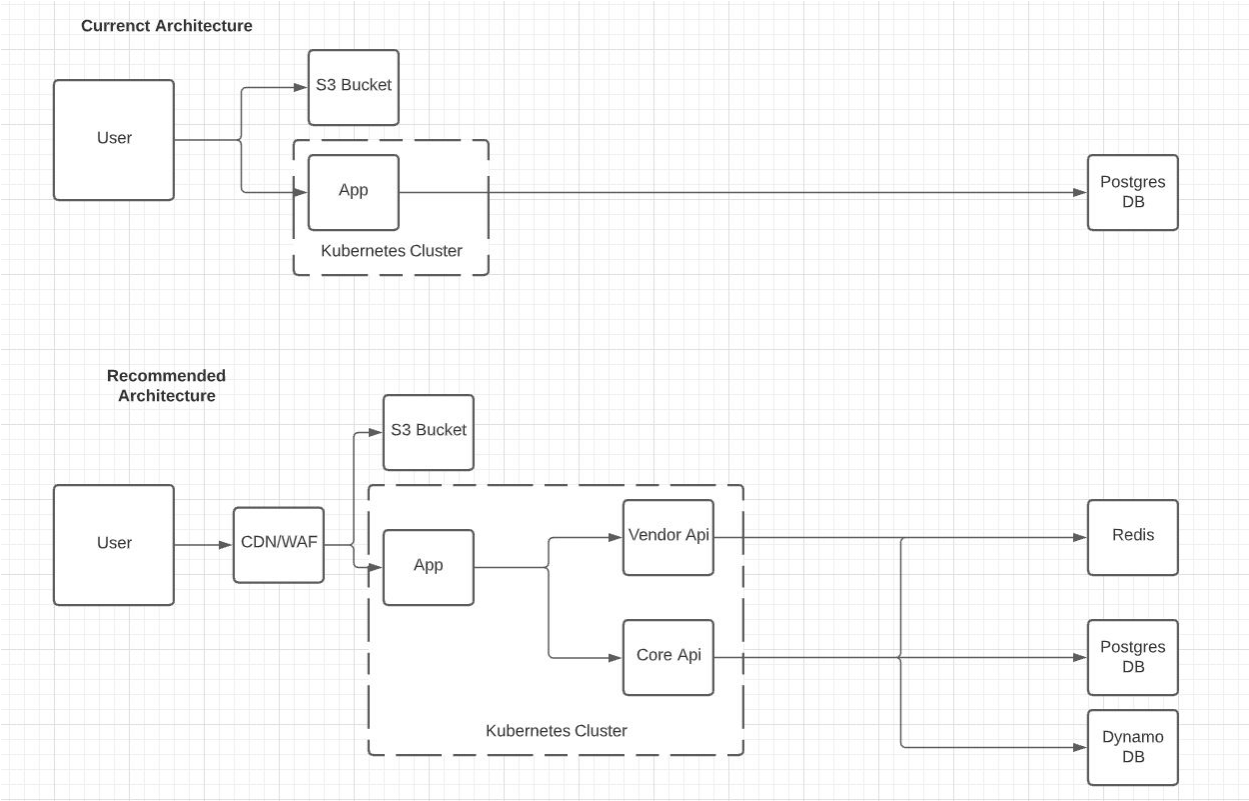
Measurement Mapper Architecture Review



Existing Architecture and Assumptions
User clicks survey via dropdown menu, this is regarded as the entry point to the specific survey, measurement and user data. It is assumed that other pins on the same map belonging to other surveys are not visible even if they are disabled. When a user clicks the pin a modal is presented with measurement and user information.

The aggregation of the three Apis (survey, measurement and user) occurs at the frontend level by calling the measurement App in Kubernetes This is probably because there was a given business requirement and a deadline to create a quick minimum viable product. This is acceptable because as companies and new products start off the business requirements are generally very fluid and developers do not wish to lock themselves to a multi-tiered architecture which could prove cumbersome to change during the development lifecycle.

However as the company and related applications moved to enterprise grade, it becomes increasingly difficult to scale both the infrastructure and the developers and thus a tiered architecture must be considered.

Improved Architecture
The improved architecture aims to move the aggregation of surveys, measurement and user apis to the backend with a tiered microservice approach. There are multiple benefits to this

approach including business, technical and social which are explored in the following improvements divided into performance, security and developer experiences. The App will merge vendor specific survey data (From Vendor Api) with that of the survey data and related measurements and users (From Core Apis). The SQL database via ORM (object relational mapping) framework will have the constructed survey specific data from Postgres, the Core APis need to map the metadata from DynamoDB back into the ORM entities since the metadata data will be migrated out of Postgres and into NoSQL. Frontend will be responsible for handling the data fetch when a user clicks on the survey dropdown which displays a modal when a pin is clicked on and other presentational aspects of the app. Design choices are explained in the improvement sections.

The assumption for a tiered architecture critically assumes the company is moving toward an ambitious plan to rapidly expand its product lines that require a common layer of business logic and data at the quality expectation of enterprise.

Then tiered architecture at the microservice level is the correct approach however if a company is near the beginning of product development, the business requirements often shift and may slow down the actual process of development during the experimental stage of a MVP and related products.

The endpoints exposed at the application layer should be one endpoint that returns a specific data related to a single survey and an endpoint that fetches all available survey Ids and aliases for a given map. For the mutation endpoints the POST endpoints for measurement, survey and user can remain the same but proxy from app down into the core api as these endpoints will most likely be shareable across products.

```
// Endpoint: Get Endpoint /surveys (For App and Core Api)

// Description: Frontend calls app to provide a list of surveys. As we do
not need to display

export type SurveyOptions = {
     list:  SurveyOption[];
}

export type SurveyOption = {
  id: number;
  name: string;
}
```

```
// Endpoint: Get Endpoint /surveys/:surveyId (For App and Core Api)

// Description: When the user selects a specific survey in the dropdown,
the frontend will call this endpoint to fetch survey specifics. This data
is used to give the frontend the measurement pins to display on the map.

export type Survey = {
  id: number;
  name: string;
  type: SurveyTypeEnum; // Have enum of Unknown as default
  meta: string;
  measurements: Measurement[];
}

export type Measurement = {
  id: number;
  name: string;
  shape: string;
  meta: string;
  user: User;
}

export type User = {
  id: number;
  name: string;
  email: string;
  meta: string
}
```

Performance Improvements

The existing architecture aggregates the results of survey, measurement and user APIs at the frontend, unfortunately this will have performance issues later on and result in a poor user experience especially those with older mobile technologies. Since the frontend React, it can be optimised. Components can be memoized where given props/state will be remembered and the result cached preventing a rerender. This is effective when the same survey is reloaded multiple times but if a user toggles between surveys (back and forth) it becomes less useful as memoize only caches the last set of props/state. PureComponents cache all props/state but memoize can target specific props/state to produce the same caching result. Assuming some of the frontend code has a deeply nested tree of child components, memoize can save unnecessary child rerenders.

For the backend specifically the measurement app, its components should be divided into several microservice tiers. As can be seen in the above diagram. The App (In this case the measurement mapper) should be a very light node application and contain minimal business logic, its role should be more to give the react frontend constructed view models and receive requests to aggregate data from the vendor and core API. Splitting the microservice allows for Kubernetes to scale different microservice tiers that require more compute. For example the Core API can be scaled independently of the app, this could be because other apps require the same Core APIs more frequently or the vendor api may scale for asynchronous processing.

The existing architecture fetches everything into the frontend including data not related to the specific chosen survey, this will eventually cause slowdowns as the database gets larger and latency will be increasingly more noticeable to the user. Instead of fetching all survey data at once, only related measurements and the specific survey record will be fetched from the backend. Dropdown clicks are a relatively slow process requiring the user to click and read through the dropdown, this is better than a long loading data initialisation process when loading the map for the first time which exists in the current architecture since all data is retrieved at this time.

The database is Postgres and with the new architecture we can keep the database as Postgres, however currently there are no relationships so this needs to be corrected to leverage SQL. A one to many relationship should be made between the Survey table and the measurements table with a many to one relationship with the measurement and user table. This can be done by adding a foreign key constraint. Fixing the primary key type mismatch in the survey table to INT and the necessary constraints such as not nullable and UNIQUE. Since survey_id in the measurements table is INT we can assume that the primary key in the survey table can be cast as an INT, the primary key of the survey table should be changed to INT so as to not break referential integrity of the Id field (survey table) which is a string which may not able to be cast as an INT (survey_id in measurement table). Sequential keys that are shorter also improve clustered index performance rather than unique random strings. All Primary keys should be auto incremented and managed by SQL.

Having the SQL Database filter for the related measurement records for a given survey is more efficient that having the frontend constantly filtering from a very large list of measurements, as the business grows this becomes less scalable and from a user perspective, loading times will become increasingly frustrating especially when they toggle between surveys. While it is possible to filter measurements for a survey in NodeJS, the entire set of measurements linked to multiple surveys will have to be transferred over the wire which will increase latency. The filtering should be kept at a database level.

Performance can be enhanced by adding an index to the survey table. While NoSQL databases offer speed increases over SQL databases, they become difficult to manage when complex relationships form when more product features are added, it is in this author’s experience to leverage both SQL and NoSQL in a hybrid model together storing the meta column with JSON data in NoSQL but maintaining the table measurement relationships with survey and users in a

SQL database. SQL databases can only scale vertically or horizontally with the more difficult process of sharding while NoSQL scales horizontally but NoSQL has the limitation of data duplication. The hybrid model is chosen for the improved architecture with AWS DynamoDB for NoSQL and Postgres for SQL to spread the load between both SQL and NoSQL, this assumes the metadata will become increasingly large over time. For performance the primary key for dynamo should be a randomised Guid and not an incremental type like number or date, this will cause a hot zone at the partition level.

The existing implementation calls a third party to determine what kind of survey the survey is. We make an assumption that each vendor call costs money and that the survey type can be cast as a survey type enum. The business logic and third party call client should be placed within the Vendor Api microservice. Since the third party integration costs money, to reduce load times on the user, the calculated survey type should be persisted to the database as an enum as shown above. If vendor calls are free then a queue service could batch process archived surveys after hours and persist the surveyTypeEnum to the database. Similarly a new survey event could invoke the vendor call asynchronously and combine with the business logic to persist into the database. The most common scenario would be to asynchronously process new surveys and have archived surveys processed on the fly but with the survey type persisted into the database afterwards. The database should have a surveyTypeEnum of Pending if a vendor call out has not been made.

The Core API can cache results that have a low probability to change with high usage into a redis cluster such as fetching the list of survey options (see above type), this is not likely to change often so having this cached into redis can offload compute from the database and improve performance. It will also depend based on business requirements and user behaviour with surveys if it's worth caching other objects.

The metadata can be transferred to the frontend as a JSON string, since it is unstructured and it is assumed we are able to normalize this data. JSON parsing in the frontend is more performant than instantiating a javascript object literal especially for large objects.

Security Improvements
Security is often considered last but as an enterprise grows they become a larger target for security intrusions.

The existing cookie authentication may be susceptible to CSRF attack, switching to storing a JWT stored at localStorage mitigates this issue but presents XSS vulnerability since any javascript can access this including third party libraries. The safest would be to store the JWT in frontend memory however this presents its own challenges since closing the page the user is automatically logged out and provides a bad user experience. For the new implementation a server side cookie should be used with an embedded JWT with all routes to validate this JWT, as per existing implementation the routes will validate the permissions and ids of the user, the new implementation has these reside in the JWT as it is signed and can be trusted. To prevent CSRF the cookie needs to have the SameSite attribute set so only requests made under the

same domain will have the cookie sent with it, the cookie will also need HttpOnly so javascript on the browser may not read the cookie to prevent XSS since the cookie contains the access token and finally the cookie should be set with the secure flag so that it is sent over HTTPs only so it is not visible as plain text. This practice provides a secure way of managing access and privileges without compromising user experience as the cookie persists when the browser is closed. The react frontend will require the permissions and ids of the user since it cannot access the server side cookie due to the httpOnly attribute, once authenticated the server side cookie can be used to fetch this information via a protected route and stored in localStorage. Having the JWT inside the cookie also means session information such as userId need not be stored in the backend reducing complexity for developers and improving performance.

Server side cookie implementation is for the app microservice which faces the frontend, for communication between microservices the same JWT can be used allowing for better logging and acting as a trace token as the request cascades down into the tiered microservice layers.

Having the Vendor API and Core API at lower tiers will improve the security posture, Vendor API because often exposing a public vendor token at the frontend level allows users to clearly see the token, take it and perhaps piggyback off the usage of the vendor (in this case, it is the survey type checker endpoint). Vendor tokens and datastore credentials can be stored securely encrypted in Hashicorp Vault or AWS Kms encrypted s3 Bucket, only core Api and Vendor Api will be allocated the appropriate security groups (Containers having different Security groups is supported in Kubernetes) to gain access to this infrastructure.

Instead of the user directly accessing kubernetes a CDN such as AWS cloudfront can act as a reverse proxy with AWS WAF (web application firewall) can be used. S3 static assets such as images can be cached on cloudfront instead of constantly fetching from s3. Backend Get requests could be cached with CDN but generally no caching is recommended but with backend requests still going through the WAF. Firewalls could be implemented at the kubernetes level such as via Nginx/F5 or with the WAF attached directly to the load balancer, but it is in this author’s experience that having a third party CDN manage this aspect can free maintenance of those rules and instantaneously scale during a denial of service attack. At a local kubernetes level, these WAFs may become overwhelmed causing an intermittent user experience. Akamai CDN provides a stateful management of its rules meaning Cross site scripting, sql injection and other attacks are tallied together before a decision to block is made, this drastically reduces false positives reducing providing a better user experience whilst maintaining security.

Developer Experience Improvements
Every technical decision has a social and real financial cost and the business needs to balance the needs of its employees as well as its own requirements.

An ORM (object relational mapping) framework such as sequelize for nodeJs improves the simplicity of database schema management. Table relationships don't have to be created through sql but with sequelize, instead as code. An ORM does not necessarily improve performance though and should be considered for the developer experience. The Sequelize

framework will be installed on the Core Api service. Sequelize syntax is much more readable than large SQL transactions.

At the individual service level having onion architecture improves readability and predictability of where components should be. For example In The Core API, we implement sequelize and redis caching in the infrastructure layer since it is data access related, the business logic can go in the application layer such as checking with the infrastructure for caching. Caching a list of surveys would make more sense than caching survey details which are subject to change from user mutations. Routing will be located in the Controllers layer, having a top down approach from controller to application to infrastructure mitigates circular dependencies and improves readability. Developers will see each service layer and automatically know how to find code based on the experience of working with one service. Having layering at the service level and tiers in the microservice layer makes code placement predictable and allows developers to focus on writing logic. For a more complex domain, the application layer can be broken into application and domain, where application has targeted app logic with domain have reusable enterprise logic.

Multi-tiered microservices improve development by separating concerns between the microservices, but it also improves testing as responsibilities like Vendor call outs and Core Api call outs can easily be mocked during testing. In this way when vendor specifications change or even database technologies change these only need to be changed at the lower microservice tiers instead of the upper microservice tiers. This improves the developer experience since the contract between App and lower tier APIs can generally be kept and tests will not need to be duplicated or rewritten when external services specifications change. While the current strategy is a 2 layer tier for microservices, when other applications are considered more layers may need to be added. Each layer adds its own code complexity and network latency, so having appropriate enterprise wide discussions is important, requirements should be gathered for each app and open contribution and concerns should be embraced to create a set of core APIs. Implementation of a contract via typescript and not vanilla javascript for the microservices will greatly improve code readability for developers, assuming this is not implemented already in nodeJs.