UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Implementación de un chatbot sobre una arquitectura serverless

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Marc Solé Fonte

Tutor: Germán Moltó Martínez

Curso 2018-2019

Resumen

El trabajo realizado ha consistido en el diseño y la implementación de un bot conversacional siguiendo patrones propios de la arquitectura *serverless*. Para ello, se ha hecho uso de herramientas de computación en la nube proporcionadas por *Amazon Web Services*, así como del entorno de desarrollo *Dialogflow*. *Dialogflow* es una suite ejecutada sobre *Google Cloud* que permite facilitar la interacción humano-ordenador ofreciendo un desarrollo simple y eficaz con el que interpretar el lenguaje natural. El *back-end* de la aplicación, un consultor meteorológico, ha sido diseñado para ser ejecutado completamente sobre la nube de *Amazon.* El código, mayormente escrito en *Python*, aunque también en *Javascript*, es ejecutado a través de funciones de *Lamba* e invocado gracias a la *API Gateway*. Por otro lado, el *front-end*, desarrollado con la ayuda del *framework Vue.js*, se encuentra almacenado en *S3* y permite la autenticación a través de *Cognito*. Se ha obtenido como resultado del trabajo una aplicación que permite profundizar en la creación de bots conversacionales, así como explotar el uso de arquitecturas sin servidor para obtener resultados eficientes a bajo coste.

Palabras clave: computación en la nube, bot conversacional, serverless, aws, dialogflow, python

Abstract

TBD

Keywords: cloud computing, chatbot, serverless, aws, dialogflow, python

Resum

TBD

Keywords: informàtica en el núvol, bot conversacional, serverless, aws, dialogflow, python

Índice de contenidos

1. Introducción

Desde la llegada de *Siri*[1] a los teléfonos de *Apple*[2] en 2011, el mundo ha experimentado una profunda revolución en lo relacionado con los asistentes personales [SIRI]. El sueño de la asistencia personal completa está cada vez más y más cerca. Tanto que, a día de hoy, es difícil encontrar un teléfono móvil que no tenga instalado *Google Assistant*[3], un ordenador que no cuente con *Cortana*[4] o una casa que no tenga en su salón a *Alexa*[5].

Esta inclinación social tan reciente hacia el trato impersonal y la respuesta rápida ha supuesto que miles de empresas y particulares hayan querido tener su propio asistente, ofreciendo servicios de preguntas frecuentes, de planificación de viajes o hasta de asistencia de primer nivel. Desde el punto de vista del mundo informático, esto ha supuesto a su vez la llegada de cientos de *frameworks*, de sistemas especializados en la interpretación del lenguaje natural y, de, por supuesto, empresas y consultorías especializadas en el sector. No hablamos de una revolución, pues el procesamiento de lenguajes naturales ya se empezó a estudiar en la década de 1950 [TURING], pero sí de un arraigado auge que promete mantener la tendencia en los próximos años.

Por otro lado, y casi con la misma madurez en lo que respecta a fama, los servicios basados en la nube han ido adentrándose en la informática a través de la última década. Su principal ventaja es ofrecer una capacidad de cómputo hasta hace poco reservada a grandes corporaciones y afamadas universidades, bajo costes y precios de entrada asequibles por parte de no solo pequeñas y medianas empresas sino también de particulares. Aspectos como la escalabilidad rápida de recursos, la abstracción del hardware y el coste estrictamente vinculado al uso permiten a los desarrolladores centrarse más en la calidad de la solución y menos en los aspectos propios de la infraestructura.

Sin embargo, los proveedores de *cloud*, entre los que destacan *Amazon Web Services*[6], *Google Cloud*[7] y *Microsoft Azure*[8], no solo se limitan a ofrecer infraestructura como servicio (*Infrastructure as a Service, IaaS*), posibilitando el alquiler recursos de cómputo y almacenamiento, sino que también ofrecen, a un nivel ligeramente superior, servicios de plataforma (*Platform as a Service, PaaS*), que permiten centrarse puramente en el desarrollo e ignorar la capa de red y de software propio del sistema operativo.

No se quedan ahí, puesto que subiendo un poco más el nivel, y abstrayendo las mismas aplicaciones se llega al *software* como servicio (*Software as a Service, SaaS*), el cual facilita la ejecución de aplicaciones específicas como aquellas de *Big Data*, *Business Analytics*, *IoT* o *Machine Learning* a expertos del sector que no quieren adentrarse en la gestión, la actualización

[1] Apple, Inc. (2019). *Siri - Apple (ES)* - https://www.apple.com/es/siri/
[2] Apple, Inc. (2019). *Apple (España)* - https://www.apple.com/es/
[3] Google LLC. (2019). *El Asistente de Google* - https://assistant.google.com/intl/es_es/
[4] Microsoft Corporation. (2019). *Cortana* - https://www.microsoft.com/es-es/windows/cortana
[5] Amazon.com, Inc. (2019). *Amazon Alexa* - https://developer.amazon.com/es/alexa
[6] Amazon.com, Inc. (2019). *AWS* - https://aws.amazon.com/es/
[7] Google LLC. (2019). *Cloud Computing Services* - https://cloud.google.com/
[8] Microsoft Corporation. (2019). *Microsoft Azure* - https://azure.microsoft.com/es-es/

y el mantenimiento del código. A un nivel similar se encuentra el modelo de negocio basado en la ejecución de código encapsulado (*Function as a Service, FaaS*), el cuál permite ejecutar código de forma dinámica y sin gestión explícita de servidores por parte del usuario, dando lugar a la arquitectura conocida como sin-servidor, o *serverless*, y al patrón arquitectónico conocido como *Back-end as a Service (BaaS)*.

Juntando ambas propuestas, es fácil imaginar cómo el uso de una arquitectura basada en la nube puede facilitar la creación y el despliegue de un bot conversacional, limitándose no solo a la respuesta previamente definida sino también a la obtención y al procesamiento de datos gracias a otros servicios del mismo u otros proveedores *cloud*. La arquitectura planteada consiste, por lo tanto, en un *front-end* capaz de recoger la interacción del usuario, sea mediante texto o mediante voz y de invocar un *BaaS*, encargado, en sus distintas funciones, de procesar el lenguaje natural, de llamar al, o a los, servicios requeridos con la información que se necesite procesar y de enviar una respuesta al *front-end*.

En resumen, trabajar de forma coordinada para hacer uso de distintos microservicios desplegados en la nube permite crear un bot conversacional rico y eficaz en el ámbito social y empresarial, teniendo cabida tanto como elemento de asistencia o soporte que como producto comercial. Sin embargo, implementar este tipo de arquitectura no resulta trivial, por lo que se plantea un problema que puede ser de gran interés resolver.

1.1. Motivación

La idea de este trabajo surge como respuesta a la mezcla entre la inquietud personal por el mundo los bots conversacionales y la vocación hacia el mundo de la computación distribuida. El primero de los ingredientes, mantenido durante todo el grado, me llevó a la creación de varios bots y al estudio personal de herramientas de análisis de lenguaje natural, de métodos para la creación de *back-ends* e incluso de integración con servicios basados en el Internet de las cosas (IoT). Debido a ello, tuve claro que quería proponer un Trabajo de Final de Grado relacionado con el mismo ámbito de estudio.

Por el otro lado, mi interés por la computación distribuida se vió potenciado por la entrada al proyecto como tutor de Germán Moltó Martínez, Doctor en Informática especializado en sistemas distribuidos, arquitecturas orientadas a servicios y proveedores de servicios en la nube; y por mi contratación en el Instituto Tecnológico de Informática (ITI), como miembro del equipo de desarrollo e investigación en la línea de Sistemas Distribuidos. Queda claro, por lo tanto, el origen de la idea de mezclar en un mismo proyecto el espíritu propio de ambos campos.

El cuerpo del proyecto tiene a su vez su origen en la falta de disponibilidad de información relacionada con la creación de bots conversacionales funcionales de forma manual. Esto se debe a que la gran mayoría de la documentación disponible es referida directamente a la utilización de *frameworks* estrictamente ligados a árboles de decisiones con respuestas predefinidas o de bajo coste computacional. Es decir, no se profundiza en la utilización del servicio como puerta de entrada a una lógica de negocio posterior, sino que se utiliza como *endpoint* autosuficiente, capaz de cumplir por sí mismo con todas las necesidades del usuario.

Las propuesta, tal y como veremos a continuación, busca, por lo tanto, presentar a través de un ejemplo no sólo cómo realizar un bot conversacional básico sino también cómo desplegarlo de forma que sea viable comercialmente suponiendo el mínimo costo en recursos. Para garantizar la visibilidad de tal trabajo y, por lo tanto, potenciar su uso, distribución y modificación, con carácter comercial o privado, se ha decidido además mantenerlo disponible en GitHub como un proyecto de código abierto bajo la licencia Apache 2.0 [APACHE].

1.2. Objetivos

El principal objetivo del proyecto es el diseño de un bot conversacional desplegado sobre una arquitectura sin servidor, así como el desarrollo, la implementación, el despliegue y la realización de pruebas sobre el mismo. Se entiende diseño por lo tanto como la concepción de la idea y el planteamiento de la arquitectura, reservando la implementación del código y su ejecución a procesos posteriores.

Acotando el objetivo a otros más específicos para garantizar un correcto seguimiento de cada uno de ellos y, por lo tanto, para satisfacer el cumplimiento de cada uno de los requisitos, queda presente esta serie de objetivos:

- Desplegar un *front-end* que permita el acceso al mismo sin tener que hacer uso de un servidor dedicado, permitiendo en él funciones de autenticación e interacción con el bot.
- Entrenar un analizador de lenguaje natural que permita redirigir la entrada de un usuario al servicio más acorde a su petición y devolver, en un formato comprensible por el mismo, la respuesta proporcionada por el *back-end*.
- Implementar un *back-end* basado en funciones como servicio que permita recibir peticiones a través de una API y responder a las mismas mediante la ejecución de funciones sin servidor, reduciendo al mínimo el coste relacionado con el mantenimiento de los mismos.
- Realizar pruebas de rendimiento, a modo de comparativa entre una arquitectura *serverless* y una en la que se gestiona el hardware directamente, en busca de diferencias a nivel de costes, tiempos de respuesta y complejidad y entre un bot conversacional y un trabajador de soporte, ofreciendo una comparativa en rendimiento, coste de uso y adaptabilidad.
- Liberar el proyecto como contribución a la comunidad con una licencia de código abierto Apache 2.0, así como difundir los resultados en aquellos foros en los que se considere interesante su diseminación.

Estos objetivos específicos se deben entender como pequeños apartados que, una vez unidos y cohesionados, supondrán el cumplimiento del objetivo principal.

1.3. Impacto esperado

Se espera que el trabajo sirva como ejemplificación de cómo desarrollar bots conversacionales eficientes y ricos gracias a una lógica promovida por la utilización de una arquitectura *serverless* y de servicios propios de proveedores de la nube. Además, y como

impactos secundarios se espera promover la utilización este tipo de bots en entornos profesionales, debido a la dotación de una lógica mucho más profunda y funcional que la habitual, y la implementación de arquitecturas basadas en la nube, sea para este tipo de proyecto o no, debido al rendimiento mostrado y a la facilidad con la que se despliegan este tipo de servicios.

1.4. Estructura de la memoria [TBD]

La memoria sigue una estructura acorde a la guía que ofrece la UPV, aunque cuenta con ligeras modificaciones. Se empieza con una referencia del estado del arte en el capítulo 2, en el que se hace una comparativa de los proveedores de de servicios en la nube actuales, así como otra relacionada con las principales plataformas para el desarrollo de chatbots. A continuación, en los capítulos 3, 4 y 5 se plantean los problemas, así como las diferentes soluciones propuestas para resolverlos, y el desarrollo de éstas en caso de existir. En los dos capítulos siguientes, 6 y 7 se habla de la implantación del sistema, es decir, de los requisitos que tiene, de cómo desplegarlo y ponerlo en marcha y de cómo ejecutarlo y realizar pruebas sobre él. El capítulo 8, a su vez, tiene como finalidad mostrar los casos de uso y los tests de rendimiento ejecutados sobre el sistema. Para finalizar, se presenta la conclusión del trabajo, exponiendo si se han alcanzado los objetivos y de qué manera, y se proponen mejoras que se pueden llevar a cabo en un futuro, correspondiendo esto al capítulo 9.

2. Estado del arte

En este capítulo se define el estado en el que se encuentran los campos de investigación que se tratan a lo largo del proyecto. Sin embargo, al surgir éste del matrimonio entre los bots conversacionales y la computación en la nube, se considera necesario dividirlo entre ambos tópicos. Como consecuencia, se ofrece así un análisis de cuáles son las alternativas actuales para la implementación de tecnologías basadas en la interpretación de lenguaje natural y otro de cuáles son los principales proveedores en la nube y qué son capaces de ofrecer.

2.1. Análisis de herramientas para la interpretación de lenguaje natural

Nuestro lenguaje, tanto escrito como hablado, constituye un tipo muy particular de datos, que pueden ser analizados y procesados de tal forma que sean entendibles por las máquinas. Sin embargo, este proceso siempre ha sido un desafío, debido a la complejidad y la ambigüedad de la comunicación humana. Como consecuencia, existen diversas formas de afrontar el mismo problema, cada una con sus ventajas y desventajas, y con un caso de uso específico en función de qué se quiera conseguir y del tipo de datos del que se parte.

Podemos dividir esta diversificación en cuatro ramas, correspondientes a las resultantes de la unión de dos ejes: complejidad en la información y complejidad en la carga de trabajo. Es decir, no es lo mismo analizar órdenes estrictas y previamente definidas que tratar de comprender una conversación fluida y, a su vez, no es lo mismo tratar de encontrar información clave en una oración que detectar si cumple con ciertos patrones. Pongamos unos ejemplos, a modo de ligera aproximación:

- La oración “Enciende la luz” puede ser analizada como una entrada de baja complejidad en ambos sentidos, puesto que detectando el verbo “encender” y la palabra “luz”, previamente establecidos, es fácil llegar a la conclusión de que se está llamando a un método concreto y bien definido. Esto puede ser obtenido a través de un árbol de decisión y un sistema de análisis menor, dando lugar así a un modelo eficiente.
- La oración “Ve al mercado y compra manzanas”, tiene la complejidad de que, pese a que la oración es sencilla, hay que extraer información de ella y, por lo tanto, no es tan sencillo como invocar un método a través de un árbol de decisión, sino que ya es necesario un sistema experto que se encargue de realizar un diagnóstico y extraer la información necesaria para la ejecución de la función posterior.

De este superficial análisis, que podría llevarse a cabo en un documento completo dedicado a ello, se pretende extraer la diferencia entre un tipo de análisis de patrones y otro. Sin embargo, a medida que la complejidad de la información sube es mucho menos viable utilizar rutinas y sistemas basados en la experiencia, y resulta mucho más factible y lógico utilizar herramientas, generalmente basadas en *Machine Learning*, que toman decisiones en función de un

entrenamiento anterior. De esta forma, se puede escalar con relativa facilidad la cantidad de oraciones a procesar, puesto que no se depende de una plantilla predefinida.

Existen cientos de servicios, de pago, gratuïtos, o con capas de uso gratuïtas, que permiten llevar a cabo este análisis, abstrayendo al máximo al programador del trabajo sucio y fácilmente automatizable que supone el entrenamiento. De entre estas, se han elegido cinco para su análisis, tanto por importancia como por la cantidad de usuari*os* que las utilizan: *Amazon Lex*[9], *Google Dialogflow*[10], *IBM Watson Assistant*[11], *Microsoft LUIS*[12] y *Wit.AI*[13].

2.1.1. Amazon Lex

La alternativa de *Amazon*[14], presentada a la comunidad de desarrolladores en Abril de 2017 [AMAZON LEX], es principalmente conocida por ser la que da vida a Amazon Alexa, el asistente virtual de la empresa. El servicio de *deep learning* tiene entre sus principales atractivos la capacidad de analizar información con origen tanto textual como sonoro, ofreciendo lo que se conoce como *speech-to-text*, es decir, una transformación de una entrada audible a una salida legible.

La utilización del poder de cómputo propio de *Amazon Web Services* y la integración directa con sus funciones, a través de *Amazon Lambda*[15], permite generar rápidamente bots complejos sin tener que trabajar en integración a nivel de red y exposición de servicios entre distintos *clouds*. Además, la documentación está ampliamente detallada y extendida [LEX DOCS], lo que, junto a la gran cantidad de elementos predefinidos, facilita en gran manera una implementación rápida y sencilla del producto deseado. Lex es, por lo tanto, una buena elección para bots implementados por desarrolladores con poca experiencia, que buscan el entendimiento de comandos simples sin perder demasiado tiempo en esta parte del desarrollo.

De entre lo malo, los principales defectos de Amazon Lex son la falta de contextos, variables que definen el contenido de la conversación que se está manteniendo, para facilitar la comprensión. En la guía de desarrollo se indica que pueden ser emulados mediante el uso de variables de sesión, pero ya se incluye una capa de profundidad que el desarrollador no podría querer manejar. Otro de sus problemas es la utilización forzosa del inglés, el único idioma con el que es capaz de trabajar, por lo que resulta poco viable utilizarlo para el desarrollo de productos enfocados a públicos de países no angloparlantes. Finalmente, el hecho de que se establezca que toda entrada debe de producir forzosamente una salida emborrona algunos desarrollos a los cuáles no les interesa esta funcionalidad.

En lo referente a precio, la herramienta ofrece un precio estándar de 0.004 $ por petición de voz y 0.00075 $ por petición de texto. Basándonos en un ejemplo de 4000 peticiones de voz de 10 segundos y 10000 peticiones de texto mensuales, el cual repetiremos a lo largo del análisis,

[9] Amazon.com, Inc. (2019). *Amazon Lex* - https://aws.amazon.com/lex/
[10] Google LLC. (2019). *Dialogflow* - https://dialogflow.com/
[11] International Business Machines Corporation. (2019). *IBM Watson* - https://www.ibm.com/watson
[12] Microsoft Corporation. (2019). *Language Understanding (LUIS)* - https://azure.microsoft.com/es-es/services/cognitive-services/language-understanding-intelligent-service/
[13] Facebook Inc. (2019). *Wit.ai* - https://wit.ai/
[14] Amazon.com, Inc. (2019). *Amazon.es* - https://www.amazon.es/
[15] Amazon.com, Inc. (2019). *AWS Lambda* - https://aws.amazon.com/es/lambda/

el coste de uso de Amazon Lex sería de 23.5 $, un equivalente, al cambio, de 20.68 €[16], situándose como una alternativa medianamente cara en relación a sus competidores directos[17].

En conclusión, Lex es una herramienta poderosa y sencilla de utilizar para desarrollos simples y con poca complejidad o que requieran la integración con otros servicios de Amazon. Sin embargo, pierde frente a otras a la hora de ofrecer la profundidad que se necesita para un proyecto más avanzado. El hecho de que se limite al idioma anglosajón y que no cuente con una capa gratuïta viable, contribuye, en definitiva, a que se quede estancada en ser una herramienta suficiente, pero no destacable, para la creación de bots conversacionales.

2.1.2. Google Dialogflow

La herramienta propuesta por *Google*[18], anteriormente conocida como *Api.ai*, ofrece una alternativa ligeramente menos robusta que la de sus competidores directos, pero cuenta con algunas ventajas únicas en relación a ellos.

Entre algunas de las principales, destacan la oferta de agentes predefinidos, la capacidad de dar soporte a múltiples lenguajes y la posibilidad de editar código directamente, ofreciendo herramientas para el desarrollador capaces de ajustar a un nivel más bajo como debe ser procesada la información. Otra ventaja es la de integrar automáticamente los agentes con *Google Home*[19], así como con otros servicios de terceros como *Telegram*[20], *Twitter*[21] o *Skype*[22]. Sin embargo, su principal beneficio es *Chatbase*, su programa de análisis, respaldado con la tan larga establecida trayectoria de *Google Analytics*[23].

Entre sus defectos, su nivel de madurez es ligeramente inferior al de sus competidores, y el estar ligado a *Google Cloud Platform* hace que sean necesarios utilizar *webhooks* y exponer servicios si se quiere utilizar con otros proveedores *cloud* más ampliamente utilizados. Otro de sus problemas es que, al igual que *Amazon Lex*, toda entrada se ve obligada a presentar una respuesta, lo que da lugar a situaciones indeseadas y difícilmente evitables.

En lo que respecta al precio, *Dialogflow* permite el uso de una capa gratuita que permite el uso ilimitado de funciones de texto y voz. Sin embargo, a medida que el número de peticiones escalen a un tamaño propio de una mediana o gran empresa, se forzará el uso de una Enterprise Edition, con sus modos *Essentials* y Plus, que van desde los 0.002 a los 0.004 $ por petición de texto y de los 0.0065 a los 0.0085 $ por 15 segundos de audio. El ejemplo presentado anteriormente, de 10000 peticiones de texto y 4000 de audio mensuales entraría dentro de la

[16] Se ha utilizado una conversión de 1 $ a 0.88 € para los ejemplos, correspondiente al precio de cambio regulado del día 17/03/2019.
[17] Los costes han sido extraídos de *https://aws.amazon.com/lex/pricing/*, a día 17/03/2019.
[18] Google LLC. (2019). *Google* - https://www.google.es/
[19] Google LLC. (2019). *Google Home* - https://store.google.com/es/product/google_home
[20] Telegram FZ-LLC. (2019). *Telegram Messenger* - https://telegram.org/
[21] Twitter, Inc. (2019). *Twitter* - https://twitter.com/
[22] Microsoft Corporation. (2019). *Skype* - https://www.skype.com/es/
[23] Google LLC. (2019). *Google Analytics* - https://analytics.google.com/analytics/

capa gratuita, aunque, en el caso de querer utilizar la versión más cara, supondría un precio de 74 $ mensuales, unos 65,12 €[24].

En conclusión, *Dialogflow* es una herramienta que, pese a sus pequeños defectos, ofrece un servicio más que suficiente para su capa gratuita, permitiendo desplegar servicios en producción directamente sobre ésta. Por otra parte, su capa de pago no resulta tan rentable como otras, pero cuenta con el respaldo de la trayectoria de Google en el análisis de datos y con soporte a múltiples lenguajes, lo que la hacen viable donde otras herramientas no lo son.

2.1.3. IBM Watson Assistant

Watson Assistant es la alternativa para la interpretación de lenguaje natural propuesta por *IBM*[25] y lanzada al mercado en 2016. Entre sus características principales están el soporte de hasta 13 lenguajes, la capacidad de ejecutarse en entornos basados en la nube u *on-premises* y un enfoque en la detección eficiente y distintivo frente a las otras opciones.

Debido a ese enfoque, las principal ventaja de *IBM Watson* es su capacidad de detección y análisis de las peticiones del usuario, siendo capaz de obtener resultados con un mucho mayor rango de precisión que sus alternativas. Para conseguirlo, la herramienta utiliza un tipo de ordenamiento específico de sus posibles entradas, de modo que proporciona prioridades distintas a cada una de ellas. Otra de sus ventajas es la inexistencia de la obligación de respuesta, de modo que se pueden crear conversaciones ricas en las que, por ejemplo, el bot no responda hasta que no tenga clara la pregunta.

Las principales desventajas de este *framework* son el difícil tratamiento de los contextos y la falta de un aprendizaje activo. Por una parte, esto se debe a que los contextos no cuentan con un tiempo de vida definido, obligando al desarrollador a prestar atención constante a su estado y a si es necesario mantenerlos, eliminarlos o iniciarlos de nuevo. Por otra, la falta de una capacidad de aprendizaje automatizada hace que *Watson* no aprenda de la interacción que está manteniendo con el usuario, lo que conduce un bot más fiable pero mucho más difícil de entrenar. Finalmente, la que podría ser la mayor diferencia respecto a los competidores ya presentados es que *Watson* no acepta peticiones a través de audios, por lo que requiere de una herramienta de transformación de *text-to-voice* adicional.

Al igual que *Dialogflow*, *IBM Watson* ofrece una capa gratuita que puede ser mejorada hasta alcanzar niveles que cumplan con requisitos empresariales. Su primer plan alcanza los 10000 mensajes de texto al mes, con una capacidad máxima de 100 nodos de diálogo y 80 entradas disponibles. Por otro lado, su capa de pago más económica tiene un coste de 0.0025 $ el mensaje de texto, y ya permite el manejo de mensajes y nodos de forma ilimitada. Finalmente, las opciones *Plus*, *Premium* y *Private* requieren del contacto directo con la empresa para establecer un contrato. Siguiendo el patrón, la capa gratuita se quedaría corta, por lo que sería

[24] Los costes han sido extraídos de *https://dialogflow.com/pricing*, a día 17/03/2019.
[25] International Business Machines Corporation. (2019). *IBM* - https://www.ibm.com/

necesario pagar un precio de 35 $ mensuales, unos 30.8 €[26], más el coste adicional de convertir 4000 mensajes de audio en voz en otro servicio.

En conclusión, la alternativa de *IBM* cuenta con la mejor detección de las intenciones del usuario y con un precio ligeramente más económico en relación al de sus competidores directos. Sin embargo, la falta de comodidades para el desarrollador obliga a éste a tener que realizar trabajo abstraído en otros *frameworks* y hasta de tener que depender de un servicio externo para la interpretación de peticiones basadas en audio.

2.1.4. Microsoft LUIS

El *framework* de la empresa poseedora de *Windows*[27] y *Office*[28], *Microsoft*[29], es conocido por sus siglas *LUIS*, correspondientes a *Language Understanding Intelligent Service*, traducido al castellano como *Servicio Inteligente para la Comprensión del Lenguaje*. Esta herramienta destaca frente a las demás por ser de bajo nivel, ofreciendo un desarrollo muy ligado al programador y al código, lo que dificulta el proceso pero permite obtener resultados muy similares a los deseados.

Este casamiento con el bajo nivel llega a tal punto que ni siquiera existe una interfaz gráfica para la edición, teniendo que conocerse y modificar el código forzosamente para obtener nuevos resultados. Esto puede entenderse como una ventaja, ya que convierte a *LUIS* en una buena plataforma para los desarrolladores que buscan una gran personalización y capacidades robustas. Además, el uso de *C#* como lenguaje de desarrollo hace que equipos especializados en el *stack* de *Windows* tengan una línea de aprendizaje reducida. Otras de sus ventajas son la inclusión de más de cien entradas por defecto y la capacidad de trabajar nativamente con *Skype*.

Sin embargo, la ventaja puede no serlo si se buscan las condiciones contrarias: facilidad de implementación y desarrollo para un equipo poco especializado. Esto se debe a que, del mismo modo que el bajo nivel permite una herramienta más poderosa, también la convierte en una más complicada. La falta de un editor visual, también, hace que muchos programadores se puedan ver superados por la cantidad de trabajo manual a realizar. Además, *LUIS* en su capa gratuita no cuenta con transformación de audio a texto, por lo que hay que plantear y/o contratar el servicio como algo adicional, al igual que con *IBM Watson Assistant*.

Respecto al precio, la capa gratuita llega a las 10000 transacciones de texto por mes, con un máximo de 5 transacciones por segundo. En la capa de pago, las transacciones por segundo llegan a las 50, y el número de ellas llega a ser ilimitado, ofreciendo un coste de 1.50 $ por cada 1000 peticiones de texto y 5.50 $ por cada 1000 peticiones de audio[30]. Asumiendo la capa

[26] Los costes han sido extraídos de https://www.ibm.com/cloud/watson-assistant/pricing/, a día 26/03/2019.

[27] Microsoft Corporation. (2019). *Windows* - https://www.microsoft.com/es-es/windows

[28] Microsoft Corporation. (2019). *Office 365* - https://www.office.com/

[29] Microsoft Corporation. (2019). *Microsoft* - https://www.microsoft.com/es-es

[30] Los costes han sido extraídos de https://azure.microsoft.com/en-us/pricing/details/cognitive-services/language-understanding-intelligent-services/ a día 26/03/2019.

gratuita como insuficiente para el ejemplo, debido a la falta del servicio *speech-to-text* en este nivel, el coste final sería de 37 $, equivalentes a 32.56 €.

Para resumir, *Microsoft LUIS* es una alternativa económica y poderosa para el desarrollo de bots conversacionales. Sin embargo, el hecho de que la única forma de desarrollar sea de bajo nivel y a través de código la convierte en un arma de doble filo, con capacidad más que suficiente para que un equipo especializado extraiga lo máximo del producto pero con una curva de entrada demasiado elevada para aquellos que solo busquen un desarrollo sencillo y funcional.

2.1.5. Wit.AI

La alternativa de *Facebook*[31] fue iniciada como producto estrella de una *startup* en 2013 y adquirida por la multinacional en 2015 [WITAI]. Su principal característica es la falta de un servicio de calidad al nivel de la de sus competidores, siendo la alternativa más lenta y con peor rendimiento de las presentadas. Sin embargo, su aliciente es que no existe una capa de pago, por lo que todo el uso que se realice de este servicio entra dentro del uso gratuito.

Su principal ventaja, por lo tanto, es la capacidad de funcionar sin generar costes, permitiendo a empresas y usuarios mantener un bot desplegado sin ningún tipo de gasto asociado. Su funcionamiento es suficiente para la mayoría de los casos no profesionales y permite la introducción al arte de la implementación de bots conversacionales. Además, *Wit.ai* ofrece soporte para hasta 50 idiomas, permitiendo extender el producto sin preocuparse de fronteras.

La parte negativa es que la falta de ingresos estáticos como en el resto de plataformas obliga a la herramienta a ofrecer un servicio más modesto, mucho más lento y con peores resultados, quedándose en segundo plano cuando se está buscando un resultado profesional y comercializable.

En conclusión, *Wit.AI* es una alternativa gratuita pero muy simple en relación a sus competidores, por lo que se queda relegada a una posición en la que la economía puede ser lo estrictamente relevante. Sin embargo, teniendo en cuenta que la capa gratuita de *Dialogflow* ofrece mejores resultados, es complicado encontrar casos de uso viables.

2.2. Análisis de los principales proveedores de servicios en la nube

Existen decenas de proveedores de servicios en la nube debido a que se trata de un sector en auge que no solo mueve miles de millones, sino que también sirve como marca de prestigio para las empresas que consiguen desplegar el suyo propio. Debido a ello, es necesario apartar el grano de la paja con frecuencia para tener la capacidad de elegir el mejor proveedor para nuestro proyecto y al mejor precio.

[31] Facebook Inc. (2019). *Facebook* - https://es-es.facebook.com/

Para ello, lo primero es tener muy claras cuáles son nuestras necesidades, debido a que no todos los proveedores son iguales y, pese a que quizás uno sea mucho mejor en lo que a bases de datos se refiere, nos interesan más las herramientas para el despliegue de una red de *blockchain* que propone otro. Por supuesto, existe también la posibilidad de utilizar un entorno *multi-cloud* en el que se convive con múltiples proveedores para un mismo proyecto. Otras opciones a tener en cuenta son la adopción global, el soporte, la disponibilidad en nuestra región y, por supuesto, el precio.

Debido a que nuestro proyecto tiene su grueso en las funciones como servicio, se dedicará especial interés a esta funcionalidad. Sin embargo, a lo largo de este apartado nos centraremos más en la adopción global porque un bot puede tener cientos de funcionalidades y resultaría limitante cortar lazos antes de conocer cuáles son nuestras necesidades reales. Por ejemplo, un bot puede utilizarse para almacenar recordatorios, por lo que nos interesarían opciones relacionadas con bases de datos o calendarios, pero también puede tener su uso en reenviar los mensajes a través de un servidor de email, por lo que serían más potentes aquellos clouds con más opciones de integración.

Volviendo a la adopción, existen muchos portales que muestran la evolución de la misma y sirven para mostrar no solo el uso actual sino la predicción de uso en los próximos años. A continuación, se muestra un gráfico que corresponde a la afiliación a los distintos proveedores por parte de las empresas que ya usan cloud público o tienen en mente utilizarlo según Flexera[32]:

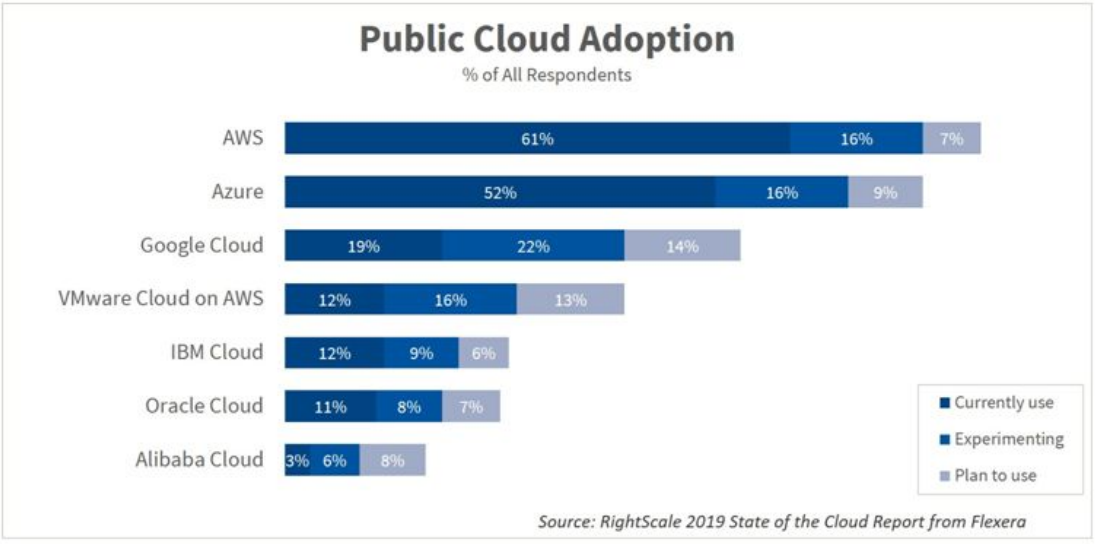

Figura 1 - Adopción de los distintos proveedores de cloud público según Flexera.

Como se extrae del gráfico, los principales proveedores a fecha de la escritura del documento son tres: *Google Cloud Platform*, *Microsoft Azure* y *Amazon Web Services*. Sin embargo, siguen muy de cerca *VMWare Cloud*[33], *IBM Cloud*[34], *Oracle Cloud*[35] y *Alibaba Cloud*[36], este último con

[32] Flexera Software. (2019). *Flexera* - https://www.flexera.com/
[33] VMware, Inc. (2019). *VMWare Cloud* - https://cloud.vmware.com/
[34] International Business Machines Corporation. (2019). *IBM Cloud* - https://www.ibm.com/es-es
[35] Oracle Corporation. (2019). *Oracle Cloud* - https://cloud.oracle.com/home
[36] Alibaba Group Holding Limited. (2019). *Alibaba Cloud* - https://www.alibabacloud.com/

especial presencia en el mercado asiático. Se puede extraer también del mismo gráfico la gran demanda hacia este tipo de servicios, ya que ningún proveedor prevé una subida inferior al 6% a lo largo del año.

Para nuestro análisis tendremos en cuenta los tres primeros.

2.2.1. Amazon Web Services

La primera de las plataformas a analizar es la de Amazon, debido a que también fue la primera en ver la luz de forma pública [AMAZON]. Este proveedor no solo destaca por ser el más maduro, sino que también lo hace por ser el que estrena más servicios al año, buscando abarcar el mayor número de clientes posibles abarcando, a su vez, el mayor mercado posible. Estos añadidos se presentan anualmente en una conferencia conocida como *re:Invent* [REINV].

La principal ventaja de AWS es la de madurez de sus soluciones, que suelen tener más características que las de sus competidores, así como su amplio abanico de servicios que se ven justificados por la actitud pionera e inquieta de la empresa. Debido a ello, es también la plataforma de servicios en la nube que más dinero mueve, según Forbes[37] [FORBES]; que más servicios ofrece y que más usuarios tiene, como hemos visto en el apartado anterior.

Respecto a las funciones como servicio, Amazon también es pionera debido a su producto AWS Lambda, introducido al mercado en noviembre de 2014 y siendo vanguardia en lo que respecta a la adopción del concepto *serverless* por parte del público general. Entre sus ventajas están el soporte de múltiples lenguajes de programación, como Node.js, Python, Java 8, C# (.NET core) y Go; su fácil integración con el resto de servicios de la plataforma y la posibilidad de aplicar políticas de uso.

Sin embargo, el principal problema de Amazon Web Services es su precio, y es que su punto flaco en relación a sus competidores. Pese a ello, muchas herramientas cuentan con una capa de uso gratuita, como la misma AWS Lambda, que permite realizar hasta un millón de solicitudes mensuales de forma gratuíta.

En conclusión, Amazon ofrece una plataforma madura y fiable que destaca no solo por ser la oferta más utilizada del mercado sino también por ofrecer más servicios y, generalmente, mejor evolucionados que los de sus competidores directos.

2.2.2. Microsoft Azure

La segunda de las plataformas de Cloud Computing en aparecer fue la de Microsoft, lanzada al mercado como producto comercial en 2010 [AZURE]. Pese a que ésta nunca ha llegado a mantener la popularidad ni la cantidad de usuarios que gestiona AWS, ha sido capaz de encontrar su hueco en el mercado a través de la especialización, ofreciendo servicios personalizados para empresas y facilitando la implementación de nubes híbridas en las que la empresas pueden seguir dependiendo de *data centers* locales.

37

Entre sus principales puntos diferenciadores está la integración con otro software de *Microsoft*, como *Windows*, *Office* o *SQL Server*, donde la empresa ya tenía un gran mercado. Además, ésta vinculación no se limita a la interoperabilidad, sino que también se traduce en significativos descuentos para la empresa cliente si deciden utilizar varios de sus productos o servicios a la vez. Otros aspectos a destacar son el ya mencionado *hybrid cloud*, el soporte de cara al *Open Source*, y la confianza de la comunidad, siendo el segundo proveedor más utilizado a día de hoy.

En lo que respecta a las funciones como servicio, *Azure* cuenta con su propia herramienta, conocida como *Azure Functions* [AZUFU]. Pese a que en general funciona ligeramente peor que *Amazon Lambda* [BENCHMARK], tiene algunas ventajas como el soporte a otros lenguajes, como *PHP* y *F#*, la no-limitación del número de funciones concurrentes, la posibilidad de desplegar código alojado en otras plataformas como *GitHub*, *One Drive* o *Dropbox* y el escalado manual y parametrizado.

Sus principales problemas están relacionados con la falta de documentación y la dificultad de uso, habiendo recibido múltiples quejas por parte de clientes en relación a la falta de soporte y de ejemplos de uso. En lo que respecta al precio, ofrece un servicio ligeramente mejor que *Amazon*, sobre todo si se cuenta con alguno de los descuentos ya presentados. Por su parte, *Azure Functions* ofrece la misma capa gratuita: un millón de peticiones mensuales.

Para resumir, *Azure* es una plataforma muy potente que, pese a que sigue ligeramente por detrás de *Amazon* es útil en necesidades concretas y sabe encontrar su mercado aprovechando las necesidades de sus clientes.

2.2.3. Google Cloud Platform

La última y quizás menos importante de las plataformas presentadas es la de *Google LLC*, que entró al mercado tarde y no tiene la confianza por parte de las empresas que tienen sus otros dos competidores. Sin embargo, ofrece algunas ventajas en lo que se refiere a sus herramientas y precios.

Empezando por éstas, la experiencia técnica de la compañía permite que destaque en sus herramientas de *deep learning, machine learning* y *data analytics*. Además, su fuerte integración con *Kubernetes*, donde sí ha sido pionera, ha permitido que sea elegida por muchos Arquitectos Cloud para la realización de sus tareas [K8S]. Se entiende así como una nube diseñada para aplicaciones ya pensadas para ella, es decir, aquellas que se conocen como *cloud-native*. Por otro lado, su precio es bastante inferior al de sus competidores directos, siendo hasta un 50% más barato que Amazon [CHP50].

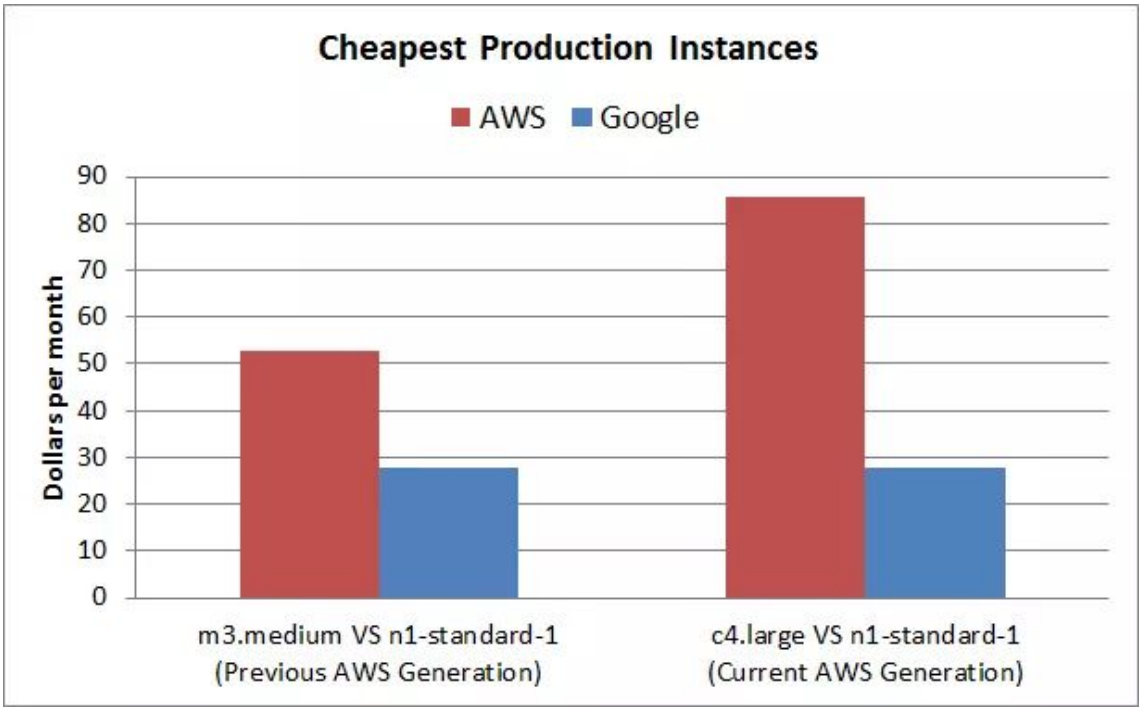

Figura 2 - Comparación de precios mínimos entre *Amazon Web Services* y *Google Cloud Engine*.

En lo que respecta a *Google Cloud Functions*, la opción serverless de *Google Cloud Engine*, éstas se quedan atrás respecto a sus competidores, ofreciendo soporte para solo dos lenguajes (*Node.js 6, Node.js 8* y *Python 3.7*), no permitiendo ejecutar más de 1000 funciones de forma concurrente, no contando con una herramienta para la orquestación y teniendo un tiempo máximo de ejecución por defecto 15 veces menor que el de *Amazon*. Hay que decir, sin embargo, que su capa gratuita es de 2 millones de peticiones y su uso resulta también más económico que el de sus competidores.

Las desventajas de *GCE* son, por lo tanto, la poca madurez de su servicio, la tardía entrada en el mercado y el poco foco en las empresas, lo que sitúa a la plataforma como la segunda opción en un entorno *multi cloud* pero muy pocas veces como la primera opción para un proyecto empresarial.

Para resumir, la plataforma de *Google* sigue siendo muy joven para competir directamente con *Amazon* y *Microsoft*, aunque poco a poco va encontrando su lugar en el mercado ofreciendo herramientas poderosas para los *DevOps* y entendiendo su misión como plataforma de refuerzo en un entorno *multi-cloud*.

2.3. Crítica del estado del arte

Tal y como se ha visto en los apartados anteriores, existen plataformas que permiten la creación de bots conversaciones de manera simple y funcional a la vez que existen proveedores de servicios en la nube que ofrecen herramientas cada vez más potentes para almacenar información, procesar eventos y analizar datos provenientes de clientes externos.

Estas plataformas, sin embargo, no están preparadas para interactuar entre sí, por lo que en ninguna de ellas se plantea el desarrollo directo de un *chatbot* capaz de analizar lenguaje natural y de responder acorde a él siguiendo el patrón establecido por funciones ejecutadas en la nube. Es decir, ambos campos están bien representados por múltiples herramientas que permiten al cliente desarrollar soluciones acorde a lo que ofrecen, pero en ningún caso se plantea la vinculación de ambos ámbitos como vía hacia la creación de bots conversacionales potentes y ricos en posibilidades.

2.4. Propuesta

Para solucionar los problemas relacionados con la falta de cohesión entre los servicios basados en la nube y el desarrollo de bots conversacionales, explicados en la sección **2.3**, se plantea la posibilidad de analizar y desarrollar diversos ejemplos en los que ambos campos interactúen entre sí, a modo de ejemplo y guía para la producción de nuevos sistemas basados en la misma arquitectura y con el mismo tipo de corte.

De entre las herramientas presentadas en el estado del arte, y para ser vinculadas entre sí, se utilizarán mayoritariamente dos para su integración. Por una parte, *Dialogflow*, el *framework* de *Google*, explicado en la sección **2.1.2**, será de utilidad debido a que permite desarrollar sistemas válidos para producción dentro de su capa gratuita, ofreciendo un servicio ligeramente inferior al de sus competidores bajo pequeños niveles de entrenamiento pero a un coste nulo, más allá de la posible recolección de datos por parte de la empresa proveedora. Por otra parte, *Amazon Web Services*, explicado en la sección **2.2.4**, será el proveedor de *Cloud* elegido, debido a la facilidad a la hora de desplegar servicios serverless con *Amazon Lambda*, a la alta integración en la comunidad y a la disponibilidad de una capa gratuita que permite ejecutar los servicios requeridos sin suponer ningún coste adicional. Por lo tanto, se obtiene del uso de ambas herramientas un servicio gratuito, eficaz y fiable.

3. Análisis del problema

Para conocer la dimensión del problema y poder realizar un análisis, es necesario conocer qué se quiere obtener con cada uno de los enfoques y cómo funciona a día de hoy la integración entre ambos tipos de servicios. Es decir, no solo es necesario comprender el funcionamiento individual de cada una de las herramientas sino también cómo interactúan entre ellas y qué resultado podemos obtener de esta comunicación.

3.1. Análisis de los distintos retos

Tal y como se ha presentado previamente en el apartado **2.4**, se considera que la mejor forma de abordar el problema es dividiéndolo en distintos retos que permitan analizar con un enfoque específico distintas configuraciones. Como consecuencia, se presentan a continuación los tres campos que se van a abordar: *front-end*, interpretación de lenguaje natural y *back-end*.

3.1.1. El front-end

El primer reto a la hora de diseñar un servicio *serverless* es el de ser capaz de ofrecer acceso al mismo al cliente sin tener que depender de una máquina que se ejecute a la espera de peticiones. Esto es debido a que, de forma clásica, existe un equipo conocido como *host* que almacena en él el código (generalmente HTML, CSS y JavaScript) que debe ser procesado por el cliente, así como sus fuentes (imágenes, vídeos), y las hace públicas exponiéndose a todo Internet o, en entornos permisionados, a una red privada.

Es lógico pensar que siempre que se quiera exponer algo a través de una red es necesario un punto de acceso correspondiente a un equipo informático que maneje las peticiones y responda a las mismas de la forma más adecuada posible. Además, si se está trabajando con ficheros, como los de HTML o JavaScript, estos deben de estar almacenados en algún lugar, por lo que todavía es más complicado no depender de un equipo.

La solución a este problema pasa por no tener que depender de un servidor completo; es decir, por contratar servicios que suplan las necesidades del sistema. Serían ejemplos, por lo tanto, un punto de acceso que ofreciera una dirección en formato de URL o dirección IP y puerto o un servicio de almacenamiento de objetos escalable y altamente disponible que permitiese mantener los datos de la aplicación y servirlos a través de peticiones REST.

La finalidad de este reto, por lo tanto, es la de descomponer la capa de presentación en pequeños servicios coordinados entre sí que permitan, de forma transparente para el usuario, responder a peticiones como la de descargar e inicializar una página web. De esta forma, se elimina completamente la necesidad de mantener un servidor dedicado y se reducen, como consecuencia directa, los costes asociados a su alquiler o compra y mantenimiento.

3.1.2. La interpretación de lenguaje natural

El segundo reto a abordar es el de configurar la herramienta de interpretación de lenguaje natural. Para ello, tal y como se ha explicado en el apartado **2.1**, existen distintos enfoques que permiten afrontar el problema desde variados puntos de vista y con costes y resultados muy diversos. Como consecuencia, es necesario encontrar qué requisitos tiene nuestro bot para localizar qué tipo de sistema va a ser implementado.

Se espera del bot a desarrollar, un simple consultor meteorológico, que no solo responda a patrones definidos como *"¿Qué tiempo hace en Madrid?"*, sino que sea capaz de encontrar la intención del usuario a través del análisis de la sentencia, teniendo en cuenta el orden, el contexto y las palabras en sí. Se entiende, por lo tanto, que serían preguntas válidas para obtener la misma respuesta o una directamente similar *"¿Voy a necesitar paraguas en Madrid?"*, "*¿Salgo a dar un paseo?"*, tras localizar al usuario en la capital, o *"¿Y en Madrid?"*, tras responder a la petición *"¿Qué tiempo hace en Barcelona?"*.

Este conjunto de posibilidades, así como la necesidad de recoger información específica de cada petición como localización, actividad o fecha, elimina la posibilidad de limitar el desarrollo a un árbol de decisión, por lo que hay que utilizar un sistema experto o, más sencillamente, una herramienta de *Machine Learning* previamente entrenada. Esta sencillez tiene su origen en los servicios de pago, gratuitos, o con capas de uso gratuitas, que permiten llevar a cabo este análisis, abstrayendo al máximo al programador del trabajo sucio y fácilmente automatizable que supone el entrenamiento.

En este segundo reto, el objetivo reside en conocer, analizar y profundizar en el campo de las herramientas de análisis y *Machine Learning* orientadas al lenguaje natural, así como en desarrollar una lógica capaz de interactuar con humanos de la forma menos artificial posible.

3.1.3. El back-end

El último de los retos, relacionado con la implementación del *back-end* y, por lo tanto, de las funciones como servicio, tiene como foco gestionar la lógica de la aplicación vinculada con el *fulfillment* o rellenado de las peticiones al bot. Esto es, dada una petición interpretada por la herramienta de análisis de lenguaje natural que no puede ser respondida directamente, el completado que se da por por parte del *Backend as a Service*.

Por ejemplo, un bot que responda a preguntas predefinidas que no tengan un resultado variable y que puedan ser memorizadas, como *"¿Cómo te llamas"* o *"¿Cuáles son tus Términos y Condiciones de Uso?"* puede tener directamente una respuesta almacenada y ofrecerla, invariablemente, ante este tipo de peticiones. Sin embargo, si la pregunta requiere de un cálculo, un procesamiento o una búsqueda como en *"¿Cuánto es dos más tres?"*, *"¿Cuál es la capital de Eslovaquia?"* o *"¿Tengo correo?"*, esta información no puede ser almacenada y, por lo tanto, debe de gestionarse tras cada petición.

Tanto por limitaciones lógicas como para garantizar la separación en capas de la arquitectura, se separa el análisis de las peticiones de su completado, y ésta segunda capa es la

que se estructura a lo largo de este reto. Dando forma al problema, para un bot que se encarga de consultas relacionadas con el clima, hace falta redirigir las llamadas a una API que obtenga esta información, así como darle un formato aceptable por el usuario a la misma. Finalmente, al no querer utilizar un servidor, todas estas funcionalidades deberán de ser construidas de forma que permitan su ejecución sin tener que depender de un equipo que funcione de forma pasiva a la espera de peticiones.

En conclusión, el objetivo de este reto es el de desarrollar la lógica que corresponde al procesamiento de las peticiones y a la obtención de respuestas válidas, con el interés de profundizar en el uso de *Cloud Computing* y, en especial, de las funciones como servicio.

3.2. Análisis de la seguridad

Para garantizar que los bots no pueden ser utilizados por personas no autorizadas, se ha de crear un modelo de seguridad en la plataforma. Éste debería estar muy relacionado con la autenticación y la persistencia, pues se quiere comprobar y asegurar que cada usuario pertenece a solamente una persona y de que estos usuarios van a ser guardados y persistidos para conexiones futuras.

Para esto, se plantea la utilización de un servicio de autenticación externa que sirva como barrera a la entrada para la comunicación con *Dialogflow* y con un servicio de autenticación implementado como una función *serverless* que permita limitar las fuentes posibles con acceso a los servicios de procesamiento y gestión de la información. Es decir, al utilizar distintos microservicios se plantea la utilización de distintos mecanismos: uno para restringir el acceso al bot a los usuarios registrados y otro para restringir el acceso al *cloud* al bot, evitando que terceros ataquen directamente a la lógica del producto.

3.3. Formalización del problema

Tal y como se comentaba en el apartado **1.2**, la finalidad del proyecto es la de mostrar el diseño de un bot conversacional desplegado sobre una arquitectura sin servidor, así como el desarrollo, la implementación, el despliegue y la realización de pruebas sobre el mismo. A continuación se exponen aquellos objetivos relacionados directamente con la implementación, añadiendo objetivos descubiertos en el análisis de la seguridad y de los distintos retos:

- **Desplegar un *front-end* que permita el acceso al mismo sin tener que hacer uso de un servidor dedicado, permitiendo en él funciones de autenticación e interacción con el bot.** Los aspectos generales a derivar de este objetivo son el de diseñar una interfaz estática, que permita su almacenamiento sobre un servicio como Amazon *S3, el servicio de almacenamiento de ficheros de AWS*; el de desplegarla de forma que pueda ser accedida a través de una función serverless y el de controlar la autenticación, la autorización e identificación tanto de los usuarios al bot como del bot a la lógica.
- **Entrenar un analizador de lenguaje natural que permita redirigir la entrada de un usuario al servicio más acorde a su petición y devolver, en un formato comprensible por el mismo, la respuesta proporcionada por el *back-end*.** De este

objetivo se extrae que es necesario configurar y entrenar la herramienta de análisis de lenguaje natural para transformar la entrada del usuario en llamadas a funciones y, además, que se requiere que la herramienta sea capaz de invocar al *back-end* a voluntad.

- **Implementar un *back-end* basado en funciones como servicio que permita recibir peticiones a través de una API y responder a las mismas mediante la ejecución de funciones sin servidor, reduciendo al mínimo el coste relacionado con el mantenimiento de los mismos.** Se requiere, por lo tanto, desarrollar las funciones serverless que puedan ser invocadas por el usuario y exponerlas de forma pública, haciéndolas independientes de la localización de la herramienta de análisis de lenguaje natural.

Una vez se han visto los objetivos del proyecto y se han relacionado con tareas más concretas y abordables, se formalizan éstas y lo que se pretende conseguir con cada una de ellas, dando lugar a los problemas que resolver en el apartado **3.5**:

- **Diseño de la interfaz estática.** Se requiere una interfaz estática que pueda ser almacenada sobre un sobre un servicio como *S3*, de modo que funcione sin que haya un servidor que reciba y procese las peticiones.
- **Despliegue de la interfaz sin hacer uso de servidor dedicado.** La interfaz estática debe de estar desplegada de tal forma que puede ser devuelta al usuario a través de la invocación de una función serverless. El usuario debe de poder interactuar con la interfaz web tal y como lo haría si se conectara a un servidor clásico.
- **Control de la autenticación, autorización e identificación.** Debe de ser posible el registro y el inicio de sesión, restringiendo el acceso al bot a aquellos usuarios dados de alta. Además, debe de ser posible la limitación del uso de la lógica del proyecto basada en servicios *cloud* a aquellos servicios autorizados.
- **Configuración de la herramienta de análisis de lenguaje natural.** La herramienta elegida debe de configurarse y entrenarse de modo que permita traducir las comunicaciones posibles que se extraigan de los casos de uso en respuestas que puedan invocar o no funciones del *back-end*.
- **Integración entre microservicios.** Debe existir una comunicación entre la herramienta de análisis y el *back-end* de modo que la primera sea capaz de autenticarse y realizar peticiones en la segunda en función de las necesidades surgidas.
- **Desarrollo de las funciones serverless.** Para cada una de las llamadas establecidas en los casos de uso que requieran la interacción con el *back-end*, deberá haber una función o método *serverless* que sea válido.
- **Exposición de las funciones de forma pública.** Una vez diseñadas todas las funciones *serverless*, debe haber alguna forma de acceder a ellas a través de un servicio RESTful abierto a la red.

3.4. Casos de uso [TBD]

FIGURA DIAGRAMA UML

La figura [FIGURA] nos muestra el diagrama UML correspondiente a las acciones que se deberían de poder realizar en el sistema que da acceso al bot. Este sistema debería de permitir distintas funciones en función de los permisos propios del actor, permitiendo el registro y el inicio de sesión a agentes sin autenticar (*anónimos*) y la interacción con el bot y el cierre de sesión a aquellos autenticados (*usuarios)*. La interacción con el bot, formalmente *Ptolomeo,* debería de permitir conocer el estado meteorológico de un lugar en un tiempo concreto, con valores definidos o por defecto, así como extrapolar esta consulta a partir de preguntas relacionadas y responder a una comunicación básica.

FIGURA DIAGRAMA UML

Analizando más profundamente la interacción en la figura [FIGURA] se aprecia que la comunicación básica se limitará a cuatro funciones: saludo o interacción de inicio, despedida o interacción de fin, agradecimiento y consulta de acciones posibles. A su vez, respecto a las funciones relacionadas con la meteorología habrá cinco principales: consultar el estado meteorológico, como en *"¿Qué tiempo hace?"*; obtener la temperatura, como en *"¿Hará frío?"*; preguntar qué tipo de prendas serán necesarias, como en *"¿Voy a necesitar abrigo?"*; consultar si el tiempo es adecuado para hacer alguna actividad, como en *"¿Podré ir a la playa?"*, y obtener una respuesta condicional, como en *"¿Va a llover?"*.

Todas las interacciones relacionadas con la meteorología tendrán valores de tiempo y lugar predefinidos, siendo éstos la hora en la que se ha realizado la consulta y la localización del bot, Valencia, respectivamente. Estos valores podrán ser modificados en la configuración. Además, todas las posibles entradas permitirán la introducción de fecha, hora y lugar de forma nativa, como en *"¿Va llover esta tarde en Sevilla?"* o *"¿Nevará en agosto en Japón?"*. Finalmente, todas las interacciones permitirán continuar el contexto o añadir más información para obtener nuevas respuestas. Es decir, será posible consultar *"¿Y mañana?"* después de obtener la respuesta a *"¿Va a llover esta tarde?"*.

3.5. Identificación y análisis de soluciones posibles

De la misma manera que se han planteado y formalizado los problemas que se tienen que atacar durante la realización del proyecto, se van a identificar y analizar las posibles soluciones para cada uno de ellos.

3.5.1. Diseño de una interfaz estática

El reto del diseño de una interfaz estática surge de la necesidad de poder almacenar el *front-end* de la aplicación de modo que siga funcionando. Esto supone un conflicto con el método moderno para el desarrollo de este tipo de aplicaciones, que se basa en utilizar un framework como *Angular 6* o *Vue.js* para desarrollar y exponer una aplicación de página única o *single-page application (SPA).*

En este tipo de aplicaciones web, todos los códigos de HTML, JavaScript y CSS se cargan de una vez y los recursos se obtienen dinámicamente, generalmente como respuesta a las acciones del usuario. Esta filosofía casa muy bien con la idea del bot pero requiere de un cliente a la

espera de peticiones, es decir, un servidor, por lo que debería de ser rechazada en favor de la página web clásica de HTML y CSS con JavaScript y AJAX. Sin embargo, existe una forma de facilitar el desarrollo utilizando un *framework*. Esto es compilándolo una vez desarrollado a un sitio web formado por tres ficheros, conocidos como *index.html*, *styles.css* y *app.js*. De esta forma no solo se facilita el desarrollo sino que se permite la reutilización o fácil migración a otros proyectos con servidor.

En lo que respecta al aspecto gráfico y siguiendo con la política de ofrecer soluciones modernas se ha tenido en cuenta la utilización de *Material Design*[38] sobre *Bootstrap 4* para no solo facilitar el desarrollo sino ofrecer un aspecto actual.

En conclusión, para el desarrollo se hará uso del framework Vue.js, compilando el código y utilizando el framework de Bootstrap 4 con adaptación a Material Design para ofrecer un aspecto moderno y un desarrollo sencillo y eficaz

3.5.2. Despliegue de la interfaz sin hacer uso de servidor dedicado

El reto a solucionar ahora es el desplegar una interfaz estática a través de un servicio de Amazon, debido a que es la forma más sencilla de servir la página web sin tener que hacer uso de un *host*.

Para ello se hará uso de *S3*, el servicio de almacenamiento de objetos de *AWS*, que no solo permite almacenar la página web sino que también permite exponerla a través a través de una *URL*. Para garantizar ambas cosas, será necesario crear un contenedor en el que se almacenará la información.

Resulta interesante recordar en este punto que para exponer el archivo no hará falta hacer uso de *API Gateway*, tal y como si hace falta para otros servicios o funciones, sino que el propio servicio de almacenamiento se encarga de gestionar el acceso público.

3.5.3. Control de la autenticación, autorización e identificación

Existen dos aproximaciones de autenticación y autorización que hay que analizar en este proyecto. Por una parte, aquella que permita el registro y la identificación de usuarios en el servicio *front-end* de las distintas aproximaciones; por otra parte, aquella que permita reducir y controlar el número de usuarios que ejecutan la lógica *serverless* de la aplicación.

Para el primer caso, se tendrá en cuenta únicamente el *front-end* que permite el acceso al bot. Este caso requiere que los usuarios que vayan a utilizarlo estén ya registrados en el sistema, permitiendo el uso de cuotas, de análisis de datos y de limitaciones de uso a los mismos.

Para garantizar esto, se plantearon dos situaciones. En la primera de ellas se decidió que los usuarios registrados se almacenen y gestionen de forma nativa en la aplicación. Sin embargo, siguiendo con la filosofía *serverless* y la explotación de servicios basados en la nube se planteó el uso de *Amazon Cognito* que permite gestionar de forma automatizada la autentificación y el

[38] Google LLC. (2019). *Material Design* - https://material.io/design/

registro de usuarios mediante la tecnología *OAuth*. Esta opción permite autenticar al usuario a través de otras plataformas, lo cual es una ventaja, pues no tienen que crear una cuenta explícitamente para nuestra aplicación.

Para el segundo caso, la autorización del uso de las funciones *serverless,* se planteó que debido a que estas solo van a ser ejecutadas desde otros servicios ya definidos (el bot en *Dialogflow*), no era necesario contar con un servicio de registro y inicio de sesión, sino que sería suficiente con un usuario y una contraseña definida que permitiese el acceso o no. Como consecuencia, se definirá para la arquitectura una función *lambda* de autentificación que comprobará si en los parámetros de la petición a otras funciones *lambda*, accedidas a través de *API Gateway*, se han incluido los tokens de acceso predefinidos para la comunicación.

3.5.4. Configuración de la herramienta de análisis de lenguaje natural

Para el análisis de la entrada del usuario se definirán los *intents* o intenciones. Cada *intent* correspondiente a una resolución posible de la interpretación de la petición del usuario. Estos se dividirán, a la vez, en dos ramas.

Por una parte, la rama social corresponderá a la comunicación que tiene que ver con el inicio de la conversación, el fin, la solicitud de interacciones posibles y el agradecimiento. Es decir, reduciéndolo a modo de ejemplos básicos, en aquellas comunicaciones que puedan significar lo mismo que *"Hola"*, *"Adiós"*, *"¿Qué puedes hacer?"* y *"Gracias"*.

Por otra parte, y tal y como se ha definido en los casos de uso, la comunicación correspondiente a la consulta meteorológica requerirá de los intents correspondientes para su interpretación, así como del uso de contextos para poder mantener aspectos como el lugar o la fecha previamente establecidos.

Finalmente, será necesario realizar un entrenamiento a base de interactuar con el bot simulando una conversación real para detectar qué oraciones y en qué situación deben de producir qué tipo de respuesta. Esa información debe de ser usada para reentrenar la herramienta hasta alcanzar un porcentaje de predicciones correctas lo suficientemente elevado.

3.5.5. Integración entre microservicios

Para la integración entre microservicios se hará uso de las herramientas de *fulfillment* facilitadas por *Dialogflow,* que permiten que ciertas funciones llamen a una *URL* con los parámetros seleccionados para obtener la respuesta necesaria.

La configuración de ésta funcionalidad es sencilla y solo requiere vincular los *intents* que requieran *fulfillment* con su función correspondiente. Además, existe una pestaña de carácter general que permite definir el *webhook* al *back-end*, siendo posible introducir *URL*, opciones de autenticación y cabeceras predefinidas.

3.5.6. Desarrollo de las funciones serverless

A la hora de hablar de las funciones como servicio es necesario tener claros tres puntos principales: cuántas y qué funciones habrá que implementar, en qué lenguaje estarán escritas y cómo van a ser desplegadas en el *cloud*.

Empezando por el principio, para saber qué funciones será necesario programar hace falta volver a los casos de uso para comprender qué interacciones requieren *fulfillment*. En este caso, existen cinco principales: consultar el estado meteorológico, como en *"¿Qué tiempo hace?"*; obtener la temperatura, como en *"¿Hará frío?"*; preguntar qué tipo de prendas serán necesarias, como en *"¿Voy a necesitar abrigo?"*; consultar si el tiempo es adecuado para hacer alguna actividad, como en *"¿Podré ir a la playa?"*, y obtener una respuesta condicional, como en *"¿Va a llover?"*. Además, hará falta una función principal que sea la que decida qué función secundaria ejecutar, en función de sus parámetros, y otra de autenticación para comprobar que el usuario es quien dice ser y tiene acceso al sistema.

Por otra parte, respecto al lenguaje utilizado, se plantea la utilización de lenguajes de *scripting* como *Python 3* y *Node.js* debido a la sencillez y efectividad que ofrecen al desarrollador y al soporte nativo por parte de *Amazon Web Services*. A lo largo del proyecto se utilizarán indistintamente ambos lenguajes para desarrollar las funciones necesarias.

Finalmente, para desplegar estas funciones de forma local se hará uso del *framework serverless.org*[39], que permitirá automatizar los despliegues de las mismas, facilitar su integración y hasta exponerlas directamente.

3.5.7. Exposición de las funciones de forma pública

Una vez desarrolladas las funciones *serverless,* éstas se *deberán* de exponer de modo que se permita su acceso público. Sin embargo, no se requiere una URL por cada una de ellas, sino que se puede exponer una sola y disponer a ésta de la capacidad de redirigir llamadas al resto.

Para la exposición, que se hará sobre *API Gateway*, aunque no será necesaria la configuración manual debido a que, al igual que en el apartado anterior, se hará uso del *framework serverless.org* para automatizar la tarea.

3.6. Solución propuesta

En la sección anterior se han detallado las posibles soluciones para cada uno de los problemas planteados. En este punto se comentarán cuáles han sido los elegidos para llevar a cabo en el desarrollo del proyecto.

[39] Serverless, Inc. (2019). *Serverless* - https://serverless.com/

- **Diseño de una interfaz estática.** Se ha optado por hacer uso de *Vue.js* para facilitar el desarrollo de la interfaz, que será compilada a *HTML*, *CSS* y *JavaScript* puros. Por otro lado, el estilo gráfico se basará en el paradigma de *Material Design.*
- **Despliegue de la interfaz sin hacer uso de servidor dedicado.** Para este paso se creará un bucket en Amazon S3 sobre el que se subirá el contenido estático y se ofrecerá acceso al mismo a través de una URL pública del mismo servicio.
- **Control de la autenticación, autorización e identificación.** Se ha elegido para la autenticación del *front-end* un sistema de identificación de terceros, haciendo uso de la tecnología *OAuth* de *Cognito* y, para la de los servicios *serverless*, una función de autenticación basada en usuario y contraseña de acceso.
- **Configuración de la herramienta de análisis de lenguaje natural.** Se analizarán y programarán las distintas intenciones posibles de usuario, separadas en aquellas sociales y aquellas relacionadas con la meteorología. Además, se realizará un entrenamiento para aumentar el porcentaje de comunicaciones con éxito.
- **Integración entre microservicios.** Se utilizarán las herramientas de fulfillment propias de Dialogflow, que permiten establecer un webhook al que enviar peticiones y del que recibir respuestas.
- **Desarrollo de las funciones serverless.** Se desarrollarán cinco funciones principales más otras dos necesarias en *Python 3* y *NodeJS*, indistintamente. Para el despliegue se hará uso del *framework serverless.org*.
- **Exposición de las funciones de forma pública.** Solo se expondrá una función de *Amazon Lambda* con *API Gateway*. Además, para ello se hará uso también del *framework serverless.org.*

4. Diseño de la solución

Tomando como punto de partida las diferentes soluciones elegidas en el capítulo anterior, se ha llevado a cabo el diseño de una solución global que integre todas las piezas definidas hasta ahora. Esta solución, que corresponde al diseño general de la aplicación, será analizada a lo largo de tres apartados: uno en el que se detallará la vista global de los componentes y cuál será la interacción entre ellos, otro en el que se profundizará en cada uno de ellos para ver cómo se ha logrado solucionar su reto particular y, finalmente, otro para realizar una reflexión acerca de la tecnología utilizada y de cómo ha sido explotada.

4.1. Arquitectura del sistema

La arquitectura del sistema estará formada por la integración de múltiples componentes o microservicios independientes entre sí a nivel de desarrollo. Frente al monolito clásico, ésta estructura permitirá gestionar más eficiente y versátilmente cada uno de los elementos, que solo tendrán que ofrecer y/o atacar a una *API*, sin tener en cuenta la lógica que haya detrás de la misma.

Otra de las ventajas de esta solución descentralizada es que permite el escalado de elementos puntuales en función de la carga que soporten. Por ejemplo, si solo existiese un cliente que realizase miles de consultas por segundo al bot, sería interesante disponer de una mayor concurrencia a nivel de *Dialogflow*, la herramienta de análisis de lenguaje natural elegida, o de las funciones de *AWS Lambda*, sin tener que rendirse a la necesidad de dedicar más recursos al *front-end* o al servicio de autenticación. Es necesario comprender que, pese a que se esté hablando de escalado, se está haciendo uso de servicios y no de máquinas, por lo que entiende éste como coste económico de uso y no como recursos o capacidad de cómputo.

Volviendo a la arquitectura, ésta está planteada de tal forma que permita al cliente seguir un procedimiento sencillo y transparente para él, accediendo a un sitio web y comunicándose con el bot sin tener que preocuparse de la integración. A nivel lógico, sin embargo, se entiende la interacción como una sucesión potencialmente infinita de cuatro casos en los que el usuario accede a la interfaz, se autentica, envía una consulta y obtiene una respuesta de ella, pudiendo repetir estos dos últimos pasos tantas veces como considere necesarios.

El primero de estos pasos, el acceso a la interfaz, pasa por acceder a la URL proporcionada por *Amazon S3* que da acceso a la página web estática alojada previamente. La descarga de este paquete de *HTML, CSS* y *JavaScript* se carga en el navegador del cliente y es la que da paso a la interacción gráfica por parte del usuario, así como a la autenticación y, por lo tanto, al uso.

El segundo de los pasos, la autenticación, se realiza gracias al servicio *Amazon Cognito*, que se encarga de mantener la *pool* de usuarios y de permitir el registro y el inicio de sesión gracias a la integración con herramientas de terceros, tal y como se explicará más adelante. La

comunicación en este punto se realiza gracias a llamadas a la *API* de *Cognito* desde el mismo *front-end*.

Una vez realizado el inicio de sesión, ya se puede interactuar con el bot de la interfaz, que no es más que un cascarón vacío que redirige todas las llamadas a la *API* de *Dialogflow*. Estas llamadas, realizadas sobre el protocolo *HTTP*, son asíncronas, es decir, no tienen por qué ofrecer una respuesta inmediata, sino que pueden demorarse debido a no solo el propio procesamiento de la herramienta de *Google*, sino al retardo añadido por el *fullfilment*. Este método de obtención de información, explicado anteriormente, no es más que otra petición *HTTP asíncrona* al *back-end*. En este paso, además, se realiza la segunda autenticación, puesto que el bot desplegado en *Dialogflow* se identifica con usuario y contraseña como cliente válido de la próxima *API*.

El último de los pasos corresponde a las funciones *serverless* de *AWS Lambda*, encargadas, en este caso, de recolectar información relacionada con la meteorología. El problema de estas funciones es que no pueden exponerse públicamente por ellas mismas, sino que es necesaria la creación de una *API* para facilitar su acceso. Ése es el papel de *Amazon API Gateway*, que a través de la exposición de una *API Rest* permite redirigir las peticiones y las respuestas desde y hacia *AWS Lambda*. Una vez solucionadas éstas, la información vuelve sobre sus pasos hacia *Dialogflow* y posteriormente hacia el navegador para ofrecerse al usuario, que puede continuar con más interacciones o dar por terminada la comunicación.

A continuación se muestra en una figura la estructura presentada a lo largo de los últimos apartados, correspondiendo cada uno de los puntos a uno de los apartados presentados anteriormente. Es interesante recordar que, mientras que los dos caminos superiores son síncronos, el tercero no lo es, y que no todas las llamadas a *Dialogflow* tienen por qué terminar siendo invocaciones a *AWS Lambda*.

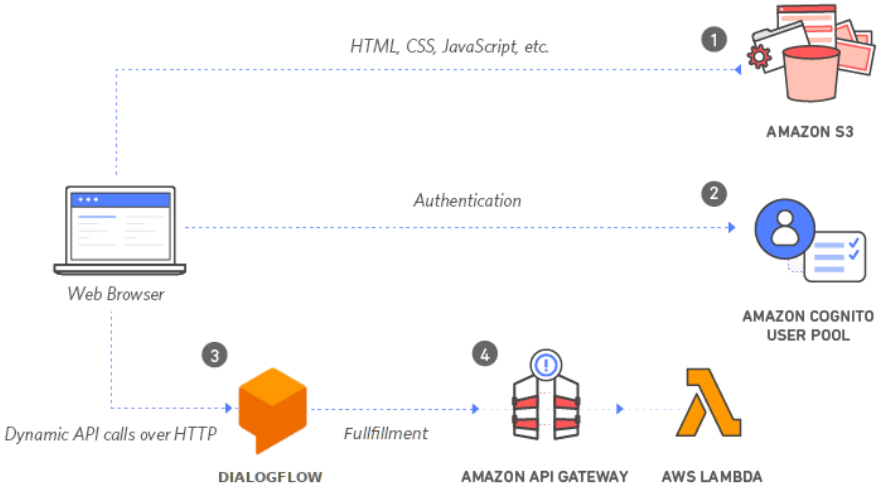

Figura 3 - Esquema general de la arquitectura del sistema

4.2. Diseño detallado

Una vez vista la arquitectura a nivel general que tendrá la plataforma, se pasa a ver cómo están diseñados a nivel interno cada uno de los componentes previamente mostrados. Se añade además un apartado correspondiente a *DialogFlow* debido a que también es interesante conocer cómo se estructurarán y comunicarán entre sí sus *intents*. Se verá, por lo tanto, cómo ha sido planeado el *front-end*, cómo se integrará el sistema de autenticación, cómo será la integración y qué patrón seguirán las funciones desplegadas sobre *AWS Lambda*, además de lo ya mencionado respecto a la herramienta de análisis de lenguaje natural. Cabe destacar que aquí solo se profundizará en el diseño y no en la implementación, la cuál quedará detallada a lo largo del capítulo 5.

4.2.1. Interfaz

Empezando por el *front-end*, es necesario definir que éste se entiende como un módulo específico y no como una web completa, de modo que debe de poder ser importado y utilizado en distintos proyectos. Esto garantiza poder separar el desarrollo del bot del de la página web, permitiendo la utilización de la misma estructura en distintos sitios. Es decir, que con la realización de ligeros cambios como la *API* de *Dialogflow* a la que se conecta, la misma estructura pueda ser utilizada en webs de distinta índole.

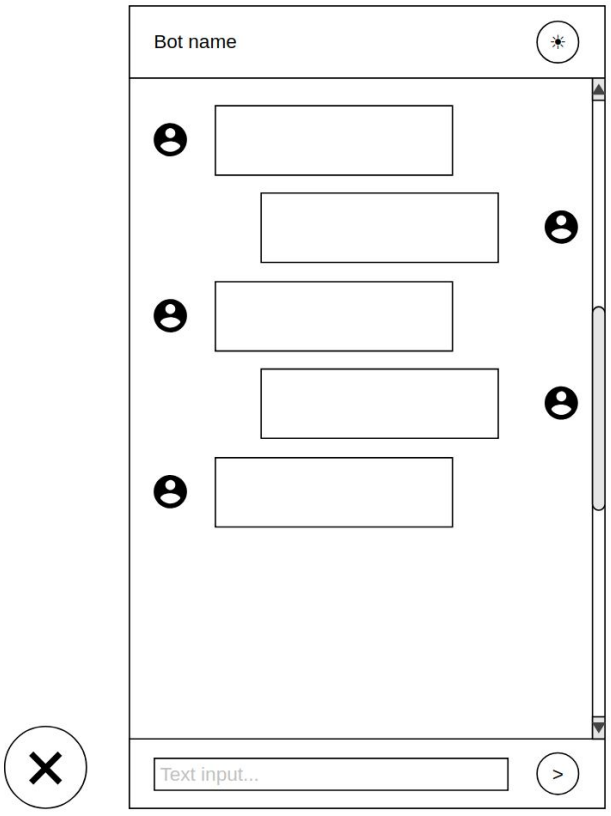

Figura X - *Mockup* del módulo correspondiente al bot en la interfaz

Para garantizar éste diseño como módulo, se hará uso del botón flotante característico de *Material Design*, el cuál mostrará un panel dedicado al chat en el que realizar la interacción con el bot. Este panel seguirá el formato típico de los chats modernos, con una estructura de conversaciones en burbujas en el que el nombre del bot aparece en la parte superior, las conversaciones en medio, con opción a *scroll*, y el área en la que escribir el nuevo mensaje en la parte inferior. Como añadido, el bot contará con la opción de cambiar entre modo día y modo noche, ofreciendo un diseño más oscuro y menos ofensivo para los usuarios de teléfonos móviles.

Otra de las características del diseño del chatbot es que, para facilitar la interacción de usuarios con distintas configuraciones de pantalla se ha optado por ofrecer un diseño *responsive*, que se adapte a las necesidades del usuario independientemente de estar conectándose a la web en un portátil, en una tablet o a través de un equipo conectado a un monitor ultra panorámico. En el caso específico de los móviles, se ha planteado una configuración especial que permite que el chat ocupe toda la pantalla en vez de parte de la misma, para facilitar así la escritura en pantallas pequeñas.

Finalmente, y para el caso concreto de la aplicación del proyecto, se ha planteado utilizar de fondo una página web ajena al mismo, debido a que de este modo se visualiza mejor la conexión entre el módulo y el resto de la web. Para tal fin, se ha elegido el sitio de *World Weather Online* [40], de cuya API se extraen, además, los datos meteorológicos ofrecidos, presentando simbiosis con el resto del proyecto y un caso de uso real del mismo.

4.2.2. Sistema de autenticación [TBD]

TBD

Acceso a S3

Acceso al bot (COGNITO)

Bot - DIalogflow (DIALOGFLOW POLICIES Y TAL)

Acceso de Dialogflow al back-end (LAMBDA)

4.2.3. Dialogflow

Analizando los requisitos y los casos de uso de la aplicación, se ha llegado a la conclusión de que son necesarias cuatro intenciones sociales y cinco relacionadas con la meteorología, de las cuáles derivan otras cinco. Podemos distinguir, por lo tanto, tres tipos de intenciones: aquellas sociales, aquellas simples relacionadas con la meteorología y aquellas que, siendo también relacionadas con la meteorología, son derivadas y, por tanto, más complejas.

[40]World Weather Online. (2019). *World Weather Online* - https://www.worldweatheronline.com

Empezando por las primeras, éstas se definen como aquellas que responden a las interacciones que puedan significar lo mismo que *"Hola"*, *"Adiós"*, *"¿Qué puedes hacer?"* y *"Gracias"*. Estas interacciones son características por dos motivos. El primero de ellos es que no tienen argumentos variables, es decir, que no hay que recoger datos específicos en cada interacción. El segundo de ellos es que la respuesta no depende de ningún estado externo a la aplicación, por lo que puede ser extraída de un *pool* de respuestas válidas previamente predefinidas. A modo de ejemplo, toda interacción que se interprete como *"Hola"* tendrá una respuesta pseudoaleatoria de entre las definidas en *["¡Hola!", "¡Hey!", "¡Buenos días!"]*.

La segunda de las categorías, correspondiente a las funciones meteorológicas simples, es muy distinta a la categoría anterior, puesto que discrepa en ambas características: Es necesario recoger información y la respuesta depende de un estado variable, por lo que no puede ser previamente definida. Para solucionar el primer problema se crearán *entidades*, una especie de diccionarios que permiten agrupar palabras o expresiones que sean valores posibles para datos que estamos intentando recoger. Por ejemplo, toda pregunta relacionada con la vestimenta, como *"¿Voy a necesitar abrigo esta noche?"*, necesita extraer el complemento, en este caso *abrigo*, por lo que se requiere definir esos valores posibles. Además, esto permite agrupar valores, como *"sudadera", "suéter"*, *"bufanda"* y "*chaqueta"* dentro de *"vestimenta de invierno"*, permitiendo aligerar carga en el el desarrollo y la ejecución *back-end* al tener que trabajar con menos valores posibles.

El segundo de los problemas está relacionado con que la respuesta depende de un estado variable y, como añadido, de las variables de entrada utilizadas. Para solucionar este problema se debe de hacer uso del *fulfillment*, una funcionalidad de *Dialogflow* que permite conectarse a una API externa y redirigir las peticiones a ella, obteniendo de forma asíncrona una respuesta para las mismas. Existe una limitación en este aspecto y es que actualmente la herramienta de *Google* no permite un *fulfillment* por *intent*, sino que debe de haber un servicio web al que apuntar compartido por todo el bot y decidir, en cada función, si se quiere hacer uso de él o no. Esta característica implica que, para el *back-end*, no se puede hacer uso de una API REST típica en la que, por ejemplo, existen los paths */v1/books* y */v1/films* para obtener libros y películas, respectivamente, sino que debe de haber un path único y procesar todas las peticiones a través de él.

Finalmente, la tercera de las categorías es aquella que tiene que ver con las funciones derivadas. Estas funciones son aquellas que añaden información a consultas anteriores, manteniendo parte del contexto presente en las mismas. Un ejemplo de esto sería la petición *"¿Y en Madrid?"* tras conocer la respuesta a "*¿Va a llover mañana en Barcelona?"*. Estas funciones requieren de un contexto finito, que puede ser especificado como activo durante un número de interacciones más. En este caso se permitirá una derivada hasta en la segunda interacción, perdiendo el contexto posteriormente a ésta.

Los contextos, sin embargo, no solo sirven para saber de qué tema se está hablando, sino que se puede profundizar en ellos para extraer información adicional. De este modo, al preguntar *"¿Y en Madrid?"* se mantendrá la fecha *mañana*, la condición *llover* y el tipo de pregunta *condicional*, sobreescribiendo únicamente el lugar. Esto se conseguirá utilizando el contexto no como simples variables de estado, sino como *JSON* completos que tengan almacenados los datos de conversaciones anteriores.

4.2.4. Integración y APIs

En lo que respecta a la agrupación del código, existen dos ramas principales en la arquitectura del software: una, más clásica, que propone mantener todo el código de los distintos servicios agrupados en una aplicación y otra que propone mantenerlo en aplicaciones tan independientes como sea posible, permitiendo la reutilización y el desarrollo independiente. Esta aproximación basada en microservicios, seguida a lo largo del proyecto, requiere que cada uno de ellos exponga o se conecte a una API para la integración global, por lo que éstas deben de estar bien definidas desde el principio.

Empezando por *Dialogflow*, puesto que al *front-end* se accederá a través de una URL definida, será necesario establecer cómo se comunicará el cliente con este servicio. Para ello, se hará uso de la *API* ofrecida por la misma herramienta. Esta *API*, actualmente en su versión *V2*, hace uso de *Google Cloud*, siendo necesario crear un *Gateway* para el acceso a la misma y utilizar los credenciales generados, en formato *JSON*, para identificar el servicio y autenticar al usuario. Una vez realizada la conexión, ésta sólo tendrá permisos para enviar y recibir respuestas, evitando todas las opciones de administración y mantenimiento del bot.

Como consecuencia, sólo será necesario hacer uso de las llamadas de autenticación, que devuelve un *token* para ser utilizado en la conexión, y de *detectIntent*, que utilizará el mismo *token* para realizar una llamada POST al agente y enviarle el mensaje que debe de ser respondido. Una característica adicional a tener en cuenta es que *Dialogflow* necesita de un identificador de sesión para mantener estados independientes entre sus usuarios, por lo que cada nueva conversación deberá de generar un identificador pseudoaleatorio que mantendrá hasta el final de la conversación.

La otra conexión a realizar será la que suceda entre la herramienta de análisis de lenguaje natural y el *back-end* en el caso de que sea necesario trabajar con datos meteorológicos y proceder al *fulfillment*. En este caso, tal y como se ha explicado en el apartado **4.2.3**, existe el problema de que *Dialogflow* solo permite atacar a un servicio web por agente, por lo que no se podrá crear una estructura *RESTful* basada en identificadores, sino que únicamente se expondrá un punto que permitirá el acceso a todos los demás.

Para realizar esta exposición se utilizará, por lo tanto, un servicio *RESTful* con un único método, */weather*, que, ante una llamada *POST,* se encargará de autenticar al usuario y de invocar a la función *Lambda* principal, conocida como *ptolomeo-weather*.

4.2.5. Funciones serverless

Tal y como se ha explicado en los apartados **4.2.3** y **4.2.4**, *Dialogflow* no permite un *fulfillment* por *intent*, sino que debe de haber un servicio web al que apuntar compartido por todo el bot y decidir, en cada función, si se quiere hacer uso de él o no. Esta característica implica que, para el *back-end*, no se puede hacer uso de una API REST típica en la que, por ejemplo, existen los paths */v1/books* y */v1/films* para obtener libros y películas, respectivamente, sino que debe de haber un *path* único y procesar todas las peticiones a través de él.

Esta característica obliga a que exista una función de *Amazon Lambda* que invoque a las demás o, en su defecto, que tenga el código de todas las funciones meteorológicas y lo que se exponga sea un validador a modo de *switch* que compruebe si la entrada es válida y redirija la petición a cada uno de los posibles módulos. El diseño se ha realizado siguiendo el segundo patrón, existiendo un script llamado *app.py*, que será el que se ejecuta con la función *Lambda*, el cuál validará la entrada y la redirigirá al método que le corresponda en función del tipo de petición, establecido en los parámetros de la misma. Es interesante comprender que solo será necesario desarrollar cinco métodos principales para la meteorología, puesto que las funciones derivadas llamarán a los mismos métodos sólo que con nuevos parámetros.

Además de esta función, se utilizará otra para la autenticación. Esta función se ejecutará automáticamente cuando se ejecute la llamada *POST* a */weather* y comprobará que los parámetros de autenticación puestos en la cabecera son válidos, tal y como han sido definidos en el apartado **4.2.2**.

4.3. Tecnología utilizada

Se ha planteado la utilización de diversos servicios, herramientas y tecnologías a lo largo del desarrollo del proyecto. Por este motivo, en este apartado se va a realizar un análisis de las más importantes, separándolas en tres aspectos: servicios de *Amazon Web Services*, el intérprete de lenguaje natural y las herramientas y lenguajes utilizados para el desarrollo de la solución.

4.3.1. Servicios de AWS [TBD]

TBD

4.3.1.1. Amazon Identity and Access Management

TBD

4.3.1.1. Amazon Simple Storage Service

TBD

4.3.1.2. Amazon Cognito

TBD

4.3.1.3. Amazon Lambda

TBD

4.3.1.4. Amazon API Gateway

TBD

4.3.2. Intérprete de lenguaje natural [TBD]

TBD

4.3.3. Herramientas y Lenguajes Utilizados [TBD]

TBD

4.3.3.1. Vue.js

TBD

4.3.3.2. Node.js

TBD

4.3.3.3. Python

TBD

4.3.3.4. Serverless.org

TBD

5. Desarrollo de la solución propuesta

Una vez finalizado el diseño, se ha procedido a la implementación de cada una de las partes o componentes especificados. Estos se han planteado de la misma forma y en el mismo orden que en el diseño, profundizando ahora en cómo se han conseguido solventar los problemas y cubrir las necesidades asociadas al cumplimiento de los objetivos. A continuación se expone, en el orden presentado, cuál ha sido el proceso de desarrollo de cada uno de los mismos.

5.1. Interfaz [TBD]

TBD

5.2. Sistema de autenticación [TBD]

TBD

5.3. Dialogflow [TBD]

TBD

5.4. Integración y APIs [TBD]

TBD

5.5. Funciones serverless [TBD]

TBD

6. Implantación

Para hacer uso de la aplicación y llevar a cabo su despliegue, se deben seguir una serie de pasos, ya que hay que realizar las conexiones de forma correcta, pues si no los componentes no se podrán comunicar entre ellos.

En primer lugar, para desplegar la lógica de la aplicación, hay que crear una cuenta de *Amazon Web Services* y, con sus credenciales y la aplicación *serverless.org* instalada localmente, desplegar el servicios *serverless* de *Ptolomeo*, disponible públicamente en *GitHub*, que automáticamente se encargarán de configurar los servicios y las configuraciones necesarias. Es necesario, sin embargo, crear el archivo *lmbda/authorizer/index.js*, a través de la copia de de *lmbda/authorizer/original.index.js*, definiendo el usuario y la contraseña de acceso a la aplicación, justo bajo el comentario *// TODO SET USERNAME AND PASSWORD HERE*. Esto se mantiene así debido a que facilita que los usuarios no utilicen variables por defecto y, por lo tanto, a que sus despliegues sean menos vulnerables.

Una vez desplegado todo sobre AWS, se requiere conectarse a la interfaz para comprobar la *URL* que han sido definida por Amazon para el servicio *POST* de *API Gateway*. Estas direcciones permiten el acceso externo a la lógica de la aplicación, siempre que la petición cuente con los tokens definidos anteriormente.

Posteriormente, para la utilización de *Dialogflow* es necesario el registro en la plataforma mediante el uso de una cuenta de *Google*, ya que ésta también da acceso a *Google Cloud*. Una vez finalizado el registro, será suficiente con importar las configuraciones de *Ptolomeo,* disponibles también *GitHub*, y completar el *fulfillment* de ambos con la *URL* ofrecida por *API Gateway* y con el usuario y la contraseña definidos para la autenticación, tal y como se ha definido en el párrafo anterior.

Con esto, ambos bots ya estarían funcionales para realizar pruebas desde la interfaz de *Dialogflow*, pero todavía quedaría dar acceso al *front-end* a los usuarios. Para esto, sería necesario ejecutar el código de la última función de *serverless.org,* también disponible en *GitHub,* que automáticamente puebla *S3* con los archivos necesarios y define la función *lambda* para su acceso. La URL definida por *API Gateway,* ahora sí, permite el acceso a los usuarios finales.

7. Pruebas y rendimiento [TBD]

TBD

7.1. Validación del sistema [TBD]

TBD

7.2. Pruebas de carga [TBD]

TBD

8. Conclusiones y trabajos futuros

Una vez terminado el trabajo, es el momento de reflexionar y tratar de analizar y descubrir no sólo si éste se ha realizado correctamente y si se han complido los objetivos, sino también cómo puede ser continuado, qué he aprendido y qué puedo ofrecer de él.

8.1. Conclusiones

Para descubrir si el trabajo ha sido gestionado correctamente, así como si se ha aprovechado los obtenidos que se deseaban de él, hay que que fijarse en dos puntos: si se han cumplido los objetivos o no y qué conocimientos se han adquirido a lo largo de este proyecto. Estas conclusiones se basan, por lo tanto, en recuperar los objetivos formalizados del proyecto y analizarlos individualmente para descubrir si han sido alcanzados con éxito.

8.1.1. Cumplimiento de los objetivos [TBD]

TBD

8.1.2. Conocimientos adquiridos

A lo largo del desarrollo de este proyecto he trabajado con herramientas y tecnologías que desconocía o en las que no había profundizado lo suficiente. Por lo tanto, y gracias a los requisitos y las necesidades del trabajo, he podido encontrar problemas y enfrentarme a ellos, buscando las mejores soluciones y descubriendo, en el camino, un conocimiento autodidacta que ha ido mucho más allá de lo simplemente planteado.

Por otro lado, el hecho de querer distribuir el código en su plenitud a través de una licencia de código libre ha supuesto que arraigue en mí estándares de calidad autoimpuestos, buscando la mejor solución posible para cada uno de los problemas. Así también me he centrado en ofrecer una documentación suficiente para las herramientas del proyecto, así como facilidades para la introducción de nuevos participantes al mismo, facilitando de este modo el acercamiento a trabajos futuros.

Finalmente, considero que este trabajo me ha hecho crecer a nivel personal, debido a dedicar tantas horas de esfuerzo a un proyecto que sirve como resumen y zénit de todo el trabajo realizado y todo lo aprendido en los estudios de Grado, tal y como explicaré en el apartado **8.4**.

Para concluir, mostraré aquellos conocimientos adquiridos que más me gustaría destacar, bien por el impacto total en el proyecto como por lo curiosos o interesantes que me han parecido a nivel personal:

- Funcionamiento y utilización de las herramientas de interpretación de lenguaje natural, especialmente *Dialogflow*.

- Conocimiento y análisis de los distintos proveedores de servicios en la nube, ya que solo conocía, y de forma vaga, *Amazon Web Services*, y he podido explorar no solo hacia los adentros de esta plataforma sino también descubrir que la hace diferente en relación a la de los otros proveedores.
- Profundización en el paradigma de las *Funciones como Servicio*, descubriendo de forma práctica *Amazon Lambda*, así como trucos que permiten eliminar completamente el servidor como desplegar el *front-end* como un archivo de *S3*.
- Uso de un *framework front-end* basado en *JavaScript* como *Vue.js*, que me ha permitido acercarme al desarrollo web de una nueva forma y con nuevas facilidades y retos que cubrir.
- Conocimiento de la tecnología *Auth0*, así como su uso e integración en la aplicación.
- Explotación de herramientas para el análisis de funcionamiento ante grandes cargas en un servicio basado en la web como *Artillery.io*.
- Utilización y familiarización con LaTeX[41].

Este trabajo ha sido, por lo tanto, muy beneficioso y enriquecedor desde mi punto de vista, por todo lo que he aprendido de él.

8.2. Trabajos futuros [TBD]

TBD

8.3. Relación del trabajo desarrollado con los estudios cursados

Pese a que ninguna de las asignaturas troncales del Grado de Ingeniería Informática tiene una relación directa con la implementación de bots conversacionales ni con la explotación de servicios basados en la nube, sí que hay muchas asignaturas que son necesarias y que me han ayuda en gran medida a alcanzar los objetivos planteados para el proyecto. A continuación, muestro un pequeño resumen de las asignaturas que he considerado más relevantes:

- **Tecnología de Sistemas de información en la Red:** Pese a que en ninguna asignatura del grado se profundiza, esta rama es la que permite obtener un acercamiento mayor al de los sistemas distribuidos y escalables. Es ahí donde descubrí los conceptos de *Infrastructure as a Service*, *Platform as a Service* y *Software as a Service*, así como los principios de *cloud*, las arquitecturas escalables y los servicios *serverless*.
- **Sistemas y Servicios en Red:** A lo largo del Grado se aborda desde distintos puntos aquello relacionado con la red y los servicios desplegados en ella. Estas asignaturas han sido de gran utilidad para comprender conceptos relacionados con qué interfaces y protocolos utilizar, cómo manejar un servicio en línea y cómo asegurarse de que sea escalable y seguro.

41

- **Programación:** Las asignaturas de programación, troncales a todo el Grado, me han permitido conocer distintos lenguajes y paradigmas de programación. Este conocimiento me ha permitido elegir de forma inteligente qué utilizar y cómo estructurar mi código.
- **Desarrollo Web:** Pese a que no forma parte del troncal del proyecto, se ha desarrollado una interfaz web que permite el acceso y la interacción con los bots implementados. Para ello, ha sido necesario hacer uso del conocimiento adquirido en las asignaturas de diseño de interfaces y diseño web, ofreciendo así una interfaz moderna, eficaz y efectiva desde el punto de vista del usuario.
- **Diseño, Configuración y Evaluación de Sistemas Informáticos:** Una vez terminado el diseño, he realizado pruebas de rendimiento para comprobar no solo la eficiencia de mi sistema sino la funcionalidad de una arquitectura *serverless* frente a una que utiliza un servidor propio. Para ello, ha sido de utilidad conocer herramientas de evaluación así como métodos y protocolos a seguir.
- **Sistemas Inteligentes:** A la hora de decidir qué tipo de intérprete de lenguaje natural utilizar para el proyecto, ha sido muy interesante contar con la base teórica obtenida en las asignaturas de esta rama. Gracias a ello, he podido distinguir entre sistemas expertos, machine learning, árboles de decisiones y demás herramientas, eligiendo aquella más útil y viable para el proyecto.
- **Gestión de Proyectos:** Pese a que no se ve reflejado directamente en el resultado, contar con una base en esta rama me ha permitiendo saber gestionar mucho mejor el tiempo, trabajar con hitos, plazos y entregas y obtener resultados claros a través de la separación del trabajo en distintos retos.

En conclusión, ha sido muy importante para mí poder contar con la base obtenida a lo largo de los estudios de Grado para la realización de este trabajo. Gracias a todo lo que he aprendido, mencionado o no, y a lo que se me ha facilitado desde la Universidad, he podido obtener los resultados deseados y aprender en el proceso.

8.4. Disponibilidad y Licencia

Todos el código relacionado con el proyecto está disponible en *GitHub* como parte de la organización *[INSERTAR]*, disponible en [INSERTAR]. Como consecuencia, éste es un proyecto de código abierto distribuido bajo la licencia *Apache 2.0* [36], la cual permite uso comercial, modificaciones, distribución y uso privado. Los derechos de autor deben conservarse tanto en el código fuente como en los binarios.

9. Glosario [TBD]

TBD

10. Referencias [TBD]

[SIRI] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.706.5567&rep=rep1&type=pdf

https://www.sciencedirect.com/science/article/pii/S026240791162647X

[alan turing]

http://www.cs.colorado.edu/~martin/SLP/Updates/1.pdf

TURING TEST

[APACHE2]

[AMAZONLEX]

https://en.wikipedia.org/wiki/Amazon_Lex

[IBMWATSON]

[DIALOGFLOW]

https://discover.bot/bot-talk/beginners-guide-bots/popular-chatbot-frameworks/

https://miningbusinessdata.com/dialogflow-vs-lex-vs-luis-vs-watson-vs-chatfuel/

https://www.slideshare.net/KonstantinSavenkov/nlu-intent-detection-benchmark-by-intento-august-2017

https://www.simform.com/aws-lambda-vs-azure-functions-vs-google-functions/

https://thehftguy.com/2016/11/18/google-cloud-is-50-cheaper-than-aws/

https://www.datamation.com/cloud-computing/aws-vs-azure-vs-google-cloud-comparison.html

https://medium.freecodecamp.org/how-to-create-a-vue-js-app-using-single-file-components-without-the-cli-7e73e5b8244f