AC215: Advanced Practical Data Science, MLOps

Transforming Computer Vision Models into Cloud-based Products

Camilo Fosco, Pedro Freitas, Juan Segundo Hevia
Tekal, Inc

Who are we?

TEKAL

Reverse-Engineering Memory and Attention

MIT spinoff incubated by:

Hi Harvard innovation lab > *Harvard Spark Grants winner*

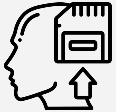
MIT Sandbox Innovation Fund Program
Massachusetts Institute of Technology

CREATIVE DESTRUCTION LAB

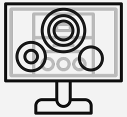
We Predict the Cognitive Impact of Visual Content

We allow clients to assess and improve **creative assets** before publishing them

We focus on two key intrinsic properties of visual stimuli:



MEMORABILITY



SALIENCY

Cognitive gatekeepers of impact

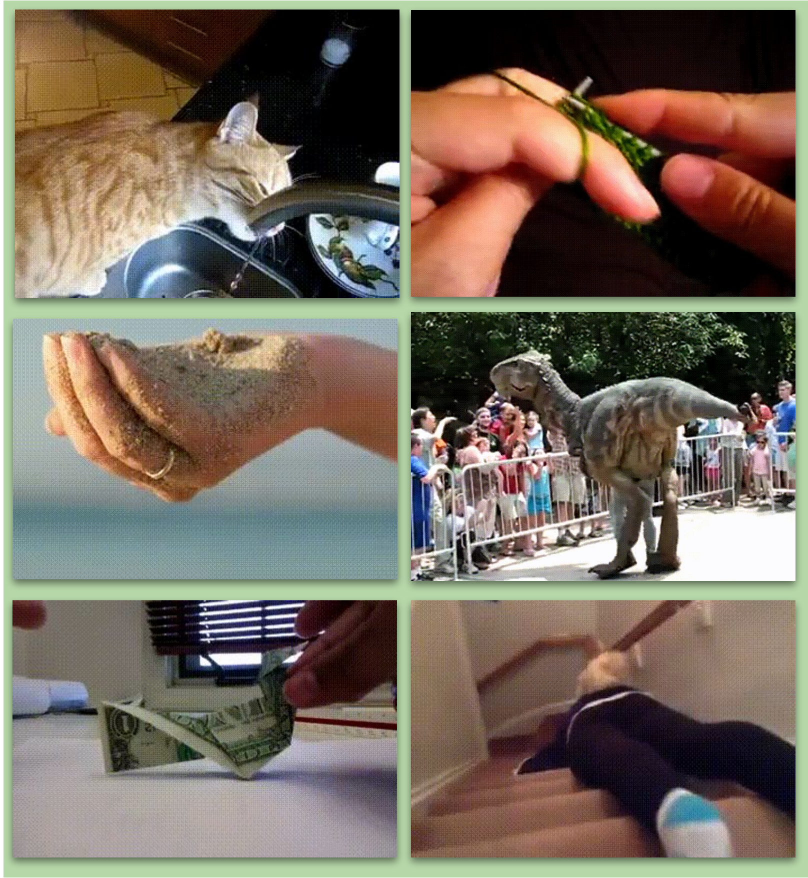
TEKAL

Memorability

The probability that a visual element will be remembered in the future after being seen once.

This helps make ads with lasting impact: building awareness faster and with less impressions.

High Memorability



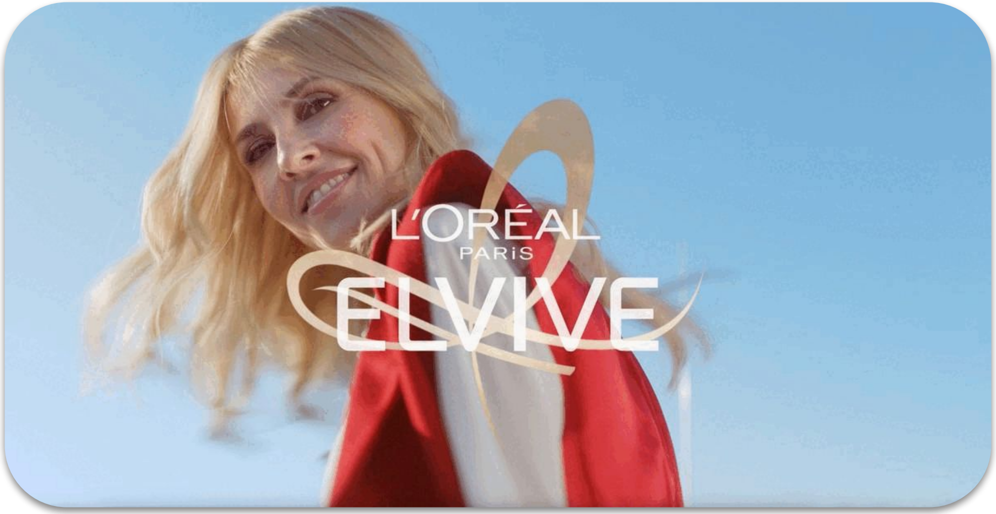
Low Memorability

TEKAL

Saliency

What catches a viewer's eye in the first 3 to 5 seconds.

This helps minimize the risk that the key elements don't capture attention - like logo, product and main claim.

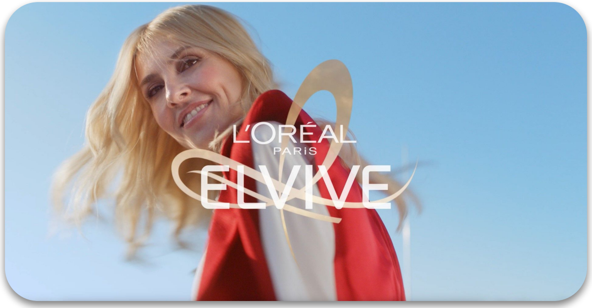


These phenomena are based on low level brain biases that are **consistent across demographics:**

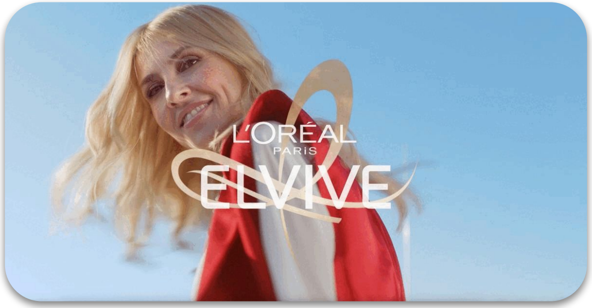
Predictable!

We build models that predict these metrics

Our nets take image or video as input, and generate saliency maps + recall indicators.



DNN models



TEKAL

How does this work?

Data collection

2,180,000+
Human Responses

1,140,000+
Images

18,000+
Videos

Deep-Learning models

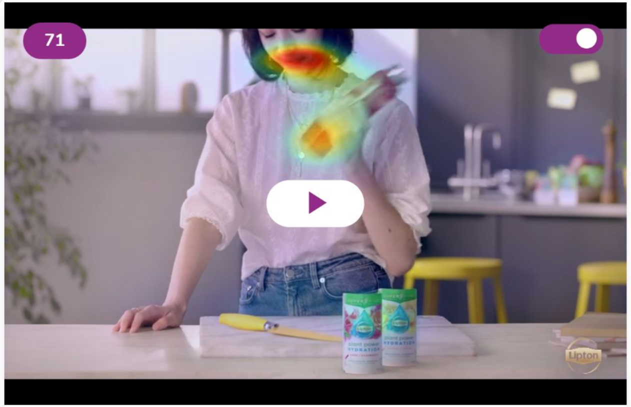
Model validation

0.04
Mean Absolute Error in Memorability

90%
Accuracy in Saliency (AUC)

High accuracy

Our product: Cognitive Analytics Dashboard



+4%

vs. **Competition**

Better than bouillon and 3 more.

+1%

vs. **Campaign**

Knorr Soffrito

+0%

vs. **Brand**

Knorr

+2%

vs. **Sector**

Processed Foods

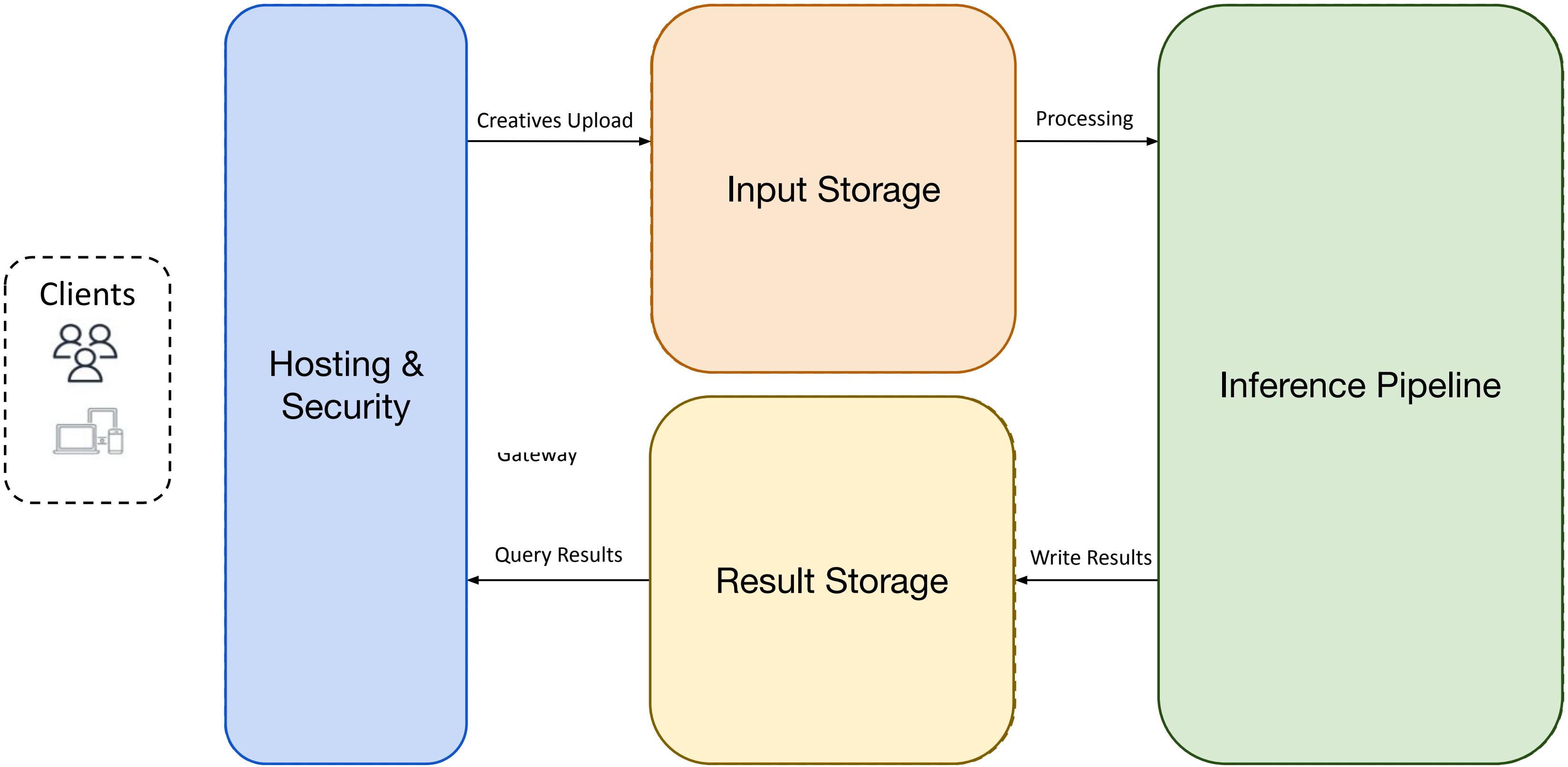
+1%

vs. **Company**

Unilever

TEKAL

Our inference pipeline

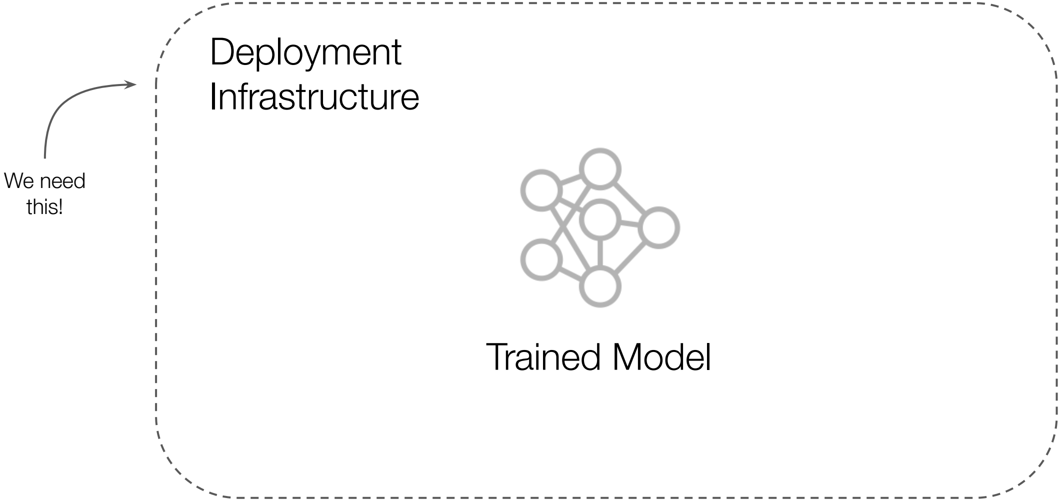


TEKAI

Let’s back up… How did we get here?

Let’s back up… How did we get here?

- Let’s go back to the point where we obtain our checkpoint
- We just finished our initial research process, and we have a trained model
- We have no way of reliably showing predictions to the user, or receiving their inputs

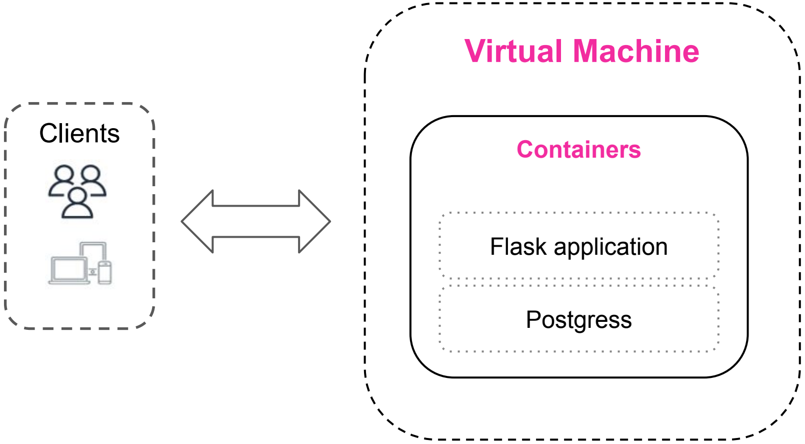


We have a trained model, now what?

To put it in production, we deploy it to a server and make it available through an API endpoint.

First Attempt

- Flask API deployed to a virtual machine
- Relational database to store information



TEKAI

We have a trained model, now what?

Main issues with first attempt:

- User behavior: usage was not frequent, but with high demand peaks when needed
  - Idle most of the time, but we were paying for it
- Too much time setting up the server and API configuration
- Database management was also a pain as usage was increasing and we were quickly adding new variables to the db.

Decision: Let’s go serverless!

> “Serverless computing is a cloud computing execution model in which the cloud provider allocates machine resources on demand, taking care of the servers on behalf of their customers.”
>
> Wikipedia

Greater Scalability

Reduced Cost

Faster go-to-market-time

AWS Serverless Stack

How does this look on AWS?

Storage: S3

Database: DynamoDB

API: API Gateway

Code execution: Lambda

Orchestration: Step Functions

How does this look on Google Cloud?

Storage: GC Storage

Database: Cloud Bigtable, GC Datastore

API: API Gateway

Code execution: GC Functions

Orchestration: GC Workflow

AWS S3

Google Cloud Storage

- Amazon Simple Storage Service (Amazon S3) is **the "hard drive" of our serverless application**;

- It has **built-in security features** to prevent unauthorized access;
    - encryption features and **access management tools** are key when handling with client's assets

TEKAL

AWS Lambda

Google Cloud Functions

Serverless, event-driven compute service that lets you **run code for an application or backend service without provisioning or managing servers**;

Scalable

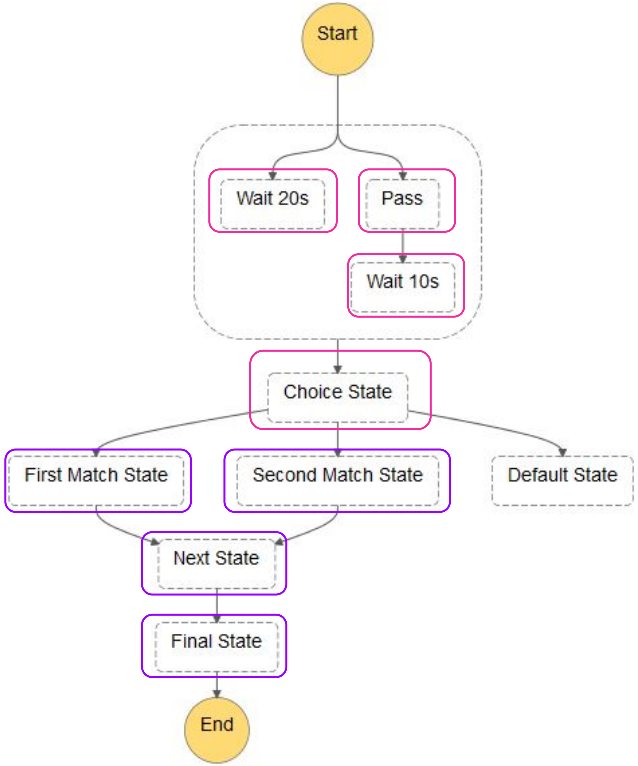
Deployment as container images

Functions can be written in many languages

Reduced Cost

AWS **Step Functions**

Google Cloud Workflows



Visual workflow service to build distributed applications, automate IT and business processes, and build data and machine learning pipelines.

AWS DynamoDB

Google Cloud Datastore

DynamoDB is a **fully-managed, NoSQL database** provided by Amazon Web Services

Primary key		Attributes	
Partition key: PK	Sort key: SK		
ORG#BERKSHIRE	ORG#BERKSHIRE	OrgName	SubscriptionLevel
		Berkshire Hathaway	Enterprise
	USER#CHARLIEMUNGER	UserName	Role
		Charlie Munger	Member
	USER#WARRENBUFFETT	UserName	Role
		Warren Buffett	Admin
ORG#FACEBOOK	ORG#FACEBOOK	OrgName	SubscriptionLevel
		Facebook	Pro
	USER#SHERYLSANDBERG	UserName	Role
		Sheryl Sandberg	Admin

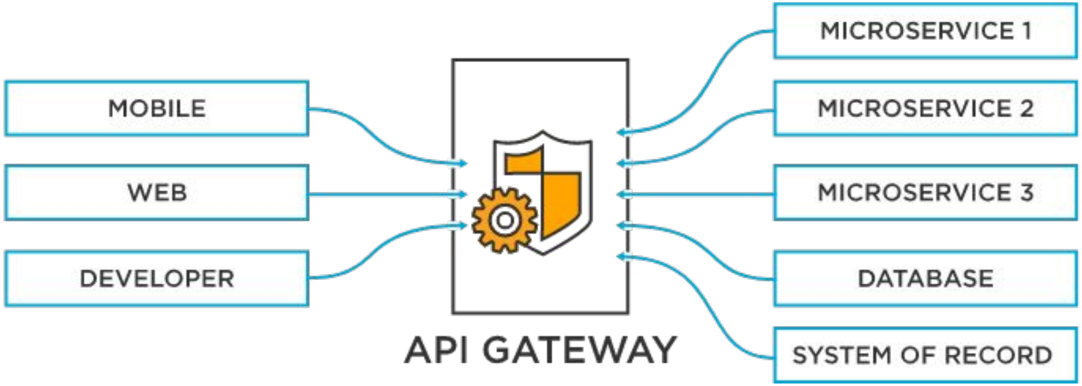
AWS API Gateway

API Gateway

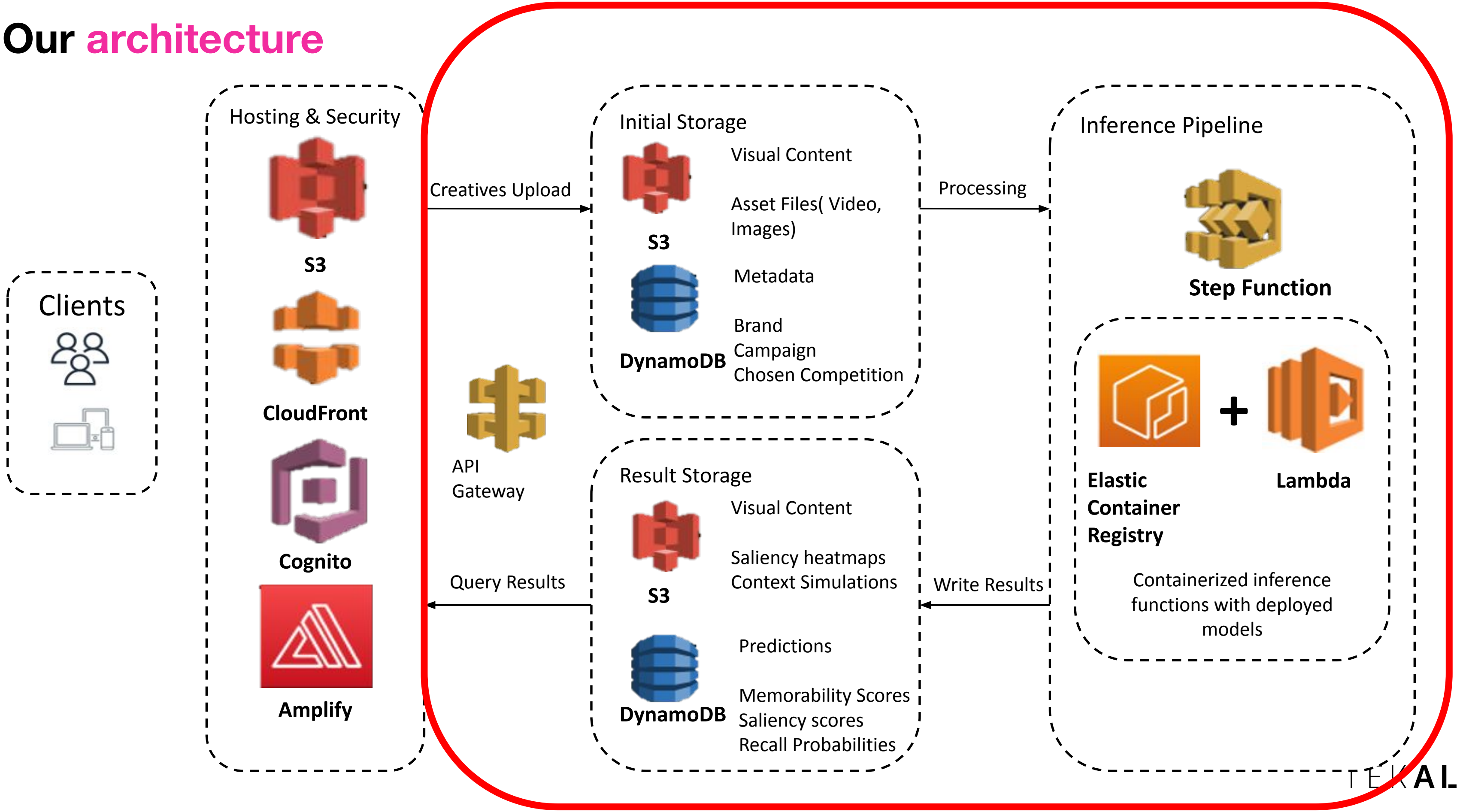
Intermediate layer to handle interactions between our front-end and our database

It provides useful functionalities for scaling applications up

- Security authentication
- Traffic handling and throttling requests

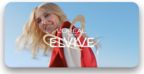


Our architecture



TEKAI

Our pipeline's Lambdas



Original img S3 key

Valid format?

Status: 200, Original img S3 key

Check if file format is supported

Original img S3 key

Simulate context

Simulated img S3 key

Simulate img in spatial context placement and upload new asset to S3

Simulated img S3 key

Process image

Model scores

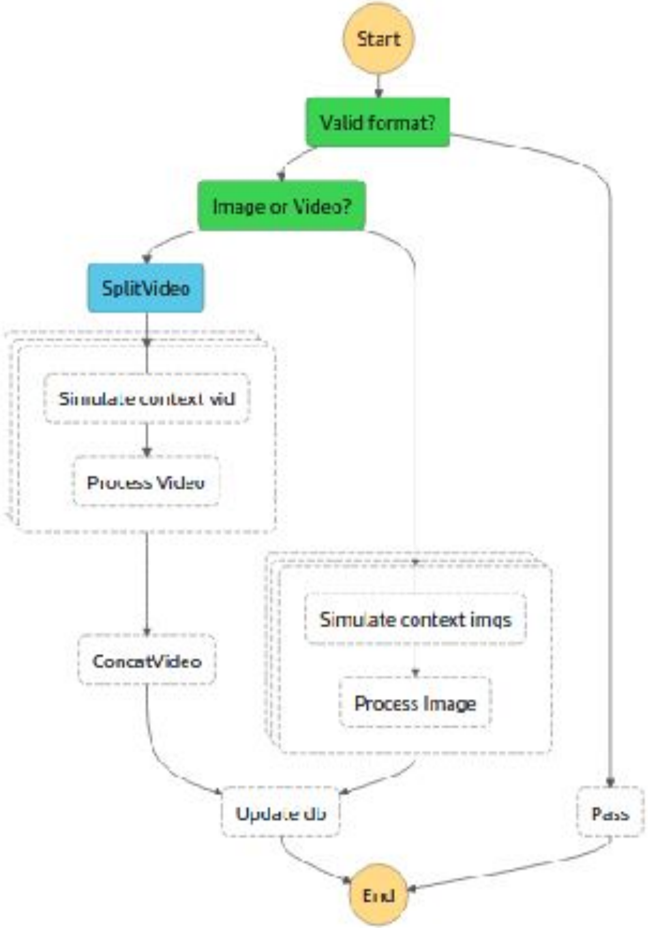
Runs inference. Uploads generated heatmap to S3

Model Scores

Update DB

Status: 200

Update DB with results


Start
Valid format?
Image or Video?
SplitVideo
Simulate context vid
Process Video
ConcatVideo
Simulate context imgs
Process Image
Update db
Pass
End

Sounds good so far, but what about limitations?

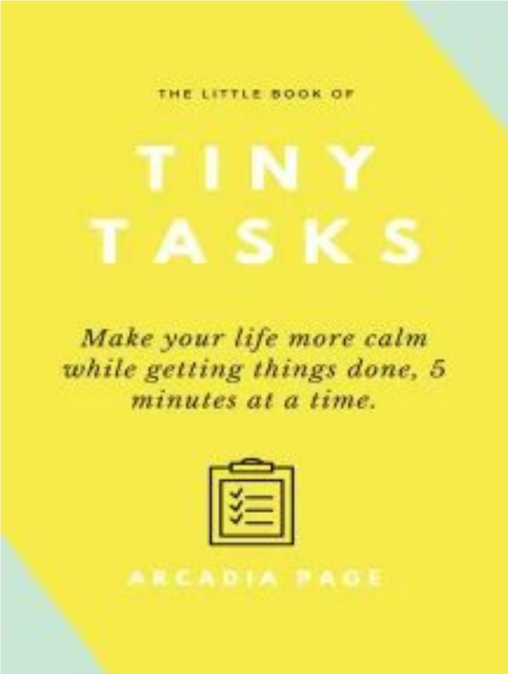
Main Lambdas **limitations for a Computer Vision application**:

- Only CPU - no GPU
- Memory allocation cap: 10240 MB
- Maximum runtime: 15 minutes

Sounds good so far, but what about limitations?

Solution: Work with them as they were meant to be to harness their benefits

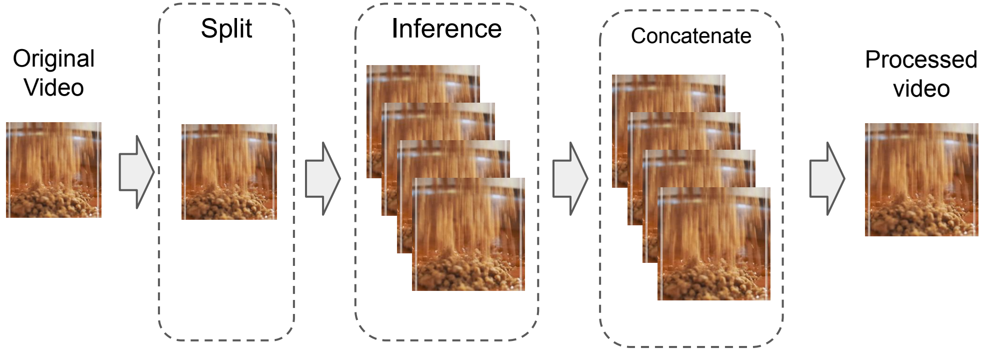
- Lambda functions **are meant to be small and quick** rather than being large applications
  - Have each lambda **perform a small, specific task**;
  - **Split tasks and run parallel lambdas**;
- Convert heavy videos to a lighter format and resolution before inference.

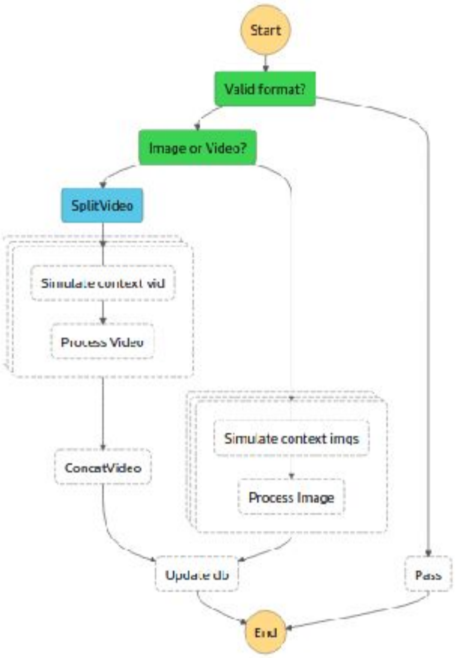


Split a longer task into smaller parallelizable tasks

Example:

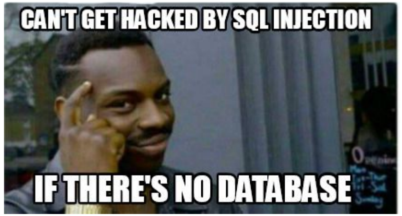
- Say we want to process one video, frame by frame.
  - If the video is too long, the processing job may take too long for an AWS Lambda;
  - A solution might be to split the video in smaller chunks, and then run the processing job in separate parallel lambda instances;
  - Later the output for each chunk are concatenated.





We’ve looked at the pipeline. What about the database?

- We need to store millions of visual assets
- We need to access them reasonably fast
- We need to be able to query specific views of our DB
- We need it to be secure



SQL DB

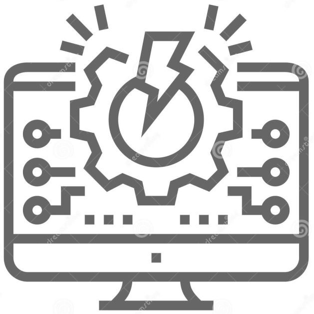
DynamoDB

SQL Limitations: Scalability

At the start we some limitations with our SQL Table

- **Scalability:** Usual OutOfMemory errors

Early on, our SQL database (AWS RDS) usually ran out of memory when handling sorting or complex filters and we had to upgrade its specs. This was both hard to scale and expensive if we expected to handle large amounts of data down the road.



SQL Limitations: Design Flexibility

At the start we some limitations with our SQL Table

- **Design Flexibility:** SQL Schema constrained fast data modelling

We found ourselves coming up with inefficient data models because we needed to iterate fast while coming up with new features.

SQL Limitations: Serverless Integration

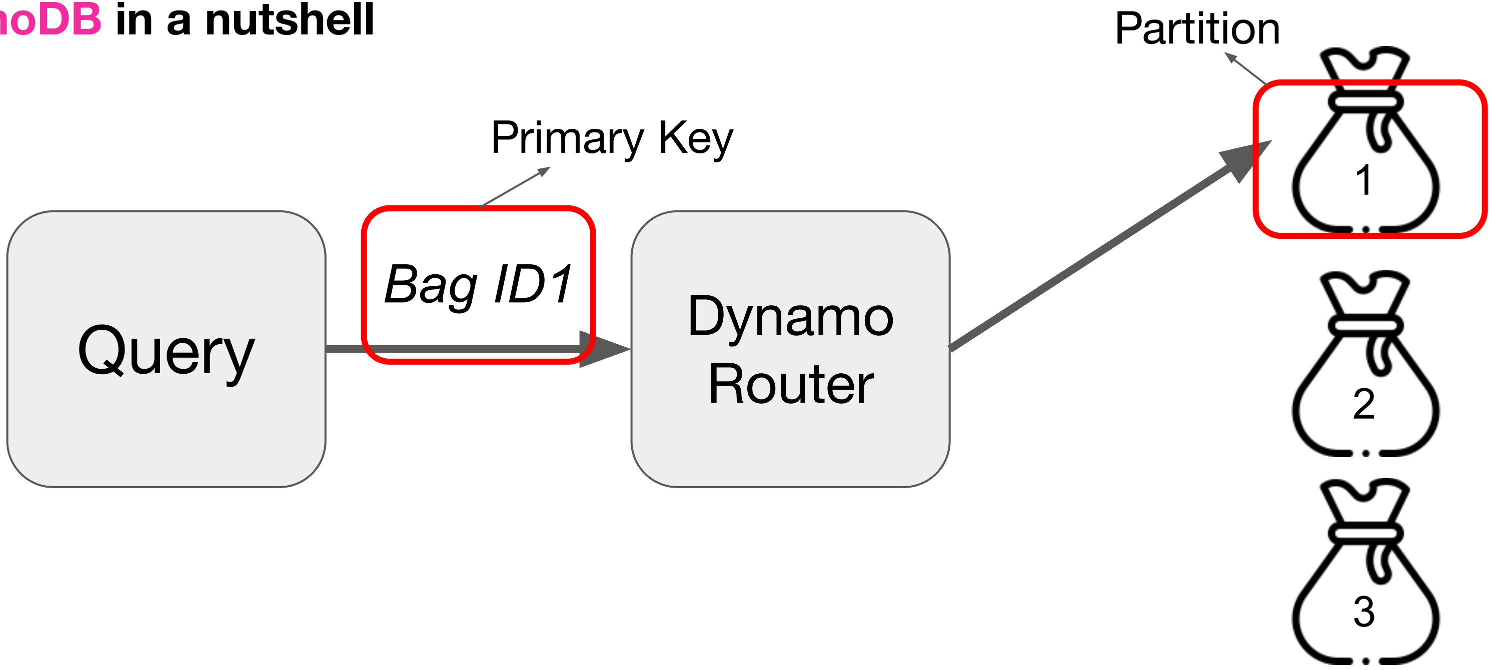
At the start we some limitations with our SQL Table

- **Serverless Integration:** Underperforming adaptive capacity

Serverless applications require their components to be constantly adapting to workloads. AWS RDS required us to actively go and upgrade the resources we had deployed to match larger workloads, leaving the product down for maintenance.

TEKAL

DynamoDB in a nutshell



This search across all bags <u>remains O(1)</u> regardless of the number of bags

Tekal's data infrastructure reminds me of...

Fortunately, both of them have

No SQL

How it **differs from SQL?**

PROS

compared to SQL

- **Built to scale.** Queries' time complexity remains **constant** independently of data storage size
- **Fast**. All our queries point to specific elements in our data model, leveraging Dynamo router logic.
- **Flexibility.** Doesn't have a constraining **schema definition** (but is definitely **not** *schema-less*)

How it differs from SQL?

CONS

compared to SQL

- **Learning curves. SQL has been here for such a long time!** There's tons of documentation for a wide range of applications.
- **No JOINs, no WHEREs. Aggregations and filtering** are not as straightforward, which can make data modelling more challenging.

Querying DynamoDB

Data modelling **depends on the way future users will be trying to query the data: the** database **access patterns**.

This ensures that we only store the data we need, with the structure we need, complying with the front end's queries.

Querying DynamoDB

- With a simple primary key

PK → Item collection → Filtering

- WIth a composite primary key (partition key-sorting key)

PK, SK → Unique Item

- WIth a composite key with a condition imposed on the sorting key

PK → Items where SK begins with ... → Filtering

A few examples: query with composite primary key

Primary key		Attributes	
Partition key: PK	Sort key: SK		
ORG#BERKSHIRE	ORG#BERKSHIRE	OrgName	SubscriptionLevel
		Berkshire Hathaway	Enterprise
	USER#CHARLIEMUNGER	UserName	Role
		Charlie Munger	Member
	USER#WARRENBUFFETT	UserName	Role
		Warren Buffett	Admin
ORG#FACEBOOK	ORG#FACEBOOK	OrgName	SubscriptionLevel
		Facebook	Pro
	USER#SHERYLSANDBERG	UserName	Role
		Sheryl Sandberg	Admin

PK = ORG#BERKSHIRE

AND

SK ***BEGINS WITH*** "USER#"

How do we **leverage NoSQL?**

In Tekal, almost all query operations happen on the *asset* entity.

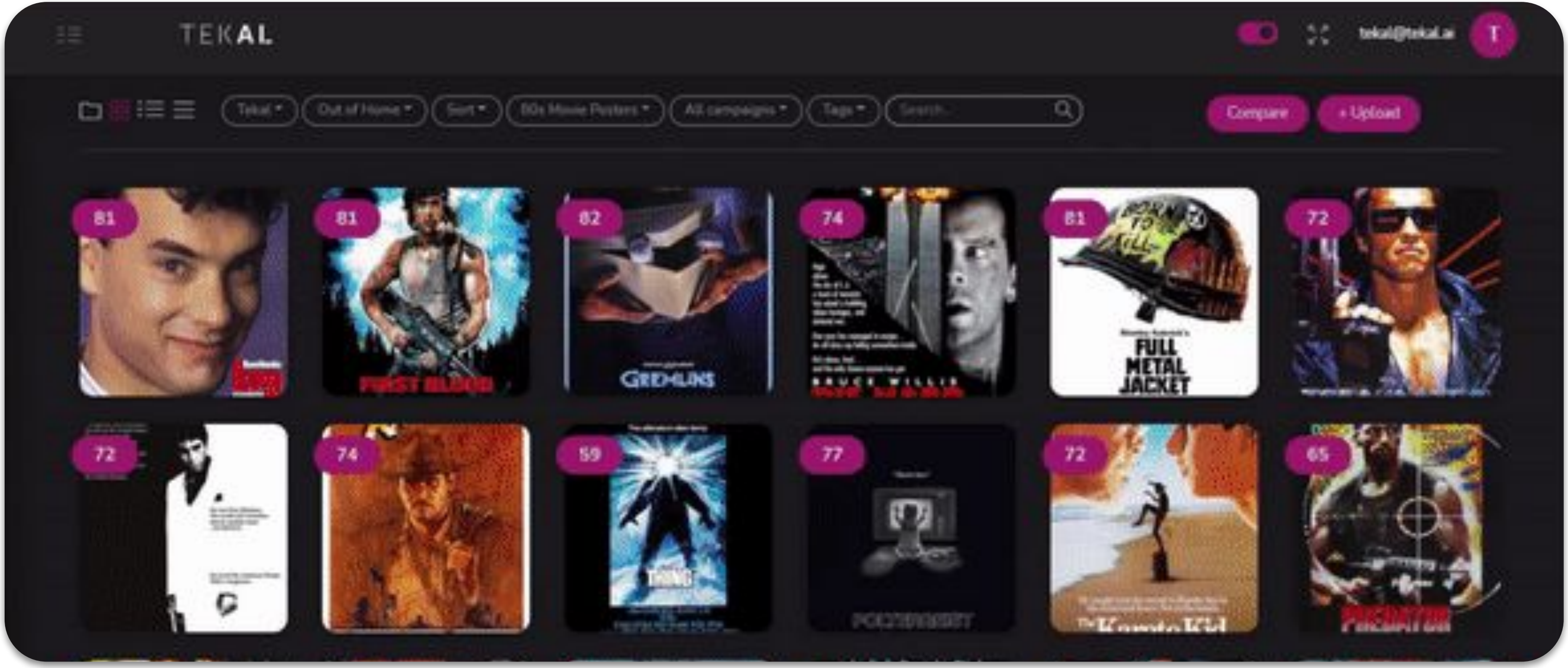
Our partitions are mostly assets

What's an asset?

Video

Image

How do we leverage NoSQL?



What if we want to get all assets for a given client?

Global Secondary Indexes come to the rescue

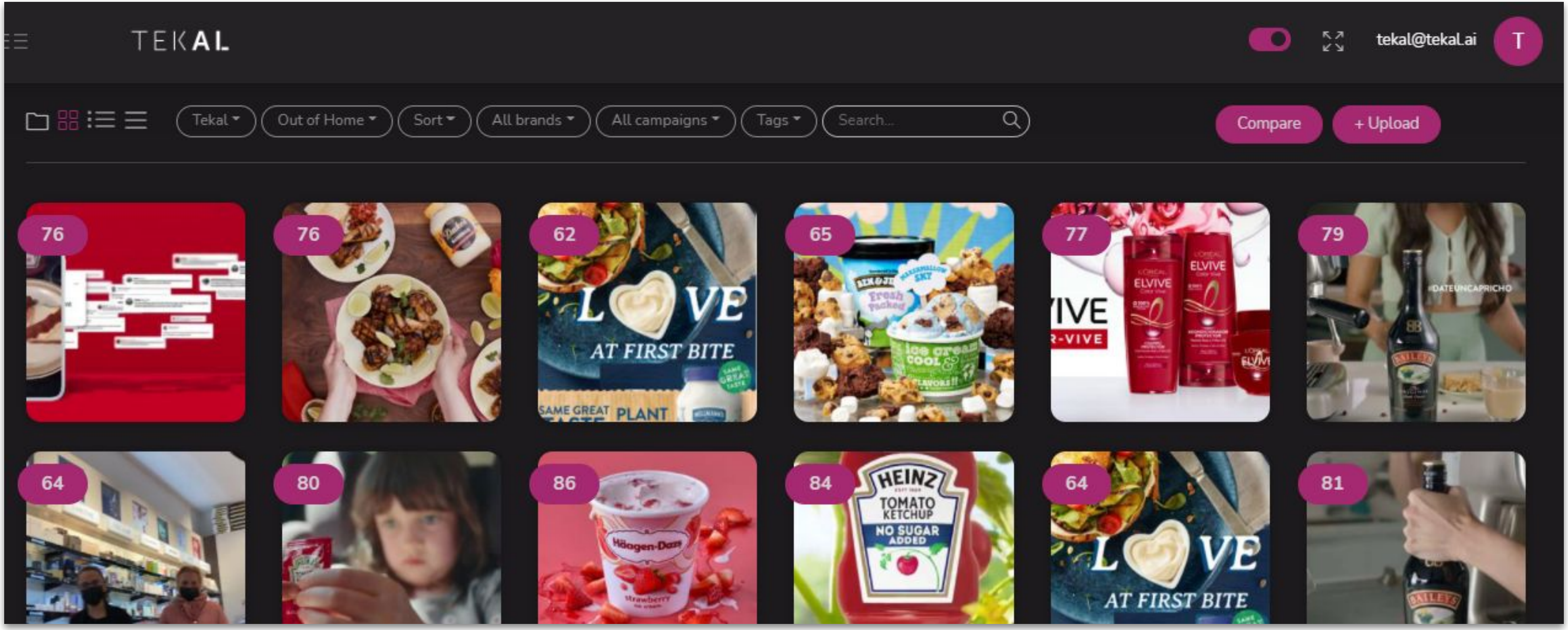
Google Cloud Datastore Indexes

GSIs allow to create a **projection of a DynamoDB table** using other attributes (distinct from PK and SK) as a primary key.

Updated automatically

Allow for reduced table views

How do we leverage GSIs?



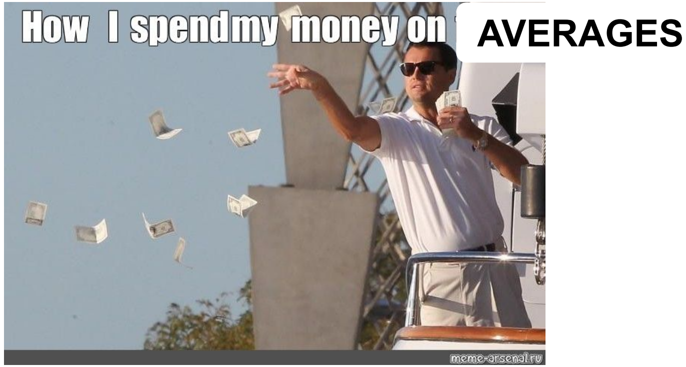
Wait but what about aggregations?

TEKAI

Could we use GSIs? Yes, sometimes

"Every time we need to compute a Brand's average score, we just query all the scores for that brand and compute the average at runtime."

Dynamo charges by queried item size (using a conversion units known as reading/writing capacity units). By querying all of the client's assets, we might end up handling very large items.



TEKAL

Dynamo Streams bring your table alive

Google Cloud Datastream

We use Dynamo Streams to **listen to Create, Update and Delete operations** in our databases. Streams allows us to code automatic actions when a specific update happens.

Update on database

New asset added

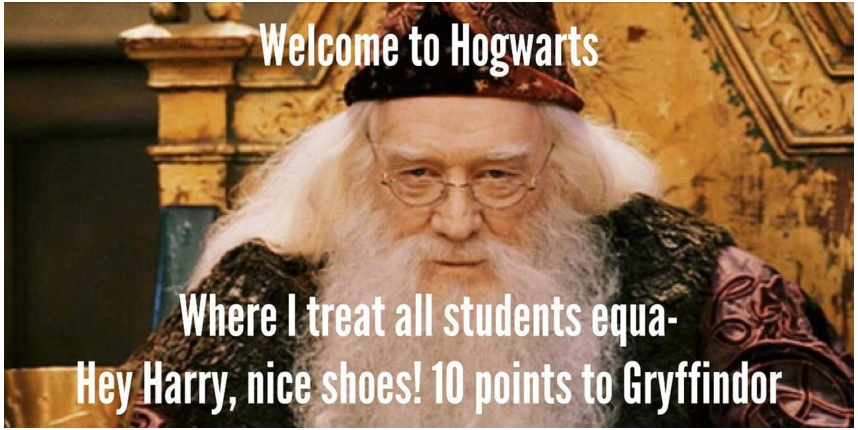
Apply logic

Identify score and client ID

Accumulate score

Add asset's score to client's cumulative score

Stealing from Hogwart's Engineering team



TEKAL

Cool. Can we version control this setup?

Code & Deploy: AWS Serverless Application Model

Ansible

The **AWS Serverless Application Model (SAM)** is a framework for building serverless applications. It provides shorthand syntax to express functions, APIs, databases, and event source mappings. Under the hood, it builds up on **AWS CloudFormation** to deploy entire architectures with YAML and a few configuration files.

Write a blueprint for the architecture

Check modifications to current resources

Launch new resource versions

Code & Deploy: AWS Serverless Application Model

Ansible

Productivity!

Code & Deploy: AWS Serverless Application Model

Ansible

Moving out of AWS User Console and into...

CORS Policies for API endpoints

Environment variables for Lambda Functions

Easy prototyping through repo branches

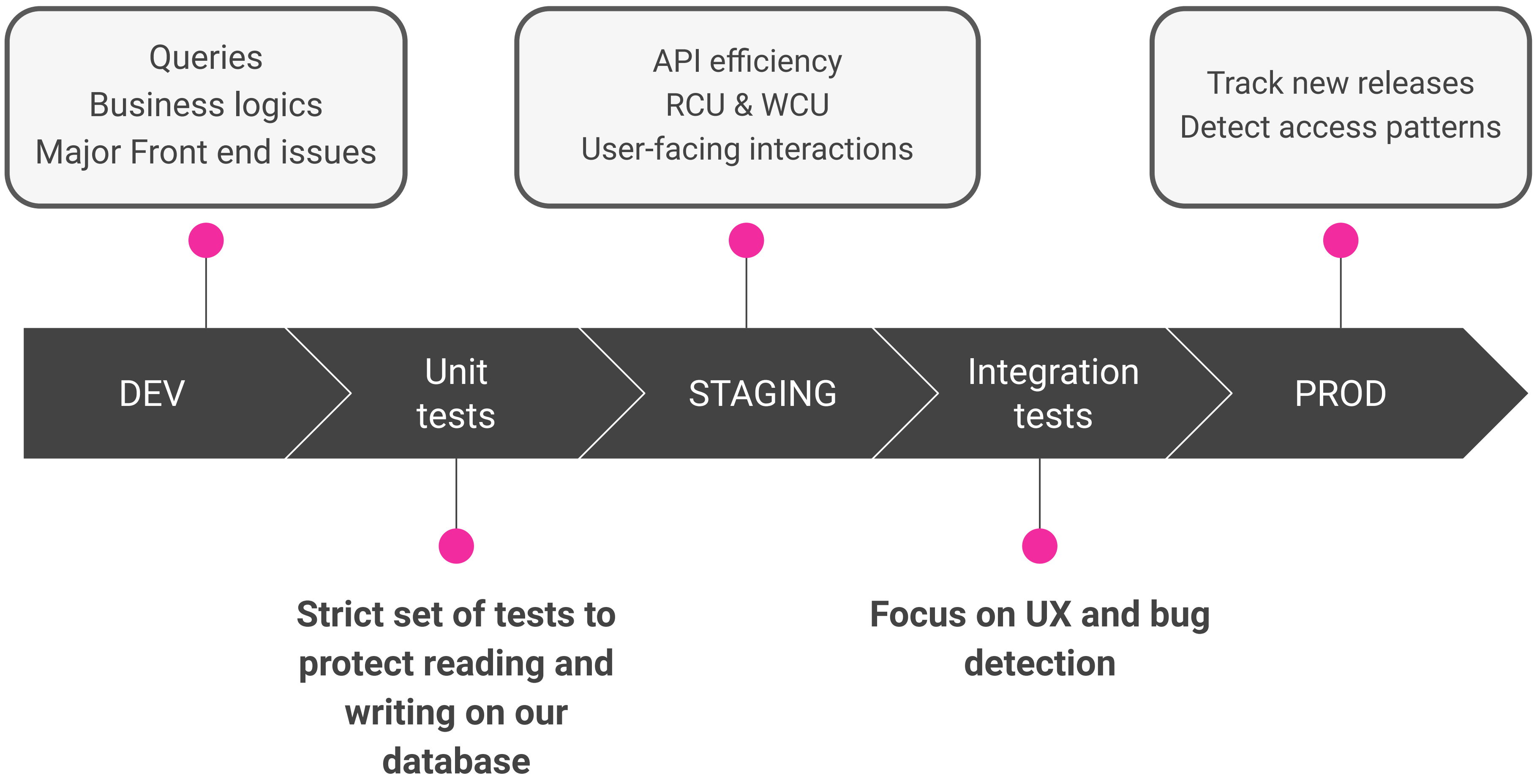
Track all inference containers in a single place

One repository to rule them all

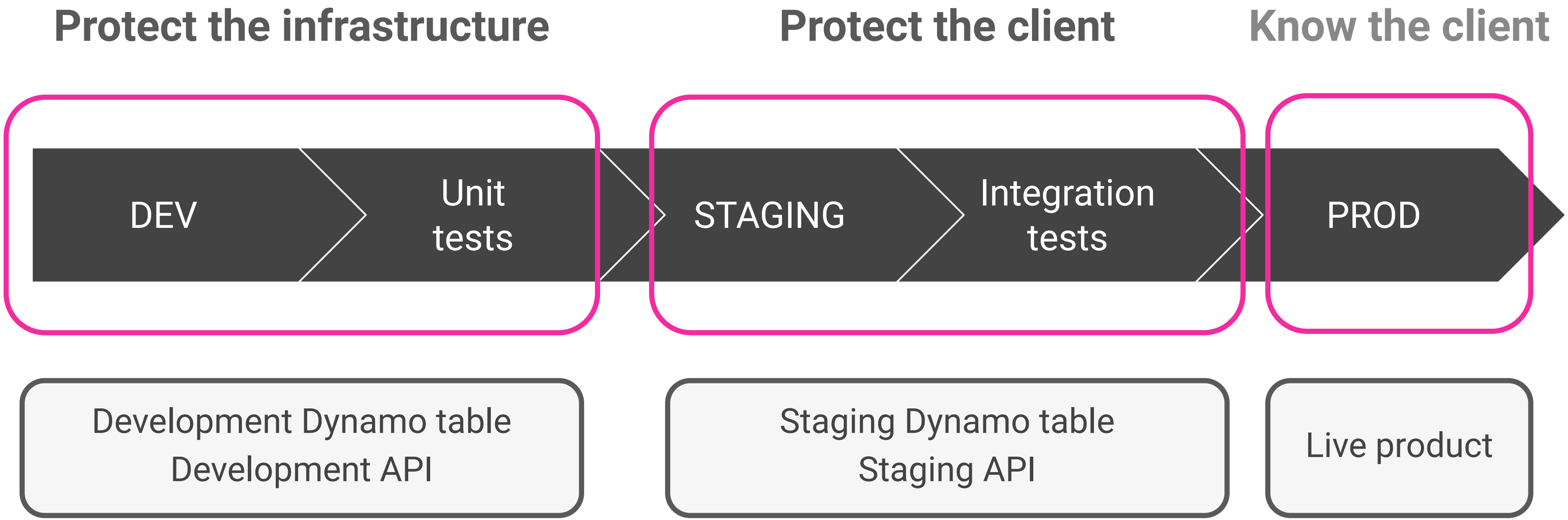
TEKAI

Continuous Integration / Continuous Deploy pipeline

Infrastructure as Code

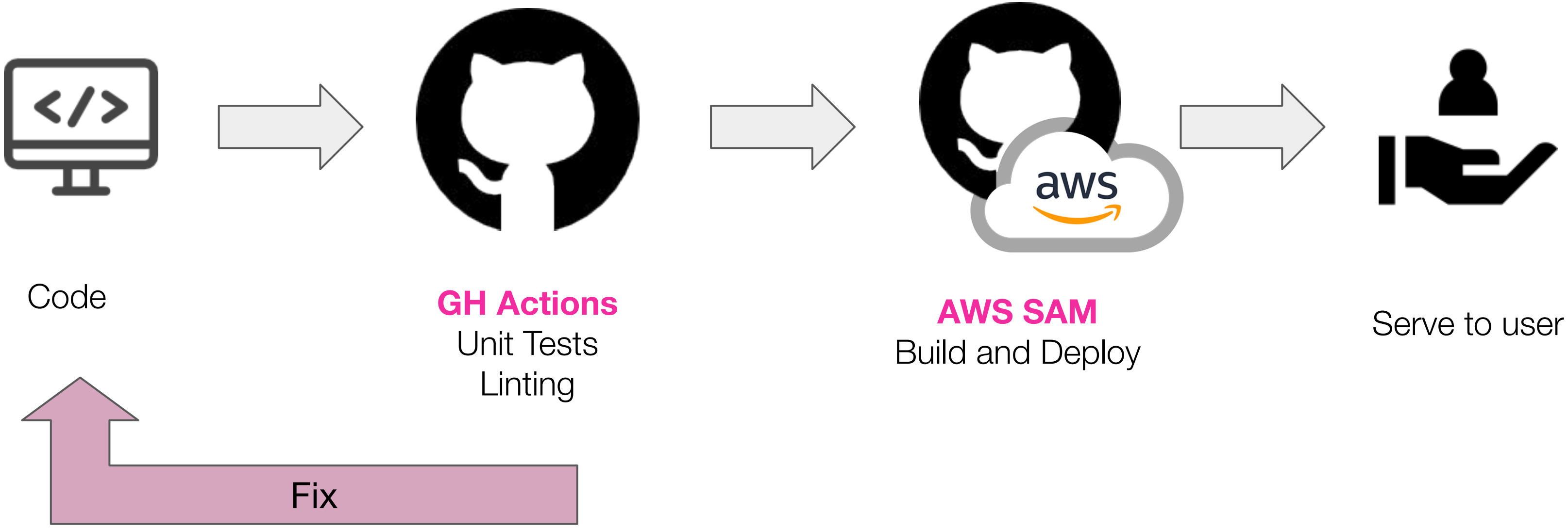

Queries
Business logics
Major Front end issues
API efficiency
RCU & WCU
User-facing interactions
Track new releases
Detect access patterns
DEV
Unit tests
STAGING
Integration tests
PROD
Strict set of tests to protect reading and writing on our database
Focus on UX and bug detection

Our approach

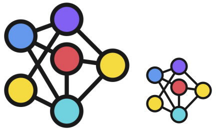


Test: Github Actions

Since implementing Actions, we found a way of **safeguarding our deployed functionalities** and our **development workflow**.

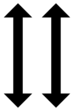


Key tips to build vision products on the Cloud

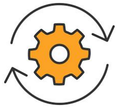


Have model versions that can run on restricted hardware

Containerize



Build a parallelizable pipeline



SAM + CI/CD!

TEKAI

Thank you

A few examples: query with simple primary key

Primary key		Attributes	
Partition key: PK	Sort key: SK		
ORG#BERKSHIRE	ORG#BERKSHIRE	OrgName	SubscriptionLevel
		Berkshire Hathaway	Enterprise
	USER#CHARLIEMUNGER	UserName	Role
		Charlie Munger	Member
	USER#WARRENBUFFETT	UserName	Role
		Warren Buffett	Admin
ORG#FACEBOOK	ORG#FACEBOOK	OrgName	SubscriptionLevel
		Facebook	Pro
	USER#SHERYLSANDBERG	UserName	Role
		Sheryl Sandberg	Admin

PK = ORG#FACEBOOK

TEKAI

A few examples: query with composite primary key

Primary key		Attributes	
Partition key: PK	Sort key: SK		
ORG#BERKSHIRE	ORG#BERKSHIRE	OrgName	SubscriptionLevel
		Berkshire Hathaway	Enterprise
	USER#CHARLIEMUNGER	UserName	Role
		Charlie Munger	Member
	USER#WARRENBUFFETT	UserName	Role
		Warren Buffett	Admin
ORG#FACEBOOK	ORG#FACEBOOK	OrgName	SubscriptionLevel
		Facebook	Pro
	USER#SHERYLSANDBERG	UserName	Role
		Sheryl Sandberg	Admin

PK = ORG#FACEBOOK

AND

SK = ORG#FACEBOOK

Global Secondary Indexes example

PK	SK	BrandID	SectorL3	BrandName
01FFKJST98NHA08S817CB2WPDX	METADATA#CLIENT	49	Skin Care	Biotherm
01FFKJST98MVA17NSHBE382B4B	METADATA#CLIENT	49	Skin Care	Biotherm
01FFKJST98JGJ3EJDKB59EYAMF	METADATA#CLIENT	50	Fragrance	Cacharel
01FFKJST9CSER5FHFP2RKTJPDW	METADATA#CLIENT	51	Hair Care	Elvive
01FFKJST9EJ50KA84JB9R5T7D3	METADATA#CLIENT	51	Hair Care	Elvive
01FFKJST99ZMN4AYMKVZHPE085	METADATA#CLIENT	51	Hair Care	Elvive
01FFKJST9EGT50ZQWGP3BW92N3	METADATA#CLIENT	51	Hair Care	Elvive
01FFKJST99HBV1NRZMTFKWHE25	METADATA#CLIENT	51	Hair Care	Elvive
01FFKJST99VKX4CAT1H7AAA7K9	METADATA#CLIENT	51	Hair Care	Elvive
01FFKJST9DHFZMGTV911JFK0VQ	METADATA#CLIENT	51	Hair Care	Elvive
01FFKJST99KQ63FG9A7BMASPMT	METADATA#CLIENT	51	Hair Care	Elvive
01FFKJST99E7Y1DEGN7Z788V1J	METADATA#CLIENT	51	Hair Care	Elvive

Global Secondary Indexes example

GSICLIENTPK	GSICLIENTSK	PK	SK	BrandID	SectorL3	BrandName
CLIENT#1ff1ef3e-4874-4e92-93b5-bd8b231e057f	BRAND#Biotherm#CAMPAIGN#	01FFKJST98NHA08S817CB2WPDX	METADATA#CLIENT	49	Skin Care	Biotherm
CLIENT#1ff1ef3e-4874-4e92-93b5-bd8b231e057f	BRAND#Biotherm#CAMPAIGN#	01FFKJST98MVA17NSHBE382B4B	METADATA#CLIENT	49	Skin Care	Biotherm
CLIENT#1ff1ef3e-4874-4e92-93b5-bd8b231e057f	BRAND#Cacharel#CAMPAIGN#	01FFKJST98JGJ3EJDKB59EYAMF	METADATA#CLIENT	50	Fragrance	Cacharel
CLIENT#1ff1ef3e-4874-4e92-93b5-bd8b231e057f	BRAND#Elvive#CAMPAIGN#	01FFKJST9CSER5FHFP2RKTJPDW	METADATA#CLIENT	51	Hair Care	Elvive
CLIENT#1ff1ef3e-4874-4e92-93b5-bd8b231e057f	BRAND#Elvive#CAMPAIGN#	01FFKJST9EJ50KA84JB9R5T7D3	METADATA#CLIENT	51	Hair Care	Elvive
CLIENT#1ff1ef3e-4874-4e92-93b5-bd8b231e057f	BRAND#Elvive#CAMPAIGN#	01FFKJST99ZMN4AYMKVZHPE085	METADATA#CLIENT	51	Hair Care	Elvive
CLIENT#1ff1ef3e-4874-4e92-93b5-bd8b231e057f	BRAND#Elvive#CAMPAIGN#	01FFKJST9EGT50ZQWGP3BW92N3	METADATA#CLIENT	51	Hair Care	Elvive
CLIENT#1ff1ef3e-4874-4e92-93b5-bd8b231e057f	BRAND#Elvive#CAMPAIGN#	01FFKJST99HBV1NRZMTFKWHE25	METADATA#CLIENT	51	Hair Care	Elvive
CLIENT#1ff1ef3e-4874-4e92-93b5-bd8b231e057f	BRAND#Elvive#CAMPAIGN#	01FFKJST99VKX4CAT1H7AAA7K9	METADATA#CLIENT	51	Hair Care	Elvive
CLIENT#1ff1ef3e-4874-4e92-93b5-bd8b231e057f	BRAND#Elvive#CAMPAIGN#	01FFKJST9DHFZMGTV911JFK0VQ	METADATA#CLIENT	51	Hair Care	Elvive
CLIENT#1ff1ef3e-4874-4e92-93b5-bd8b231e057f	BRAND#Elvive#CAMPAIGN#	01FFKJST99KQ63FG9A7BMASPMT	METADATA#CLIENT	51	Hair Care	Elvive
CLIENT#1ff1ef3e-4874-4e92-93b5-bd8b231e057f	BRAND#Elvive#CAMPAIGN#	01FFKJST99E7Y1DEGN7Z788V1J	METADATA#CLIENT	51	Hair Care	Elvive

Dynamo Streams example

New asset is **added** for Client A

Client A's cumulative score **+ =** New asset score

Client A's asset **score is modified**

Client A's cumulative score **+ =** (New score - Old Score)

New asset is **deleted** for Client A

Client A's cumulative score **- =** New asset score

Dynamo Streams example

New client is set, now asset belongs to client B

Client's A asset has its **metadata modified**

Client A's cumulative score **- =** asset score

Client B's cumulative score **+ =** asset score

Another approach… why not use GSIs?

"Every time we need to compute a Brand's average score, we just query all the scores for that brand and compute the average at runtime."

- *One very rich and patient Software Engineer*

We resolved the dilemma by comparing **costs**

Dynamo charges by queried item size (using a conversion units known as reading/writing capacity units). By querying all of the client's assets, we might end up handling very large items.

By storing the cumulative counts, we significantly ***reduce query size*** while ***writing operations*** for updates ***are kept to a minimum***, on-demand basis. So far, Dynamo Streams free-tier quotas are more than enough for the short term demands.

Roadmap for enhancing our deployment workflow

LaunchDarkly

Controlled deployment strategies and handle time-to-market more efficiently

LogRocket

Track user behaviour and assess user experience through the user themselves

DATADOG

In-depth monitoring of the application’s execution with detailed tracebacks on exceptions