PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA

DESARROLLO DE HIVE MEDIANTE LA MEJORA DE RENDIMIENTO E INTEGRACIÓN DE PROCESOS DE NEGOCIOS

RODRIGO IGNACIO HANUCH GONZÁLEZ

Trabajo de Título para optar al título de
Ingeniero Civil de Industrias, Diploma en Ingeniería de Computación

Profesor Supervisor:
JORGE BAIERA ARANDA

Santiago de Chile, Diciembre 2021

PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA

DESARROLLO DE HIVE MEDIANTE LA MEJORA DE RENDIMIENTO E INTEGRACIÓN DE PROCESOS DE NEGOCIOS

RODRIGO IGNACIO HANUCH GONZÁLEZ

Miembros del Comité:
JORGE BAIERA ARANDA
MEMBER A
MEMBER B
MEMBER C

Trabajo de Título para optar al título de
Ingeniero Civil de Industrias, Diploma en Ingeniería de Computación

Santiago de Chile, Diciembre 2021

Gratefully to my parents and siblings

ACKNOWLEDGEMENTS

Write in a sober style your acknowledgements to those persons that contributed to the development and preparation of your thesis.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ABSTRACT

The abstract must contain between 100 and 300 words. The abstract must be written in English and Spanish. In the case of doctoral theses, the layout of the abstract page is different, so please check the template provided by the OGRS.

Keywords: thesis template, document writing, (**Write here the keywords relevant and strictly related to the topic of the thesis**).

RESUMEN

El resumen debe contener entre 100 y 300 palabras. El resumen debe ser escrito en inglés y español. En el caso de tesis de doctorado, el formato de la página del resumen es distinta, por favor verifique la plantilla entregada por la Dirección de Postrgrado.

Palabras Claves: plantilla de tesis, escritura de documentos, **(Colocar aquí las palabras claves relevantes y estríctamente relacionadas al tema de la tesis)**.

CAPÍTULO 1. INTRODUCCIÓN

1.1. Contextualización de la empresa

Beetrack S.A. (de ahora en adelante, "Beetrack" o "la empresa") es una empresa que brinda un SaaS (*Software as a Service*) dentro del rubro logístico, con un enfoque en la resolución del problema de última milla. Esta fue fundada el año 2013 por Sebastián Ojeda y Nicolás Kipreos como un emprendimiento con el fin de apoyar al creciente comercio electrónico en el proceso de seguimiento y despacho. El día de hoy, 8 años después, la empresa cuenta con más de 650 clientes activos y más de 100 empleados (Corporateit, 2021) entre Chile, Argentina, Perú, Colombia y México. Al mismo tiempo, esta se ha posicionado a si misma como uno de los líderes del mercado SaaS en América Latina por el éxito que ha tenido su principal producto, LastMile.

Junto con LastMile, Beetrack también ofrece un SaaS de planificación, como lo es PlannerPro. Por un lado, LastMile permite tener un seguimiento en tiempo real de las entregas que se realizan, disminuyendo la incertidumbre de los clientes sobre el estado de su pedido. Por otro lado, PlannerPro es utilizado para planificar, optimizar y diseñar rutas de entregas para reducir los costos operacionales de reparto de la empresa que lo utilice.

Actualmente la empresa es la más grande del rubro de SaaS de última milla en latinoamérica, siendo su competencia directa SimpliRoute y Drivin. Beetrack busca posicionarse como la mejor alternativa del mercado constantemente mediante la inclusión de nuevos productos y el servicio de calidad que entrega.

Desde el punto de vista organizacional, la empresa se divide en múltiples áreas. Las anteriores son: administración, *customer success*, *payments*, desarrollo, recursos humanos, *design ops*, *marketing*, operaciones, producto, *revenue operations*, ventas y *data science*. Cada una de estas áreas trabaja en forma independiente con algunos cruces entre áreas en los que se tiene interacciones cruzadas entre distintos equipos. En particular, el trabajo

Múltiples partes del presente trabajo se basan fuertemente en (Burotto, 2021), especialmente las secciones 1 y 2.

realizado fue en el área de operaciones con interacciones con los equipos de ventas, *design ops* y *revenue operations*, entre otros.

1.2. Descripción del proyecto

Los objetivos del proyecto son dos. En primer lugar, el llevar tecnología a todas las áreas de la empresa de manera de poder mejorar y facilitar los procesos internos. Dentro de esta misma idea, el segundo objetivo es poder lograr la mayor automatización posible en estos procesos de manera de que se cometan menos errores y que las diferentes áreas de la empresa puedan enfocarse realmente en lo que les compete por sobre la realización de procesos estandarizados.

Para poder lograr los objetivos propuestos, la empresa decidió crear su propio ERP (*Enterprise Resource Planning*) interno de manera de poder adaptar su *software* a sus proceos y no sus procesos al *software*. El nombre de este ERP es *Hive*, el cual cuenta con diferentes partes y módulos, cada uno con el acceso restringido a los empleados correspondientes por área del módulo. Actualmente, el módulo más utilizado es el de *Payments*, el cual corresponde al del área contable de la empresa que permite generar facturas a los diferentes clientes de Beetrack.

Cabe mencionar que a pesar de que *Hive* tiene un enfoque interno, este *software* posee conexiones con, principalmente, 3 *softwares* externos a este. En primer lugar, posee conexión con los productos de la empresa, esto es LastMile y PlannerPro. En segundo lugar el ERP está conectado con un Hubspot, el CRM (*Customer Relationship Manager*) que Beetrack utiliza. En esta plataforma se encuentra toda la información relacionada a todos los clientes, como el estado de estos dentro de la empresa. Finalmente, *Hive* también se encuentra conectado fuertemente con Quickbooks, un *software* utilizado para llevar la contabilidad de la empresa, permitiendo generar facturas, como también conciliar pagos de clientes. En este sentido, *Hive*, al estar conectado con estos sistemas externos, facilita

la comunicación y automatización de procesos internos, tanto contables como de administración, de manera centralizada al agregar lógica de negocio a los diferentes procesos involucrados.

1.3. Descripción de los trabajos realizados

El alumno ejerció como Ingeniero de *Software Full Stack*, es decir, que estuvo a cargo tanto de las decisiones de diseño de las soluciones tecnológicas, como del desarrollo del *backend*, que corresponde a la lógica de las aplicaciones, y del *frontend*, que corresponde a la interfaz que el usuario final utiliza. Las principales tareas desarrolladas corresponden al análisis y modelamiento de esquemas de negocio y el análisis de espacios que pueden llevar a mejoras en el *software*, junto con el desarrollo de las soluciones a los problemas encontrados.

Por un lado, la empresa le propuso al alumno abordar el desafío de la reestructuración de la forma en que se estaba pensando y utilizando las condiciones comerciales de facturación de cada cliente en Hubspot. En particular se sugirió rehacer todo lo que se tenía desde cero, creando un *software* especializado para esta tarea. Por otro lado, el alumno realizó su práctica profesional en la empresa y se dio cuenta durante ese período de tiempo que existía una falta de pruebas unitarias y de integración en el ERP de la empresa. Por el motivo anterior, el alumno le propuso a la empresa aumentar significativamente el nivel de cobertura de código de su ERP, junto con la integración de estas pruebas al flujo de integración continua en el *software* Circle CI.

1.4. Competencias evidenciadas

Mediante el trabajo realizado dentro de Beetrack, el alumno busca evidenciar las competencias del perfil de egreso. La primera competencia que se busca evidenciar es el "**Aplicar conocimientos avanzados de Ciencia de la Computación, Ingeniería de Software y Sistemas de Información para entender problemas complejos y abiertos**". Esta

se vio ampliamente evidenciada mediante la implementación de las pruebas de integración en *Hive*, junto con el la forma en que se desarrolló el servicio de condiciones comerciales. La segunda competencia corresponde a "**Diseñar y desarrollar modelos y artefactos de software y simular soluciones a problemas de la Ciencia e Ingeniería de Computación, cumpliendo con restricciones técnicas, sociales y éticas**", la cual se puede ver evidenciada por la implementación y subsecuentes pruebas realizadas con el servicio de condiciones comerciales para la facturación automática el mes de octubre junto con la simulación de ejecución de código en las pruebas unitarias creadas. En tercer, y último lugar, la competencia "**Analizar en forma sistemática las diferentes problemáticas de los usuarios, y diseñar productos o sistemas de software que queden expresados mediante algún lenguaje de programación, de acuerdo a los estándares de la ingeniería de software**" se pudo ver evidenciada mediante el proceso de diseño e implementación de tanto el *frontend* en *Hive* para la creación de las condiciones comerciales con el nuevo servicio creado, como el *backend* de este servicio.

CAPÍTULO 2. DESARROLLO DEL PROYECTO

2.1. Metodologías de trabajo

Durante el trabajo de título, el alumno trabajo junto al equipo de *TechOps*, el cual incluye a *Hive* y al *Customer Portal*, proyecto que pretende facilitar la visibilidad de los productos contratados y entregar información de manera transparente a los clientes de Beetrack. Este equipo, conformado por Tomás Burotto, Juan José Martinez y el alumno, utiliza herramientas y metodologías propias de desarrollos ágiles para sus proyectos, en específico, se utiliza *Scrum* con *Sprints* de dos semanas.

Scrum se define como "...un proceso en que se llevan a cabo un conjunto de tareas de forma regular con el objetivo principal de trabajar de manera colaborativa, es decir, trabajo en equipo." (APD, 2019), siendo esta metodología la más utilizada hoy en día en el mundo de desarrollo. Un *Sprint* se caracteriza por comenzar con un *Sprint Planning*, en el cual se determinan las tareas que cada colaborador realizará en el período de tiempo establecido, asignándole a cada tarea lo que se denomina como puntos de esfuerzo. Estos últimos son una forma de medir cuánto tiempo o dificultad tiene cada tarea, de manera de poder mantener un nivel de tareas constante a lo largo de los diferentes ciclos e ir afinando el nivel de carga que cada integrante del equipo puede abarcar.

Tanto las tareas como los puntos de esfuerzos asignados a estas se encuentran ordenadas en un tablero *Kanban*, el cual permite organizar las labores. En particular, en la empresa se utiliza Jira para llevar el seguimiento de las tareas, ciclos de desarrollo y documentación y especificación de las labores. Cada integrante del equipo junta los requerimientos que llegan a lo largo del ciclo y los escribe en el tablero para ser posteriormente asignados en el comienzo del próximo. En la figura 2.1 se puede ver un ejemplo de cómo se ve el tablero en Jira.

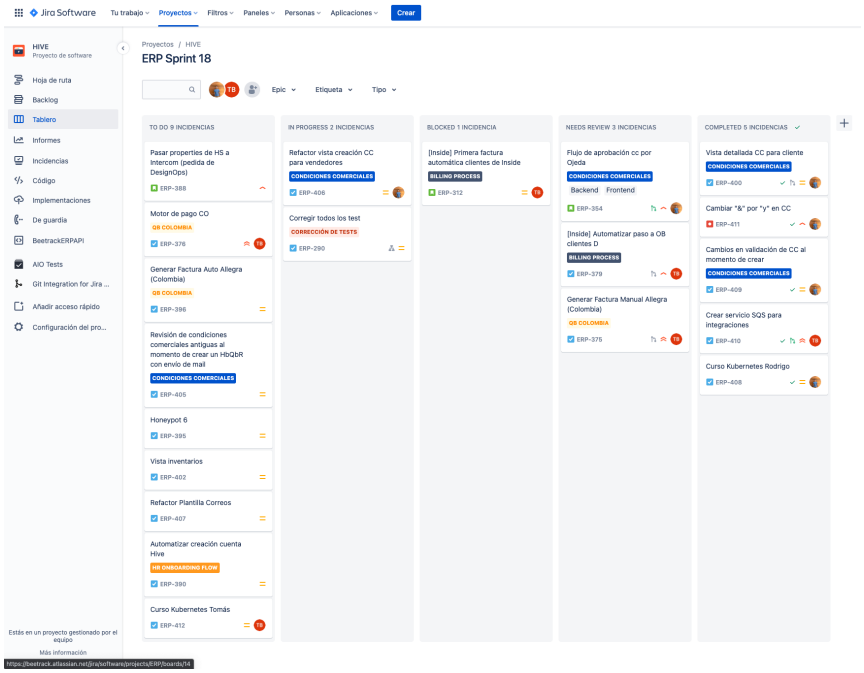


Figura 2.1. Tablero *Kanban* utilizado en el proyecto.

Debido a la cantidad de cambios que se deben realizar constantemente en el código, se requiere un versionamiento de este, por lo que se utiliza GitHub. Esta herramienta permite el uso de Git con *Gitflow*. La principal ventaja de esta herramienta es que permite mantener diferentes versiones, como también utilizar un repositorio en donde se almacena el código en diferentes estados mediante *branches*. Este uso de *branches* o "ramas" permite mantener diferentes ambientes aislados uno del otro. El proyecto de *Hive* cuenta con 3 ramas fijas, que nunca son eliminadas, *master*, *staging* y *develop*. La primera rama es aquella que se encuentra en producción, es decir, es la que es visible para los usuarios finales, la segunda es un ambiente de pruebas (principalmente de estabilidad y de *quality assurance*), y finalmente, la última es la rama de desarrollo, que es aquella en la cual se encuentra el código no probado por personas y que no se encuentra en ningún servidor de manera activa.

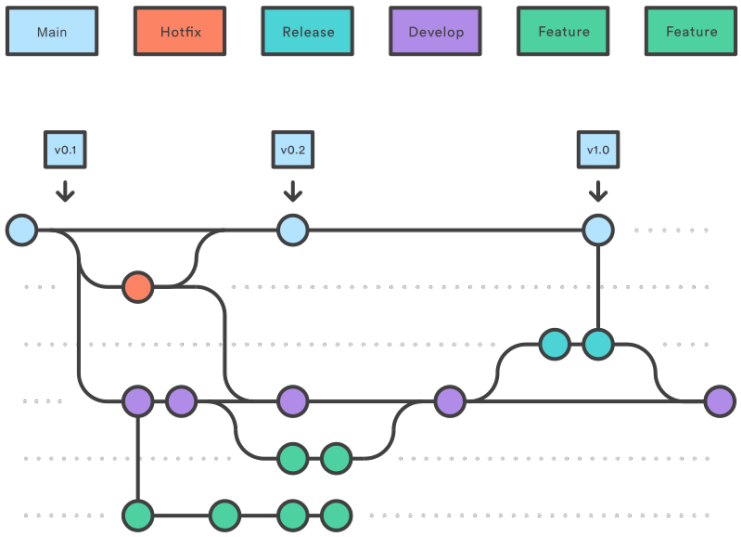


Figura 2.2. *Gitflow* (Atlassian, 2021).

Cuando se quiere desarrollar una nueva funcionalidad, hacer correcciones o realizar cualquier tipo de cambio de código, la práctica es crear una nueva rama, hacer los cambios respectivos en el código, hacer una solicitud de cambio y revisión cruzada entre los miembros que tienen acceso al código mediante lo que se denomina un *pull request*, para finalmente integrar los cambios a la rama principal una vez que los cambios hayan sido aprobados por los otros miembros del equipo. En la figura 2.2 se puede ver cómo se vería un diagrama de un flujo en Git.

Finalmente, dentro de la empresa se utilizan procesos de integración continua, los cuales consisten en escritura de código que permite la puesta en producción del código que se subió a GitHub de manera automatizada en servidores remotos. En particular, se utiliza el SaaS Circle CI como intermediario. El flujo de integración continua, normalmente, comienza al subir el código a la rama *master* o *staging*, lo que gatilla una acción en los servidores de Circle CI, que descargan el código recién subido, corren pruebas sobre este, verifican su integridad, y finalmente compilan el código para subirlo a los servidores remotos especificados. La importancia de estas automatizaciones dentro de las metodologías de trabajo es que en el minuto en que estas automatizaciones fallan se despliegan

alertas que permiten evitar que los servidores en producción se caigan por errores no previstos, sumado al hecho de que se permite ahorrar tiempo con tareas sistemáticas y bien establecidas.

2.2. Tecnologías utilizadas

El ERP de la empresa se basa en una arquitectura que separa de manera lógica los componentes. *Hive* mismo tiene una separación entre *frontend* y *backend*, el primero correspondiendo a todo lo que concierne a la forma de visualizar e interactuar con la aplicación, mientras que el segundo se encarga de todas las operaciones y lógica de negocio de la aplicación, junto con sus interacciones. Estas dos partes se comunican mediante varios *endpoints* a través de una API (*Application Programming Interface*) que el *backend* expone.

Por un lado, una API es un conjunto de definiciones y protocolos que se utiliza para desarrollar e integrar el software de las aplicaciones (Red Hat, 2021). Por otro lado, un *endpoint* es una URL (*Uniform Resource Locator*), o punto de acceso, mediante el cual una aplicación, o API, puede utilizar para obtener recursos y comunicarse con otra aplicación. En el caso del ERP, los *endpoints* cuentan con una capa de seguridad para evitar accesos indeseados de aplicaciones no verificadas, teniendo que identificarse con un JWT (*JSON Web Token*), el cual funciona como llave de acceso para la API y el *backend*. En particular, se utiliza la librería Devise para generar los JWT que son utilizados por la aplicación *frontend* y son únicos por sesión de usuario.

El *backend* de la aplicación está desarrollado en el lenguaje Ruby con el *framework* Rails, por lo que se denomina *Ruby on Rails* (o RoR). Este *framework* permite construir aplicaciones rápidamente que sigan los patrones de diseño de ingeniería de *software* de manera muy simple y que puedan escalar rápidamente. El patrón más utilizado por este *framework* es el MVC (Modelo Vista Controlador), el cual divide a la aplicación en las tres partes mencionadas. En primer lugar, el modelo es un conjunto de clases que representan

la información del mundo real que el sistema debe procesar. En segundo lugar, las vistas son el conjunto de clases que se encargan de mostrar al usuario la información contenida en el modelo. Finalmente, el controlador es un objeto que se encarga de dirigir el flujo del control de la aplicación debido a mensajes externos, como datos introducidos por el usuario u opciones del menú seleccionadas por él (Bascón Pantoja, 2004).

En particular, para almacenar los datos de largo plazo del modelo se utiliza una base de datos con un motor PostgreSQL (o PSQL). Cabe mencionar que no todos los datos se almacenan en este tipo de memoria y que no todas las operaciones se pueden realizar con esta. Es por lo anterior que también se hace uso de Sidekiq, una librería que permite la ejecución de tareas de manera asíncrona y/o en una fecha determinada mediante el almacenamiento de los datos necesarios en memoria *caché*, para lo que se utiliza el motor Redis.

Dada la naturaleza del ERP, como se mencionó anteriormente, este interactúa con otros servicios de dos maneras. Por un lado, para el servicio de condiciones comerciales, proyecto que el alumno realizó, se utiliza GraphQL, mientras que para el resto de los servicios, tales como Hubspot y Quickbooks, entre otros, se utiliza APIs REST. Una API REST es una API que se ajusta a los principios de diseño de REST, un estilo de arquitectura también denominado transferencia de estado representacional (IBM Cloud Education, 2021). Este estilo es uno de los más populares para hacer APIs, pero es menos flexible al compararlo con GraphQL. Este último, es un lenguaje de consultas y servicio para APIs que prioriza el entregar a los consumidores exactamente la información solicitada y no más que eso. El principio de este lenguaje es hacer las APIs más flexibles, rápidas y posicionarse como alternativa a REST (Red Hat, 2019).

El *frontend* de *Hive* está desarrollado en JavaScript con el *framework* React. El código está construido sobre el principio de componentes reusables en las diferentes partes de la aplicación de manera de seguir el principio DRY (*Don't Repeat Yourself*). Dado que el *frontend* es aquel con el cual los usuarios interactúan, este requiere estilo, el cual está dado por el *desing system, Honeypot*, de Beetrack. Este sistema de diseño es utilizado en todas

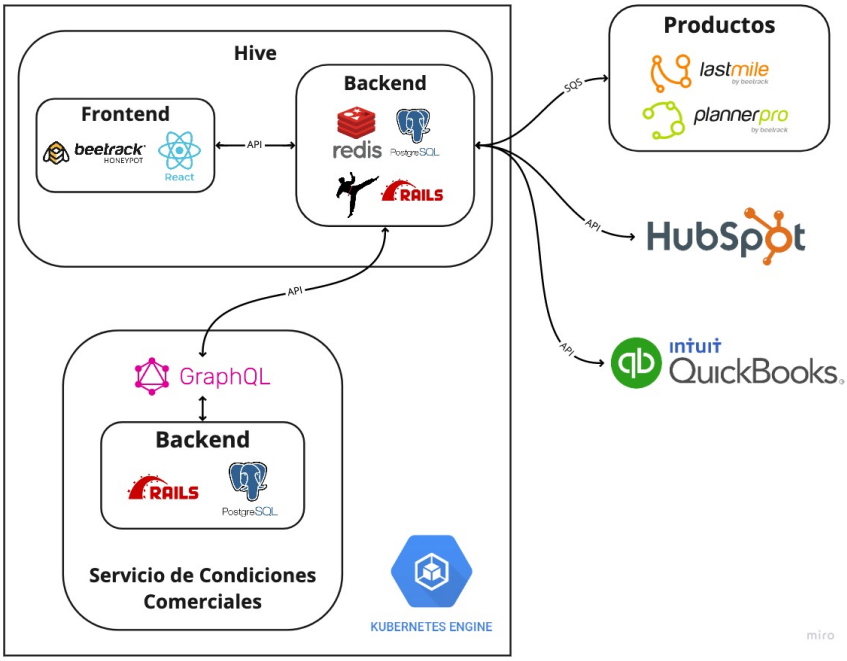


Figura 2.3. Diagrama de tecnologías y arquitectura.

las aplicaciones de la empresa de manera de que se pueda mantener una consistencia visual y de usabilidad entre las diferentes partes que componen los *softwares* que se producen dentro de la empresa. En particular, este *design system* abarca elementos como paleta de colores, diseños visuales de botones, formularios y componentes, entre otros, mientras que al mismo tiempo se tratan como paquetes prehechos, lo que permite reutilizarlos.

En último lugar, tanto la aplicación de *Hive* como el servicio de condiciones comerciales están alojadas en GKE (Google Kubernetes Engine). Este servicio permite mantener las aplicaciones mediante imágenes compiladas de Docker en servidores que son fácilmente escalables tanto vertical como horizontalmente, mientras que al mimso tiempo se encarga de los procedimientos para generar estos servidores y que se coordinen entre ellos. Otra ventaja que provee GKE es la facilidad para poder volver imágenes que ya no se encuentran en uso en el caso de que ocurra algún error, lo que permite aumentar la disponibilidad del servicio entregado. En la figura 2.3 se puede ver la arquitectura con la cual el alumno trabajó.

CAPÍTULO 3. AUMENTO DE COBERTURA DE PRUEBAS E INTEGRACIÓN CONTINUA

3.1. Objetivos y contextualización

Una práctica muy utilizada dentro del desarrollo de proyectos grandes de *software* es el uso de testing unitario de los difernte componentes de la aplicación que sobre la que se trabaja. Esta práctica tiene dos objetivos principales: primero el poder tener seguridad de que el código escrito efectivamente cumple el propósito por el cuál fue programado, y en segundo lugar, el tener la seguridad de que al cambiar una parte del código, este siga cumpliendo su función original. En la práctica, el segundo punto es sumamente útil al momento de reazlizar cambios al *code base* existente, dado que se quiere seguir teniendo la misma funcionalidad, pero cambiando el código interno, en lo que se conoce como *refactor* y mediante las pruebas se puede tener la seguridad de que todo siga en orden después de haber realizado el o los cambios.

En este sentido, es paradójico que, una empresa que se dedica a producir *software*, no haga pruebas de sus desarrollos internos. Es por lo anterior, que el alumno le sugirió a su superior, Nicolás Kipreos, aumentar significativamente la cantidad de pruebas que existen para el ERP, junto con la integración de estas a un flujo de integración continua de manera de que se pudiese tener una mayor seguridad y cofianza al momento de poner en producción el nuevo código.

El pricipal beneficio que el área de *TechOps* tendría mediante la implementación de una mayor cantidad de pruebas y la integración de estas pruebas a un flujo de integración continua serían tres. En primer lugar, el tener la seguridad de que el código efectivamente hace lo que se pretende que haga, en segundo lugar el saber que el código no se caerá de manera inesperada frente a un cambio y, en tercer lugar, el forzar a que el código sea probado antes de ser puesto en producción (IBM, 2021), evitando que los errores lleguen a los usuario de la plataforma.

La meta que se estableció por parte del alumno fue de llegar a una cobertura mínima de 60 % sin pruebas fallidas, junto con el forzar a que el flujo de *deployment* de la aplicación fallara en caso que alguna prueba fallase en Circle CI.

3.2. Desarrollo

En RoR, se hace uso de RSpec para hacer el *testing* de manera automatizada y poder correr las pruebas correspondientes. En el caso de *Hive*, cuando el alumno llegó al proyecto, este tenía una cobertura de 35,44 %. Esto quiere decir que solamente una de cada tres líneas de todo el código existente en *Hive* era ejecutada y probada. Cabe mencionar también que aproximadamente un tercio de las pruebas que existían fallaban, es decir, el código no cumplía con las pruebas que se relizaban sobre el, teniendo un comportamiento diferente al esperado. En la figura 3.1 se puede observa el estado inicial de la cobertura de las pruebas del ERP.

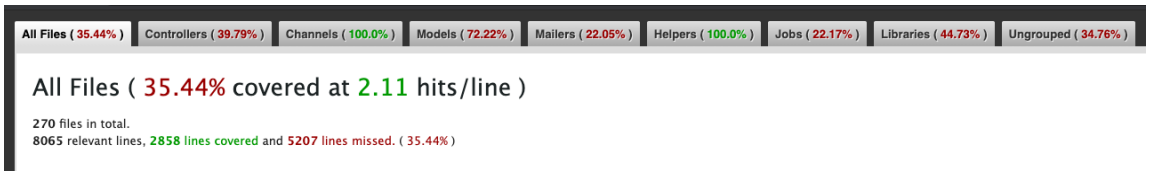
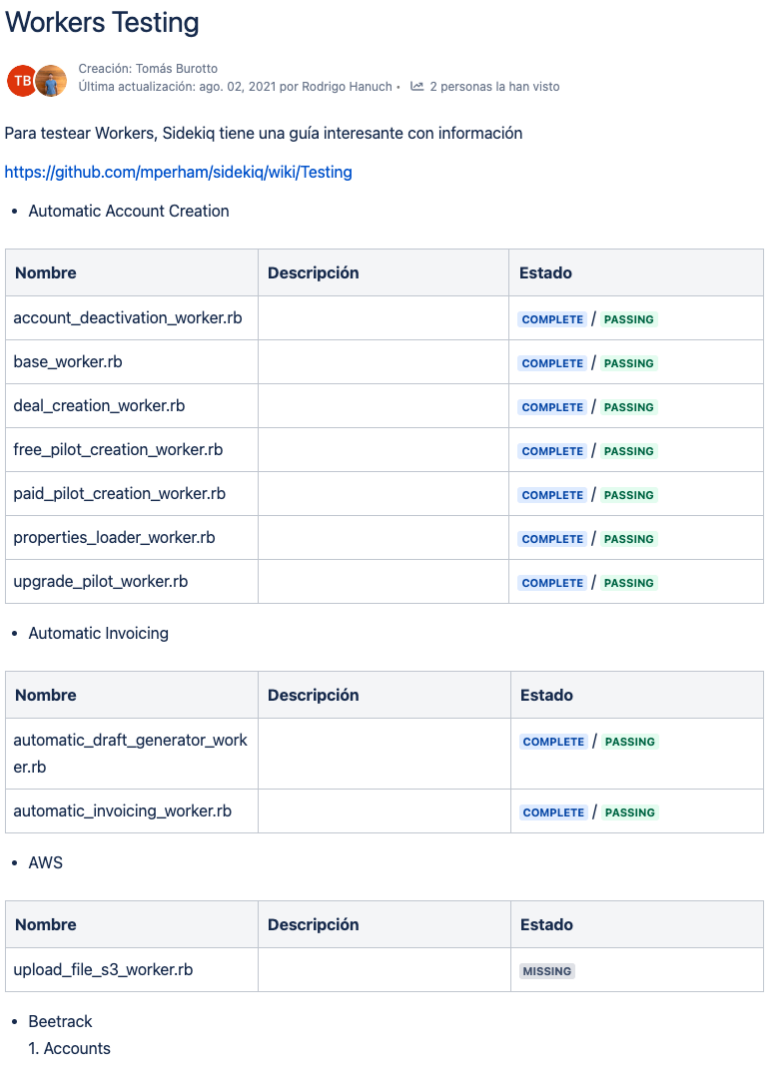


Figura 3.1. Cobertura inicial del ERP (captura tomada el 3 de agosto de 2021).

Dado que *Hive*, en agosto, tenía aproximadamente 8000 líneas de código, el proceso del aumento de la cobertura del *testing* comenzó mediante la revisión y mapeo de todos los archivos existentes y su estado de cobertura de pruebas. En otras palabras, se revisó todo el código fuente del ERP para saber cuáles archivos eran cubiertos por las pruebas automatizadas, cuáles no y cuáles requerían correcciones. De esta manera se podía ver con claridad en qué partes se tenía que enfocar los esfuerzos de trabajo y se podía saber cuáles archivo era más críticos de revisar y probar.



Workers Testing

Creación: Tomás Burotto
Última actualización: ago. 02, 2021 por Rodrigo Hanuch • 2 personas la han visto

Para testear Workers, Sidekiq tiene una guía interesante con información

https://github.com/mperham/sidekiq/wiki/Testing

- Automatic Account Creation

Nombre	Descripción	Estado
account_deactivation_worker.rb		COMPLETE / PASSING
base_worker.rb		COMPLETE / PASSING
deal_creation_worker.rb		COMPLETE / PASSING
free_pilot_creation_worker.rb		COMPLETE / PASSING
paid_pilot_creation_worker.rb		COMPLETE / PASSING
properties_loader_worker.rb		COMPLETE / PASSING
upgrade_pilot_worker.rb		COMPLETE / PASSING

- Automatic Invoicing

Nombre	Descripción	Estado
automatic_draft_generator_worker.rb		COMPLETE / PASSING
automatic_invoicing_worker.rb		COMPLETE / PASSING

- AWS

Nombre	Descripción	Estado
upload_file_s3_worker.rb		MISSING

- Beetrack
  1. Accounts

Figura 3.2. Clasificación y etiquetado de los diferentes archvios y sus estados.

El alumno realizó el mapeo de los archivos junto a Tomás Burotto y dejaron los registros en una sección especial de Jira. Esta documentación se organizó por secciones y subsecciones de archivos y carpetas, siguiendo la misma estructura de *Hive*. Una vez que se transpasaron todos los archivos, se procedió a ponerles etiquetas del estado en que se encontraban, tal como se puede observar en la figura 3.2.

Se decidió tener 3 estados para la cobertura: "*complete*", en el cual 100 % de las líneas correspondientes del archivo eran cubiertas, "*partial*", para indicar que no todas las líneas

del archivo eran cubiertas y "*none*", para indicar que el archivo de pruebas existía, pero ninguna línea era cubierta. Por otro lado, también existían 3 etiquetas para el estado en que se econtraban las pruebas: "*passing*" que aplicaba en caso de que todos los *test* estuviesen pasando, "*failing*" para indicar que una o más pruebas fallaba y "*skipped*", que indicaba que una o más de las pruebas era saltada. Finalmente, se hizo una etiqueta adicional llamada "*missing*" que se aplicó en el caso de que no existiese el archivo de pruebas correspondiente.

Una vez que el alumno tuvo visibilidad del estado del *testing* de la aplicación, este procedió a programar las pruebas automatizadas del código existente. Esto implicó el crear los archivos faltantes para el código que no era probado y leer el código existente, de manera de poder realizar pruebas adecuadas y no simplemente hacer pruebas que pasaran con cualquier resultado. El punto de esto, nuevamente, es que al realizar pruebas detalladas, se puede tener un desgolse muy granular de qué falla en caso de que falle algo dentro del sistema, de manera de que se pueda corregir rápidamente y sin mayor conflicto. En particular, el alumno se enfocó en realizar pruebas automatizadas para la sección de *workers*. El principal motivo del enfoque en estos archvios es debido a que los *workers*, son procesos que corren de manera asíncrona del código principal del ERP y que son automatizaciones sensibles, como, por ejemplo, la generación de facturas y *drafts* de manera automática, notificación a usuarios internos de la empresa, y cambios de estados en los contratos y datos de Hubspot, los cuales son vistos por los vendedores.

Luego de que se cumplió la meta establecida, el alumno procedió a la siguiente tarea, la cual consistió en el cambió del *deployment code* de Circle CI. El flujo de puesta en producción de Circle CI funciona mediante la declaración de pasos a seguir al momento de hacer un *code deploy*. Previamente, la implementación de este código solo contenía la compilación de los archivos y subida a GKE. Para poder realizar la automatización de las pruebas en este entorno, se tuvo que levantar una imagen virtual de PostgreSQL en el contenedor de Circle CI y hacer que no finalizara su conexión luego de que se construyera,

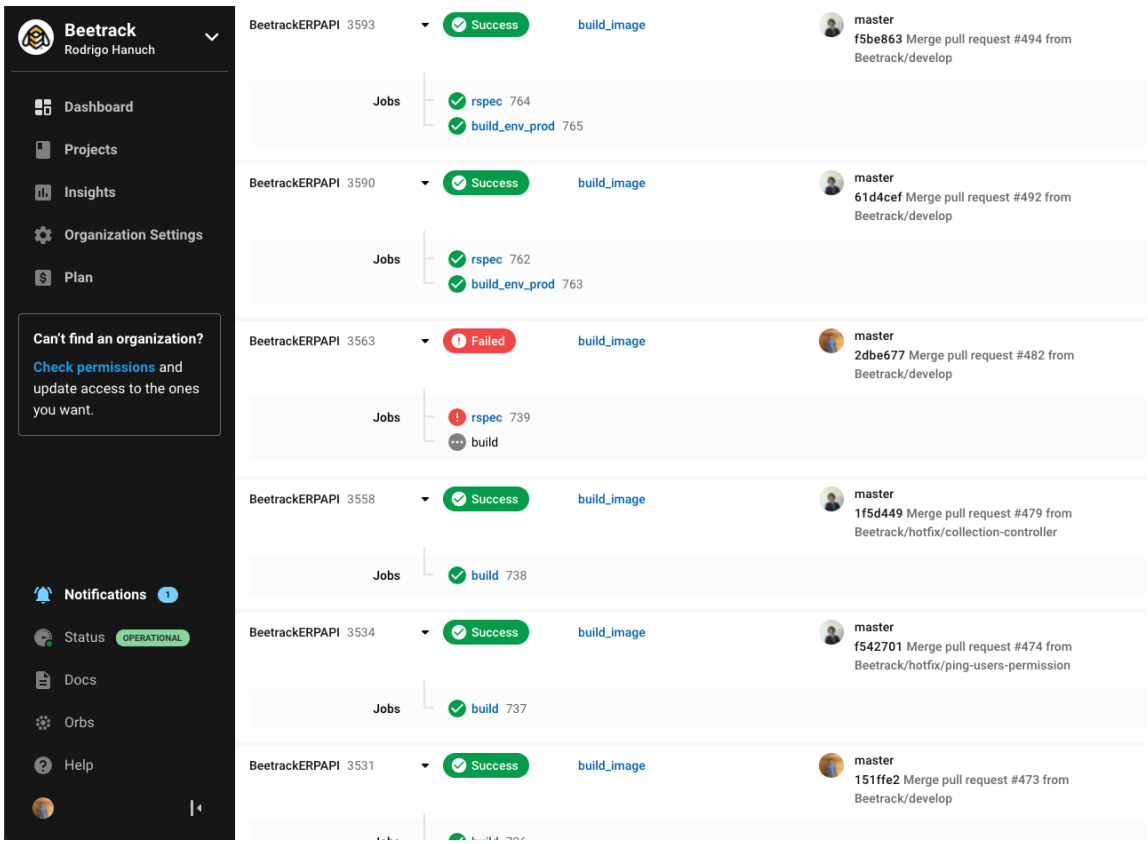


Figura 3.3. Cambio en flujos de Circle CI y falla de *RSpec* previa a la puesta en producción del código.

esto dado que para correr *RSpec* (el comando utilizado para correr las pruebas), se tiene que tener entradas en las bases de datos y para esto se tiene que tener una conexión abierta.

El código que el alumno escribió para el flujo de Circle CI fue hecho de tal manera de que tuviese 2 pasos, en primer lugar las pruebas automatizadas, y luego, en caso de que el primer paso fuese existoso, el segundo paso fuese el *deploy* y compilación del código para ser puesto en producción. Lo anterior permite que se pueda cortar el flujo en caso de que el primer paso falle, liberando antes los servidores que se utilizan de manera común en Beetrack para hacer *deployment* tanto de los productos principales, como del *software* interno de la emprsa. En la figura 3.3 se puede observar tanto el cambio de un flujo único sin pruebas automatizadas a uno con pruebas y el hecho de que cuando falla *RSpec*, el flujo se corta de inmediato, evitando el poner código defectuso en producción.

3.3. Resultados

Los principales resultados obtenidos de este proyecto fueron superiores a los propuestos originalmente. En particular se logó lo siguiente:

1. Cobertura de final de 64,04 % en *Hive*.

   Se logró una cobertura de casi un 65 % del ERP, tal como se puede ver en la figura 3.4. Esto fue posible mediante la **simulación** de la ejecución de código previo a su puesta en producción y la aplicación de **conocimientos avanzados** de *testing* para el *framework* de RoR. Sumado a esto, también se tuvo extremo cuidado al momento de desarrollar las pruebas, evitando el escribir contraseñas o información sensible en estos mediante el uso de las herramientas proveídas por Rails, **cumpliendo con restricciones técnicas y éticas** para evitar que estas sean filtradas de manera pública a internet.

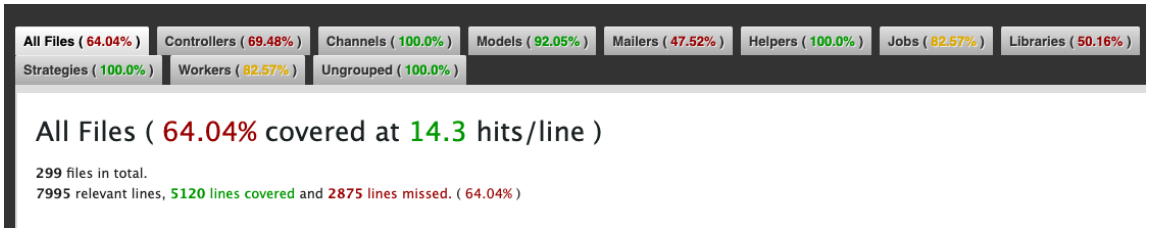


Figura 3.4. Cobertura final del ERP (captura tomada el 15 de noviembre de 2021).

2. Escritura de más de 2800 líneas de pruebas automatizadas.

   El incremento de la cobertura se logró mediante la escritura o modificación de 115 archivos con un total de 2816 líneas de código nuevas dedicadas exclusivamente a la automatización del proceso de pruebas, lo cual se puede ver en la figura 3.5.

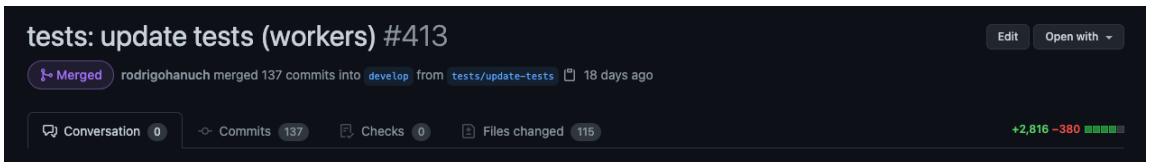


Figura 3.5. Archivos modificados y generados durante creación de nuevas pruebas.

3. Implementación de flujo de integración continua con pruebas automatizadas.

   La implementación del flujo de pruebas automatizadas en Circle CI tiene como principal resultado el incremento en confiabilidad del código escrito, y, por lo tanto, la disminución del *downtime* generado por líneas no probadas para el equipo *TechOps* y para la aplicación de *Hive*. Sumado a lo anterior, esta implementación también consideró un flujo con menor cantidad de código al reescribir, en gran parte, los comandos que generaban los *deployments* a GKE, reutilizando código y haciéndolo más legible al aplicar **patrones de diseño de *software*** y eliminando muchos *code smells* (antipatrones de diseño, que ocurren cuando el código no sigue estándares y que pueden indicar problemas más profundos (Fowler, 2006)).

CAPÍTULO 4. SERVICIO DE CONDICIONES COMERCIALES

4.1. Objetivos y contextualización

Actualmente, la empresa tiene más de 700 clientes activos, cada uno con diferentes contratos, o condiciones comerciales, y se utiliza un archivo de Excel para mantener estas condiciones actualizadas. Al mismo tiempo, este excel también contiene la forma en que se calculan los costos totales por el uso del servicio para cada cliente. Similar a lo que ocurría con la falta de pruebas automatizadas, también resulta curioso el hecho de que Beetrack, siendo una empresa SaaS y teniendo su propio ERP, utilice esta plantilla para calcular cuánto cobrarle a los clientes. Es por lo anterior que decidió crear un servicio especializado en el almacenamiento y cálculo de estas condiciones comerciales, y cuyo único propósito fuese resolver el problema de cómo calcular los costos por cliente y poder almacenar los modelos que representaran los contratos generados con estos.

Uno de los puntos más cruciales sobre este servicio y su contexto es el hecho de que ya se había intentado dos veces algo similar. El primer intento fue mediante un desarrollo interno en el ERP, el cual no tuvo los resultados esperados en términos de precisión al momento de facturar y se concluyó que no era responsabilidad de *Hive* como tal tener esta información. El segundo intento fue la migración de estos planes a un proyecto que, en su minuto, se encontraba en fase *beta* en Hubspot, en el cual se prometía poder tener la información siempre actualizada. Finalmente terminó ocurriendo lo mismo que en el primer intento, pero con peor precisión al momento de facturar de manera automática, entregando, en el mejor de los casos, una precisión de 43 %.

Teniendo en consideración los requisitos y los resultados anteriores, los objetivos principales del proyecto fueron dos. El primer objetivo, fue construir un modelo lo suficientemente flexible como para poder abarcar los diferentes tipos de ventas que los vendedores realizan, tanto para productos como servicios. El segundo objetivo fue el programar un servicio de resolución comercial, o calculadora, que fuese, nuevamente, lo suficientemente flexible como para poder abarcar el modelo, pero que al mismo tiempo no necesitara

información contextual más allá del uso para saber cuánto cobrarle a cada cliente, mientras que al mismo tiempo entregaba glosas prehechas por producto.

El equipo de Operaciones decidió no acoplar este proyecto a *Hive* de manera de evitar el cruce indebido de información y para así poder mantener las ideas aisladas y para así comenzar a tener una arquitectura de sistemas distribuidos en base a servicios. Lo anterior sumado al hecho de que el se prevee que, eventualmente, otros servicios y productos de la empresan, tales como el *Customer Portal*, consuman los recursos expuestos por este servicio de manera de que los clientes también puedan saber cuánto se les cobrará o el nivel de costo que tienen para un día determinado del mes. Como se puede ver en la figura 2.3, del capítulo 2, la arquitectura que se decidió utilizar es una en la cual el servicio de condiciones está aislada del ERP, mientras que al mismo tiempo no limita a que otro servicio pueda consumir la información del servicio nuevo, lo cual hubiese ocurrido en caso de haber escrito el código en *Hive* dada su naturaleza monolítica.

4.2. Desarrollo

4.2.1. Modelamiento

El proceso de desarrollo del nuevo servicio de condiciones comerciales comenzó con un análisis de los diferentes planes existentes y que se encuentraban en uso en por parte de la empresa. Luego de realizar este análisis se definieron 4 modelos diferentes a nivel de base de datos que permitirían almacenar de manera flexible las condiciones con las cuales se les factura a los clientes.

En primer lugar se tiene el modelo del plan, en este se almacena la condición comercial como tal, junto con una descripción del uso que puede tener esta condición y el producto y moneda asociado a la condición. Dentro del campo "`condition`" se almacena el tipo de tramo y costos asociado que tendrá por ellos. Se decidió que se puede tener tres tipos de tramos diferentes: base, variable y por paso. A continuación se detalla cada tipo de tramo:

1. Tramo base: va desde cero unidades hasta la cantidad de unidades determinada con un costo fijo total por esas unidades. Solamente se puede tener un tramo de este tipo por plan.
2. Tramo variable: funcionan mediante una definición de inicio a final del tramo en término de cantidad de unidades, con un precio unitario por la cantidad de unidades utilizadas. Se pueden tener entyre cero e infinitos tramos variables por plan.
3. Tramo por pasos: se utilizan para poder representar los productos que se venden mediante paquetes, esto es especialmente útil para el servicio de correos en el cual se cobra mediante paquetes de 10.000 correos, siendo los primeros 10.000 gratis y los siguientes teniendo un costo de 1 UF por bolsa adicional. Este tipo de tramo es excluyente con los dos anteriores y solo se puede tener, el cual funciona de manera recurrente.

Adicionalmente, el campo "`condition`" contiene dos variables utilizadas al momento de calcular el costo total al momento de generar la factura para un cliente. Estos se detallan a continuación:

1. `general_minimum`: se utiliza para asegurar un monto mínimo de facturación del producto independiente del uso del cliente una vez que ya se haya calculado el costo por el uso que el cliente le haya dado al producto o servicio.
2. `add_base_to_variable`: se utiliza al momento de calcular el costo del uso del producto, y sirve para saber si el plan que se negoció con el cliente se entiende de manera de que se cobre sumando el costo base o utilizando el costo unitario del tramo variable en caso de aplicar.

El esquema de información del campo "`condition`" se puede ver en la figura 4.1, mientras que en la figura 4.2 se puede observar una manera visual de entender el modelo de precios y condiciones comerciales de los clientes. Es importante mencionar que se hizo uno uso avanzados de RoR mediante la creación de artefactos de validación que permniten mantener los esquemas de datos establecidos en la figura 4.1. Al mismo tiempo, también

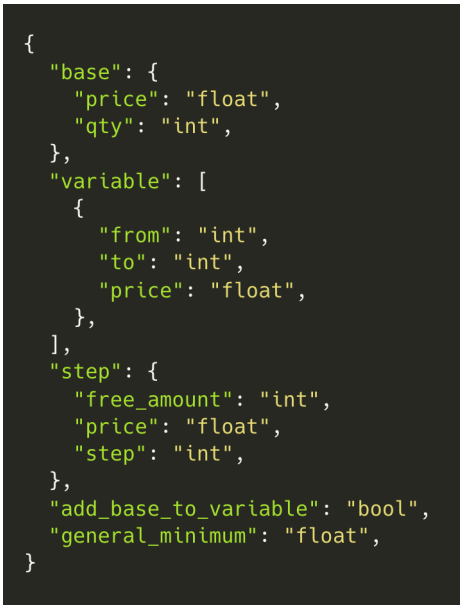


Figura 4.1. Esquema de información de condición comercial.

se utilizó validación de los datos en si mismos, forzando a que siempre se tengan tramos positivos, sin sobrelape y que sean rangos acotados, de manera de que las condiciones siempre tengan las formas de la figura 4.2.

En segundo lugar, el modelo de relación plan (`plan_relations`) tiene el rol de asociar el contrato del cliente con el plan creado de manera desacoplada y permitiendo la reutilización de planes estándares.

En tercer lugar, "`plan_relation_proposals`" tiene el propósito de permitir que se pueda tener una agrupación de relaciones de planes en un mismo modelo en caso de que un vendedor haya pactado con un cliente condiciones no estándar. Las condiciones estándar se definieron en conjunto con los vendedores a lo largo de múltiples reuiones y una posterior validación de Felipe Porter, jefe del área de ventas de la empresa. En caso de que las condiciones que un vendedor no sean las predefinidas y acordadas se definió que estas requerirían aprobación de Felipe Porter o Sebastián Ojeda (CEO). En caso contrario, las condiciones para el cliente determinado se aprueban de manera automática y se pueden utilizar al momento de facturar.

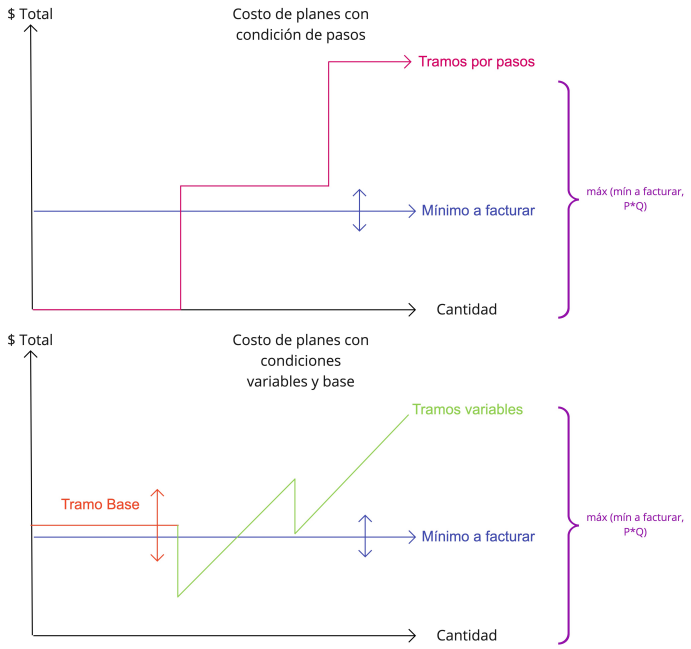


Figura 4.2. Modelos de cobros y condiciones comerciales.

En cuarto y último lugar el modelo "`comments`", tiene el propósito de simplemente contener el cuerpo del comentario por parte del vendedor en caso de que quiera hacer notar algo de los planes que está creando.

En la figura 4.3 se puede observar los diferentes modelos y sus relaciones, junto con sus campos y tipos de datos por campo.

4.2.2. *Solver* de condiciones y arquitectura

El paso siguiente a la creación de los modelos a utilizar, fue el de la creación de un módulo dentro del servicio de condiciones comerciales que permitiese calcular la facturación adecuada para un cliente en específico dado el uso que este tuvo durante el mes. El módulo de cálculo se basa en un servicio específico para cada producto existente dentro de la empresa, es decir, por cada producto se tiene un servicio específico para calcular y resolver el cobro a realizar para ese producto dado el uso entregado.

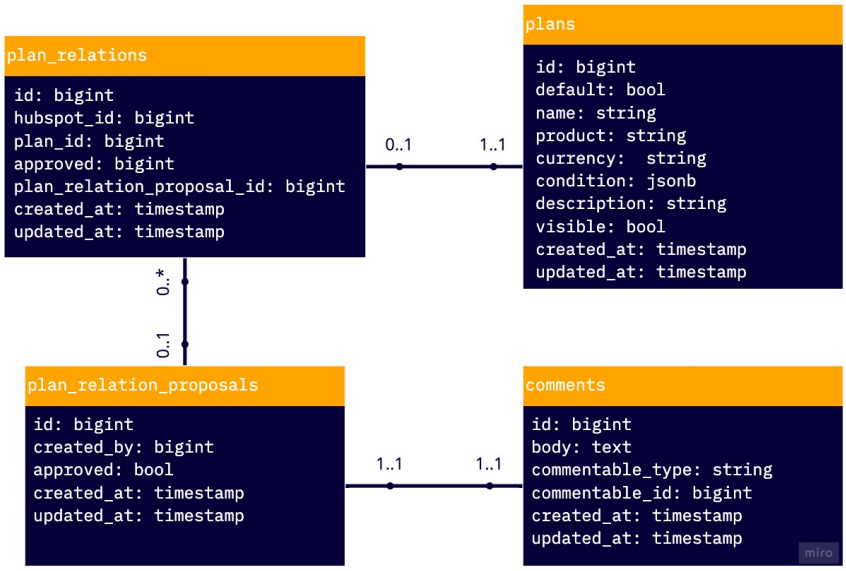


Figura 4.3. Relaciones de modelos de condiciones comerciales.

La información externa que el servicio de cálculo requiere es el identificador del cliente, junto con el uso de los productos. Al momento de hacer el llamado al servicio, este, con la información entregada, se programó de manera de que pase por una calculadora genera y delegador, de manera de que se subdelegue el cálculo del monto a facturar por cada producto a los subservicios específicos por productos. Este último diseño se basó en la implementación del patrón de diseño "mediador", en el cual se restringe las comunicaciones directas entre los objetos, forzándolos a colaborar únicamente a través de un objeto mediador (Refactoring Guru, 2021b) y "fachada", en el cual se proporciona una interfaz simplificada a una biblioteca (Refactoring Guru, 2021a). Por otro lado, la información del tipo de cobro se obtiene directamente desde la base de datos del servicio mediante el producto que se quiere calcular y el identificador del cliente para el cual se está calculando.

Una vez que se realizó el cálculo por producto, el servicio retorna la información a un método que agrupa los diferentes resultados, suma el total y entrega un desglose por producto y por moneda, especificando el servicio de resolución utilizado, junto con la glosa a utilizar al momento de generar la factura en el ERP. En la figura 4.4 se puede ver de manera general el flujo utilizado en el *solver* y calculadora de costos.

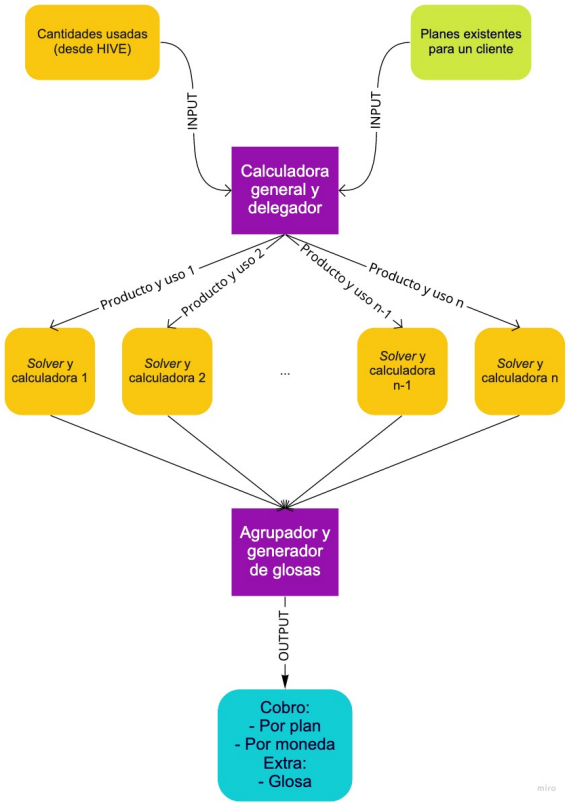


Figura 4.4. Esquema de delegación de *solver* y calculadora.

Posteriormente, una vez realizado el servicio de cálculo, se procedió a conectar el ERP con el servicio de condiciones comerciales. Para realizar la conexión se optó por utilizar una API con GraphQL expuesta por parte del servicio de condiciones comerciales. Por parte de *Hive*, se utilizó Artemis, una librería de RoR que permite conectar el cliente, en este caso el ERP, con la API de manera casi invisible, exponiendo los métodos como si fuesen nativos de la aplicación (Nishijima, 2021). Esta implementación permite a su vez que otras aplicaciones dentro de Beetrack consuman el servicio mediante el ERP, en caso de ser requerido. En la figura 4.5 se puede observar la arquitectura con los dos *pods* diferentes de GKE, uno para el ERP y otro para el servicio de condiciones comerciales y su comunicación mediante una API con GraphQL.

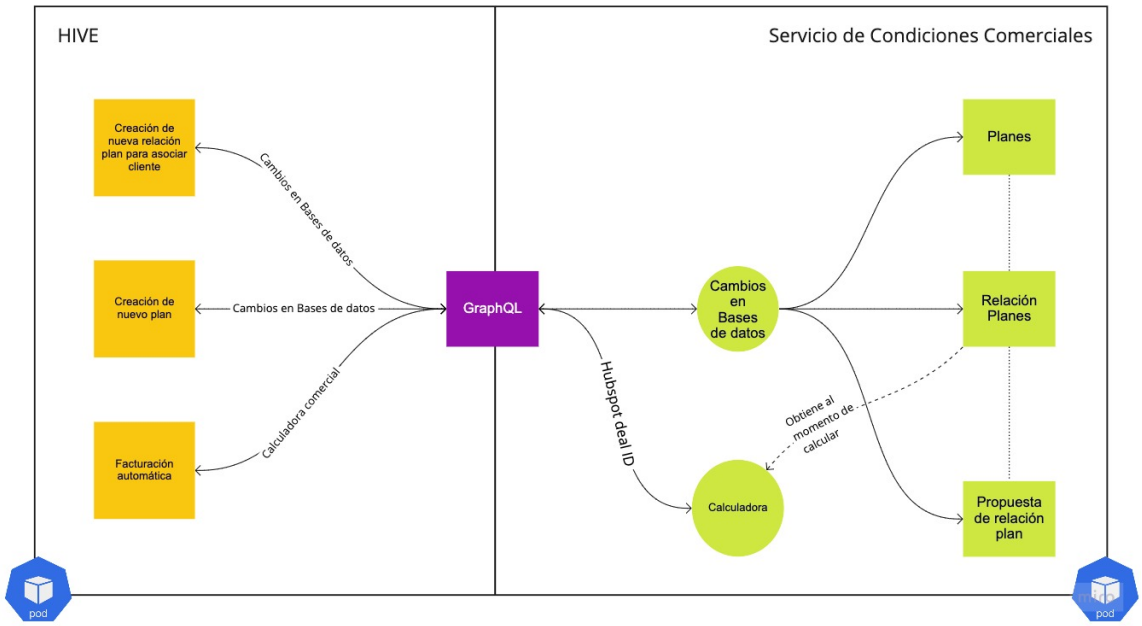


Figura 4.5. Relaciones de modelos de condiciones comerciales.

4.2.3. *Refactoring* y generación de facturas automáticas

Una vez que el alumno conectó el servicio de condiciones comerciales al ERP, este procedió a desacoplar el sistema antiguo que generaba las facturas de manera automática los 28 de cada mes. Esto se hizo en favor del servicio nuevo externo a *Hive*, adaptando el código antiguo al sistema nuevo permitiendo un desacoplamiento de la lógica por parte del ERP, delegándosela al sistema creado. Este cambio implicó la eliminación de más de mil líneas de código en el ERP, con cosas tales como la generación de glosas y cálculo de montos a facturar.

Posteriormente, el alumno le pidió al CEO de la empresa, Sebastián Ojeda, el archivo de Excel en el cual mantenía las condiciones comerciales de manera de poder poblar el servicio. Para esto, el alumno se comprometió a no entregar los datos de facturación a nadie sin autorización previa, manteniendo un código ético intachable dada la sensibilidad de los datos. Adicionalmente, dado que el elevado número de contratos de la empresa

(mayor a 500), se decidió escribir código que transcribiese el archivo excel al formato del servicio, generando más de 1500 planes diferentes válidos.

Luego de que se transcribieran los planes generados al servicio de condiciones comerciales, el alumno se coordinó con el equipo de *Data Science* de manera de poder hacer simulaciones de facturación antes de poner en funcionamiento el servicio de generación de facturas reales. De esta manera, se simuló una ejecución del problema real y se puso a prueba el código desarrollado para tener conocimiento de cuán preciso era. El alumno trabajó con el equipo de *Data Science*, dado que este es el encargado de la carga de datos de uso de los clientes, por lo que se pudo generar simulaciones con diferentes niveles de uso y para diferentes clientes. Los resultados obtenidos fueron altamente prometedores, obteniendo aproximadamente una precisión de facturación del orden de 87,9 %. Finalmente, el 28 de octubre corrió el servicio de generación automática de pre-facturas exitosamente y recibió comentarios muy positivos por el equipo contable de la empresa, dado que no tuvieron que modificar las pre-facturas antes de emitirlas.

4.2.4. Creación de condiciones por parte de vendedores y aprobación de gerentes

En paralelo junto con la generación automática de pre-facturas, el alumno trabajó en 4 vistas dentro del ERP, las cuales correspondían a dos formularios y dos vistas generales. Adicionalmente se crearon 3 plantillas de correos mediante HTML, CSS y Ruby embebi; estas se detallarán más adelante. Estas vistas fueron validadas y diseñadas junto a Grace Lillo, del equipo de *design ops*, de manera de que estas fueran realmentes útiles para los usuarios finales. Cabe mencionar que estas vistas fueron generadas utilizando los componentes de *Honeypot* en React.

En primer lugar, el estudiante programó una vista general en donde los vendedores pueden ver las condiciones comerciales preaprobadas por los gerentes, de manera de poder tener una noción de cuáles son las condiciones de cada plan antes de utilizarlo para un determinado cliente. A continuación, en la figura 4.6, se puede ver la vista mencionada anteriormente.

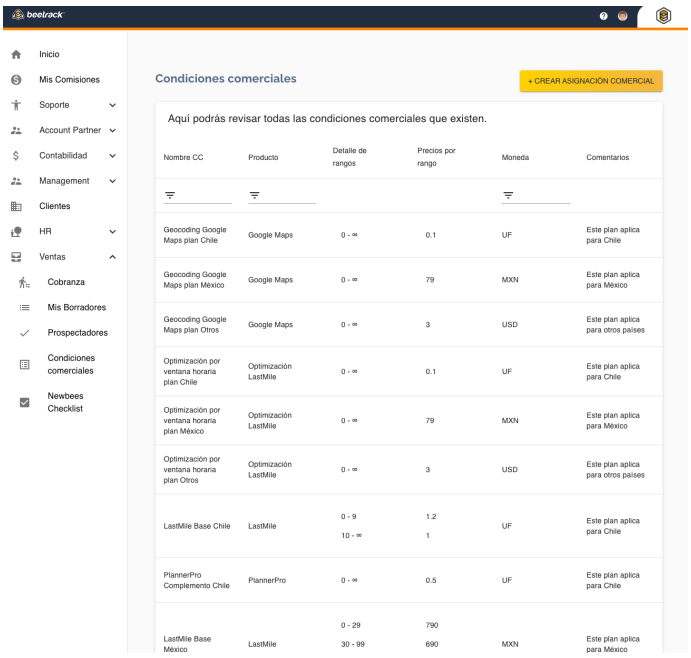


Figura 4.6. Vista general de condiciones comerciales preaprobadas.

En segundo lugar, el alumno realizó una formulario que le permite a los vendedores crear asociaciones comerciales, es decir, asociar una condición comercial existente y preaprobada a un cliente determinado. Parte crucial del formulario es el hecho de que se puedan definir precios distintos y montos mínimos a facturar solamente para los productos y no para los servicios, ya que esto va en línea con el plan de la empresa de tener planes estandarizados y asimilarse más a un SaaS tradicional.

Un punto importante de este flujo es el hecho de que si el vendedor cambia el precio predefinido del tramo de un producto, entonces se notificará mediante un correo como el de la figura 4.7 a los gerentes de esta propuesta, de manera de que esta propuesta pueda ser aprobada o rechazada. En caso de que el plan preaprobado no tenga cambios en los precios, este se aprobará de manera automática. Lo anterior fue un pedido específico por parte de los gerentes con el propósito de de poder tener mayor control sobre los tipos de planes que crean los vendedores para así evitar la generación de planes poco comunes.

El formulario mencionado se puede ver en la figura 4.8.

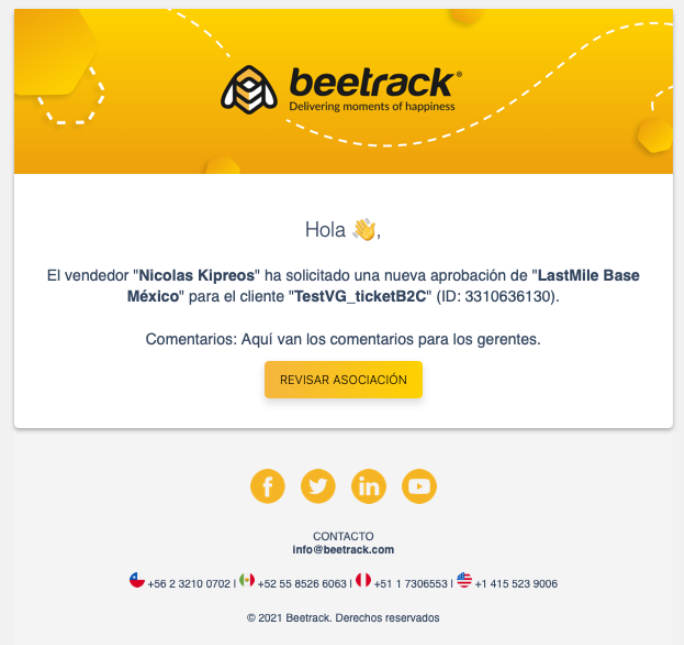


Figura 4.7. Correo electrónico de solicitud de aprobación de asociación comercial.

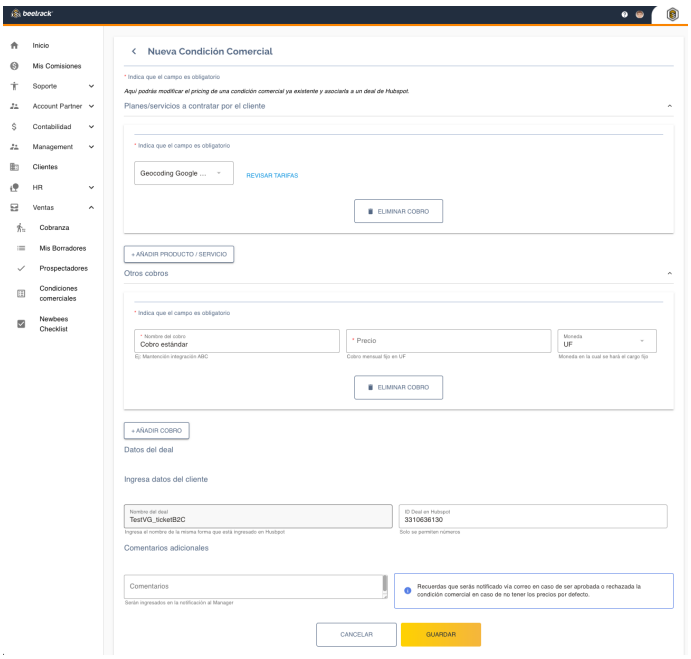


Figura 4.8. Vista general de condiciones comerciales preaprobadas.

En tercer lugar, el alumno programó una vista para los gerentes en la cual estos pueden aprobar o rechazar las propuestas de asociaciones comerciales creadas con precios no estándares. Esta vista se puede ver en la figura 4.10. Una vez que los gerentes aprueben o rechacen la propuesta, esto se le notificará al vendedor correspondiente el correo correspondiente de la figura 4.9.

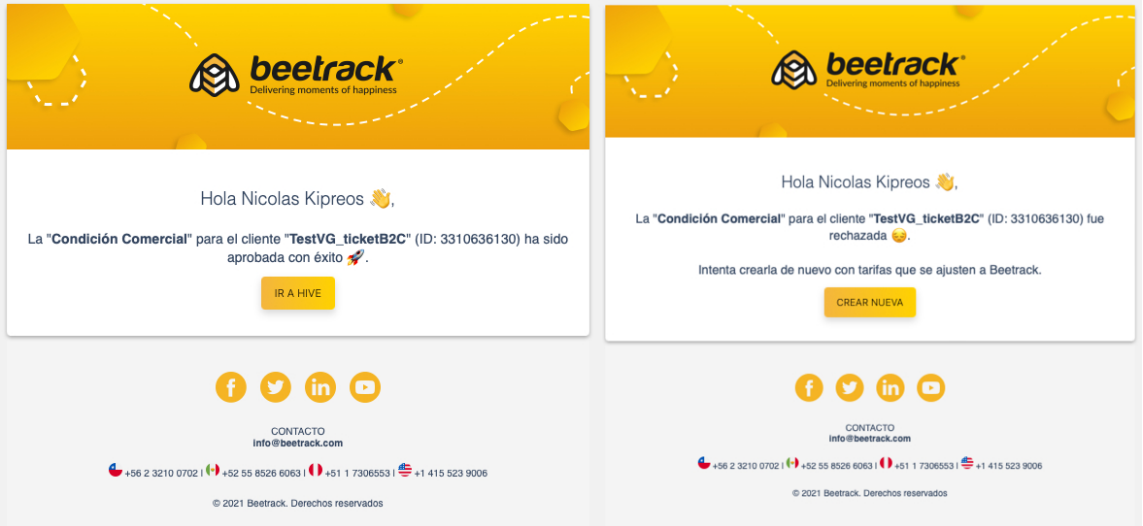


Figura 4.9. Correo electrónico de aprobación (izquierda) y rechazo (derecha) de asociación comercial.

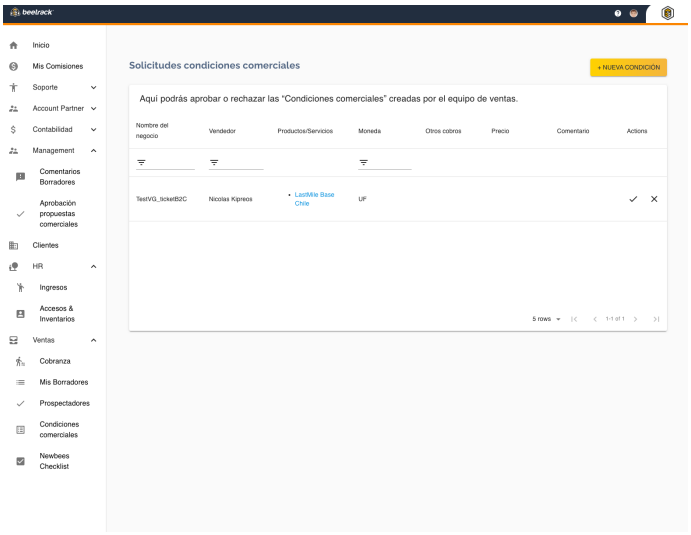


Figura 4.10. Vista de aprobación y rechazo de asociaciones comerciales.

En cuarto, y último luar, se realizó un formulario totalmente personalizado para la creación de planes no estandarizados, tanto para productos como servicios. Este formulario solo es accesible por los vendedores, sino que solamente los genertes tienen acceso. El propósito de este formulario es poder crear un plan totalmente a medida para un cliente

en caso de que un plan estándar no permita abarcar el tipo de contrato que se cerró. Esta decisión de restricción del formulario a los gerentes se generó principalmente para evitar el fomentar que los vendedores vendan el producto de manera no estandarizada y Beetrack se pueda acercar más a un SaaS tradicional. En la figura 4.11 se puede observar el formulario descrito anteriormente.

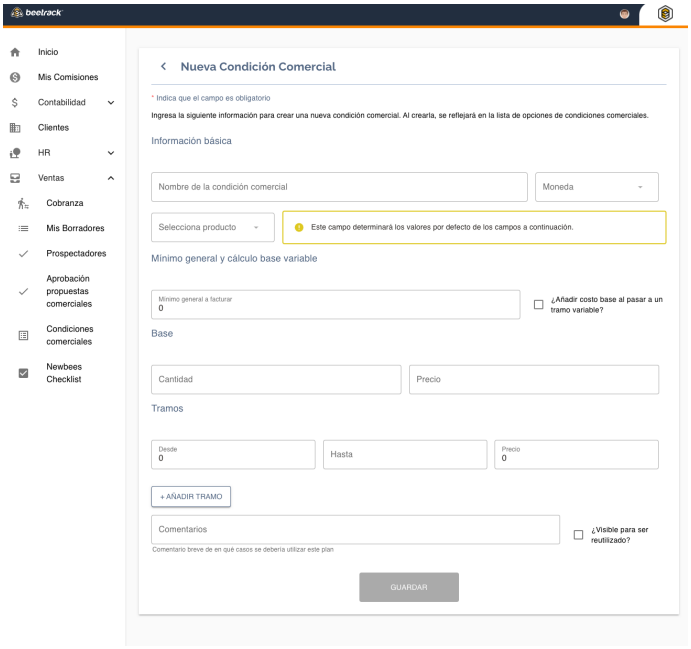


Figura 4.11. Vista general de condiciones comerciales preaprobadas.

4.3. Resultados

Los resultados obtenidos para este proyecto estuvieron por sobre lo esperado en un comienzo. En particular se logró lo siguiente:

1. Servicio independiente de *Hive*.

   Se creó un servicio independiente del ERP, que permitirá que se tenga la información descentralizada y se pueda enfocar en su única función de cálculo de costos y almacenamiento de planes de los clientes con tecnologías que no habían sido utilizadas dentro de la empresa anteriormente (GraphQL).

2. Creación de modelos de planes flexibles.

   Se logró diseñar modelos de planes y condiciones lo suficientemente flexibles de manera que abarcaran a todos los casos existentes y por existir. Junto a esto, la forma en que se abarcó el campo "`condition`" del modelo perimitió tener un *solver* flexible que se adaptase a la forma de resolver el csoto de los planes de manera correcta. Esta flexibilidad se logró mediante al trabajo entre múltiples áreas de la empresa, con especial foco en operaciones y ventas de manera de que se pudiese entender de mejor manera el problema y dolor existente. El esquema del modelo se logró definir gracias a un **análisis sistemático de los problemas** de los usuarios y trabajo en conjunto con las diferentes áreas de la empresa.

3. Servicio con 87,9 % de precisión de facturación.

   Se creó un servicio de resolución de condiciones comerciales independiente del estado de los clientes, que permite calcular el costo asociado al uso de cada cliente, por producto, por servicio y por moneda. Esto permitió que se tenga una facturación mucho más precisa que otros intentos para la resolución de este problema, subiendo en más de un 40 % la precisión de la facturación de más de 500 facturas desde el último intento que fue en Hubspot con una precisión de 43 %. Esto se pudo lograr mediante una batería de **simulaciones de soluciones** y aplicación de **conocimientos avanzados de ingeniería de software** de manera de crear un *software* escalable mediante diferentes patrones de diseño.

4. Formularios de creación de condiciones.

   El alumo programó las vistas de *frontend* dentro de *Hive* mediante el diseño en conjunto con *design ops*, resultando en los formularios especializados para los vendedores y el totalmente personalizable para los gerentes. Estos formularios quedaron **expresados en JavaScript** en el *framework* de React.

5. Vista de aprobación.

   Se desarrolló una vista especializada que permite que los gerentes de la empresa aprueben o rechacen las propuestas de asociaciones comerciales creadas por los vendedores en caso de que no sean tengan valores preaprobados. El objetivo de

esta vista es poder tener una visión rápida de lo que están haciendo los vendedores de la empresa y también poder tener un control fino de qué tipos de condiciones comerciales se tiene para los clientes.

6. Correos de estado propuestas comerciales.

   Se crearon 3 correos automáticos que se envían en diferentes momentos acorde a los cambios de estado que ocurran para los planes y para las propuestas de de asociaciones comerciales.

CAPÍTULO 5. CONCLUSIONES

5.1. Competencias evidenciadas

El trabajo realizado por el alumno como ingeniero de *software*, en Beetrack durante los cuatro meses evidencian de manera clara cada una de las tres competencias declaradas del perfil. En específico se pudo evidenciar lo siguiente:

1. Aplicar conocimientos avanzados de Ciencia de la Computación, Ingeniería de Software y Sistemas de Información para entender problemas complejos y abiertos.

   El estudiante desarrolló un sistema especializado con una arquitectura de software diseñada para ser escalable y replicable rápidamente mediante el uso de tecnologías como GKE, como también hizo uso avanzado de del *frmework* RoR utilizando validadores personalizados en los modelos. Para poder abordar los dos puntos mencionados anteriormente, el estudiante tuvo que comprender las diferentes problemáticas del problema y utilizar sus conocimientos en las diferentes áreas para abarcarlos.

   Por otro lado, para poder diseñar de manera adecuada, tanto los aparatos de *software* creados, el estudiante tuvo que aplicar conocimiento teórico de patrones avanazdos de diseño, de manera de que se pudiese tener un código altamente cohesivo y con bajo acoplamiento, algo que es sumamente deseado. Dentro de esta misma línea, el estudiante utilizó de manera amplia el *framework* de pruebas RSpec, lo que implica un uso de buenas prácticas de *software* que lo permite hacer más escalable, predecible y estable.

   Finalmente, el estudiante tuvo que utilizar conocimientos de sistemas de información, tales como la lógica de negocio de un ERP, que fue en lo que el alumno trabajó. Esto se traduce directamente en la combinación e integración que se realiza con Hubspot y Quickbooks, junto con el uso que *Hive* le da a nivel de lógica de negocio.

2. Diseñar y desarrollar modelos y artefactos de software y simular soluciones a problemas de la Ciencia e Ingeniería de Computación, cumpliendo con restricciones técnicas, sociales y éticas.
3. Analizar en forma sistemática las diferentes problemáticas de los usuarios, y diseñar productos o sistemas de software que queden expresados mediante algún lenguaje de programación, de acuerdo a los estándares de la ingeniería de software.

5.2. Conclusiones generales

REFERENCIAS

APD. (2019). *¿Cómo aplicar la metodología Scrum en tus proyectos empresariales?* Descargado 2021-11-08, de https://www.apd.es/metodologia-scrum-que-es/

Atlassian. (2021). *Gitflow Workflow.* Descargado 2021-11-08, de https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

Bascón Pantoja, E. (2004, 12). El patrón de diseño Modelo-Vista-Controlador (MVC) y su implementaciónn en Java Swing. *Acta Nova*, *2*, 493 - 507. Descargado de http://www.scielo.org.bo/scielo.php?script=sci_arttext&pid=S1683-07892004000100005&nrm=iso

Burotto, T. (2021). *Mejoras de procesos de negocio mediante el uso de tecnologías de la información: Hive (ERP interno).* Descargado 2021-10-21, de https://drive.google.com/file/d/1A_3ISZfxddopIL8OPwtdGeAmBQbz9j5O/view?usp=sharing

Corporateit. (2021). *Beetrack duplica su cartera de clientes y su facturación creció en más de un 80 %.* Descargado 2021-11-03, de https://corporateit.cl/index.php/2021/03/04/beetrack-duplica-su-cartera-de-clientes-y-su-facturacion-crecio-en-mas-de-un-80/

Fowler, M. (2006). *Codesmell.* Descargado 2021-11-15, de https://martinfowler.com/bliki/CodeSmell.html

IBM. (2021). *How does software testing work?* Descargado 2021-11-14, de https://www.ibm.com/topics/software-testing

IBM Cloud Education. (2021). *¿Qué es una API REST?* Descargado 2021-11-10, de https://www.ibm.com/cl-es/cloud/learn/rest-apis

Nishijima, Y. (2021). *Artemis: Ruby GraphQL client on Rails that actually makes you more productive.* Descargado 2021-11-21, de https://github.com/yuki24/

artemis

Red Hat. (2019). *What is GraphQL?* Descargado 2021-11-10, de https://www.redhat.com/en/topics/api/what-is-graphql

Red Hat. (2021). *Qué son las API y para qué sirven.* Descargado 2021-11-10, de https://www.redhat.com/es/topics/api/what-are-application-programming-interfaces

Refactoring Guru. (2021a). *Facade.* Descargado 2021-11-22, de https://refactoring.guru/es/design-patterns/facade

Refactoring Guru. (2021b). *Mediator.* Descargado 2021-11-22, de https://refactoring.guru/es/design-patterns/mediator