SCTE CABLE-TEC EXPO 22
SEPTEMBER 19-22 • PHILADELPHIA, PA

2022 Fall Technical Forum
SCTE • CABLELABS • NCTA

Translating Customer & Employee Experience with Shaw's Data Journey

A Technical Paper prepared for SCTE by

Greg Bone
Principal Architect
Shaw Communications
2400 32 Ave NE, Calgary, AB T2E 6T4
greg.bone@sjrb.ca

Goutam Agarwal
Principal Enterprise Architect
Shaw Communications
2400 32 Ave NE, Calgary, AB T2E 6T4
goutam.agarwal@sjrb.ca

Table of Contents

List of Figures

1. Introduction

Through industry partnerships and customer connects, we have pursued insights on how the pandemic has fundamentally shifted consumer and worker behavior, exploring the increased importance of engagement in a world of remote work, and touchless experiences. We discovered a significant shift in customer expectations and behaviors: customers have moved beyond speed and are looking for a unified and simple experience with channel of choice. In a post-pandemic world, preferences have shifted to more personalized, convenient, and connected experiences.

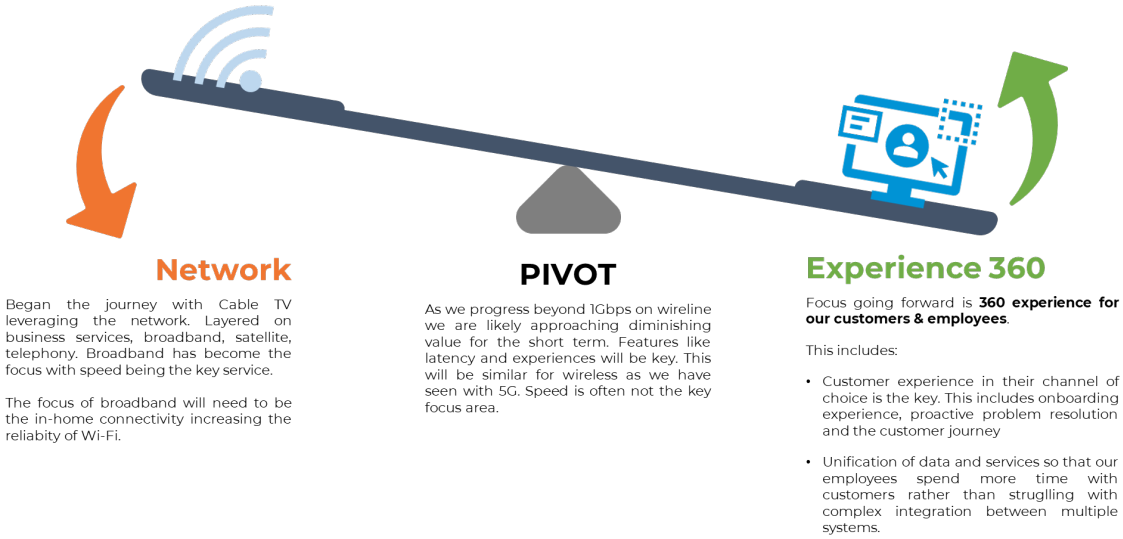


Figure 1 - Pivot in Customer Expectations

A unified channel touchpoint and agent experience is central to providing a connected customer experience. Because agents are behind the various customer interaction channels, their experience is the most crucial factor in meeting these quickly evolving customer expectations. To obtain a 360-degree view of the customer, agents have traditionally needed to access information from multiple systems and applications. However, this “swivel chair” experience was inefficient and impacted call times, wait times and general customer experience. Something had to change.

A Unified Customer Platform (UCP) has since been designed to empower agents through seamless visibility to any Customer in one place – no more swivels. UCP provides an accurate and centralized look at all the services a customer subscribes to, across all lines of business. Available 24/7, UCP visualizes customer and service information (using simple but intuitive UIs and modern search capabilities), allowing sales and support teams to quickly understand the customer is subscribed services. It also serves as the foundation to create customer communications and digital journeys, automated marketing and sales campaigns, enriched customer segmentation and enhanced reporting.

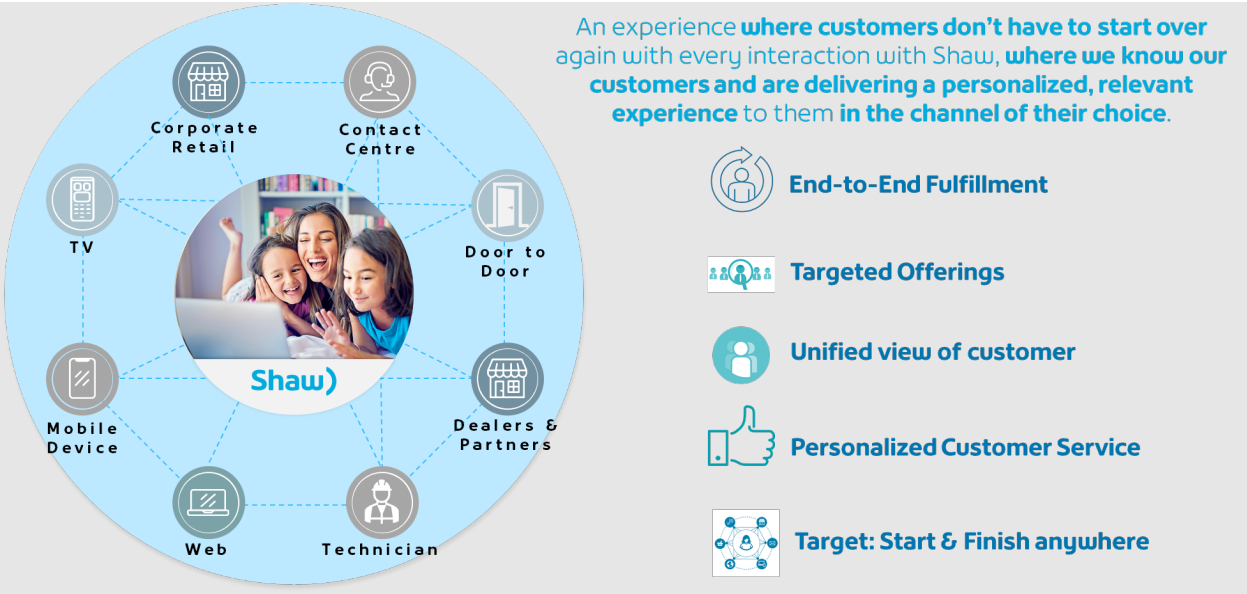


Figure 2 – Shaw's Customer Experience Strategy

In this paper we introduce the Unified Customer Platform (UCP), a modern platform that helps Shaw identify customers who meet eligibility requirements for new product offerings.

Figure 3 - Shaw's Agent Experience Strategy

Eligibility requirements often require taking a 360-degree view of the customer across multiple lines of business to highlight the products and services that are a good fit. This analysis leads to cross sell/upsell opportunities and would typically take place outside of a source system with the data analysis occurring in siloed data sets. Centralizing customer information (account information, subscribed services, history) in

UCP creates a single-entry point for reviewing cross-sell/upsell opportunities. Accessing customer information in UCP via an application programming interfaces (API) or a Google-like search interface was an important design decision. This will be the first time Shaw has had access to a single platform capable of retrieving customer information for all our customers across all lines of business.

Overall, the care teams using UCP are incredibly happy with the platform. The consolidated customer information helps reduce the "swivel-chair" previously required to jump between different billing systems. It provides them with the consolidated account details to best help support the customer base. The key findings that emerged from building and deploying UCP are:

1) Customer care teams benefit from platforms that aggregate account data from multiple source billing systems. We saw good adoption from Wireless Care teams who needed to find authorized users listed on a Wireline Internet services account.
2) Searching for customer accounts using the Elasticsearch search engine is a comfortable and time saving feature. Expanding customer search to include multiple service addresses and other business-class? options has been collected.
3) Modern technology stacks inside public clouds like AWS help reduce the overhead of building a new, common data platform and can deliver extreme performance at reasonable costs.
4) The lack of real-time data feeds into UCP was a constraint placed on us because of how data gets batched and scheduled upstream. For key data attributes like upgrades to new services, care teams would like to see an indicator that something is pending rather than no data.

Our intent is to continue to develop the UCP platform as an aggregation platform for a variety of new use cases that reach beyond care teams. Also adding additional account details like pending upgrades and customer interactions would help paint a better picture of our customers. Migrating some of the data loads into event-based streams could also help reduce the data load lag times caused by batched daily loads of data.

This paper will be organized as follows: we begin by introducing some of the drivers that helped ground us on why Shaw pursued building UCP in the first place. We will then present an overview of UCP and how it is being used to improve customer experiences. This is followed by the platform architecture and design challenges. We conclude the paper with a look at UCP's journey so far, and likely future states.

2. Key Drivers

The key drivers to provide a personalized, convenient, and connected experiences to our customers started with the rebranding of the Shaw wireless product suite. If the Unified Customer Platform helps with selling Shaw Mobile, it could be used for other customer offers in the future. The relatively short project timelines had us looking at ways to modernize the technology stack so that we can focus on building business value without the incumbrance of managing infrastructure at scale.

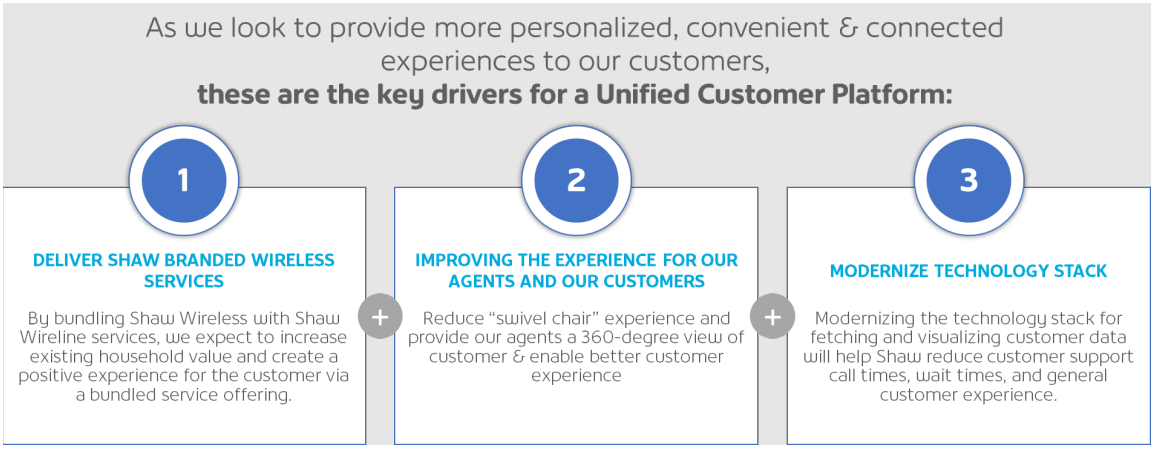


Figure 4 – Key Drivers for UCP

- Shaw wants to deliver a new Shaw-branded wireless product in Western Canada. By bundling Shaw Wireless with Shaw Wireline services, Shaw expects to increase existing household value and create a positive experience for the customer via a bundled service offering.
- Having a complete picture of the customer will provide a better agent experience and improve Shaw's ability to support the current customer base, improve upsell recommendations, and maximize the effectiveness of marketing campaigns.
- Modernizing the technology stack for fetching and visualizing customer data will help Shaw reduce customer support call times, wait times, and general customer experience. Specifically, we were interested in how a query language for your API (GraphQL, graphql.org) can potentially create a more consumable API when compared to traditional RESTful (REpresentational State Transfer) API's.

3. Overview of Unified Customer Platform (UCP)

The Unified Customer Platform enables searching for both residential and business customers across all lines of business using a dedicated search engine. Work was done to build data pipelines that aggregated this fragmented data into a single search index. Searching against this index matches your search terms against customer names, account numbers, phone numbers, and email addresses. The search results contain a list of customers, ordered by relevance, that match or partially match the search criteria along with all the associated child accounts for each customer. Why does this matter? UCP provides a single view of the customer instead of an account-centric view that was previously spread across five different source systems. Knowing the customer from an aggregated data perspective provides an enriched customer interaction, creates cross/up sell opportunities, exposes data insights to better serve the customer and allows the right offers to be recommended at the right time.

We felt it necessary to build this complete picture of the customer across three distinct layers. 1) a relational data set that can support reporting and marketing campaigns. 2) An API layer that functions as a data fetching API and gives clients the flexibility to select what data they need with minimal transformations. 3) And because agents need to interact with this data while assisting customers, it was necessary for us to build a new front-end to search and see all the customers data on one screen.

The Unified Customer Platform can be divided up into the following components:

- A front-end user interface that supports customer search and viewing customer details.
- An application programming interface (API) layer that uses the GraphQL to control exactly what data you get back from the API.
- Local data stores that optimize data fetching and retrieval.
- A data transfer layer that handles loading of the local data stores.
- A cloud data warehouse that aggregates customer data.
- A master data management (MDM) platform that uses a set of business rules for linking customer accounts across 5 incongruent billing systems.

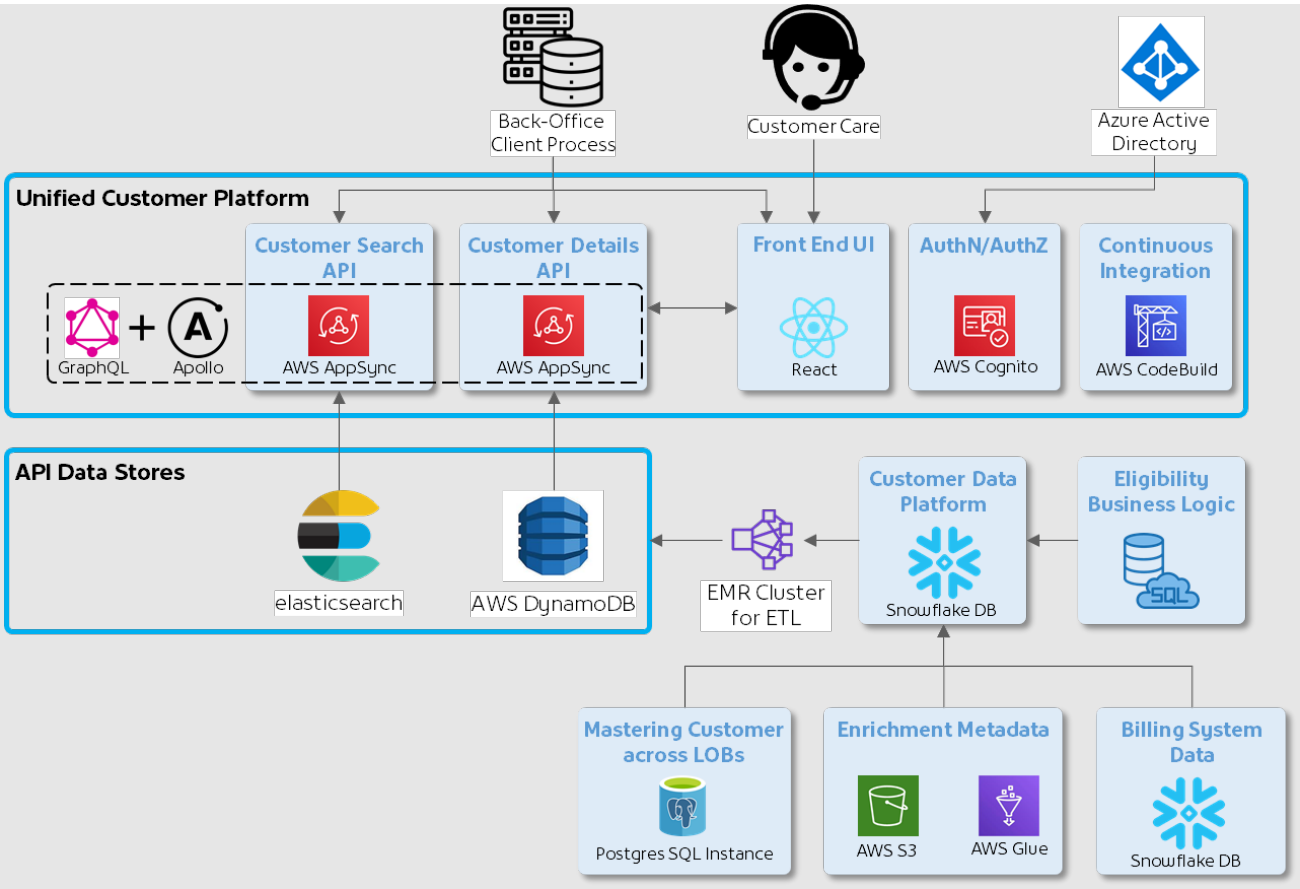


Figure 5 – High Level System Overview

4. Guiding Design Principles

When we sat down to develop this new platform, we were getting several different requirements from different teams. The scope of potential end-users was broad and included Back Office teams, Care teams, Retail support, Marketing, Finance, Revenue Assurance, Technical Operations, and corporate stores across multiple lines of business.

Some teams don't typically interface directly with the customer and have more time to perform troubleshooting steps. Teams like the Back Office, Marketing, and Finance are good examples of teams disconnected from the customer experience. Other teams like the technical operations and retail support provide the necessary support but operate under different expectations for a timelier resolution of issues. And then there are the customer facing teams like the care teams that operate the call centers and the retail store reps. The customer facing teams often need to navigate systems and applications while talking to a customer.

To avoid feature bloat and build a usable product, the design and development teams grounded ourselves on a few key principles. These include:

Basecamp Like Delivery

- Structure work and teams into cycles that last six weeks. We experimented with three, two-week sprints and two, three-week sprints. The number of actual sprints and length of sprints could vary depending on the type of work. But roughly every six weeks we wanted to finish a batch of product work and start preparing for a new batch of work. We were largely influenced by how Basecamp (Cisco, 2018)[1] structures their work.

Right Sizing Teams

- Team sizes are kept small. A team is two or three people that is dedicated to a portion of the product. We had a 2-person team for the UX design, a 2-person team for the AWS infrastructure, a 2-person team for the front-end, and a 2-person team for data engineering work. In a 6-week cycle, team sizes could flex up or down depending on the scope of work that needed to be done.

Prioritize Based on Value

- We could not do everything that we wanted to do and do it well. We did not have the time, resources, people, etc., so we prioritized the features to make the minimal viable product that executes on a few things and does them extremely well. The top requested feature was the ability to view account details for wireless and wireline to see if the account owner or authorized users on the account are eligible for promotions or offers.

API-First Approach

- Employ an API-first approach to building products. An API-first approach means that for any given development project, the API's are treated as "first-class" citizens. The API's can be consumed by both the client applications, other system platforms, or other development teams. Therefore, the API's need to be designed in an intuitive and reusable manner.

5. Minimum Viable Product Scope

The initial set of users for UCP was reduced to the care teams, marketing teams and back-office teams. Having a targeted user base helped us get both timely feedback and a reduced set of requirements plus allowed the development teams to course correct between sprint cycles.

The search feature was limited to customer names, account numbers, phone numbers, and email addresses. Searching by customer name needed to allow for partial matches on either the first name or last name.

Account details needed to include total monthly revenue totals by line of business, identification of the different product subscriptions, how long has this person been a customer, account status, payment status, and any authorized users on the account. Address information for billing and service addresses was also needed.

The unified customer channel needed to include three distinct access patterns for fetching customer data:

- A relational data warehouse that functions as the foundation for data aggregation and enrichment of customer data that can be accessed using Structured Query Language (SQL)
- A GraphQL API that provides some flexibility for the client to choose what data gets returned
- A React Web Application that uses the GraphQL API to efficiently fetch customer data

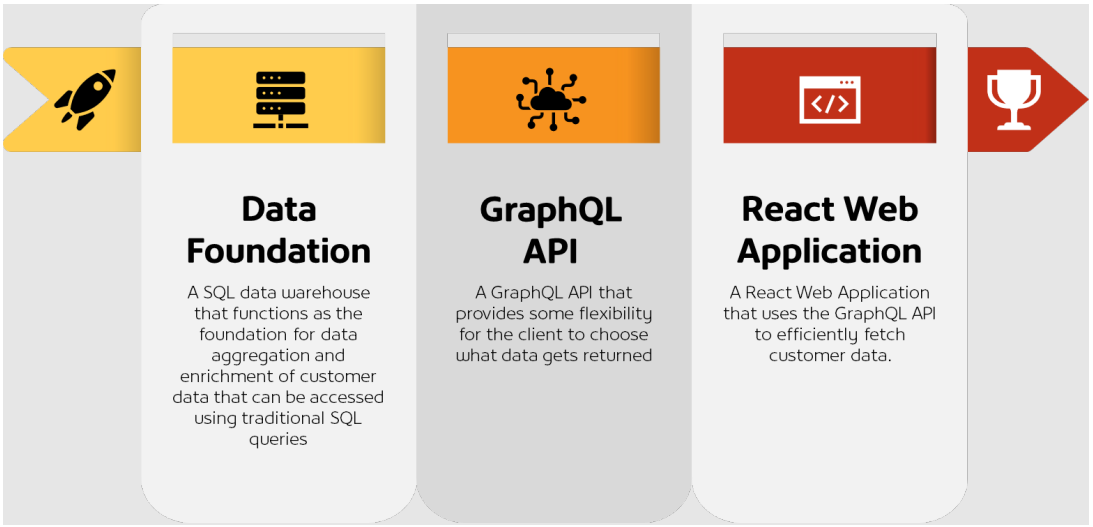


Figure 6 – UCP - Minimum Viable Product Scope

6. Components

6.1. Customer Search

The goal was to develop a single customer search endpoint that would search across the entire customer base in a performant manner. To optimize search performance, we are using Elasticsearch, a distributed search and analytics engine.

Initially, we considered using traditional relational database engines for search, but the search performance that we got from Elasticsearch was far superior to that of a Postgres database. In addition, we also had plans to take advantage of Elasticsearch features for including alternate spellings and nicknames for a customer's first name and addresses. Overall Elasticsearch felt like the best tool for the job.

We wanted searching to be simple, intuitive and with minimal navigation. We evaluated different design options and preferred a single search bar component with some place holder text that tells the user what to enter for search terms.

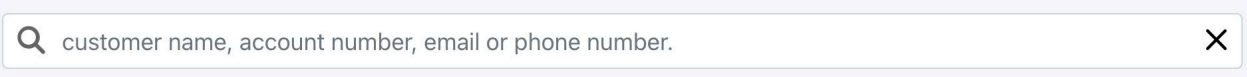
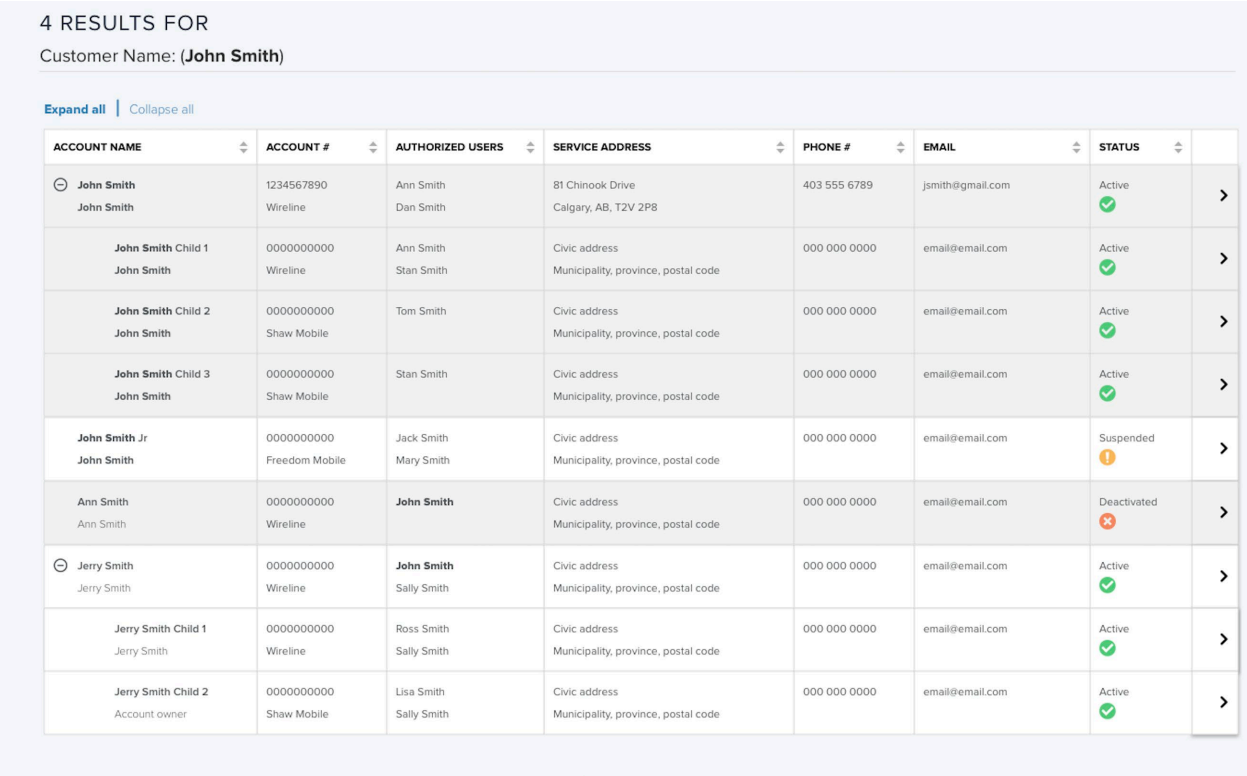


Figure 7 – Top Search Bar

Searching across residential and business accounts is the default without a need to provide any extra context. We also wanted the search engine to support partial matching of customer names and email addresses.

6.2. Search Results

Search results needs to show the customer, and all their associated linked accounts. (The linking of accounts will be covered later under the Customer Account Linking section.) When accounts are linked, there is one account that gets designated as the primary account. The rest of the linked accounts are considered child accounts. Child accounts can and will span multiple lines of business and be sourced from different billing systems.



4 RESULTS FOR

Customer Name: (**John Smith**)

Expand all | Collapse all

ACCOUNT NAME	ACCOUNT #	AUTHORIZED USERS	SERVICE ADDRESS	PHONE #	EMAIL	STATUS
John Smith **John Smith**	1234567890 Wireline	Ann Smith Dan Smith	81 Chinook Drive Calgary, AB, T2V 2P8	403 555 6789	jsmith@gmail.com	Active
John Smith Child 1 **John Smith**	0000000000 Wireline	Ann Smith Stan Smith	Civic address Municipality, province, postal code	000 000 0000	email@email.com	Active
John Smith Child 2 **John Smith**	0000000000 Shaw Mobile	Tom Smith	Civic address Municipality, province, postal code	000 000 0000	email@email.com	Active
John Smith Child 3 **John Smith**	0000000000 Shaw Mobile	Stan Smith	Civic address Municipality, province, postal code	000 000 0000	email@email.com	Active
John Smith Jr **John Smith**	0000000000 Freedom Mobile	Jack Smith Mary Smith	Civic address Municipality, province, postal code	000 000 0000	email@email.com	Suspended
Ann Smith Ann Smith	0000000000 Wireline	**John Smith**	Civic address Municipality, province, postal code	000 000 0000	email@email.com	Deactivated
Jerry Smith Jerry Smith	0000000000 Wireline	**John Smith** Sally Smith	Civic address Municipality, province, postal code	000 000 0000	email@email.com	Active
Jerry Smith Child 1 Jerry Smith	0000000000 Wireline	Ross Smith Sally Smith	Civic address Municipality, province, postal code	000 000 0000	email@email.com	Active
Jerry Smith Child 2 Account owner	0000000000 Shaw Mobile	Lisa Smith Sally Smith	Civic address Municipality, province, postal code	000 000 0000	email@email.com	Active

Figure 8 – Search Results page variation where "Service Type" is shown with text in the Account # column

By clicking on one of the arrows on the right side of the page, the user will navigate to a customer details view.

6.3. Customer Details View

The customer details page provides a summary of accounts by line of business. Each line of business is reflected in a separate tab and has a summary of services, date connected, total number of accounts, the total revenue for all services, and whether the customer is in the mobile network and is eligible for a wireless bundle.

Below the tabs, there is an account summary section consisting of the account number, authorized users, revenue per account, source billing system, and account status/payment status. Below the summary are the account details that includes billing and service addresses and the details for the distinct types of services.

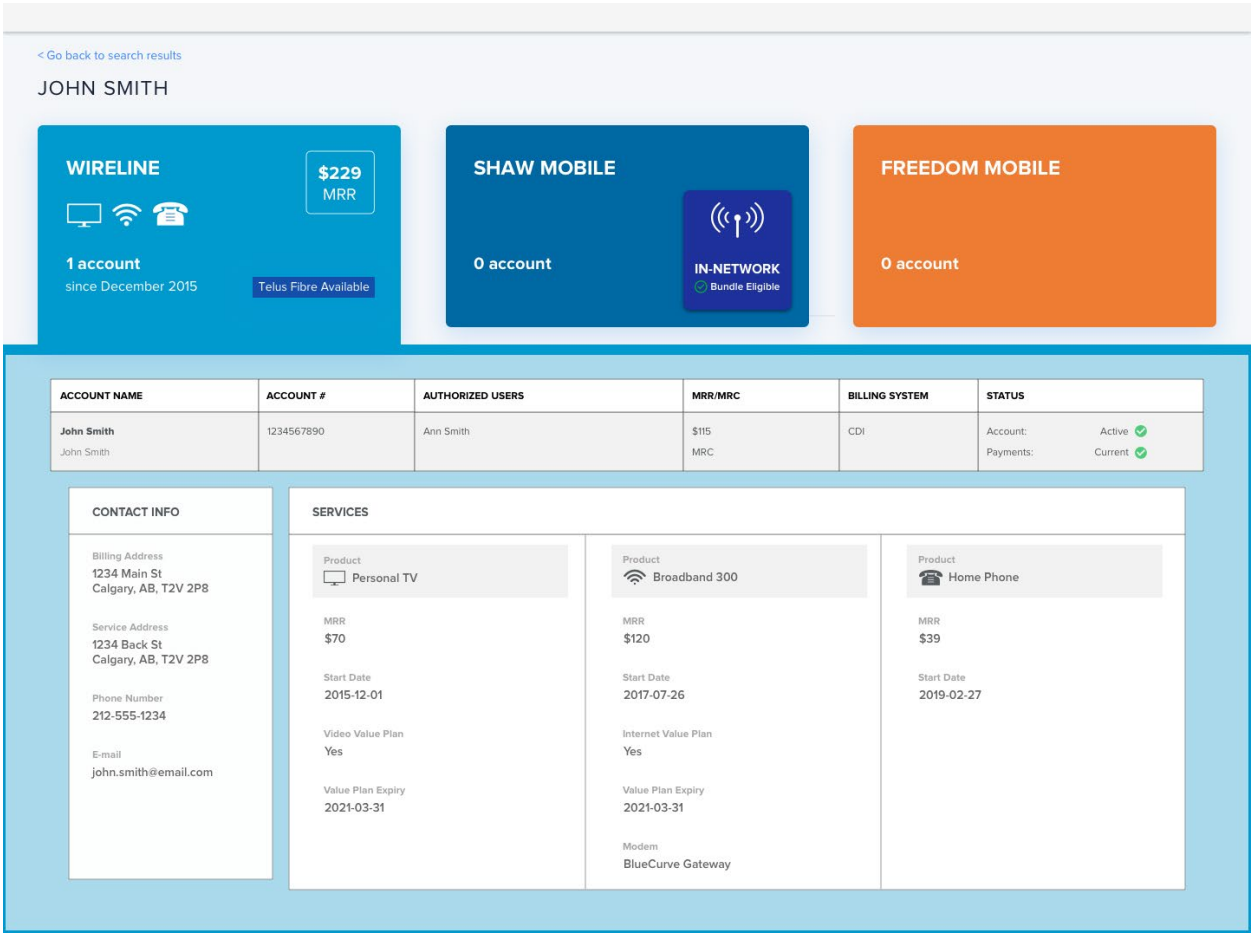


Figure 9 – Wireline, no mobile service (in-network wireless quality & eligible for bundle)

6.4. Data API Considerations

A customer data API needs to be designed in a reusable way to support multiple use cases. The primary use case is to serve as the back-end data fetching API for the new UCP front-end. In addition, we received requests from other platform owners to use this API to enrich customer data by making a simple HTTP request to our API. This type of integration is attractive because we eliminate the need for replicating customer data sets between various systems. Replicating large data sets across disparate systems is expensive, slow, and difficult to maintain.

The decision to use GraphQL over REST was an easy decision from the perspective that by 2020 GraphQL was a proven technology offering many advantages over some of the challenges that faced traditional REST APIs.

For example,, retrieving an aggregated view of accounts using a RESTful API might require making multiple calls to the API to get all the required data for each account. Or, if the data has been aggregated to a centralized landing area, the REST API would most likely be broken up into individual resources that return a set of data specific to a use case and still require a client to make multiple API calls to retrieve all

the required data. Also, the data returned from a REST API call has the potential to return a lot of unnecessary data to the client.

GraphQL is a more modern approach to building API's that gives clients more control over the data that they want to get back from an API request. This quote is from the graphql.org web site [2] which does an excellent job summarizing GraphQL:

> *"GraphQL is a query language for APIs and a runtime for fulfilling those queries with your existing data. GraphQL provides a complete and understandable description of the data in your API, gives clients the power to ask for exactly what they need and nothing more, makes it easier to evolve APIs over time, and enables powerful developer tools."*

A query language for APIs means that clients can change the shape of data returned by a GraphQL server. Clients will have control to specify different return fields, run multiple queries in a single request, and add aliases to fields to accommodate naming differences across front-end and back-end code. This all helps reduce the amount of data transformations needed on the client to process the data.

In summary, GraphQL is typically served over HTTP/s via a single endpoint with a data schema that informs clients about the shape and types of data that can be returned from the API. This contrasts with REST APIs over HTTP which typically expose a suite of URLs (multiple endpoints), with each URL endpoint exposing a single resource that defines a data format for the return data.

6.5. Snowflake Cloud Data Warehouse

One of the core requirements was to provide a unified customer channel that can be accessed via SQL as well as an API. It seemed logical that the cloud data warehouse would be a landing zone for customer data, so we built a customer data mart for this purpose. This customer data mart was designed as the primary source of data for UCP.

The primary consideration was that we wanted to use a modern data platform that enabled separation of compute and storage so that we can scale compute and storage separately. This requirement had us considering a cloud data warehouse (e.g., Snowflake) or a modern cloud data lake in AWS. We ultimately chose Snowflake because it was an SQL-based cloud data warehouse that allowed us to leverage existing skill sets with the minimum amount of training required to adopt the new platform. Snowflake's user administration was also simpler and more integrated with the product compared to the AWS data lake.

6.6. API Data Stores

The API data stores were needed to meet our performance needs. For the customer search API, we used Elasticsearch in AWS. There is nothing specific about the Elasticsearch configuration that required it to be hosted in AWS, so this decision was made mostly out of convenience.

For the customer details lookup API, we used DynamoDB, a fast, NoSQL Key-Value database. The main consideration was that we needed a database that could manage semi-structured customer data in JSON format. Other NoSQL datastores like MongoDB could be drop-in replacements for DynamoDB.

6.7. Customer Account Linking

Customer account linking is the process of matching accounts by different attributes like name, service addresses, phone numbers, etc. Once a grouping of accounts is known for a customer, one account gets assigned as the primary account and other accounts become child accounts and are linked via a data relationship.

We took a conservative approach to linking customers by building a recommendation engine that only recommends accounts to be linked. Administrators need to accept the recommendation before one account gets assigned the primary account and all the other accounts get linked as child accounts. The recommendation engine factors in a set of rules that helps prioritize which accounts are considered the primary.

This process runs on a separate platform and the results for account linking get replicated into Snowflake for consumption by UCP.

7. Challenges

Building a GraphQL API that returns data for all customer types across all lines of business required defining a customer data type in a GraphQL schema document. Bringing residential and business customers together in a single data type requires some flexibility in the design so that we do not create dependencies for things that are independent.

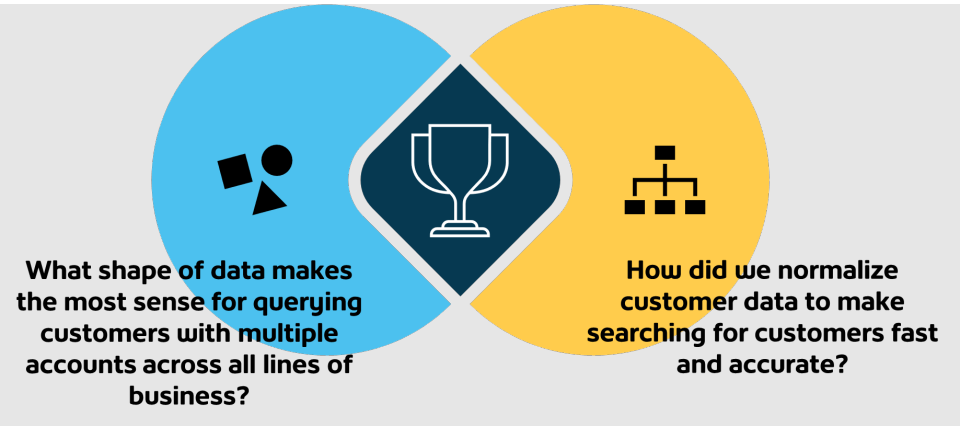


Figure 10 – UCP – Challenges

What shape of data makes the most sense for querying customers with multiple accounts across all lines of business?

The customer API must be flexible and return a range of possible data types depending on the subscribed products and services for each line of business. A customer could be a combination of wireline and wireless services with data coming from different source billing systems. We wanted to future proof the API by not adding any dependencies across billing systems. Billing systems can make changes that are independent of other billing systems and the customer API needs to be able to handle any type of data.

The types of data for a residential customer could differ dramatically from a business customer and the API needs to dynamically handle this. We wanted to remain consistent with GraphQL best practices by having a single API endpoint for all the distinct types of customers and to be able to return all the customer data in one HTTP request. If you only are interested in detail for one type of customer, for example, wireline customer details, the request payload can change to reflect the exact details for a specific customer or even multiple customers.

For the API to handle a range of possible field types, there is a union type that's part of the GraphQL specification.

```
union ServiceProvider = ShawWirelineServiceProvider | ShawMobileServiceProvider | FreedomMobileServiceProvider | AccountNotFound
```

Figure 11 – Union of Customer Types

A query that returns the ServiceProvider type is designed to receive more than one billing account number that could be any one of the ServiceProvider types. If the account lookup fails, the AccountNotFound type gets populated.

In cases where a client is passing in a single account number and already knows the type of service provider, they can build an extremely specific query that reduces the API response to the pieces of data that they care about.

```
query GetInNetworkStatus {
  getAccountsByServiceProvider(getDashboardInput: [{billingAccountId: "TESTESI-11111111"}]) {
    serviceProvider {
      __typename
      ... on ShawWirelineServiceProvider {
        accounts {
          accountName
          billingAccountId
          detailType
        }
        inNetwork
        wirelessCoverageQuality
      }
    }
  }
}
```

Figure 12 – Example of Getting In-Network Status for a Single Account Number

In this example, the client is only interested in finding out whether the customer is in network and has good wireless coverage quality to be able to offer them a Shaw Mobile bundle.

In other cases, we could enter multiple billing accounts for a single customer. In this example, the $getDashboardInput is an array that accepts multiple billing accounts of type getAccountInput. The account numbers getting passed in could be one of the three different service providers or a failure condition when account number is not found.

```
query getAccountsByServiceProvider($getDashboardInput: [getAccountInput!]!) {
  __typename
  getAccountsByServiceProvider(getDashboardInput: $getDashboardInput) {
    serviceProvider {
      __typename
      ... on ShawWirelineServiceProvider {
        accounts {↔}
        telusFibreAvailable
        inNetwork
        bundleEligible
        minStartDate
      }
      ... on ShawMobileServiceProvider {
        minStartDate
        accounts {↔}
      }
      ... on AccountNotFound {
        __typename
        missingaccounts
      }
      ... on FreedomMobileServiceProvider {
        minStartDate
        accounts {↔}
      }
    }
  }
}
```

Figure 13 – Multiple Account Numbers as Input; Aggregate by Service Provider

How did we normalize customer data to make searching for customers fast and accurate?

As mentioned above, one of the design considerations for search was to use Elasticsearch as the search engine for finding customers by customer name, account number, email address or phone number. Elasticsearch, or one of its derivates like AWS OpenSearch, typically ranks extremely high in popularity for enterprise search capabilities. DB-Engines ranks Elasticsearch #1 in popularity as of June 2022. [3]

Today, a lot of the undifferentiated heavy lifting for provisioning and scaling an Elasticsearch cluster is handled by third-party cloud providers. We will not go into tremendous detail on the physical infrastructure that is needed to search across millions of customer records. Instead, we will spend more time discussing how the customer data is indexed to achieve fast and efficient searches.

The physical infrastructure for an Elasticsearch cluster in AWS that will support searching across millions of customers could look like the following.

- Use Amazon OpenSearch Service for managing a six-node search cluster running either an Elasticsearch 6.8 cluster or an Opensearch 1.2 cluster.
- Typically, there would be three master nodes and three data nodes all running on instance types of either c5.xlarge.search (Elasticsearch) or m6g.xlarge.search (OpenSearch).

Indexing data in Elasticsearch is done using analyzers that translate data into tokens that are more suited for search. A token in Elasticsearch contains the string, type, and some positional offsets. In large blocks of unstructured text, the same token could appear in multiple positions. For the examples here, we are dealing with structured data as text or keywords and therefore the positional offsets are not as important.

In Elasticsearch there is a difference between a text field and a keyword field. The text fields get analyzed and broken down into different tokens, whereas a keyword field is typically represented as a single token.

The tokenization of a customer's full name as a keyword field could look like this:

Figure 14 – Keyword Field Tokenization Keeps Josh Smith as a Single Token

Whereas the tokenization of a text field for a customer's full name as a text field would look more like this:

Figure 15 – Text Field Tokenization Separates Name into Two Tokens

Only text fields can be sent through an analysis process that structures data into tokenized formats that are optimized for search. Elasticsearch ships with built-in [4] and custom analyzers. Custom analyzers will be needed to improve search hits for account numbers and phone numbers.

Normalizers are like analyzers but are performed on keyword fields and only produce a single token. Keyword fields are usually structured data types that are recognizable fields like email addresses, phone numbers, and account numbers that are typically used for filtering and sorting.

Now that there is a better understanding of the difference between keywords and text fields, we can jump into some of the things that we had to do to improve searching by account numbers and phone numbers.

Account numbers defined as keywords will get normalized to make searching against them simpler. Because we are dealing with a handful of different billing systems, we could see variations with how account numbers are formatted. Some accounts are all digits; sometimes accounts include leading zeros, other times they do not; and some accounts have characters that prefix a set of digits that could be capitalized or lower case.

The first step is to deal with the leading zeros, so we do not require this as part of search. The char_filter will replace any of the leading zeros with an empty string. For this example, we will just focus on the customer_account_number.keyword field. This field uses the keyword normalizer with a custom filter (char_filter) that removes the leading zeros and forces the final token to be lowercase.

```
PUT scte_index
{
  "settings": {
    "analysis": {
      "char_filter": {
        "no_leading_zeros": {
          "type": "pattern_replace",
          "pattern": "^(0*)",
          "replacement": ""
        }
      },
      "filter": {},
      "normalizer": {
        "keyword_normalizer": {
          "type": "custom",
          "char_filter": ["no_leading_zeros"],
          "filter": [
            "lowercase"                    ]
        }
      },
      "analyzer": {}
    }
  },
  "mappings": {
    "properties": {
      "customer_account_number": {
        "type": "text",
        "analyzer": "account_number",
        "fields": {
          "keyword": {"type": "keyword", "normalizer":
            "keyword_normalizer"
        }
      },
      "customer_name": {}
    }
  }
}
```

Figure 16 – Removing Leading Zeros from Account Number

Using the above settings, searching for a customer using an account number no longer requires specifying leading zeros. The query below uses account number “123456789” as the search term.

```
GET scte_index/_search
{
  "query": {
    "match": {
      "customer_account_number.keyword": {
        "query": "123456789"
      }
    }
  }
}
```

Figure 17 – Account Number Search Query (Leading Zero not Specified)

The query string of “123456789” normally will not match an account number that is indexed with “0123456780” because of the zero prefix. However, if you look at the results below you will notice that the document returned from the query “123456789” matches a document that contains a zero prefix.

```
{
  "took" : 3,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 1,
      "relation" : "eq"
    },
    "max_score" : 0.2876821,
    "hits" : [
      {
        "_index" : "scte_index",
        "_type" : "_doc",
        "_id" : "-EXVc4EB6ViCQ5dG6p-7",
        "_score" : 0.2876821,
        "_source" : {
          "customer_account_number" : "0123456789",
          "customer_name" : "john smith"
        }
      }
    ]
  }
}
```


Figure 18 – Search Results for Query "123456789"

Normalizing phone numbers required a little more work to get satisfactory results. Phone numbers needed to be only numerical digits, 10 digits minimum, not empty, required removal of all zero prefixes, so that different formats for phone numbers could be handled. In the example below, we use a custom phone number analyzer and search analyzer.

```
{
  "scte_index_03" : {
    "mappings" : {
      "properties" : {
        "customer_phone_numbers" : {
          "type" : "text",
          "analyzer" : "phone_number",
          "search_analyzer" : "phone_number_search"
        }
      }
    }
  }
}
```

Figure 19 – Phone Number Mapping Example

And below you can see how the phone number analyzer is configured. There is a char_filter for removal of all non-digit characters from the phone number. The us_phone_number is a custom tokenizer that removes any leading 1's from the phone number and preserves the original number.

```
    "analyzer": {
      "phone_number": {
        "char_filter": "digits_only",
        "tokenizer": "keyword",
        "filter": [
          "us_phone_number",
          "ten_digits_min"
        ]
      },
      "phone_number_search": {
        "char_filter": "digits_only",
        "tokenizer": "keyword",
        "filter": [
          "not_empty"
        ]
      }
    }
```

Figure 20 – Phone Number Analyzer

We can test how the phone_number analyzer works by using the _analyze feature of Elasticsearch. The _analyze feature runs an analyzer with a text string and outputs the generated tokens.

```
GET scte_index_03/_analyze
{
  "analyzer": "phone_number",
  "text": "18005551111"
}

GET scte_index_03/_analyze
{
  "analyzer": "phone_number",
  "text": "1 (800) 555-1111"
}

GET scte_index_03/_analyze
{
  "analyzer": "phone_number",
  "text": "800.555.1111"
}
```

Figure 21 – Examples of How to Test Different Phone Number Formats

The first two _analyze examples resolve to these two tokens [18005551111, 8005551111] while the last example resolves to [8005551111]. The important thing to notice that each example has the 8005551111 as one of its tokens. Actual output from the _analyze command is shown below.

```
{
  "tokens" : [
    {
      "token" : "18005551111",
      "start_offset" : 0,
      "end_offset" : 11,
      "type" : "word",
      "position" : 0
    },
    {
      "token" : "8005551111",
      "start_offset" : 0,
      "end_offset" : 11,
      "type" : "word",
      "position" : 0
    }
  ]
}
```


Figure 22 – Results for Analyzing Phone Numbers

Because account numbers, email addresses, and phone numbers are stored in different fields we needed to use what Elasticsearch calls a multi-match query. In a multi-match query, we are running multiple queries with the same search term. Because the search term gets applied to different search fields, we rely on the relevance score to build the search results page.

Search algorithms apply rules and mathematical calculations to derive a relevance score so we then can do a little manipulation of the scores. For example, we boost complete matches against full name and last name higher than a partial match that uses a wild card search. Elasticsearch uses Lucene under the hood so by default it uses the Practical Scoring Function [5]. The details of the scoring function go beyond this paper, but it is worth noting that Elasticsearch does give you the ability to tune the scoring algorithm and change relevance scores.

8. What's next for Unified Customer Portal

There are a lot of opportunities for us to continue development of UCP so that Care agents continue to have better interactions with the customers. Below are three features that we are currently pitching to the UCP product owners.

8.1. Search Improvements

For customer search we would like to add synonyms for both customer first names and alternate spellings for address searches. To accomplish this in Elasticsearch you need to start with a synonyms file that correlates between a name and different nicknames.

```
aaron => erin, ron, Ronnie
alan => al
…
benjamin => ben
…
josh => joshua
…
william => bela, bell, bill, billy, will, willie, willy
```

Figure 23 – Example Synonym File Provides Alternate Names

Maintaining a synonym list is difficult to do manually so we would like to explore some novel ways on how to automatically push updates to this file. The plan will be to start with an initial list of nicknames that we can eventually compare against actual search logs so that we can measure the synonym list for completeness.

Once a synonym list is added to an Elasticsearch index you can map the synonyms to an analyzer. The example below uses a synonyms field instead of a file, but it is sufficient to prove out how we can take advantage of nicknames. The filter object defined next to number 1 creates the synonyms. The number 2 section tells Elasticsearch to use the name_synonyms analyzer for the name field. And Section 3 configures the analyzer to tokenize text into lowercase and add the synonyms data.

```
PUT scte_index_02
{
  "settings": {
    "analysis": {
      "filter" : {
        "name_synonym_filter" : {
          "type": "synonym",
          "synonyms": [
            "josh,joshua",
            "ben,benjamin"
          ]
        }
      },
      "analyzer": {
        "original": {
          "type": "custom",
          "tokenizer": "keyword",
          "filter": [ "lowercase" ]
        },
        "partial": {
          "type": "custom",
          "tokenizer": "standard",
          "filter":  [ "lowercase" ]
        },
        "name_synonyms": {
          "type": "custom",
          "tokenizer": "standard",
          "filter": [ "lowercase", "name_synonym_filter" ]
        }
      },
      "mappings": {
        "properties": {
          "name": {
            "type": "text",
            "analyzer": "name_synonyms"
          }
        }
      }
    }
  }
}
```

1 2 3

Figure 24 – Mapping Settings that Applies the Name_Synonyms Analyzer

We created three analyzers in the example above and we can use the _analyze function to show how text gets tokenized and synonyms are incorporated.

```
GET scte_index_02/_analyze
{
  "analyzer" : "original",
  "text" : "Josh Smith"
}
```

Figure 25 – Testing the Original Analyzer without Synonyms

The tokenizer named original keeps the text “Josh Smith” as a single token, but it does make it lower case. Take note that these synonyms are not included in this example when using the original tokenizer.


```
{
  "tokens" : [
    {
      "token" : "josh smith",
      "start_offset" : 0,
      "end_offset" : 10,
      "type" : "word",
      "position" : 0
    }
  ]
}
```


Figure 26 – Results of the Original Analyzer

By changing the analyzer from 'original’ to ‘name_synonms’ you will see the tokenization of “Josh Smith” look different from the above example.

```
GET scte_index_02/_analyze
{
  "analyzer" : "name_synonyms",
  "text" : "josh Smith"
}
```

Figure 27 – Testing the Name_Synonyms Analyzer

There are now three separate tokens for “Josh Smith” that can be used for searching [“josh”, “joshua”, “smith”]


```
{
  "tokens" : [
    {
      "token" : "josh",
      "start_offset" : 0,
      "end_offset" : 4,
      "type" : "<ALPHANUM>",
      "position" : 0
    },
    {
      "token" : "joshua",
      "start_offset" : 0,
      "end_offset" : 4,
      "type" : "SYNONYM",
      "position" : 0
    },
    {
      "token" : "smith",
      "start_offset" : 5,
      "end_offset" : 10,
      "type" : "<ALPHANUM>",
      "position" : 1
    }
  ]
}
```


Figure 28 – Tokenization Results of the Name_Synonyms Analyzer

Searching for “Joshua Smith” without synonyms will get a match on “smith” but both “josh smith” and “ben smith” are scored the same.

```
GET scte_index_02/_search
{
  "query": {
    "match": {
      "name": {
        "query": "joshua smith"
      }
    }
  }
}
```

Figure 29 – Searching Names without Synonyms


```
{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 2,
      "relation" : "eq"
    },
    "max_score" : 0.2876821,
    "hits" : [
      {
        "_index" : "scte_index_02",
        "_type" : "_doc",
        "_id" : "90W8coEB6ViCQ5dGSZ9S",
        "_score" : 0.2876821,
        "_source" : {
          "name" : "benjamin smith"
        }
      },
      {
        "_index" : "scte_index_02",
        "_type" : "_doc",
        "_id" : "YsO8coEBexGa4gSBP-bi",
        "_score" : 0.2876821,
        "_source" : {
          "name" : "josh smith"
        }
      }
    ]
  }
}
```


Figure 30 – Both Benjamin Smith and Josh Smith have the Same _Score

After adding synonyms to the search query, you get a good match for a “Joshua Smith” query which will be ranked higher than “ben smith.”

```
GET scte_index_02/_search
{
  "query": {
    "match": {
      "name": {
        "query": "joshua smith",
        "analyzer": "name_synonyms"
      }
    }
  }
}
```

Figure 31 – Searching Names with Synonyms


```
{
  "took" : 17,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 2,
      "relation" : "eq"
    },
    "max_score" : 0.5753642,
    "hits" : [
      {
        "_index" : "scte_index_02",
        "_type" : "_doc",
        "_id" : "YsO8coEBexGa4gSBP-bi",
        "_score" : 0.5753642,
        "_source" : {
          "name" : "josh smith"
        }
      },
      {
        "_index" : "scte_index_02",
        "_type" : "_doc",
        "_id" : "90W8coEB6ViCQ5dGSZ9S",
        "_score" : 0.2876821,
        "_source" : {
          "name" : "benjamin smith"
        }
      }
    ]
  }
}
```


Figure 32 – Josh Smith Now has a Higher Score with the Synonyms File

8.2. Customer Device Details

We would like to enhance the customer details by adding device metrics. Device data that is associated with the service line can provide a Care agent more context on how well their mobile plan is working and if changes are needed.

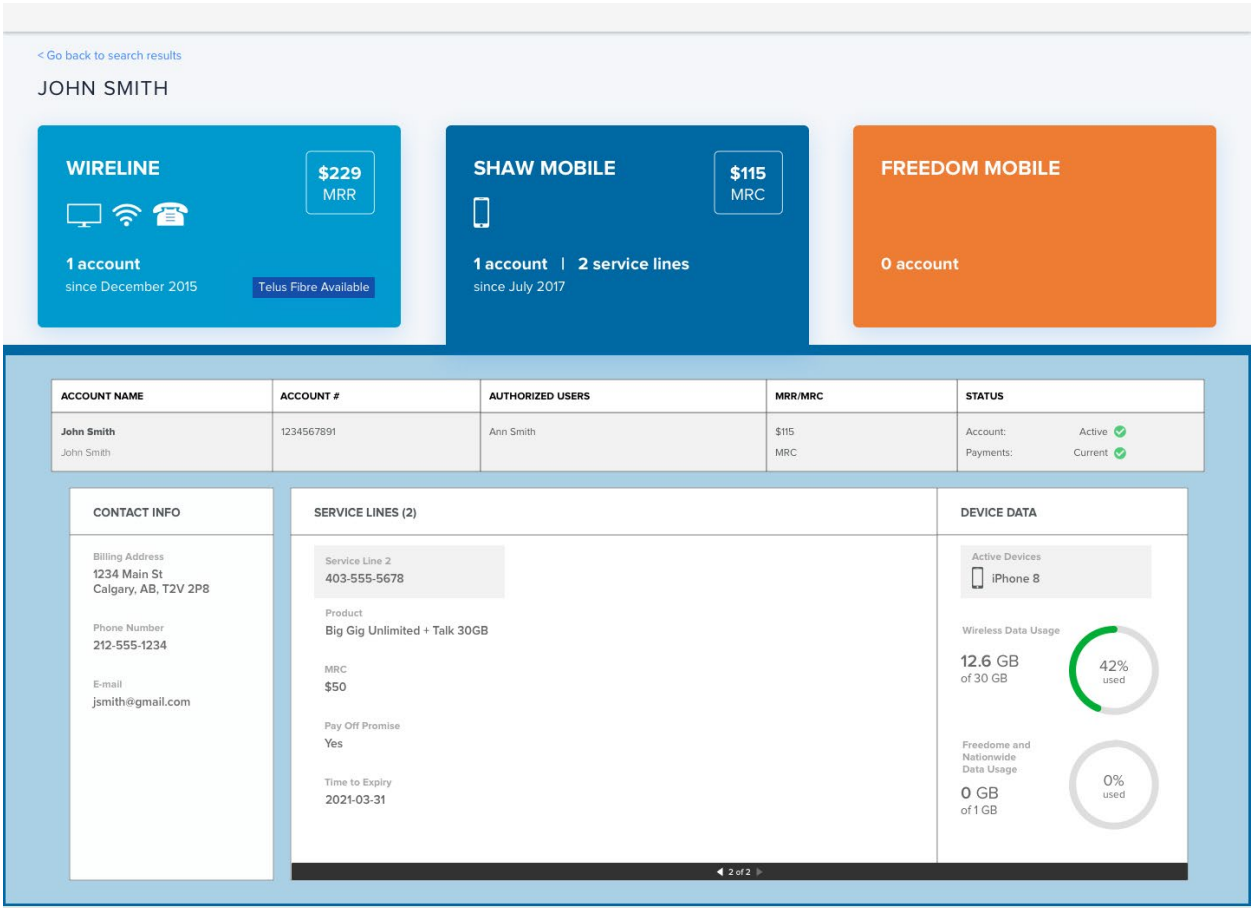


Figure 33 – Device Data Wireframe

8.3. Real-Time Data Loading Process

Our goal has always been to move to more of a real-time data load process for customer account linking and customer search. The challenge is that we are currently dealing with a lot of upstream legacy systems that only support batch exports of data.

There is still some room to optimize the schedules being used for batch loads. Transitioning to event-based schedules will help us load data as soon as it is ready. Snowflake recently announced its Unistore workload which expands the capabilities of Snowflake to support modern transactional data and analytical data in one platform. Keeping data in snowflake could eliminate the Snowflake-to-DynamoDB data load which moves the freshness of data up by 30 minutes.

Real-time data processing will require source systems to support event-based data streams. The UCP platform would subscribe to these event streams and use the data to build additional metrics about a customer.

9. Conclusion

Prior to the introduction of the Unified Customer Platform, our front-line agents would need to access multiple back-office systems across different lines of business. This “swivel chair” experience was time consuming and very inefficient while serving customers that lead to customer frustration and longer call times.

UCP simplifies this for our agents by allowing them to quickly search for a customer by name, account number or phone number. Once the customer is found, UCP displays the accounts and service attributes for all services that the customer subscribes to, eliminating the need to find these accounts across multiple interfaces. UCP provides all the information needed for an agent to understand the current customer engagement with Shaw across different LOBs. In addition, our agents can use the information in the tool to determine if a customer is eligible for any current promotions. UCP serves the intent that we were after: to utilize the data to best serve our customer base and allow our agents to talk intelligently with them.

Over time, we will continue to build on the foundation that we have built and extend the platform by including the ability to link customer accounts in a more formal and verified manner in real time. Our value-based iterative delivery and ability to unlock the true value of enterprise data will help us meet the goals that we have set for ourselves.

Abbreviations

5G	5th Generation
API	Application Programming Interface
AWS	Amazon Web Services
DB	Database
EMR	Elastic Map Reduce
ETL	Extract Transform Load
Gbps	Gigabit per second
GraphQL	Graph Query Language
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
LOB	Line of Business
MDM	Master Data Management
MRC	Monthly Recurring Charge
MRR	Monthly Recurring Revenue
MVP	Minimum Viable Product
NoSQL	Not Only SQL
REST	REpresentational State Transfer
S3	Simple Storage Service
SQL	Structured Query Language
UCP	Unified Customer Platform
URL	Uniform Resource Locator
UX	User Experience

Bibliography & References

[1] https://m.signalvnoise.com/how-we-structure-our-work-and-teams-at-basecamp/

[2] https://graphql.org/

[3] https://db-engines.com/en/ranking/search+engine

[4] https://www.elastic.co/guide/en/elasticsearch/reference/current/analysis-analyzers.html

[5] https://www.elastic.co/guide/en/elasticsearch/guide/current/practical-scoring-function.html