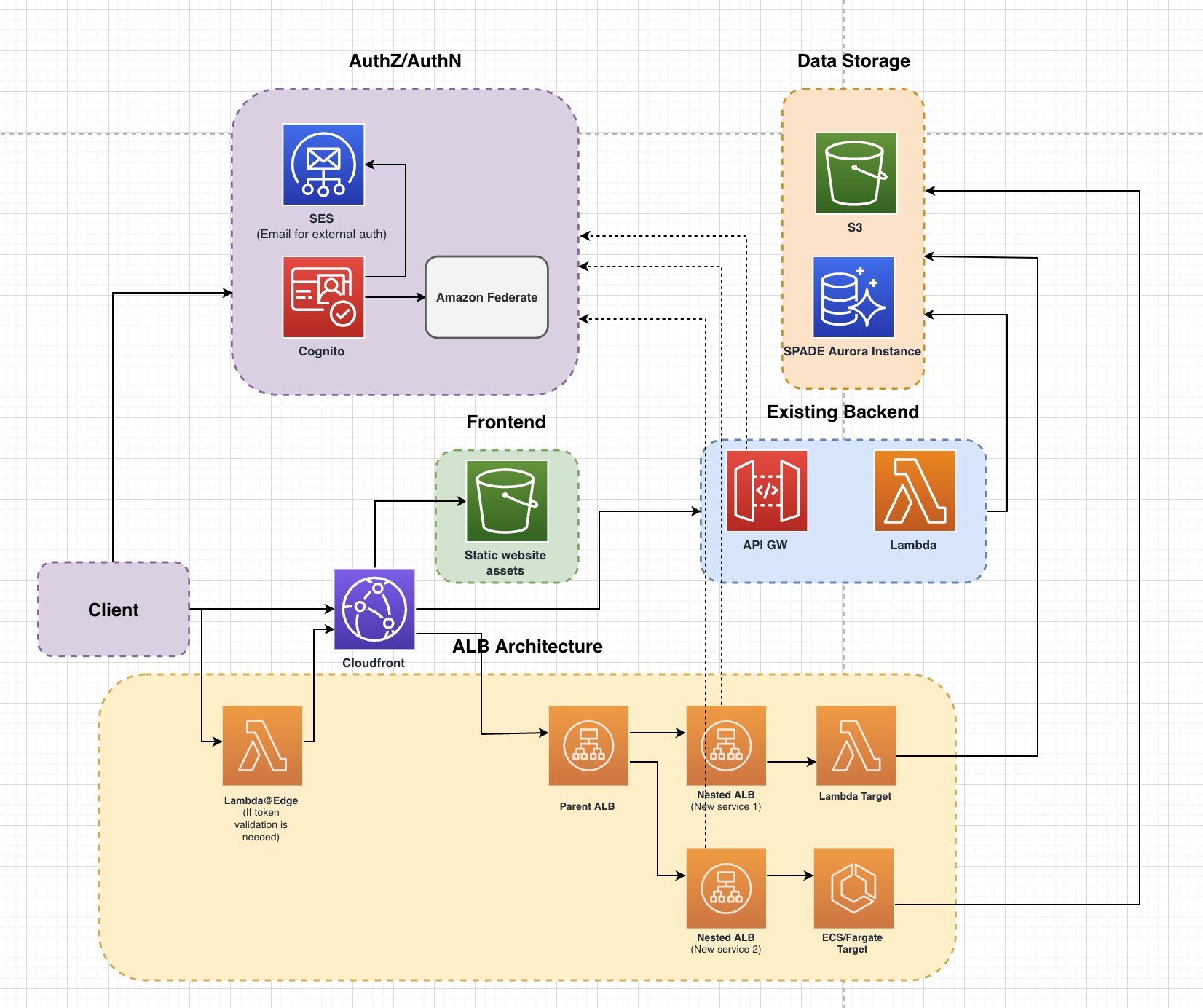
## Alternatives Considered

### Nested ALB Solution

This solution assumes that we want to keep the same domain as the entry point for these applications rather than separate domains per application.

Now that Application Load Balancers support Lambda functions as targets, the current proposal is to use an Application Load Balancer to serve as the entry point for all of EinSight’s new backend services moving forward. ALBs are generally cheaper and much more scalable than API Gateway, though ALBs need more configuration and don’t have as many convenient service integrations. Since ALBs have a soft limit of 50 listeners, we will create ALB as an entry point (and add that to the Cloudfront distribution), and then each one of these 50 listeners can point to an application-level ALB that handles the routing the API request from the client to the correct Lambda/EC2/etc target.

|  |  |
| --- | --- |
| **Pros** | **Cons** |
| Backward compatible migration for existing endpoints under the single API Gateway with path /api/\* | Soft limit of 50 ALBs in a region (account wide) and 50 listeners per ALB |
| Maintain the single domain name entry which can avoid a lot of issues with browsers’ CORS policy | ALB can not set an API Gateway as a listener |
| Scalable to many services due to ability to nest ALBs | Less intuitive to setup listeners, you'll have to manuall setup HTTPS listener with path-based rule |
| Authentication/token can be standardized via lambda@edge |  |
|  |  |



[Design Link](https://design-inspector.a2z.com/#IEinSight Design Architecture)

An important consideration to note here is that this architecture is an extension of the existing system, and should serve as the default for new applications moving forward. This will introduce minimal change/disruption to the existing architecture (and should allow for deployments with little downtime due to smaller blast radius).

The main reason this approach is recommend is that nearly all the components of this approach are officially supported by AWS (or at least as decent documentation), so running into sudden issues during development is less likely to occur. ALB is also a generally cheaper alternative to API Gateway and has decent feature parity, although it lacks in some service integrations. The main downside is some added complexity upfront for implementation but the ability to scale to a good number of API services while having the APIs exposed under one endpoint (which saves work needing to endpoint/domain mapping and CORs for each service) is worth it.

### Cloudfront with Redirect via Cloudfront Functions/Lambda@Edge

This solution keeps the existing Cloudfront distribution but with the addition of Cloudfront Functions/Lambda@Edge which can intercept the request from the client and re-route/redirect the request to another Cloudfront distribution/API gateway URL by sending a custom HTTP response (307/308). Redirection can be used instead of adding origins to Cloudfront to overcome the origin limit, but comes at a cost of an additional client network request to the redirect resource.

There are some differences between [Cloudfront Functions and Lambda@Edge](https://aws.amazon.com/blogs/aws/introducing-cloudfront-functions-run-your-code-at-the-edge-with-low-latency-at-any-scale/) but they should be functionallity similar in our use-case with the exception of runtime (1ms vs 5+ seconds).

[Example Cloudfront Function code for redirection/CORs Headers](https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/functions-example-code.html)

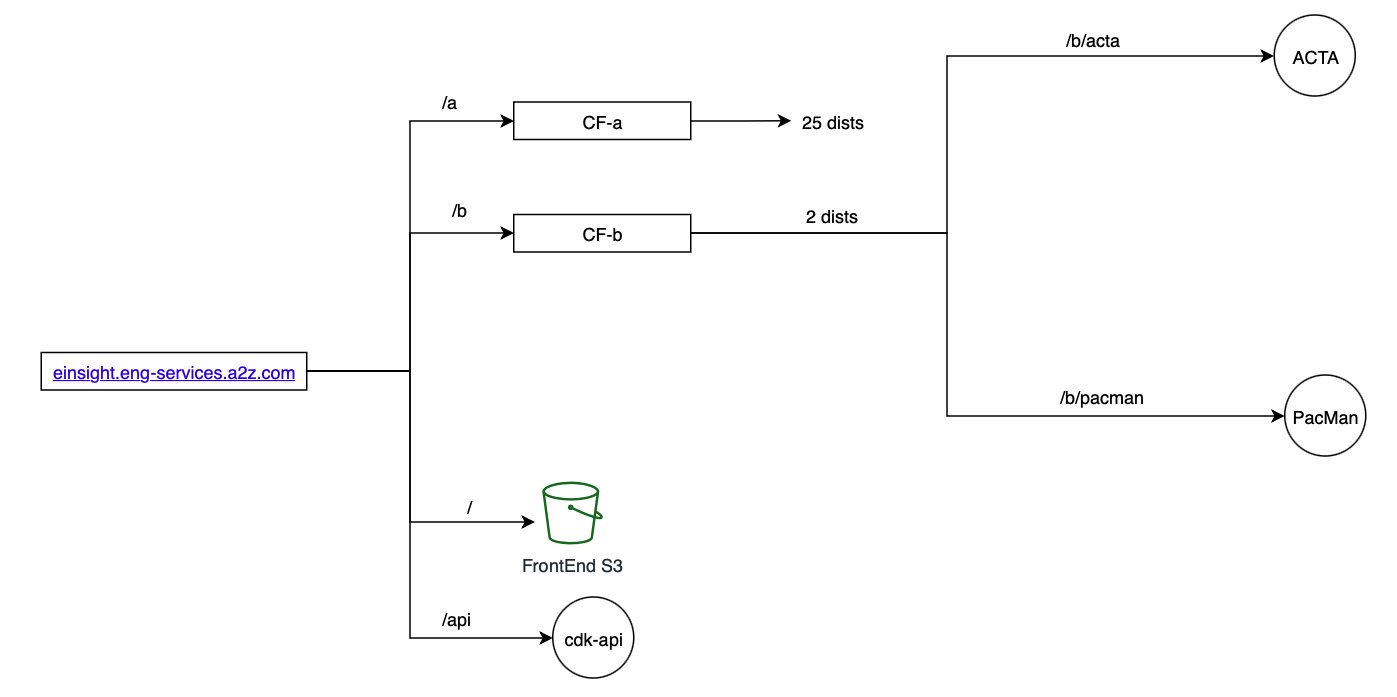
|  |  |
| --- | --- |
| **Pros** | **Cons** |
| Backward compatible migration for existing endpoints under the single API Gateway with path /api/\* | Introduces latency as the client needs to make a new request to the respective Cloudfront/API Gateway endpoint after initially connecting to the parent Cloudfront |
| Frontend client won't need to store/retrieve endpoints for other APIs since the domain entry point is still the same as before | Requires mapping/redirection logic that needs to be updated in the Cloudfront Function/Lambda@Edge to determine which API endpoint to redirect the request to |
|  | Still need to configure CORS because the Cloudfront here redirects API requests to a different domain |
|  |  |
|  |  |

### 2 Layer Nested Cloudfront

Note: This assumes that a Cloudfront distribution origin can set a custom origin that points to another Cloudfront distribution (**which is a big if currently**).

In this approach, a single API gateway endpoint is mapped to a single Cloudfront origin. Using one Cloudfront as the entry point for a custom domain, this [gives us 25 origins per distribution](https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/cloudfront-limits.html). The current Cloudfront distribution already reserves 3 origins: one for the static website, one for the current backend API, and one for the ACTA API Gateway. This means that we are restricted to 22 origins, or 22 API gateways.

But if we pointed a Cloudfront origin to another nested Cloudfront origin, we could have 22 Cloudfront distributions instead, each which can support 25 API endpoints. This theoretically extends our limit of 23 APIs to potentially 550 APIs (22 distribution x 25 origins). I have ran into issues implementing this solution in practice, as Cloudfront trying reach another Cloudfront is resulting in 403 errors.



[Design Inspector Link](https://design-inspector.a2z.com/?#IEinSight)

|  |  |
| --- | --- |
| **Pros** | **Cons** |
| Backward compatible migration for existing endpoints under the single API Gateway with path /api/\* | **Adding a Cloudfront distribution as an origin to another Cloudfront may not be functionally possible and result in difficult to debug 403 errors** |
| Maintain the single domain name entry which can avoid a lot of issues with browsers’ CORS policy | Implementing API-level will becomes even more complex if not impossible with two levels of Cloudfront |
| Scale up to hundreds of services with each service has it own API Gateway. Can support 500+ services. |  |
| Authentication setup and details can be standardized for APIs |  |
|  |  |

### Independent API Gateways

This solution is similar to the Cloudfront with redirect though it skips the Cloudfront entirely and instead of redirecting, the client should know which endpoint to make the proper API request to. The only restriction is to ideally keep the API Gateway in it’s own Cloudformation stack and ensure that it is accessible from the client. Then the main task here is to determine how the client will get the endpoint to the API depending on the service and the stage.

The major challenge here to solve is to either prevent API Gateway URLs from changing between deployments (by setting a custom domain for example) or creating a dynamic data store of endpoints for each service and each stage (RECAR on beta, ACTA on prod, etc).

|  |  |
| --- | --- |
| **Pros** | **Cons** |
| Fully de-coupled solution with very little service level restrictions | Need to tighten CDK reviews and ensure that API GW best practices are being met |
| Can easily be used cross-account if needed to | No centralization/re-use mandated, so common updates to all API Gateways need to be coordinated independently |
| Gives a lot of independence for API deployment/hosting | If client does not have API endpoints built in, then additional network calls need to be made |
|  | APIs are independent in this case, so we will need to configure auth/CORs for each API |
|  |  |

## Scaling Considerations

<Define security considerations (how will authentication work, can Cloudfront call Cloudfront?)>

<Define operational concern (scaling to regions, fallback needed?)>

<Define scaling concern (how many tools does this support, how many users can this support)>

<Rough estimate of IMR/costs>

## Risks

* Latency?
* Caching issues?
* IMR often considered after the fact

## Development/Rollout Plan

This task plan assumes that we are continuing to use the Cloudfront distribution from StaticWebsite construct in the Frontend CDK package. Note that the StaticWebsite construct is deprecated and additional work is needed if we need to migrate away from it. However for the development for this project, the main concern is significantly increased onboarding time if we assign this task to someone who is not familiar with CDK or BonesConstructs. This is because documentation and support for the deprecated resource is sparse.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E | F | G | H |
| 1 |  |  |  |  |  |  | **P1** | **P2** |
| 2 | **No** | **Feature/Dev Task** | **Details** | **DEV Sprint Classification** | **Status** | **Comments/Questions** | **Estimate (Days)** |  |
| 3 | 1 | Design Doc Review | Create design doc with basic proposal and task breakdown |  | Done |  | 0.5 |  |
| 4 |  |  | Finalize design doc with alternatives, risks, scaliblity considerations, etc |  | IN PROGRESS |  | 2 |  |
| 5 |  |  | Design doc review and approval |  |  |  | 1 |  |
| 6 |  |  | Create plans/timelines for potentially blocking beta pipelines and downtime for prod deployment |  | IN PROGRESS | This task is important, once we roll out a CDK change like this one, it'll be a bit more difficult to roll-back so once the code gets pushed to beta, we should prevent other changes from blocking a prod release | 0.5 |  |
| 7 | 2 | Main CDK/Code Development | Create parent application load balancer and associate ALB to the Cloudfront Distribution as an origin |  |  |  | 2 |  |
| 8 |  |  | Created nested ALB for testing or use a in-development project to test - Setup listeners and lambda/EC2 targets - Should be common construct/stack that is extensible/re-useable |  |  |  | 2 |  |
| 9 |  |  | Integrate ALB with Cognito authentication (ALB has Cognito integration, Lambda@Edge function may be needed for token validation) |  |  |  | 3 |  |
| 10 |  |  |  |  |  |  |  |  |
| 11 | 3 | End-to-end alpha testing | Create testing page on Einsight to test the newly created API |  |  |  | 1 |  |
| 12 |  |  | Deploy CDK stacks to alpha via CDK CLI |  |  |  | 2 |  |
| 13 |  |  | Test newly created API endpoint from local frontend, test hello world endpoint (very primitive regression test) |  |  |  | 0.5 |  |
| 14 |  |  | Create plan and schedule beta deployment (may need to block pipelines) |  |  |  | 0.5 |  |
| 15 | 4 | End to end beta testing | Run pipeline deployment to deploy new changes to beta |  |  |  | 2.5 |  |
| 16 |  |  | Regression test on beta |  |  |  | 0.5 |  |
| 17 |  |  | Display banner on beta/prod giving notice of CDK deployments |  |  |  | 0.1 |  |
| 18 |  |  |  |  |  |  |  |  |
| 19 | 5 | Prod Deployment/Testing | Create in-depth plan for prod deploy including fallbacks |  |  |  | 0.5 |  |
| 20 |  |  | Maybe: Create MCM for prod deploy |  |  |  | 0.5 |  |
| 21 |  |  | Prod deployment |  |  | Prod deploy will likely be split to period over the weekend if deploy runs into hiccups, because re-deploying CDK iterations will likely take 30+ minutes each deployment | 1 |  |
| 22 |  |  | Testing of new API and regression testing for existing APIs |  |  |  | 1 |  |
| 23 |  |  | Email notice to dev team and customers |  |  |  |  | 0.5 |
| 24 | 6 | Documentation/Wrap-up | Document process, unexpected blockers, and create instructions on how to create/deploy new EinSight backend APIs |  |  |  | 2 |  |
| 25 |  |  | Create a CR rule for CDK packages to better ensure that best practices are followed with SDE review  Optional: Add CR verification tool to fully enforce that code is reviewed |  |  |  |  | 2 |
| 26 |  | Total Estimated Time: |  |  |  |  | 23.1 | 25.6 |
| 27 |  | Crunched Estimate: | Estimate with faster than expected progress and minimal blockers/interruptions/roadbumps |  |  | #1 - 3 days #2 - 7 days #3 - 2 days #4 - 3 days #5 - 3 days #6 - 1 day  **Note: CDK deployments are very trickly to iterate on and debug due to the nature of them being related to AWS resources. Issues are often discovered at deployment even if the code builds successfully.**  I would add an extra 3-5 days realistically unless we somehow get it in the first attempt. | 14 days |  |

### Test Plan

For testing, this should be pretty straightforward, we would ideally need to test each API gateway with an HTTP request that requires body and one that doesn’t (like POST and GET). If a given HTTP method works for one endpoint in an API, the same can be said for every other endpoint in the API assuming that these endpoints were previous working.

## Document History

|  |  |
| --- | --- |
| **Date** | **Event** |
| 10/10/2022 | Initial document creation |
| 10/25/2022 | Basic design detials and high level tasks |
| 11/2/2022 | Added various design |
|  |  |
|  |  |

One Pager on