How We Architected

A Cutting Edge Customer Success Platform

Amit Singh
February 6, 2020

Contents

Table of Figures

1 SmartKarrot Customer Success Platform

1.1 The Customer Success Platform

Businesses are shifting from selling products to providing software and services with a subscription based model.

The model generates hundreds - sometimes thousands - of monthly revenue streams instead of a few large product sales.

This complicates the process of measuring and managing “success” with a customer. While revenue is a good metric of success, it is a **lagging** indicator. A customer may downgrade or outright cancel a subscription within months or even weeks if they don’t use or reduce use of some of the features of a subscription-based product

Just tracking revenue or profit hides the problem of customers going away or downgrading in a few months only to be replaced by new customers who also last only a few months - a phenomenon the industry calls **“customer churn”** or **“logo churn”**[1]. A high customer churn is a ticking time bomb but is difficult to spot, especially if revenues continue to grow.

Recurring revenues are far more critical in a subscription based model than one-off revenue. Traditional cost accounting systems and performance measurements systems like the balanced scorecard are not sufficient in a subscription based model.

The SmartKarrot Customer Success platform aims to be the go-to point for managing customer success for subscription based platforms. The platform provides early warnings and indicators like feature usage, user engagement, customer churn and revenue churn that are invaluable drivers for customer success and business success.

1.2 A Success Oriented Architecture

As we started architecting the platform, we realized that it needed to track and measure every meaningful user activity and engagement in real time, mash it together with operational data from CRM systems, data from sales planning and tracking systems like Salesforce, and financial data from enterprise systems.

The platform needed to mine and analyze this large large mound of data to generate succinct scorecards and clean dashboards with well thought through widgets.

[1] "Logo Churn | SaaSOptics." https://www.saasoptics.com/saaspedia/logo-churn/. Accessed 6 Feb. 2020.

It still needed to be a robust, high volume, and secure SaaS platform.

> *We soon concluded that our bleeding edge customer success platform needed a brand new form of architecture - what we called the **"Success-Oriented" Architecture**.*

So what did we need from this architecture?

Firstly, a strong **data pipeline** that collects user engagement data from our customers' systems. This came from a host of SDKs for the Web, iOS and Android platforms. The backend needed to be capable of handling high volume streams, parsing and transforming them and storing them. We built **JavaScript SDKs for the Web** and **native iOS and Android SDKs** for the mobile platforms.

We needed a robust integration platform that would connect with enterprise finance and planning systems and third party sources like Salesforce, Freskdesk, Asana, Hubspot, JIRA, SugarCRM,… We built this integration using **microservices, webhooks[2], an ETL engine and a NoSQL datastore**. With a **serverless platform**, we could do this without running or managing a single server ourselves.

The platform's Web user interface is built with **React** - with carefully designed **state management, routing, AJAX networking, and graph and chart libraries**. Widgets are architected as **micro front-ends**.

Transactions on the server are managed with a **serverless microservices architecture** running on top of a **NoSQL database**.

The whole system is built as a **multi-tenant app with a sharded multi-tenant database model**. This approach is a combination of two approaches: tenant name table partitioning and tenant index partitioning.

The system needs strong analytics. We use **data lakes** and a **query engine** to manage this. A data lake allows us to store structured and semi-structured information. On top of this, we run big data processing, real-time analytics and - in the future - machine learning.

The platform itself runs on a SaaS model. We **meter the usage** of our own services by our customers. Our metering module uses log analysis to measure resource usage for **compute** (to the 100 millisecond level), **data** usage (at rest and in motion for the NoSQL data, data in data lakes and file storage), and **notifications** (SMS, emails, and in-app notifications).

[2] "Building WebHook is easy — using AWS Lambda and API ...." 26 Apr. 2019, https://medium.com/mindorks/building-webhook-is-easy-using-aws-lambda-and-api-gateway-56f5e5c3a596. Accessed 6 Feb. 2020.

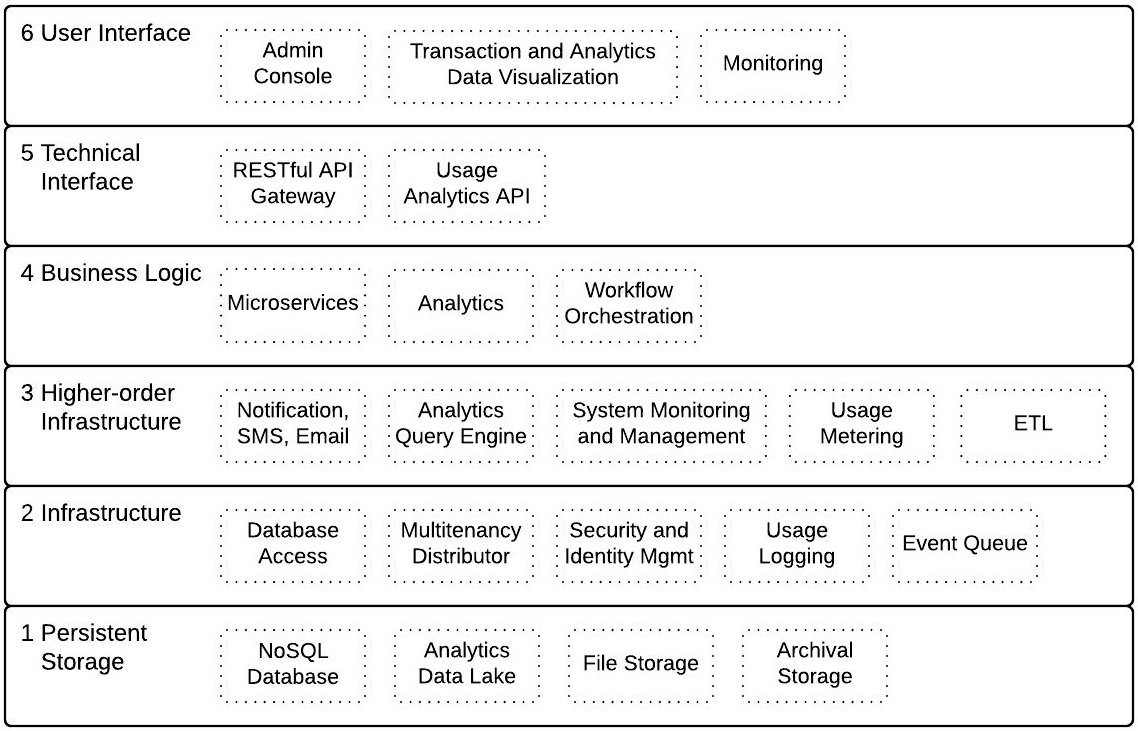
Our customers love having a **sandbox** environment where they can quickly try out our service. Most of our features are available in the sandbox. Our architecture neatly ringfences the sandbox and allows us to **throttle** some sandbox services to ensure that the sandbox does not affect runtime, production system.

The system has sensitive customer data and is designed ground-up to be **secure**. The platform is used by customer success managers and senior managers. Even when processing large volumes of data, it still needs to be quick and **performant**. Good use of caching, pre-processing and optimized widgets help with this.

Finally, a good platform is only as good as its **documentation**. For API documentation, we liked the approach that Stripe and PayPal have taken. Built using the **Slate** API document generator, our documentation is beautifully designed, our documentation has everything on a single page, with a table of content on the left pane, details at the center and code examples on the right.

Want to know more about how we architected a cutting edge customer success platform? Read on.

2 Server Architecture



2.1 SaaS Platform

SmartKarrot is delivered as a SaaS platform - not as a product. While most customers prefer a SaaS model, a small number want to deploy it on a private cloud.

The platform is multi-tenanted and is designed to support hundreds or customers, hundreds of thousands or their users.

2.2 Microservices

Business components and services use the microservices variant of SOA (service-oriented architecture).

The platform is structured as a collection of loosely coupled services as opposed to being a monolithic architecture. Code written with the microservices pattern is forced to be modular and easier to understand, develop, test, and more amenable to parallel development and continuous refactoring.

We use the AWS Lambda serverless compute engine to run our microservies. We like the scaling, and the 100ms metering. This keeps costs low even with fluctuating computing demands.

2.3 Reporting, Analytics and Data Warehouse

SmartKarrot has significant logging, metering, reporting and analytics.

At the data layer, this will be separated into a reporting database. The data could use the S3 file storage. At a later stage, we could shift to using Amazon Redshift. Redshift works well with denormalized fact tables and data warehouses organised in the star and snowflake schemas.

Amazon Glue will be used as the ETL tool to shunt transaction data in DynamoDB into Redshift. Amazon Kinesis Firehose will do the job for real time data.

Business intelligence (BI) and data visualisation will come from the cloud-based Amazon QuickSight. It can be used for ad-hoc analysis, and quickly get business insights from data.

2.4 NoSQL Database

2.4.1 Amazon DynamoDB

The platform stores its data in a NoSQL database. We chose Amazon DynamoDB.

2.4.2 Low Latency Operations

Unencumbered by data joins and relational mappings, and with data hosted on fast solid-state drives, we consistently get sub 10 millisecond response times even at scale.

2.4.3 In-Memory Cache

For the moment, this is more than enough for our systems. In the future we will enable the use of the DynamoDB Accelerator (DAX) in-memory cache to improve the latency from milliseconds to microseconds.

2.4.4 Security

Data at rest is encrypted using the AES-256 algorithm. Data in motion, to and from the database, is also similarly encrypted.

2.4.5 ACID Properties

Though the database supports ACID properties completely, we like the speed that comes from eventually consistent read operations. We almost always use eventually consistent database reads. This satisfies most of our use cases - other than a handful - where we switch to strongly consistent reads.

3 Web User Interface with React

3.1 Why React

We use React to build our user interface. The clean, encapsulated component based design pattern that React articulates goes well with our widget-based UX design.

Each widget, or a set of related widgets, are decoupled from the others. They connect with the analytics backend, intelligently cache, refresh, and invalidate data, and reload themselves on the user interface.

With this widget-component based pattern, we make full use of React's mapping of its virtual DOM to the browser DOM and ensure that the minimum possible update of the screen DOM objects.

All this results in a highly responsive user interface.

3.2 State Management

The Redux state container is the first thing that comes to mind when architects think of a state management solution to React. Though cumbersome to use, Redux solves the state management riddle well.

Another very attractive option is Redux's own Context API. Released a little over a year back in August 2018, the API provides a single global context to manage shared data and actions. Context API is a part of React library.

The Context API has a problem though. It is not up to snuff with high frequency updates. We plan to switch to the Context API when we are sure that this is not an issue any more. In the meantime we have rolled up a custom state and action management library of our own.

3.3 AJAX Networking and API Access

The SmartKarrot Customer Success platform exposes its services through a RESTful API. Some good choices for a pre-built solutions are Axios, the React Fetch API and jQuery AJAX.

The React Fetch API is built into React and is often the natural choice.

We use Axios on our platform.

When comparing Axios and the Fetch API, we liked two aspects of Axios:

1. The code is more concise and clean. We don't need an intermediate function call to convert the data returned from the server into a JSON object. Axios automates the JSON transformation.
2. The Axios promise handles errors in a more intuitive way. For example, on a 400 error response from the server, the Axios promise runs the "catch" block rather than the "then" block.

3.4 Routing

Single Page Applications (SPAs) - like those built using React - do not load new content each time a user navigates to a new page.

We use the React Router library to route content to manage links and route the user to the new page. As a fresh page is not loaded from the server, the link loads very quickly.

3.5 Graphs

The SmartKarrot Customer Success platform uses graphs and charts extensively to concisely display information to senior managers. The platform has pie charts, bar graphs, heat maps and gauges. This calls for an effective and flexible chart library.

We use D3.js JavaScript library - more specifically React wrappers over the D3 library.

4 Micro Front-Ends

A year back we started building the SmartKarrot Customer Success Platform using AngularJS as the front-end technology.

The platform has multiple dashboards, each with carefully thought through widgets. We offer a choice to our customers to integrate their finance, HR, and operations enterprise systems and external systems like Salesforce, Asana, and Freshdesk. Widgets switch on and off based on what systems are integrated.

We soon realized that that React JavaScript library would suit this requirement much better. But this brought up the challenge of either rewriting the Angular code in React, or running the same dashboard with two different technologies. Micro front-ends to the rescue!

> We think of widget-based micro front-ends providing similar benefits on the front-end as microservices do on the backend.

Each component in the micro front-end connects with its own micro-service on the backend. This deeply specialised ecosystem lets us build large and complex dashboards without the inefficiencies from a monolithic single-page app architecture on the front-end with a similar monolith on the back-end.

5 Mobile SDK Architecture

The mobile SDK exposes two layers of services:

1. A native iOS and Android SDK wrapper over the functional REST API.
2. A UI view that lets developers quickly build functionality.

5.1 Mobile Hierarchy of Layers

Much of the UI layer on the mobile is structured using the MVVM design pattern.

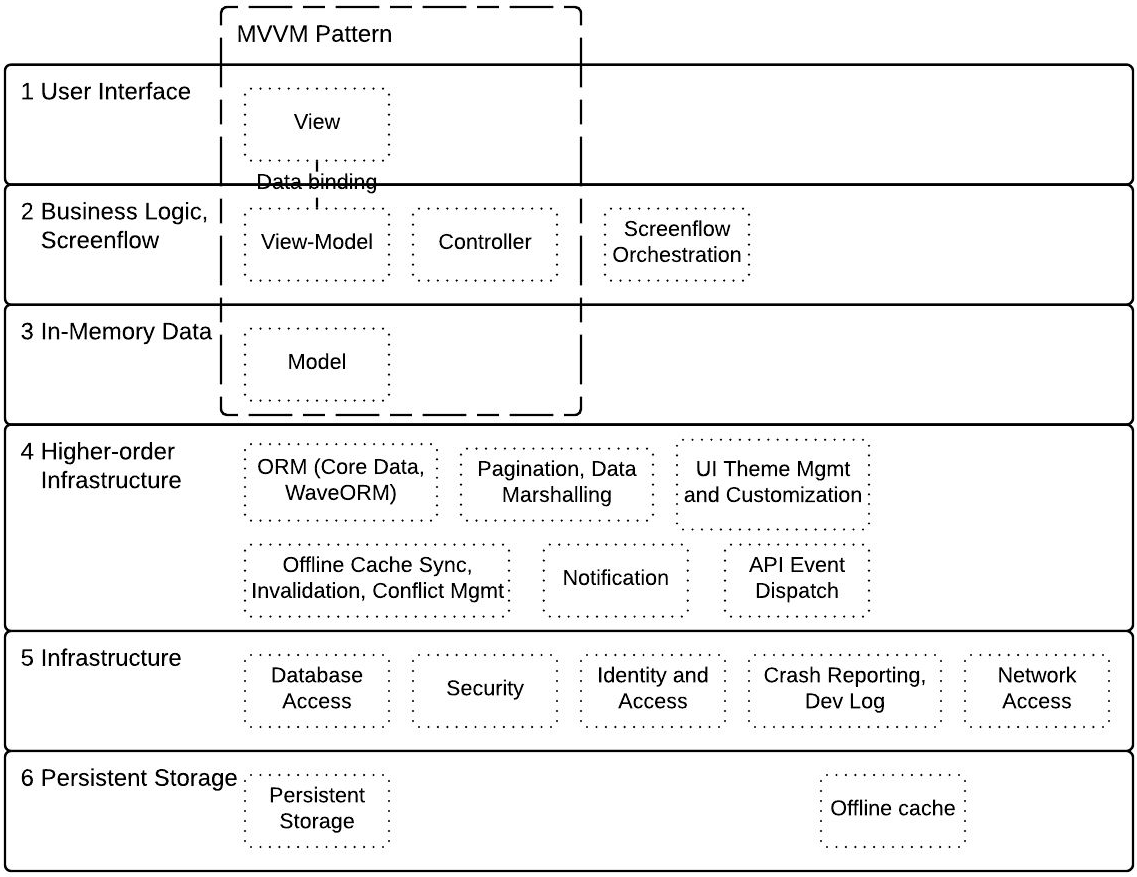


Figure 3: Mobile development view

5.2 UI Theme Management and Customization

This module implements the default themes and exposes a view customization interface to developers.

5.3 MVVM (Model-View-Viewmodel) Pattern

Our system screens have to display and input a large number of fields (example surveys), perform validations, access business services on the backend, and marshal data for storage. Using the traditional MVC pattern will result in bloated and unmanageable controller classes. We will structure the mobile-side code using MVVM.

5.4 API Event Dispatch

App usage events are generated at a high frequency - sometimes tens of them in a minute. Such API calls need to be buffered and dispatched in a separate thread. This API Event Dispatch module will buffer and dispatch all high-frequency APIs.

5.5 Persistent Storage, Cache

The single-version-of-truth data will reside on the server. The mobile will cache a relevant part of it for quicker user responses and offline use. A SQLite database will be used for persistent storage and cache.

5.6 Object Relational Mapping (ORM)

An ORM utility will be used to convert the table-style relational storage in SQLite into an object-oriented structure. WaveORM could be used for this on Android systems.

5.7 Offline

Cached data and offline facilities will be critical to good response times on the mobile. A GraphQL utility will be used for automatic sync and conflict resolution of the cache with the server storage.

5.8 Identity, Access, Security

The security layer will come from AWS Cognito and IAM. TLS will be used to enable secure communication between the mobile and the server.

Cached data on the mobile need not be encrypted and will rely on the mobile OS providing a secure sandbox.

6 Multi-tenanted Architecture

The architecture that we are following on our platform is Multi-Tenant Architecture. Multi-Tenant architecture simply means that the same app, running on the same OS, with the same hardware and same data storing mechanism, servers multiple tenants(users). This architecture is very cost effective (lesser number of licenses = lesser cost), data aggregation/ data mining effort is minimal and it simplifies release management for the tenants. But this architecture is a little

complex and security testing is more stringent owing to the fact that multiple customers' data is being commingled.

6.1 Database Multi-tenanting

There are three approaches DynamoDB provides us to partition our tenants data::

6.1.1 Linked Account Partitioning (Separate Database)

This is the most extreme option available. It provides a separate database namespace and footprint to every tenant. This is achieved by introducing separate linked AWS accounts for each tenant(enabling the AWS Consolidated Billing feature) and one common Payer's account. Once the mechanism is established, we can provide a separate linked account for each new tenant . These tenants would then have distinct AWS account IDs and, in turn, have a scoped view of DynamoDB tables that are owned by that account.

Advantages :

- A bit simpler to manage the scope and schema of each tenant's data
- Provides a natural model for evaluating and metering a tenant's usage of AWS resources.

Disadvantages :

- Cumbersome to manage
- Impractical if there are a large number of tenants

6.1.2 Tenant Name Table Partitioning (Same Database, Separate Schema)

This model embraces all the freedoms that come with an isolated tenant scheme, allowing each tenant to have its own unique data representation. We may use a distinct naming schema that prepends a table name with some tenant id, helping us to identify ownership of the table.

Advantages :

- We can apply AWS IAM roles at table level to constrain access based on tenant role
- AWS Cloudwatch metrics can be captured at table level
- IOPS can be applied, allowing to create distinct scaling policies for each tenant

Disadvantages :

- Downside is more on operational and management side. For e.g.: The operational team will require some awareness of the tenant table naming scheme in order to filter and present information in a tenant-centric context.
- It adds a layer of indirection to any code you might have that is metering tenant consumption of DynamoDB resources.

6.1.3 Tenant Index Partitioning (Shared Everything)

This approach places all the tenant data in the same table(s) and partitions it with a DynamoDB index. This is achieved by populating the hash key of an index with a tenant's unique ID. This means that the keys that would typically be your hash key (Customer ID, Account ID, etc.) are now represented as range keys.

Advantages :

- It promotes a unified approach to managing and migrating the data for all tenants without requiring a table-by-table processing of the information.
- Enables a simpler model for performing tenant-wide analytics of the data helping in profiling trends.

Disadvantages :

- Inability to have more granular, tenant-centric control over access, performance, and scaling.
- Data has to be isolated very carefully, as queries can, in error, access another customer's data.
- This approach could be viewed as creating a single point of failure. Any problem with the shared table could affect the entire population of tenants.

6.1.4 Preferred Approach

Multi-tenancy can be present at any layer or all the layers. As mentioned above there are various approaches to achieve multi-tenancy. We are going to go ahead with a combination of model 6.2.1.2 (Tenant Name Table Partitioning) and model 6.2.1.3 (Tenant Index Partitioning). This approach is known as **Multi-tenant app with sharded multi-tenant database model.**

Most SaaS applications access the data of only one tenant at a time, which allows tenant data to be distributed across multiple databases or shards, where all the data for any one tenant is contained in one shard. Combined with a multi-tenant database pattern, a sharded model allows almost limitless scale.

6.2.1.4.1 Manage Shards

Sharding adds complexity. A catalog is required to maintain the mapping between tenants and databases. In addition, management procedures are required to manage the shards and the tenant population. For example, procedures must be designed to add and remove shards, and to move tenant data between shards.

6.2.1.4.2 Smaller database - Easily manageable

By distributing tenants across multiple databases, the sharded multi-tenant solution results in smaller databases that are more easily managed. For example, restoring a specific tenant to a prior point in time now involves restoring a single smaller database from a backup, rather than a larger database that contains all tenants.

6.2.1.4.3 Tenant identifier in the schema

Depending on the sharding approach used, additional constraints may be imposed on the database schema. If we use this model we will need to use a Tenant identifier which will be used as primary key for any user/tenant.

6.2.1.4.4 Elastic pool of shards

Sharded multi-tenant databases can be placed in elastic pools. In general, having many single-tenant databases in a pool is as cost efficient as having many tenants in a few multi-tenant databases. Multi-tenant databases are advantageous when there are a large number of relatively inactive tenants.

7 Metering Usage

Metering data means accurate tracking of client usage and also providing the capacity for analyzing client usage patterns. The main thing to meter here is the API usage by every user. API usage comprises of both API gateway and Lambda calls. All this will be metered in a centralized manner. There are different approaches that we can take to log every user's API usage. The two main ways are

- Store logs in dynamoDb
- Store log text files

7.1 Event and Compute Logs

In this process we have a separate table for Logs and store all API calls made by users in it. The schema for this would be like below (subject to change) :

```
{
    String userID;
    String apiCall;
    String time;
}
```

We can have scripts which will fetch us user/ tenant specific data and statistics when required.

7.2 Store log text files

There are two ways we can go about this. The first method is a combined usage of **Amazon's Cloudwatch** logs to store our log files and then we can use **Elastic Search/ Amazon's CloudSearch** feature to search and consolidate user specific logs. The other option is to use Amazon's out of the box solution which is **AWS Athena.** Athena is an interactive query service that makes it easy to analyze data in Amazon S3 using standard SQL. Athena is serverless, so there is no infrastructure to manage, and you pay only for the queries that you run.

We are going ahead with the Database approach for now. We will write a common utility function which can be initialized at the beginning of every API call. This will store logs in the Log table and we can keep a track of API usage.

<Write out the architecture for throttling.>

7.3 Throttle

The metering API will be coupled with a throttle mechanism. This is to prevent degradation of services across the platform due to a badly design or implemented customer app or a malicious attack.

<Write out the design for throttling.>

7.4 Sandbox

7.4.1 Requirement

Developers love having a sandbox environment where they can try out a SaaS service without the elaborate procedure of setting up a full account.

SmartKarrot will provide a sandbox to developers. All the features and functionality will be available - with minor exceptions and with a throttle. The throttle will ensure that the sandbox does not affect runtime, production system.

7.4.2 Implementation

The sandbox will internally be implemented as a normal account, with limitations as described below.

7.4.3 No Production Access

Sandbox APIs and SDKs will not be available to apps that that in production. Only apps in development can use them. This restriction is possible to do in mobile SDKs, but not on the Web.

7.4.4 Limits to Functionality

The sandbox will have all functionality available to developers.

7.4.5 Throttle

Some functionality will be throttled to prevent impact to production systems. An example is the number of events that can be tracked.

<Write out how throttling will be designed for>

8 Sandbox

8.1 Requirement

Developers love having a sandbox environment where they can try out a SaaS service without the elaborate procedure of setting up a full account.

SmartKarrot will provide a sandbox to developers. All the features and functionality will be available - with minor exceptions and with a throttle. The throttle will ensure that the sandbox does not affect runtime, production system.

8.2 Implementation

The sandbox will internally be implemented as a normal account, with limitations as described below.

8.3 No Production Access

Sandbox APIs and SDKs will not be available to apps that that in production. Only apps in development can use them. This restriction is possible to do in mobile SDKs, but not on the Web.

8.4 Limits to Functionality

The sandbox will have all functionality available to developers.

8.5 Throttle

Some functionality will be throttled to prevent impact to production systems. An example is the number of events that can be tracked.

<Write out how throttling will be designed for>

9 Security Implementation

With users using the system over the internet, and the system supporting financial transactions, the system is designed to be highly secure.

9.1 Data Encryption

9.1.1 Data at Rest

Server

SmartKarrot persists data on the server in these formats:

1. NoSQL database tables.
2. File storage.
3. Analytics data in data lakes.

SmartKarrot encrypts data at rest using the 256-bit Advanced Encryption Standard (AES-256). The encryption key is managed using AWS' key management service (KMS).

This encryption applies to table data, keys, indices, replicas, backups, in-memory data structure caches, file storage, file archives, and analytics data lakes.

SDK

The SDK on the mobile and the Web mainly stores transient data in a store-and-forward event buffer. A small amount of persistent storage is on a SQLite database that is inside the app sandbox.

9.1.2 Data in Motion

SmartKarrot makes extensive use of RESTful API services. All network traffic to and from the server is encrypted using HTTP over TLS.

9.2 Centralized Key Management

Keys used for encryption are stored in HSMs (hardware security modules) that have been validated under FIPS 140-2. Keys are centralized, rotated, and audited. Key management is enabled using AWS KMS (key management service).

9.3 Identity and Access Management

Amazon Cognito will be used to maintain users' identities and provide authentication. Authentication can be using a mobile number and OTP.

The system will be used by a wide range of users from the Internet cloud. A strong IAM (identity and access management) module will be used to provide fine grained access control. For privileged users (like SmartKarrot administrators) who have access to sensitive functionality, the system will be protected with additional layers of multi-factor authentication (MFA). One of the factors will be the use of TOTP (Time-based One-Time Password) generated from an MFA app like Google Authenticator or Authy.

The app and backend use Amazon Cognito. Use of tokens (temporary, limited-privilege credentials) will mean that real user credentials are never passed back and forth between the mobile and the server.

9.4 Authenticated Access

All server resources will be authorized for use only by Amazon Cognito authenticated users. The Amazon Cognito identity pool's access to unauthenticated identities will be disabled. And to be doubly sure, the AWS Identity and Access Management (IAM) role corresponding to the Amazon Cognito unauthenticated user will have a zero-access role policy.

9.5 Trusted Service Identities and Trusted Subsystems

The server will have a single trusted user service identity, configured through a Cognito identity pool and their corresponding authenticated role.

The server-side will be divided into separate security subsystems. Each component will have separately managed authorization as per a security matrix.

9.6 Security Audit Trail - Monitoring and Logging

Events are on the server is logged to form an audit trail. This event history has both management and data events. Logs are encrypted.

9.7 Access Control

A small number of staff have controlled, need-to-know access to the SmartKarrot production environment. Development and test activities are separated into a different cloud account.

9.8 Compliance to Security Standards

Our server runs in the AWS data center in North Virginia, which is certified for many security standards. Some important ones are listed below.

9.8.1 ISO 27018 - Personal Data Protection

ISO 27018 is a code of practice that focuses on protection of personal data in the cloud. It is based on ISO information security standard 27002 and provides implementation guidance on ISO 27002 controls applicable to public cloud Personally Identifiable Information (PII). It also provides a set of additional controls and associated guidance intended to address public cloud PII protection requirements not addressed by the existing ISO 27002 control set.

9.8.2 PCI DSS Level 1 Service Provider

The Payment Card Industry Data Security Standard (also known as PCI DSS) is a proprietary information security standard administered by the PCI Security Standards Council, which was founded by American Express, Discover Financial Services, JCB International, MasterCard Worldwide and Visa Inc.

PCI DSS applies to all entities that store, process or transmit cardholder data (CHD) and/or sensitive authentication data (SAD) including merchants, processors, acquirers, issuers, and service providers. The PCI DSS is mandated by the card brands and administered by the Payment Card Industry Security Standards Council.

Level 1 of the standard applies to any service provider that stores, processes and/or transmits over 300,000 transactions annually.

9.8.3 ISO 27001 - Security Management Standard

ISO 27001 is a security management standard that specifies security management best practices and comprehensive security controls following the ISO 27002 best practice guidance. The basis of this certification is the development and implementation of a rigorous security program, which includes the development and implementation of an Information Security Management System (ISMS) which defines how AWS perpetually manages security in a holistic, comprehensive manner.

10 Performance

A cutting edge customer success platform needs cutting edge performance. Right from the get-go the SmartKarrot Customer Success Platform analyzed masses of data to show managers concise health scores, trends and insights. This data is sliced by customer accounts, time, etc. Senior managers are impatient and don't want to wait while their screen loads with such analysis.

Lambda at Edge

11 Documentation

12 References

1. "Logo Churn." *SaaSOptics*, 15 Jan. 2020, www.saasoptics.com/saaspedia/logo-churn/.
2. "Context." *React*, reactjs.org/docs/context.html.
3. Morales, Emmanuel. "Micro Front-Ends: Web Components." *Medium*, Embengineering, 4 Apr. 2018, medium.embengineering.com/micro-front-end-and-web-components-ce6ae87c3b7f.

*** End of Document ***