UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

REDISEÑO DE LA INFRAESTRUCTURA DE SOPORTE DE RESERVO.CL

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

ROBERTO IVÁN SAPIAIN CARO

PROFESOR GUÍA:
SERGIO OCHOA DELORENZI

MIEMBROS DE LA COMISIÓN:
JOSÉ PINO URTUBIA
JOSÉ BENGURIA DONOSO

SANTIAGO DE CHILE
2018

RESUMEN

REDISEÑO DE LA INFRAESTRUCTURA DE SOPORTE DE RESERVO.CL

Reservo.cl es una aplicación Web creada por la empresa SC3 SpA para reservar horas en consultorios médicos. Aunque esta aplicación está en producción y es exitosa, tiene muchas limitaciones para poder aumentar la tasa de atención de usuarios, lo cual limita su expansión en el mercado chileno, y eventualmente en el Latinoamericano. Por lo tanto, el objetivo de este trabajo de memoria es identificar los problemas que limitan su expansión, proponer soluciones para abordarlos, e implementar algunas de ellas. Particularmente se realizó: (1) un análisis de la infraestructura de soporte actual y de los puntos donde sería necesario intervenir el software, (2) un listado detallado de necesidades de mejoras a la aplicación y a la empresa, y (3) un diseño de la solución a cada una de las necesidades identificadas.

Algunos de los principales problemas identificados en el análisis fueron los siguientes: hay funcionalidades que ocupan muchos recursos, los datos se encuentran almacenados en una única base de datos, la aplicación no tiene capacidad de escalar, no se puede garantizar un cierto nivel de uptime, y se desconoce el nivel de vulnerabilidad de la aplicación ante ataques externos.

La gran mayoría de estos problemas son el resultado de la arquitectura monolítica que tiene actualmente la aplicación. Por lo tanto, para ayudar a paliar esta situación se definió una arquitectura basada en microservicios, que desacopla los componentes de software, dándole mayor flexibilidad, capacidad de evolución y de atención de transacciones a la solución. Los servicios de la nueva solución son implementados con servicios de Amazon AWS, lo cual permite obtener mayor escalabilidad y alta disponibilidad. Respecto a seguridad de la plataforma, la solución propuesta cuenta un nivel de seguridad bueno, pues está basado en componentes ya probados, los cuales pueden además ser configurados para implementar posibles mejoras.

Debido al amplio alcance del problema abordado y al limitado tiempo disponible para realizar el trabajo de memoria, algunas de las soluciones propuestas quedaron implementadas, otras en desarrollo y otras están sólo diseñadas. Sin embargo, todas ellas fueron evaluadas por expertos del área de software para asegurarse que son pertinentes para abordar los problemas planteados.

Dedicatoria

Dedico esta memoria, a todos quienes encontraron su propósito en esta vida, y sueñan con crear un mundo mejor.

Conócete a ti mismo.
-- *Oráculo de Delfos*

Cada gran sueño comienza con un soñador.
Siempre recuerda que tienes dentro de ti la fuerza, la paciencia
y la pasión por alcanzar las estrellas y cambiar el mundo.

-- *Harriet Tubman*

"Estás aquí, porque te has dado cuenta de que debes lidiar contigo mismo;
agradece a todos quienes te den la oportunidad de ello."

-- *G.I. Gurdjieff*

"¿Acaso sólo eres una brizna de pasto que será arrastrada por el viento?"

-- *Arngrim, Valkyrie Profile*.

Agradecimientos

A Raúl y María Milagro, mis padres; personas grandiosas que me apoyaron siempre. Isabel y Daniel, mis hermanos, por la vida que hemos tenido, llena de felicidad. A mi familia extendida: mis tíos y tías. María Eugenia, quien estuvo muchos años en mi infancia. A mi abuela materna María Graciela (Q.E.P.D), por los seis maravillosos años en que vivimos juntos.

Ramón Cruzat y el resto del equipo de SC3 SpA, por la posibilidad de aprender algo realmente trascendente a nivel profesional, como lo es sentar las bases tecnológicas y organizacionales para hacer crecer a nivel internacional una aplicación, y hacer ese aporte en la plataforma de 'Reservo'.

Agradezco al profesor Sergio Ochoa por su apoyo para concretar este trabajo, y por su paciencia y guía. Gracias a Cristián Rojas Poblete, por los detalles en el apartado de Seguridad Computacional. Gracias a profesores José Pino y José Benguria por los comentarios importantes respecto a computación en el ‘mundo real’.

Muchas gracias a OpenBeauchef por el contacto con la empresa SC3.

Gracias al Grupo de Sistemas del DCC, por haberme permitido años de valioso aprendizaje importante para mi carrera profesional, y que me sirvió para encontrar mi nicho principal de trabajo. Al Club de Rol de InJeniería (CRI), por la experiencia ganada organizando eventos, y por ser una ventana a un mundo de sueños y amistades excepcionales en el día a día.

Gracias a Gonzalo M., Luis, Leonardo, Juan Francisco, Yerko, Alejandra, Carlos, Andrés, Guillermo y María Luisa; primer año no habría sido lo mismo sin ustedes. Gracias Denisse, María Luz, Alain, Alex y Enrique por segundo año. Gracias a Jaime R., Alfredo, Guillermo, Nicolás, Cristian, Eduardo, Ricardo, José Fernando, María Francisca, Gonzalo L., Jaime S., Ignacio, César, y el resto de la gente del CRI mientras estuve ahí hasta el 2009. Gracias a Sandra y Rossana, ambas Q.E.P.D., por su apoyo en vida.

Gracias a Daniela Lillo Fiabane, por el tiempo que compartimos nuestras vidas y en el que aprendimos mucho el uno del otro, y porque ese tiempo me mostró lo que era vivir realmente.

Gracias a Francisca Gallardo Palacios por mostrarme el kung fu, aunque no lo haya vuelto a practicar en años. Por extenderme la mano cuando más lo necesité y no era capaz de darme cuenta de ello; y finalmente por mostrarme con tu forma de ser y tus actos, lo que es 'la esencia de la vida'.

Finalmente, gracias a quienes haya conocido en mi carrera, y se me haya olvidado mencionarles.

Tabla de Contenido

Índice de Figuras

Índice de Tablas

Capítulo 1: Introducción

En la actualidad las herramientas computacionales se utilizan para automatizar tareas repetitivas, muchas de ellas involucran cálculos aritméticos o actividades temporales como, por ejemplo, cumplida una cierta fecha y hora, lanzar una tarea programada o enviar un mensaje. Un uso común de estas herramientas es como apoyo a la gestión de un pequeño comercio de servicios, donde se requiere reservar una hora para atender al cliente como actividad principal.

Esto es aplicable, por ejemplo, a la atención de pacientes en centros de salud, o de ciudadanos que requieren atención personalizada por parte de algunos organismos de gobierno (por ejemplo, SENAMA, JUNAEB, Juzgados, etc.). En esos escenarios las herramientas computacionales permiten aligerar la carga de tareas repetitivas por parte de las secretarias o del personal de administración, que usualmente están a cargo de llamar por teléfono a los involucrados para confirmar horas de atención, enviar correos con los datos de la reserva de hora, sacar balances de los indicadores de atención, y realizar cálculos de comisiones en caso de que a los profesionales que prestan servicios se les pague de esa manera. En este escenario de trabajo se enmarca el desarrollo de esta memoria.

Particularmente, la empresa SC3 desarrolló la plataforma Web llamada *Reservo.cl*[1] (en adelante '*Reservo*'), que inicialmente tenía el objetivo facilitar la gestión de reservas de horas para atenciones médicas en centros médicos de Chile. Sin embargo, esta plataforma fue creciendo en funcionalidad y visibilidad en el mercado, y actualmente es una aplicación exitosa que entrega a sus usuarios los siguientes servicios:

- Implementación de una agenda.
- Envío de correos electrónicos para confirmar de horas de atenciones médicas.
- Reserva en línea de horas.
- Control de sesiones prepagadas por pacientes a centros médicos.
- Registro financiero.
- Cálculo de comisiones.
- Generación y envío de presupuestos.
- Generación de estadísticas y reportes.
- Implementación de ficha médica electrónica, en caso de que se requiera.
- CRM: Módulo de gestión de relaciones con clientes.
- Emisión de giftcards.
- Implementación de un mecanismo de fidelización de clientes, a través de un "Club de Puntos".

De los servicios anteriormente listados, los más críticos y esenciales son los siguientes: la agenda, los servicios para reserva en líneas de horas, y el mantenimiento de la correctitud en los registros financieros. En base a estos servicios se pueden calcular correctamente las comisiones asociadas a los profesionales que brindan parte del servicio, llevar los presupuestos contra gastos, y brindarle un buen servicio al cliente, entre otras actividades.

[1] '*Reservo*': https://www.reservo.cl

La empresa SC3 tiene ya más de 200 clientes en los poco más de 3 años que lleva operando. La gran mayoría de los clientes están ubicados en Chile, y unos pocos en el extranjero; principalmente en países latinoamericanos y en España. La mayoría de los clientes utiliza el servicio común de reservo, pero existen algunos que pagan adicional por instancias dedicadas sólo para uso de ellos. La arquitectura de la plataforma de apoyo es exactamente la misma en todos los casos, no así la capacidad de cómputo y la tasa de atención.

Dentro de las características que hacen a esta plataforma deseable para las empresas, están las notificaciones que recibe un cliente que tiene abierta la aplicación, las cuales son mostradas en tiempo real, de forma similar a las que utiliza Facebook tanto para usuarios desktop como móviles. Esto se implementó en *'Reservo'* con WebSockets, pero fue recientemente cambiado a utilizar un servicio llamado Scaledrone, que ofrece este servicio con escalabilidad automática ante un eventual aumento de carga sobre la plataforma, la cual funciona como un servicio Amazon (Amazon Web Services).

'Reservo' como servicio está justamente en la etapa en que debe comenzar a escalar a una mayor cantidad de clientes. Eso implica atender a más usuarios simultáneos, con más peticiones hacia el sistema, con mayor carga operativa en las diferentes componentes. Sin embargo, el software cuenta con diversos problemas de nivel técnico que requieren ser abordados efectivamente, para poder cumplir con los requerimientos de crecimiento que impondrá el negocio en el futuro cercano. Además, '*Reservo*' cuenta son algunos competidores en el mercado, como por ejemplo *AgendaPro* y *Dentalink*, los cuales aumentan la presión para que SC3 reduzca o elimine las limitaciones actuales de esta plataforma.

Para tratar de identificar los puntos clave de mejora de la plataforma, a fin de poder enfrentar la operación en un escenario internacional, el autor de esta memoria realizó un análisis exhaustivo de la plataforma de soporte de la aplicación y del software mismo. En la siguiente sección se describen los problemas identificados en *'Reservo'*, cuya solución también hacen parte de este trabajo de memoria.

1.1 Problemas identificados en *'Reservo'*

A continuación, se indican los siete problemas de la plataforma, los cuales no le permiten a *'Reservo'* crecer y consolidarse en un escenario de mayor demanda de servicios. Particularmente para llevar la actual plataforma '*Reservo*', desde el plano de operación local, a un escenario internacional.

1.1.1 Gestión de zonas horarias

La plataforma *'Reservo'* presupone que un cliente de SC3 (por ejemplo, una empresa de estética o de salud), puede tener varias sucursales físicas; sin embargo, estas sucursales estarán ubicadas en el mismo país, con una zona horaria común. Por lo tanto, los pacientes que reservan hora se encuentran ubicados en la misma ciudad o zona metropolitana que los centros que usan *'Reservo'* para su gestión. Por el momento se considera como excepcional el caso de la región de Magallanes, con su zona horaria diferenciada.

Como se mencionó antes la empresa SC3 desea expandir el servicio hacia varios países, principalmente hispanoparlantes, pero cuenta con una limitación importante en la coordinación de los husos horarios. Particularmente se debe soportar múltiples zonas horarias en la plataforma, y la implantación para cada Cliente de SC3 debe funcionar de acuerdo al país en que reside.

En la actualidad la plataforma funciona con el servidor sincronizado a la hora de Chile Continental, y los eventos programados se lanzan en concordancia. Por un tema de diferencia horaria, para clientes de España, por ejemplo, los correos que actualmente se envían a las 8:00hs (hora chilena), ellos los ven desde las 12:00 a 14:00hs dependiendo de la diferencia horaria. Esa es una inconsistencia que debe ser solucionada si se desea una expansión en serio que considere al mercado hispanoparlante.

1.1.2 Arquitectura más bien monolítica

Actualmente en la aplicación se involucra elementos interconectados entre sí, que corren (se ejecutan) en la misma máquina para efectos prácticos, aunque con un uso de servicios externos que permite que el servidor se enfoque sólo en procesamiento de transacciones, más que en recuperación y entrega de los datos.

La separación existente involucra tres capas: capa de presentación (*front-end*), y las capas de procesamiento y de datos (*back-end* y *capa de datos*). Las primeras dos se manejan en una sola máquina que hace procesamiento y generación de vistas, y la última capa que corre en otra máquina operando con una base de datos dedicada.

Existen casos específicos de servicios de procesamiento (por ejemplo, generación y envío de correos) que requieren mayores recursos, debido al uso de componentes que no están dentro de la aplicación, lo que hace que atender más clientes requiera mayores prestaciones en la máquina usada. Aun así, existen límites de lo que puede procesar una sola instancia del servidor, además de volverse innecesariamente un punto de falla y un cuello de botella.

1.1.3 Alto uso de recursos

A medida que ha ido aumentando la cantidad de clientes, debido a cómo está estructurada la aplicación y como consecuencia del problema anterior (descrito en la sección [1.2.2]), se observa un colapso en el uso de recursos de la máquina que efectúa las labores de *back-end* y generación de vistas para cliente.

El colapso se suele manifestar mediante la ralentización de la plataforma, caída de ciertos procesos específicos, los cuales deben ser monitoreados constantemente para levantarlos rápidamente cada vez que se caen.

1.1.4 Concentración de clientes en una sola base de datos

La aplicación Web fue diseñada originalmente de forma tal que todos los clientes están en una sola base de datos y su información también. A nivel técnico detallado hay una tabla que guarda toda la información del registro de horas, la cual sólo crece y crece en el tiempo; y así para todos

los datos de cada cliente. Esta forma de manejar los datos se conoce como ‘Tenencia Simple’ (*Single-Tenancy*), y es altamente riesgosa.

Adicionalmente durante el 2016 se observó un incidente con un cliente que por error creó 5 millones de citas, lo cual a nivel técnico hizo muchas operaciones innecesarias sobre la base de datos. Particularmente, cada vez que se mostraba una agenda de un cliente, se efectuaban varias operaciones JOIN, las cuales requieren espacio de disco temporal en la instancia de base de datos, el cual se llenó y ralentizó excesivamente la operación de la plataforma.

Una vez que el problema fue solucionado, se realizaron diversas mitigaciones para no volver a caer en este problema. Sin embargo, la forma de manejar los datos sigue siendo frágil, debido a que la información de todos los clientes está en la misma base de datos.

Por lo tanto, se corre riesgo muy alto al realizar ciertas operaciones, y al efectuar respaldos. Además, se toman bloqueos de exclusión mutua a nivel de tablas y/o base de datos, demorando el servicio completo cuando hay clientes agregando datos a la hora que se efectúa el respaldo.

1.1.5 Escalabilidad limitada

Debido al diseño monolítico, y debido a los temas detallados en las secciones [1.1.2], [1.1.3], y [1.1.4], ante problemas de capacidad de máquina, la primera solución es poner más recursos computacionales de hardware a disposición de la aplicación.

Sin embargo, este enfoque tiene límites a nivel de piezas de software, sistema operativo, y en algunas componentes (como la base de datos) se llega a un tope por límite de conexiones concurrentes. En este documento se identifican los puntos de quiebre que demoran o impiden efectuar un escalamiento del sistema a nivel de tasa de atención de transacciones, y además se plantean soluciones a ellos, de acuerdo a lo utilizado actualmente por expertos implementadores en la industria de *Software as a Service*.

1.1.6 Uptime no considerado en la implementación de la aplicación

La aplicación Web se implementó en forma monolítica, como prácticamente todos los productos mínimos viables para solucionar un problema, y en su evolución se han hecho supuestos que no consideran posibles fallas de componentes o de comunicación. Con esto se puede concluir no se abordó, desde el diseño original, el tema del *uptime* (o disponibilidad) de la aplicación. Por lo tanto, desde el punto de vista de la estructura del sistema, no hay una garantía razonable de que el servicio estará disponible el cliente cuando lo solicite.

Poco a poco, y a través del proceso de corrección de errores del sistema, se han ido implementando mecanismos para evitar que el sistema deje de estar disponible, sin que la empresa SC3 lo sepa. Sin embargo, estas son soluciones parche, pues el uptime de la aplicación no está considerado a nivel del diseño estructural de la misma.

El tema de una posible falla de servicio, a pesar de que no se pierdan datos, genera un problema de imagen para SC3, que puede derivar en que clientes decidan irse. Además, al tener la empresa más de 200 clientes, en caso de un error atribuible al proveedor, la empresa SC3 entra en

un modo de control de daños, atendiendo por teléfono y e-mail a muchos clientes en poco tiempo. Lo cual implica su respectivo seguimiento, lo que hace además que se deban dejar de lado muchas actividades para atender la crisis.

Adicionalmente, no se considera la alta disponibilidad del servicio en el diseño; esto a nivel de proveer un servicio es crítico. Tampoco permite hacer pasos a producción 'en caliente'; es decir, sin pérdida de servicio para usuarios que estén en la plataforma. En otras palabras, no hay forma de garantizar la operación continua de la plataforma.

1.1.7 Desconocimiento del estado de la seguridad de la aplicación

Además de lo antes mencionado, existe un desconocimiento sobre el estado de la seguridad de la plataforma, en cuanto al riesgo de exposición ante un 'hackeo' o intrusión. Existen componentes que son gestionadas y tienen actualizaciones automáticas, otras que requieren intervención manual, y otras que posiblemente requieren una migración completa de la aplicación, lo que implica un pago de deuda técnica acumulada.

Empíricamente se ha demostrado que cada vez que se proclama a un sistema informático como el más seguro del mundo, existe alguna forma de vulnerarlo que será explotada; incluso si son sistemas de instituciones que usan alta tecnología con expertos especializados, como son la NSA, la CIA, o las Fuerzas Armadas de países del primer mundo.

Para determinar el nivel de vulnerabilidad de la aplicación web *'Reservo'*, se analizan tres ejes de trabajo: Diseño, Desarrollo, Operación. Cada uno de ellos incluye algunos puntos que se abordarán, y se indica lo que se debe efectuar para mitigar o solucionar dichos problemas.

A nivel de prácticas utilizada por la empresa SC3 en el ciclo de desarrollo, se verá si se debe agregar algunas metodologías que impacten en la operación del servicio, como por ejemplo la *integración continua*, que permite hacer pruebas automatizadas y verificaciones de uso de dependencias sin vulnerabilidades críticas, e incluso automatizar pasos a producción, sin tener pérdida de servicio, con un solo click.

1.2 Justificación del trabajo

Una investigación sobre el potencial de crecimiento de la empresa SC3, indicó que el servicio *'Reservo'* puede atender a muchos clientes, en muchas partes del mundo, aun manteniéndose dentro del universo hispanoparlante. Sin embargo, para llegar a cubrir ese segmento del mercado es necesario pensar en un servicio como el de Spotify, Salesforce, Netflix o Amazon: cuentan con sitios en Internet que atienden desde varias partes del mundo a clientes en todo el globo terráqueo, con características específicas de locación, y perfiles de cliente por país, entre otras cosas.

Este es un problema que dichas empresas ya enfrentaron para hacer crecer su negocio, para el cual diseñaron en forma específica una solución de acuerdo a sus necesidades, con el patrón común de considerar el uso total o parcial de:

- *Arquitectura orientada a microservicios [1]*. Esto implica tener servicios específicos que hacen una sola cosa, en forma genérica, con bajo acoplamiento entre sí. Esto permite reutilizar código y mejorar la mantenibilidad de los sistemas.

- *Aplicación Multi-Tenant (Tenencia Múltiple) [2]*. Esto implica tener una sola instancia de software que sirva a múltiples clientes, pero manteniendo la privacidad entre ellos, de forma tal que un cliente no pueda ver los datos asociados a otro.

Las aplicaciones creadas para dar un servicio a muchos clientes (por ej. Amazon, Netflix, Spotify, Salesforce), usan estas dos características como puntos fundamentales. Es decir, usan microservicios para gran parte del procesamiento de tipo *back-end* invisible al usuario en forma directa, y son sistemas que por el volumen de datos se organizan como uno de tenencia múltiple. Además, las arquitecturas de estos sistemas están pensadas para que el servicio que entregan cuente con alta disponibilidad, seguridad, resiliencia y escalabilidad.

Las dos características ya mencionadas son esenciales para soportar el crecimiento de estas aplicaciones en el mediano y largo plazo. Además, les permite a las empresas crecer su base de usuarios, mantener la calidad y a sus usuarios contentos, lo que da sustentabilidad al servicio en el tiempo.

1.3 Objetivos de la memoria

El objetivo general de este trabajo de memoria es definir, diseñar e implementar mejoras para la plataforma Web 'Reservo.cl', las cuales deben satisfacer el requerimiento de negocio principal: "Permitir aumentar la cantidad de clientes que se atienden, sin que se note degradación de servicio y dejar las bases declaradas para una expansión internacional a mayor escala".

En la siguiente sección se establece claramente cuáles de los problemas planteados en la sección [1.1] son resueltos completamente, y cuáles en forma parcial. Para aquellos solucionados parcialmente se entregará además una hoja de ruta a seguir como trabajo a futuro.

1.4 Alcance de la memoria

Respecto a los problemas identificados en [1.1], se detalla a continuación hasta qué punto serán abordados en este trabajo. La aplicación para clientes de diferentes países, con sus respectivas unidades monetarias, funciona en forma transparente y no será abordada en este trabajo, debido al supuesto de que los Clientes funcionan con sus Pacientes en forma local en una misma ciudad, por lo que no sería necesario convertir divisas.

Primero, se presenta una matriz de resumen que muestra para cada detalle, en qué estado quedó al final de este trabajo de título cada uno de los puntos. Los estados posibles son los siguientes:

- *Definido*: Significa que se hizo una propuesta para abordar el punto. Esto implica realizar un análisis del estado actual y luego detallar cómo mejorarlo.

- *Diseñado*: Hay una propuesta de solución que se encuentra lista para ser implementada.
- *En Implementación*: Significa que la solución fue diseñada, y actualmente se la está implementando y probando.
- *En Producción*: A la fecha de entrega se encuentra funcionando ya para 'Reservo'.

A continuación, se presentan las tablas que resumen el alcance de la memoria en cada uno de los siete puntos indicados en la sección [1.1]. Los elementos relacionados con la seguridad (punto 7) se explican aparte, puesto que consideran el Ciclo de Desarrollo de SC3, y los aspectos de Diseño, Implementación y Operación del servicio 'Reservo'. La siguiente tabla muestra un resumen del alcance de la memoria, y el nivel de logro de los seis primeros puntos de mejora.

Tabla 1: Resumen del alcance del trabajo de memoria

Ítem / Sub-Ítem	**Definido**	**Diseñado**	**En Implementación**	**En Producción**
1. Zona horaria				
Correos programados			X	
Display en cliente	X			
2. Definición nueva arquitectura			X	
3. Alto uso de recursos				
Funciones back-end relacionadas con e-mails vía Lambda			X	
Remoción de Celery y Redis			X	
4. Base de Datos				
Multi-Tenancy		X		
Soportar tenencias simple y múltiple en paralelo		X		
Plan de migración	X			
5. Escalabilidad				
Front-end 'Reservo'		X		
Back-end 'Reservo'			X	
Base de Datos		X		
6. Alta Disponibilidad				

	Diseño de arquitectura		X		
	Recuperación rápida ante problemas de capa de datos		X		
	Recuperación ante desastres		X		

Las tablas 2 y 3 muestran el resumen del trabajo realizado en el aspecto de seguridad de la plataforma; primero las prácticas del ciclo de desarrollo, y luego las prácticas de en diseño, implementación y operación de '*Reservo*'. Posteriormente se explica, en las subsecciones correspondientes, el detalle de cada uno de los puntos abordados.

Tabla 2: Prácticas incorporadas al ciclo de desarrollo usado por SC3

Ítem / Sub-Ítem	**No Existe**	**Definido**	**Diseñado**	**En Implementación**	**En Producción**
7.1 Prácticas Ciclo Desarrollo					
Metodologías Ágiles, Lean					X
Control de Versiones					X
Test Driven Development	X				
Integración Continua	X				
Entornos Dev/Test/Stage/Prod		X			
Detectores de patrones de programación insegura	X				

Tabla 3: Estado de Seguridad en Diseño, Implementación y Operación de '*Reservo*'

7.2 Seguridad Aplicación: Diseño	**No Usado**	**Investigado**	**En Implementación**	**En Producción**
Roles de Usuario				X
Autenticación				X
Uso Segundo Factor	X			
Almacenamiento seguro de claves				X
Gestión de Riesgo ante cambios en Aplicación				X
7.3 Seguridad Aplicación: Desarrollo				

	Manejo Versiones Django y Dependencias		X		
	Controles de Seguridad de Código Fuente		X		
	Pruebas contra estándar OWASP	X			
7.4 Seguridad Aplicación: Operación					
	Uso de SSL				X
	Uso de Firewall WAF		X		
	Pasos a Producción después de aprobadas pruebas OWASP	X			
	Configuración de Servicios de Amazon segura.			X	

1.4.1 Respecto al problema de correctitud de zona horaria desde el punto de vista del software

En esta memoria se definen todos los casos que requieran tener correctitud en la zona horaria, de acuerdo con las funcionalidades del software 'Reservo'. Las primeras que son identificables son las de los correos programados, como parte de las características ofrecidas a Clientes: envío de correos recordatorios a las 8 AM, pero en cada país; para esto se diseñará e implementará una solución.

La implementación permite considerar independencia del huso horario, y dar servicio a Clientes en cualquier parte del mundo. Cuando se notifique la hora de algo al usuario, debe indicarse la hora local y su diferencia horaria contra UTC (Tiempo Universal Coordinado). También permitiría considerar el caso de los clientes en la Región de Magallanes, siempre y cuando sea abordado ese caso en la aplicación 'Reservo', por SC3; solucionar este caso permite abordar casos de un mismo país con zonas horarias diferentes como lo son los Estados Unidos de Norteamérica, Canadá, algunas zonas del Reino Unido, Rusia, entre otros.

Se identificarán a nivel de funcionalidades en la interfaz de usuario, los puntos donde se requiere intervenir en código fuente de '*'Reservo"* para abordar el tema de los cambios de horario. Particularmente, se busca mostrar las zonas horarias correctas para cada país, y dimensionar el esfuerzo de realizar los cambios correspondientes. Por razones de tiempo y de prioridad en la implementación de funcionalidades de la plataforma, esta funcionalidad no será implementada como parte de este trabajo de título.

1.4.2 Respecto al problema de correctitud de zona horaria desde el punto de vista de la infraestructura

En este documento se reporta la solución a este problema, a través de la implementación de funciones que utilizan Celery y Redis, mediante llamadas a los siguientes servicios de Amazon[2]: *Lambdas, DynamoDB* (con y sin *Streams*), *SNS, SQS.*

Celery y Redis son componentes que usan muchos recursos en las máquinas actuales. Removerlos libera al servidor, para atender más clientes en forma más eficiente. Además, reduce componentes monolíticas, permitiendo que la escalabilidad sea más simple de lograr y con menores riesgos, que se traducen en demoras en el desarrollo o posibles incidentes al poner esa funcionalidad en producción.

1.4.3 Respecto al problema de arquitectura monolítica

En este documento se plantea un rediseño de arquitectura que reduce la parte monolítica a lo estrictamente necesario; o sea a la generación de vistas de la página Web una vez procesados los datos. Esto se logra mediante uso de microservicios, que deja partes de la implementación fuera del servidor de aplicación, de forma tal que la creación de pequeños procesos sea en respuesta a eventos en forma elástica. Esto además reduce el tamaño del monolito, dejando sólo lo estrictamente necesario para la generación de vistas; en principio, removiendo tareas de *back-end* que impliquen un procesamiento extendido. Esta componente puede ser replicable para temas de balance de carga en forma simple para atender más usuarios.

Por otra parte, el uso de esta arquitectura además permite sentar bases para solucionar problemas adicionales, como el uso de recursos, la concentración de las bases de datos de clientes, la escalabilidad de aplicación, la atención con alta disponibilidad, y los controles de seguridad por diseño e implementación. Todos estos puntos de mejora se explican en las siguientes subsecciones. El diseño de la nueva arquitectura fue explicitado, y se hizo un plan de trabajo para iniciar implementación de la misma.

1.4.4 Respecto a la concentración de clientes en una sola base de datos

La solución al tema de la concentración de información en una sola base de datos relacional, consiste en pasar a un esquema de 'Tenencia Múltiple' (*Multiple-Tenancy*), en el cual se usan:

- Una base de datos para cada cliente, con nombre e ID; incluso es posible separarlas en servidores diferentes.
- Una '*meta base de datos'* que para cada base de datos indica: tipo de cliente (ST o MT), los puntos de conexión, y parámetros de autenticación. Incluso desde diseño de la nueva arquitectura se puede considerar usuarios de sólo lectura, y otros con diferentes privilegios. Así se puede acceder a la funcionalidad de la plataforma en forma diferenciada según necesidad.

[2] En el Capítulo 2 están definidos todos los servicios mencionados.

En este documento se plantea una hoja de ruta que consiste en crear un diseño, compatible con '*Multi-Tenancy*', que no implique efectuar grandes cambios en plataforma actual. Dicho esquema permite, mediante la configuración de un parámetro N, que indica cuántos clientes van en una base de datos, y mantener los clientes existentes actualmente tal como están. En el momento que ese parámetro sea ajustado al valor N = 1, todos los clientes creados estarán en su propia base de datos. Ajustado de esa manera, será posible garantizar que cada cliente tendrá acceso sólo a los datos que legítimamente le corresponden.

1.4.5 Respecto a la escalabilidad limitada

Como se mencionó antes, en este documento se aborda el tema de la escalabilidad de la aplicación, en los siguientes tres ejes:

- *Vista web de usuario*: Se diseñó e implementó una solución que permite escalar la plataforma, y adicionalmente permite hacer pruebas de cambios después de pasar por Desarrollo y Pruebas.

- *Procesamiento Back-end*: Se diseñó y está en proceso de implementación una solución para aumentar la capacidad de atención de transacciones (principalmente procesamiento back-end) de la plataforma. La solución es escalable en forma automática mediante Amazon *Lambda*.

- *Base de Datos*: Se diseñó una solución usando técnicas de replicación y balance de carga, la cual fue estudiada en detalle. Esta solución permite abordar mejoras de desempeño, escalabilidad, alta disponibilidad, y recuperación ante desastres.

1.4.6 Respecto a la no consideración del uptime de la aplicación

En este documento se explica el diseño que se debe seguir para que la plataforma de 'Reservo.cl' cuente con alta disponibilidad. Para ello se propone un plan de recuperación rápida de aplicación, en otro proveedor, en caso de que haya problemas en el principal. Esto además permite asegurar la capacidad de reconstrucción de la plataforma, y su último estado de trabajo en caso de un desastre mayor. La implementación parcial o completa de las soluciones no abordadas en la memoria depende exclusivamente del interés y esfuerzo de SC3.

Para impactar positivamente el uptime de la plataforma, se propone realizar el paso a producción 'en caliente'; esto es, sin pérdida de servicio para los usuarios. Esto se logra con el balanceador de carga *ELB.*

1.4.7 Respecto al desconocimiento del estado detallado de la seguridad de la aplicación

El apartado de seguridad está estructurado en dos puntos: (1) prácticas del equipo de desarrollo, y (2) la aplicación en sí. Ambos se explican brevemente a continuación:

- *Prácticas del Equipo de Desarrollo*: Se propone el uso de una metodología ágil y principios Lean, reuniones Scrum internas, y versionamiento del código fuente utilizando Github. Se identifica la falta de técnicas de revisión automatizada de código, integración continua, y

entornos de desarrollo con detección de patrones de programación insegura. Adicionalmente se ve la cantidad de personas requerida para el desarrollo, y también aquellas necesarias para operar el servicio.

- *Seguridad en la Aplicación*: Este apartado se separa en tres ejes, cada uno con puntos específicos.

    1. *Diseño*:
        - *Gestión de Usuarios*: Roles, autenticación, mecanismos de segundo factor (2FA), almacenamiento seguro de claves.
        - *Gestión de Riesgos* en el diseño de aplicación y sus cambios.
    2. *Desarrollo*:
        - Versiones de Django y sus dependencias Python.
        - Control de Seguridad de Código.
        - Pruebas OWASP.
    3. *Operación*:
        - *Uso de SSL*: Se encuentra implementado y configurado con nivel seguro según el test automatizado de Qualys (SSL Labs), con nota A+, el máximo posible.
        - *Resistencia a Intrusiones:* Se cuenta con Web Application Firewall (WAF) y controles de seguridad en desarrollo que ayudan a mitigar posibles intrusiones (accesos no autorizados).
        - Configuración de Amazon Web Services en los puntos y servicios necesarios, de acuerdo al diseño de la aplicación.

Capítulo 2: Marco Teórico

Para efectos de este documento, en la sección [2.1] se definen los términos específicos que se usarán en él, tal cual como están escritos, incluyendo las comillas. En la sección [2.2], se detallan los términos técnicos pertenecientes a las Ciencias de la Computación.

2.1 Terminología

A continuación, se definen los términos que será necesario conocer a fin de comprender a cabalidad el trabajo realizado en esta memoria.

- *'AWS'*. Es el acrónimo de Amazon Web Services.
- *'Cliente'*. Es un usuario de reservo.cl (cliente de SC3), usualmente un centro de belleza, consulta médica, o centro deportivo.
- *'Endpoint'*. Representa un punto de conexión a algún servicio de red.
- *'Multi-Tenant'*. Referente a Multiple Tenancy (Tenencia Múltiple), definida en sección 2.2.11.
- *'Paciente'.* Es un cliente de los 'clientes' de SC3. O sea, no es el cliente directo de SC3 o el usuario directo de 'Reservo', pero si se ve afectado por las acciones de los 'clientes' en la plataforma.
- *'Reservo'*. Es la plataforma de software como marca, o producto.
- *'SC3'.* Empresa SC3 SpA, propietaria de 'Reservo' (Saintard, Concha, Cruzat y Cruzat SpA).
- *'Single-Tenant'*. Referente a Single Tenancy (Tenencia Simple), definida en sección 2.2.10.

2.2 Conceptos de computación

A continuación se detallan los conceptos relacionados con las Ciencias de la Computación, tecnologías, metodologías, entre otros.

2.2.1 Sistemas distribuidos

Un sistema distribuido[3] es una colección de computadoras independientes, que colaboran entre sí a través de redes de datos, con el propósito de efectuar una o más tareas, y que ante sus usuarios aparenta ser un único sistema. Estos sistemas surgen como respuesta a la necesidad de responder a un mayor número de usuarios distribuidos en diferentes áreas geográficas, incluir tolerancia a fallas, y crecer en forma simple a medida que se incrementan los usuarios. Ejemplos de estos sistemas son, una red de estaciones de trabajo de la universidad, o el timbrado de facturas y boletas electrónica en Chile, o los diferentes servicios de computación en la nube (*Cloud Computing*).

2.2.2 Computación en la nube

Según el NIST[3] de los E.E.U.U. de América, este es un modelo que permite el acceso por demanda, de forma distribuida y simple, a recursos computacionales compartidos y configurables (por ej., a redes, servidores, almacenamiento, aplicaciones y servicios), que pueden ser puestos en servicio en forma rápida y liberados con muy poco esfuerzo de administración o interacción con personal del proveedor de servicios[4]. La computación en la nube es un ejemplo de *Sistema Distribuido*, que además debe cumplir con cinco características esenciales: autoservicio en demanda, acceso amplio a través de la red, *pooling* de recursos, elasticidad rápida, y servicios medidos cuidadosamente. Existen varios proveedores de servicios cloud, siendo los más relevantes Amazon Web Services (AWS), Google Cloud, Microsoft Azure, IBM SmartCloud, Oracle Cloud.

2.2.3 Arquitectura monolítica

Arquitectura monolítica es aquella arquitectura de software donde un solo programa efectúa todas las tareas necesarias para su funcionamiento, salvo las que explícitamente se hayan puesto en componentes aparte, como por ejemplo bases de datos. En estas arquitecturas las componentes lógicas están íntimamente relacionadas, por lo que hay un acoplamiento muy cerrado[5]. La Figura 1 muestra un ejemplo de una arquitectura monolítica.

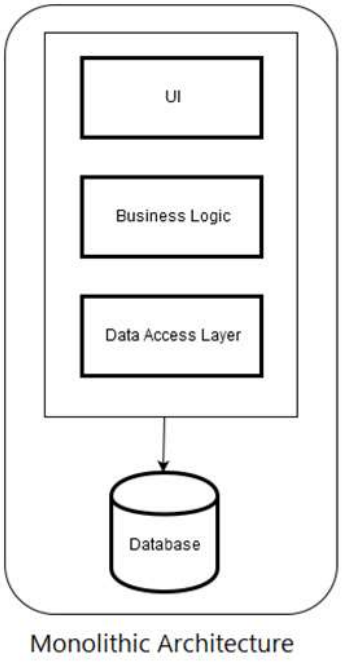

Figura 1: Ejemplo de arquitectura monolítica

[3] NIST: National Institute of Technology Standards.

2.2.4 Arquitectura de microservicios

Una arquitectura de microservicios es una arquitectura de software, en la que una aplicación es abordada en su desarrollo como una colección de pequeños servicios, cada cual corriendo en su propio proceso y comunicándose entre sí mediante mecanismos ligeros [1]. La idea de usar estas arquitecturas es para mejorar la reutilización de código fuente, la mantenibilidad de la aplicación a medida que crecen los equipos de desarrollo, reducir los tiempos de entrega para puesta en producción, y permitir la respuesta a eventos en forma simple y autónoma. Este es un enfoque muy utilizado en muchas componentes de aplicaciones que corren en plataformas de Cloud Computing, como Netflix, Spotify, Salesforce y Facebook. La Figura 2 muestra un ejemplo de arquitectura de microservicios.

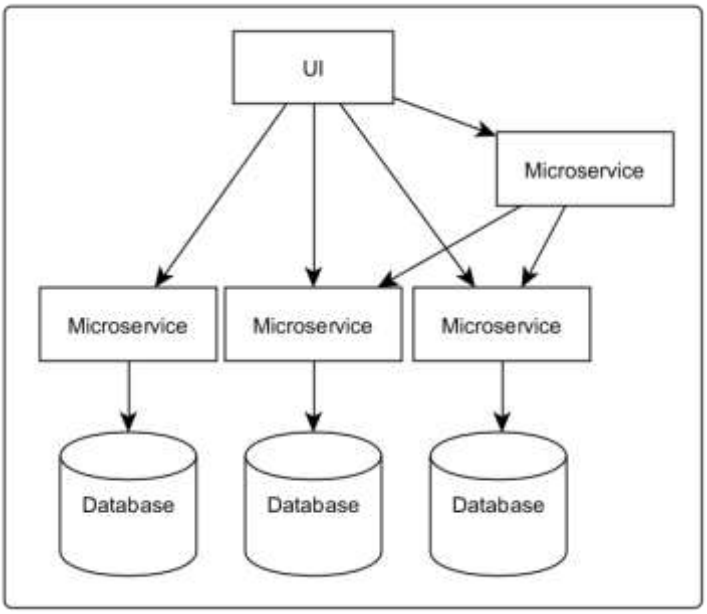

Figura 2: Ejemplo de microservicios

2.2.5 Arquitectura serverless

Serverless es una arquitectura de Infraestructura Computacional, pensada en abstraer al desarrollador del provisionamiento y administración de las plataformas en que corren sus aplicaciones y/o servicios, de forma tal que le permita enfocarse sólo en crear sus aplicaciones. Los microservicios son implementados utilizando plataformas serverless, las cuales son otro caso más de sistemas distribuidos. Ejemplos de dichas plataformas son: Amazon Lambda y Google Functions.

2.2.6 Replicación

Visto a nivel de datos, la replicación representa la capacidad de un sistema de mantener duplicada su información, usando almacenamiento independiente y redundante. Los sistemas de

origen y destino son, para todo efecto práctico, independientes entre sí. La replicación de información puede ser síncrona, asíncrona, semi-síncrona, o periódica por lotes [7].

2.2.7 Alta disponibilidad

Según la SNIA[4], alta disponibilidad se define como *la habilidad de un sistema de desempeñar su función en forma continua, por un periodo de tiempo significativamente mayor al que sugerirían las confiabilidades de sus componentes individuales. Se logra usualmente mediante tolerancia a fallas, y no es fácilmente definible. Tanto los límites de un sistema definido como 'altamente disponible' y el grado al que su disponibilidad es extraordinaria, debe ser claramente entendido caso a caso [7]*.

Para efectos prácticos, consideramos que un sistema tiene alta disponibilidad cuando ante falla de alguno de sus componentes o canales de comunicación, puede mantenerse funcionando y sirviendo a sus usuarios. En la sección [2.2.17] se explica el concepto de "porcentaje de disponibilidad", que es la unidad utilizada para medir *uptime*[5] en centros de datos, y que usualmente forma parte de un Service Level Agreement (SLA).

2.2.8 Escalabilidad

Se entiende por *escalabilidad* a la capacidad de un sistema, red o proceso, para gestionar una cantidad creciente de trabajo, o su potencial para acomodar dicho trabajo. Para cada caso particular (es decir sistema específico) es necesario definir los requerimientos de escalabilidad que sus componentes deben cumplir, para así poder determinar la escalabilidad esperada para el sistema [8].

2.2.9 Seguridad Informática

Se define Seguridad Informática como la práctica de prevenir accesos no autorizados, el uso, divulgación, disrupción, alteración, inspección, copia o destrucción de datos, información, conocimiento o sistemas informáticos. Esto es independiente de si los datos están en formato físico, o electrónico.

2.2.10 Tenencia Simple

Este concepto usualmente se usa para representar la forma de la tenencia de datos en instancias dedicadas para un cliente; usualmente, se refiere a tenencia de datos en las dependencias de éste. A nivel de Bases de Datos, se reconoce como Tenencia Simple (o Single Tenancy), a las instancias donde se tiene uno de estos dos casos:

---

[4] SNIA: Storage Network Industry Association (http://snia.org/dictionary).
[5] Nivel de funcionamiento ininterrumpido.

- Cada cliente cuenta con su propia instancia de la aplicación, en su sitio, con una base de datos exclusiva.

- Se cuenta con una sola base de datos dedicada, en la cual están todos los datos de todos los clientes que uno tenga, sin mayor separación física o lógica entre ellos.

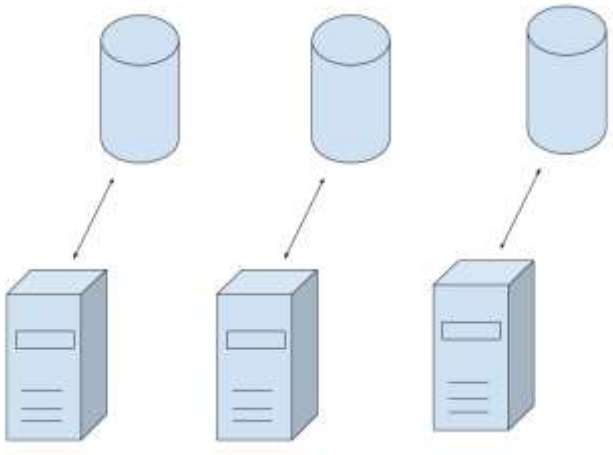

Figura 3: Ejemplo de Single-Tenancy para diferentes clientes

2.2.11 Tenencia Múltiple (Multiple Tenancy)

En el contexto de aplicaciones, el concepto de tenencia múltiple se entiende como una sola instancia de la aplicación para todos los clientes, implementada en una plataforma redundante (por ejemplo, en la nube), y de forma tal que un cliente sólo puede acceder a los datos que le corresponden. Típicamente los usuarios dueños de una porción de los datos, pueden administrar parte del acceso a esos datos (por ejemplo, configurar algunos aspectos visuales o reglas de negocio del sistema), pero no el código fuente de la aplicación.

A nivel de Bases de Datos[6] se considera *Tenencia Múltiple (*o *Multiple Tenancy)*, cuando se tiene alguna forma de que un cliente sólo puede acceder a los datos que le corresponde, mediante separación lógica y/o física[7]. Esto se logra mediante uno de dos enfoques:

1. La implementación de una base de datos para cada cliente, y una meta-base de datos para saber la correspondiente a cada cliente.
2. Una sola base de datos para todo de información, con un Esquema (*Schema*[8]) diferente para cada uno de los clientes. Esto en el caso que el motor de base de datos soporte esta característica, como por ejemplo PostgreSQL.

---

[6] https://www.vinta.com.br/blog/2016/understanding-database-multitenancy/
[7] Puede que un punto de entrada (endpoint) para conectar a una base de datos sea un servidor, o un punto-virtual para un clúster; pero efectivamente dos bases de datos para clientes diferentes podrían estar en endpoints distintos.
[8] https://www.postgresql.org/docs/9.1/static/ddl-schemas.html

Utilizar una tabla adicional para identificar al cliente, teniendo todos los datos en una sola base de datos y/o esquema, no corresponde a una práctica razonable si se desea cumplir con la propiedad de aislamiento lógico.

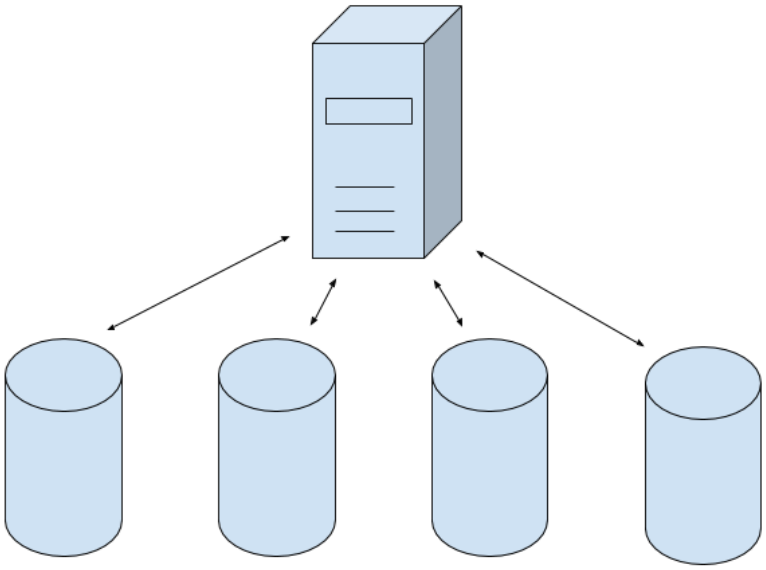

Figura 4: Esquema de B.D. Multi-Tenant en un solo endpoint

2.2.12 Metodologías Ágiles

Las metodologías ágiles describen un conjunto de valores y principios para el desarrollo de software, en el que requerimientos y soluciones evolucionan mediante el esfuerzo colaborativo de equipos multi-disciplinarios y auto-organizados. Se aboga por el uso de planificación adaptativa, desarrollo evolutivo, entrega temprana y mejora continua; se busca además rapidez y flexibilidad para llevar a cabo los cambios en la funcionalidad del producto[10].

2.2.13 Scrum

Scrum es un marco de trabajo [9] que cuenta con una variante orientada al desarrollo ágil de software. Está diseñado para equipos de entre 3 a 9 personas quienes:

- Dividen su trabajo en ciclos llamados *sprints*, que duran desde una a cuatro semanas.
- Verifican diariamente el progreso en reuniones de 15 minutos en las que se debe de estar de pie (stand-up meetings).
- Al final de cada sprint se entrega software funcional.

2.2.14 Metodologías Lean de Desarrollo

Lean es un marco de trabajo de Metodología Ágil, en el cual se llevan al desarrollo de software, los principios definidos en '*Lean Manufacturing*'. Este enfoque de desarrollo ofrece un marco conceptual sólido, valores, principios y buenas prácticas derivadas de la experiencia, para ser un apoyo a empresas con Cultura Ágil.

2.2.15 CRM

Un CRM (*Customer Relationship Manager*) es un sistema informático que cuenta con funcionalidades para gestionar relaciones con clientes, a partir de sus datos. Esto, con el objetivo de que el propio negocio se vea beneficiado.

2.2.16 ERP

Un ERP (Enterprise Resource Planner) es un sistema informático que tiene funcionalidades de manejo de recursos de la empresa, pensado en llevar un detalle exacto y actualizado de Compras y Ventas, entre otros datos. Típicamente los ERP apoyan el proceso de toma de decisiones de los niveles gerenciales.

2.2.17 SLA

Un nivel de acuerdo de servicio (o SLA: Service Level Agreement), en lo referente a Computación Cloud, es el compromiso por parte del proveedor de tener un máximo de tiempo de servicio sin interrupciones en el período de un año. Este se mide como porcentaje de tiempo máximo, bajo 100% en el que el servicio no estará disponible, por ejemplo, por razones de mantenimiento imposible de soslayar o fuerza mayor atribuible a ellos. Cada SLA se mide y certifica de acuerdo con las guías (por ejemplo del *Uptime Institute*[9]*)*, y es lo que el proveedor dice que puede certificar dentro de sus dependencias. En el siguiente punto se detallan los SLA, desde *Tier 1* a *Tier 4*.

2.2.18 Niveles de Disponibilidad en DataCenters

A nivel de Centros de Datos, la entidad *Uptime Institute* define los siguientes 4 niveles de servicio[10]:

- *Tier 1*: Para sistemas sin redundancia; 99.671% uptime (hasta 29 horas de sin servicio por año).
- *Tier 2*: Para redundancia parcial en electricidad, comunicaciones y/o climatización; 99.749% uptime (no más de 22 horas al año fuera de servicio).

[9] Uptime Institute: https://es.uptimeinstitute.com/
[10] https://journal.uptimeinstitute.com/explaining-uptime-institutes-tier-classification-system/

- *Tier 3*: Para redundancia completa en electricidad, climatización y comunicaciones. Permite efectuar mantenciones sin tener fallas de servicio. Autonomía eléctrica de al menos 72 horas. Se garantiza un uptime de 99.982%, esto es no más de 1,6 horas sin servicio al año.
- *Tier 4*: Además de todo lo de *Tier 3*, los sistemas en este nivel cuentan con un segundo proveedor de electricidad, por cableado totalmente diferente e independiente. Esto asegura al menos 96 horas de autonomía eléctrica en caso de falla de ambos proveedores. Además, garantiza uptime de 99.995% al año (menos de media hora fuera de servicio por año).

2.2.19 REST

La Transferencia de Estado Representacional (o REST: Representational State Transfer) es un estilo de arquitectura software para sistemas Web distribuidos. REST permite desacoplar la interdependencia entre sistemas utilizando interfaces de software, las cuales emplean HTTP(S) para obtener datos o generar operaciones sobre estos en cualquier formato especificado.

2.3 Explicación de Tecnologías Utilizadas

En esta sección se indican las tecnologías utilizadas para llevar a cabo la mejora de la plataforma Reservo.cl.

2.3.1 SSL (TLS en la actualidad)

El protocolo SSL (Secure Sockets Layer), en la actualidad se lo conoce como TLS (Transport Layer Security), es un protocolo criptográfico que proporciona comunicación segura a través de alguna red de datos. Este protocolo es utilizado ampliamente en comercio electrónico, comunicación entre servidores de correo electrónico, y en diversos escenarios de manejo de datos sensibles, entre otras aplicaciones en Internet.

2.3.2 Firewall

Un *Firewall* o Muro Cortafuegos es un sistema de seguridad de red que monitorea y controla todo el tráfico entrante y saliente desde una red, hacia otra. Esto lo hace a nivel de protocolo de conexión.

2.3.3 Firewall WAF

Un *Web Application Firewall (WAF)* es un firewall que además cumple con reconocer protocolos de datos usados por las aplicaciones, como lo son HTTP, DNS, FTP, y SMTP, entre otros. Esto permite realizar revisiones y determinar si una solicitud es válida, o si es potencialmente maliciosa, de acuerdo con los criterios especificados por el fabricante (empresa desarrolladora).

2.3.4 Python

Python es un lenguaje de programación interpretado, que enfatiza la legibilidad del código fuente, y tiene una sintaxis que permite expresar en forma sucinta cosas que requieren muchas más líneas de código en otros lenguajes como Java o C++. Se utiliza mucho en el desarrollo de aplicaciones web en la actualidad. Otros usos de este lenguaje apuntan a la creación de scripts para pruebas de seguridad de aplicaciones, o tareas de mantenimiento de sistemas operativos, o análisis intensivo de datos.

2.3.5 Django

Django es un *framework* para desarrollo de aplicaciones web, escrito en Python, y que sigue el Patrón de Arquitectura de software conocido como Modelo-Vista-Plantilla (*Model-View-Template*). Este framework es utilizado por varios sitios conocidos, como *Instagram*, y *Washington Times*, *Disqus*, entre otros.

2.3.6 Nginx

Nginx es un programa servidor web, que implementa el protocolo HTTP y que entrega peticiones mediante un enfoque basado en eventos asíncronos (*asynchronous event-driven approach*). La adhesión a este enfoque permite a los sistemas ser más livianos en uso de recursos, por ejemplo, comparado con un servidor web Apache. A la larga esto redunda en un menor impacto negativo sobre el rendimiento de la aplicación desarrollada, y permite una mayor atención de usuarios por máquina.

2.3.7 uWSGI

uWSGI es un servidor de aplicaciones diseñado con una arquitectura que soporta plugins, lo cual le permite interactuar con varias plataformas y aplicaciones desarrolladas en diferentes lenguajes. De esta manera logra versatilidad, tener un desempeño alto, uso de recursos eficiente y confiabilidad.

2.3.8 MySQL

MySQL es un motor de bases de datos relacionales, que soporta el lenguaje de consulta SQL, con su código fuente liberado bajo licencia GNU GPL. Actualmente es propiedad de *Oracle Corporation* y es ampliamente utilizado como soporte para el manejo de datos en muchas aplicaciones Web.

2.3.9 Celery

Celery es un programa servidor de colas asíncronas, basado en paso distribuido de mensajes. Éste soporta agendamiento de mensajes, aunque su foco es en operaciones en tiempo real[11]. Es factible implementarlo en varias máquinas en red para crear un servicio distribuido de colas.

2.3.10 Redis

Redis es una base de datos en memoria RAM, de código abierto[12], que implementa un almacén de tipo llave-valor y soporta varios tipos de datos abstractos.

2.3.11 OWASP

La *Open Web Application Security Project (OWASP)* es una organización sin fines de lucro constituida en los EEUU. Ésta se originó como una comunidad en línea orientada a producir y disponibilizar en forma libre y abierta artículos, metodologías, documentación, herramientas y tecnologías en el campo de la *Seguridad de Aplicaciones Web.* Este organismo ha definido un conjunto de pruebas para aplicaciones Web, llamado *OWASP Top 10*, con diferentes versiones. Hasta principios de noviembre 2017, la última versión liberada de OWASP Top 10 correspondía a la del año 2013.

2.3.12 Amazon Web Services

Amazon Web Services (o AWS) es una empresa subsidiaria de Amazon.com que provee computación en la nube, por demanda, a personas, empresas y gobiernos. El servicio se brinda en base a una suscripción pagada, con un nivel de servicio gratuito con límite de uso (free-tier) hasta por 12 meses.

Los equipos que provee AWS como máquinas virtuales se ejecutan en los centros de datos que posee Amazon. Esta plataforma permite altos niveles de configurabilidad para obtener el desempeño deseado, de acuerdo a lo que se esté dispuesto a pagar. Cada servicio tiene su esquema diferente de tarificación.

Hasta el 2016, *AWS* tenía disponibles más de 70 servicios para sus usuarios[13]; cada uno de ellos es un sistema distribuido. A continuación, se definen algunos de ellos, específicamente los que son utilizados en este trabajo de título.

---

[11] https://en.wikipedia.org/wiki/Celery_(software)
[12] https://en.wikipedia.org/wiki/Redis
[13] https://en.wikipedia.org/wiki/Amazon_Web_Services ; Mayor descripción y documentación completa en: https://aws.amazon.com/documentation/

1. *Lambda*. Es una plataforma de computación 'sin servidores' basada en respuesta a eventos (*serverless computing event-driven platform*), que ejecuta código en respuesta a eventos, y gestiona de forma automática los recursos computacionales requeridos para ello.

2. *EC2*. *Elastic Cloud Computing* es el principal servicio proporcionado por AWS. Consiste en levantar máquinas virtuales por hora, con poder de cómputo, conectividad y almacenamiento total y absolutamente configurables. Las aplicaciones que corren ahí pueden configurarse para tener acceso a todos los demás servicios, una vez configurados los accesos y permisos, e implementados en la aplicación mediante el SDK (*Software Development Kit*).

3. *S3*. *Simple Storage Service* es un servicio de almacenamiento en línea pensado en tener muy alta disponibilidad, muy bajo riesgo de pérdida de datos (inexistente para efectos prácticos). Se usa principalmente para el contenido estático de las aplicaciones: scripts, imágenes, archivos. También es posible desde las aplicaciones trabajar con esto mediante URLs por protocolo HTTP.

4. *RDS*. *Relational Database Service* es un servicio gestor de motores de bases de datos, diseñado para simplificar la instalación, operación y escalamiento de un motor de bases de datos relacional, para su uso en aplicaciones. Este servicio provee tareas automatizadas para la actualización, respaldos y restauraciones.

5. *DynamoDB*. Éste es un servicio de base de datos NoSQL, el cual está diseñado para permitir la persistencia de datos y su alta disponibilidad, mediante replicación síncrona entre varios *datacenters*. Este servicio está diseñado para ser utilizado en forma simple y directa desde las aplicaciones, mediante operaciones en tablas (conjuntos de datos sería un nombre más adecuado). El nivel de desempeño requerido (operaciones de lectura y escritura por segundo), junto al almacenamiento utilizado, es lo que determina el costo de este servicio.

6. *DynamoDB Streams*. Ésta es una componente adicional de la plataforma de Amazon, la cual se puede activar sobre uno de los conjuntos de datos definidos. Esta componente que permite detectar cuándo son ingresados nuevos elementos, actualizados, o eliminados, de forma tal de activar funciones como respuesta a estos eventos.

7. *SNS*. *Simple Notification Service* es un servicio de notificaciones que permite la entrega masiva de mensajes, mediante el modelo Editor/Suscriptor (*Publisher/Subscriber*) en modalidad '*multicast*'; es decir, a partir de un único envío de la información, y todos los nodos de la red reciben el mismo mensaje. SNS permite enviar notificaciones por e-mail a suscriptores, a dispositivos móviles mediante conexión de datos (mensajes PUSH), y a teléfonos a través de SMS (esto sólo está disponible en E.E.U.U).

8. *SQS*. *Simple Queue Service* es un servicio de colas de mensajes para uso de aplicaciones, que garantiza que los mensajes se entreguen al menos una vez (*at-least-once delivery*). El tipo de cola *FifoQueue* permite realizar las entregas exactamente una vez (*exactly-once-delivery*), pero con un tope máximo de desempeño. SQS está diseñado para ayudar a las aplicaciones a lograr escalabilidad y con alta disponibilidad.

9. *SES*. *Simple Email Services* es un servicio que permite envío masivo de correos electrónicos a bajo costo, y que permite enviar grandes volúmenes en poco tiempo, de acuerdo a disposición de pago.

10. *Route53*. Éste es el servicio que sirve como DNS público hacia internet; o para resolución interna en zonas pensadas para nombres de uso privado en las aplicaciones. Está diseñado para apoyar alta disponibilidad y escalabilidad de aplicaciones.

11. *VPC* (*Virtual Private Cloud)* es un entorno de centro de datos virtual en la nube, el cual permite (mediante la interfaz de Amazon) estructurar una red de la forma que necesitemos configurar.

12. *AWS API Gateway*. Éste es un servicio automáticamente escalable y altamente disponible que permite definir llamadas tipo REST, y utilizarlas como *trigger* para lanzamiento de *lambdas* u otras acciones en los servicios de AWS que uno tenga. Funciona sobre HTTP y HTTPS.

Capítulo 3: Diagnóstico de la Plataforma 'Reservo'

En este capítulo se reporta el diagnóstico realizado acerca del estado de la plataforma 'Reservo'. Para ello se detallan sus principales módulos, una revisión de sus elementos, y una breve descripción de su uso y utilidad. También se presentan testimonios de usuarios de diferentes empresas que tienen contratado el servicio de '*Reservo*', la arquitectura actual y una lista de las funcionalidades que son parte de la aplicación, pero invisibles a los usuarios. Finalmente se indican los puntos en los que debe mejorarse a nivel de aplicación, y se presenta un diagnóstico de los procesos de desarrollo de software utilizados en SC3, y sus prácticas relativas a seguridad informática.

El último apartado es esencial para lograr la expansión del servicio a nivel internacional. Para lograr ese objetivo, se requerirá un equipo de desarrollo dedicado y con conocimientos de programación consciente de la importancia que la Seguridad Informática tiene para el servicio 'Reservo.cl'.

Este grupo de trabajo requerirá tener prácticas orientadas a minimizar errores, vulnerabilidades, y el tiempo de desarrollo y depuración. Todo esto basado en '*Lean Software Development*', con técnicas utilizadas en empresas internacionales que proveen software como servicio (*SaaS: Software As A Service*). Una vez adoptadas estas técnicas, con el uso continuado, pasan a formar parte de la cultura de los desarrolladores, haciendo que nuevo personal las adopte, y con ello se logre mantener la cadencia de generación de valor por parte del equipo.

3.1 Consideraciones de Negocio

Sobre los clientes de SC3, la plataforma se configura y cobra de acuerdo a si trabajan por cantidad de 'Boxes' o de 'Profesionales'. Esto repercute en las vistas de los módulos y en algunas de las funcionalidades.

'*Reservo*' no cobra por volumen de transacciones o agendamientos, salvo el punto de envío de correos electrónicos en el apartado CRM – Fidelización. Allí se utiliza un esquema de compra de bolsas de mensajes, que sirven para que los clientes de SC3 envíen correos promocionales masivos a sus pacientes, descontando de la cantidad total de mensajes disponibles.

En el punto [3.5] se comenta sobre la situación del equipo de trabajo actual, y las necesidades que hay para que la plataforma tenga éxito en su expansión internacional. Antes de entrar en detalles relativos a la seguridad, en dicho apartado se indica la carencia de personal dedicado en equipo de desarrollo.

3.2 Módulos y Funcionalidades

La plataforma '*Reservo*' tiene los siguientes módulos: *Agenda*, *Pacientes*, *CRM*, *Finanzas*, *Estadísticas*, *Configuración* y *Soporte*. Tiene también un acceso directo a los tutoriales interactivos (que están dentro del módulo de Soporte). Estos instructivos están hechos con la plataforma

'*iorad*'[14], la cual permite crearlos de forma tal, que tengan pasos guiados para ilustrar al '*Cliente'* o usuario de la plataforma, y de esa manera solucionar sus dudas. A continuación se describen brevemente cada uno de estos módulos.

3.2.1 Agenda

Este módulo permite a los usuarios del centro de salud, especificados por el *Cliente*, ver un calendario acorde a la forma de trabajo de la institución; por ejemplo, indicando box, nombre del profesional, y a quienes tiene que atender. Este módulo permite diferenciar por color, las citas con diferentes profesionales (Figura 7).

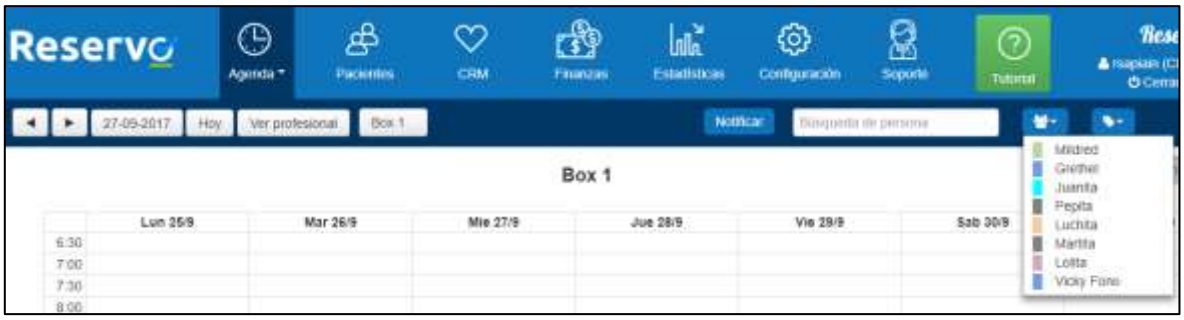

Figura 5: Rótulos colorados para indicar qué profesional usa qué box de atención

El servicio también usa un mapa de colores para representar los estados de las diferentes atenciones, que además incluye íconos para indicar el estado de cada de cita, como se muestra en la Figura 6.

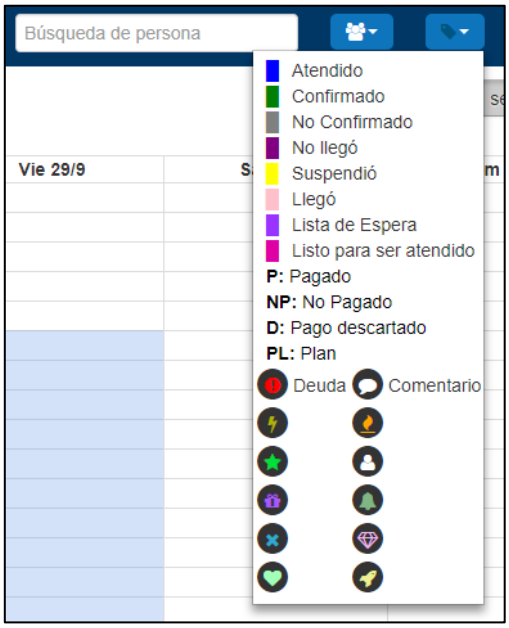

Figura 6: Estado de una cita

[14] https://www.iorad.com/

Cada *Cliente* (o centro de salud) cuenta con una interfaz que debe ser integrada por SC3 en la página de dicho centro, y otra para Facebook, de forma tal que éste, desde su página empresarial en dicha red social pueda tener un widget de agendamiento. De esa forma el *Paciente* puede reservar hora de manera simple usando Facebook. Adicionalmente, '*Reservo*' envía correos de confirmación de citas unos minutos después de agendada una cita, y correos automáticos de confirmación a las 8:00hs del día de la cita.

3.2.2 Ficha electrónica personalizable

'*Reservo*' permite a sus clientes tener una ficha a la medida de sus necesidades. Esta es configurada en la fase de instanciación del servicio, de acuerdo a las necesidades de un nuevo cliente. Dicha adaptación se hace mediante una o más entrevistas con los representantes de la organización cliente.

A continuación, se presentan ejemplos de algunas vistas, con datos de la cuenta de prueba, para mostrar los detalles.

Figura 7: Lista de pacientes

Figura 8: Detalles de una ficha

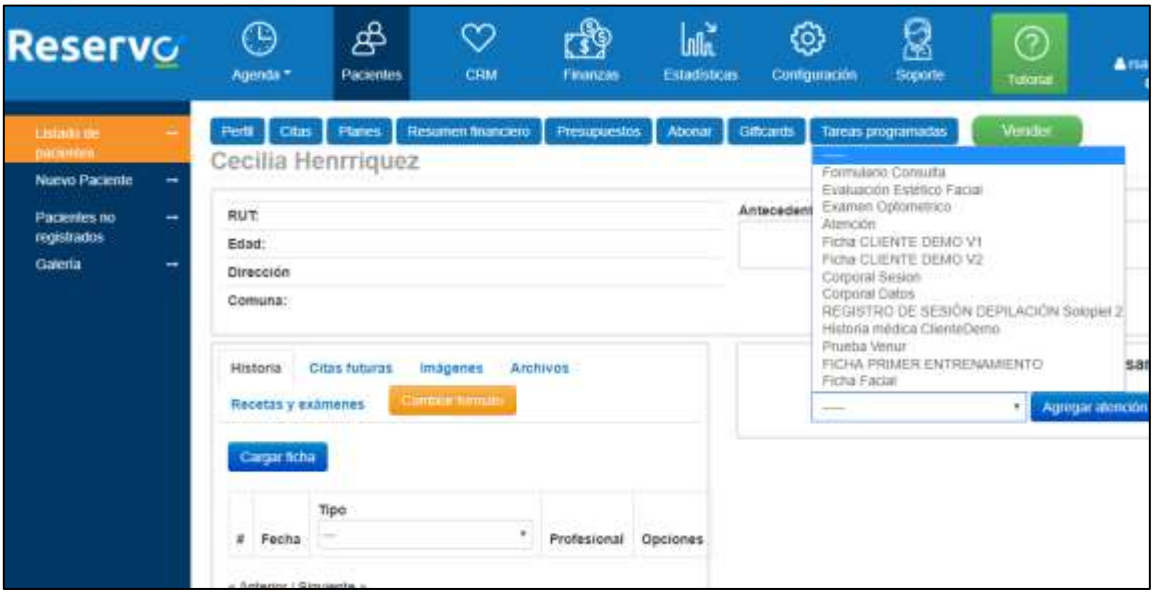

Figura 9: Más detalles de Ficha

3.2.3 CRM

Este módulo apoya la fidelización de clientes mediante campañas de correo electrónico y puntos en las compras. Esto permite categorizar a los clientes, para hacer ofertas dirigidas. El servicio base de '*Reservo*' incluye el envío de un saludo de cumpleaños personalizado, de parte de los *Clientes,* a los *Pacientes*.

A continuación, se muestran algunas capturas de pantalla de este módulo. Como aclaración, en el mensaje plantilla usado en los saludos de cumpleaños, en cada mensaje se instancia el logo que tenga el *Cliente* configurado; en este caso como es una cuenta de prueba, los mensajes están con el de *Reservo*.

Figura 10: Listas de destinatarios

Figura 11: Detalles de a quién se le hizo un envío de una campaña

Figura 12: Parametrización de saludo de cumpleaños de parte de Cliente a Paciente

3.2.4 Registro Financiero

Este módulo permite a los centros llevar el control de los movimientos monetarios: retiros, ingresos por medio de pago, venta de productos o servicios, generar presupuestos, calcular comisiones[15], ver cuentas por cobrar, y otras funcionalidades.

[15] A diferencia de *AgendaPro* (uno de los competidores directos), '*Reservo*' permite calcular comisiones diferenciadas por profesional, por lo que cumple con las necesidades de muchos clientes de SC3. Se da bien seguido el caso de que tipos de profesionales cobran comisiones diferentes por ingresos, y junto a la ficha médica personalizada dan una ventaja comparativa.

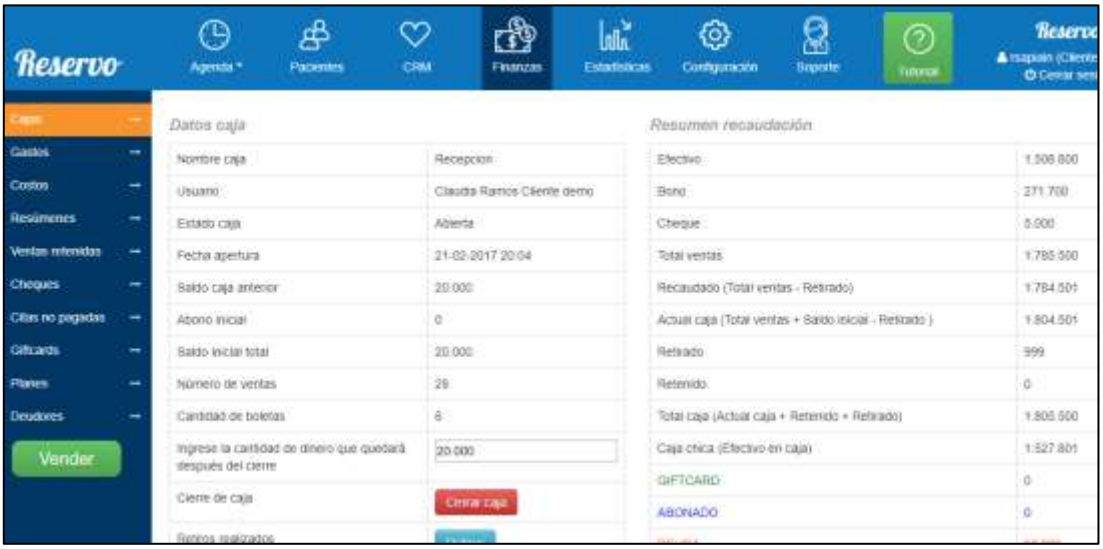

Figura 13: Informe de una caja específica

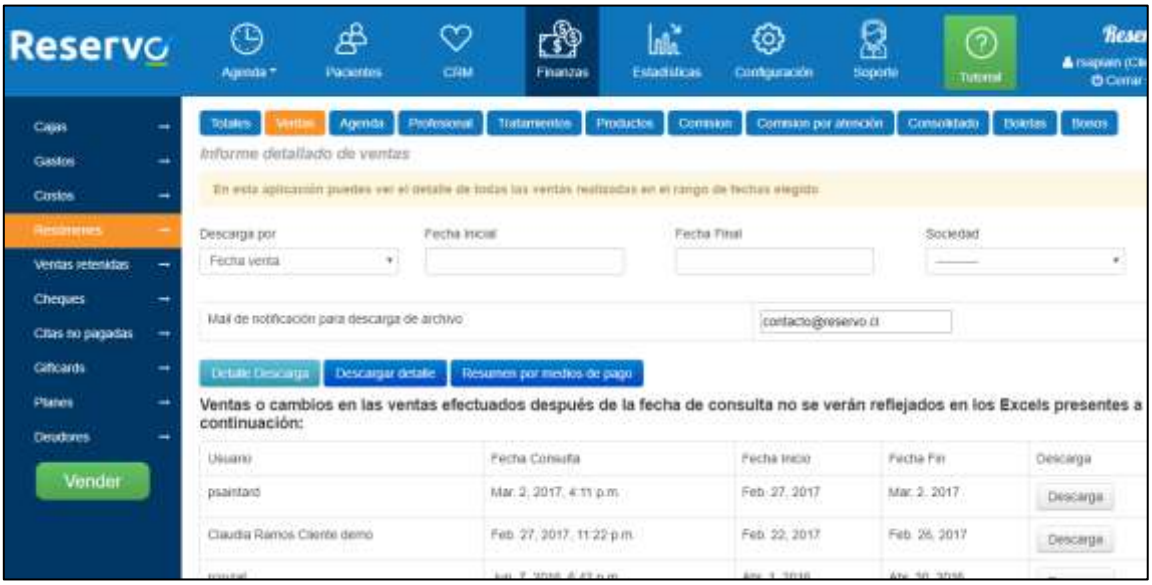

Figura 14: Generación de informe de ventas

3.2.5 Reportes y Estadísticas

Este módulo permite a los usuarios ver datos agregados de su negocio, con el fin de poder visualizar el rendimiento de diferentes aspectos, observar cómo va el negocio, e investigar causas de la situación actual, y así tomar acciones en caso necesario.

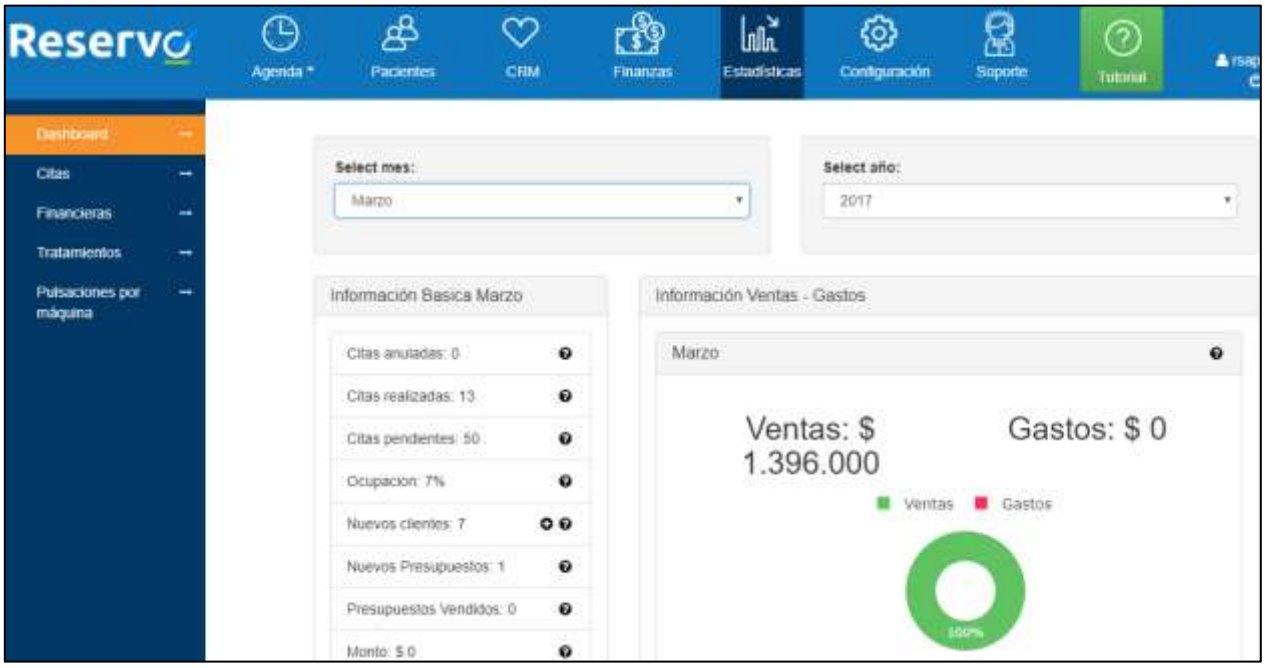

Figura 15: Tablero (*Dashboard*) para visualización rápida de indicadores

Figura 16: Estadística Financiera, agregada en forma mensual

3.2.6 Configuración

Este módulo permite controlar los detalles del servicio de los clientes de SC3; esto es, las Agendas, Profesionales, Salas o Box, Tratamientos, Productos, Planes, Usuarios. También existe un punto de 'detalles avanzados' que permite gestionar muchas más cosas: Giftcards, Insumos, Presupuestos, Costos, entre otros elementos.

Figura 17: Configuración, vista de datos de SC3, para uso de *Cliente*

3.2.7 Soporte

En este apartado se incluyen los siguientes enlaces, y se describen sus funcionalidades:

- *Soporte*: Permite enviar mensajes a '*Reservo*' en forma directa. Se muestra también cómo contactar en forma directa por teléfono móvil.
- *Mensajes*: Corresponde a un historial de las comunicaciones entre SC3 y el *Cliente*.
- *Tutoriales*: Lista detallada de todos los tutoriales guiados.
- '*Actualizaciones Reservo*': Se incluye un detalle de los cambios que han sido añadidos en un *changelog*[16], orientado a los usuarios de la plataforma.

[16] Registro de cambios.

Figura 18: Listado de tutoriales existentes

3.3 Testimonios de *Clientes* actuales

A principios de noviembre de 2017, '*Reservo*' publicó tres testimonios de clientes en su página Web principal (Fig. 20), y tiene además documentados muchos otros que concuerdan con los actuales en cuanto a la utilidad de la aplicación.

Figura 19: Testimonios de Clientes

Muchos de estos testimonios se encuentran publicados en la memoria de Sebastián Cruzat, "Plan de Negocio para un servicio online de reserva y gestión de clientes: caso Reservo.cl" (p.68) [12]. Además, en el sitio web https://www.reservo.cl/testimonios/ se encuentran testimonios adicionales de clientes nuevos.

3.4 Arquitectura Actual

En este punto se detallan los componentes tecnológicos de *'Reservo'*, el servicio DNS público, el modelo de datos, las funcionalidades invisibles a usuarios, y el nivel de disponibilidad de la plataforma.

3.4.1 Componentes

El software Reservo.cl está implementado como aplicación web utilizando los siguientes componentes tecnológicos (detallados en [2.3]) en el proveedor Amazon AWS:

- Lenguaje de Programación: *Python*.
- Framework: *Django*.
- Servidor Web: *NGINX*.
- Servidor de aplicaciones: *uWSGI*.
- Motor de Base de Datos: *MySQL*.
- Sistema de Colas: *Celery*.
- Servicio de Notificaciones: *Scaledrone*.
- Envío de Correo: Amazon *SES*.

En la siguiente figura se muestra la separación lógica de los componentes de la plataforma 'Reservo'. Debido al lugar donde está alojada la aplicación, se utilizan los servicios de Amazon AWS, por un tema de flexibilidad y por consideraciones de costo. Los precios de los servicios de Datacenter o de máquinas virtuales en Chile son simplemente prohibitivos para una pequeña empresa, comparados con los de Amazon que son aproximadamente una tercera parte de los primeros.

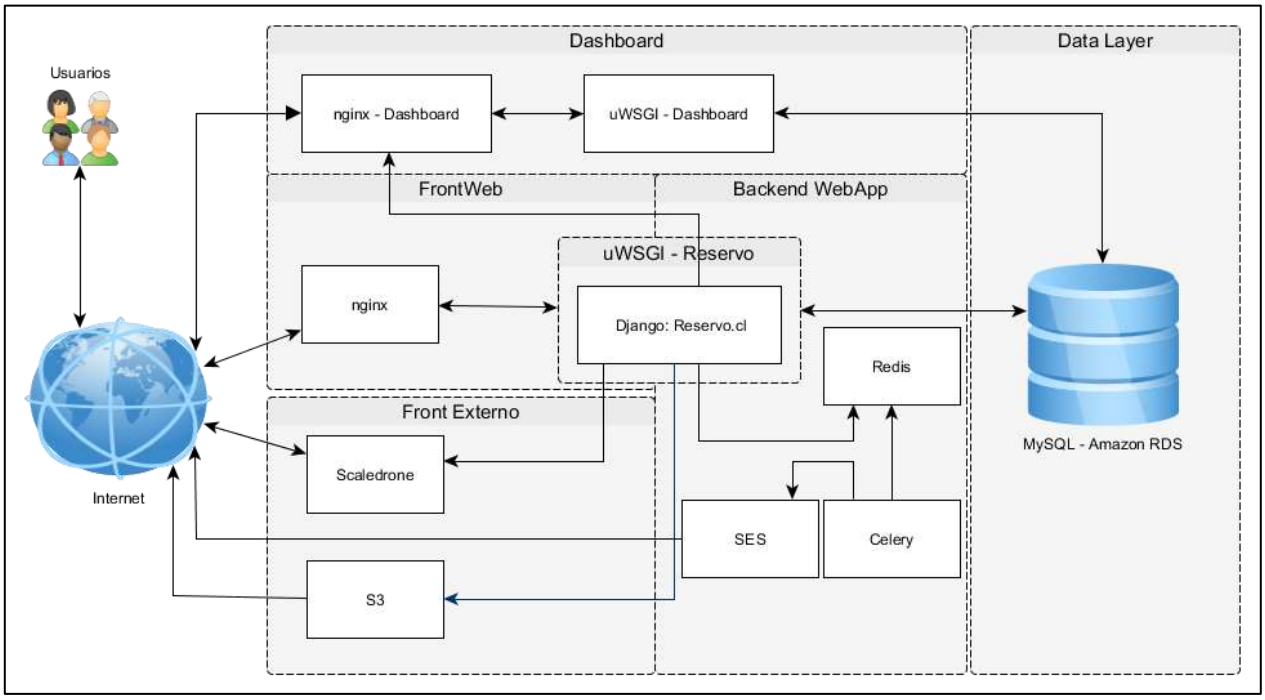

Figura 20: Arquitectura Actual de Reservo

'*Reservo*' actualmente utiliza los servicios en la zona US-EAST-1, la cual corresponde el estado de Virginia en EEUU. Adicionalmente, AWS ofrece servicios específicos que, al ser usados desde las aplicaciones, el desarrollador no debe preocuparse de asignar recursos para escalarlos; esto debido a que son gestionados en forma interna por Amazon AWS. Ejemplos de estos servicios son *SQS*, *SES*, *SNS*, y *DynamoDB*, entre otros.

La siguiente Figura muestra la asignación de componentes a Servidores y Servicios. Particularmente ilustra los puntos que se indican a continuación de la Figura 21.

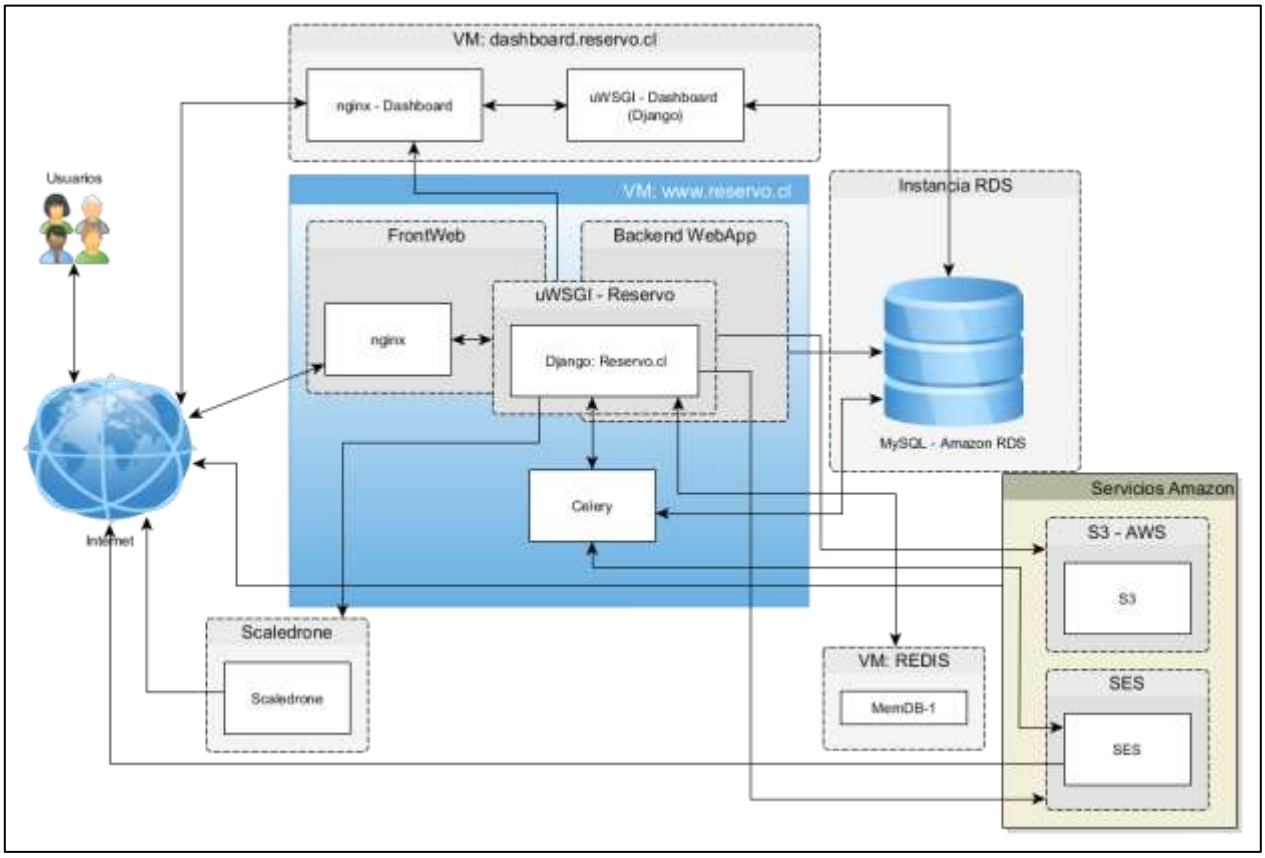

Figura 21: Asignación a Servidores y servicios de los elementos de arquitectura

- En una instancia en EC2 (una *Virtual Machine*, o VM), se ejecuta:
  - La aplicación '*Reservo'* dentro de un servidor *uWSGI*, con un programa NGINX como servidor. Estos dos componentes se comunican mediante un socket-unix.
  - *Celery*, el cual corre en modo daemon utilizando una máquina virtual Java como parte de su funcionamiento.
- En otra instancia en EC2 se ejecuta el dashboard utilizado en el módulo de estadísticas, el cual fue separado como una aplicación Django pequeña y aparte. Esta aplicación sólo permite recibir peticiones autenticadas desde las máquinas de 'Reservo'. Existen también peticiones a medida de los clientes, las cuales utilizan también *dashboard* pero con los datos que corresponden a ellos.
- Los datos se encuentran en una base de datos MySQL, dentro del servicio Amazon RDS. A la BD sólo pueden acceder la aplicación de *Reservo* y el *dashboard* a través de una llamada en la primera de ellas; de lo contrario da un error.
- El servicio Scaledrone es externo, y es invocado desde *Reservo* cuando se necesita realizar una notificación en tiempo real a un usuario.
- En el servicio S3 de Amazon se alojan los archivos de contenido estáticos usados por *Reservo*. También se usa el caso de archivos generados en la funcionalidad de "enviar un e-mail con documento Excel adjunto", si es que dicho archivo es demasiado grande.

Este esquema de funcionamiento de la plataforma tiene los siguientes problemas:

- Todos los componentes críticos para el usuario, están concentrado en una sola máquina virtual. La segunda máquina no puede efectuar nada sin la primera, y esto es así por un tema de diseño, y no será cambiado en el corto plazo, por lo que se debe mejorar la resiliencia de ambos casos.
- No existe políticas o planes para proveer resiliencia a la plataforma ante eventuales caídas de *nginx* o *uWSGI*, ni en *Reservo* ni en el *dashboard*.
- *Celery* y *Redis* consumen recursos que pueden ser usados para atender a más usuarios de *Reservo*.
    - El uso de una JVM[17] hace que se requiera memoria y uso de procesador adicional.
    - Celery hace el manejo y agendamiento, con uso de recursos de la VM.
    - Redis también hace uso de recursos de la VM.
- No se están usando los mecanismos de caché para manejar los contenidos. Resolver esto es algo que se debe efectuar con cuidado, pues hay diferentes partes en las que se debe evaluar el uso de *caching*. Esto será detallado en el siguiente capítulo.
- Debido a limitaciones estructurales del software (por ejemplo, en el acceso a los puertos red que usa la solución, y a los cambios de contexto en sistema operativo), el número de usuarios concurrentes que puede atender una sola máquina llega a su tope, y no mejora sin importar cuanta memoria o procesadores se le pueda agregar a la infraestructura física.
- Algunas funcionalidades que dependen de la hora del país donde está el cliente (por ejemplo, el envío de correos recordatorios a 08:00hs y envío de saludos de cumpleaños), son activados considerando sólo la hora chilena, lo cual a veces genera problemas con los clientes extranjeros.
- Los datos de todos los *clientes*, y sus respectivos *pacientes* se encuentran en una sola base de datos, lo cual limita su acceso y convierte a la BD en un punto de falla.

Todos estos problemas o limitaciones se pueden resumir en los siguientes puntos:

1. *Reservo* no distingue zonas horarias para clientes, ni para tareas programadas se efectúan en los horarios predeterminados de los países de los clientes.
2. *Reservo* tiene una Arquitectura de Sistema casi Monolítica (todo en una pieza única), salvo por el uso de *dashboard*, S3, SES e Scaledrone.
3. *Reservo* en ocasiones puede tener un uso de recursos muy alto, en particular por Celery y Redis.
4. *Reservo* no tiene los datos compartimentalizados por cliente. Existen casos en que esto puede ser crítico.
5. *Reservo* tiene escalabilidad limitada por la arquitectura.

[17] Java Virtual Machine

6. *Reservo* no tiene una tolerancia a fallas alta, a nivel del software en sí. Si bien la infraestructura en que corre la plataforma tiene disponibilidad alta, pero el software no cuenta con ello.

3.4.2 Servicio DNS Público Zona Reservo.cl

El dominio principal *reservo.cl* utiliza el servicio de DNS de Google para no depender completamente de Amazon. Esto debido a que hace un tiempo atrás hubo un incidente en el servicio Route53 (Servicio DNS de Amazon AWS), en el que, si bien el servicio resolvía nombres, tuvo una alta probabilidad de resolver erróneamente o dar un timeout. Esto fue solucionado en su momento, y la causa raíz fue por un error externo a Amazon: un error '*route leak*'[18] causado por Malasia Telecom en junio del 2015.

Debido a eso se tomó la decisión de utilizar otro proveedor de DNS, como precaución adicional, para los subdominios en la zona *reservo.cl*. En ese momento, al cambiar a Google-DNS se solucionaron prácticamente todos los problemas, una vez actualizado el registro y propagado a nivel del país, pues la mayoría de los usuarios de *Reservo* se encuentran geográficamente ubicados en Chile.

Visto el punto anterior, SC3 cuenta con una zona Reservo.cl en Route53, la cual se mantiene actualizada al día con la que se encuentra alojada en Google-DNS. Como servidor de nombres (en la configuración de dominio del NIC Chile) actualmente están los servidores de Google, con posibilidad de indicar los de Amazon.

Este último proceso se efectúa manualmente, sólo si fuese necesario. Para las necesidades de la empresa, esta solución cumple con cambiar rápidamente de proveedor de DNS (Amazon o Google) en caso de que haya problemas con el actualmente utilizado.

3.4.3 Modelo de Datos

De acuerdo con la información provista por SC3, el modelo cuenta con más de 300 tablas, lo cual se explica debido a la alta configurabilidad de las funciones de CRM y ERP para una pequeña empresa. Por razones de confidencialidad (y también porque no es mayormente relevante para este trabajo de memoria), no se entrega información sobre el modelo de datos de 'Reservo'.

3.4.4 Funcionalidades Invisibles a Usuarios

A continuación se indica funcionalidad relevante de la plataforma, la cual es usualmente invisible a los usuarios de la misma.

[18] Route Leak: Error en capa 3 (nivel de transporte), que consiste en la publicación errónea de rutas por parte de los ISP, lo cual hace que paquetes de datos puedan tomar un camino más largo, o dar timeout si se llega a un ciclo, alterando el flujo de datos a nivel de enrutadores troncales de internet.

Envío de recordatorios por email a las 08:00hs. Cuando un paciente tiene una cita agendada, una de las características del sistema, es que poco después de las 8:00 AM, se les envíe a todos los pacientes que tengan citas agendadas para ese día, un recordatorio por e-mail con los datos de estas citas. En este momento no existe una separación de los envíos por zonas horarias, por lo que se utiliza como guía la hora oficial de Chile. Por lo tanto, para un cliente en Argentina, Perú, o incluso España, los correos de sus recordatorios saldrán cuando sean las 8 AM en Chile. En la actualidad Django delega en Celery y Redis esta tarea programada.

Envío de Mails confirmación 5 minutos. Cinco minutos después que se crea una cita, el software debe enviar un correo de confirmación al paciente con los datos y lugar de la cita. A veces este correo es también enviado al centro mismo, según cómo esté configurado el servicio. El envío de estos recordatorios se encuentra implementado en Django, llamando a Celery y Redis cada vez que se necesita.

Campañas de correos masivos. El módulo de CRM cuenta con una funcionalidad de '*mailing'* que permite enviar correos personalizados a los pacientes de un centro, mediante Campañas y Listas de Destinatarios, con un cupo de bolsas de correos:

- En una campaña se define el contenido a enviar, incluyendo posibilidad de parametrización de los pacientes a nivel de Nombre, y Apellidos Paterno y Materno.
- Las listas de correo se pueden actualizar subiendo una planilla Excel, con un formato específico.
- Un envío de correos es generado al dar la instrucción de "envía esta campaña a la siguiente lista que especificaré". Para ello se revisa primero el saldo de mensajes remanentes, y sólo si éste es mayor al número de mensajes a enviar, se procederá a realizar dicha acción.

Generación de e-mails con archivos adjuntos XLS para usuario. Para ciertos reportes específicos, existe una funcionalidad que recolecta los datos para el reporte, los ingresa a un archivo .xls, y luego envía el archivo por e-mail al usuario. En el caso de que el archivo generado supere los 10MB, éste será puesto en el servicio S3, y se le enviará al usuario un enlace por correo electrónico para la descarga.

Notificaciones vía Scaledrone. Actualmente la plataforma cuenta con la funcionalidad de notificaciones de reservas en tiempo real. Los clientes recibirán un mensaje en el mismo navegador cuando un paciente de ellos haya hecho una reserva. Hace un año atrás, esto funcionaba con WebSocket, de manera local en el servidor en el que corre Reservo. En la actualidad, funciona con el servicio externo *Scaledrone* debido a que, si éste falla, *Reservo* sigue operando normalmente. Este servicio fue externalizado debido a que, en la versión legada de 'Reservo', se requería hacer una configuración que era costosa en tiempo y recursos, y además proveía baja flexibilidad en cuanto a escalado. En cambio al externalizarlo, se paga por niveles de uso de servicio y se evita el tener que pensar en su escalabilidad. Por otra parte, debido a que el personal en equipo de desarrollo de 'Reservo' es escaso, esta externalización resulta al final una solución más eficiente y no requiere optimizaciones para efectos de este trabajo.

Generación de archivos descargables en forma directa. Algunas funcionalidades de finanzas y CRM cuentan con este servicio. De momento no habría necesidad alguna de hacer *offloading* de

esto, debido a que los tamaños de archivos generados son muy pequeños, y no son tantos clientes como para que haya un uso frecuente de esta funcionalidad.

3.4.5 Nivel de disponibilidad actual de plataforma

Como se mencionó antes, el nivel de disponibilidad (SLA) de '*Reservo*' debe ser mejorado, particularmente:

- No se cuenta con balanceadores de carga, ni frontal para las peticiones HTTP, ni para las peticiones a la capa de datos.

- No se cuenta con la característica de *Multi-AZ* de Amazon, a nivel de *VPC*.

- No se cuenta con un esquema de replicación *Master-Slave* para base de datos, ni balance de carga en la base de datos.

    - Con esto, sólo se pueden tomar respaldos completos de toda la base de datos, interrumpiendo potencialmente las transacciones mientras dure el proceso. En un escenario donde el servicio debe funcionar siempre, independiente del huso horario, no es aceptable tenerlo implementado de esta forma.

- Sólo se cuenta con el uso de EC2, como la parte de disponibilidad documentada (EC2 SLA Level). Para este servicio, AWS dice que tiene un 99.95% de disponibilidad dentro de una misma región[19]. Esto significa que, en el plazo de un año (8.760 horas aproximadas), a nivel interno de una 'región', lo más que habrá como interrupción de servicio por fallas (antes de empezar a devolver dinero), será poco menos de 5 horas.

Para ilustrar en forma más detallada la disponibilidad de servicios, a continuación se elabora un poco más sobre la configuración de Amazon, teniendo como referencia la distribución de *Regiones* y *Zonas de Disponibilidad*[20] en EEUU.

- Amazon define una Región como un área geográfica, usualmente dispersa en varios estados de EEUU, en la cual tiene varias Zonas de Disponibilidad (*Availability Zone, AZ)*. Una *Región,* por ejemplo, la de E.E.U.U. zona este en Virginia del norte, se define como *US-EAST-1*. Las regiones se encuentran aisladas entre sí, y es posible transferir datos entre ellas.

- Una Zona de Disponibilidad, para efectos prácticos, corresponde a un centro de datos *lógico*. Esto es uno o más *datacenters* físicos. En caso de ser más de uno, estos se encuentran espejados mediante un enlace de alta velocidad. Cada uno de los centros de datos físicos cuenta con comunicaciones, electricidad y climatización redundante, al menos a nivel de '*Tier 3*' según *Uptime Institute*. Una *Zona de Disponibilidad* se representa como *us-east-1a*, una de las zonas en la región *US-EAST-1*.

---

[19] No se especifica si es para cada 'zona de disponibilidad', pero parece razonable asumirlo.
[20] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html

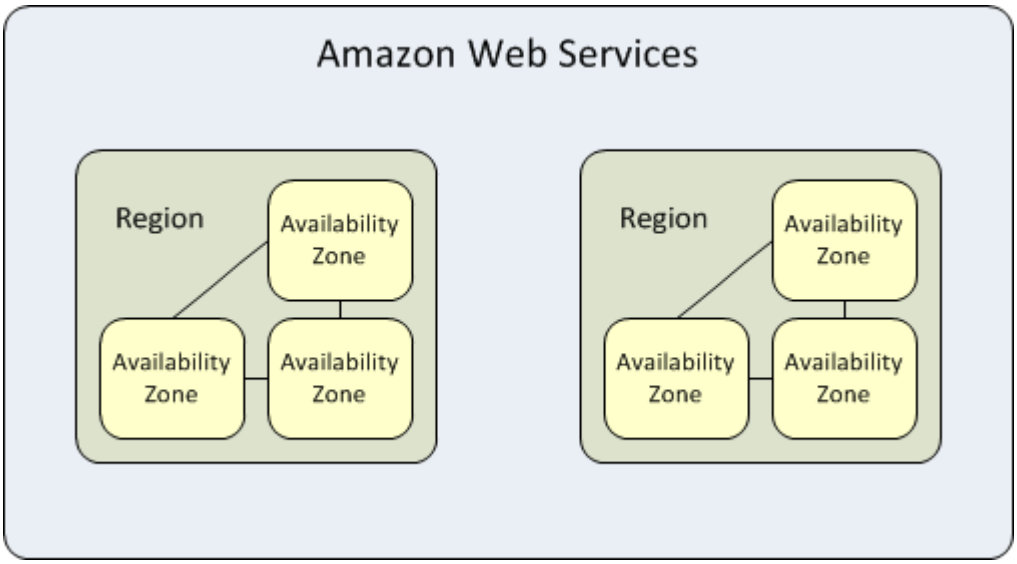

Figura 22: Esquema de Regiones y Zonas de Disponibilidad

El esquema presentado permite tener un nivel de disponibilidad bastante alto, especificado de 99.95% para la plataforma EC2 de computación base. Este indicador toma en cuenta posibles fallas en los enlaces de comunicación, debido a que *Tier 2* ofrece un 99.749%, y *Tier 3* se compromete a 99.982%[21].

A pesar del bajo tiempo de falla, es posible que haya problemas en la plataforma por parte de AWS, o bajas de servicio pre-agendadas por parte de SC3. En la actualidad no existe un grado de mitigación ante fallas accidentales, pero las mantenciones se programan y avisan a clientes con antelación.

En base a todo lo anterior, como mucho es factible comprometer un 99,95% de disponibilidad hacia los clientes, a nivel de Amazon. Sin embargo, debido a que el servicio que está ubicado fuera del país, en caso de alguna falla de conectividad ya sea de proveedor de internet del cliente, o un caso de falla del enlace internacional, la disponibilidad del sistema podría verse afectada. Estas son causas de fuerza mayor, no atribuibles a SC3. Está pendiente el mejorar la plataforma para no tener problemas de disponibilidad, salvo los casos mencionados en el párrafo anterior.

3.5 Diagnóstico de las prácticas internas de SC3

En este apartado se efectúa un diagnóstico de las prácticas internas de SC3, tanto como empresa de desarrollo, así como responsable de la operación de la plataforma. En este último punto, existen deficiencias que vienen de las prácticas de desarrollo de la empresa, y otras que vienen del nivel operacional, que afectan la seguridad informática de la plataforma.

[21] Definición de los Tier: https://www.colocationamerica.com/data-center/tier-standards-overview.htm

Estos últimos dos aspectos del diagnóstico resultan ser cruciales en el avance y desarrollo futuro de la empresa, porque se especifica cómo se irá a trabajar, y de qué forma ésta avanzará en la evolución de su servicio 'Reservo', siempre agregando valor. Adicionalmente, este aprendizaje permitirá que los nuevos proyectos, sigan una metodología de desarrollo que considere el alto desempeño desde la concepción misma de la solución.

3.5.1 Diagnóstico en prácticas de desarrollo de SC3

Como fue mencionado al principio de este capítulo, escalar la aplicación a nivel internacional requiere de un equipo de desarrollo dedicado, que use buenas prácticas de ingeniería de software, que le permita realizar detección temprana de errores, verificación segura de programación en etapas tempranas, automatización de parte del desarrollo y de las pruebas de la aplicación, y uso de versionamiento para manejar el código fuente.

Actualmente se utiliza el principio de Lean Software Development, con un modelo Evolutivo Incremental, pero sin una planificación de ciclos de desarrollo, o un tablero Kanban de visualización. En relación al aspecto de soporte al desarrollo del software, a continuación se indican algunas de las limitaciones más evidentes:

- No se utiliza Test Driven Development (TDD).
- No se utiliza Integración Continua.
- Los entornos usados son solamente Desarrollo y Producción; los pasos a producción se hacen a conciencia, y nunca un viernes.
- Se utiliza versionamiento con 'git' en la plataforma 'github'.
- Se utiliza monitoreo sólo a nivel de servidores y de métricas de desempeño de la aplicación, mediante el servicio Newrelic.

En relación con la organización interna de SC3 en lo que se refiere al desarrollo del software, a continuación se indican las limitaciones más importantes:

- El personal dedicado al desarrollo de '*Reservo'*, es una persona tiempo completo.
- El personal dedicado a la operación de '*Reservo*' es de una persona a medio tiempo, con apoyo del sistema de monitoreo online NewRelic, el cual envía alertas al encargado de operación cuando es necesario.
- Para efectos prácticos, cuando el desarrollador efectúa algún cambio en el software, sólo él conoce qué se hizo, por lo que la evolución del software termina dependiendo de esa persona.
- No existe un ciclo de pruebas automatizado. Sólo se prueba mientras se desarrolla, y las pruebas buscan determinar que la nueva funcionalidad cumpla con lo solicitado.

En relación con el uso de prácticas de programación y documentación del producto, se evidencian las siguientes limitaciones:

- No se utilizan patrones de diseño, de construcción o de validación del código generado. Por lo tanto, no hay forma de asegurar la calidad del producto de software en ningún sentido que vaya más allá de su funcionalidad.

- No existe un documento de diseño consolidado y mantenido, ni un mapeo detallado de requisitos funcionales. Esto puede jugar en contra para el tema de evaluar correctamente qué cosas deben tratarse con cuidado en la implementación de soporte de diferentes zonas horarias.
- La documentación del producto es principalmente el código fuente, el cual es mantenido en base a los comentarios incluidos en él.

En resumen, se puede decir que, aunque la empresa tiene más de 3 años de existencia, su estrategia de desarrollo sigue en el estado de "*Artesanía de Software*". A fines de octubre 2017 la empresa SC3 reconoce la necesidad de pasar a un proceso de desarrollo más ordenado, estructurado y predecible. Para lograr ese objetivo, se requiere trabajar en la definición del nuevo proceso y contar con más personal dedicado al desarrollo del servicio '*Reservo*'.

3.5.2 Diagnóstico del aspecto de seguridad informática de 'Reservo'

En este apartado, se detallan las limitaciones de seguridad encontradas en Diseño, Implementación y Operación de '*Reservo*' en sí. En el texto se marcan las advertencias con [!], y con [!!] los detalles que en seguridad informática se consideran como peligrosos. Estas limitaciones representan un potencial canal para explotar la vulnerabilidad del sistema. Los principales detalles de seguridad encontrados en el diseño de la aplicación son los siguientes:

- Para acceder a la plataforma sólo se requiere un nombre de usuario y clave.
- Para gestionar roles de usuario se cuenta con un 'Administrador', el cual permite crear otros usuarios para un *Cliente*, con permisos de uso limitados para cada funcionalidad en forma granular. El administrador tiene acceso total a todas las funcionalidades de '*Reservo'* para todo lo que es visible para ese *Cliente.*
- Para la gestión de contraseñas se utiliza el default de Django 1.5.x, el cual es el algoritmo PBKDF2, para cifrar las claves[22]. Además se utiliza el mecanismo de recuperación de *password* de Django. [!] No se utiliza un segundo mecanismo de autenticación, como es usual en este tipo de plataformas.

Respecto a los detalles de seguridad encontrados en lo que es el ambiente de desarrollo y en el software mismo, se destacan los siguientes:

- [!!] Se utiliza la rama de versiones 1.5.x del framework Django, la cual ya no cuenta con soporte por parte de dicho proyecto.

- [!!] De momento no hay planes ni posibilidades en corto plazo de migrar la base de código existente a una rama con soporte, como por ejemplo a la 1.11 LTS[23] de Django.

- Para lograr ese proceso de migración, se debe contar con herramientas de apoyo a la integración continua, y que incluyan tests automatizados. Adicionalmente, se requerirá un

[22] https://django-chinese-docs.readthedocs.io/en/latest/topics/auth/passwords.html
[23] Soporte de Django: https://www.djangoproject.com/download/#supported-versions

equipo de personas dedicado a esto, pruebas detalladas, además de una migración de formato de datos (posiblemente manejada por Django mismo, pero eso debe probarse en el proceso).

- La inyección de SQL se evita utilizando las funcionalidades de Django, y nunca se acepta input directo de los usuarios en la creación. Adicionalmente, las consultas que no son provistas directamente por el *framework* se encuentran parametrizadas, y los valores que se obtienen de parte de usuario, son validados antes de efectuar cualquier operación.

- [!!] La validación de anti-CSRF, mediante tokens generados por Django se encuentra desactivada.

- [!] No se utilizan herramientas de detección de posibles vulnerabilidades, como '*Bandit*'[24].

- [!] No se utilizan herramientas de detección de patrones de calidad de código (*SonarLint* y *SonarQube*), ni forma de validar que las bibliotecas en uso estén libres de vulnerabilidades críticas por falta de actualización. Para implementar esto se requiere también integración continua con testing automatizado.

- [!!] Los desarrolladores que de momento están a cargo del mantenimiento y evolución del software, tienen muy poco conocimiento de vulnerabilidades en seguridad informática. Esto no le permite a la empresa tener conciencia de desarrollo orientado a la seguridad de la aplicación.

- No se utiliza la batería directa de pruebas de *OWASP Top 10 versión 2013*[25].

- No se utiliza la práctica de efectuar revisiones de código fuente, para poner al día al equipo de desarrollo, y que además permite el aprendizaje y la detección adicional de errores.

En relación con los detalles de seguridad encontrados en la operación, se destacan los siguientes:

- Los servidores que se han implementado tanto para clientes a medida, como clientes generales, utilizan *Ubuntu 16.04 LTS*, el cual tiene soporte para actualizaciones de seguridad[26] sólo hasta principios del 2021.

- Se desconoce el proceso de aplicación de actualizaciones de seguridad del sistema operativo que corre la plataforma '*Reservo*'. Sin embargo se utiliza cifrado en la transmisión de datos entre un *cliente* y '*Reservo*', mediante uso de *SSL*. La configuración del servidor para los cifrados se dejó para que se cuente con la calificación A+ del test de Qualys[27].

---

[24] Bandit: https://wiki.openstack.org/wiki/Security/Projects/Bandit
[25] A mediados de noviembre salió la versión 2017, que debiese ser la que se debe utilizar cuando se ponga en marcha el plan.
[26] Ubuntu LTS: https://www.ubuntu.com/info/release-end-of-life
[27] Qualys SSL Labs test: https://www.ssllabs.com/ssltest/analyze.html

- Está pendiente el soportar protocolo de cifrado *TLS v1.3*, pero al parecer aún no se implementa en *OpenSSL* (componente utilizado por servidor web NGINX). De aplicarse, esto se vería en la versión 1.1.1 (*Ubuntu 16.04* usa la 1.0.2), la cual vendrá en *Ubuntu 18.04 LTS*. Para implementar este proceso de cambio, puede que sea requerido además de migrar la versión de Django, migrar la versión de Python debido a que podría no venir la rama 2.x en *Ubuntu 18.04*.

- No se utiliza un firewall a nivel de aplicación (*Web Application Firewall*), el cual permitiría pasar sólo cierto tipo de peticiones válidas al servidor.

- No se utiliza una prueba exhaustiva contra *OWASP Top 10*, ni manuales ni automatizadas, ni otro tipo de pruebas de vulneración de aplicación.

Dado que '*Reservo*' funciona en la plataforma *cloud* de Amazon, y una pequeña parte en Google Cloud, existe un acceso central a las cuentas, y accesos diferentes a máquinas en que corren las diferentes instancias, particularmente el servicio general y los clientes a medida.

El acceso a las cuentas de Amazon y Google es manejado exclusivamente por el gerente general de la empresa. El acceso a las máquinas virtuales se realiza mediante certificados digitales emitidos por la plataforma de Amazon; no se utiliza *passphrase*. A los certificados de estos servidores sólo tiene acceso el personal de SC3.

Capítulo 4: Necesidades de mejora

En el capítulo 1 de este documento se detallan las necesidades de mejora de la plataforma. En este apartado se aborda qué es particularmente lo que se debe mejorar, haciendo referencia a lo enunciado en las secciones [1.1.1] a [1.1.6].

4.1 Respecto a las zonas horarias

Se requiere que la plataforma sea funcional en el eje horario del *Cliente* de SC3, independiente de la hora local del servidor. En este sentido existen dos casos a considerar en la aplicación.

4.1.1 Caso 1

Supongamos que un *cliente* en Perú, uno en España y otro en Chile deben ver, en forma independiente de la hora local del servidor, su información de día, en su hora local. Por ejemplo, un tablero que cuente los ingresos del día, debe partir con ventas desde las 00:00:01 en la zona horaria del *cliente*. Otro ejemplo sería el envío de correo de recordatorio el día de la cita; estos correos deben comenzar a salir a las 08:00 en la hora local del *cliente*.

En el caso del cliente en Perú, la segunda tarea se efectuaría a las 13:00 UTC; para el cliente en España, a las 07:00 UTC (08:00 UTC en horario de invierno); mientras que, para el cliente en Chile, se efectuaría a las 11:00 UTC. Esta situación ilustra el caso 1, el cual representa una fuente importante de problemas para la actual provisión de servicios automatizados.

4.1.2 Caso 2

En el caso 2 la aplicación debe soportar la ejecución de tareas en *background*, de forma invisibles al usuario, a la hora que éste agende considerando su hora local. Por ejemplo, se está estudiando implementar la apertura y cierre de caja contable diaria en forma automatizada de acuerdo con un rango horario. Esta apertura y cierre sería a las 00:00 hs de cada día (o configurable por usuario), pero en la hora local del *cliente*.

Con esto, un cliente en Chile cerraría y abriría caja con los desfases horarios respectivos (que debe también contemplar las diferentes zonas horarias de Chile), en forma similar a lo especificado en el caso 1.

4.1.3 Requerimientos y supuestos

Los requerimientos de usuario para abordar este problema se pueden resumir fundamentalmente en dos:

- **RS1**: Sistema debe considerar la zona horaria del *cliente* para toda la interfaz en lo referente a gestión, y en lo referente a tareas automatizadas.
- **RS2**: Sistema debe considerar el caso de que un país cuente con múltiples zonas horarias, al momento de mostrar la información al *cliente.*

Existe el supuesto de que las personas que asisten a los centros vinculados a 'Reservo' agendarán en una misma localidad geográfica (en un radio de unos 50 km) sin cambios horarios. Esto es porque el servicio está pensado para acercar pacientes a centros de salud o estética ubicados en su misma ciudad, o en área aledañas al lugar donde se encuentran los pacientes.

4.2 Respecto a la arquitectura

Se requiere una arquitectura que permita desacoplar los componentes y escalar en forma simple la plataforma. Como se mencionó en la sección 1.1.2, existen límites respecto a lo que puede procesar una instancia de servidor, específicamente de sistema operativo, independientemente de que este problema de tasa de atención abordable a través de la paralelización del servicio.

Las empresas que proveen servicios escalables y de alta disponibilidad, que ya son de uso cotidiano (por ejemplo, *Slack, Salesforce, Netflix, Pipedrive, y Spotify),* partieron con un producto monolítico, pero tuvieron que evolucionar a uno que utiliza microservicios en alguna medida, lo cual les permite escalar en forma simple. Además, permite sacar ciertos servicios a plataformas distribuidas que da un proveedor; en este caso, remover del monolito el envío de correos. Este enfoque de solución permite que la carga de procesamiento de ciertas tareas del sistema no esté en los *nodos* pagados en forma directa para servir la aplicación, sino que en máquinas del proveedor cuyo uso se paga en la medida que se utiliza.

La mejora de la arquitectura permite abordar en forma total las limitaciones enunciadas en las secciones [1.1.2] y [1.1.3], y en forma parcial lo especificado en las secciones [1.1.5] y [1.1.6]. Actualmente '*Reservo'* no cuenta con tareas que estén varios minutos procesando para poder mostrar algo al usuario. Incluso los *dashboards*, que son generados en el momento, se muestran en menos de 10 segundos al usuario. Al igual que el tiempo de respuesta de la aplicación, también es importante mantener los detalles acerca de qué está haciendo cada usuario y correlacionar dichas acciones. Actualmente, esto se hace siguiendo los logs de *Nginx* y *uWSGI* utilizando una simple búsqueda en texto. Sin embargo, a medida que crezca la aplicación, no está claro cómo se mantendría este servicio. Por ejemplo, qué pasaría si se hace con microservicios que corren en diferentes máquinas virtuales, en horarios o días diferentes. Tampoco está claro cómo se consolidaría esa información para cumplir con la trazabilidad de acciones de usuario.

Dado lo anterior, se plantean los siguientes requerimientos de mejora de la arquitectura del servicio '*Reservo*':

- **RS3**: Se requiere una arquitectura que permita escalabilidad a nivel de clientes a atender, y que el desempeño sea igual o mejor a lo actualmente existente.
- **RS4:** Se requiere mantener la trazabilidad de las acciones de los usuarios, con el sistema de registro que muestra *uWSGI.*

4.3 Respecto a la capa de datos

Actualmente, los datos de todos los clientes que utilizan el servicio general se encuentran en una única base de datos. Los clientes de servicios '*a la medida*', que son pocos, tienen cada uno su propia base de datos.

La expansión internacional, requiere tener adecuadamente separados los datos de cada cliente, no sólo a nivel operativo para poder segmentar la información de los diferentes clientes. '*Reservo*' provee la capacidad de crear fichas médicas personalizadas específicamente para necesidades del *Cliente*, aunque no todos hacen uso de esta funcionalidad. Si bien en Chile no hay una regulación específica vigente como *HIPAA*[28] en E.E.U.U., o su equivalente en Europa, sí es de un nivel de principio Constitucional y se necesita considerar eso en el desarrollo.

Cumplir con este tipo de regulaciones da una ventaja competitiva, y también garantiza un correcto cumplimiento con normas internacionales de confidencialidad, disponibilidad e integridad de los datos médicos, hacia *Clientes* y sus *Pacientes*. Esa segregación también permite efectuar otros procesos, como respaldar los datos en horarios que se sabe que el cliente no estará utilizando directamente la plataforma.

Adicionalmente un esquema que mejore el modelado de datos, en lo referente a seguridad, debe ser concebido como un servicio. Si bien puede ser una capa más de procesamiento/enmascarado, a nivel de temas de información personal sensible, debe considerarse esto en el futuro del desarrollo de la plataforma. Con esto se aborda completamente, desde la perspectiva de modelado de los datos, la limitación indicada en la sección [1.1.4]. El desempeño de la base de datos, y cómo mejorarlo ante un aumento permanente de usuarios, se verá en el siguiente punto.

Dado lo expuesto anteriormente, el requerimiento que surge en relación con la capa de datos es el siguiente:

- **RS5**: Se requiere asegurar que los datos asociados a un *Cliente* sean solamente visibles por usuarios que él defina en su perfil de administrador.

4.4 Respecto a la escalabilidad

Una aplicación para ser considerada escalable debe estar diseñada para funcionar ante estos dos casos: (1) incremento sostenido en el tiempo de carga de usuarios, y (2) incremento repentino de las visitas en un intervalo de tiempo corto. Esto requiere que la arquitectura esté rediseñada, orientada a tener componentes desacoplados y uso de servicios automáticamente escalables.

Esto implica varios pasos de mejoras en diferentes partes y componentes de la arquitectura. Usualmente estos consisten en poner un programa que hace de caché, hacer que el contenido estático sea servido por un proveedor que tenga capacidad elástica ante alta demanda, y/o instalar

[28] HIPAA: Health Insurance Portability and Accountability Act., https://en.wikipedia.org/wiki/Health_Insurance_Portability_and_Accountability_Act

algún tipo de balanceador de carga en diferentes etapas de la arquitectura, a nivel frontal y de acceso a datos.

El primer caso considera una línea de base de funcionamiento, pensando en que la plataforma está estructurada de esta forma, y a medida que aumentan los usuarios por niveles de tramos, simplemente aumenta la cantidad de piezas presentes. Inicialmente esto puede ser hecho de forma manual, y después puede evaluarse el automatizar este incremento.

El segundo caso considera elementos que, en caso de mayor cantidad de peticiones, automáticamente pueden servir ese nivel. Principalmente son balanceadores de carga y redes de distribución de contenidos (*CDN*: *Content Distribution Network*, como *Cloudflare, Amazon Cloudfront, o Akamai*). Para algunos casos, es posible poner componentes de caché de datos de la siguiente manera:

- Caché de páginas generadas.
- Caché de datos en las consultas, en caso de que las páginas deban generarse siempre, pero con datos idénticos.
- Servidor DNS local sólo para la máquina, replicado de un origen válido, y que las componentes utilicen este enfoque.

De estas tres alternativas, la última corresponde a uno que se suele efectuar al final, y sólo si resulta ser un caso completamente necesario, debido a que la configuración es algo más compleja. El balance de carga va en las siguientes partes:

- *Landing page*: Se hace necesario el separar la página de información corporativa del servicio '*Reservo*', del acceso a la aplicación en sí misma. Actualmente es todo parte de la misma aplicación en Django.

- *Contenido estático servido externamente*: En la actualidad '*Reservo*' tiene almacenados en Amazon S3 los archivos estáticos (hojas de estilo, imágenes, javascript y otros scripts) que le proveen escalabilidad automática para servirlos.

- *Balanceo de peticiones HTTP*: Esto se maneja a nivel frontal de internet. Amazon provee el servicio ELB (*Elastic Load Balancer*), que permite manejar la recepción en forma elástica automática.

- *Balanceo de procesamiento back-end*: Con la arquitectura redefinida, una vez balanceada la petición HTTP, el procesamiento de operaciones a nivel de modelo se efectúa en un solo "*nodo*"[29]. Éste es una instancia de uWSGI sirviendo Django, el cual sólo efectúa el trabajo necesario para generar la vista que será mostrada.

- *Balanceo de acceso a base de datos*: Esto requiere hacer algo a medida. Utilizando el programa *SQLProxy* ya sea en la máquina local, o en otra dedicada sólo a ello. El balance se hace cuando se tiene al menos el esquema más básico: una BD con su réplica de lectura (esquema *Master-Slave*). Esto porque eventualmente la base de datos, aunque se tenga más

[29] Nodo: máquina virtual con nginx+uWSGI-Django.

capacidad provisionada, si son demasiados usuarios, podría verse sobrecargada en cuanto al número de conexiones abiertas.

- *Migración de funcionalidades a microservicios*: Esto es un proceso que debe ser hecho en forma paulatina; lo primero que se está abordando es el sacar la generación y el envío de e-mails. A futuro se debe orientar a que la mayor parte de actividades que puedan demorar tiempo de generación, sean convertidas a microservicios. Lograr esto hará que un *nodo* pueda atender más usuarios.

4.5 Respecto a la alta disponibilidad

Para considerar una plataforma como "altamente disponible", es necesario que las componentes de hardware, comunicación[30] y software tengan niveles de redundancia. En el caso de proveedores de servicios Cloud, ellos se encargan de proveer un suministro eléctrico ininterrumpido, como fue mencionado en el punto [3.4.5] en lo referente a los centros de datos certificados *Tier 3*. Para que Reservo cuente con alta disponibilidad, se deben abordar los siguientes puntos:

- *Configuración de DNS público*: Usar dos proveedores diferentes (Google y Amazon, por ejemplo), con una configuración replicada. En principio esto se puede manejar manualmente, pero si crece demasiado, el DNS debe ser centralizado de forma tal que en un solo lado sea cambiado, y en el otro se replique automáticamente.

- *Balanceador de peticiones HTTP*: Usar el servicio Amazon ELB, además de permitir escalar en forma automática la cantidad.

- *Firewall de aplicación*: El servicio Amazon WAF permite aceptar solo peticiones dentro de un patrón específico que cumpla con lo requerido para la aplicación.

- *Máquinas adicionales*: Se debe considerar al menos dos máquinas virtuales para NGINX+uWSGI.

- *Replicar datos en un esquema master/multi-slave*: Esto permitirá tener un mayor desempeño ante un aumento de peticiones.

- *Balanceador de consultas SQL*: Configurar para que peticiones de INSERT/UPDATE/DELETE sean enviadas al *Master*, y los SELECT distribuidos en forma pareja a los *slave*.

- *Centralizador de logs*: Se requiere centralizar y correlacionar los logs de ELB, NGINX, uWSGI, para efectuar un adecuado seguimiento de las acciones de los usuarios en el software, y para efectos de detección de posibles errores en la operación.

---

[30] Para efectos de una aplicación Cloud, es mejor considerar la Comunicación entre componentes de software como un elemento fundamental, aunque sean transparentes al HW y SW. Son elementos configurables.

En casos de contar con una mayor cantidad de usuarios, donde se requiere que la cantidad de máquinas para procesar aumente mucho, se suele utilizar un esquema con servicios de DNS de uso de red interna; por ejemplo, provisto por *Amazon Route 53* como una Zona Privada (mismo proveedor de servicios cloud donde corre la aplicación, para ser preciso).

La inclusión del servicio DNS a nivel interno es para referenciar a ciertos hosts/servicios por nombre, pero que la IP a la que resuelven los punteros, pueda cambiar pasado un periodo de tiempo, o una cierta cantidad de peticiones (esquema Round-Robin). En este momento no sería necesario para '*Reservo*', implementar completamente un upgrade, pero hay casos como en Spotify donde usar un resolver/caché local[31], da una hora de independencia en caso de alguna falla. En esos casos se podría considerar utilizarlo, además de que esto brinda un aumento de desempeño del servicio.

4.6 Respecto al proceso de desarrollo

'*Reservo*' está en el punto en el que necesita inversión de recursos, particularmente tiempo: para repensar la evolución de la plataforma, de personal dedicado directamente a su desarrollo, y a su operación. Se necesita adoptar herramientas que permitan mejorar la generación de valor, específicamente que apoyen:

- *Test Driven Development*: El uso de tests permite validar las funcionalidades cuando se efectúan cambios dentro de diferentes módulos.

- *Desarrollo Asistido*: Uso de complementos que permitan detectar duplicación de código y patrones poco eficientes.

- *Integración Continua*: Cada vez que se efectúa un cambio en el repositorio de código fuente, un programa descarga la base de código, compila y se corren las pruebas en forma automatizada. Si todo resulta se avisa a los encargados, para pasar a las revisiones que no son automatizables. Esto permite detectar errores en una etapa temprana. Por otra parte, se deben incluir también herramientas que verifiquen dependencias vulnerables y patrones de programación segura en el servidor de integración continua.

Se necesita además contar con una mayor cantidad de entornos: Desarrollo (*Dev*), Pruebas (*Test*), Producción (*Production*). En algunos casos, podría ser necesario tener un entorno de Ensayo (*Staging*), que sea lo más parecido al de Producción. Allí se pueden simular casos y verificar el comportamiento del software antes de efectuar cambios que sean complicados como, por ejemplo, alterar el modelo de datos.

Respecto al equipo de desarrollo, para el mantenimiento y la evolución de la aplicación '*Reservo*' misma, se debiese considerar un equipo de tres personas que trabajen a tiempo completo. Se requiere también agregar revisiones de código fuente al final de cada '*sprint*', para fomentar un correcto aprendizaje, y con ello minimizar riesgos por bajas temporales o rotación de personal.

---

[31] Caso Spotify y DNS: https://labs.spotify.com/2017/03/31/spotifys-lovehate-relationship-with-dns/

4.7 Respecto a la seguridad informática

La única forma de dar una garantía de que un sistema que se está desarrollando es 'seguro', es que el ciclo de desarrollo tenga una orientación a incluir la seguridad desde el diseño mismo del producto, y en los pasos siguientes correspondientes a su implementación. Para ello, el equipo que construya '*Reservo*', debe tener entrenamiento en desarrollo seguro de aplicaciones, conocimiento de las diferentes fallas de seguridad a las que se está expuesto, como por ejemplo, las detalladas en OWASP Top 10 2013, y cómo prevenirlas de manera natural.

Las pruebas manuales de seguridad informática (Pentesting) deben ser realizadas por alguien externo al equipo de desarrollo normal, que tenga especialización en pruebas de penetración Web, antes de pasar a producción. Además, se debe contar con tiempo suficiente para validar y corregir según los resultados de las pruebas. Por otra parte, se deben incluir pruebas automatizadas, y una batería de pruebas manuales en el ámbito de la seguridad de la aplicación, para dar una adecuada garantía a la hora de pasar a producción un software.

Es recomendable seguir las líneas de desarrollo seguro de aplicaciones de OWASP.
Dado que Django 1.5.x se encuentra sin soporte, se necesita urgentemente migrar '*Reservo*' a la rama que tenga soporte LTS, como la 1.11.x, para asegurar mantenibilidad de la base de código sobre la que funciona, por al menos 3 años. Dentro de este plan de trabajo, se debe incluir también el cambio de versión de sistema operativo, para cuando sea necesario. Además, se requiere habilitar y trabajar con las protecciones contra CSRF que provee Django.

Dentro de la integración continua de aplicaciones, se debe incluir chequeos con herramientas como Bandit, u otras para verificación de calidad de código, las dependencias vulnerables, y para asegurar una mejor revisión en lo que respecta a seguridad.

Por otra parte, se recalca la importancia del firewall de aplicación que fue mencionado en el punto [4.5], como una forma de asegurar una capa adicional de seguridad. Los certificados que se utilizan para el acceso a máquinas virtuales deberán utilizar passphrase, y se debe buscar alguna forma de que se acepten conexiones a administrar los servidores sólo desde la oficina de SC3.

Se sugiere también habilitar el segundo factor para la cuenta de Amazon que tiene la aplicación en sí, y para la cuenta de Google Cloud, para tener un nivel de seguridad más razonable. Desde dichos paneles de control es posible borrar en forma permanente las máquinas virtuales y sus discos duros con los datos almacenados, y también se pueden eliminar las instancias de bases de datos.

Capítulo 5: Diseño de mejoras y soluciones a problemas

El software ‘Reservo’ cuenta con funcionalidades de mucha utilidad para sus clientes, con algunos módulos con mayor criticidad que otros. El trabajo detallado en los puntos anteriores de este documento no consistió en cambios mayores en la implementación de módulos ya creados. Sin embargo, se agregaron componentes externas al software desarrollado en Python, los cuales son invocables desde el sistema; esto se hizo para eliminar sobrecarga de uso de memoria y colapso de componentes que se ve actualmente. El resto fue un trabajo de análisis y diseño de mejoras a nivel técnico y organizacional.

Para dejar en buen pie a la plataforma ‘*Reservo*’, para llegar a un público internacional, se requiere abordar los temas técnicos y aplicar mejoras organizacionales, particularmente lo que tiene que ver con los desarrolladores, la consideración de la Seguridad Informática como parte integral del proceso de desarrollo.

Todos los cambios a nivel de ‘*Reservo*’, que sean de tipo estructural de la aplicación, requerirán trabajar también en la plataforma de back-end interno (*reservointerno*) para que ésta permita configurar los nuevos cambios a efectuar. Esto permitirá que el proceso de creación y gestión de nuevos *Clientes* sea consistente.

Las mejoras que se plantean a lo largo de este capítulo ya han sido aplicadas en la industria del software, en especial por empresas que ofrecen servicios de alta demanda a nivel internacional, como por ejemplo: *Netflix, Salesforce, Spotify*, etcétera. Estas empresas son pioneras en este ámbito y varias de ellas comparten su conocimiento acerca de cómo lo han hecho. Esta información la comparten mediante un blog técnico, y en algunos casos presentan muestras de implementaciones y configuraciones útiles. Algunas de las mejoras corresponden a cambios en la infraestructura de soporte del servicio, y otras implican modificaciones al software mismo; algunas mejoras impactan en ambas partes, y eso se detalla en la Tabla 4.

Los apartados [5.7] y [5.8] afectan no sólo a la infraestructura de soporte y a la aplicación, además tienen implicancia en la empresa SC3 misma, lo cual trae cambios serios, pero positivos, de acuerdo a la experiencia en la industria, acerca de cómo será mantenida y evolucionada la plataforma ‘*Reservo*’.

Tabla 4: Correspondencia de que se ve afectado en cada campo de plataforma

Mejora	Infraestructura	Aplicación
Zonas Horarias	X	X
Arquitectura	X	X
Capa de Datos	X	
Escalabilidad	X	
Alta Disponibilidad	X	

5.1 Mejoras en manejo de zonas horarias

Para abordar ambos casos, los servidores y la aplicación en Django deben de ser configurados en la zona horaria UTC, la hora del sistema sincronizada contra una fuente de tiempo oficial a través del protocolo NTP. Los proveedores de servicios *cloud* tienen su propia fuente de tiempo sincronizada contra los relojes atómicos del gobierno de E.E.U.U.[32]. Esto requerirá cambios, de configuración en la infraestructura actual (y establecer lo requerido para los *nodos* futuros), como en el software mismo, para que éste sea capaz de operar en la zona horaria del *Cliente*.

Se requiere también realizar respaldos automatizados de la base de datos en Servicio RDS, Amazon las muestra en UTC desde la interfaz. En caso de implementar una base de datos por cliente, se deberá hacer una tarea diferente de respaldo de cada BD, a través de algún gestor externo. Esto puede ser una instancia micro.t1, la más pequeña, que sólo haga dicha tarea, configurada por 'cron' para cada rango horario; por ejemplo, un script que cada 1 hora revise un archivo diferente, y que indique los clientes a respaldar.

Se debe investigar si existe en Django un soporte de zonas horarias diferentes para cada usuario, o si hay que hacerle mejoras a uno existente, de forma tal que para un *Cliente* todos sus usuarios vean la plataforma con esa zona horaria. Con esto es posible desarrollar una mejora en '*Reservo*', para asegurar el correcto funcionamiento de la interfaz y las funciones automáticas dependientes de la zona horaria.

En el caso de redefiniciones de zonas horarias, actualizarlas a nivel de sistema operativo es usualmente simple (existen procedimientos documentados, o es una simple actualización de sistema). Sin embargo, cuando se trata de lenguajes interpretados por alguna máquina virtual JIT (caso de Python y Java), se debe actualizar las definiciones de la forma que se indique en dicho lenguaje. Todo lo anterior nos permitirá abordar los casos 1 y 2 mencionados en la sección 4.1.

Para abordar el caso de las tareas programadas para realizarse en cierto horario, como por ejemplo el envío de emails de saludo de cumpleaños (en nombre del *Cliente* al *Paciente*), o el envío de recordatorios por email a las 08:00 hora del *cliente* cuando éste tiene una cita pendiente, lo más simple de implementar en esta etapa es utilizar una máquina virtual con scripts lanzados por un '*cron'* a las horas que corresponda.

Al hacer que la plataforma reconozca la zona horaria del usuario de '*Reservo*', y que en las tareas invisibles esto funcione en forma consistente, todos los clientes internacionales tendrán el servicio de la forma esperada por ellos.

5.2 Mejoras a la arquitectura

Las mejoras que se plantean en este apartado permiten abordar las necesidades expuestas en la sección [4.2], e implican una redistribución lógica y física de componentes de la plataforma.

[32] En Chile se sincroniza contra el punto horario provisto por la Armada (SHOA, vía ntp.shoa.cl), el cual se considera la hora oficial del país.

En el nuevo escenario se pasa a depender del proveedor que se utilice (en este caso, Amazon AWS), pero los servicios de asistencia están garantizados con alta disponibilidad y escalabilidad por parte del proveedor. Estos servicios son configurables de acuerdo con la disposición de pago, y escalables en forma automática. En cambio, si uno deseara implementar la nueva arquitectura con las piezas actuales (*Celery*, *Redis*), debe preocuparse también de configurarlos para lograr alta disponibilidad, lo cual hace más complejo el trabajo de diseño, escalamiento y la operación de la plataforma.

El objetivo de la nueva arquitectura es permitir operar a nivel internacional, atendiendo a una creciente población de usuarios, y cumpliendo con la escalabilidad, disponibilidad, y seguridad requerida para este escenario.

5.2.1 Mejoras en la separación lógica

Como componente de procesamiento desatendido, salvo por datos iniciales y resultados finales, se utilizará el servicio *Lambda*. En este caso se elimina *Redis* y *Celery*, cambiándolos por *DynamoDB, SQS, SNS, Lambda*, en una nueva organización lógica de la solución. La siguiente figura muestra la separación lógica propuesta.

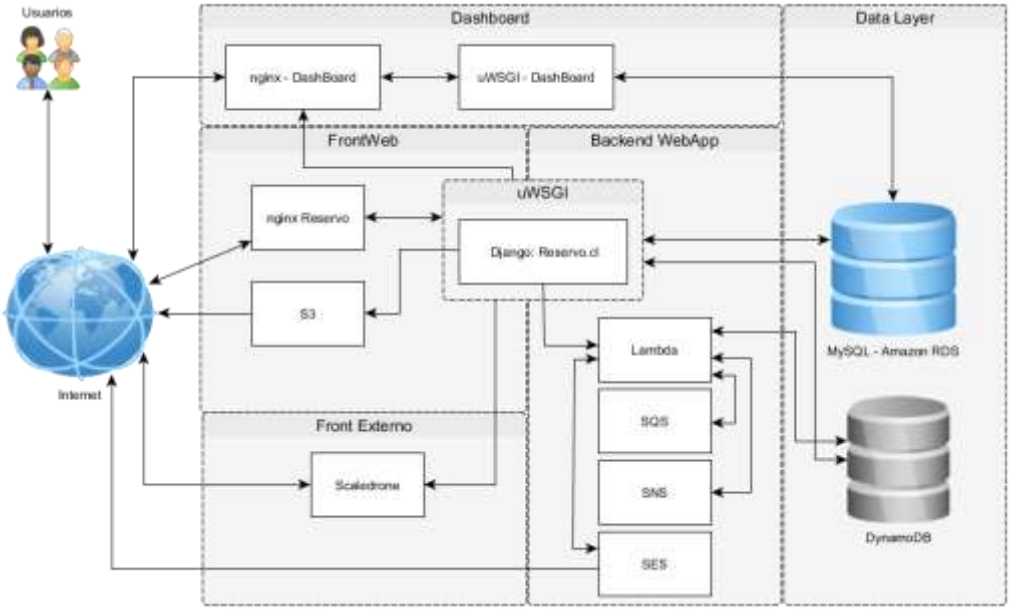

Figura 23: Arquitectura propuesta, organización lógica

Esta separación tiene como mejora esencial la remoción de *Celery* y *Redis*, reemplazándolos por los servicios *Lambda*, *SQS*, *SNS*, *DynamoDB*. Estas componentes tienen escalabilidad y alta disponibilidad integrada. La seguridad por omisión consiste en que solamente pueden acceder las máquinas virtuales y los procesos corriendo en la cuenta usada para '*Reservo*', o aquellas a las que se les otorgue acceso específico, aunque en este caso no se hace uso de esa característica.

Se recomienda mantener el lenguaje de programación *Python*, y su framework *Django*, así como el servidor Web *NGINX* y el servidor de aplicaciones *uWSGI*. Se recomienda el uso del motor Base de Datos *MySQL*, *el s*istema de colas *Amazon SQS*, el servicio de notificaciones *Scaledrone*, *el s*istema de BD *NoSQL Amazon DynamoDB, el s*ervicio On-Demand Computing provisto por *Amazon Lambda, el s*ervicio de notificaciones (para *Lambda*) *Amazon SNS*, y el servicio de envío de correo *Amazon SES*.

5.2.2 Mejoras en asignación de componentes a servidores

A continuación se presenta la figura con nuevos componentes, y se detalla cómo funciona la solución.

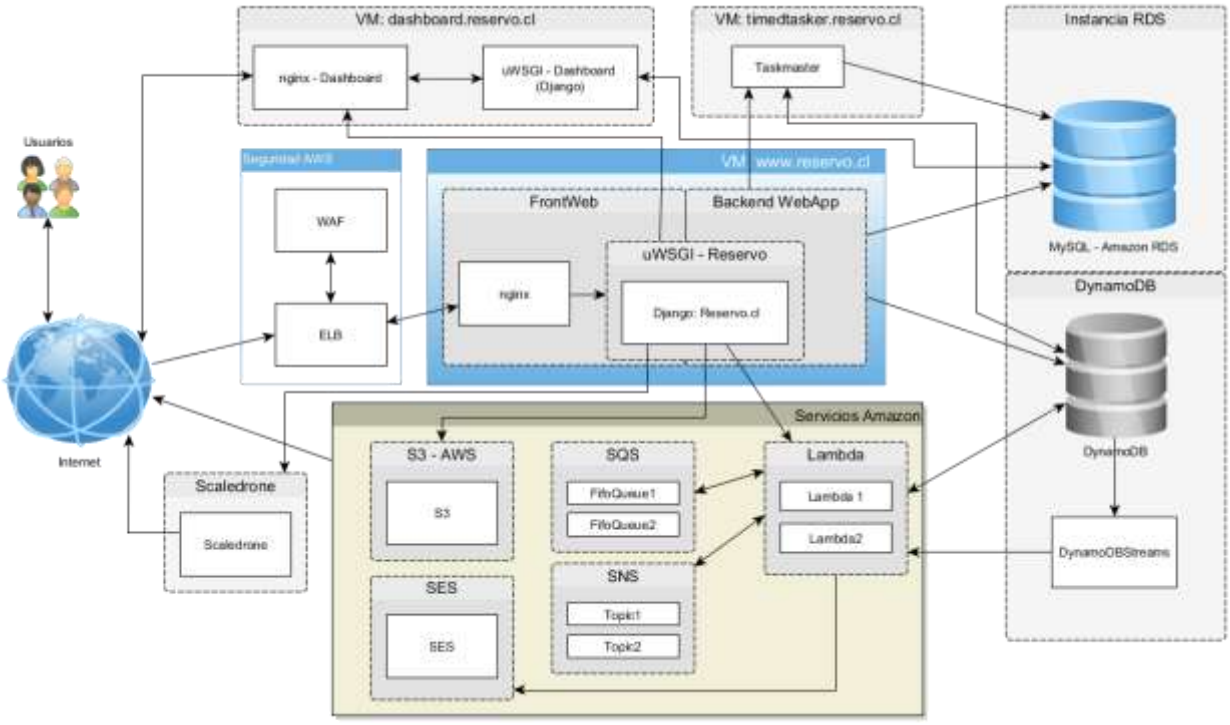

Figura 24: Propuesta de Distribución de componentes en Servicios y Servidores

Los servicios escalables y altamente disponibles de la solución son los siguientes:

- *DynamoDB*: Servicio de almacenamiento de datos *NoSQL*, organizado por *tablas*, con elementos de la forma *{clave, valor1, valor2, ... }*.
- *DynamoDBStream*: Sub-servicio utilizado para detectar eventos de cambio en una tabla-DynamoDB en forma asíncrona.
- *SNS*: Servicio que permite crear *tópicos*[33] y suscribir elementos allí; esto va desde casillas de correo hasta funciones *lambda*. SNS funciona como un canal *one-to-many*.

[33] *SNS-topic*, o *topic-SNS*

- *Lambda*: Se definen funciones llamadas '*Lambda1'* y '*Lambda2'* como ejemplos genéricos. Éstas pueden ser lanzadas en forma manual, o automática mediante algún disparador (*trigger*) configurable, de acuerdo a situaciones como las siguientes:
    - Cuando se ingresan nuevos elementos a una tabla de *DynamoDB*, estando habilitado el *stream* respectivo, y se utiliza como trigger de suscripción.
    - Si una *lambda* está suscrita a un *tópico SNS*, cuando se envía un mensaje a ese tópico (ya sea desde una aplicación web, otra *lambda*, o ingresado directamente), se gatillará el lanzamiento de la lambda.
- *SQS*: Servicio de cola distribuida y altamente disponible que permite crear dos tipos de colas: *Queues* y *FifoQueues*. Para efectos de '*Reservo*', sólo se usarán las segundas debido a que proveen un requisito fundamental: garantizan el orden en el envío y recepción de los mensajes.

Lambda y este juego de servicios, permite implementar una aplicación con una arquitectura orientada a microservicios y de base *Serverless*. Esto se puede lograr enlazando las diferentes lambdas mediante suscripción a tópicos *SNS* específicos, como si fuera una posta.

Después del tiempo de estudio, se determinó que el esquema recién descrito, es la mejor manera de enlazar funciones *lambda*, debido a que si bien se puede llamar a una lambda desde una ya en ejecución (padre), el 'padre' no termina hasta que el 'hijo' retorna. Por lo tanto, eso es tiempo muerto de proceso, y con el límite de procesamiento de 5 minutos, puede haber efectos inesperados difíciles de reproducir.

La ventaja principal de usar este esquema, radica en que reduce la carga de uso de procesador en el *nodo*, permitiendo atender más usuarios. Además, permite ganar escalabilidad, disponibilidad y seguridad para la aplicación web. Como desventajas se visualizan las siguientes dos:

1. El utilizar esta forma de desarrollo, deja a '*Reservo*' altamente dependiente de AWS, y por lo tanto cambiarlo a otro proveedor (o crear una réplica, en Google Cloud) requiere un nuevo trabajo y estudio de entrelazado de servicios detallado. Además, representa otra base de código a mantener, o implica la necesidad de crear una API interna que envuelva los servicios, para hacer que la aplicación se abstraiga del proveedor *cloud* en que se ejecute. Sin embargo, esto es algo que está fuera del alcance de este trabajo.

2. En ocasiones los mensajes *SNS* pueden ser entregados más de una vez, lo que haría que una *lambda* sea gatillada más de una vez. Este sería un impacto menor, y sólo se traduciría en unos costos levemente mayores a lo que corresponde, en los casos que usen *FifoQueues* en *SQS*. En caso de que accedan a elementos de una *FifoQueue*, el primero podría verlos, pero el segundo no. Así se cumpliría la condición de que si hay mensajes nuevos que el primero no haya sacado, funciona como en el caso normal pero con otros mensajes. Sin embargo, si no hay ninguno más, entonces llega a la condición de término.

Máquina Virtual Adicional: TaskMaster

En la solución se define una nueva máquina virtual orientada al procesamiento sincronizado, para casos como el de correos que deben enviarse a horas específicas. Este servidor debe estar configurado en zona horaria UTC[34]. Esta máquina puede ser después utilizada para otros propósitos, como envíos masivos de correo u otros que requieran un gran procesamiento por lotes.

Solución al Caso 1: Zona Horaria

Para cada caso se puede implementar un script diferente en cron, que corra cada una hora (o con la frecuencia que se requiera). Por ejemplo, un caso para el recordatorio a enviarse a las 08:00 hora local, sería que a las 11:00 UTC: (1) busque todos los clientes que tengan zona horaria UTC-3, (2) para cada uno de esos clientes, obtenga la agenda de hoy, (3) vea a quiénes se les debe enviar correo, y (4) cree un correo con los detalles, personalizado para cada cliente, y enviar.

El invariante base es el siguiente: dada la hora actual en UTC, calcular en qué zonas horarias son las 08:00, y determinar todos los clientes que se encuentren en dichas zonas horarias. Con esa lista de *Clientes*, se pueden obtener las agendas, crear todos los recordatorios, enviarlos desde esta misma máquina a medida que se generan, y enviarlos mediante llamadas directas en script a *SES* o vía *Lambda+SES*.

Siendo preciso, hay que considerar el caso en que un cliente tenga sucursales en la XI y XII región, siendo la misma empresa. En ese caso, en ciertas fechas del año, las sucursales tendrán diferencia horaria aun estando en el mismo país. Este es un caso de borde que convendrá abordar como parte del trabajo a futuro.

Solución al Caso 2: Zona Horaria

El invariante en este caso sería similar a la solución del punto anterior, en cuanto a determinar las zonas horarias para efectuar cierre y apertura de caja en forma automatizada, de acuerdo a la hora en UTC del servidor. Se mantendría el mismo caso de cuidado con las sucursales de las regiones XI y XII. Esto puede ser algo realmente crítico dependiendo del tipo de negocio del *Cliente*, como por ejemplo un centro médico, donde la congruencia de las horas es relevante para *Clientes* y *Pacientes*, y no puede haber errores

5.2.3 Funciones lambda creadas específicamente para '*Reservo*' actual

La necesidad de disminuir carga de proceso en los *nodos*, al ser analizada teniendo a la vista la arquitectura planteada en [5.2.1], se llegó a la conclusión de que usando *Lambda*, *SNS*, *SQS*, *DynamoDB* (con y sin *Streams*) se logra cumplir con el requisito, y además permite atender más usuarios en un *nodo*.

[34] *UTC*: Coordinated Universal Time, estándar horario utilizado como hora 0 en el mundo. Coincide con la línea del meridiano de Greenwich, pero cuenta con los segundos intercalares para corrección de desfases.

Como mejoras adicionales por el uso de los servicios de *AWS*, se ganan Escalabilidad, Alta Disponibilidad y un grado mayor de seguridad en los datos.

En general, usando microservicios se debe tener cuidado con los siguientes temas:

- *Al implementar y probar*: Evitar que haya lanzamientos totalmente arbitrarios y en cantidades variables, si se desea tener un control de costos. Se debe seguir un diseño adecuado para evitar un crecimiento desmesurado, para el caso de '*Reservo*'. En algunos casos podría ser necesario algún tipo de encolamiento que garantice que ciertas llamadas serán ejecutadas una sola vez.

- *En cuanto a mantenibilidad*: Cambios en la forma de llamar puede que requieran un cuidado mayor en el versionamiento, y en los pasos de construcción automática en pasos a entornos de *Test, Staging, Prod.*

Esto se encuentra en proceso de implementación, a la fecha de entrega.

Envío de email, 5 minutos después de creada una cita

Cuando un *Paciente* toma una hora en un *Cliente*, después de cumplidos 5 minutos, el sistema debe enviar un correo electrónico hacia el *Paciente* con los datos de la reserva. El nivel de registro, de momento no se maneja, salvo en *SNS*. A nivel de '*Reservo*' podría ser necesario saber incluso el tipo de error que da una casilla de e-mail, al enviar un correo de confirmación. Por un lado, a SC3 le es relevante saber esto debido a que, por rebotes o errores de envío, pueden suspender temporalmente los envíos de e-mail. Para dar solución a esto, se vieron dos enfoques posibles para el envío de estos mensajes: utilizando *DynamoDB Streams* o *SNS* como trigger.

Al recibir el sistema los mensajes de una cola *SQS*, estos no son borrados automáticamente, sino que se hacen 'invisibles' a otros consumidores por una duración de tiempo, mediante una recepción de un *receipt_handle*. Ese *receipt_handle* es como un cupón con una expiración de tiempo, que indica por cuánto tiempo tendremos la visibilidad exclusiva del mensaje. Sin ese valor, es imposible remover el elemento de la cola. Cuando se leen mensajes de una cola *SQS*, el máximo que entregará será siempre 10 mensajes. Por lo que para leer varios, como en nuestro caso, hay que tener un ciclo de lectura con la condición específica de que, si recibe dos veces una indicación de que no hay nuevos mensajes, o se pasa del tope, debe seguir el procesamiento.

Dentro de los experimentos dejamos un *timeout* de recepción de cerca de 295 segundos (casi 5 minutos: máximo tiempo de ejecución de una *Lambda-function)*. La idea es que un e-mail sea generado y enviado, sólo si el *timestamp+300 segundos* es menor o igual al *timestamp* actual. Aquí se miden valores enteros absolutos, *epoch-time*, considerados como si fuera UTC. En otras palabras, enviamos un e-mail después de que hayan pasado 5 minutos, pues el mensaje saldrá una vez que el timestamp está vencido.

En la 'preparación de un mensaje' la *lambda-function* debe poder conectarse a la base de datos de '*Reservo'*, para obtener todo lo necesario para crear el mensaje para el *Paciente*. Como se sabe todos los elementos a traer, sería posible hacer una única consulta que traiga todos los

elementos que cumplan con ciertos identificadores únicos, y almacenar el resultado en un 'dict' para luego usarlo en dicha tarea.

En un futuro, todos los cambios de estructura, por ejemplo en el modelo de datos, también deben verse reflejados en las *lambda-functions* que sean utilizadas. Los cambios serían a nivel de datos que se pasarían por *SNS*, mensajes a las colas, etc. Si no se reflejan apropiadamente los cambios, podrían haber errores que serían difíciles de encontrar. Los nombres son referenciales[35], por lo tanto deberían incluir un prefijo que identifique el nombre de proyecto interno de SC3 al que pertenece.

Uso de DynamoDB Stream como trigger

La siguiente figura ilustra el uso de DynamoDB Stream como trigger, y luego explica el proceso paso a paso.

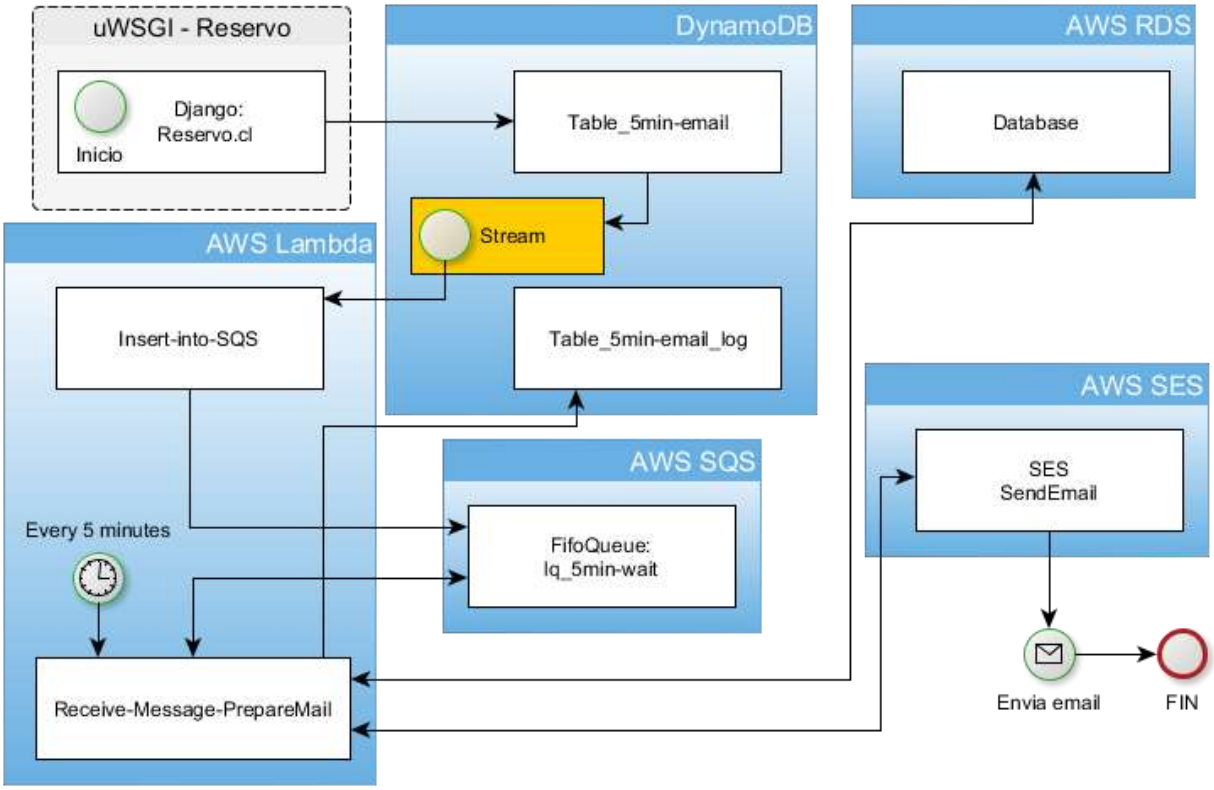

Figura 25: Uso de *DynamoDB* como trigger desde *Django*

Los pasos por seguir son los siguientes:

1. El usuario (ya sea *Cliente* o *Paciente*) creará una cita, y por lo tanto *Django* hará una inserción en la tabla '*Reservo_5min-email*'. Esta tabla se encuentra creada de antemano con un stream activado. Estos *streams* tienen un *ARN*[36] específico, y son parte del

[35] Naming Things, según Martin Fowler: https://martinfowler.com/bliki/TwoHardThings.html
[36] *ARN: Amazon Resource Number*, un identificador específico para el recurso de Amazon, útil cuando se programa con las bibliotecas de ellos.

servicio *DynamoDB*. Los datos por guardar deben ser suficientes para identificar exactamente la cita en un paso posterior, incluyendo el timestamp desde el servidor.

2. Con el *stream* habilitado se lanza un evento, el cual puede ser un INSERT, UPDATE o DELETE. La *lambda-function* 'Insert-into-SQS' está suscrita a este evento como *trigger* de lanzamiento. En su definición, sólo hará trabajo útil si es que el evento es de tipo INSERT. En dicho caso, recopilará los datos que fueron insertados desde el mismo stream.

   Cuando la función 'Insert-into-SQS' es lanzada, tomará del *Stream* qué fue lo que se insertó, y los ingresará en una cola *FifoQueue* en SQS, incluyendo una marca de tiempo generada por *Django.*

3. Existe otra *lambda-function* llamada '*Receive-Message-PrepareMail*', la cual se ejecuta cada 5 minutos y funciona de la siguiente forma:

   - Verifica si hay mensajes en la cola '*lq_5min-wait*'. Si no hay, no hace nada. Si existen mensajes, entonces lee hasta 500 mensajes, o bien que dos lecturas seguidas de la cola, con una espera de 5 segundos entre ellas, retornen 0 mensajes. Esto indica que no llegaron más mensajes.
   - Procesa los mensajes recuperados verificando si están vencidos o no, siempre obteniendo el *receipt_handle.* Si el mensaje cumple con el criterio de vencimiento (pasaron al menos 5 minutos de la creación de la cita), entonces se almacena en una lista incluyendo su *receipt_handle*. Luego se recorre esa lista preparando cada mensaje, enviándolo, y una vez enviado se realiza lo siguiente:
     - Se remueve el mensaje de la cola, especificando el *receipt_handle*.
     - Se registra el resultado del envío (OK o error) en la tabla '*5min-email_log*'.

Con este esquema se tiene solucionado el problema de que los correos salgan después de 5 minutos, y haya mucho uso de CPU.

Uso de SNS como trigger

La siguiente figura ilustra el uso de SNS como trigger. En este caso, desde *Django* se envía un mensaje específico a un *SNS-topic*, que tenga una *lambda* suscrito. Éste lo único que hace es recibir datos desde SNS y guardarlos en la cola (SQS) respectiva. Después sigue la *lambda* que se lanza cada 5 minutos para revisar la cola. Los pasos a seguir en este proceso son muy similares a los descritos para cuando se usa *DynamoDB* como trigger por lo tanto, no serán detallados en mayor medida.

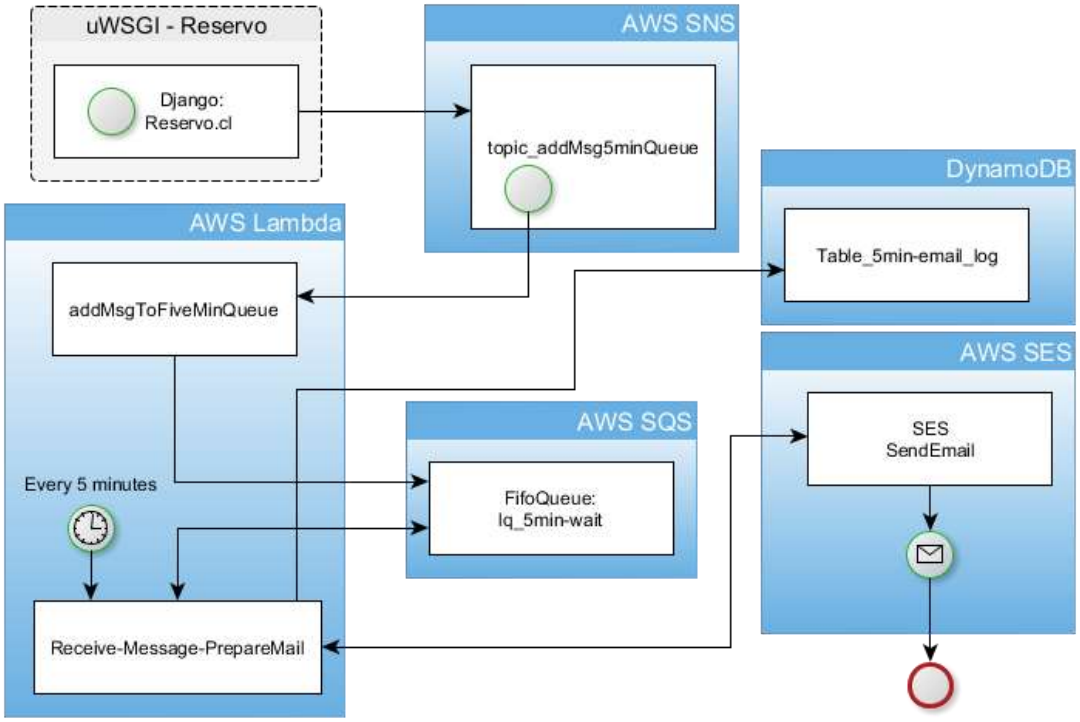

Figura 26: Uso de *SNS* como trigger desde *Django*

Los pasos por seguir son los siguientes:

1. Al crearse una cita, *Django* hará la llamada a una función para enviar un mensaje al *SNS-topic addMsg5minQueue*, con lo que se lanzará la *lambda* '*addMsgToFiveMinQueue*'.
2. La función *lambda* '*addMsgToFiveMinQueue*' toma los datos recibidos, y crea un mensaje *SQS* en la cola '*lq_5min-wait*'. Esos datos la función '*Insert-into-SQS*', ingresará en una cola en SQS de tipo FifoQueue, llamada '*lq_5min-wait*', los datos de identificación con el timestamp en el que fue ingresado en Django.
3. La *lambda-function llamada* '*Receive-Message-PrepareMail*', se ejecuta cada 5 minutos y realiza las mismas acciones descritas en la sección anterior.

Justificación de la opción escogida

Cabe mencionar que en cualquiera de las situaciones, debido a que los correos son removidos de la cola en *SQS* a medida que se envían, en el caso de que un correo no logre enviarse debido a expiración de tiempo de *lambda*, será incluido en la siguiente ejecución agendada cada 5 minutos, dentro de los primeros lugares, debido a que se está utilizando la cola de tipo *FifoQueue*.

La mejor opción, la cual está en proceso de implementación en este momento, es la que tiene como trigger el mensaje de *SNS*. La principal razón es simplemente porque se usa una tabla menos de *DynamoDB*, lo cual implica menos tiempo de mantenimiento, y menos costos mensuales. Respecto a esto, en una tabla con streams activados, si tiene una *lambda* suscrito, cada vez que ocurra un evento, el proceso será lanzado. Cada lanzamiento de *lambda* hasta el término puede ser

desde 0,5 segundos hasta 1,5 segundos en el caso de no hacer nada. Este es un costo evitable si no se usa tabla con streams que deba ser limpiada a futuro, como sería este caso.

Es decir, que cuando se quieran borrar elementos uno por uno de *DynamoDB*, se lanzará una *lambda*, con una duración inferior a 1 segundo por cada elemento. Amazon factura desde los 300 milisegundos como base, contando de 100 en 100, preocupándose ellos de aumentar en forma automática la memoria y CPU usados. No hay forma simple de "limpiar la tabla", como en SQL donde es suficiente con realizar un *DELETE FROM TABLE* (sin un *WHERE*).

Para borrar todo el contenido de una tabla de *DynamoDB*, debe eliminarse la table completa; con ello, se borra el *stream.* Al crear uno nuevo, tendrá un valor identificador diferente (cambia el *ARN)*, pues ese identificador incluye un *timestamp* de cuando fue generado; y eso puede hacer al código más complejo.

Envío de emails masivos tipo campaña por parte de un cliente

En este caso lo que se busca es despachar los emails lo más rápido posible, dentro de las restricciones existentes. Amazon ofrece una cadencia de envío de correos por hora, pero configurable de acuerdo al precio que se esté dispuesto a pagar.

La forma actual en que esto funciona es la que se explica a continuación. *Django* obtiene los datos para el envío de correos, crea los contenidos de los mensajes, y mediante almacenamiento en *Redis* y una notificación a *Celery* de que debe efectuar el envío de correo.

El nivel de registro necesario para esta acción, como mucho es saber cuántos correos fueron correctamente enviados; no necesariamente cuáles correos. En un principio se pensó en utilizar una forma similar a la del caso anterior, con una inserción en *DynamoDB* como *trigger*, pero resulta tener el mismo problema que en el caso anterior con respecto a limpieza de las tablas. Sin embargo, mediante un aviso en *SNS*, es posible lanzar una *lambda createEmailsSaveToSQS* cuya misión es obtener todos los datos para generar los correos a enviar, crear el cuerpo de email instanciado y almacenarlos en una cola *SQS*. Luego los mensajes pueden ser enviados utilizando alguna de las siguientes alternativas:

a) En otro proceso *ProcessAndSendMassEmails*, en forma temporalizada cada 5 minutos, se puede enviar la mayor cantidad de mensajes posibles, y guardar un registro de envío, actualizando totales existentes para dicho envío.

b) Sin salir de *lambda-1*, sin usar la cola *SQS*, se puede indicar los datos para enviar el correo mediante *SNS* a una función *lambda-2b. Esta última* se encargará de crear el mensaje instanciado, y enviarlo.

La primera opción (a) es lo que se ve más administrable, ya que los correos serán enviados eventualmente; la Figura 27 ilustra el proceso que se debe seguir. La segunda opción (b) es para pensar en implementar durante una etapa futura, en la que haya sobre 5 mil clientes. Dado que no es una prioridad que los e-mails salgan imperativamente rápido, por lo tanto lo mejor es utilizar la opción (a).

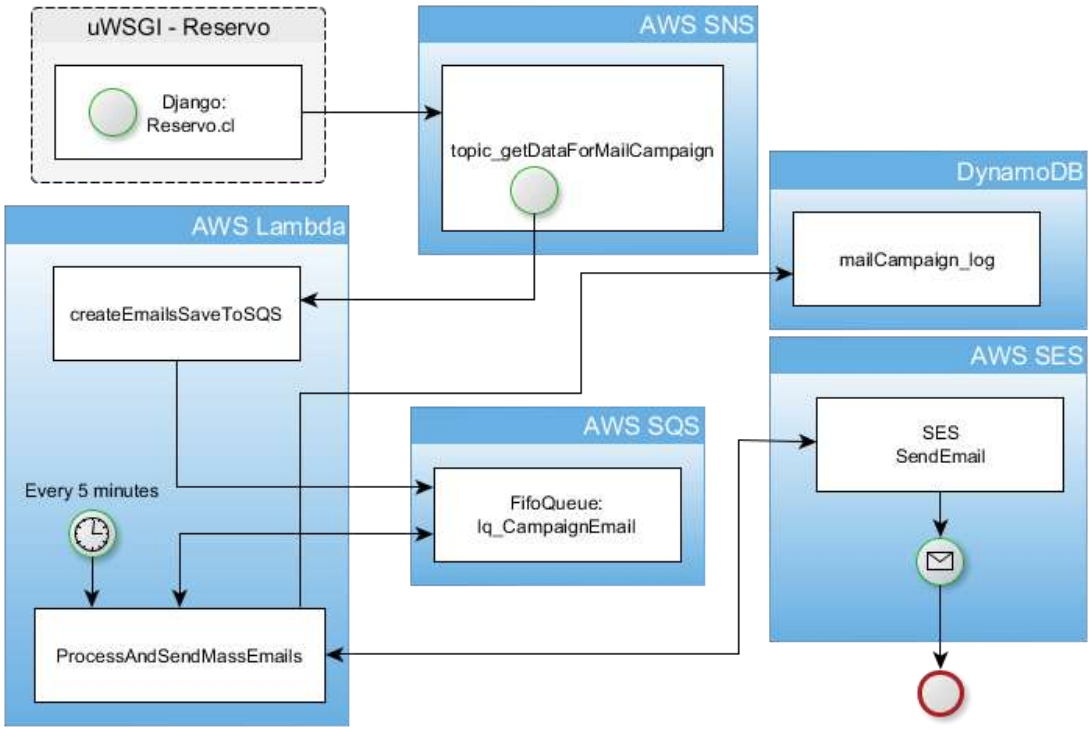

Figura 27: Forma (a) para envíos masivos

5.2.4 Otras acciones de reducción de carga en *nodos*

Para reducir la carga de la máquina virtual que corre '*Reservo*', el principio es sacarle de encima tareas no esenciales para su procesamiento. Mediante uso de servicios provistos por Amazon, se puede quitar lo siguiente:

1. Procesamiento de HTTPS, usando el servicio *ELB* como *endpoint.*
2. Balanceo de carga de peticiones HTTP, con *ELB.*
3. Balanceo de carga de acceso a Base de Datos.
4. Escalamiento horizontal, es decir usar múltiples nodos de procesamiento. Se entiende como un nodo a una máquina virtual corriendo *Nginx+uWSGI* para '*Reservo'*.

Un punto importante que puede influir en la complejidad de la solución es el requerimiento de cifrar la información de las fichas médicas, que en toda la red interna el tráfico hasta el motor de base de datos. Al menos en temas de una entidad que genera certificados digitales para propósitos de identidad, todo el tráfico dentro de la red en sistemas productivos y piloto, debe estar cifrado de punta a punta[37].

[37] Fue mi experiencia laboral en E-Sign, con reglas de Verisign-Symantec. Aparece en un documento propietario llamado "Security and Auditory Requirements", que especifica como un deber el que en sistemas críticos de una CA (*Certification Authority*), el tráfico se encuentre cifrado, debido a que los certificados se utilizan para validar identidad,

5.2.5 Registro de acciones de usuarios

Esto es una necesidad para el caso que ya se tiene escalamiento horizontal, con múltiples nodos de procesamiento. Se debe usar un 'Correlacionador de Logs'. En cada nodo de procesamiento, se debe poner un agente de colección hacia un servidor central de logs de *Nginx* y *uWSGI*, que tenga alta disponibilidad de red. El uso de ese servidor de correlación permitirá mantener un registro fidedigno y consistente de las acciones de un usuario de un *Cliente*, independientemente de cual nodo atienda las peticiones en instantes diferentes del día.

5.3 Mejoras en capa de datos

No se incluye en este documento un detalle del modelo de datos utilizado en '*Reservo*', debido a que como se mencionó antes, éste supera las 300 tablas y no es particularmente relevante para este trabajo de memoria. Para los efectos de las mejoras a abordar en este trabajo, dado que no se cambiarán las funcionalidades generales, pero si el determinar datos de cita, las modificaciones en esta etapa serán muy pocas.

Para abordar lo expuesto en la sección [4.3], se debe convertir la base de datos a un modelo Multi-tenant. Para esto, se plantea un modelo en el que se permitirá mantener la estructura completa actual de base de datos, con una meta-BD que permita ver con cual base de datos debemos operar. Esto permite mantener a los clientes actuales operativos en la BD existente, mientras que es posible tener cada nuevo cliente en una base de datos, para compartimentalizar en forma efectiva toda la información.

Después, en forma paulatina es posible migrar de a poco los clientes existentes a una base de datos dedicada para cada uno, e indicar de qué base de datos deben ser obtenidos los datos de la agenda. Los cambios que resulten de estos esquemas deben implementarse tanto en '*Reservo*', como en el back-end (llamado *Reservointerno)*, para mantener la consistencia. La forma de generar respaldos debe considerar a todas las bases de datos existentes.

Las mejoras propuestas a la capa de datos permiten abordar las limitaciones de seguridad, escalabilidad y disponibilidad de la plataforma utilizando herramientas provistas por Amazon, teniendo como consecuencia una solución permanente a los problemas encontrados durante la etapa de recopilación de información y diagnóstico.

5.3.1 Modelo para la capa de datos

En '*Reservo*', la unidad nuclear es la cita que tiene agendada un *Paciente* en un *Cliente*. En el esquema actual cada cita tiene un identificador único, y cada cliente también tiene un identificador único. Dado un universo de varios *Pacientes* agendando *citas* en diversos *Clientes*, cuyos usuarios llegan a través de www.reservo.cl, pero cada uno con una base de datos exclusiva,

---

o firmar aplicaciones; adicionalmente de que la red es monitoreada con sistemas que inspeccionan tráfico, por lo que se debe garantizar confidencialidad.

la forma de identificar una *cita* en forma única sería mediante una tupla (*id_cliente*, *id_cita*). Para ello, se requiere una meta-base de datos, que indique para cada *Cliente*:

- BaseDeDatos: cruzado contra el *id* del cliente,
- El nombre de base de datos, donde se encuentran su agenda.
- El host en que está alojada la BD, incluso el puerto, por si acaso.
- Estado del cliente: Activo, Mantención, Desactivado.
- DatosCliente: Esto podrían ser varias tablas; habría que ver el caso cuando sea necesario modificar para soportar sucursales de un mismo cliente que estén en zonas horarias diferentes.
- LocacionCliente: El id_cliente, zona-Horaria y locación permitirá tener una configuración específica incluso con sucursales en zonas horarias distintas.

Esta meta-base de datos puede ser implementada en *DynamoDB* o en *RDS* (con una base de datos sólo para ese propósito), utilizando el servicio *Elasticache*[38] configurado con *Redis* como un caché de lectura. Esto permite tener un mejor desempeño cada vez que se necesite acceder a estos datos. De esta manera podemos lograr un escenario en el que sea posible reconocer si un cliente está en la base de datos antigua, o en una nueva, y trabajar con la que corresponda.

Se requiere una forma de generar los identificadores de cita y de cliente en forma consistente con lo actual, iniciando la cuenta en el valor actual en la base de datos. La generación del número de puede dejarse a alguna de las dos maneras siguientes:

1. Usar un identificador universal de cita para todos, lo que requeriría la creación de un servicio dedicado, en base a un tipo de número entero "infinito". Eventualmente los números de 64 bits podrían quedarse cortos en la teoría; en la práctica podría no pasar hasta después de 40 años de operación y varios miles de clientes.

2. Que en cada base de datos se administren los identificadores de cita, y utilizar el identificador compuesto (cliente, cita). En ese caso sólo habría que asegurar que la generación de *id* de cliente sea consistente. Esto puede efectuarse mediante un servicio que consulta datos, y cuando tiene que generar un id nuevo, que sepa cuál es el último creado anteriormente. De esa manera se soluciona directo con la meta-base de datos para Clientes.

El segundo enfoque es mucho más razonable para todo efecto práctico, e implica menos piezas de software que mantener directamente por parte de SC3.

En la Figura 28 se muestra cómo quedará la distribución lógica de bases de datos. Los *Clientes* existentes, mientras no sean migrados, estarían en 'ReservoBD'. Sin embargo los *Clientes* nuevos cada uno quedará en su respectiva base de datos, y la correspondencia se encuentra en 'ClientDatabase'. Esto está pensado considerando el segundo enfoque de los planteados recientemente.

[38] *Elasticache*: Otro servicio de AWS, que provee caché en RAM, administrado por ellos

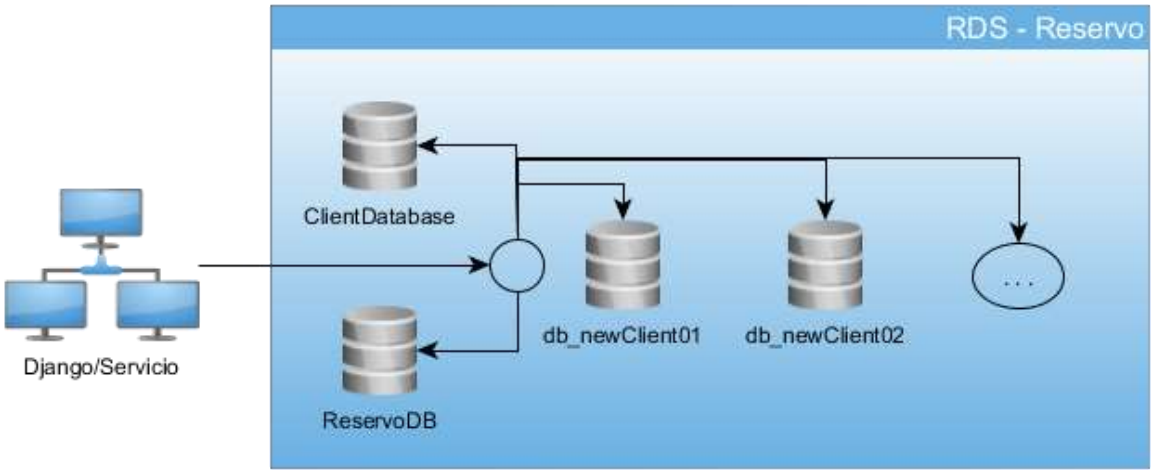

Figura 28: Distribución de bases de datos en esquema *Multi-Tenant*

A futuro, para incrementar el nivel de seguridad en el acceso a los datos desde la aplicación, conviene implementar un Servicio de Acceso a Datos, que evitaría efectuar las consultas directamente a la base de datos desde la aplicación. Esto ofrece una barrera de contención en caso de que la aplicación fuese comprometida, evitando que sea posible extraer todos los datos críticos de *Pacientes* mediante una sola consulta.

Finalmente, se debe crear un servicio para administrar los clientes, incluyendo algún mecanismo partiendo desde los existentes. Este servicio podría ser implementado en forma local (mediante *django-rest-framework*) con una API, o utilizando microservicios basados en *lambda* activados por *AWS API Gateway*. Ambos tienen sus ventajas y desventajas. A continuación, se ilustrará ambos casos, y se hará un listado de pros y contras.

En ambos casos, para aumentar la seguridad de la aplicación conviene considerar un servicio generador de tokens de tiempo limitado, y un solo uso para un recurso específico, como una base de datos particular. El servicio también debe permitir verificar si el token es válido en la operación, para el recurso que está accediendo. Este servicio debe ser independiente de '*Reservo*', y contar con alta disponibilidad y escalabilidad. Si no se utiliza, lo único que se puede garantizar es que un usuario no inyecte consultas en forma directa, pero no se da una validación de si el usuario tiene permisos para acceder a dicho recurso de datos.

Las ilustraciones suponen el uso de la API de tokens, aunque como se mencionó antes, pueden ser obviadas. La siguiente figura muestra el uso de la API Django-REST-Framework.

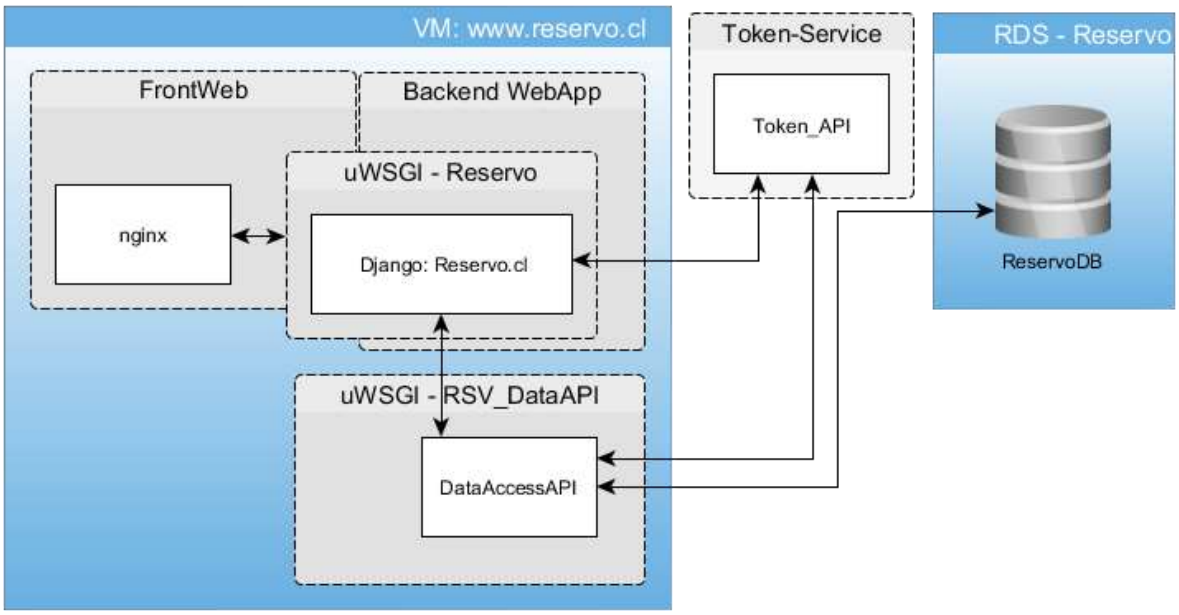

Figura 29: API Local en un uWSGI adicional en la misma máquina virtual

Como aspectos positivos esta solución brinda comunicación rápida (sólo 2 salidas a servicio externo) y una buena mantenibilidad. Sin embargo involucra un importante uso de recursos en nodos. La siguiente figura muestra el uso de una API que usa lambdas lanzados mediante AWS API Gateway.

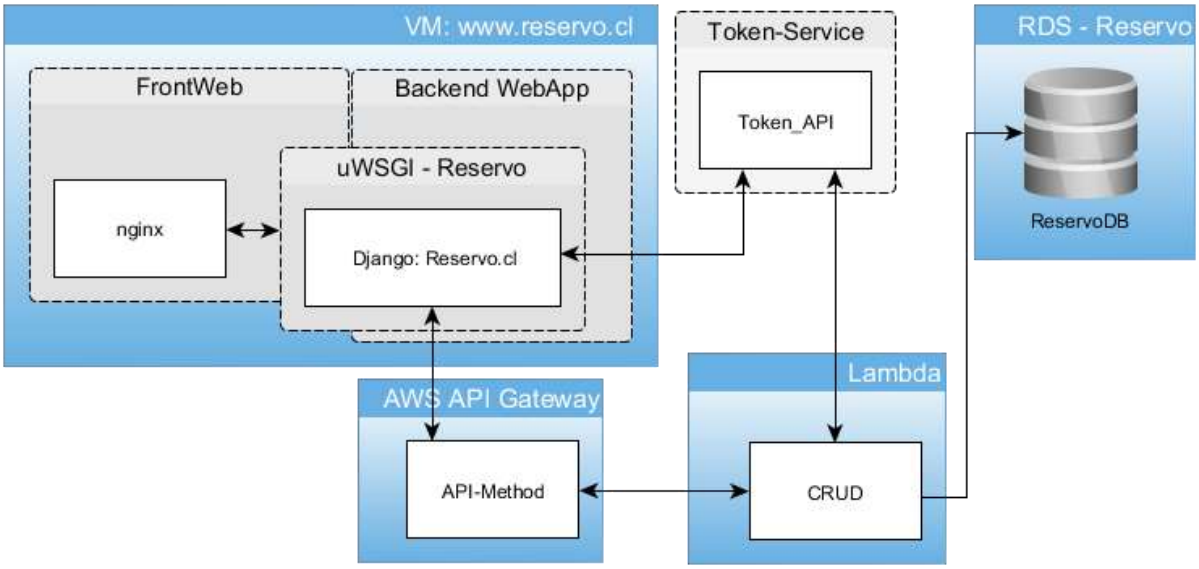

Figura 30: API usando *Lambdas y API Gateway*

Como aspectos positivos de esta solución podemos decir que cuenta con un buen uptime potencial (disponibilidad), es automáticamente escalable, e involucra menos código en aplicación. Sin embargo, los tiempos de comunicación son levemente mayores que la opción anterior, debido al acceso a servicios externos.

Sobre cuál de las dos opciones elegir, la decisión corresponde a SC3; en pos de la seguridad, la recomendación es ir por el de *Lambda+API-Gateway*, debido a que en caso de compromiso de la máquina virtual, no se podría llegar a subvertir este servicio en forma arbitraria.

5.3.2 Proceso de cambio

Dado que esto es un paso crítico en cuanto a trabajar en la escalabilidad del sistema, las etapas a seguir son las siguientes:

1. Implementar meta-BD para *Clientes*, con la intensión de que al final de esta etapa, el software esté listo para una posible migración. Por lo tanto, los pasos a seguir son los siguientes:
    - Crear una meta-base de datos sólo de clientes, de acuerdo a lo indicado en el punto anterior.
    - Hacer las modificaciones en la plataforma de software '*Reservo*' y en el módulo '*reservointerno'*, con el fin de poder operar con la nueva base de datos de clientes, y con la base de datos de Reservo actual. Por lo tanto, cuando un nuevo cliente es creado, su registro se crea la base de datos correspondiente.
    - Verificar funcionamiento de plataforma.

2. Asegurarse que los respaldos de bases de datos sean tomados en horas fuera de servicio para los clientes, por ejemplo, a las 03:00 AM en hora local de cliente. Para esto se debe ser consistente con la zona horaria correspondiente. Para lograr este objetivo se debe:
    - Crear un script que recorra la meta-BD viendo qué clientes hay que respaldar, y luego ejecutar esos respaldos uno por uno.
    - Dejar la ReservoBD actual para el final del proceso de respaldo.

3. Crear una prueba de concepto de migración de un cliente particular a su propia BD. Para ello se debe:
    - Poner al registro del cliente en estado Mantención.
    - Copiar todas las tuplas de datos que le correspondan a la nueva tabla.
    - Actualizar los valores de identificadores de tablas con autoincremento para que sean consistentes. Por lo tanto, una nueva cita debe tener un identificador igual al siguiente valor de la última cita registrada en la nueva base de datos.
    - Sacar de mantención al cliente, probar que éste vea sus datos correctamente, y verificar que opera normalmente 'Reservo' para él.

De esa forma es posible ir migrando a todos los clientes, a través de un proceso batch que involucra a algunos pocos de ellos por día. Esto permite que quede cada uno en su propia base de datos, y así dejar la seguridad de sus datos dentro del estándar que se utiliza en la industria de *Software-as-a-Service*.

5.3.3 Acciones para el futuro

Dado el escenario de trabajo de 'Reservo' y el estado actual de dicha plataforma, hay varias acciones que sería bueno tomar en el futuro, pero que actualmente no están dadas las condiciones para llevarlas a cabo. Una de ellas es trabajar con bases de datos *RDS* con el almacenamiento cifrado. Amazon AWS provee una opción para hacer esto en forma simple, siguiendo los siguientes pasos:

1) Crear una API de acceso a datos como un servicio al que se le solicita efectuar operaciones *CRUD*[39] mediante *lambdas*. Cada uno utiliza 'cupones' generados en el momento, los cuales son válidos para solicitar un acceso por un tiempo limitado, y para un usuario específico. De esta forma se potencia la seguridad del acceso a datos de la aplicación por parte de usuarios autorizados. Además, se logra eliminar los riesgos de inyecciones directas de consultas SQL. Para realizar esto se necesita crear y/o utilizar:

    a) Un servicio generador de cupones aleatorios que evite re-usos y no sea predecible. El cupón debe poder ser validado aparte, y el servicio debe ser auto-escalable y altamente disponible.

    b) Funciones lambda para efectuar operaciones CRUD en la base de datos. En este caso se necesitaría crear y mantener una función para cada operación diferente.

La alta disponibilidad y escalabilidad del servicio de acceso a datos se puede lograr con el uso de servicios como '*AWS API Gateway*' como punto de llamada, uso de *Lambda* y de otros servicios. Sin embargo, en el caso de crear el servicio para mayor seguridad, el punto crítico sigue siendo la conexión a la base de datos en sí, como se ilustra en la figura siguiente.

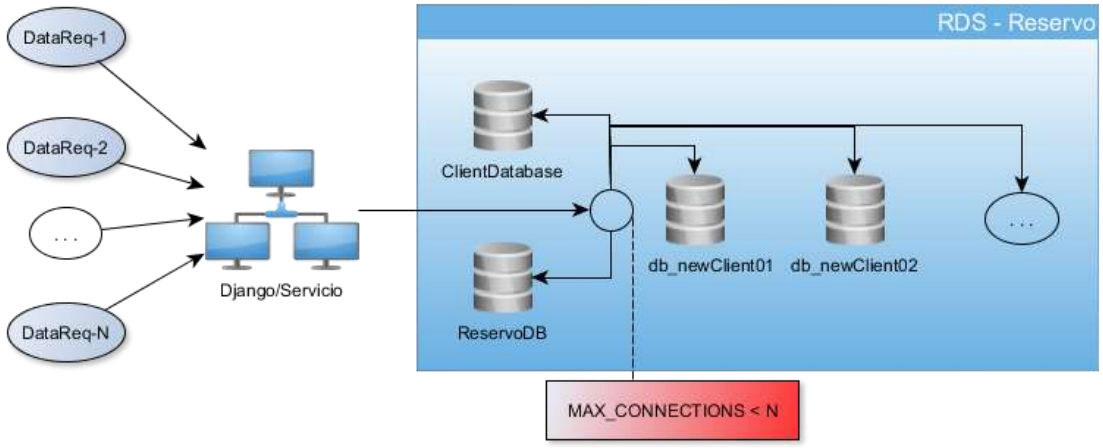

Figura 31: Más conexiones que el máximo de la base de datos

Si se lanzan demasiadas *lambdas*, eventualmente se pueden copar las conexiones a la base de datos. Por lo tanto, mientras no sean desocupadas las conexiones, la aplicación mostrará errores de conexión a los usuarios. En las secciones [5.4] y [5.5] se detalla cómo solucionar este problema.

[39] CRUD: Create, Read, Update, Delete.

5.4 Mejoras en escalabilidad

Para lograr atender a más usuarios, la plataforma debe considerar los siguientes puntos:

1. Separar la LandingPage reservo.cl, de la aplicación.
2. Contar con una línea basal de funcionamiento, involucrando múltiples máquinas virtuales de forme permanente.
3. Soportar para el crecimiento repentino de las solicitudes de atención.

Sobre el punto (1), y para los efectos de página corporativa, SC3 vio que les conviene más tener una pequeña aplicación Web con información, testimonios y formulario de contacto. Esto debido a que el formulario de contacto está integrado con la solución CRM *Pipedrive* que utiliza internamente SC3 para gestionar posibles ventas futuras. Además, por un tema de seguridad se debe separar la página de contacto de la aplicación '*Reservo*', en hosts o máquinas diferentes. Así un intento de denegación de servicio a la página corporativa no inutilizaría la aplicación.

La importancia que se dé a los puntos (2) y (3) depende de la naturaleza de la aplicación. Dado que 'Reservo' es un servicio que requiere autenticarse, no se requiere incorporar en el sitio del servicio (en forma estructural), todo lo requerido para soportar cantidades inesperadas de visitas. Sin embargo, la página corporativa sí podría requerir elementos para ello.

En el diseño nuevo y en la propuesta de producción, se utilizan algunas de las técnicas que abordan (3) en la plataforma. Esto ayuda a optimizar el funcionamiento de la máquina virtual (*nodo),* en la plataforma que hasta ahora funciona el sistema. Estas mejoras permiten atender a una mayor cantidad de usuarios, en forma más eficiente, y de acuerdo a la experiencia reportada por las grandes compañías que han abordado este escenario de operación.

5.4.1 Base de funcionamiento

En este apartado se indican las mejoras en escalabilidad que deben efectuarse para atender más usuarios, si estos llegaran en forma permanente. Se contempla crear máquinas virtuales adicionales, lo cual se puede efectuar en forma manual, hasta llegar a unos 2 mil *Clientes*. En ese momento habría suficientes usuarios autenticados concurrentes y creciendo, como para pensar en automatizar el proceso de creación.

En la actualidad la plataforma cuenta con una pequeña aplicación Django para generar los dashboards que ven los clientes de *Reservo*. A estos dashboards sólo puede acceder un usuario desde el servicio, haciendo click en el botón que apunta a dashboard.reservo.cl. Esta aplicación está bajo el alero del servicio WAF, para prevenir DDoS.

Por otra parte, las mejoras necesarias para la base de funcionamiento son las siguientes:

- *Landing page aparte de la aplicación*. La plataforma que soporte la aplicación no tendrá la carga directa de acceso desde Internet, por lo tanto ésta podría mantenerse en dominio *app.reservo.cl* (tentativo, para efectos de este trabajo).

- *Balanceo de peticiones HTTP*. Esto se logra en la plataforma de Amazon mediante el servicio *Elastic Load Balancer* (*ELB*), el cual provee:
    - Alta disponibilidad.
    - Elasticidad: aumenta y disminuye su capacidad de procesamiento en base a lo que se necesita.
    - Cifrado SSL: permite que los usuarios puedan ver las páginas https, sin tener que ser servidas directamente desde Nginx, liberando así capacidad de procesamiento.
    - Al tener separada la *landing page* de la aplicación, y usando varias máquinas virtuales para procesar las vistas, es posible realizar actualizaciones sin tiempo y sin servicio apreciable para usuarios. Para esto se debe:
        (a) Tomar dos VMs y removerlas del balanceador *app.reservo.cl*.
        (b) En otro balanceador, de uso interno para SC3 (por ejemplo *stage.reservo.cl)*, se deben poner las máquinas virtuales removidas, e instalar la nueva versión.
        (c) Efectuar todas las pruebas necesarias contra stage.reservo.cl, y en caso de que no cumpla con todo, volver a versión anterior y poner las MV en el balanceador. Si cumple con todo, entonces se debe sacar las VM de stage y, en el balanceador de producción, remover las otras VMs y agregarlas al software nuevo.

El procedimiento anteriormente detallado, con algunas modificaciones como tomar una *snapshot*[40] de disco duro, sirve para cuando se debe instalar actualizaciones de seguridad en sistema operativo de las VM que requieren reinicio de ellas. Para esto no se requiere cambiar la aplicación de '*Reservo*', y además permite probar el funcionamiento correcto de las VM después de actualizar. En general éstas funcionan siempre en una versión de Ubuntu Linux LTS (que no tiene cambios incompatibles), pero que en una cultura de desarrollo seguro y consistente, éstas deben ser probadas antes de darles el alta.

Los cambios que alteren las tablas de base de datos, deben tener algún cuidado adicional en general, para que los usuarios no perciban la pérdida del servicio. La replicación y el balanceo del acceso a la BD es uno de las soluciones más utilizadas en la industria para abordar este tema. Para efectos de Reservo, conviene contar con un esquema del tipo "*One Master, Several Slaves*", en el que los datos que se escriben en el *Master*, son replicados al poco tiempo en el *Slave* (Figura 32). Esto cumple con los objetivos de mantener un respaldo operativo en línea, pero para aprovecharlo se debe balancear apropiadamente. Por ejemplo, las peticiones que pidan alterar la base de datos se deben dirigir al *Master*, mientras que las de lectura pueden ser enviadas a alguno de los *Slaves*.

[40] Instantánea, para después volver a ese punto de datos en tiempo.

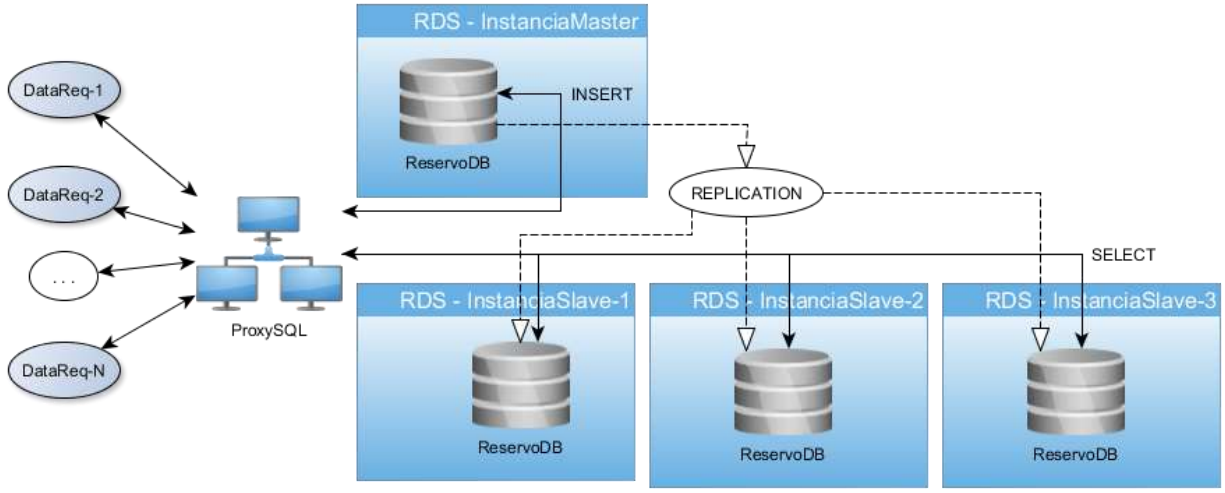

Figura 32: Replicación y balance de carga con Bases de datos

El esquema exacto de replicación que sería usado en 'Reservo' depende de *AWS* (o del proveedor *cloud*), en el caso de si provee o no replicación como funcionalidad. *Amazon Web Services*, por ejemplo, permite crear *réplicas de lectura* de una instancia que maneja la sincronización entre *Master* y los *Slaves* de forma automática, entregando un *endpoint* de conexión diferente para cada esclavo. Es deber de SC3 el implementar alguna forma de diferenciar qué consultas se van al *Master* y cuáles al *Slave*.

El servicio *ProxySQL* debe estar en una máquina virtual dedicada, que resuelva por nombre *DNS* interno. Poner la VM en la misma máquina de un nodo *Nginx+uWSGI* no hace sentido si se desea aumentar la seguridad de la plataforma, pues esto agregaría un paso más de conexión de red. El contar con el servicio *ProxySQL* adicionalmente tiene la ventaja de proveer caché de base de datos, por lo que se agrega un punto de mejora al desempeño de la plataforma ante consultas repetidas.

Esto soluciona casi todos los detalles de escalabilidad y uptime que necesitaría reservo en cuanto al acceso a bases de datos relacionales. A niveles mayores de usuarios, donde se requiera un uso intensivo de base de datos, se podría usar un protocolo de clúster distribuido. Esto usualmente se da en soluciones de nivel empresarial.

Respecto al tema de disponibilidad de la plataforma, como se mencionó antes, se propone la migración de más funcionalidades a microservicios. Adicionalmente habría que migrar parte de las funcionalidades invisibles a usuarios, listadas en la sección [3.4.4] y que no fueron consideradas en el punto [5.2.3]. Además, habría que considerar eventualmente otras que no hayan sido identificadas en este documento, pues el objetivo de esta memoria no es documentar todas las funcionalidades existentes en *Reservo.*

5.4.2 Soporte para el aumento inesperado de visitas legítimas

En este apartado se mencionan los elementos que sirven para responder al caso de muchas visitas en un momento inesperado. La forma ideal de enfrentar esto a nivel de procesamiento,

contempla gran parte de las funcionalidades implementadas con microservicios *serverless*, que cuentan con escalabilidad y alta disponibilidad. La capa de datos está pensada en servicios, como *DynamoDB*, que provisiona capacidad de cómputo en la medida de lo requerido. Se paga el desempeño de operaciones por unidad de tiempo, y el pago de espacio es en la medida que se utiliza.

'*Reservo*' no necesita eso en forma directa en esta etapa, pero si llegase a superar los 10 mil clientes a nivel mundial, este tipo de provisión de servicio es algo a considerar. Sin embargo, hay que destacar que las empresas que requieren de estas soluciones usualmente tienen aplicaciones móviles que funcionan con una API. Esto permite tener compartimentalizado el procesamiento en un back-end único, y trabajar en la solución específica por separado. De momento, ese no es el enfoque que sigue '*Reservo*'.

Muchas de estas medidas están probadas en la práctica, por lo que se comprueba la efectividad. Un caso reciente de esto fue un sitio que entregaba información sobre el huracán Harvey que afectó a *Houston – Texas* a mediados del 2017. El caso está descrito en un artículo en ArsTechnica[41]: "*How to hurricane-proof a web server*". Básicamente la propuesta pasa por utilizar contenido estático (scripts, imágenes).

Considerando lo anterior, se sugieren algunas mejoras que deben ser evaluadas por SC3:

1. Que la landing page de *Reservo*, en vez de ser una aplicación, sea sólo contenido estático. Se propone que sea susceptible de ser servido mediante S3 con algún tipo de mitigación por *Web-Application-Firewall*, para que un intento de *DDoS*[42] no termine con una cuenta muy alta por concepto de transferencia de datos hacia internet.

2. Uso de una CDN (*Content Distribution Network*), como *Cloudflare* o *Amazon Cloudfront*, para que los archivos sean servidos desde las locaciones más cercanas a usuarios. Por ejemplo, sitios para clientes chilenos alojados en E.E.U.U. al usar *Cloudflare*.

3. Caché en los siguientes puntos: *Varnish* entre *ELB* y *Nginx* para páginas frecuentemente generadas.

5.5 Mejoras en el uptime

Actualmente se cuenta con un DNS Público implementado en dos proveedores diferentes, replicado en forma manual. Este esquema de trabajo es aún manejable dado que la plataforma tiene pocos elementos, pero en un futuro deben evaluarse alternativas para mejorar este aspecto. Entre las mejoras necesarias están las siguientes:

1. Crear una zona DNS, privada e interna para el uso de *Reservo*, utilizando *Route53* (o el servicio DNS interno del proveedor *cloud*). Esto permite aprovechar los beneficios de

[41] https://arstechnica.com/information-technology/2017/09/how-to-hurricane-proof-a-web-server/
[42] *DDoS: Distributed Denial of Service*, es un ataque orientado a evitar que se pueda entregar el servicio, teniendo demasiados orígenes de petición, de diferentes partes en un intervalo de tiempo muy corto.

contar con *round-robin resolution*, en caso de tener dos direcciones IP diferentes, en máquinas diferentes para un mismo servicio.

2. Las máquinas de procesamiento administradas en forma propia, que tengan un *resolver DNS* local, con *caching* de una hora si es que se empieza a utilizar resolución por DNS interno.

A continuación se presentan detalles adicionales relativos a H.A. (*High-Availability*, o Alta Disponibilidad) de la solución propuesta:

- El servicio *ELB* cuenta con alta disponibilidad garantizada por parte de Amazon.
- En la sección [4.7] se propuso considerar el uso del servicio *WAF*, con el que se puede evitar un exceso de peticiones, o derechamente bloquear ciertos países en base a geolocalización. Esto permitiría detener ataques desde China, Rusia, Ucrania, y Sudeste Asiático en general. Otras funcionalidades de este servicio son permitir el paso de cierto tipo de peticiones específicas y nada más. El servicio cuenta con alta disponibilidad y escalabilidad por construcción.
- Para configurar una base de datos residente en AWS con alta disponibilidad[43], al usar el esquema planteado en [5.4.1], se tiene una parte del problema solucionado. Sin embargo queda pendiente lo siguiente:
    - Garantizar la inserción de elementos en caso de que la instancia *Master* quede inaccesible temporalmente, lo cual puede ser logrado mediante el siguiente procedimiento:
        - Al crear la instancia master, marcar Multi-AZ. Eso crea una réplica *standby* que se activará en forma transparente para la aplicación si se dan las condiciones de *failover*, pero puede demorar unos 45 segundos en ser efectivo.
    - Alta Disponibilidad del componente *ProxySQL*. Esto es solucionable mediante el uso de dos máquinas virtuales corriendo el programa, con los mismos ajustes. Estas VMs deben estar configuradas en modo activo-pasivo para usar una dirección IP virtual[44], utilizando *keepalived* y el servicio de *Amazon ENI*[45].

En el transcurso de lo estudiado, utilizando *MySQL* en servicio *RDS*, no existe forma de implementar la tecnología *MySQL Fabric*[46]. A fines de noviembre 2017 se invitó a una *preview*[47] de la tecnología *Multi-Master* de *Amazon Aurora* (*RDBMS* de reemplazo de *MySQL*, pero implementado y mantenido por Amazon), por lo que pasará un tiempo antes de que esté disponible en *MySQL*, si es que lo está alguna vez. La única forma de paliar eso sería no usar *RDS*, y usar

---

[43] https://severalnines.com/blog/mysql-cloud-pros-and-cons-amazon-rds
[44] Tutorial how-to: https://twindb.com/setup-high-availability-for-proxysql-via-keepalived-in-aws/
[45] ENI: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-eni.html#scenarios-enis
[46] Esquema multi-master de MySQL, como una grilla. https://gbsys.com/2015/09/mysql-fabric-simplifical-la-alta-disponibilidad/
[47] https://forums.aws.amazon.com/ann.jspa?annID=5220

máquinas virtuales en *EC2* y administrar uno las bases de datos, con los temas de tiempo que ello implica.

5.6 Mejoras al proceso de desarrollo

Parte de lo requerido para este punto, impacta también en el siguiente, referido a Seguridad. La cantidad de personas necesarias en el equipo se estima en dos personas, como mucho tres a tiempo completo, junto a alguien que efectúe revisiones manuales de seguridad informática, una vez que un '*build*'[48] haya pasado la compilación automática. A nivel de personal de desarrollo, para poder trabajar en la plataforma de cara al futuro, se requiere que:

1. Las personas que quieran trabajar en el desarrollo tengan conocimientos básicos de seguridad informática aplicada al desarrollo. Por ejemplo, cómo desarrollar enfocado en evitar los errores mencionados en *Top 10 OWASP 2013* (o la versión que siga en su momento). Por lo tanto, se deberá invertir en capacitación si no cuentan con esos conocimientos.

2. Los desarrolladores deben tener nociones de qué se puede lograr con ciertas vulnerabilidades explotadas.

3. Equipo de desarrollo debe realizar revisiones de código. Éstas sirven para ponerse al día, para que todos tengan una noción de las componentes que no trabajaron, e incluso para mejorar un componente ya existente.
4. Tener visibilidad del estado del desarrollo, mediante un tablero Kanban, es algo ampliamente usado en la industria.

5. Se debe pasar a contar con varios entornos: *Dev*, *Test*, *Staging*, *Prod*.

Por otra parte, es necesario adoptar las medidas mencionadas en la sección 4.6. Por ejemplo, usar *Test Driven Development* e *Integración Continua* es a estas alturas, casi un estándar en la industria. Ambas prácticas han mostrado que su uso permite lograr un flujo de desarrollo más eficiente. Se sugiere también el uso de complementos, por ejemplo:

- A nivel de IDE, existen herramientas (como *SonarLint*) que permiten detectar patrones de programación redundante, o duplicidad de código en el momento de edición.
- A nivel de integración continua, se sugiere usar herramientas de medición de calidad de código y cobertura de pruebas, por ejemplo, usar *SonarQube*, junto al plugin "*OWASP Dependency Check*". Con esto mide la calidad en forma automatizada, junto con verificar el no tener dependencias con vulnerabilidades severas.

Las medidas de automatización de compilación y pruebas son extensivamente utilizadas en la industria, y de acuerdo a quienes las han usado, no volverían a un esquema sin estructura definida ni pruebas automatizadas. Esto se debe fundamentalmente a los ahorros de tiempo logrados, involucrando un menor tiempo de implementación de correcciones y nuevas características.

[48] Compilación o construcción del software, para después efectuar pruebas de funcionamiento.

5.7 Mejoras a la seguridad informática

Este es un servicio que apunta a ser parte esencial y crítica del negocio, por lo tanto debe ser diseñado con esquemas seguros, e incluirse mitigaciones y contramedidas. Incluir la seguridad en el proceso de construcción de la aplicación y en la cultura de quienes desarrollan, permitirá que la empresa se sostenga en el tiempo, dado que éste es su servicio estrella.

Los diversos incidentes de seguridad que se han hecho públicos han terminado con pérdida de valor real, credibilidad, multas, investigaciones y demandas en muchas empresas. Por lo tanto, tomar estas medidas en la etapa justo antes de lanzarse a crecer, permitirá dar continuidad al producto '*Reservo*', y atender cada vez a más clientes.

Como fue mencionado en la sección [5.6], lo primero es que los que vayan a desarrollar, deben ser capacitados en desarrollo seguro, conceptos de seguridad informática, y conocimiento de vulnerabilidades. Se debe incluir en la cultura de desarrollo los *Top 10 OWASP Proactive Controls*[49]:

1. Verificar Seguridad en forma temprana y frecuente.
2. Parametrizar las consultas a bases de datos.
3. Codificar datos en tránsito a texto Base64, siempre.
4. Validar todas las entradas.
5. Implementar controles de identidad y autenticación.
6. Implementar adecuadamente los controles de acceso a datos.
7. Proteger los datos.
8. Implementar registros y detección de intrusiones.
9. Aprovechar a favor propio los *frameworks* y bibliotecas de Seguridad.
10. Manejo de errores y excepciones.

Parte de estos controles están ya abordados por el framework Django, pero otros deben ser implementados a través del accionar de las personas, herramientas y procesos. Después se deben establecer planes para migrar la plataforma actual a una versión de Django que tenga actualizaciones de seguridad, y los demás detalles mencionados en la sección [3.5.2]. Además, debe asegurarse de habilitar y que funcione la protección Anti-CSRF.

Es posible que se requiera una consultoría en cuanto a diseño e implementación de software seguro. En este sentido se sugiere fuertemente contar con un especialista certificado en *CSSLP* (*Certified Security Lifecycle Software Professional)*, o con alguna otra especialización en desarrollo seguro de aplicaciones. Esto es porque se necesita cambiar el desarrollo actual, a uno que esté orientado a construir seguridad en la aplicación desde el diseño mismo. Además, se deben abordar los puntos mencionados en el apartado [4.7], en especial el tener pruebas de seguridad automatizadas y después revisiones manuales en una plataforma de Staging.

[49] https://www.owasp.org/index.php/OWASP_Proactive_Controls

Se sugiere fuertemente utilizar el servicio de Amazon WAF (Web Application Firewall) en '*Reservo*', debido a la flexibilidad y capacidad de defensa que éste posee. En cuanto a esta característica, el firewall permitir sólo peticiones que cumplan con cierto tipo de patrón, así se deja pasar lo que consideramos válido, y se evitan muchos tipos de ataques. SC3 también debe definir un plan de continuidad operacional para este caso, el cual contemple desde las actualizaciones del sistema operativo, hasta la recuperación de la plataforma ante desastre mayor.

5.8 Evaluación de las mejoras propuestas

Antes de proponer las soluciones se investigó de cómo hacer aplicaciones web y/o móviles, que deben atender muchas solicitudes de usuarios, algunas con carga sostenida que aumenta o disminuye de a poco (como en el caso de Netflix, Spotify, Twitch), y otras con alta necesidad en instantes puntuales y en ocasiones con bajo nivel de trabajo (Salesforce, Facebook). En todos los casos estudiados las empresas dueñas de la solución llevaron sus propuestas a una arquitectura de Microservicios, buscado minimizar o desatender la necesidad de asignar o quitar recursos de procesamiento. Por lo tanto, las propuestas de mejora indicadas en este documento cuentan con respaldo documentado en otros escenarios de operación, que incluso son más exigentes que el de 'Reservo'.

Por otra parte, se consultó con varios egresados del DCC que tenían experiencia en el desarrollo de aplicaciones que debían operar en este tipo de escenario. Todos ellos indicaron que llevar la solución hacia un esquema de microservicios era una buena idea, sobre todo si hay costos variables que pueden ser bastante apreciables, los cuales pueden evitarse si se usa un provisionamiento manual de más máquinas virtuales y recursos para algunas de las funciones. Para ello se debe categorizar las vistas de la aplicación según lo que se requiera. Llevado al caso de '*Reservo*', esto implica considerar:

- Despliegue rápido del front-end de la aplicación (página Web), considerando una potencialmente alta cantidad de visitas directas por contacto. Aislar la página Web en servicio *S3*, como se propone en la sección [5.4], garantiza cumplir con ello. Si se desea bajar aún más los costos, habría que estudiar la factibilidad de migrarla a *BackBlaze*[50].

- Algunas funcionalidades de la plataforma requieren procesamiento intensivo, unas a petición de usuario y otras invisibles. Con la combinación de servicios escalables y altamente disponibles adecuada, se obtienen beneficios evidentes que se financian mediante el atender una mayor cantidad de clientes en un *nodo*. El nuevo servicio para acceder a los datos requerirá una revisión de las métricas de NewRelic, en cuanto a escrituras diarias en la BD y a costos en base a ellas. En un nivel con menos de diez mil *clientes*, utilizar *Lambda* y *API Gateway* debiese resultar en una solución que es menor complejidad de implementar y mantener.

[50] www.backblaze.com es un servicio que provee lo mismo de S3 a un costo menor (además de un servicio de respaldo de datos ilimitado por USD$5 mensuales), pues se lanzaron antes que Amazon S3, y optimizaron su funcionamiento al punto que le llevan mucha ventaja en los temas de almacenamiento y transferencia de datos, siempre manteniéndose por delante de Amazon, rechazando incluso ofertas de compra multimillonarias.

Sobre los ciclos de desarrollo, todas las empresas de servicio de nivel mundial reconocido, como *Salesforce* o *Netflix*, comentan en sus blogs técnicos (por parte de sus expertos) que el uso de TDD e Integración Continua se da por asumido. Con respecto a la seguridad, este es un tema que cuidan delicadamente. Por ejemplo, Netflix busca prevenir accesos directos a material con derechos de autor en muy alta calidad.

Capítulo 6: Plan de trabajo propuesto para implementar mejoras

El plan de trabajo que se describe en este capítulo tiene tres objetivos, los cuales se describen a continuación:

- Seguir un camino hacia lograr un mayor uso de microservicios, que tenga logros apreciables en el corto plazo.
- Corregir los problemas de SC3 que en la actualidad son una piedra de tope, para rectificar la falta de escalabilidad y la poca resiliencia.
- Seguir trabajando con ese enfoque cultural, para que la aplicación nunca se vea en riesgo de quedar obsoleta por infraestructura o componentes de software, y así evitar potenciales vulnerabilidades que puedan comprometer la existencia de '*Reservo*'.

Existen actividades que pueden ser implementadas en relativamente poco tiempo, inferior a dos meses, pero en otros casos se requiere un tiempo de trabajo mayor por parte de SC3. Para llevar a cabo esto, se sugieren cinco etapas. Las dos primeras están enfocadas a mejorar ciertos problemas de seguridad y escalabilidad más bien técnicos, que no necesitarían ser rehechas completamente en el futuro. La tercera etapa implica más tiempo, pues sería capacitación e implantación de nueva cultura de desarrollo, y la migración de base de código de 'Reservo'. La cuarta etapa considera temas técnicos, que deben ser abordados después de tener conocimientos de desarrollo seguro. Finalmente, la quinta etapa considera detalles técnicos que, si bien se pueden implementar antes, el valor agregado real se muestra cuando ya se tiene el producto listo para escalar, y con la cultura de desarrollo seguro asimilada en la empresa. Los tiempos para llevar a cabo estas etapas quedarán a criterio de SC3, debido a que algunos pueden requerir reorganización interna, personal adicional, y otros recursos.

En este momento, antes de iniciar el proceso, SC3 debe tener disponible el personal (tres personas full time) que será dedicado al proyecto a futuro. Es aconsejable que al menos uno de ellos sea uno de los miembros fundadores, o en su defecto, que tenga serios incentivos de permanencia a largo plazo. De esa forma, se minimizan los riesgos de '*Reservo*' en caso de rotación de personal.

6.1 Etapa I: Primeras mejoras

En este caso queda la base para efectuar implementaciones orientadas a mejorar la escalabilidad de la plataforma, y dar solución a varios problemas existentes. Como primera medida de seguridad de datos, el logserver debe ser una máquina virtual que funcione con alta disponibilidad, y que esté visible sólo internamente. En la Figura 33 se plantea que hay 5 tareas que pueden iniciarse simultáneamente sin interferirse entre ellas, y dos que dependen de la creación de una máquina virtual. Todas son necesarias para las etapas siguientes.

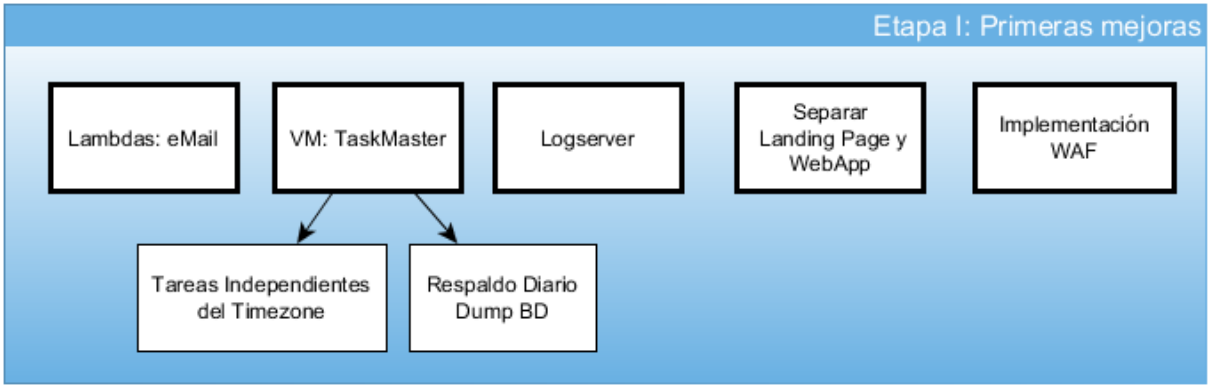

Figura 33: Mejoras consideradas en la Etapa 1

6.2 Etapa II: Mejora de la disponibilidad

En esta etapa se busca implementar los aspectos críticos para poder contar con alta disponibilidad y escalabilidad en la plataforma. En la Figura 34 se muestra que hay dos bloques de trabajo a abordar: Alta Disponibilidad en Base de Datos y en Atención a usuarios. Ambas son líneas aparte que pueden trabajarse en forma paralela.

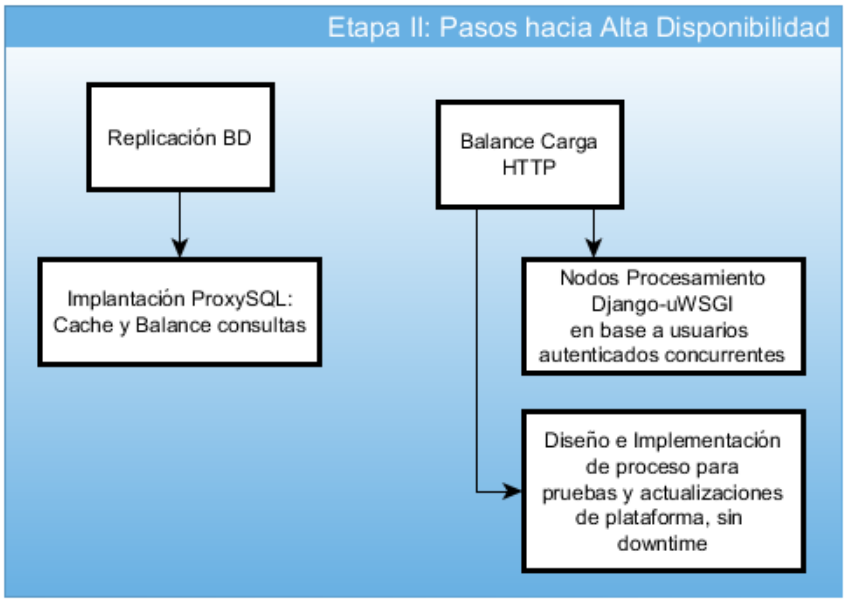

Figura 34: Mejoras consideradas en la Etapa II

6.3 Etapa III: Desarrollo seguro

Esta es la parte que tomará más tiempo, pues requerirá dejar el desarrollo activo de '*Reservo*' por un tiempo, tener el personal necesario dedicado y preparado para el desarrollo. Se debe contar también con capacidad de efectuar pruebas activas de seguridad informática, no contempladas en el ciclo de integración continua.

De acuerdo con la Figura 35, el objetivo es iniciar una migración a *Django* 1.11. Para ello se requiere completar las 4 actividades anteriores, siendo la consultoría la que tomaría más tiempo debido a que son actividades que requieren interiorización por parte del equipo. Ésta debe ser guiada por un profesional certificado en Seguridad Informática.

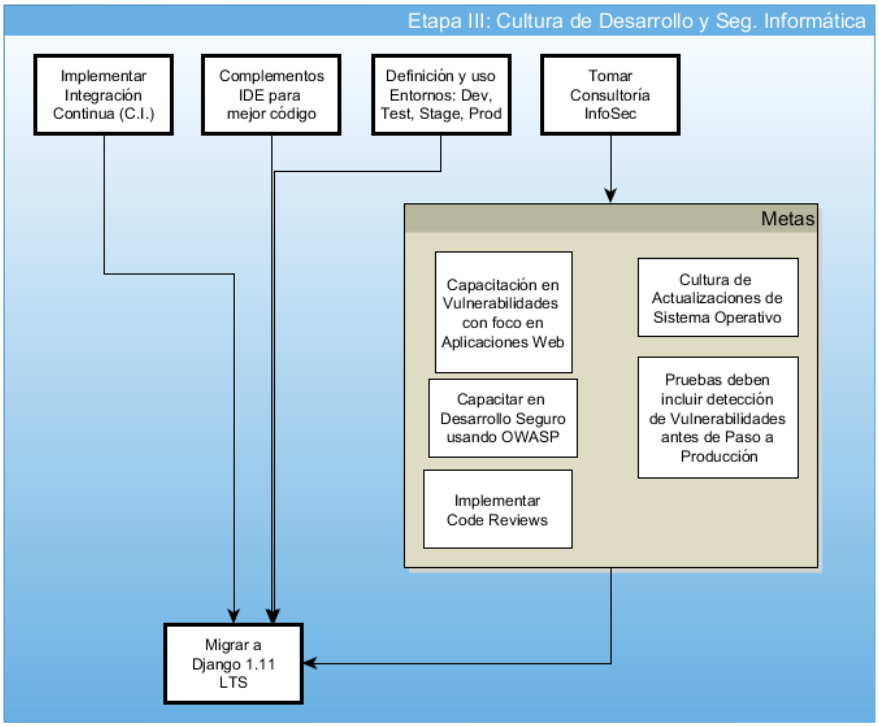

Figura 35: Mejoras consideradas en la Etapa III

Al final de esta etapa, cada vez que salga una versión de 'Reservo' al entorno de pruebas, una vez superadas las pruebas automatizadas, incluyendo detecciones de seguridad, es aconsejable efectuar pruebas de penetración de la aplicación, por parte de alguien que sepa de ello, idealmente que no haya trabajado directamente en el proyecto, pues en los pasos automatizados no es posible efectuar algunas verificaciones más avanzadas.

6.4 Etapa IV: Escalabilidad de la solución

Una vez que SC3 implante una cultura de desarrollo seguro al interior de la organización, se puede pensar en convertir la aplicación en Multi-Tenant, con nuevos clientes creados cada uno en su propia base de datos, y así sentar las bases para poner capa de seguridad adicional en acceso a datos.

En la Figura 36 se muestra, que se requiere tres niveles de trabajo, para en el cuarto contar con la base de datos convertida en *Multi-Tenant*, y quedar listo para implementar el servicio de acceso a datos que dará una mayor garantía de seguridad en el acceso a datos, lo cual es necesario

si se trabaja con fichas médicas de pacientes, incluyendo fotografías de ellos, que los centros médicos guardan como parte del proceso de diagnóstico y tratamiento.

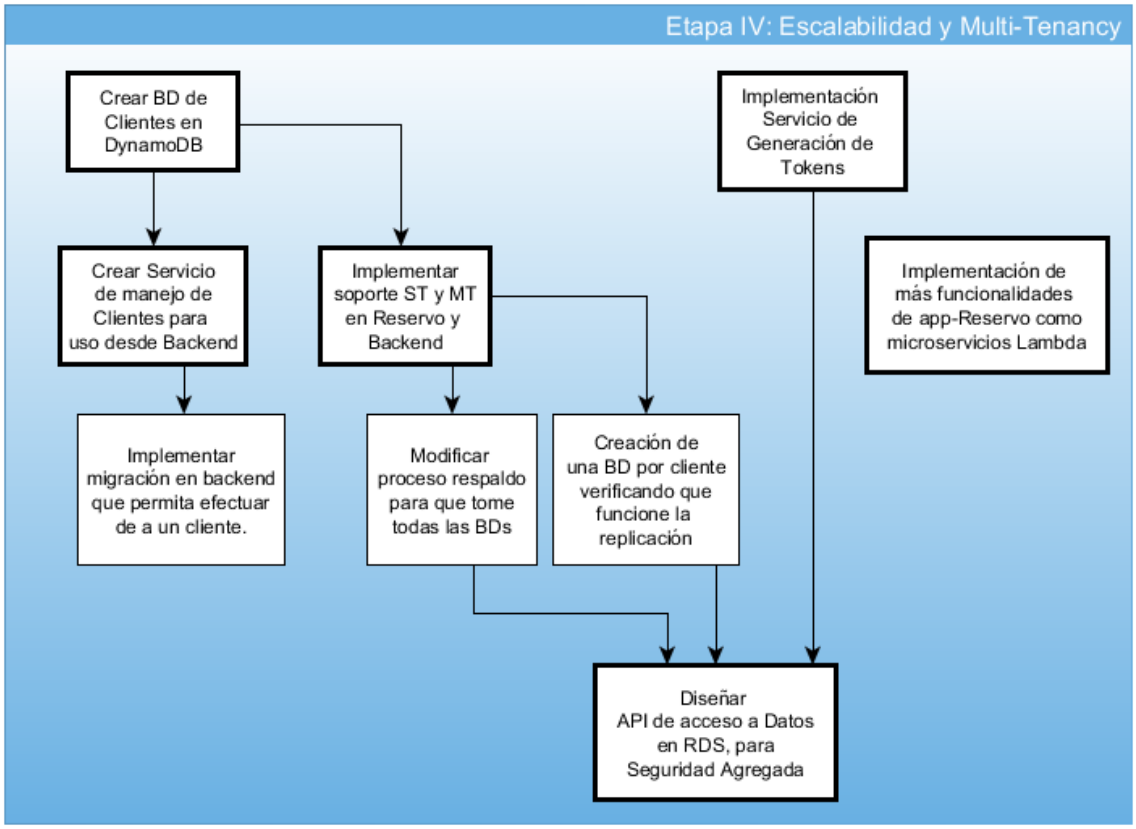

Figura 36: Mejoras consideradas en la Etapa IV

El servicio de generación de tokens debe estar implementado antes de crear el servicio de acceso a datos (Fig. 36). Como se menciona en [5.3.1], la idea es que la aplicación solicite un ticket, y el servicio de datos pueda verificar que el ticket es válido y no haya sido caducado por uso. Adicionalmente, dado que se incluye el convertir funcionalidades a microservicios, debe estudiarse cuáles conviene, debido a que hay que tener en mente los costos monetarios, comparado a los beneficios obtenidos.

6.5 Etapa V: Mejoras adicionales

La alta disponibilidad del servicio de *ProxySQL* es lo más crítico, ya que permite asegurar que toda la cadena interna de componentes de '*Reservo*' cuente con continuidad de servicio. Las otras mejoras son para casos futuros.

En la Figura 37, se muestran dos niveles, en el cual el primero y de mayor criticidad, es el de implementar Alta Disponibilidad para la componente *ProxySQL*.

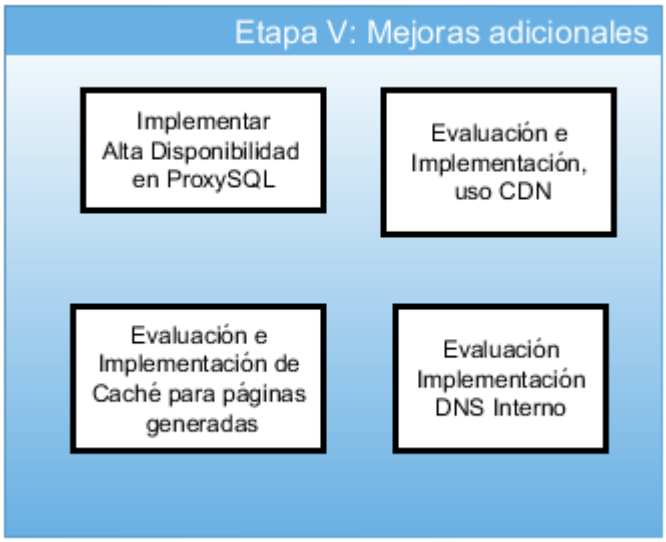

Figura 37: Mejoras consideradas en la Etapa V

Las otras, son mejoras que pueden tener un impacto positivo, pero tienen detalles importantes, son las siguientes:

- El uso de una *CDN* permite que el contenido estático de la aplicación sea obtenido de una localización geográfica muy cercana al usuario, usualmente dentro de la red del mismo país, mejorando la experiencia de uso y carga de la aplicación. Se debe probar muy bien en caso de actualizaciones, o cambios de punteros de DNS públicos, que es algo que en ocasiones hace aparecer errores inesperados por no estar sincronizada correctamente.

- Utilizar DNS interno permite mayor flexibilidad en cuanto al provisionamiento de *nodos* de proceso interno, pero aumenta un poco la complejidad de la administración.

- El uso de caché mejora la experiencia de usuario al cargar más rápido el contenido que ya está generado y no cambia mucho, pero a primera impresión, muchas de las vistas de '*Reservo*' tienen contenido generado en forma dinámica, el cual no necesariamente se repetiría.

Capítulo 7: Conclusiones y trabajo a futuro

La primera conclusión que es posible extraer de este trabajo de título, es que, si se va a desarrollar un servicio (aplicación Web) que funcionará a través de internet en la modalidad Cloud, lo más fácil de solucionar es todo aquello que tenga que ver con los temas técnicos. Sin embargo, lo realmente crítico para escalar adecuadamente la solución a un escenario global, es que la cultura de la organización esté completamente alineada con eso. Ella influirá en cómo serán abordados los problemas que surjan en el desarrollo permanente. Una cultura de desarrollo seguro, orientada a crear y mejorar continuamente un servicio de excelencia para el usuario, asegura mantenerse operando en el mundo actual. Si no lo hace uno, o se deja de hacer, lo seguro es que alguien más verá la oportunidad y la tomará. Amazon, por ejemplo, busca evitar esto manteniendo su cultura en el 'día uno'.

La organización requiere contar con suficiente personal, que además de ser competente en lo técnico, cuente con la capacidad de aprender cosas nuevas siempre. Este personal debe estar familiarizado con nociones de seguridad informática a nivel de desarrollo, buscando prevenir vulnerabilidades y solucionar a la brevedad las contingencias que puedan ocurrir en el entorno productivo. El personal debe también trabajar con procesos validados por la industria, como lo son el *Test Driven Development* y la integración continua de soluciones, debido a la utilidad que estas prácticas brindan a los desarrollos.

Fue interesante ver la importancia del manejo de zonas horarias para un 'Software como Servicio'. Al apuntar a una cantidad de usuarios 'apreciable', como los de países de habla hispana, el hecho de que estén en diferentes zonas horarias, requiere repensar muchos de los servicios automatizados. En el caso de '*Reservo*', los recordatorios enviados a las 08:00 son un ejemplo que debe estar situado de acuerdo con el horario local. Por otra parte, los respaldos de datos que también dependen de la hora, deben ser realizados sin afectar la experiencia del usuario.

En un servicio que funciona en una plataforma *cloud*, siempre habrá alguna forma de escalar, asegurar y dar alta disponibilidad dicha solución. Habrá también algunos costos asociados, ya sean monetarios o compensaciones (una cosa a cambio de otra). Por ejemplo, tener más garantía de disponibilidad de datos a cambio de una leve demora adicional en obtenerlos. A nivel técnico, dichas demoras son casi imperceptibles si consideramos las ventajas de contar con un entorno de red y centro de datos propio, que evita tener que hacer la inversión de infraestructura propia para un solo servicio.

A nivel de entendimiento de sistemas distribuidos, las plataformas Cloud permiten que los servicios crezcan sin preocuparse de provisionar electricidad, hardware y comunicaciones. *Amazon AWS* provee funcionalidades, y en base al desempeño (e incluso características adicionales que uno necesite) puede obtenerse capacidad de atención extra siempre y cuando se esté dispuesto a pagarlo. El hecho de que haya algunos de los servicios en los que se ofrece directamente alta disponibilidad y escalabilidad automática de la solución, mientras que en otros esto es opcional, ilustra la flexibilidad que tienen este tipo de infraestructuras para dar soporte a soluciones de terceros.

En ocasiones se puede dar que una actividad se efectúe dos o más veces debido a que los servicios, en forma interna, funcionan con el principio de *'At-least-once delivery'*. Esto garantiza

que llegue al menos un mensaje al destinatario, sin un control estricto de duplicidad, y no necesariamente en orden, y eso es algo que viene asumido en un sistema distribuido.

Sobre el uso de sistemas distribuidos como enfoque para el desarrollo en múltiples plataformas, el caso de los proveedores *cloud* utilizados por *Instagram* o *Netflix*, son un buen ejemplo de ello. Estos proveen un servicio automáticamente escalable y altamente disponible para que puedan definir una API REST, con la cual se contactarán las aplicaciones clientes de computador, teléfono, tablet, o incluso un sistema automatizado. Si se desea tener un software como servicio que compita "a nivel mundial", es necesario utilizar una plataforma cloud para reducir el riesgo y el nivel de desafíos a enfrentar en el camino.

En cuanto a resiliencia, los proveedores de *Infrastructure-as-a-Service*, entre ellos *Amazon AWS*, *Google Cloud Platform* y otros, tienen claridad acerca de que muchos servicios dependen totalmente de ellos. Por lo tanto, ellos apuntan directamente a proveer servicios de una calidad muy alta, incluso al punto de compensar los gastos, pasados ciertos niveles de falta de servicio. Con esto, es factible construir aplicaciones resilientes que soporten altos volúmenes de tráfico, con seguridad desde el diseño.

Desarrollar para plataformas cloud tiene un detalle, que si se aborda sin cuidado, es posible implementar algo que dependa exclusivamente de un solo proveedor. Para un trabajo futuro habría que ver si es posible crear alguna capa que abstraiga el desarrollo, haciéndolo menos dependiente del proveedor de la infraestructura, reduciendo así sus implicaciones.

Finalmente, se debe destacar el hecho de que se debe considerar seguridad informática desde la concepción misma de la solución. Durante este último año se ha visto casos con mucha publicidad, como el de Equifax, que por una sola vulnerabilidad que no fue atendida a tiempo, se le causó un daño importante a la empresa. Por lo tanto, es necesario siempre estar investigando y mejorando la plataforma del servicio ofrecido, ya que eso es lo que dará real garantía de seguridad de la solución.

Bibliografía

[1] Microservices Oriented Architecture. [en línea] <https://en.wikipedia.org/wiki/Microservices/ > [Última consulta: 2017-11-30].

[2] Software Multitenancy. [en línea] <https://www.vinta.com.br/blog/2016/understanding-database-multitenancy >, [Última consulta: 2017-11-30].

[3] Distributed Systems Principles and Paradigms, Tanenbaum et al. - 2nd Edition.

[4] Peter Mell and Timothy Grance (September 2011). The NIST Definition of Cloud Computing (Technical report). National Institute of Standards and Technology: U.S. Department of Commerce. doi:10.6028/NIST.SP.800-145. Special publication 800-145.

[5] Beginning Software Engineering, Rod Stephens, 2015 - John Wiley & Sons.

[6] Martinfowler.com. Serverless Architectures. [en línea] <https://martinfowler.com/articles/serverless.html/ > [Última consulta: 2017-11-30].

[7] Blueprints for High Availability, Evan Marcus, Hal Stern - John Wiley and Sons, 2nd Edition, 2003.

[8] Scalability. [en línea] <https://en.wikipedia.org/wiki/Scalability/ > [Última consulta: 2017-11-30].

[9] Scrum – Software Development. [en línea] <https://en.wikipedia.org/wiki/Scrum_(software_development) > [Última consulta: 2017-11-30].

[10] Agile Software Development. [en línea] <https://en.wikipedia.org/wiki/Agile_software_development/ > [Consulta: 2017-07-30].

[11] Lean Software Development. [en línea] <https://en.wikipedia.org/wiki/Lean_software_development/ > [Última consulta: 2017-11-30].

[12] CRUZAT H., Sebastián G., 2016. *Plan de Negocio para un servicio online de reserva y gestión de clientes: caso Reservo.cl*. Memoria para optar a título de Ing. Civil Industrial, Universidad de Chile. 122p. [en línea] <http://repositorio.uchile.cl/handle/2250/140292/ >. [Última consulta: 2017-11-30].

[13] Celery Website: http://www.celeryproject.org/ [Última consulta: 2017-11-30].

[14] Scaledrone Website: https://www.scaledrone.com/docs/ [Última consulta: 2017-11-30].

[15] “Amazon AWS documentation”, https://aws.amazon.com/documentation/ [Última consulta: 2017-11-30].

[16] “Amazon DynamoDB”, https://aws.amazon.com/documentation/dynamodb/ [Última consulta: 2017-11-30].

[17] “Amazon EC2”, https://aws.amazon.com/documentation/ec2/ [Última consulta: 2017-11-30].

[18] “Amazon SNS”, https://aws.amazon.com/documentation/sns/ [Última consulta: 2017-11-30].

[19] “Amazon SES”, https://aws.amazon.com/documentation/ses/[Última consulta: 2017-11-30].

[20] “Amazon SQS”, https://aws.amazon.com/documentation/sqs/ [Última consulta: 2017-11-30].

[21] “Amazon S3”, https://aws.amazon.com/documentation/s3/[Última consulta: 2017-11-30].

[22] “Amazon Route53”, https://aws.amazon.com/documentation/route53/ [Última consulta: 2017-11-30].

[23] Google Cloud DNS, https://cloud.google.com/dns/docs/ [Última consulta: 2017-11-30].

[24] EC2 Instance Types: https://aws.amazon.com/ec2/instance-types/[Última consulta: 2017-11-30].

[25] “AWS RDS Provisioned IOPS really worth it?”, https://stackoverflow.com/questions/18777070/aws-rds-provisioned-iops-really-worth-it [Última consulta: 2017-11-30].

[26] Redis, https://www.redis.io [Última consulta: 2017-11-30].

Anexo A

A.1 Resumen de estudio de escalabilidad

Después de este trabajo, la conclusión es que la escalabilidad del Servicio de '*Reservo*' puede aumentar considerablemente, siempre y cuando se realicen los cambios en cuanto a procesos y prácticas de desarrollo, y se adopte la cultura de '*Build Security In'*. A nivel técnico es factible de escalar, pero se deben realizar muchos cambios y adoptar herramientas de apoyo a la fase de construcción de un Prototipo Mínimo Viable. Además, se requiere especializar al personal que desarrollará la aplicación, introduciéndole nociones de Seguridad Informática.

En cuanto a diseño y desarrollo de la aplicación, si se observa que puede ser escalable a muchos clientes y usuarios, conviene crear el modelo de datos desde un principio como *Multi-tenant*, para evitar tener temas de migración de datos en el futuro, las cuales pueden ser muy complejas, e incluso propensas a fallas. Si bien hace más complejo tareas como un respaldo, al escalar, pasa a ser una decisión acertada.

Los puntos anteriores son válidos para cualquier emprendimiento que apunte a proveer un software como servicio (SaaS), con una aplicación pensada en ser algo esencial para sus clientes. En estos casos se requiere una Arquitectura Escalable, con elementos de Alta Disponibilidad, y Segura desde la concepción. Por otra parte, llevar a cabo la construcción con elementos de este tipo, ayuda a minimizar la deuda técnica con la que se llega cuando se desea escalar. Dependiendo del rubro, una sola falla crítica de seguridad puede destruir lo avanzado en un emprendimiento, o dañar muy gravemente el prestigio de la empresa dueña del servicio, obligando incluso a operar con pérdidas si se desea recuperar parte del mercado perdido, si es que fuese posible.

Al ofrecer un servicio SaaS en el mercado actual, una buena forma de diferenciarse (además de las características visibles hacia los clientes), es haciendo que en el proceso interno se tenga una cultura de trabajo orientada a construir un producto de calidad; esto usualmente redunda en poder brindar un servicio de calidad. Para ello se debe de estar consciente que la competencia pasa a ser a nivel mundial (es decir, actores del escenario global), y los ejemplos a seguir son como los que muestran los servicios de software exitosos que operan a nivel mundial: Netflix, Spotify, Salesforce, entre otros.

Un servicio que no se tome la Seguridad Informática en serio, muy probablemente será infiltrado y/o destruido, con daño severo a su reputación, y consecuencias legales serias. Dentro del año 2017, los casos más recientes de Uber, y Equifax son muestras reales de impacto de ataques externos. Durante el tiempo de investigación en '*Reservo*', se observó que hay casos en los que se maneja información sensible y privada de *Pacientes* por parte de los *Clientes*. Si bien en organismos del estado[51] se ha visto mal manejo de datos privados y personales, en particular en el sector salud, para garantizar la continuidad de servicio y empresa, se debe tomar la seguridad como parte esencial de la manipulación de la información.

[51] http://www.24horas.cl/nacional/minsal-responde-a-error-informatico-por-filtracion-de-datos-de-miles-de-pacientes-1952173

A.2 Trabajo futuro para SC3 con ‘Reservo’

Queda pendiente implementar completamente los siguientes requisitos de sistema.

- **RS1**: Sistema debe considerar la zona horaria del *Cliente* para toda la interfaz en lo referente a gestión, y lo referente a tareas automatizadas.

- **RS2**: Sistema debe considerar el caso de que un país cuente con múltiples zonas horarias, para mostrar la información al *Cliente*.

- **RS3**: Mejorar la capacidad de atención de transacciones de clientes, haciendo que el desempeño de la plataforma sea igual o mejor a lo actualmente existente.

- **RS4**: Se requiere mantener la trazabilidad de las acciones de los usuarios, con el sistema de registro que muestra uWSGI.

- **RS5**: Se requiere una mejora que permita asegurar que los datos asociados a un *Cliente*, sean solamente visibles por usuarios que él defina en su perfil de administrador.

Por diversas razones (de tiempo, económicas, operativas y de prioridades) que escapan al autor de esta memoria, a la fecha de cierre del Semestre de Primavera 2017 quedan pendientes las siguientes actividades:

- Poner en el entorno productivo los *Lambda* para envíos de correos.

- Iniciar la implementación de máquina virtual *Taskmaster*, con sus procesos de respaldo de datos.

- Terminar de abordar temas de seguridad informática, mejorar de procesos de desarrollo y la cultura de la empresa para con su producto.

- Evaluar (e implementar si es viable) la creación de una API en Django, que independice las llamadas a microservicios dependientes de un proveedor de servicios *cloud*.

- Migrar algunas funcionalidades a microservicios, una vez implantadas las nuevas prácticas de desarrollo.

- Considerar caso de borde de *Cliente* con sucursales en ciudades que se encuentren en zonas horarias diferentes.

- Crear un servicio generador de tokens de uso único, dependiente del *Cliente* que esté en la aplicación y de otros factores. Debe contar con Alta Disponibilidad, Escalabilidad, y Seguridad para el acceso a los usuarios.

- Crear un servicio de acceso a datos, usando un token de un uso único como validación adicional. Debe contar con Alta Disponibilidad, Escalabilidad, y Seguridad en el acceso.

- Llevar la BD actual a una que tenga almacenamiento cifrado.
- Crear una zona privada *internal.reservo.cl* para el trabajo con máquinas virtuales.
- Implementar el Caché local DNS en máquina virtual, para mejorar resiliencia de la plataforma.

A.3 Revisión de costos aproximada de '*Reservo*' y su impacto

Los costos de una aplicación cloud tienen una estructura con base fija y componentes variables, dependiendo de los recursos utilizados. A continuación se enumeran detalles importantes a tener en cuenta:

- Las máquinas virtuales en *EC2*, tienen un costo fijo por hora de funcionamiento, y para calcular un costo mensual se aproxima por 720 (el proveedor envía en el *billing* el costo exacto). Mientras más recursos pueda ofrecer, más caro será el costo por hora.
- En cuanto a costos variables, existen servicios que cobran por transacción, y algunos incluso por nivel de concurrencia para desempeño. De estos, algunos pueden configurarse para tener una cota superior para no sobrepasar costos y otros pueden quedar completamente elásticos sin límite alguno; a continuación algunos ejemplos:
    - *DynamoDB* puede configurarse para funcionar con un máximo de operaciones por segundo.
    - *SES* puede configurarse para un nivel de cadencia máxima de correos por segundo que serán enviados, y un tope diario máximo.
    - *SNS* no tiene limitaciones para efectos prácticos, en cuanto a la cantidad de peticiones que se le pueda hacer.
    - *Lambda* cobra en base a una especie de 'taxímetro': una base de 300 milisegundos, con aumento cada 100 ms, multiplicado por un factor dependiente del consumo de memoria RAM, por intervalos. Acotado a un tiempo máximo de 5 minutos.
    - *WAF* y *ELB* cobran por cantidad de peticiones recibidas y crecen en forma elástica.

Dados los puntos anteriores, se puede concluir que para el caso visto en esta memoria, agregar temas de seguridad, escalabilidad y resiliencia utilizando los servicios debe efectuarse de forma inteligente, para evitar sorpresas desagradables a la hora de los pagos, dado que se entra a depender del proveedor. Adicionalmente, a la hora del desarrollo debe considerarse que las llamadas e interacciones entre microservicios deben estar optimizadas orientando a minimizar su uso, y con ello el costo variable mensual.

A.3.1 Estimación de costos nuevos

También se debe notar que una aplicación que da servicios a clientes por un pago mensual que es fijo para efectos prácticos (aunque se cobre diferenciado de acuerdo con el tamaño del cliente: cantidad de usuarios, boxes, etc.), mientras más se utilicen los microservicios en plataforma serverless, habrá un mayor costo variable mensual. Si eso no se controla de forma adecuada, la utilidad por cliente bajará mucho. A la fecha, el único servicio provisto, que toma algo con costos variables es el de envío de correos electrónicos por campaña dirigida; se hace mediante bolsas de correos electrónicos, y ese monto cubre con creces el costo de envío por cada correo.

La ventaja es que, usando estos servicios de costo variable, y sacando los servicios pesados como Redis y Celery de las instancias dedicadas al servicio de '*Reservo*':

- Actualmente '*Reservo*' funciona en una VM *m4.xlarge* (a futuro pasará a ser *m5.xlarge*), la cual con 4 vCPU, 16 GB en RAM soporta más que adecuadamente a los aproximadamente

1000 usuarios en los horarios laborales, los cuales no tienen un uso concurrente masivo constante, como por ejemplo *Facebook* o *Netflix*.

- Removiendo *Celery* de la máquina, sería posible llegar a atender 1500 o 2000 usuarios sin ningún problema.

Para lograr estimar los costos variables adicionales, de acuerdo con los precios de Amazon AWS, para los servicios *SNS*, *Lambda*, *DynamoDB*, *SQS*, se harán los siguientes supuestos, para 1000 usuarios.

1. 1 usuario hace 200 pageviews al día, con lo que se tendría una base de 200.000 peticiones web por día (esto para *ELB* y *WAF*). Más otras que lleguen por buscadores, o referencia a la página principal, como estimación, es posible sumar unas 50.000 más. En un mes, podríamos considerar que para los 1000 usuarios clientes hay aproximadamente 250 millones de peticiones al mes.
    a. Las estimaciones de costos en estos dos casos son bastante complejas, pero para abreviar se supondrá que hay una cota superior, y pensando en los 500 *clientes* actuales, costará alrededor de USD$60 por mes cada uno de los servicios.
    b. USD$120 sería el uso de *WAF* y *ELB*. El costo marginal de aumentar un cliente queda estimado en USD$0,1.

2. Siendo optimistas, que todos los clientes tendrán sus agendas llenas, con unas 15 citas por día y considerando un promedio de 3 profesionales/box por cliente, da un total de 45 citas-día para cada cliente. Esto se traduce en que para 500 clientes:
    a. Se podrían enviar cada mañana 22.500 correos electrónicos, y 22.500 durante el transcurso de cada día. El total da 45.000 e-mails diarios.
    b. Un total de 900.000 e-mails al mes.
    c. Cada cliente nuevo podemos contabilizarlo como 45 correos de confirmación adicionales por día y 45 correos de recordatorio diarios, 90 correos diarios, 1800 correos adicionales por mes: aumentar 1 cliente en cuanto a costos suma alrededor de USD$0,2.
    d. Descontando el free-tier, Amazon por mes facturaría alrededor de 850.000 e-mails, convertido a costos sería aproximadamente USD$85 al mes.

3. Para los efectos de *SNS* dado que se usa para gatillar lanzamiento de *lambda*, de acuerdo a la documentación de precios de *AWS*, no representa costo alguno para SC3.

4. Uso de SQS dentro de la misma región en Amazon, no tiene costo.

5. Los costos asociados a *lambda* son algo más complejos de estimar, debido a cuantas veces. Para efectos prácticos consideraremos que se consumirán como tope 256 MB de memoria. Que el tiempo de ejecución será de 1 minuto, y que se ejecutará 1 vez cada 5 minutos, lo cual da 12 veces por hora. Usaremos esto para el conteo de correos de confirmación.
    a. Sólo se lanzará una vez por plataforma.
    b. Podemos considerar las mismas cantidades para el caso de los correos de recordatorio.
    c. En un mes, la función lambda se lanzaría 8640 veces.

d. Para esos parámetros, la calculadora de costos provista por *Amazon*, indica que sería gratuito[52]. Contando dos veces, para 17500 lanzamientos, sigue siendo 0 el costo.
e. Recién estimando en 60 mil lanzamientos de lambda por mes, nos acercaríamos a USD$10, con un costo despreciable de aumento en base a la cantidad de clientes; sería apreciable solo cada 100 clientes en ese caso, si es que.

6. Respecto al uso de *DynamoDB*, el tema son los costos variables, y como solo se utiliza para registro, podría ponerse un coste de USD$20, para estimar y redondear. El esquema de cálculo depende mucho de los recursos efectivamente usados.

Resumen de costos mensuales variables:

Actual:

- E-mail: USD$85; USD$0,2 adicionales por cada cliente nuevo.

Nuevos:

- Uso de *ELB* y *WAF*: USD$120, y USD$0,1 adicionales por cada cliente nuevo
- Uso de *DynamoDB*: USD$20.

Beneficios:

- El uso de estos servicios, con el software diseñado para hacer un uso eficiente de estos recursos, simplemente aumenta la cantidad de usuarios que es posible atender en una máquina virtual *EC2* digamos en al menos 50%, sin perder desempeño alguno.

A.3.2 Reducción de costos actuales

A nivel de estrategia de reducción de costo mensual, a finales del 2017 SC3 utilizó instancias reservadas (*reserved instances*) de Amazon: Esto permite un menor costo mensual, pero un compromiso irrevocable de pago de estas máquinas. En la figura siguiente, se aprecia que pagar la mitad del costo anual reservado, disminuye el pago mensual mucho; los descuentos son más apreciables al reservar 3 años[53].

[52] https://s3.amazonaws.com/lambda-tools/pricing-calculator.html
[53] https://aws.amazon.com/ec2/pricing/reserved-instances/pricing/

m4.xlarge

STANDARD 1-YEAR TERM					
Payment Option	Upfront	Monthly*	Effective Hourly**	Savings over On-Demand	On-Demand Hourly
No Upfront	$0	$90.45	$0.124	38%	$0.2 per Hour
Partial Upfront	$517	$43.07	$0.118	41%	
All Upfront	$1013	$0	$0.116	42%	

Figura 38: Niveles de descuento para 1 año en instancias reservadas

Además, existen 5 clientes chilenos especiales, los cuales tienen entornos aparte completamente separados y dedicados, algunos con solicitud de desarrollos específicos a medida, los cuales fueron cotizados y cobrados acorde a ello. Utilizan máquinas específicas, las cuales se financian solas.

A nivel de costos, en los últimos 3 meses del año 2017, las facturas de uso de Amazon fueron aproximadamente:

- Octubre 2017: USD$1500
- Noviembre 2017: USD$700
- Diciembre 2017: USD$650

El uso de instancias reservadas se sugiere fuertemente, para lograr una reducción apreciable de los costos en cuanto a uso de máquinas virtuales. Esto sirve para cualquier empresa que use servicios cloud y tenga un nivel base de ingresos asegurados, pues le permitirá reducir apreciablemente los costos mensuales, y con la estrategia contable adecuada, que efectivamente sirva para los propósitos de la compañía.

A.3.3 Costos de una consultoría orientada a formar cultura de desarrollo seguro

En base a mi experiencia profesional, el costeo de estos cursos de capacitación son usualmente por persona que asiste. La duración es de 10 días hábiles, durante 3 horas cada uno, en horario vespertino usualmente. Para el caso de *Reservo.cl* estimo que el costo no debiese superar los CLP$3 millones.

Dicho monto sería un pago de una sola ocasión, con convenios y facilidades de acuerdo al proveedor; la empresa debe considerarlo como una inversión necesaria, para después cambiar su forma de desarrollar.

El beneficio, es que desarrollar en un marco seguro permite estar siempre un paso delante de los atacantes informáticos, salvo casos realmente de excepción, como fueron las recientes fallas en procesadores Intel y AMD. Contar con esa ventaja, es estratégico para una empresa que provee *Software-as-a-Service*.