Eng.Feature Implementation Standards

V0.1 *Dec/07/2020*

Contents

Overview

The decisions below must be adhered to unless a P2/P1/E2 decision overrides them for a specific product.

Software Implementation

Best Practices		ITD
1	Write simple code, do not overengineer. Implement the minimum needed to meet the stated requirements. Do not add abstraction layers or frameworks to make it simpler to extend the functionality in the future.	
2	Write modular code. Software components should have well defined interfaces.	
3	Write readable code. Lines of code are free. Better to write 3 readable lines of code than one line that requires a magic decoder ring. Use meaningful names for variables and methods so the code is self documenting. Use comments when the intent of the code is not obvious.	
4	Do not add complexity to address scaling or performance concerns unless the P2/P1/E2 spec indicates a particular target requirement.	
5	We will use TypeScript to write backend and front end code	**ITD 1**
6	Develop using the latest stable version of the selected programming language and libraries.	
7	Product business logic will live in the backend, not in the frontend.	

8	**Object Oriented Programming**: When business logic operates on data attributes pertaining to a single entity, put it in methods within the same class as the data attributes. For example, if a User entity contains ‘firstName’ and ‘lastName’ attributes, then a method that returns the full name of the user should be added to the User class representing the entity:
	
	Class User {
	  firstName: string;
	  lastName: string:
	
	  getFullName() {
	    return firstName + “ “ + lastName;
	  }
	}
	
	If the data class is auto-generated (e.g. from an AppSync schema) then put the business logic in a wrapper class.
	
	Business logic that operates across 2 or more entities should go in a separate Service class.
		ITD 2

9	Adhere to the SOLID design principles.	
10	Use AOP techniques (Decorators) to separate concerns in TypeScript code	**ITD 3**

TypeScript

Best Practices		ITD
1	We use TypeScript instead of JavaScript precisely because it is ‘typed’. Do not undo the benefits of typing by using ‘any’ declarations unless required to integrate a third party library.	
2	When calling multiple async methods that are independent (e.g. inputs to one method don’t depend on outputs from another), call them in parallel by wrapping the async calls using Promise.all(). See ITD for example.	ITD 4

Infrastructure as Code (IaC)

Best Practices		ITD
1	Use AWS CDK to setup and maintain AWS resources.	

CI/CD

Best Practices		ITD
1	Use GitHub Actions for CI/CD	
2	Use GitHub hosted runners where possible. Self-hosted runners can be used for internal network access, to support specific hardware or system architectures, or for specialized build environments.	
3	For AWS access use the GitHub OIDC provider to directly assume a role, instead of using a secret key.	

Authentication

Best Practices		ITD
1	Must enable case insensitivity for user ID field	
2	Minimum password strength must meet the following requirements: ● Minimum length: 8 ● Require a number ● Require an uppercase letter ● Require a lowercase letter ● Do NOT require a special character	
3	Enable a self service password reset (e.g. a ‘I forgot my password’ link)	
4	Do not require periodic password changes (e.g. once a month).	

5	Use Triggers to map complex authorization scenarios (such as multi-tenant, multi-level permissions) to `cognito:groups`.	

Sensitive Data

Best Practices		**ITD**
1	Store sensitive application data such as passwords, credentials, and API keys in AWS Secrets Manager. Setup rotation schedules for services that support it (e.g. RDS). Store sensitive user data in the application database after encrypting with a KMS key.	ITD 6
2	No sensitive data should be checked into a repo (e.g. embedded within IaC code such as CloudFormation scripts).	

APIs

General

Best Practices		**ITD**
1	API endpoints are responsible for validating all inputs. It should never be left solely to the front end client code to enforce constraints (although a good front end will guide the user through providing valid inputs). This is true even if the API endpoints are consumed only by front ends built as part of the product (i.e. not third party).	

GraphQL (AppSync)

GraphQL defines an API query language and a server side runtime for executing those queries. AWS AppSync is an instance of a GraphQL service.

References:

- Production Ready GraphQL book
- GraphQL Schema Design @ Scale (Marc-André Giroux)
- The complete guide to AppSync Subscriptions
- Handling GraphQL errors like a champ with unions and interfaces
- Lessons from 4 Years of GraphQL
- Ten Tips And Tricks for Improving Your GraphQL API with AWS AppSync (…
- How to design a kick-ass GraphQL schema
- Lessons learned: AWS AppSync Subscriptions

Best Practices	**ITD**

1	Enable X-Ray tracing for observability	
2	When using VTL mapping template with a DynamoDB datasource Resolver, If pagination is detected (nextToken field is present and not null in DynamoDB result in AppSync) by the Response Mapping Template, it must add an error "Unexpected Pagination - [FIELD PARENT.FIELD]" to enable monitoring and know if the resolver needs to become a Lambda direct resolver so that all pagination is resolved correctly, or pagination needs to be introduced for this field.	
3	For pagination when using DynamoDB Single Table Design, follow the specification in Appendix - GraphQL Pagination Standard for DynamoDB	
4	Create opaque Pagination Cursors from Entity's PK and SK fields	Link
5	VTL Field Resolvers that use a DynamoDB Resolver must detect if the Query result was unintentionally paginated and, if so, add a GraphQL error "Pagination unexpected error" to the response.	
6	When using Lambda Resolvers, create one Lambda instance per Resolver	
7	Lambda handlers, in their method bodies, must have only: input validation, authorization and delegation to business methods.	

Data Storage

DynamoDB

References:

- The DynamoDB Book by Alex DeBrie
- AWS re:invent 2020 videos:
    - **AWS re:Invent 2020: Data modeling with Amazon DynamoDB – Part 1**
    - **AWS re:Invent 2020: Data <modeling with Amazon DynamoDB – Part 2**
    - **AWS re:Invent 2020: Amazon DynamoDB advanced design patterns – Part 1**
    - **AWS re:Invent 2020: Amazon DynamoDB advanced design patterns – Part 2**
- How Amazon DynamoDB adaptive capacity accommodates uneven data access patterns (or, why what you know about DynamoDB might be outdated) | Amazon Web Services
- Fundamentals of Amazon DynamoDB Single Table Design with Rick Houlihan

Best Practices		**ITD**
1	Use AWSDateTime and ISO-8601 strings to store timestamps. (*not applicable to TTL fields)*	**Link**
2	Use ULID as a global unique identifier for keys.	**Link**
3	Use UpdateItem with conditional checks to update items in VTL Resolvers, passing the minimal amount of attributes to update as possible.	

Compute

Best Practices		ITD
1	Use the latest stable AWS OS images and runtimes (e.g. Node.js, JVM, etc.)	
2	Select Graviton over x86 when deploying compute instances on Lambda, EC2, etc. unless there is a product requirement to use x86 (e.g. a required third party binary only runs on x86).	

Lambda

Best Practices		ITD
1	Enable CloudWatch logging and insights	

DevFlows

Best Practices		ITD
1	Each new library (e.g. AWS, Messaging, etc.) should go into its own repo. New Apps should be placed in a subdirectory within their library's repo. Apps will have their own version and be deployed independently.	DDD
2	Package new code written for an Adapter in a package specific to that Adapter. If you determine the functionality you need has already been written for another Adapter then refactor that code into a common package and leverage it from your Adapter (and the other Adapter). Don't put code into a common package because you 'think it might be' needed by future Adapters.	ITD 5

Client Endpoints

General Best Practices		ITD
1	Text displayed to an end user should never be hard coded. Use standard i18n/L10n internationalization/localization procedures even when only one language is the initial target.	

Web Endpoints

General Best Practices		ITD
1	Do not use the CSS '!important' rule unless there is no other mechanism for achieving the desired behavior (rare).	

Unit Testing

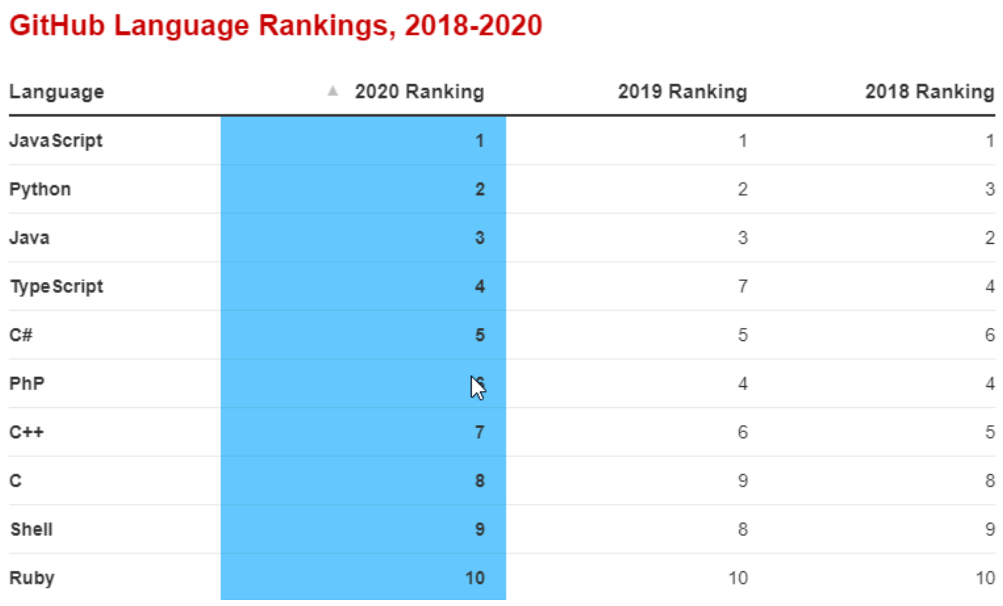
Best Practices		ITD

Observability

Best Practices		ITD
1	Enable CloudWatch logging for all AWS Services that support it.	
2	Enable X-Ray monitoring for all AWS Services that support it. Instructions here.	
3	Follow these guidelines to enable uptime reporting for the product	

ITDs

ITD 1 - Use Typescript to implement software for the backend software components of all new products (includes 5K rebuilds)	
THE PROBLEM	There are a number of programming languages that we could use to implement a New or 5K product's backend software. Which one should we choose?
OPTIONS CONSIDERED (Decision in bold)	1) Javascript **2) TypeScript** 3) Java 4) C/C++ 5) Go 6) Python 7) Choose the language most optimal for the product being developed
REASONING	A perfect programming language does not exist. Each was created and optimized to address specific problems (e.g. fast to write, fast to execute, easy to maintain, portable, etc.). If we were developing a single product we would choose Option 4) in order to produce the most optimal implementation. We reject it for our factory model, however, because it would add significant complexity to our hiring, training, tooling, and support systems to do so. We reject Javascript and Python because they do not have compile time type checking. This pushes the detection of many types of programming errors from compile time to runtime and therefore decreases both quality and efficiency. Although an argument can be made that Go is better technically than the other options, it doesn't rank in the top 10 of the most used languages on GitHub (see table in Feedback section). This would complicate recruiting and training. More importantly, tooling (IDEs, build tools, debuggers/profilers, etc.) and general community support also lag significantly behind those for the other options. We reject Go for these reasons. To evaluate the remaining options it is important to define the important attributes of the New and 5K products that we will be building: ● Serverless, AWS cloud native ● Preference for SPA clients connected to backend APIs ● Backends that are predominantly 'glue code' that connect AWS services together to implement core functions. ● Microservices preferred over monolithic architecture, code reuse is not a primary concern ● Built using the minimum amount of code required to implement the functionality needed today rather than complex frameworks and abstraction layers Given the above, we choose Typescript over Java and C/C++ for the following reasons: - TypeScript syntax is more 'relaxed' and therefore quicker and simpler to write (e.g. a single number type rather than Java's int/float/ double., type inference, etc.) - Common type definitions can be used across web clients and the backend - First class support for JSON (a format commonly used with AWS services) - Although less of an issue now, Node.js (TypeScript runtime) loads faster than a Java VM on Lambda - AWS services have replaced the need for common frameworks and application servers that Java developers would normally take advantage of (e.g. Tomcat, Jetty, etc.) In short, TypeScript represents the sweet spot between a scripting language (JavaScript) and a general purpose programming language (Java). It marries the speed of development of the former with just enough of the benefits of the latter (typing) to allow for high quality and maintainable code.

FEEDBACK	● This decision only applies for New or 5K products. We will not rewrite existing products unless an explicit decision to do so has been made in the P2/P1/E2. ● E2s that believe an alternative programming language is warranted for a particular product can make an ITD stating the reasoning why the choice would be 'significantly better' and therefore warranted (e.g. 'we choose to use Python because we need to take advantage of the Pandas data manipulation library'). 

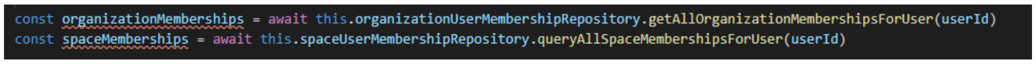
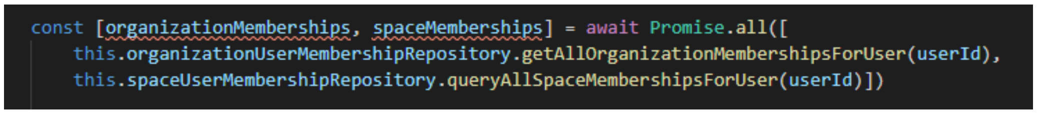
GitHub Language Rankings, 2018-2020

Language	2020 Ranking	2019 Ranking	2018 Ranking
JavaScript	1	1	1
Python	2	2	3
Java	3	3	2
TypeScript	4	7	4
C#	5	5	6
PhP	6	4	4
C++	7	6	5
C	8	9	8
Shell	9	8	9
Ruby	10	10	10

ITD 2 - Use a Rich Domain Model to model a product's domain	
THE PROBLEM	How should we model a product's domain (behavior and data) within our software?
OPTIONS CONSIDERED (Decision in bold)	1. Use an Anemic Domain Model **2. Use a Rich Domain Model**
REASONING	The Anemic Domain Model is a procedural design pattern that appears OOP'ish because it collects data attributes in a class object. Business logic, however, is located separately in services classes. The Anemic Model therefore suffers from the same primary drawbacks as standard procedural design: lack of data encapsulation. Use of the Anemic Model is primarily seen in simple applications where tooling such as ORMs automatically generate data classes and it is therefore simpler to write the logic separately. We choose the Rich Domain Model because OOP design principles are important to ensure data consistency and software maintainability. This choice does not prevent us from using tooling to generate code (e.g. from an AppSync schema).

ITD 3 - Use AOP techniques (Decorators) to separate concerns in TypeScript code	
THE PROBLEM	How to handle cross-cutting concerns (e.g. authorization, logging or caching) in Typescript?
OPTIONS CONSIDERED	1. Handle them with direct invocations of appropriate methods 2. Delegate them to a dedicated service

(Decision in bold)	3. Extract a base class and handle them there 4. **Use AOP techniques (Decorators) to separate concerns in Typescript code**
REASONING	There are several methods of handling cross-cutting concerns. A good comparison is presented in this article. In short: ● Option #1 creates a lot of code duplication and is the most error-prone ● Option #2 still creates some code duplication (e.g. you need to handle results of *isUserAuthenticated* method in each of the methods) so it still violates DRY (Do not Repeat Yourself) principle. ● Option #3 reduces boilerplate code but violates SRP (Single Responsibility Principle) as the service class is now responsible for both your business logic and some cross-cutting concerns (logging, authorization). ● Option #4 is the best option as it does not violate either SRP or DRY and does not introduce any boilerplate code. We select option #4 as it's a widely adopted technique for separating cross-cutting concerns from the actual business code.

ITD 4 - Prefer parallel calls over sequential calls when calling independent async methods	
THE PROBLEM	When there is a need for making multiple, independent async queries (e.g. DynamoDB calls) should we use parallel or sequential await statements?
OPTIONS CONSIDERED (Decision in bold)	1. Always use sequential awaited queries 2. Use parallel calls only for calls with high latency 3. **Always use parallel calls**
REASONING	We reject option #1 because making sequential calls for high latency operations (e.g. paginated DynamoDB calls) would result in poor performing code. We reject option #2 because it is simpler to treat all async calls the same rather than attempting to define what 'high latency' means (especially since query response times can change over time). We select Option 3) because it is more performant than the other options and its only downside is adding a line of wrapper code. Ex: Slower code  `const organizationMemberships = await this.organizationUserMembershipRepository.getAllOrganizationMembershipsForUser(userId)` `const spaceMemberships = await this.spaceUserMembershipRepository.queryAllSpaceMembershipsForUser(userId)` Faster code:  `const [organizationMemberships, spaceMemberships] = await Promise.all([` `this.organizationUserMembershipRepository.getAllOrganizationMembershipsForUser(userId),` `this.spaceUserMembershipRepository.queryAllSpaceMembershipsForUser(userId)])`
FEEDBACK	Note that there are cases where independent queries may need to be serialized in order to prevent exceeding API rate limits or other business logic specific issues.

ITD 5 - Move code that is reused across adapters to a commons repository	
THE PROBLEM	There are some functions that are shared across different adapters (e.g. create bucket, put file to a bucket, deploy a lambda function), where should we put such common functions that are reused across

	adapters?
OPTIONS CONSIDERED (Decision in bold)	**1) Move code that is reused across adapters to a commons repository** 2) Pro-actively move code that might be reused to a commons repository
REASONING	Option 2 doesn't make any sense in an ordinary world, there is no way we can predict a code will be reused unless we have a real use case. We select option 1 because we don't want the commons repository to be a fat repo that receives all the code including specific client API code such as Zendesk, CloudCRM and Jira.
FEEDBACK	The default choice is to not move code to commons and place it underneath each adapter repository.

ITD 6 - Store sensitive application data in Secrets Manager and sensitive user data in the application DB	
THE PROBLEM	Where should we store sensitive data such as passwords, credentials, and API keys?
OPTIONS CONSIDERED (Decision in bold)	1. AWS Secretes Manager 2. Application DB (encrypted using KMS) **3. Both 1) and 2)**
REASONING	AWS Secrets Manager is purpose built to store sensitive application data. It makes it simpler to refresh/rotate secrets rather than implementing the functionality in the application code. We therefore choose it for application level data (e.g. RDS passwords, API credentials to a third party service, etc.). We reject it for sensitive user data, however, because Secrets Manger was not built to handle user level scale (it has 40K secret hard limit) and it is generally more efficient to load user secrets in the same application database query that loads the rest of their profile data rather than making two dips. We therefore choose Option 2) for storing sensitive user data after encrypting that data using a KMS key.

Proposed ITDs

ITDs in this section are proposed. They will be moved to the appropriate section above once approved.

ITD ? - Use one AWS account for development, IT test, and staging instances and one for production	
THE PROBLEM	We will need to deploy multiple instances of the product's tech stack in AWS for development, IT testing, staging, and production purposes. How many AWS accounts should we use?
OPTIONS CONSIDERED (Decision in bold)	1. One account for all instances 2. One account for each development, test, staging, and production instance **3. One account for all development, testing, and staging instances, one for production**
REASONING	Stability of the production instance is critically important. While it is possible to use IAM permissions to protect the production instances of most AWS services, some services cannot be protected at the instance level. We therefore reject Option 1) as it can't guarantee production stability. Option 2) ensures complete control over access permissions for each instance but is inefficient for development and test purposes as some ICs will need to access multiple dev and test accounts.

	We therefore choose Option 3) as it ensures stability of the production system while maintaining efficiency for development and test purposes.

ITD ? - Use both naming prefixes and tags to identify environments, ownership, and prevent namespace collisions	
THE PROBLEM	Per the above ITD, we need to deploy multiple stack instances to the same AWS dev account. How can we make it easy to identify which resource instances belong to which environment, manage resource name collisions, and identify which user deployed/owns each resource?
OPTIONS CONSIDERED (Decision in bold)	1. Adding a prefix to each resource name based on the environment: a. Dev: Dev-<username>-<user defined resource name>-<service name> b. IT Test: IT-<GitHub Run ID>-<service name> c. Staging: Staging-<service name> 2. Use tags to identify the environment type [dev,it,staging,production]. For 'dev' types, set the value to the username who deployed the environment **3. Both Options 1) and 2)**
REASONING	We choose Option 1) because it prevents namespace collisions and makes it simple to determine which environment a resource belongs to and who owns the environment. We also choose Option 2) however as it allows users to use the Resource Groups tool to view only those resources associated with one or more tags and therefore increases efficiency. It also enables automation to make decisions based on type and owner (e.g. if a dev account hasn't been modified in 30 days, the owner will be notified it has been marked for deletion).
FEEDBACK	● We choose to use a hyphen as the separator for our prefixes as it is supported by all commonly used services whereas an underscore is not supported by S3 or CloudFormation. ● The <user defined resource name> portion of the prefix allows a user to have more than one dev environment when needed. ● We may need to truncate usernames and service names if names become too long to easily view within the Management Console.

ITD ? - Use *Retain* deletion policy for key resources (with data) by default, but also allow for override to *Destroy* with a context variable	
THE PROBLEM	When destroying CloudFormation stack, which deletion policy should we use for resources?
OPTIONS CONSIDERED (Decision in bold)	1. Use *Destroy* for all resources 2. Use *Retain* for all resource 3. Use *Retain* only for key resources with user data and *Destroy* for all of the others **4. Use *Retain* for key resources by default, but also allow override to *Destroy* with a context variable**
REASONING	Option #1 is the easiest to implement as *Destroy* is the default deletion policy for almost all resources, but it will also mean that any data stored by the user will be lost when someone accidentally triggers the stack destroy procedure. We reject this option as it can create serious problems in the production environment. Option #2 is the safest one but not all resources keep the state - e.g. we do not have to keep lambdas or

	IAM policies, we only want to keep key ones with data (e.g. S3 Bucket, DynamoDB Table, Cognito UserPool etc). Keeping all the resources will create additional burden for the user to manually manage/delete those stateless, thus we reject this option. From options #3 and #4, the former is easier to implement, however we also acknowledge that the most common use case for development is to remove all resources on stack destruction. We still want to keep this functionally to make development easier, thus we select option #4.
FEEDBACK	It's up to Feature team to decide how to implement this functionality, but preferably some kind of framework support should be provided (e.g. a sample code should be put in eng-template repository)

ITD ? - Auto-generate Domain Types from GraphQL schema	
THE PROBLEM	OOP languages such as Java and TypeScript require the domain model to be represented natively. How should we implement these representations?
OPTIONS CONSIDERED (Decision in bold)	1. Manually map Domain Objects from GraphQL Schema **2. Auto-generate Domain Types from GraphQL Schema**
REASONING	Option #1 would require us to maintain 2 different object models - one in GraphQL SDL files where 'types' are stored, and another one in TypeScript types/interfaces. Both would need to be kept in sync manually, which is an additional burden for developers and also a place where regression errors can appear. To simplify the process, we select option #2 which will allow us to keep only one source of truth about the domain objects and auto generate TypeScript types/interfaces from it. This will work in a similar way as the Swagger Codegen tool is used to generate code from OpenAPI specification.
FEEDBACK	Use GraphQL Code Generator to generate Types for TypeScript.

ITD ? - Extend auto-generated Domain Types to add business logic.	
THE PROBLEM	Types generated from GraphQL schema don't have business logic and contain only a subset of the required attributes to be correctly persisted in the database. How should we enhance the auto-generated domain types?
OPTIONS CONSIDERED (Decision in bold)	1. Edit auto-generated types directly **2. Extend auto-generated types to add business logic.** 3. Place all business logic in other types, such as [Entity]Service types
REASONING	Editing auto-generated types directly would mean we need to store them in repository and not auto-generate each time they are required. This would also mean that we will need to handle any GraphQL schema change manually, i.e. regenerate them from GraphQL schema and apply our changes later on. We reject this option as it requires manual intervention and an automated process can overwrite manual changes. Using Service types to store business logic contradicts ITD 2 and promotes Anemic Domain Model, which is considered an anti-pattern. Due to this reasoning, we reject it. We select option #2 as it supports the Rich Domain Model and allows us to extend our business logic without interfering with auto-generated code.

FEEDBACK	Feedback from Feature on how to implement it is located here.

ITD ? - Map errors and have them returned correctly	
THE PROBLEM	When Web API callers pass an invalid arguments, what should the service implementation do?
OPTIONS CONSIDERED (Decision in bold)	1. Return false/null/other valid values 2. Throw an application exception **3. Return specific error messages**
REASONING	It is a best practice to map API errors in known situations (such as illegal arguments) to standard errors to make API easy to develop against and provide client or developer with information to help troubleshoot the problem. Simply returning a valid value (such as null when searching for an entity) can lead to ambiguity about the result. So we reject Option #1. Option #2 can lead to generic error messages from the underlying service - AppSync, for example, creates a generic error message when Lambdas throw Exceptions. So we choose Option #3 to enforce that errors are properly mapped instead of letting the underlying error (such as database exception) leak to the API.
FEEDBACK	The correct way to return errors depend on the API technology: ● HTTP Error codes with RFC7807 for REST API ● Proper $util.error usage in VTLs for AppSync This presentation about error/response handling can help on getting proper modeling ideas for these scenarios for GraphQL - instead of mapping everything to errors, the use of union types might be useful to discern real errors (service unavailability) from responses (user is disabled).

ITD ? - Map subscriptions to the existing business logic mutations	
THE PROBLEM	We can use AppSync subscriptions for communicating transient state changes between client and server. How should we map these subscriptions?
OPTIONS CONSIDERED (Decision in bold)	1. Create mutations for the specific purpose of notifying the UI and assign subscriptions to those mutations **2. Map subscriptions to the existing business logic mutations**
REASONING	While Option #1 allows the backend to have explicit control of the events sent to Sococo clients and make for a simpler subscription scheme, we reject it because it adds complexity to the backend in the form of event mapping - the backend needs to map each and every possible UI notification - and an extra GraphQL call for each generated notification. Option #2 is the main reason to use AppSync - since all Activity Manager changes are triggered by a mutation, the mutation response contains all necessary data to update every connected UI, without any management overhead in our business code. The backend is then simpler because it doesn't have to map business logic to intended notifications nor call extra services to send notifications to the UI. This option then allows further decoupling between the UI and Backend code, since the UI can decide what different changes it wants to observe and react to, instead of having this logic tied in the backend.

	For changes or events that happen outside of mutations (such as a Zoom meeting ending), the event will be fed to Sococo via Event Bridge - in this case, the Event Bridge integration calls the appropriate mutation to notify the Activity Manager. This option requires proper GraphQL API design, in order to consider both the caller needs and all effects the mutation has on the object graph. These considerations are mapped in the Feedback section and Std.ITDs below.
FEEDBACK	A GraphQL subscription runs after the mutation has been run. The output of the mutation will be the input of the subscription, so the type returned by the subscription must match the type returned by subscribed mutation. We use subscription filters to filter what will be delivered. These are the subscription parameters. All subscription parameters must exist as a field in the returning type, as that's what will be used to filter what's incoming. More information is available in this post about working with subscriptions. GraphQL subscriptions enforce security using resolvers. The subscription resolvers must resolve properly, so if the mutation has fields as return values, they need to be returned by these resolvers. So, in order to allow a correct response for security resolvers, all mutations that will be the target of subscriptions must be allowed to return null - that's what should be returned by the security resolver. More information at AWS documentation. GraphQL mutations can impact more than their immediate returned value - when joining a room, a meeting can be started. The Payload Pattern is being used in GraphQL mutation response types to allow easy mapping (like a shortcut) of the possible affected objects - a JoinRoomPayload, for example, can include a Meeting object to show a meeting was started or is already in place in that room. We'll follow this Pattern to also guarantee that the filter attributes exist in all mutations. The UI must be able to filter by any of the lists managed by Sococo - Organization, Space and Room - and we must have an interface to enforce this. An example of how this looks in practice is found in the GraphQL API with subscriptions.

ITD ? - The SPA and its resources must only be accessed through Cloudfront URL	
THE PROBLEM	How should we enable access to the SPA and its resources?
OPTIONS CONSIDERED (Decision in bold)	1. **The SPA and its resources must only be accessed through Cloudfront URL**. 2. The SPA and its resources can be accessed directly via the S3 URL or through Cloudfront URL.
REASONING	We reject Option #2 because if users access the files directly in S3, they bypass the controls provided by CloudFront signed URLs or signed cookies. This includes control over the date and time that a User can no longer access content, and control over which IP addresses can be used to access content. In addition, if Users access files both through CloudFront and directly by using Amazon S3 URLs, CloudFront access logs are less useful because they're incomplete.

ITD ? - Create opaque Pagination Cursors	
THE PROBLEM	GraphQL pagination requires a cursor to be specified for each Node. How do we generate such a cursor?
OPTIONS CONSIDERED (Decision in bold)	1. Re-use DynamoDB pagination cursor (when DynamoDB is the data store) 2. Create new cursor attribute for each entity

	3. Create opaque pagination cursors from entity's key fields
REASONING	We reject option #1 because DynamoDB pagination cursors are not created for each entity - they are created only when pagination occurs. We reject option #2 because it would require a new index to effectively be able to skip to the "after" cursor value when querying the second page onwards. We select option #3 because the cursor can be created by applying the formula BASE64({ [keyMap] }). This allows the Database Access layer to understand and "decrypt" the cursor to apply the correct PK/SK query comparators.
FEEDBACK	Cursor should be calculated by the Database Access layer.

Unit Testing

Best Practices		**ITD**
	Follow our current Testing ITDs	

Logging

Best Practices		**ITD**
1	Use log4js library for logging, as per ITD.CODE.3 from the Ship Every Merge document.	
2	If logs are supposed to be stored in CloudWatch (streamed directly or through other services, e.g. from lambdas), use JSON format. It will be easier to process logs later on, if required	
3	Use log4js-node-appenders for logging inside lambdas. Use ‘messagePassThrough’ layout to avoid duplicate logging of timestamp and logging level in lambda.	
4	Use log4js-cloudwatch-appender for logging directly into CloudWatch, if required	
5	You may consider using ‘basic’ layout instead of the default ‘colored’ one if your terminal does not support colored output or if you save your logs into a file.	

Appendix - GraphQL Pagination Standard for DynamoDB

This appendix describes how pagination must work in GraphQL API that is backed by a DynamoDB Single Table datasource.

These rules apply to all Connections that are forward-only.

- Paginations are translated to `Query` operations in DynamoDB
- The DDB `Query's` "**Limit**" value must be equal to **`limit` field**
- If the **after** field is set, it must be decoded and its value must be used as the ExclusiveStartKey value of the Query.

- A new `Connection` object must always be created to represent the `Query` results, with fields:
  - `Connection.edges` = an array of `Edge` objects, as described below. If the `Query` has no results, an empty array.
  - `Connection.pageInfo` = a `PageInfo` object, as described below.
- For each returned DynamoDB Item (`DdbItem`), a new `Edge` object is created in the `Connection.edges` array, with fields:
  - `Edge.node` = `DdbItem`
  - `Edge.cursor` = BASE64({ "pk": `DdbItem.pk`, "sk": `DdbItem.sk` })
    - If the Connection is based on a M:N Relationship Item (such as SpaceUserMembership), the DdbItem to be used for cursor generation is not the Edge entity (User), but the Relationship Item (SpaceUserMembership).
    - If the Query was based on an index, such as GSI or LSI, the index keys need to be added as well, so that the cursor can be properly used in the `ExclusiveStartKey`. That means that if querying over LSI, you need to add the LSI sort key to the cursor; if querying over GSI, you need to add the GSI partition key and sort key (if the GSI uses a sort key).
- The `Connection.pageInfo` fields must be:
  - `hasPreviousPage` = `false`, always
  - `hasNextPage` = `true` if and only if the `Query` Result's `LastEvaluatedKey` is present; `false`, otherwise.
  - `startCursor` = `Connection.edges[0].cursor`
  - `endCursor` = `Connection.edges[N].cursor`, with N being the last item in the edges array.
  - For queries that return empty responses, `startCursor = null` and `endCursor = null`.
- There might be a case of `Connection.edges.length < limit` and `Connection.pageInfo.hasNextPage == true`. This is a case of having filters applied to results and not enough items were queried (because of limit) to pass the filtering. This behavior can be avoided if the implementation detects it and calls the same query again, passing the `LastEvaluateKey` value from the result to the new Query's `ExclusiveStartKey`, while this situation occurs. The implementation of this avoidance is not enforced nor prohibited by this definition.