



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA



Escola Tècnica
Superior d'Enginyeria
Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Implementación de un chatbot mediante 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 *Lambda* 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

Resum

El treball realitzat ha consistit en el disseny i la implementació d'un bot conversacional seguint patrons propis de l'arquitectura *serverless*. Per a això, s'ha fet ús d'eines de computació en el núvol proporcionades per *Amazon Web Services*, així com de l'entorn de desenvolupament *Dialogflow*. *Dialogflow* és una suite executada sobre *Google Cloud* que permet facilitar la interacció humà-ordinador oferint un desenvolupament simple i eficaç amb el qual interpretar el llenguatge natural. El *back-end* de l'aplicació, un consultor meteorològic, ha estat dissenyat per a ser executat completament sobre el núvol d'*Amazon*. El codi, majorment escrit en *Python*, encara que també en *Javascript*, és executat a través de funcions de *Lambda* i invocat gràcies a *API Gateway*. D'altra banda, el *front-end*, desenvolupat amb l'ajuda del *framework Vue.js*, es troba emmagatzemat en *S3* i permet l'autenticació mitjançant *Cognito*. S'ha obtingut com a resultat del treball una aplicació que permet aprofundir en la creació de bots conversacionals, així com explotar l'ús d'arquitectures sense servidor per obtenir resultats eficients a baix cost.

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

Abstract

The work carried out has consisted in the design and implementation of a conversational bot following the proper patterns of the serverless architecture. To do this, cloud computing tools provided by *Amazon Web Services* have been used, as well as the *Dialogflow* development environment. *Dialogflow* is a suite executed on *Google Cloud* that allows to facilitate the human-computer interaction offering a simple and effective development with which to interpret the natural language. The application's back-end, a weather consultant, has been designed to be fully executed on the *Amazon* cloud. The code, mostly written in *Python*, but also in *Javascript*, is executed through *Lambda* functions and invoked thanks to the *Gateway API*. On the other hand, the front-end, developed using the *Vue.js* framework, is stored in *S3* and allows authentication through *Cognito*. As a result of the work, it has been obtained an application that allows deepening on the creation of conversational bots, as well as exploiting the use of serverless architectures to obtain efficient results at low cost.

Key words: cloud computing, chatbot, serverless, aws, dialogflow, python

Índice general

| | |
|-------------------|-----|
| Índice general | V |
| Índice de figuras | VII |

| | | |
|----------|---|-----------|
| 1 | Introducción | 1 |
| 1.1 | Motivación | 2 |
| 1.2 | Objetivos | 3 |
| 1.3 | Impacto esperado | 3 |
| 1.4 | Estructura de la memoria | 4 |
| 2 | Estado del arte | 5 |
| 2.1 | Análisis de herramientas para la interpretación de lenguaje natural | 5 |
| 2.1.1 | <i>Amazon Lex</i> | 6 |
| 2.1.2 | <i>Google Dialogflow</i> | 7 |
| 2.1.3 | <i>IBM Watson Assistant</i> | 8 |
| 2.1.4 | <i>Microsoft LUIS</i> | 9 |
| 2.1.5 | <i>Wit.AI</i> | 9 |
| 2.2 | Análisis de los principales proveedores de servicios en la nube | 10 |
| 2.2.1 | <i>Amazon Web Services</i> | 11 |
| 2.2.2 | <i>Microsoft Azure</i> | 12 |
| 2.2.3 | <i>Google Cloud Platform</i> | 12 |
| 2.3 | Crítica al estado del arte | 14 |
| 2.4 | Propuesta | 14 |
| 3 | Análisis del problema | 15 |
| 3.1 | Análisis de los distintos retos | 15 |
| 3.1.1 | El <i>front-end</i> | 15 |
| 3.1.2 | La interpretación de lenguaje natural | 16 |
| 3.1.3 | El <i>back-end</i> | 16 |
| 3.2 | Análisis de la seguridad | 17 |
| 3.3 | Formalización del problema | 17 |
| 3.4 | Casos de uso | 18 |
| 3.5 | Identificación y análisis de soluciones posibles | 20 |
| 3.5.1 | Diseño de una interfaz estática | 20 |
| 3.5.2 | Despliegue de la interfaz sin hacer uso de servidor dedicado | 20 |
| 3.5.3 | Control de la autenticación, autorización e identificación | 21 |
| 3.5.4 | Configuración de la herramienta de análisis de lenguaje natural | 21 |
| 3.5.5 | Integración entre microservicios | 22 |
| 3.5.6 | Desarrollo de las funciones <i>serverless</i> | 22 |
| 3.5.7 | Exposición de las funciones de forma pública | 22 |
| 3.6 | Solución propuesta | 22 |
| 4 | Diseño de la solución | 25 |
| 4.1 | Arquitectura del sistema | 25 |
| 4.2 | Diseño detallado | 27 |
| 4.2.1 | Interfaz | 28 |

| | | |
|----------|---|-----------|
| 4.2.2 | Sistema de autenticación | 29 |
| 4.2.3 | <i>Dialogflow</i> | 29 |
| 4.2.4 | <i>Integración y APIs</i> | 31 |
| 4.2.5 | <i>Funciones serverless</i> | 31 |
| 4.3 | Tecnología utilizada | 32 |
| 4.3.1 | Servicios de <i>AWS</i> | 32 |
| 4.3.2 | Herramientas y lenguajes utilizados | 33 |
| 5 | Desarrollo de la solución propuesta | 35 |
| 5.1 | <i>Funciones serverless</i> | 35 |
| 5.2 | <i>Integración y APIs</i> | 36 |
| 5.3 | <i>Dialogflow</i> | 38 |
| 5.4 | Sistema de autenticación | 40 |
| 5.5 | Interfaz | 42 |
| 6 | Implantación | 47 |
| 6.1 | Despliegue de la lógica de la aplicación | 47 |
| 6.2 | Configuración del intérprete de lenguaje natural | 48 |
| 6.3 | Preparación del servicio de autenticación | 49 |
| 6.4 | Exposición del <i>front-end</i> de forma pública | 50 |
| 7 | Pruebas y rendimiento | 51 |
| 7.1 | Validación del sistema | 51 |
| 7.2 | Pruebas de carga | 54 |
| 8 | Conclusiones y trabajos futuros | 57 |
| 8.1 | Conclusiones | 57 |
| 8.1.1 | Cumplimiento de los objetivos | 57 |
| 8.1.2 | Conocimientos adquiridos | 61 |
| 8.2 | Trabajos futuros | 62 |
| 8.3 | Relación del trabajo desarrollado con los estudios cursados | 63 |
| 8.4 | Disponibilidad y licencia | 64 |
| | Glosario | 67 |
| | Bibliografía | 71 |

Índice de figuras

| | | |
|------|--|----|
| 2.1 | Adopción de los distintos proveedores de <i>cloud</i> público según <i>Flexera</i> . . . | 11 |
| 2.2 | Comparación de precios mínimos entre <i>Amazon Web Services EC2</i> y <i>Google Cloud Engine</i> | 13 |
| 3.1 | Diagrama <i>UML</i> correspondiente a las acciones que se deberían de poder realizar en el sistema. | 19 |
| 3.2 | Diagrama <i>UML</i> con profundización en las actividades que puede realizar el usuario. | 19 |
| 4.1 | Diagrama de flujo de la aplicación. | 26 |
| 4.2 | Esquema general de la arquitectura del sistema. | 27 |
| 4.3 | <i>Mockup</i> del módulo correspondiente al bot en la interfaz. | 28 |
| 4.4 | Detalle del <i>fulfillment</i> en <i>Dialogflow</i> donde se aprecian los campos <i>username</i> y <i>password</i> | 30 |
| 5.1 | Detalle del <i>YAML</i> utilizado para el despliegue de las funciones como servicio. | 37 |
| 5.2 | Detalle del campo <i>events</i> en <i>YAML</i> utilizado para el despliegue de <i>ptolomeo-weather</i> | 37 |
| 5.3 | Detalle de la configuración de <i>Action and parameters</i> en el <i>intent weather.outfit</i> | 39 |
| 5.4 | Detalle de la configuración de <i>Action and parameters</i> en el <i>intent</i> derivado. | 40 |
| 5.5 | Detalle de la vista de Inicio de Sesión en el <i>front-end</i> | 41 |
| 5.6 | Detalle de la vista de Registro en el <i>front-end</i> | 41 |
| 5.7 | Detalle del <i>JSON</i> utilizado para la autorización de la interfaz sobre <i>Dialogflow</i> | 42 |
| 5.8 | Panel flotante en el que se realiza la conversación con el bot en el que se muestra el mensaje por defecto. | 43 |
| 5.9 | Módulo de interacción con el bot en una pantalla de reducidas dimensiones. | 44 |
| 5.10 | Módulo de interacción con el bot en un teléfono móvil. | 44 |
| 5.11 | Interfaz web del bot en la que se está manteniendo una conversación con el mismo. | 45 |
| 6.1 | Ejemplo de configuración de los credenciales de <i>AWS</i> en <i>./aws/credentials</i> | 48 |
| 6.2 | Ejemplo de configuración de <i>AWS</i> en <i>./aws/config</i> | 48 |
| 7.1 | Resultados de la ejecución de un test sobre <i>ptolomeo-weather</i> en <i>AWS Lambda</i> | 52 |
| 7.2 | Resultado de una de las pruebas de validación del agente en el <i>front-end</i> | 53 |
| 7.3 | Resultado de una prueba en la que se intenta el registro con datos incorrectos o inválidos | 53 |
| 7.4 | Recta de carga correspondiente a las peticiones por segundo a lo largo del test. | 55 |
| 7.5 | Evolución de los valores de latencia obtenidos lo largo del proceso de test. | 55 |
| 7.6 | Datos obtenidos por <i>CloudWatch</i> a lo largo del test junto a un posible comportamiento por parte de <i>AWS Lambda</i> | 56 |

CAPÍTULO 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 [1]. 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 [2], 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 de 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.

¹Apple, Inc. (2019). *Siri - Apple (ES)* - <https://www.apple.com/es/siri/>

²Apple, Inc. (2019). *Apple (España)* - <https://www.apple.com/es/>

³Google LLC. (2019). *El Asistente de Google* - https://assistant.google.com/intl/es_es/

⁴Microsoft Corporation. (2019). *Cortana* - <https://www.microsoft.com/es-es/windows/cortana>

⁵Amazon.com, Inc. (2019). *Amazon Alexa* - <https://developer.amazon.com/es/alexa>

⁶Amazon.com, Inc. (2019). *AWS* - <https://aws.amazon.com/es/>

⁷Google LLC. (2019). *Cloud Computing Services* - <https://cloud.google.com/>

⁸Microsoft Corporation. (2019). *Microsoft Azure* - <https://azure.microsoft.com/es-es/>

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 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 de vuelta 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 ya sea como elemento de asistencia o 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 vio 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 decisión 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*⁹ como un proyecto de código abierto [3] bajo la licencia *Apache 2.0* [4].

1.2 Objetivos

El principal objetivo del proyecto es el diseño de un bot conversacional centrado en el ámbito de la meteorología y 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 consultas meteorológicas 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.
- 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 pueden 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 en la nube. Además, y como impactos secundarios se espera promover la utilización este tipo de bots en entornos

⁹GitHub, Inc. (2019). *GitHub* - <https://github.com/>

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

La memoria empieza con una introducción, que presenta el proyecto y sus objetivos y sigue con una referencia al 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 qué pruebas, de validación y rendimiento, se han ejecutado sobre éste. 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 8.

CAPÍTULO 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. Como consecuencia, la ejecución no es tan simple 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, 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. De entre estas, se han elegido cinco para su análisis, tanto por importancia como por la cantidad de usuarios que las utilizan: *Amazon Lex*¹, *Google Dialogflow*², *IBM Watson Assistant*³, *Microsoft LUIS*⁴ y *Wit.AI*⁵.

2.1.1. *Amazon Lex*

La alternativa de *Amazon*⁶, presentada a la comunidad de desarrolladores en Abril de 2017 [5], 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 *AWS Lambda*⁷, 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, 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 producir forzosamente una salida emborrona algunos desarrollos a los cuáles no les interesa esta funcionalidad.

¹Amazon.com, Inc. (2019). *Amazon Lex* - <https://aws.amazon.com/lex/>

²Google LLC. (2019). *Dialogflow* - <https://dialogflow.com/>

³International Business Machines Corporation. (2019). *IBM Watson* - <https://www.ibm.com/watson>

⁴Microsoft Corporation. (2019). *Language Understanding (LUIS)* - <https://azure.microsoft.com/es-es/services/cognitive-services/language-understanding-intelligent-service/>

⁵Facebook Inc. (2019). *Wit.ai* - <https://wit.ai/>

⁶Amazon.com, Inc. (2019). *Amazon.es* - <https://www.amazon.es/>

⁷Amazon.com, Inc. (2019). *AWS Lambda* - <https://aws.amazon.com/es/lambda/>

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, el coste de uso de *Amazon Lex* sería de 23.5 \$, un equivalente, al cambio, de 20.68 €⁸, situándose como una alternativa medianamente cara en relación a sus competidores directos⁹.

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 gratuita 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*¹⁰, 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 cómo debe ser procesada la información. Otra ventaja es la de integrar automáticamente los agentes con *Google Home*¹¹, así como con otros servicios de terceros como *Telegram*¹², *Twitter*¹³ o *Skype*¹⁴. Sin embargo, su principal beneficio es *Chatbase*, su programa de análisis, respaldado con la tan larga establecida trayectoria de *Google Analytics*¹⁵.

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* cuenta con 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 capa gratuita, aunque, en el caso de querer utilizar la versión más cara, supondría un precio de 74 \$ mensuales, unos 65,12 €¹⁶.

⁸Se ha utilizado una conversión de 1 \$ a 0.88 € para los ejemplos, correspondiente al precio de cambio regulado del día 17/03/2019.

⁹Los costes han sido extraídos de <https://aws.amazon.com/lex/pricing/>, a día 17/03/2019.

¹⁰Google LLC. (2019). *Google* - <https://www.google.es/>

¹¹Google LLC. (2019). *Google Home* - https://store.google.com/es/product/google_home

¹²Telegram FZ-LLC. (2019). *Telegram Messenger* - <https://telegram.org/>

¹³Twitter, Inc. (2019). *Twitter* - <https://twitter.com/>

¹⁴Microsoft Corporation. (2019). *Skype* - <https://www.skype.com/es/>

¹⁵Google LLC. (2019). *Google Analytics* - <https://analytics.google.com/analytics/>

¹⁶Los costes han sido extraídos de <https://dialogflow.com/pricing>, a día 17/03/2019.

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*¹⁷ 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-premise* y un enfoque en la detección eficiente y distintivo frente a las otras opciones.

Debido a ese enfoque, la principal ventaja de *IBM Watson* es su capacidad de detección y análisis de las peticiones del usuario, siendo capaz de obtener resultados con un rango de precisión mucho mayor 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 a 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 mensajes de audio, por lo que requiere de una herramienta de transformación *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 \$ por 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 necesario pagar un precio de 35 \$ mensuales, unos 30.8 €¹⁸, 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.

¹⁷International Business Machines Corporation. (2019). *IBM* - <https://www.ibm.com/>

¹⁸Los costes han sido extraídos de <https://www.ibm.com/cloud/watson-assistant/pricing/>, a día 26/03/2019.

2.1.4. Microsoft LUIS

El *framework* de la empresa poseedora de *Windows*¹⁹ y *Office*²⁰, *Microsoft*²¹, es conocido por su sigla *LUIS*, correspondiente 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, a su vez, hace que muchos programadores se puedan ver superados por la cantidad de trabajo manual a realizar. Por último, *LUIS* no cuenta con transformación de audio a texto en su capa gratuita, 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 audio²². Asumiendo la capa 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*²³ fue iniciada como producto estrella de una *startup* en 2013 y adquirida por la multinacional en 2015 [6]. 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 de una capa gratuita.

¹⁹Microsoft Corporation. (2019). *Windows* - <https://www.microsoft.com/es-es/windows>

²⁰Microsoft Corporation. (2019). *Office 365* - <https://www.office.com/>

²¹Microsoft Corporation. (2019). *Microsoft* - <https://www.microsoft.com/es-es>

²²Los costes han sido extraídos de <https://azure.microsoft.com/en-us/pricing/details/cognitive-services/language-understanding-intelligent-services/> a día 26/03/2019.

²³Facebook Inc. (2019). *Facebook* - <https://es-es.facebook.com/>

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 su propia plataforma. Debido a ello, es necesario apartar el grano de la paja con frecuencia para tener la capacidad de elegir el mejor proveedor para un proyecto y hacerlo al mejor precio.

Para ello, lo primero es tener muy claras cuáles son las necesidades del proyecto, 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, pueden interesar 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 análisis se profundizará también en las características globales, puesto que un bot puede tener cientos de funcionalidades y resultaría limitante cortar lazos antes de conocer cuáles son sus necesidades reales. Por ejemplo, un bot puede utilizarse para almacenar recordatorios, por lo que 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 correo electrónico, 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*²⁴.

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 embar-

²⁴Flexera Software. (2019). *Flexera* - <https://www.flexera.com/>

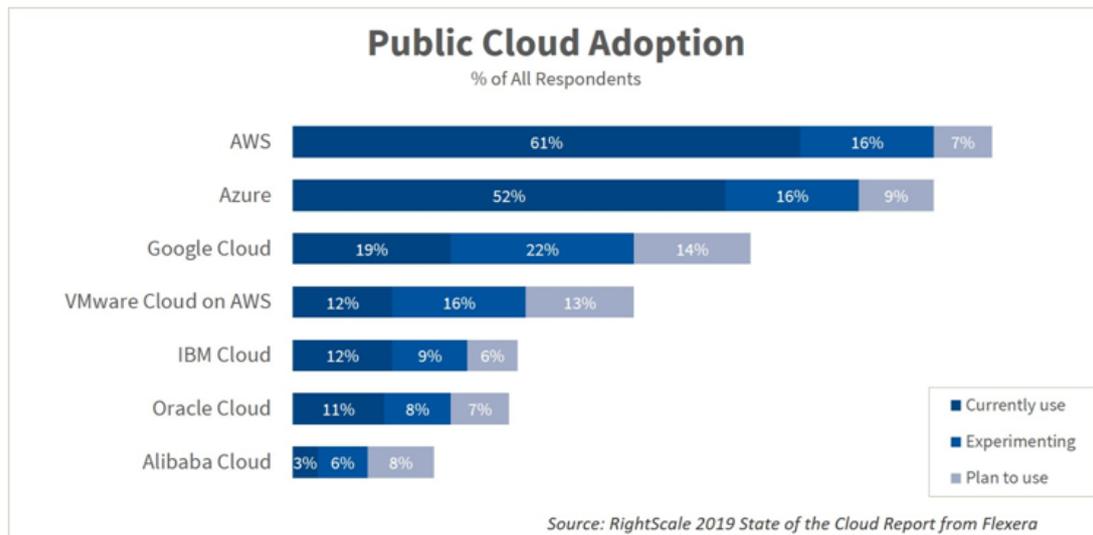


Figura 2.1: Adopción de los distintos proveedores de *cloud* público según Flexera.

go, siguen muy de cerca *VMWare Cloud*²⁵, *IBM Cloud*²⁶, *Oracle Cloud*²⁷ y *Alibaba Cloud*²⁸, este último con 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 este análisis se tendrán 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 [7]. 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* [8].

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* [9]; 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.

²⁵ VMware, Inc. (2019). *VMWare Cloud* - <https://cloud.vmware.com/>

²⁶ International Business Machines Corporation. (2019). *IBM Cloud* - <https://www.ibm.com/es-es>

²⁷ Oracle Corporation. (2019). *Oracle Cloud* - <https://cloud.oracle.com/home>

²⁸ Alibaba Group Holding Limited. (2019). *Alibaba Cloud* - <https://www.alibabacloud.com/>

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 gratuita.

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 [10]. 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 las empresas pueden seguir dependiendo de *data centers* locales.

Entre sus principales puntos diferenciadores está la integración con otro software de *Microsoft*, como *Windows*, *Office* o *SQL Server*²⁹, 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*. Pese a que en general funciona ligeramente peor que *AWS Lambda* [11], 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*³⁰ o *Dropbox*³¹ 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*, 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.

²⁹Microsoft Corporation. (2019). *SQL Server 2019* - <https://www.microsoft.com/es-es/sql-server/sql-server-2019>

³⁰Microsoft Corporation. (2019). *Microsoft OneDrive* - <https://onedrive.live.com/about/es-es/>

³¹Dropbox, Inc. (2019). *Dropbox* - <https://www.dropbox.com/>

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*³² [12], donde sí ha sido pionera, ha permitido que sea elegida por muchos arquitectos *cloud* para la realización de sus tareas [13]. 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* [14].

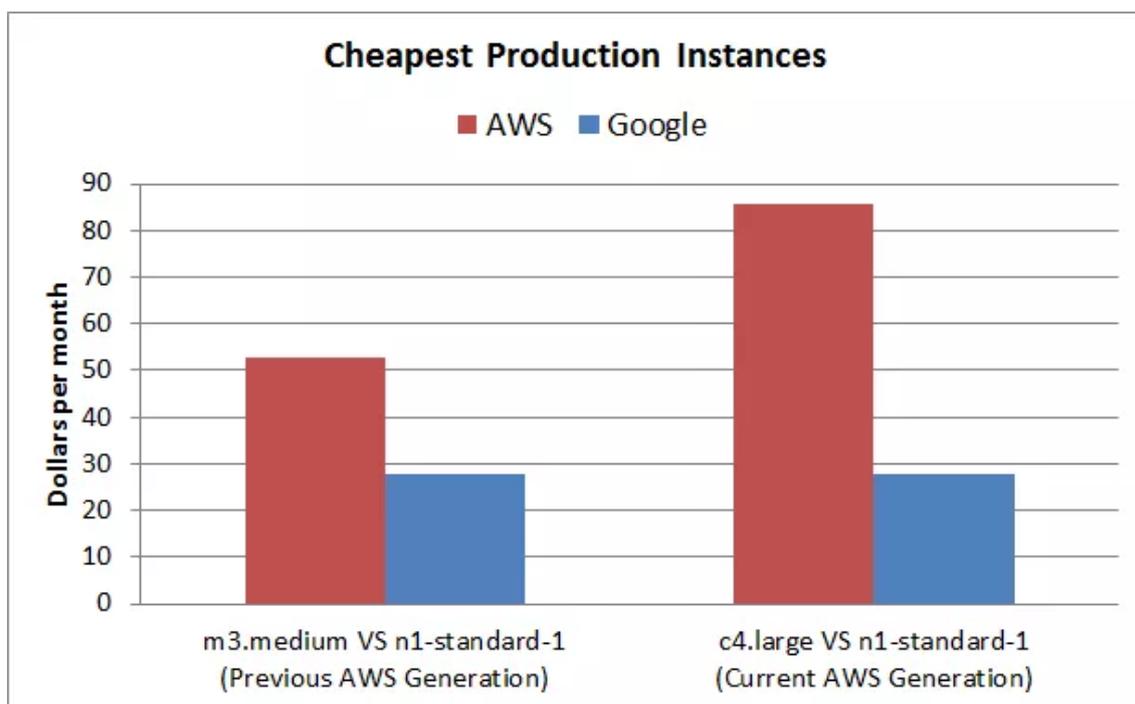


Figura 2.2: Comparación de precios mínimos entre *Amazon Web Services EC2* y *Google Cloud Engine*.

En lo que respecta a *Google Cloud Functions*, la opción *serverless* de *Google Cloud Engine*, ésta se queda 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 *Google Cloud Platform* 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* [15] y entendiendo su misión como plataforma de refuerzo en un entorno *multi-cloud*.

³²Google LLC. (2019). *Kubernetes* - <https://kubernetes.io/>

2.3 Crítica al estado del arte

Tal y como se ha visto en los apartados anteriores, existen plataformas que permiten la creación de bots conversacionales 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 un bot 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 2.2.1, será el proveedor de *cloud* elegido, debido a la facilidad a la hora de desplegar servicios *serverless* con *AWS Lambda*, a la alta adopción por parte de 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.

CAPÍTULO 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 permissionados, 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 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 parte del *Back-end 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 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 de-

berán 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 ha definido 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 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 estar desplegada de tal forma que puede ser devuelta al usuario a través de la invocación de una función *serverless*. El usuario debe 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 ser posible el registro y el inicio de sesión, restringiendo el acceso al bot a aquellos usuarios dados de alta. Además, debe 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 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

La figura 3.1 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 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 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.

Analizando más profundamente la interacción en la figura 3.2 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

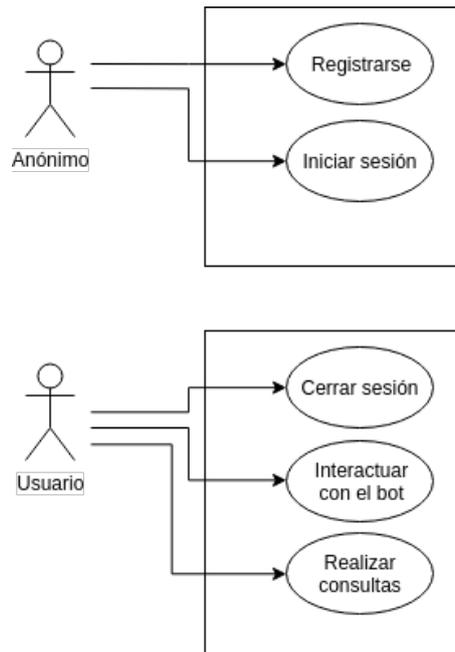


Figura 3.1: Diagrama *UML* correspondiente a las acciones que se deberían de poder realizar en el sistema.

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?”.

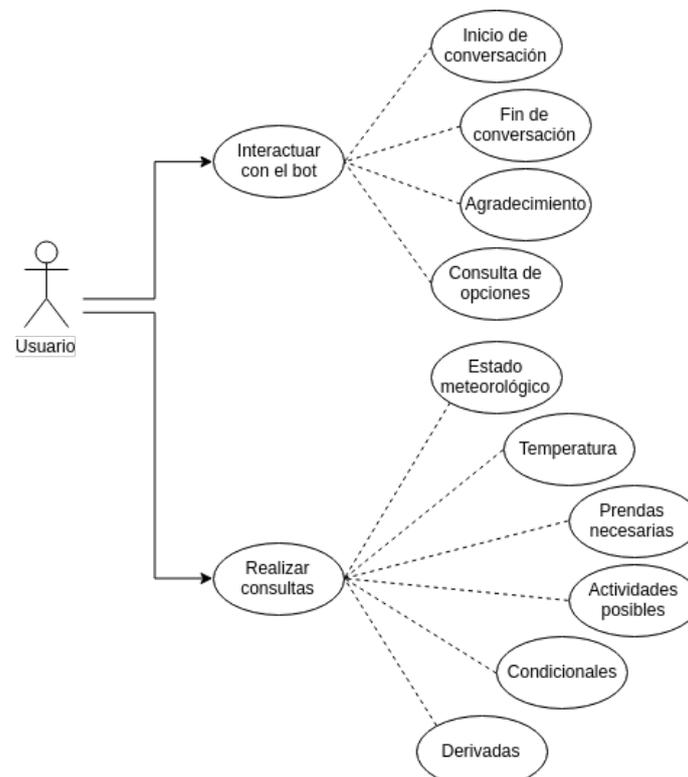


Figura 3.2: Diagrama *UML* con profundización en las actividades que puede realizar el usuario.

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 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* 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 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 autenticación y el registro de usuarios mediante la tecnología *OAuth 2.0*¹. 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 autenticación 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 producir qué tipo de respuesta. Esa información debe ser usada para reentrenar la herramienta hasta alcanzar un porcentaje de predicciones correctas lo suficientemente elevado.

¹OAuth. (2019). *OAuth 2.0* - <https://oauth.net/2/>

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*², 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* 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.

²Serverless, 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 *Node.JS*, indistintamente. Para el despliegue se hará uso del *framework serverless*.
- **Exposición de las funciones de forma pública:** Sólo se expondrá una función de *AWS Lambda* con *API Gateway*. Además, para ello se hará uso también del *framework serverless*.

CAPÍTULO 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 **conurrencia** 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.

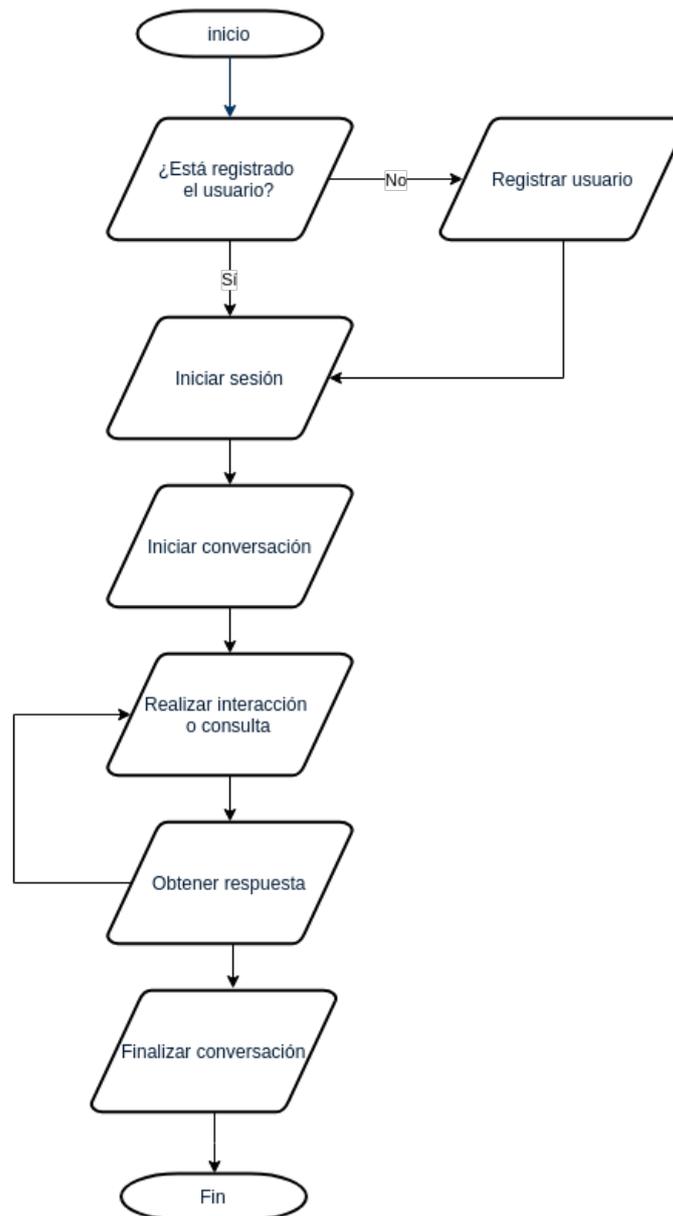


Figura 4.1: Diagrama de flujo de la aplicación.

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 *fulfillment*. 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*.

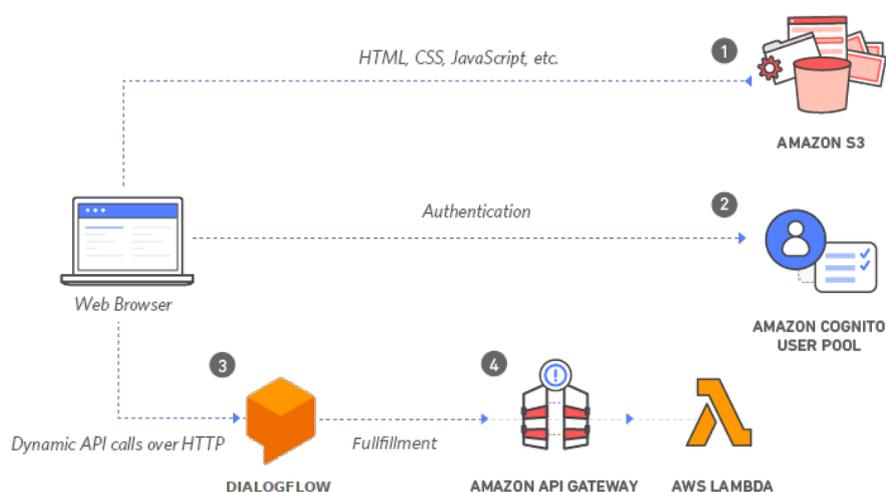


Figura 4.2: Esquema general de la arquitectura del sistema.

4.2 Diseño detallado

Una vez presentada 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 *Desarrollo de la solución propuesta*.

4.2.1. Interfaz

Empezando por el *front-end*, es necesario definir que éste se entienda como un módulo específico y no como una web completa, de modo que debe 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.

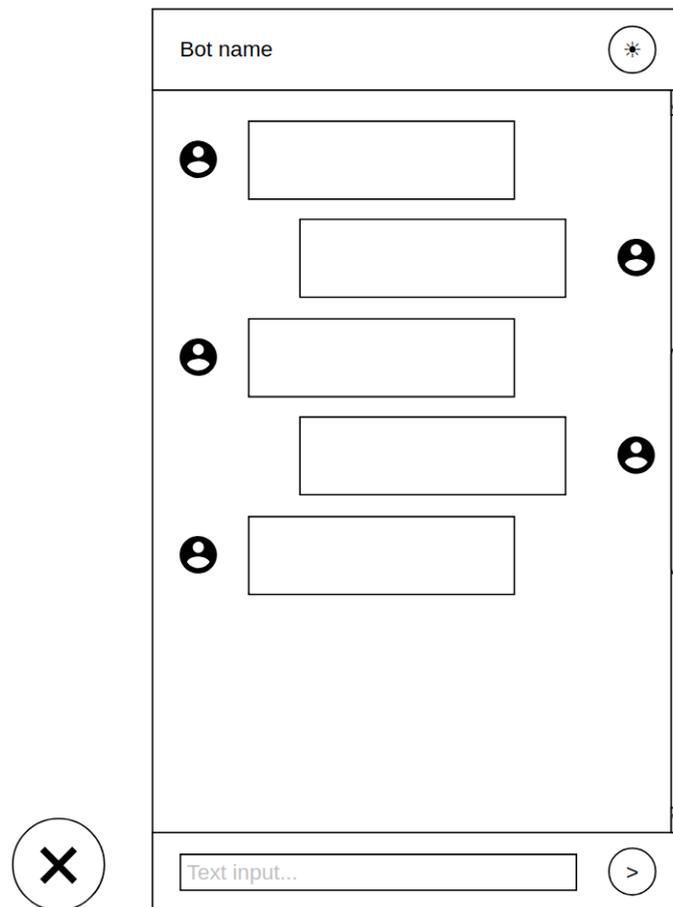


Figura 4.3: Mockup del módulo correspondiente al bot en la interfaz.

Para garantizar éste diseño como módulo, se hará uso del botón flotante característico de *Material Design*, que 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*¹, 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

Existen tres puntos a lo largo de la aplicación en los que es necesario establecer un sistema de autenticación: el inicio de sesión en la interfaz gráfica del bot, para permitir su uso; la conexión entre el *front-end* y el intérprete de lenguaje natural y la integración entre este último y el *back-end*.

Empezando por el primero, para permitir el inicio de sesión en la interfaz gráfica del bot se plantea el uso del servicio *Amazon Cognito*. Ésta solución permitiría no solo hacer uso de usuario y contraseña, sino que también sería válido explotar *OAuth 2.0* para facilitar la autenticación con otras cuentas como la de *Google*, *Facebook* o *GitHub*. Para ello, habría que crear una *pool* de usuarios y exigirles, como mínimo, tres datos: correo electrónico, que serviría como identificador único y como objetivo de mensajes del servicio; contraseña, para la autorización, y nombre de usuario, para permitir que el bot pueda utilizar tal campo en sus mensajes.

A raíz de tal sistema, y como es lógico, habría que modificar el *front-end* presentado anteriormente para incluir en él nuevos elementos. El primero de estos sería un sistema de detección que comprobase si el usuario está autenticado o no, y, como complemento, pestañas correspondientes al inicio de sesión y al registro por parte de nuevos usuarios. Para simplificar la lógica, esta información sería mantenida por el mismo *JavaScript*, en vez de utilizar elementos como *cookies* o variables de sesión.

El siguiente de los puntos, correspondiente a la conexión entre el *front-end* y el intérprete de lenguaje natural requeriría que, en la conexión entre estos, el primero hiciese uso de la *key* proporcionada por *Google Cloud* para el acceso al *gateway* que conecta con *Dialogflow*. Estos campos son proporcionados directamente por la herramienta de *Google* y forman parte del *JSON* de conexión estándar, por lo que no habría que profundizar en ellos más allá de hacer correcto uso de los mismos.

Finalmente, para la conexión entre *Dialogflow* y *API Gateway* sería necesario crear una función de autenticación. Ésta, entendida como una función más de *AWS Lambda* debería ejecutarse con cada llamada a *API Gateway* y comprobar que la cabecera de la petición incluye un token de credenciales válido, tal y como haya sido definido en su configuración. Se entiende, a su vez, que *Dialogflow* debería persistir el usuario y la contraseña con el que invocar al *fulfillment* y, por tanto, al *back-end*. Tal y como se aprecia en la figura 8, la herramienta de *Google* ya cuenta con campos preparados para tal uso en su propia configuración.

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 interacciones: aquellas sociales, aquellas simples relacionadas con la meteorología y aquellas que, siendo también relacionadas con la meteorología, son derivadas y, por tanto, más complejas.

¹World Weather Online. (2019). *World Weather Online* - <https://www.worldweatheronline.com>

⚡ Fulfillment

Webhook ENABLED

Your web service will receive a POST request from Dialogflow in the form of the response to a user query matched by intents with webhook enabled. Be sure that your web service meets all the [webhook requirements](#) specific to the API version enabled in this agent.

| | | |
|------------|----------------|----------------|
| URL* | Enter URL | |
| BASIC AUTH | Enter username | Enter password |
| HEADERS | Enter key | Enter value |
| | Enter key | Enter value |

[+ Add header](#)

Figura 4.4: Detalle del *fulfillment* en *Dialogflow* donde se aprecian los campos *username* y *password*.

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 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 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 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 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, sobrescribiendo 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 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 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, llamada *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 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 haber un *path* único y procesar todas las peticiones a través de él.

Esta característica obliga a que exista una función de *AWS 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. Pese a que ambas opciones son lógicamente válidas, sólo la segunda tiene sentido, debido a que la invocación de una función Lambda a través de otra supondría pagar por el tiempo consumido por ambas, dando lugar al problema de *double billing* [16]. Esto es así ya que, debido a que la primera debería esperar a que la segunda terminase para obtener su resultado, se generaría un espacio de tiempo igual al tiempo de ejecución de la segunda función en el que se estaría pagando el doble.

Por los motivos presentados, 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 cual 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

A continuación se detallan los servicios de *Amazon Web Services* planteados para el desarrollo.

Amazon IAM (Identity and Access Management)

Servicio de administración de identidades de *AWS* que permite administrar el acceso a los servicios y recursos de la plataforma de manera segura. Además, puede crear y administrar usuarios y grupos, así como utilizar permisos para conceder o negar el acceso de estos a los recursos.

Se trabajará con este servicio, sin profundizar en él, para darle los permisos necesarios al usuario del *framework serverless* y a las funciones de *AWS Lambda*.

Amazon S3 (Simple Storage Service)

Servicio de almacenamiento de objetos que ofrece escalabilidad, disponibilidad de datos, seguridad y alto rendimiento y que se paga en función del tráfico y del almacenamiento total almacenado. Esto significa que clientes de todos los tamaños y sectores pueden utilizarlo para almacenar y proteger cualquier cantidad de datos para diversos casos de uso, como sitios web, aplicaciones móviles, procesos de copia de seguridad y restauración, operaciones de archivado, aplicaciones empresariales, dispositivos *IoT* y análisis de *Big Data*.

Se utilizará para almacenar la interfaz web de forma pública así como el código de las funciones de *AWS Lambda* de forma privada.

Amazon Cognito

Servicio de identificación y sincronización de usuarios que garantiza simplicidad y seguridad para el control de acceso, la inscripción y el inicio de sesión. Permite trabajar con *OAuth 2.0* con cuentas de servicios como *Google*, *Amazon* y *Facebook* y tiene una capa gratuita que permite trabajar con hasta cincuenta mil cuentas de usuario.

Se utilizará para la autenticación del usuario frente al bot en el *front-end*.

AWS Lambda

Servicio de informática sin servidor que ejecuta código en respuesta a eventos y administra automáticamente los recursos informáticos subyacentes. *Lambda* ejecuta el código en una infraestructura informática de alta disponibilidad y se encarga de la administración integral de los recursos informáticos, incluido el mantenimiento del servidor y del sistema operativo, el aprovisionamiento de capacidad y el escalado automático, la implementación de parches de seguridad y código, así como la monitorización de código y los registros.

Se utilizará para el despliegue de las funciones como servicio.

Amazon API Gateway

Servicio completamente administrado que facilita a los desarrolladores la creación, la publicación, el mantenimiento, la monitorización y la protección de *APIs* a cualquier escala. Permite exponer de forma pública a través de *RESTful APIs* internas desplegadas en instancias *EC2*, balanceadores de carga de *EKS* o funciones de *AWS Lambda*.

Se utilizará para exponer las funciones como servicio de modo que sean accesibles desde el intérprete de lenguaje natural.

4.3.2. Herramientas y lenguajes utilizados

A continuación se detallan las herramientas, los *frameworks* los lenguajes de programación planteados para el desarrollo.

Dialogflow

Dialogflow es una herramienta propiedad de *Google* para el desarrollo de tecnologías de interacción entre humanos y ordenadores basada en la interpretación de conversaciones en lenguaje natural.

Se utilizará para para desarrollar la lógica del bot conversacional.

Vue.js

Framework Open Source escrito en *JavaScript* que permite desarrollar interfaces de usuario y aplicaciones de página única o *single-page applications (SPA)*. Permite hacer uso de una arquitectura adaptable que permite funcionalidad avanzada como mantenimiento de estado, *routing* y reactividad inmediata.

Se utilizará para el desarrollo de la interfaz del bot conversacional.

Node.js

Entorno en tiempo de ejecución multiplataforma, de código abierto, para la capa del servidor (pero no limitándose a ello) basado en el lenguaje de programación *ECMAScript*, asíncrono, con *I/O* de datos en una arquitectura orientada a eventos y basado en el motor *V8* de *Google*. Su enfoque es el de ser útil en la creación de programas altamente escalables.

Se utilizará para el desarrollo de las funciones *serverless* de autenticación.

Python

Lenguaje de programación de propósito general interpretado, de alto nivel. La filosofía de diseño de este lenguaje enfatiza la legibilidad del código con su notable uso de espacios en blanco significativos. Sus construcciones y su enfoque orientado a objetos apunta a ayudar a los programadores a escribir código claro y lógico para proyectos de pequeña y gran escala.

Se utilizará para el desarrollo de las funciones *serverless* correspondientes a la lógica del bot.

Serverless

Framework Open Source escrito en *Node.js* que permite desplegar fácilmente aplicaciones sobre *AWS Lambda*. Actualmente permite trabajar con otras plataformas como *Microsoft Azure*, *Google Cloud* o incluso *Kubernetes*.

Se utilizará para el despliegue automatizado del *back-end*.

Artillery.io

Kit de herramientas para pruebas de carga y pruebas funcionales modernas, potentes y fáciles de ejecutar. Su utilidad es la de realizar test sobre *back-ends* escalables, *APIs* y servicios que se mantengan eficientes y resistentes bajo carga alta.

Se utilizará para realizar las pruebas de carga y rendimiento del *back-end*.

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. Este proceso se ha planteado en un orden inverso al del diseño, debido a que de forma inversa se trabajaría con ciertas dependencias antes de tener preparada su implementación, como en el caso de llamar a un *back-end* que todavía no existe. El objetivo es profundizar 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.

A continuación se expondrá, por orden de realización, cuál fué el proceso de desarrollo de cada uno de los objetivos.

5.1 Funciones *serverless*

El primer aspecto en el que se ha profundizado es el desarrollo y el despliegue de las funciones como servicio haciendo uso de la herramienta *AWS Lambda*. Éstas funciones, tal y como se han definido en la fase de diseño, son dos: una que permite verificar la autenticación del usuario y otra que se encarga de responder a las consultas relacionadas con la meteorología.

Empezando por la primera, ésta es una función escrita en *JavaScript* que se ejecuta sobre *Node.js* en su versión *8.10* y se encarga de comprobar que el usuario y la contraseña que se incluyen cifrados en la cabecera de las peticiones a *API Gateway* son válidos. Este método de validación es una opción alternativa al uso de *tokens* o *Amazon Cognito* cuando se controla el acceso a la *API*.

Para implementar esta función, llamada *ptolomeo-authorizer*, se ha diseñado un objeto llamado *AuthPolicy* que se encarga de autorizar o denegar las peticiones, así como de mostrar el motivo en el segundo caso. Este proceso, que verifica la igualdad entre los parámetros extraídos del token y los almacenados, permite enviar distintos tipos de respuesta. De este modo, si el usuario o la contraseña no son válidos, la respuesta será *401 Unauthorized*, mientras que si las credenciales sí son válidas pero no se puede acceder a la función, la respuesta sería *403 Access Denied*. Es interesante recordar que, en caso de que no hubiese ningún error, la respuesta no sería *200 OK*, sino la obtenida de la ejecución de la función lambda a la que apunta tal recurso de la *API*.

La otra función desarrollada, mucho más extensa, es la que se encarga de responder a las consultas relacionadas con la meteorología que llegan a través de *Dialogflow*. Su nombre es *ptolomeo-weather* y está compuesta por distintos módulos escritos en *Python 3.6*. El motivo del uso de dos lenguajes de programación para las funciones es que ésta

se basa ligeramente en un un proyecto de consulta a la API de *World Weather Online* escrito en *Python* y expuesto como otra API gracias al *framework Flask*¹. Por lo tanto, con la intención de no diseñar otro cliente para el mismo sitio, el grueso del desarrollo ha estado en la adaptación del código a la ejecución como función y no como servicio; sin embargo, también se han realizado cambios sustanciales y mejoras al código ya existente.

Para garantizar el uso como función de *AWS Lambda*, el primer paso es establecer cuál será el punto de entrada. En este caso, se hace uso del archivo *lmbda/weather/app.py*, que cuenta con el método *handler(event, context)*, al que se llama por defecto en la ejecución del *FaaS*. Este método se encarga de llamar a la función *make_response*, un balanceador a los distintos métodos específicos, para obtener una respuesta a la petición y, posteriormente, construye un *JSON* con la misma, que se devuelve al cliente junto a una cabecera *200 OK*.

Los métodos a los que puede llegar la petición son cinco: *weather*, *weather_activity*, *weather_condition*, *weather_outfit* y *weather_temperature*. Todos estos, exceptuando ligeras diferencias relacionados con cómo se construye la respuesta, tienen un comportamiento similar. Éste se basa en validar los parámetros de entrada, obtener la predicción del tiempo de la API de *WWO* en el lugar, y entre la fecha de inicio y la fecha de fin seleccionados; y generar la respuesta a través de la información recibida. Por ejemplo, “¿Qué tiempo hace hoy?”, “¿Va a llover hoy?” y “¿Cojo paraguas?”, suponen obtener la misma predicción, aunque cada respuesta se genera de forma personalizada según el tipo de pregunta.

Los caracteres especiales y sus asociaciones, como que “*chaqueta*” y “*gorro*” se encuentran dentro de la categoría “*frío*”, requeridos para el entendimiento de las cuestiones, se encuentran dentro del archivo *.weather_entities.py*. A su vez, la traducción inversa, un *pool* de respuestas válidas, se encuentra en *.weather_responses.py*. Un ejemplo de las mismas sería *LIST_COLD* = [*'Hace bastante frío allí.'*, *'Hace bastante frío, diría.'*, *'Yo no me olvidaría de mis guantes.'*].

Finalmente, para el despliegue de estas funciones se hace uso del *framework serverless*, que, a través de un *YAML* [Figura 5.1], se encarga de crear un *bucket* en *Amazon S3*, de poblarlo con el código de las funciones y de crearlas, así como su entrada en la API en uno de los casos. Este detalle se explicará con más profundidad en el siguiente apartado.

5.2 Integración y APIs

Una vez creadas las funciones, estas deben ser expuestas para poder ser accedidas desde fuera de la red interna de *Amazon Web Services*. Para ello, se hace uso de la herramienta *API Gateway*, que expone una dirección *IP* con una *URI* asociada a la cuál se puede acceder desde toda la red, siempre que se tenga acceso a *Amazon.com*.

Haciendo uso de tal herramienta se debería crear el árbol de métodos y recursos típico de los servicios *RESTful*. Sin embargo, tal y como se ha definido el diseño, la función debe ser invocada por *Dialogflow*, que solo permite atacar a un mismo *path*. Debido a esto sólo se ha creado un recurso, */weather*, asociado al método *POST*, que redirige la llamada a la función lambda *ptolomeo-weather*, presentada en el apartado anterior.

Además, para prevenir que haya otros usuarios haciendo uso de la API privada, se han definido un usuario y una contraseña. Estos credenciales se comprueban haciendo uso del método *ptolomeo-authorizer*, definido también en el apartado previo.

Para crear la *RESTful API*, preparada para producción, se ha hecho uso del *framework serverless* de nuevo. En este caso, ampliando la figura 5.2, se pretende incidir en el campo

¹Google LLC. (2018). *Dialogflow Fulfillment Weather Sample Python (Flask)* - <https://github.com/dialogflow/fulfillment-weather-python>

```
1 service: ptolomeo
2
3 provider:
4   name: aws
5   stage: dev
6   region: eu-west-1
7   memorySize: 256
8   deploymentBucket:
9     name: ptolomeo.tfg.marsofon
10  plugins:
11    - serverless-python-requirements
12
13 functions:
14   weather:
15     handler: lambda/weather/app.handler
16     name: ptolomeo-weather
17     description: Ptolomeo Weather Logic
18     runtime: python3.6
19     memorySize: 128
20     timeout: 30
21     events:
22       - http:
23         path: weather/
24         method: post
25         integration: lambda
26         authorizer: authorization
27
28   authorization:
29     handler: lambda/authorizer/index.handler
30     name: ptolomeo-authorizer
31     description: Ptolomeo Authorization Logic
32     runtime: nodejs8.10
33     memorySize: 128
34     timeout: 60
```

Figura 5.1: Detalle del *YAML* utilizado para el despliegue de las funciones como servicio.

events de la función *ptolomeo-weather*, donde se ve que se declara una *API HTTP*, que ejecuta la función *lambda* ante una llamada *POST* sobre *weather/* y la autoriza a través de la ejecución de *authorization*, que, como se define más arriba, corresponde a *ptolomeo-authorizer*.

```
1 functions:
2   weather:
3     handler: lambda/weather/app.handler
4     name: ptolomeo-weather
5     description: Ptolomeo Weather Logic
6     runtime: python3.6
7     memorySize: 128
8     timeout: 30
9     events:
10      - http:
11        path: weather/
12        method: post
13        integration: lambda
14        authorizer: authorization
```

Figura 5.2: Detalle del campo *events* en *YAML* utilizado para el despliegue de *ptolomeo-weather*.

Como nota, en este subapartado sólo se ha explicado la *RESTful API* correspondiente a las funciones como servicio y no la de *Dialogflow*, que si estaba presente en el diseño. Esto es debido a que la interfaz de la segunda ya está definida y, por lo tanto, no había que implementarla.

5.3 *Dialogflow*

El siguiente elemento a desarrollar e implementar es el intérprete de lenguaje natural, *Dialogflow*. Para ello, hay que cumplir ciertos objetivos que no sólo están ligados a la misma interpretación, sino que también es necesario configurar el *fulfillment* y exponer el bot gracias a la *API V2* y a *Google Cloud*.

Empezando por lo primero, la interpretación, es necesario recordar que tal y como se ha definido en el diseño existen tres tipos de interacciones: aquellas básicas, aquellas simples que requieren de *fulfillment* y aquellas que, requiriendo el mismo *fulfillment*, son derivadas y, por tanto, más complejas. Sin embargo, antes de profundizar en ese aspecto es interesante ahondar en cómo se crea un bot en la plataforma de *Google*.

Para crear un bot es necesario ir a la página web de *Dialogflow* y registrarse en ella haciendo uso de una dirección de correo *gmail.com*, sobre la que se inicia también una cuenta de *Google Cloud*. Una vez dentro, el botón de *CREATE AGENT* permite definir el primer agente, sobre el que se especifica un nombre, un idioma y una zona horaria. Otros aspectos, más específicos, como la descripción, el identificador del proyecto o su avatar se pueden establecer, si se requiere, posteriormente en el panel de control del mismo.

Una vez creado, ya se puede volver a tratar el problema de los tipos de interacciones. Empezando por el primero de estos, las interacciones básicas, que corresponden a las sociales, pueden ser creadas accediendo a la pestaña de *Intents* del agente y pulsando sobre *CREATE INTENT*. Este tipo de interacciones no requieren de aspectos avanzados como contextos o parámetros, por lo que es suficiente con establecer un nombre, establecer un número suficiente de *training phrases* y crear un *pool* de respuestas válidas.

Por ejemplo, para el saludo, ejemplos de frases de entrenamiento para el saludo o *Default Welcome Intent* serían “hola”, “hey”, “saludos”, “chao”, “buenas”, “buenos días”, mientras que un *pool* válido de respuestas sería “¡Hola!”, “¡Hey!”, “¡Buenos días!”. Es de interés destacar que mientras que las frases usadas para el entrenamiento pueden ser mayúsculas, minúsculas o sin signos de puntuación, por ejemplo, ya que se procesan de la misma forma, la respuesta sí que debe contar ya con el formato que recibirá el usuario.

La siguiente de las categorías corresponde a la de las oraciones simples que requieren *fulfillment* y recogida de datos, como “¿Qué tiempo hará mañana en Madrid?”. Estas oraciones requieren de una lista de elementos conocidos como *Entities* que establecen tipos de valores dentro de una misma categoría y relaciones entre ellos.

Un ejemplo de esto sería el siguiente, correspondiente a un fragmento de la *entity @outfit*: [“value”: “umbrella”, “synonyms”: [“paraguas”], “value”: “jacket”, “synonyms”: [“chaqueta”, “abrigo”]]. Esta lista establece un conjunto de relaciones que interviene en el análisis de una oración, estableciendo, por ejemplo, “abrigo”, en la oración “¿Voy a necesitar abrigo esta noche?” como un tipo de *@outfit* válido y con el valor “jacket”, que permite simplificar el *back-end* al existir una lista de variables sencilla en la resolución pero completa en la interpretación.

Siguiendo esta política, se han definido cinco *Entities* para el proyecto: *@activity*, para actividades al aire libre; *@outfit*, para prendas de ropa; *@temperature*, para condiciones como frío o calor; *@unit-temperature*, para las unidades de retorno como *Celsius*, *Kelvin* o *Fahrenheit*, y *@weather-condition*, para fenómenos climatológicos como lluvia, nieve o

tormenta. Otras entidades de uso común, como fechas, tiempos, localizaciones y nombres propios, ya están definidas por defecto en *Dialogflow*, por lo que no se requiere ahondar en ellas.

Una vez definidas todas las entradas de recolección de datos, ya se ha procedido a la creación de las intenciones simples con *fulfillment*. Éstas, que son cinco, *weather*, *weather.activity*, *weather.condition*, *weather.outfit* y *weather.temperature*, tienen que contener ciertas entidades para ser válidas. Por ejemplo, toda interacción de *weather.outfit* debe contener un *@outfit* que defina cuál es la prenda de ropa a analizar y un patrón similar al de las frases utilizadas en el entrenamiento, como “¿Me hará falta abrigo esta noche?” o “¿Necesitaré un paraguas el próximo viernes en París?”. Nótese que, como se aprecia en las oraciones de ejemplo, no todos los datos son requeridos, ya que la fecha, *@sys.date-time*, y la dirección *@sys.location* ya cuentan con valores por defecto en el *back-end*, correspondientes a ahora y a Valencia.

Para obtener la respuesta de este tipo de preguntas no existe un *pool* de contestaciones válidas, sino que se ha generado una acción con los parámetros y se ha habilitado el *webhook*, que se definirá más adelante. La acción, en este caso, es *weather.outfit* y los parámetros son *outfit*, *address* y *date-time*, sobre los que se vinculan *@outfit*, *@sys.location* y *@sys.date-time*, respectivamente [Figura 5.3].

Action and parameters ^

weather.outfit

| REQUIRED ? | PARAMETER NAME ? | ENTITY ? | VALUE | IS LIST ? |
|--------------------------|------------------|----------------|-------------|--------------------------|
| <input type="checkbox"/> | outfit | @outfit | \$outfit | <input type="checkbox"/> |
| <input type="checkbox"/> | address | @sys.location | \$address | <input type="checkbox"/> |
| <input type="checkbox"/> | date-time | @sys.date-time | \$date-time | <input type="checkbox"/> |
| <input type="checkbox"/> | Enter name | Enter entity | Enter value | <input type="checkbox"/> |

[+ New parameter](#)

Figura 5.3: Detalle de la configuración de *Action and parameters* en el *intent weather.outfit*.

Finalmente, para dar por terminados este tipo de *intents* relacionados por la meteorología, se ha añadido un contexto de salida, *output context*, llamado *weather* y con una duración de dos entradas. Este detalle se permite hacer uso de las intenciones derivadas.

La última de las categorías, correspondiente a este tipo de entradas, abarca aquellas que son válidas si previamente a las mismas se ha realizado una entrada de un carácter similar. Por ejemplo, la consulta “¿Y chaqueta?” tras obtener la respuesta a “¿Voy a necesitar paraguas?”. Para esto, se hace uso de los contextos, presentados en el párrafo anterior. Estos, que se especifican en el *input context* del nuevo *intent* y existen sólo si han sido declarados como *output context* por alguna función anterior.

Pese a que quitando este aspecto, el desarrollo de las funciones derivadas con *fulfillment* es completamente igual al de las funciones simples que también requieren llamar a una *API*, existen diferencias relacionadas con el contexto. Esto es debido a que esta variable no es solo un nombre, sino que contiene la información que ha sido extraída de las entradas anteriores, como los campos que se han completado o qué tipo de acción se ha

resuelto. De esta forma “¿Y mañana?” reutiliza las variables de *address* y *outfit* y el tipo de acción de “¿Voy a necesitar paraguas esta tarde?”. Para hacer uso de estas variables se utiliza el carácter # seguido del nombre del contexto, tal y como se muestra en este ejemplo del *intent weather - context:weather - comment:date-time*, que permite añadir una fecha nueva a un evento ya existente [Figura 5.4]. Nótese que solo se actualizan los valores existentes en el contexto, dejando vacíos el resto.

Action and parameters ^

#weather.intent_action

| REQUIRED | PARAMETER NAME | ENTITY | VALUE | IS LIST |
|--------------------------|----------------|--------------------|----------------------|--------------------------|
| <input type="checkbox"/> | date-time | @sys.date-time | \$date-time | <input type="checkbox"/> |
| <input type="checkbox"/> | address | @sys.location | #weather.address | <input type="checkbox"/> |
| <input type="checkbox"/> | activity | @activity | #weather.activity | <input type="checkbox"/> |
| <input type="checkbox"/> | condition | @weather-condition | #weather.condition | <input type="checkbox"/> |
| <input type="checkbox"/> | outfit | @outfit | #weather.outfit | <input type="checkbox"/> |
| <input type="checkbox"/> | temperature | @temperature | #weather.temperature | <input type="checkbox"/> |
| <input type="checkbox"/> | unit | @unit-temperature | #weather.unit | <input type="checkbox"/> |
| <input type="checkbox"/> | Enter name | Enter entity | Enter value | <input type="checkbox"/> |

[+ New parameter](#)

Figura 5.4: Detalle de la configuración de *Action and parameters* en el *intent* derivado.

Una vez diseñados todos los *intents*, se ha creado el *fulfillment* al que apuntan todas las funciones del agente que lo requieren. Para esto, se ha ido a la pestaña *fulfillment* y, además de habilitarlo, se han establecido los tres parámetros necesarios: la *URI* correspondiente al recurso de la *RESTful API* al que se quiere apuntar, */dev/weather*; el nombre de usuario y la contraseña utilizados para la autenticación, tal y como se han definido en el código de la función lambda *ptolomeo-authorizer*.

Finalmente, para permitir el acceso al bot a través de herramientas externas se ha ido a la pestaña de control del agente, desde la cuál se puede acceder a *Google Cloud*, haciendo *click* sobre *Service Account*. Una vez en la plataforma de *Google*, y dentro de la pestaña *IAM & admin* se ha creado un nuevo *Service account* en formato *JSON*. Este archivo define un *Gateway* con contiene los datos que son utilizados por el *front-end* para realizar la conexión, como el nombre del proyecto, su identificador y la clave privada que permite la interacción con el mismo.

5.4 Sistema de autenticación

Tal y como se ha definido en el diseño, existen tres puntos en los que la autenticación es necesaria: el acceso a al bot por parte del usuario, la conexión a *Dialogflow* desde el *front-end* y la conexión entre el intérprete y el *back-end*. Pese a que en cada uno de los

apartados correspondientes se ha explicado o se explicará cómo se ha realizado la conexión, en este apartado se profundiza en cada uno de estos puntos.

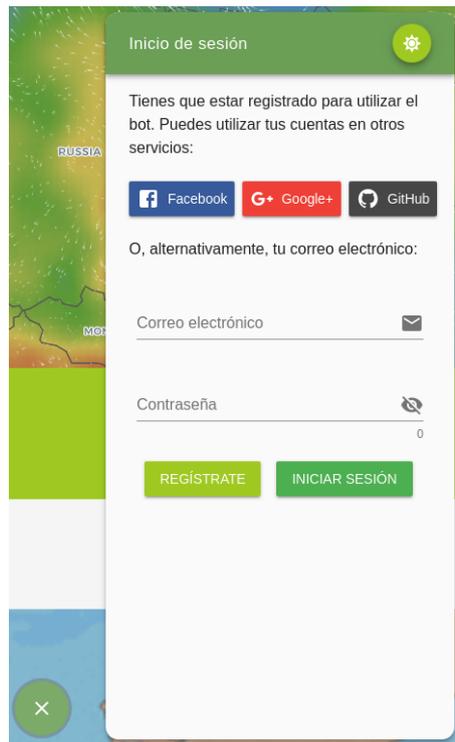


Figura 5.5: Detalle de la vista de Inicio de Sesión en el *front-end*

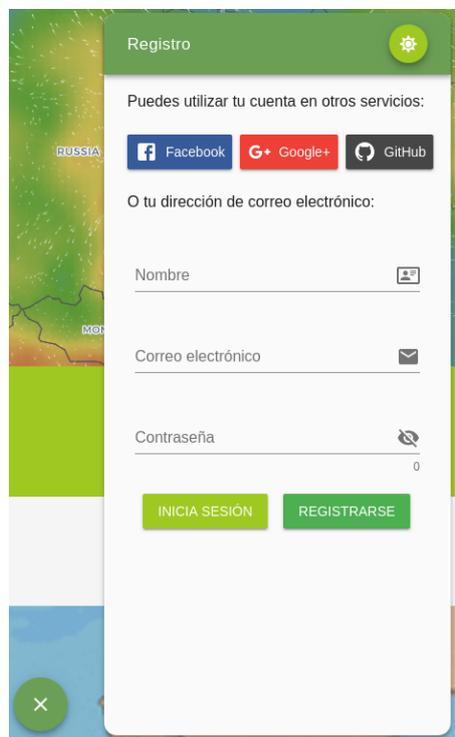


Figura 5.6: Detalle de la vista de Registro en el *front-end*.

El primero de los puntos, el acceso al bot por parte del usuario, se ha resuelto haciendo uso de *Amazon Cognito*. Para ello, se ha añadido a la interfaz un campo correspondiente al usuario actual que, mientras no sea válido, requiere al mismo de autenticarse o registrarse

y le impide hacer uso de la lógica. Con tal finalidad, se han añadido dos pestañas más al bot, una correspondiente al inicio de sesión [Figura 5.5] y otra correspondiente al registro [Figura 5.6].

Tal y como se aprecia en las figuras, estas pestañas deberían de permitir no solo el acceso a través de dirección de correo electrónico y contraseña, sino también a través de *OAuth 2.0*. Sin embargo, el desactualizado *SDK* de *Amazon Cognito* para *JavaScript* ha hecho que, pese a que la interfaz se hubiese preparado para ello, la autenticación utilizando cuentas externas como la de *Google* o *Facebook* no funcione.

Por lo que respecta al servicio de *Amazon*, sobre él se ha creado un *pool* de usuarios de los cuales se requiere nombre, para la interacción; correo electrónico, para la identificación, y contraseña, para la autenticación. Además, para asegurar la autenticidad de los usuarios registrados se ha hecho uso de un sistema de verificación de correos electrónicos, de modo que, previamente a hacer uso del bot, todo usuario registrado debe acceder a un enlace de confirmación enviado a su correo electrónico.

El siguiente punto relacionado con la autenticación es el que existe entre la interfaz y *Dialogflow*, el cuál consiste en realizar llamadas a la *API* de *Google Cloud* haciendo uso de los credenciales correctos. Para ello, se ha preparado un archivo de configuración que debe ser poblado con los datos extraídos del *Gateway*, manteniendo el mismo formato que el *JSON* obtenido del mismo [Figura 5.7]. Para obtener estos datos tal y como se ha explicado en el apartado anterior, hay que acceder al panel de control de *IAMs* en la plataforma de *Google*, crear un nuevo acceso en formato *JSON* y descargar el archivo resultante.

```
1 {  
2   "type": "",  
3   "project_id": "",  
4   "private_key_id": "",  
5   "private_key": "",  
6   "client_email": "",  
7   "client_id": "",  
8   "auth_uri": "",  
9   "token_uri": "",  
10  "auth_provider_x509_cert_url": "",  
11  "client_x509_cert_url": ""  
12 }
```

Figura 5.7: Detalle del *JSON* utilizado para la autorización de la interfaz sobre *Dialogflow*.

Finalmente, el último de los puntos corresponde al que existe entre el intérprete de lenguaje natural y *API Gateway*. Para ello, tal y como se ha explicado en las funciones de *AWS Lambda* se ha creado una función llamada *ptolomeo-authorizer* que se ejecuta con cada llamada *POST* a */weather*, en la *API*, y sólo permite el acceso a la misma si el token de autenticación establecido en la cabecera es válido. Este *token*, formado por usuario y contraseña, se establece en la configuración antes del despliegue y debe ser añadido a *Dialogflow* en el panel de *fulfillment* [Figura 4.4].

5.5 Interfaz

El último de los puntos a tratar en el desarrollo es el de la interfaz del bot. Ésta, tal y como ha sido definido en el diseño, se ha concebido como un módulo específico y no como una web completa, de modo que puede ser importado y utilizado en distintos pro-

yectos. Para ello, se ha hecho uso del *framework* *Vue.js*, para facilitar el desarrollo, aunque se ha terminado compilando para su despliegue como página estática sobre *Amazon S3*.

El cuerpo principal del desarrollo ha sido la ventana sobre la que se aloja el chat, así como las pestañas de inicio de sesión y registro definidas en el apartado anterior. Para ello, se ha hecho uso de una estructura en la que arriba de todo aparece el nombre del bot con un acceso directo a los ajustes y al botón de cambio entre día y noche, dando paso más abajo al cuerpo en el que aparecen los mensajes.

Este panel [Figura 5.8], que empieza siempre con un mensaje por defecto en el que el bot, *Ptolomeo*, presenta sus capacidades, se aprovecha de la característica de *Dialogflow* de que todo mensaje produce siempre una respuesta. Esto facilita la conversación en burbujas puesto que siempre que exista una entrada se puede preparar el espacio físico para la respuesta y obtener el resultado a través de una llamada asíncrona al intérprete de lenguaje natural. Esta facilidad permite que varios mensajes no modifiquen la estructura de la interfaz o la vuelvan confusa. Un ejemplo de esta situación compleja sería un bot en el que el usuario enviase dos mensajes y, tras procesarse estos en el servidor, se obtuviesen dos respuestas cuyo orden sería propenso a la confusión debido a la naturaleza asíncrona del proceso.



Figura 5.8: Panel flotante en el que se realiza la conversación con el bot en el que se muestra el mensaje por defecto.

Finalmente, la parte inferior del panel muestra un espacio en el que el usuario escribe su mensaje antes de enviarlo, contando con un botón a la derecha para el envío. Cabe destacar que, tanto esta vista como el resto están adaptadas no solo a todo tipo de pantallas debido a su naturaleza *responsive* [Figura 5.9], sino que cuentan con una versión adaptada para teléfonos móviles y pantallas especialmente pequeñas, en la que el panel ocupa todo el espacio disponible y el texto se extrae del mismo teclado del dispositivo [Figura 5.10].

Otro de los aspectos trabajados en el *front-end* es la utilización de un *iframe* o contexto de navegación anidado que permite cargar una página web bajo el módulo, a modo de ejemplo de cómo quedaría si se integrase en la misma. Para ello, se ha hecho uso del sitio

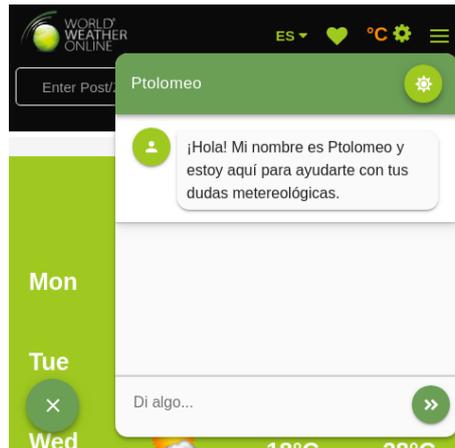


Figura 5.9: Módulo de interacción con el bot en una pantalla de reducidas dimensiones.

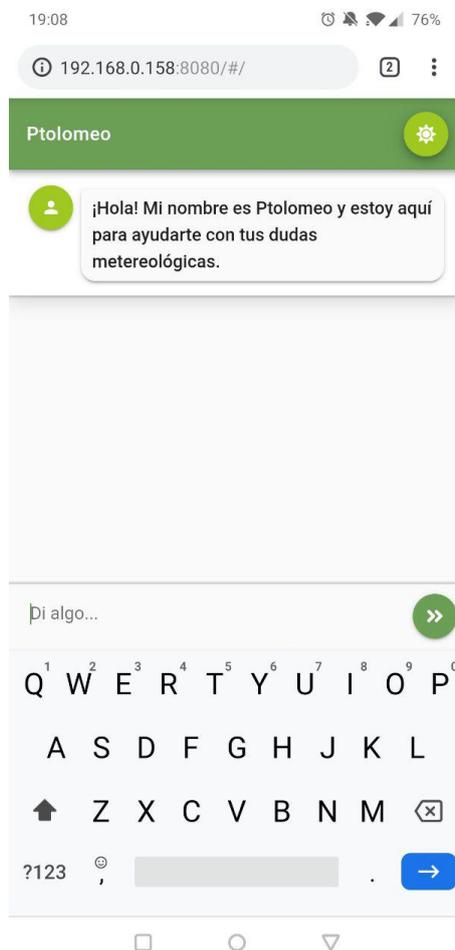


Figura 5.10: Módulo de interacción con el bot en un teléfono móvil.

World Weather Online, de donde se extraen los datos meteorológicos de la *API*. Esta comunión ha sido la que ha definido los colores y el aspecto primario del bot, tanto en su modo día como en su modo noche, primando los verdes y el blanco y negro. Estos ajustes se pueden modificar en el archivo de configuración de la interfaz, facilitando la integración con nuevos desarrollos. A continuación se muestra el aspecto global del resultado, en el que se está haciendo ya un uso realista del bot [Figura 5.11].

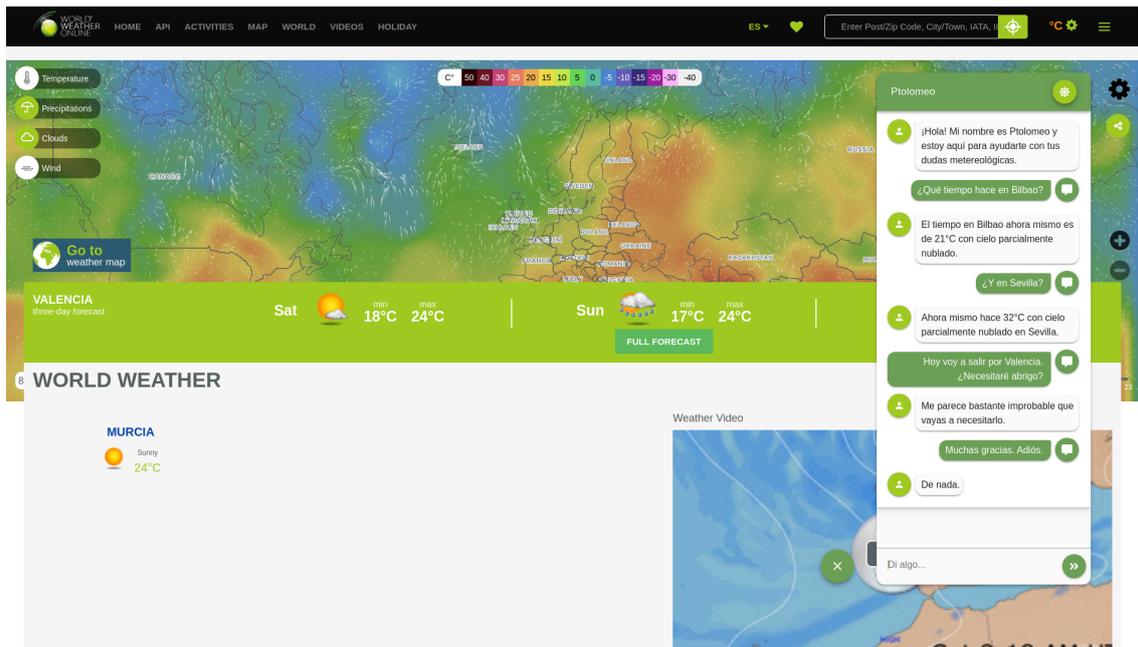


Figura 5.11: Interfaz web del bot en la que se está manteniendo una conversación con el mismo.

Finalmente, para el despliegue, se ha compilado el código haciendo uso del comando `npm run build` y se ha subido el contenido de la carpeta `dist` a un *Bucket* de *Amazon S3* público creado para el acceso global al *front-end*. Este paso será explicado con más detalles en la implantación.

CAPÍTULO 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. Podemos dividir este proceso en cuatro pasos, que se ejecutarán en el orden presentado: despliegue de la lógica de la aplicación, configuración del intérprete de lenguaje natural, preparación del servicio de autenticación y exposición del *front-end* de forma pública.

6.1 Despliegue de la lógica de la aplicación

En primer lugar, como requisito para desplegar la lógica de la aplicación, hay que contar con una cuenta de *Amazon Web Services*. Asumiendo que ésta ya está disponible y que se conocen sus credenciales de acceso, se puede continuar con el proceso. Para ello, hay que clonar el repositorio del proyecto [3] y, una vez descargado, acceder a la carpeta de */backend*. En la misma, existe un archivo *serverless.yml*, visto anteriormente 5.1 que define y automatiza el despliegue. Sin embargo, para que este sea posible y válido hay que cumplir con ciertos requisitos y realizar ciertas tareas.

El primer aspecto a considerar es el de configurar la función *ptolomeo-authorizer*. Para ello, hay que crear el archivo *lmbda/authorizer/index.js*, a través de la copia 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`. Esta política de modificar los archivos antes del despliegue existe debido a que facilita que los usuarios no utilicen variables por defecto y, por lo tanto, a que sus despliegues sean menos vulnerables. Debido a esta ventaja, será algo que se repita a lo largo de la implementación.

Una vez preparados los archivos, hay que contar con las dependencias necesarias, que son dos: tener configurado correctamente al acceso a *AWS* y disponer de la herramienta *Serverless*. Existen dos formas de abordar el primer problema: crear un archivo *./aws/credentials* y poblarlo tal y como se muestra en la figura 6.1 o hacer uso de la herramienta *awscli*¹, el cliente de *Amazon Web Services*, disponible públicamente en *GitHub*² y en los servidores de *PyPi*^{3,4}.

Hay que tener en cuenta que, en la configuración de *AWS*, no se está teniendo en cuenta la región debido a que el proyecto está preparado para ser desplegado en la región

¹Amazon.com, Inc. (2019). *Interfaz de línea de comandos de AWS* - <https://aws.amazon.com/es/cli/>

²Amazon.com, Inc. (2019). *Universal Command Line Interface for Amazon Web Services* - <https://github.com/aws/aws-cli>

³Python Software Foundation. (2019). *PyPi* - <https://pypi.org/>

⁴Amazon.com, Inc. (2019). *Universal Command Line Environment for AWS* - <https://pypi.org/project/awscli/>

```

1 [ default ]
2 aws_access_key_id=AKIAIOSFODNN7EXAMPLE
3 aws_secret_access_key=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY

```

Figura 6.1: Ejemplo de configuración de los credenciales de AWS en `./aws/credentials`.

eu-west-1 (Irlanda). Como consecuencia, si se quiere modificar, se puede cambiar en el mismo archivo `serverless.yml`, correspondiendo a ésta variable a `provider.region`. Como alternativa, se puede eliminar este campo del archivo YAML y depender completamente de la configuración establecida en `./aws/config`, configurada manualmente como en la figura 6.2 o a través de `awscli`,

```

1 [ default ]
2 region=us-west-2

```

Figura 6.2: Ejemplo de configuración de AWS en `./aws/config`.

Por otro lado, para la instalación de la herramienta *Serverless* se requiere tener instalado el gestor de paquetes `npm`, que será utilizado también para la compilación de la interfaz, y ejecutar el comando `npm install -g serverless`. Este comando instala la herramienta para el acceso desde cualquier punto de la estructura de archivos, por lo que es posible que requiera privilegios de administrador.

Solucionadas ambas problemáticas es posible ejecutar ya el comando `serverless deploy`, que se encargará de configurar AWS según el YAML y de instalar y desplegar todo lo que se requiera. Una vez terminado, el comando mostrará por pantalla una serie de datos, siendo especialmente importante quedarse con la URL obtenida, que corresponde al 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 de autenticación correctos, y será utilizada por *Dialogflow* para el *fulfillment*.

Por último, es interesante destacar que, si ya existiese un *deployment* anterior, *Serverless* directamente lo sobrescribiría. Además, si, en un momento dado, se quisiese eliminar completamente lo desplegado de *Amazon Web Services* bastaría con ejecutar el comando `serverless remove` en el mismo directorio.

6.2 Configuración del intérprete de lenguaje natural

Tras el despliegue de la lógica de la aplicación, el siguiente paso es crear y configurar un agente en *Dialogflow*, para lo que sería necesario registrarse en la plataforma mediante el uso de una cuenta de *Google*. Esto es así ya que, de forma natural, el mismo proyecto también da acceso a *Google Cloud Platform*, aspecto que se explotará más adelante.

Una vez creada la cuenta e inicializado el agente con un nombre y una descripción, tal y como se ha explicado en 5.3, el siguiente paso es explotar la funcionalidad de importación de agentes de *Dialogflow* para cargar a *Ptolomeo*, bien como nuevo bot, bien como añadido al bot ya definido. Esto es así debido a que la herramienta de *Google* no permite importar si no existe ya un agente, por lo que es necesario contar al menos con uno, aunque vaya a eliminarse, para proceder con la instalación.

Para proceder con la importación hay que acceder al panel de ajustes del agente ya existente e ir a la pestaña *Export and Import*, donde los botones *RESTORE FROM ZIP* y *IMPORT FROM ZIP* permiten reemplazar o actualizar el agente, respectivamente. El archivo para importar está disponible en el proyecto de *GitHub* con el nombre `ptolomeo.zip`

bajo el *path* `/dialogflow/release`. Como nota, el contenido del mismo puede ser consultado también en el mismo proyecto, esta vez con la dirección `/dialogflow/content`. Este ZIP incluye ya *Intents*, *Entities*, ajustes básicos como nombre y descripción y un entrenamiento básico realizado. El único aspecto a completar es el del *fulfillment*, por lo que habrá que proceder a introducir los datos correctos en el mismo apartado.

Los pasos a seguir para configurar el *fulfillment* son, por lo tanto, acceder al panel del mismo, accesible desde el menú lateral y rellenar los campos correspondientes a *URL*, con el localizador obtenido en la sección 6.1 y con los campos usuario y contraseña establecidos en `lmbda/authorizer/index.js`, definidos también en el mismo punto.

En este punto el bot estaría ya disponible para su uso desde *Dialogflow*, pero habría que realizar unos pasos más para permitir el acceso al mismo desde el *front-end*. Este acceso requeriría de exponer públicamente el agente haciendo uso de *Google Cloud Platform*, aprovechando la vinculación presentada anteriormente. Para ello, hay que acceder a los ajustes del bot y, en la pestaña general buscar la sección *Google Project*, accediendo al enlace proporcionado por *Service Account* para acceder al proyecto en la nube de Google.

Una vez allí, la pestaña por defecto, *Service accounts*, dentro del apartado *IAM & admin*, es la que permite crear cuentas para la ejecución del sistema desde fuera de Google. El proceso a seguir es el de crear una nueva cuenta, mediante el botón *CREATE SERVICE ACCOUNT* y, en esta pestaña, establecer un nombre, un identificador y una descripción para el acceso y continuar con los valores por defecto hasta la tercera pestaña, donde se tiene crear una *key*, en formato *JSON*, que será la utilizada por el *front-end*. En este punto, el intérprete de lenguaje natural ya estaría completamente configurado.

6.3 Preparación del servicio de autenticación

El siguiente paso antes de configurar y desplegar la interfaz es el de configurar el servicio de autenticación, correspondiente a *Amazon Cognito*. Para ello, hay que acceder al panel web de *AWS*, buscar el servicio y, dentro del mismo, acceder a la sección de *Manage User Pools* para crear una nueva *pool*, que requiere un nombre para ser creada.

Una vez definida, hay que acceder a los atributos de la misma para establecer como requisitos *address* y *nickname* y para establecer el correo electrónico como clave primaria en el inicio de sesión. Este comportamiento permite que pueda haber usuarios con el nombre repetido, permitiendo que, por ejemplo, dos usuarios con el mismo nombre mantengan conversaciones con el bot sin que eso genere conflictos.

A continuación, en las políticas se puede definir la seguridad mínima de la contraseña, que por defecto está configurada para requerir una longitud mínima de seis caracteres. Para sistemas en producción se recomendaría requerir más caracteres, así como la inclusión de números y caracteres especiales para prevenir ataques de fuerza bruta. Cabe destacar que, para mantener esta característica consecuente con el *front-end*, habría que modificar la vista de registro para que presentase los mismos requisitos y no confundiese al usuario.

El siguiente paso que hay que configurar es el del mensaje de verificación que se enviará al correo del usuario registrado para comprobar que éste tiene acceso a tal cuenta. Otras opciones disponibles son la utilización de protocolos de Autenticación multifactor (*MFA*) o Autenticación de dos factores (*2FA*), que permiten aumentar la seguridad mediante la comprobación de la identidad del usuario a través del uso de tecnologías como

*Google Authenticator*⁵ o SMS, o la configuración de *Identity providers*, que permiten el registro a través de *OAuth 2.0* con cuentas externas como las de *Facebook*, *Google* o *Amazon*.

Finalmente, una vez creado y configurado el servicio correctamente hay que acceder a la pestaña de Aplicaciones cliente y, una vez en ella, crear un nuevo identificador para permitir el uso del mismo desde el *front-end*. Los datos requeridos para ello son este identificador, llamado *app_client_id* y el identificador del *pool*, disponible en la vista general del mismo.

6.4 Exposición del *front-end* de forma pública

Una vez terminada toda la configuración previa, el último paso de la implantación es configurar el *front-end* para que se comuniquen con *Cognito* y *Dialogflow*, compilarlo y desplegarlo de forma pública. Todo el proceso descrito a continuación se realizará desde la carpeta */frontend* del proyecto.

El primer paso a realizar es de configurar la interfaz. Para ello, hay que acceder a la carpeta */config* del proyecto y crear los archivos *cognito/config.json* y *dialogflow/config.json* siguiendo el formato establecido por los archivos *template.config.json* disponibles en ambas carpetas. Los datos a introducir en la carpeta *dialogflow* y *cognito* serán los obtenidos a través de seguir los pasos establecidos en 6.2 y 6.3, respectivamente. Otra configuración puede ser realizada en el archivo */src/assets/default.json*, donde se establecen aspectos como el nombre del bot, el título de la página o los colores utilizados en la misma.

Una vez terminada la configuración hay que instalar las dependencias de *JavaScript* definidas en *package.json*. Para ello, basta con ejecutar el comando *npm install*. En este punto, es posible comprobar que la interfaz funciona correctamente y está bien configurada mediante la ejecución del comando *npm run dev*, que levanta un servidor local para *testing*. Si todo funciona correctamente ya se puede desplegar la aplicación, aunque para ello es necesario compilarla. Esto se realiza mediante la ejecución del comando *npm run build*, que genera la carpeta *dist* con todo el contenido necesario para un despliegue estático.

El último paso de la implantación corresponde a la exposición del *front-end* de forma pública. Para ello, hay que acceder de nuevo al panel de control de *Amazon Web Services* para crear un *bucket* público sobre *S3*. Para ello, basta con acceder a la consola del servicio de almacenamiento, crear un nuevo *bucket* y establecer el nombre y la región del mismo. A continuación, y antes de proceder con el fin de la creación, hay que acceder a la pestaña de permisos y desactivar la opción de *Block all public access*.

Una vez creado el *bucket* habría que acceder al mismo en el mismo panel y, una vez dentro, utilizar el botón de *Upload* para subir todo el contenido de la carpeta *dist*. Con los archivos ya disponibles, el *Object URL* de *index.html* sería el punto de entrada de la aplicación. De este modo, se daría por desplegada la aplicación.

⁵Google LLC. (2019). *Google Authenticator* - <https://github.com/google/google-authenticator>

CAPÍTULO 7

Pruebas y rendimiento

A continuación se detallan las pruebas que se han realizado en el sistema, tanto a nivel de validación como de carga y rendimiento. El objetivo ha sido, por lo tanto, no solo comprobar que el sistema funciona correctamente y cumple con los casos de uso sino también analizar su comportamiento ante gran número de usuarios y peticiones.

7.1 Validación del sistema

A la hora de validar el funcionamiento del sistema se ha seguido una metodología empírica. Debido a que los servicios se han ido desarrollando y configurando de forma independiente, se ha comprobado cada uno de ellos de forma individual antes de proceder con su siguiente. De este modo, se ha intentado evitar al máximo la presencia de errores arrastrados que, si se detectasen tardíamente, obligarían a realizar cambios en varios de los puntos del desarrollo.

Para proceder a la explicación de cada uno de estos análisis hay que profundizar primero en el orden de desarrollo, que ha seguido una política de adentro hacia afuera. Es decir, tal y como se ha explicado en la sección 5, primero se ha desarrollado el *back-end*, luego el intérprete de lenguaje natural y, finalmente, el *front-end*. Los pruebas de validación han sido ejecutados, por lo tanto, también en el mismo orden y siguiendo una política acumulativa. Esto es, se ha comprobado no sólo que el nuevo elemento funcione correctamente, sino que su adición no haya afectado al funcionamiento normal de su anterior.

Empezando por el *back-end*, se ha comprobado primero el funcionamiento del código ejecutándolo en un equipo local, con la finalidad de detectar errores ajenos a la implementación del mismo como función como servicio. Las pruebas han abarcado diversos casos de cada uno de los campos posibles, realizando múltiples comprobaciones en referencia a las vertientes *weather*, *weather.activity*, *weather.condition*, *weather.outfit* y *weather.temperature*, presentes en el diseño. Se ha comprobado, además, no solo el correcto funcionamiento de las llamadas, sino su gestión de errores ante la situación de no obtener los parámetros requeridos o obtenerlos pero de forma errónea o incompleta.

Una vez comprobado el código de *ptolomeo-weather* y *ptolomeo-authorizer* y de haberlo subido a la herramienta de *AWS Lambda*, se ha hecho uso del entorno de test de la misma. Para ello, se han creado varios *JSONs* de peticiones similares a la de una llamada de *fulfillment* típica y se han ejecutado los test para comprobar que los datos se estaban obteniendo correctamente y la respuesta tenía el formato adecuado. También se han realizado los mismos test desde *API Gateway*, una vez expuesta la función, para comprobar que la conexión era correcta y, por tanto, que tanto el input como el output de la misma

fueran procesados y coherentes entre sí. Cabe destacar que, en este punto, se ha comprobado también que los datos introducidos en el despliegue de la función, como memoria asignada, eran válidos. Tal y como se aprecia en el resultado [Figura 7.1], una ejecución de ejemplo, tras un *warm up*, tarda menos de 100 ms y requiere menos de 128 MB de memoria.

The screenshot shows the execution result for a Lambda function. It includes a success status, a details section with a JSON response, a summary table, and a log output section.

Execution result: succeeded (logs)

Details

The section below shows the result returned by your function execution.

```
{
  "fulfillmentText": "El domingo en Barcelona va a haber cielo parcialmente nublado, con temperaturas sobre 15.0°C",
  "outputContexts": []
}
```

Summary

| | |
|---|--------------------------------------|
| Code SHA-256 | Request ID |
| lsGxvksE9UWeFX2o3JXAhW1fTQ1wDHUHqICPtr5p9r8 | be09922d-a0c1-4a66-a267-e02a23e1013b |
| = | |
| Duration | Billed duration |
| 27.50 ms | 100 ms |
| Resources configured | Max memory used |
| 128 MB | 56 MB |

Log output

The section below shows the logging calls in your code. These correspond to a single row within the CloudWatch log group corresponding to this Lambda function. [Click here](#) to view the CloudWatch log group.

```
START RequestId: be09922d-a0c1-4a66-a267-e02a23e1013b Version: $LATEST
END RequestId: be09922d-a0c1-4a66-a267-e02a23e1013b
REPORT RequestId: be09922d-a0c1-4a66-a267-e02a23e1013b  Duration: 27.50 ms
Billed Duration: 100 ms          Memory Size: 128 MB      Max Memory Used: 56 MB
```

Figura 7.1: Resultados de la ejecución de un test sobre *ptolomeo-weather* en *AWS Lambda*.

Finalizada la comprobación del *back-end* y habiendo ya configurado el intérprete de lenguaje natural se han realizado sobre la misma interfaz de *Dialogflow* múltiples pruebas relacionadas con cada uno de los *intents*. De este modo, no sólo se ha intentado comprobar su correcto funcionamiento, indistintamente de los datos solicitados, sino que también se han utilizado varias configuraciones de oraciones para comprobar que eran interpretadas o, en su defecto, para entrenar al agente y prevenir futuros errores. Para ello, la herramienta de *Google* cuenta con la opción de comprobar entradas previas y validar o corregir el comportamiento del intérprete frente a ellas.

Por último, para la validación del *front-end* se ha seguido varias veces el ciclo presentado en los casos de uso, sección 3.4, correspondiente a registro, inicio de sesión y conversación, correspondiente, a su vez, a los ciclos utilizados en la validación del agente en *Dialogflow* [Figura 7.2]. Se han realizado pruebas también relacionadas con el diseño, como comprobar el diseño *responsive* y el funcionamiento correcto en dispositivos móviles. Además, en el caso específico del registro y el inicio de sesión, se han realizado otras pruebas como intentar utilizar direcciones de correo electrónico inválidas o ya utilizadas,

contraseñas de correo demasiado cortas, caracteres peligrosos o, directamente, campos en blanco [Figura 7.3].



Figura 7.2: Resultado de una de las pruebas de validación del agente en el *front-end*.

Una captura de pantalla de un formulario de registro. El campo 'Nombre' está vacío y tiene un icono de usuario a la derecha. El campo 'Correo electrónico' contiene el texto 'correoinvalido' y tiene un icono de correo rojo con una 'X' a la derecha. El campo 'Contraseña' contiene tres puntos y tiene un icono de contraseña roja con una 'X' a la derecha. Debajo del campo de contraseña, se muestra el mensaje de error 'Mínimo de 6 caracteres' y el número '3'. En la parte inferior del formulario, hay dos botones: 'INICIA SESIÓN' y 'REGISTRARSE'.

Figura 7.3: Resultado de una prueba en la que se intenta el registro con datos incorrectos o inválidos

7.2 Pruebas de carga

Una vez terminado el desarrollo del proyecto y comprobado su correcto funcionamiento, se ha procedido a realizar un análisis de su rendimiento. Este aspecto resulta muy interesante ya que es aquí donde se establece la principal diferencia entre utilizar una arquitectura con servidor o una *serverless*. Para ello se han realizado distintas pruebas aunque, tal y como se explicará más adelante, no todas han resultado tan eficaces como deberían de ser.

Empezando por el diseño de las mismas, se ha planteado la ejecución de tres pruebas distintas: aumento de 0 a 50 peticiones por segundo durante 30 segundos, aumento de 0 a 50 peticiones por segundo durante 60 segundos y aumento de 0 a 50 peticiones por segundo durante 120 segundos. La intención de éstas era ver cómo respondía el código ejecutado en *AWS Lambda* a cambios de escalado muy rápidos, debido a la naturaleza de su escalado automático. Cabe destacar que no se han realizado pruebas relacionadas con el rendimiento máximo o el punto de saturación debido a que *AWS Lambda* puede soportar hasta 1000 ejecuciones concurrentes por cuenta [17], por lo que se ha considerado que se saturarían con más facilidad los equipos en los que se ejecutaban los test que el mismo servicio de *Amazon*.

Tras realizar los test haciendo uso del kit de herramientas para pruebas de carga *Artillery.io* se ha podido observar que, en las métricas de *AWS CloudWatch*, servicio utilizado para la medición del uso de recursos de las funciones de *AWS Lambda*, el tiempo mínimo de consulta de recursos utilizados es de un minuto, por lo que no se han podido extraer correctamente los datos de carga de los test ejecutados en 30 y 60 segundos y, por lo tanto, solo se ha mantenido el de 120 segundos. No se ha procedido a analizar nuevos test con una mayor duración porque el objetivo principal de las pruebas era detectar picos en la latencia máxima, característicos del cold start de las funciones *serverless* [18], y no se iban a obtener datos más precisos por realizar cargas durante más tiempo.

Manteniendo, por lo tanto, un único test como válido, se ha procedido a analizarlo y extraer de él los datos necesarios para el análisis del proyecto. Empezando por cómo se ha realizado, el bombardeo de carga se ha hecho desde un equipo personal, ajeno a la infraestructura de *Amazon*, y se ha atacado directamente a *API Gateway* en vez de al *front-end* o a *Dialogflow*, ya que se ha considerado más representativo del coste real del *serverless*. El hecho de utilizar un equipo local en vez de una instancia de *EC2* ha tenido la finalidad de detectar la latencia en una situación de uso real, en vez de la que se obtendría desde dentro de la misma red. Como nota, la latencia media entre el equipo con el que se ha bombardeado el *back-end* y *amazonaws.com* era de 108 ms en el momento de la prueba.

La medición desde la herramienta se ha realizado en periodos de 10 segundos, motivo por el que el test, de 120 segundos, presenta datos desde el segundo 10 al segundo 130, correspondientes, respectivamente, la primera y a la última carga detectada del sistema dentro del periodo de medición. El aumento en las llamadas por segundo, *requests per second*, ha sido progresivo [Figura 7.4]. El número total de peticiones realizadas a lo largo del test ha sido de 3101, obteniendo un total de 3101 (100 %) códigos de respuesta 200 OK.

Los resultados obtenidos [Figura 7.5], muestran la latencia con la que se han obtenido a lo largo del mismo periodo las respuestas a las peticiones. Se pueden destacar bastantes aspectos de esta gráfica como que la latencia mediana se mantiene constante por mucho que aumente la carga, así como la carga mínima, y que la diferencia entre éstas, e incluso el *p75* es muy pequeño, llegando a ser, entre el mínimo y el *p75* de sólo 12 % en el momento de carga máxima.

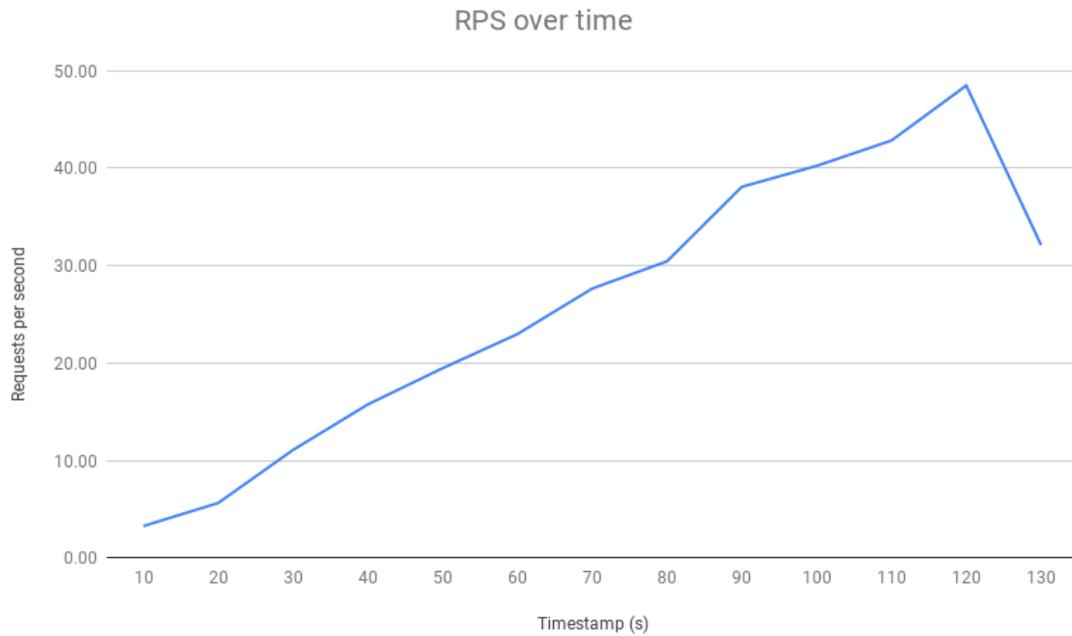


Figura 7.4: Recta de carga correspondiente a las peticiones por segundo a lo largo del test.

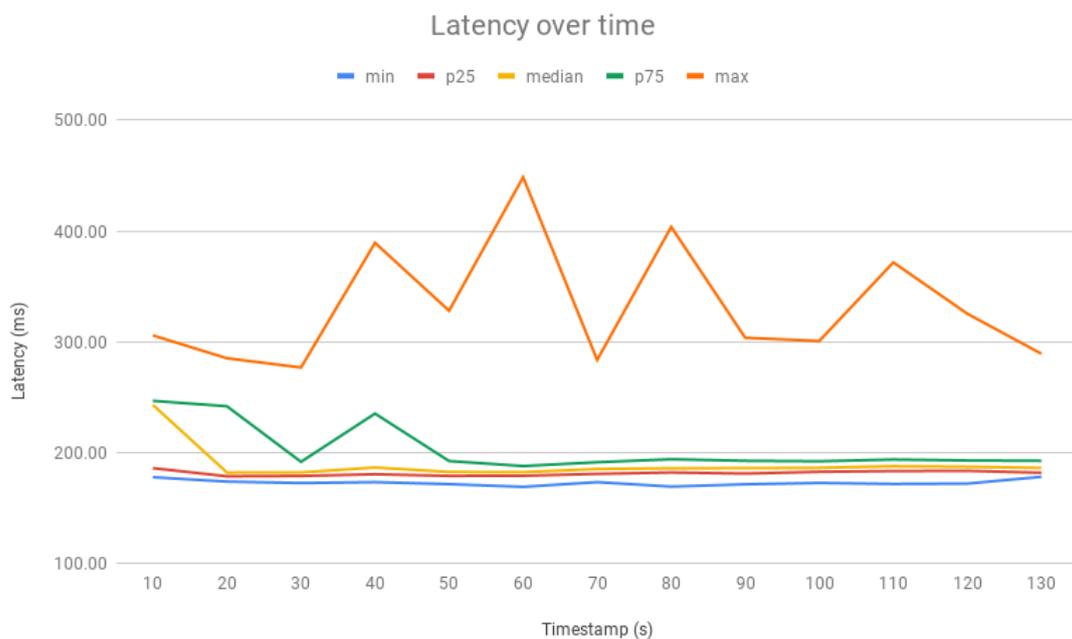


Figura 7.5: Evolución de los valores de latencia obtenidos lo largo del proceso de test.

Sin embargo, el principal aspecto a destacar de la gráfica es que pese a que la latencia del percentil 75 es bastante baja, los tiempos máximos llegan a ser muy altos en relación, mostrando picos muy destacables en el periodo previo a los segundos 10, 40, 60, 80 y 110. Este factor podría ser explicado gracias al problema típico de las funciones como servicio conocido como *cold start*. Para explicar esto hay que entender que las funciones como servicio se ejecutan dentro de un contenedor y que, para aumentar la concurrencia, el proceso a seguir es aumentar el número de contenedores activos ejecutando el código. Esto hace que la carga asociada a crear un nuevo contenedor se junte al tiempo de respuesta

de la petición que lo levanta, dando lugar a picos y a retardos en la ejecución de ciertas llamadas [18].

Para comprobarlo, se ha accedido al *CloudWatch*, el servicio de obtención de métricas de *Amazon Web Services*, buscando encontrar una relación directa entre los picos y la instanciación de nuevos contenedores. Sin embargo, tal y como se ha explicado al principio del capítulo, las métricas obtenidas por el servicio con relación a *AWS Lambda* se obtienen cada minuto, por lo que no es posible localizar el segundo exacto en el que se ha realizado la instanciación. Sin embargo, a raíz de analizar los datos presentados y de juntarlos con los resultados del cliente, se ha podido dibujar una posible línea que explicaría la existencia de los picos y justificaría el problema conocido como *cold start*. Por supuesto, existen distintas formas de evitar o minimizar esta problemática; sin embargo, no son el objetivo de estudio de este proyecto.

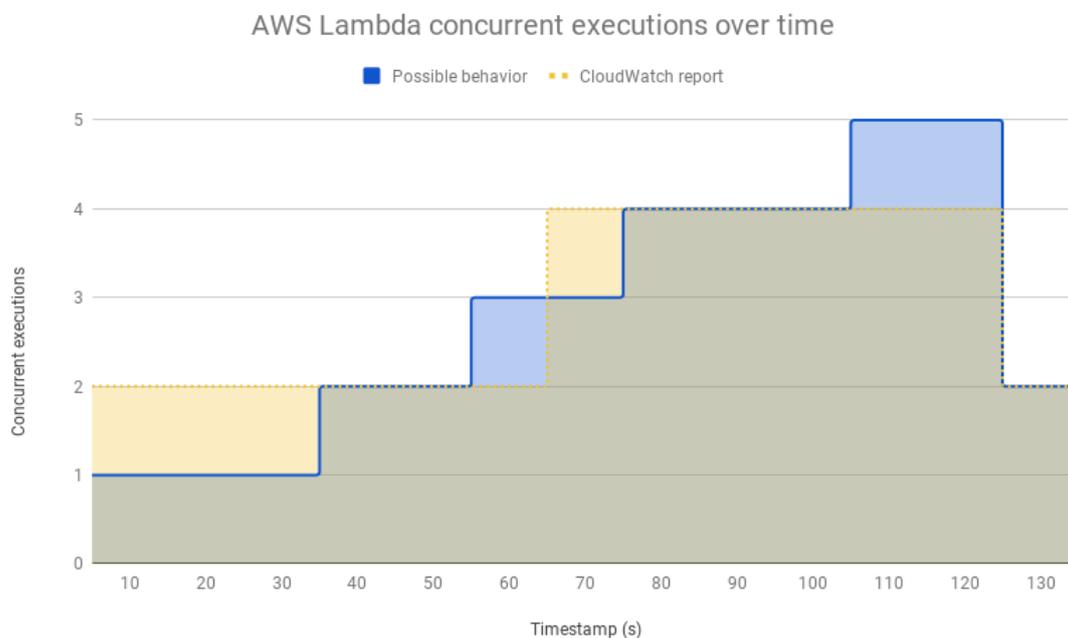


Figura 7.6: Datos obtenidos por *CloudWatch* a lo largo del test junto a un posible comportamiento por parte de *AWS Lambda*.

En conclusión, se puede decir que el *back-end* escala perfectamente gracias al uso de funciones como servicio, que no solo garantizan un coste mínimo, pagando solo por el uso, sino que ofrecen un tiempo de ejecución prácticamente lineal por mucho que aumente la carga. Es interesante destacar que existe el problema del *cold start*, que hace que algunas funciones tarden en ejecutarse mucho más de tiempo del habitual. Sin embargo, *Amazon* suele gestionar este tiempo por debajo de los 1000 ms [19], lo cuál hace que no suponga un aprieto real a ninguna aplicación a la que se le haga un uso estándar.

CAPÍTULO 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 cumplido 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 han aprovechado los resultados obtenidos del mismo, hay 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

En la presentación de los objetivos del proyecto, sección 1.2, se recogían los requisitos que se querían satisfacer a lo largo del trabajo. Posteriormente, en la formalización del problema, sección 3.3, se descomponían aquellos relacionados con la implementación para profundizar en cada uno de los elementos que debían de asegurarse para entender como completo cada uno de los retos presentados. De esta manera, y una vez terminado el proyecto, se va a realizar una comprobación, punto por punto, del cumplimiento de los objetivos en base al nivel de cumplimiento de las tareas que se le asociaron.

Despliegue de un *front-end* estático, sin servidor asociado y con funciones de autenticación

El aspecto fundamental de este objetivo era el de desarrollar una interfaz estática que no dependiese de un servidor que ejecutase código acorde a las peticiones del usuario o de un *host* que ofreciese acceso público de forma activa al sistema. De este forma, se pretendía desplegar la interfaz siguiendo un patrón sin servidor, en el que el único coste asociado, tanto a nivel económico como de recursos, fuese el estrictamente necesario para la transferencia de los archivos estáticos al usuario.

Para solucionar este problema se ha hecho uso de código cliente formado por la unión de *HTML*, *CSS* y *JavaScript*, obtenido de la compilación de un proyecto desarrollado con el *framework* *Vue.js*. El despliegue se ha realizado sobre el servicio de almacenamiento de objetos de *Amazon S3*, cuyo principal coste está asociado al tráfico generado a través de

peticiones *GET*, siendo el precio de almacenamiento del código del proyecto, inferior a 0,0025 \$ mensuales, despreciable.

A raíz de ello, no solo se elimina el coste asociado a la ejecución de código en el servidor de un patrón Modelo-vista-controlador (*MVC*), sino que también se elimina el coste asociado al mantenimiento de un anfitrión, que debe ser mantenido aunque no se esté usando activamente. Finalmente, esta solución ofrece también las ventajas de ser naturalmente escalable y altamente distribuida frente al modelo clásico en el que, si solo hubiese una o pocas máquinas contratadas, el servicio sería susceptible a fallos de hardware, apagones o caídas completas de zonas de disponibilidad.

Por otro lado, la autenticación se ha conseguido haciendo uso de un servicio de mantenimiento y gestión de usuarios externo, *Amazon Cognito*, el cuál permite eliminar también el coste asociado a la gestión de usuarios en una base de datos propia o contratada. Estas soluciones clásicas, asociadas al mantenimiento de archivos en un servidor y, por tanto, de un estado, requieren de un coste asociado al mantenimiento de los recursos que no existe si estos se delegan a un servicio externo.

La ventaja obtenida es especialmente notoria en servicios con una cantidad de usuarios limitada, debido a que, mientras que el servicio de *AWS* ofrece una capa gratuita para los primeros 50000 usuarios, no es posible utilizar y pagar por una porción de una base de datos. Además, en el segundo caso, no solo se requiere una implementación completamente funcional desde un inicio sino que, además, si se desea una replicación mínima y alta disponibilidad deberían de ser utilizados, como mínimo, tres equipos, hecho que dispararía el precio en desarrollos pequeños.

Cabe destacar que, pese a que todas las tareas asociadas al objetivo han sido cumplidas, se quería garantizar también, como añadido, la facilidad de autenticar a los usuarios gracias al uso de otras de sus cuentas en otros servicios como *Google* o *Facebook* y no ha sido posible. El problema, que existiría igual si se hubiese hecho uso de una base de datos clásica, ha estado asociado al uso de un *SDK* de *JavaScript* concreto y no al uso de un servicio de autenticación externo. Como consecuencia, no se considera un problema de la arquitectura o del diseño sino de la implementación de la misma.

En conclusión, todas las tareas asociadas a este objetivo han sido completadas de forma efectiva. Además, tal y como se había planteado en un inicio, se han obtenido mejores resultados a nivel de coste y gestión de recursos que si se hubiese hecho uso de una arquitectura con servidor clásica.

Entrenamiento de un analizador de lenguaje natural capaz de obtener respuestas mediante la comunicación con otro servicio

Las tareas derivadas del entrenamiento del analizador de lenguaje natural eran dos: desarrollar un sistema capaz de interpretar y extraer información de oraciones relacionadas con el contexto de la meteorología y vincularlo con un sistema externo de forma que fuese posible obtener respuestas que dependiesen de procesamiento o de fuentes de datos externas. Para la resolución de ambas tareas se ha hecho uso de la herramienta de *Google Dialogflow*.

La primera de las tareas, correspondiente al desarrollo de un sistema capaz de interpretar y extraer información de oraciones relacionadas con el contexto de la meteorología, podría dividirse, a su vez, en tres: la interpretación de oraciones simples, la interpretación de oraciones derivadas y la interpretación de oraciones de interacción, simples y ajenas al contexto meteorológico. De otra forma, y en el mismo orden, estas subtareas deberían de entenderse como la interpretación de oraciones que requieren recogida de datos y cuya respuesta depende de un servicio externo, la interpretación de oraciones derivadas,

iguales a las anteriores pero que además dependen de un contexto anterior, y la interpretación de oraciones simples, que no requieren recogida de datos ni procesamiento de los mismos.

La primera de las subtareas ha sido solucionada gracias al uso de, por una parte, *entities* y, por otra parte, la vinculación a un servicio externo a través de un *webhook* conocido como *fulfillment*. En el primer caso, las entidades, asociaciones de palabras a significados dentro de un rango, han permitido definir qué tipo de datos se estaban buscando, qué valores eran válidos y qué información había que extraer de ellos. Un ejemplo de esto sería asociar las palabras "bufanda" y "gorro" como *ropa de invierno*, dentro de la entidad *outfit*, de modo que, estableciendo *@outfit* como parámetro deseado en una interacción, la herramienta de *Dialogflow* se encargue de darle un valor válido.

En el segundo caso, la vinculación a un servicio externo, correspondiente al *back-end*, a través de un *webhook* ha permitido al intérprete resolver las consultas que dependen de procesamiento para generar una respuesta. En el caso del proyecto, este procedimiento ha correspondido al necesario para obtener el estado meteorológico en un lugar y un tiempo determinados y decidir, en función de los parámetros de entrada, que tipo de respuesta devolver. Esta subtarea soluciona, de forma paralela, la otra de las tareas, correspondiente a la obtención de respuestas mediante la comunicación con otro servicio

Por otro lado, la segunda de las subtareas, interpretar oraciones que dependan de un contexto anterior, ha sido solucionada gracias al uso de los contextos de *Dialogflow*. Estos contextos son variables temporales en formato *JSON* que son establecidas posteriormente a cada interpretación de una función meteorológica y que almacenan la entrada de las mismas. De esta forma, cuando se llama a una función derivada, carente de información como tipo de consulta, localización o tiempo, estos datos se generan accediendo al histórico y obteniéndolos del contexto más reciente, siempre que éste no haya caducado.

Finalmente, para la interpretación de oraciones simples sin recogida de datos ni procesamiento se ha optado por hacer uso de *pools* de entradas y respuestas válidas. De este modo, toda interacción que se asocie con un nivel de certeza suficiente a una una de las oraciones del *pool* de entradas generará una respuesta extraída de forma pseudoaleatoria del *pool* de respuestas.

Implementación de un *back-end* basado en funciones como servicio que solucione consultas de carácter meteorológico

El aspecto fundamental de este objetivo era el de desarrollar una o una serie de funciones como servicio para solucionar consultas de carácter meteorológico, así como hacer uso de las herramientas necesarias, también sin servidor, para exponer y explotar tales funciones como servicio. De esta forma se pretendía presentar un *back-end* sin servidor asociado, de modo que el coste estuviese, casi en su totalidad, directamente y puramente relacionado con la ejecución del código y no con la adquisición, el mantenimiento o el cubrimiento de costes asociados a la utilización de uno o más servidores.

Para ello, se ha hecho uso de la herramienta *AWS Lambda*, que permite implementar arquitecturas del tipo *Back-end as a Service* asociando el total del coste de la ejecución del código a la ejecución del mismo. Además, con la intención de minimizar ese coste, se han desarrollado las funciones, programadas en *Python* y en *JavaScript*, pensando en que su duración media y su uso de memoria sea inferior al establecido por los bloques mínimos de la plataforma de Amazon. Es decir, para que tarden menos de 100 ms y ocupen menos de 128 MB de memoria RAM en su ejecución. De este modo, se ha conseguido explotar todavía más la capa gratuita, que permite ejecutar hasta un millón de solicitudes mensuales sin cargo.

Por otro lado, para exponer éstas funciones se ha hecho uso del servicio *API Gateway*, que permite definir *RESTful APIs* en las que la llamada a un *path* expuesto haciendo uso de un método concreto dispare otros servicios. En este caso, las funciones Lambda desarrolladas.

Como consecuencia, el coste asociado al uso de este servicio está directamente relacionado con las llamadas realizadas al mismo, por lo que se calcula en relación al tráfico que está recibiendo la aplicación y no al mantenimiento del mismo *proxy*. Es decir, se eliminan todos los costes asociados al mantenimiento de uno o más equipos que actúen como *proxy* inverso y cuyos costes deben cubrirse pese a que la aplicación no esté siendo utilizada por ningún usuario. Además, *API Gateway* ofrece, naturalmente, un servicio replicado y altamente disponible, que supondría un coste adicional en el caso de querer ser replicado en soluciones *on-premise* o basadas en el mantenimiento de máquinas físicas o virtuales.

Como conclusión, se han conseguido solucionar todos los problemas asociados a las tareas derivadas del objetivo de forma efectiva. De esta forma, no solo se ha obtenido un *back-end* capaz de responder a llamadas asociadas a consultas meteorológicas sino que, además, se ha implementado sin tener que realizar un desembolso inicial y obteniendo un coste de utilización estrictamente ligado al uso y no al mantenimiento.

Realización de pruebas de rendimiento a modo de comparativa entre una arquitectura serverless y una con servidor

Uno de los objetivos principales de este proyecto era el de conocer las ventajas y desventajas de hacer uso de arquitecturas *serverless* en contraposición a una arquitectura clásica con servidor. Este objetivo, repetido a lo largo del proyecto, ha sido analizado a lo de varios puntos, tal y como se puede ver en las conclusiones previas. Sin embargo, más allá de los beneficios asociados al coste asociado al uso y a la ausencia de un fuerte desembolso inicial, la finalidad de este punto era el de conocer si estas ventajas, a priori competitivas, suponían una diferencia en el rendimiento de la aplicación.

Para ello, se han realizado diversas pruebas en las que se ha comprobado el tiempo de ejecución de una función *serverless* sobre *AWS Lambda*. Los resultados obtenidos de tales pruebas han mostrado que, en situaciones estándar, no existe una diferencia apreciable entre la ejecución en una máquina estándar y en un contenedor ejecutado como servicio. Sin embargo, en ocasiones, debido al fenómeno conocido como *cold start*, las soluciones basadas en funciones como servicio presentan una latencia notablemente mayor a la latencia media.

Este hecho, pese a que puede ser minimizado con ciertas técnicas, presenta un problema real en situaciones en los que el tiempo de ejecución sea crítico o estricto. Ejemplos de esto podrían ser sistemas en tiempo real o servicios web, como servidores de videojuegos, en los que mayor latencia se refleje, de forma proporcional, en la percepción negativa del servicio por parte del usuario o en la completa incapacidad de uso normal. En otro tipo de servicios, como aquellos que toleran una respuesta asíncrona variable, como un bot conversacional, un sistema de autenticación o un servicio de mensajería, este problema no sería percibido con la misma crudeza y, por lo tanto, serían más propensos a la adaptación.

Otro de los aspectos a destacar es que las funciones como servicio ofrecen un escalado rápido y preciso que puede ser utilizado para optimizar los costes, mientras que un sistema clásico requiere de un número mínimo de equipos que, aunque escalable, raramente representa la carga real del sistema. Poniendo un ejemplo, si un servicio autoescalable basado en servidores está configurado para aumentar el número de máquinas cuando el

uso total de CPU supere el 80 % y, en un momento dado, se requiere un 90 % de CPU, se estará pagando por dos máquinas cuando no se llega a utilizar plenamente ni una de ellas.

Por otro lado, el ajuste de grano fino proporcionado por las funciones como servicio permite ajustar el coste económico al uso real, o, como mínimo, ajustarlo en bloques más pequeños como los tiempos de ejecución de 100 ms de *AWS Lambda*. Además, cabe destacar que implementar un servicio de **autoescalado** *on-premise* no resulta trivial, y requiere de un gran número de máquinas preparadas y, por lo tanto, previamente pagadas para ello, mientras que la solución *cloud* presenta un soporte nativo y alta tolerancia al aumento de la solicitud de recursos.

En conclusión, las pruebas de rendimiento han mostrado que las funciones como servicio no son únicamente una alternativa mejor en la vía económica. Esto se debe a que no solo ofrecen un coste más ajustado al uso y no requieren de un desembolso inicial, sino que también funcionan de forma similar a nivel de rendimiento y toleran mejor situaciones de adaptación a cambios de recursos y de escalado puntual. Pese a ello, hay que tener presente que la latencia añadida puede resultar crítica en sistemas en tiempo real, por lo que no deberían de ser usadas para estos casos.

Liberación del proyecto como contribución de código abierto a la comunidad

El último de los objetivos de este proyecto era la liberación del código asociado al mismo, en cada una de sus vertientes, en función de código abierto, de modo que fuese disponible y accesible por la comunidad. Éste, por su naturaleza *Open Source*, debía no solo ser accesible para consulta, sino también permitir la utilización, la modificación y la redistribución, comercial o no, de forma gratuita.

Para garantizar este objetivo se ha liberado todo el código generado en la plataforma de *GitHub* [3] con licencia *Apache 2.0* [4]. Además, a lo largo del desarrollo se ha contribuido con múltiples desarrollos *Open Source* de otras plataformas, de modo que problemas relacionados con el proyecto han sido solucionados o reportados en otros proyectos de la misma índole en forma de *issues* o *pull requests* y se plantea seguir colaborando con ellos, especialmente con aquellos que han servido de base o ayuda para la ejecución de éste.

En conclusión, se ha conseguido solucionar el último de los objetivos ampliamente, ya que no solo se ha liberado este proyecto sino que se ha contribuido y se plantea seguir contribuyendo con otros proyectos de la comunidad.

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 cenit de todo el trabajo realizado y todo lo aprendido en los estudios de Grado. Se profundizará más en este aspecto en el apartado 8.3.

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*. De este modo, he podido explorar no solo hacia los adentros de esta plataforma sino también conocer otros proveedores y profundizar en sus diferencias.
- Profundización en el paradigma de las Funciones como Servicio, descubriendo de forma práctica *AWS Lambda*, así como trucos que permiten eliminar completamente el servidor como desplegar el *front-end* sobre un *Bucket* 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*¹.

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

Pese a que, tal y como se ha comentado en la sección 8, todos los objetivos del proyecto han sido alcanzados correctamente, todo proceso es dado a la mejora y al rediseño, bien sea por carencias del trabajo previo como por la incorporación de nuevas herramientas y métodos. Hay que tener en cuenta también la limitación surgida de ser éste un Trabajo de Fin de Grado, con una carga lectiva de 12 créditos *ECTS* y con unos plazos establecidos, lo cuál ha obligado a que los objetivos se seleccionen y acoten en función de una carga de trabajo máxima. A continuación se detallan brevemente qué aspectos hubiesen sido incorporados al trabajo realizado:

- **Creación de aplicaciones nativas para dispositivos móviles.** Pese a que existe una versión *responsive* optimizada para teléfonos móviles de la interfaz, hubiese sido interesante profundizar en la creación de aplicaciones nativas en las plataformas de *Android*² e *iOS*³.

¹Leslie Lamport. (2019). *LaTeX - A document preparation system* - <https://www.latex-project.org/>

²Google LLC. (2019). *Android* - https://www.android.com/intl/es_es/

³Apple Inc. (2019). *iOS 12* - <https://www.apple.com/es/ios/ios-12/>

- **Integración del uso de mensajes de voz al *front-end*.** Debido a que *Dialogflow* cuenta con herramientas de *text-to-speech* y *speech-to-text*, hubiese sido interesante preparar el *front-end* para enviar y procesar peticiones de audio.
- **Solución de los problemas de autenticación relacionados con *OAuth 2.0*.** Tal y como se ha presentado en la sección 5.4, los problemas surgidos con el uso del *SDK* de *Amazon Cognito* para *JavaScript* han hecho imposible la autenticación a través de cuentas externas como las de *Google*, *Facebook* o *GitHub*. Las soluciones planteas pasarían por profundizar en la utilización del *SDK* y de las soluciones de los usuarios o hacer uso de servicios externos como *Firebase Auth*⁴.
- **Comparación a nivel de coste y rendimiento entre distintas plataformas *FaaS*.** Una profundización en el capítulo 7, de pruebas y rendimiento, hubiese sido la de explotar distintas plataformas para el desarrollo de funciones como servicio y compararlas entre ellas. Además, hubiese sido muy interesante mostrar especial interés en la diferencia entre las soluciones basadas en la nube, como *AWS Lambda*, *Azure Functions* y *Google Cloud Functions* y las basadas en soluciones *on-premise*, como *OpenWhisk*⁵ o *Kubeless*⁶.
- **Ampliación del *back-end* para ofrecer servicios más complejos.** Debido a las limitaciones del proyecto se ha planteado el uso de un bot de consulta meteorológica ya que permitía afrontar sencillamente todos los problemas que querían ser estudiados. Sin embargo, una aproximación más profunda, en la que se intentasen desarrollar aplicaciones más complejas como un asistente virtual, un planificador de viajes o un asistente de primer nivel podrían ser mucho más interesantes para conocer la envergadura que es capaz de soportar la arquitectura planteada.
- **Integración con otros servicios del proveedor *cloud*.** Debido a que las consultas meteorológicas se han realizado a través de la comunicación con una *API* externa, las funciones de *AWS Lambda* no se han comunicado con otros servicios de la misma plataforma. Sin embargo, hubiese sido interesante tratar de integrar las funciones con otros servicios, de modo que, por ejemplo, se pudiese almacenar y consultar información de *DynamoDB*⁷, se pudiese explotar el reconocimiento de *AWS Rekognition*⁸ o se pudiesen generar mensajes traducidos al idioma del cliente con *AWS Translate*⁹.

En conclusión, pese a que los objetivos del proyecto sí que han sido conseguidos, los trabajos del futuro podrían estar relacionados con la ampliación del proyecto obtenido. De este modo se podría obtener un entorno completamente funcional, a nivel de plataforma, que permitiese a los clientes desarrollar sus propios bots con lógica asociada, capaces de comunicarse con otros servicios del proveedor cloud y de atender a usuarios móviles o no a través de mensajes de voz y texto.

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 explo-

⁴Google LLC. (2019). *Firebase Authentication* - <https://firebase.google.com/docs/auth>

⁵The Apache Software Foundation. (2019). *Apache OpenWhisk* - <https://openwhisk.apache.org/>

⁶Bitnami. (2019). *Kubeless* - <https://kubernetes.io/>

⁷Amazon.com, Inc. (2019). *Amazon DynamoDB* - <https://aws.amazon.com/es/dynamodb>

⁸Amazon.com, Inc. (2019). *Amazon Rekognition* - <https://aws.amazon.com/es/rekognition/>

⁹Amazon.com, Inc. (2019). *Amazon Translate* - <https://aws.amazon.com/es/translate/>

tación 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.
- **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 el bot implementado. 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, sistemas basados en *machine learning*, árboles de decisión 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 permitido 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* [3]. Como consecuencia, éste es un proyecto de código abierto distribuido bajo la licencia *Apache 2.0* [4],

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.

Glosario

A | B | C | D | E | F | H | M | O | P | S | W

A

API

Conjunto de subrutinas, funciones y procedimientos (o métodos, en la programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro *software* como una capa de abstracción. [3](#)

autoescalado

Proceso o rutina mediante el cuál un sistema escala de forma automática. [61](#)

B

back-end

Parte del software que procesa la entrada recibida desde el *front-end*. [2](#)

C

cloud

Tecnología destinada a utilizar a distancia recursos de ejecución o almacenamiento. [2](#)

cloud computing

Paradigma que permite ofrecer servicios de computación a través de una red, que usualmente es Internet. [12](#)

cloud-native

Dicho de un sistema que ha sido desarrollado para ser ejecutado directamente sobre una plataforma *cloud*. [13](#)

concurrency

Simultaneidad en la ejecución de múltiples tareas interactivas. [25](#)

D

disponibilidad

Grado en que un sistema, subsistema o equipo se encuentra en un estado específico operable y comprometible al comienzo de una misión, cuando la misión se solicita en un momento desconocido, es decir, aleatorio. [10](#)

E

endpoint

Dispositivo informático remoto que se comunica con una red a la que está conectado. [2](#)

escalado

Adaptación y respuesta de un sistema con respecto al rendimiento del mismo a medida que aumenta de forma significativa la carga o el número de usuarios. [12](#)

F**framework**

Conjunto estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática particular que sirve como referencia, para enfrentar y resolver nuevos problemas de índole similar. [1](#)

front-end

Parte del *software* que interactúa con los usuarios. [2](#)

H**hybrid cloud**

Entorno de *Cloud Computing* que utiliza una mezcla de soluciones *on-premise* y *cloud*. [12](#)

M**microservicio**

Pequeños servicio que se ejecuta en su propio proceso y se comunican con otros a través de una o más *APIs*. [2](#)

O**on-premise**

Tipo de instalación de una solución en la que ésta se lleva a cabo dentro del servidor y la infraestructura (TIC) de la empresa. [8](#)

P**path**

Camino que toma el *software* para acceder a los datos contenidos en una unidad de almacenamiento. [30](#)

S**serverless**

Modelo de ejecución de computación en la nube en el cual el proveedor de la nube ejecuta el servidor y administra dinámicamente la asignación de recursos de la máquina. [2](#)

speech-to-text

Protocolo de reconocimiento de voz en el que se transforma el audio en texto. [6](#)

W**webhook**

Sistema utilizado por una aplicación para enviar información en tiempo real a otras aplicaciones. [7](#)

Bibliografía

- [1] López, G., Quesada, L. & A. Guerrero, Luis. (2017). Alexa vs. Siri vs. Cortana vs. Google Assistant: A Comparison of Speech-Based Natural User Interfaces. *Advances in Intelligent Systems and Computing, (AISC, volume 592)*.
- [2] Jurafsky, D & Martin, J. H. (2018). Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition. 7-9.
- [3] Solé Fonte, M. (2019). Ptolomeo. *GitHub* - <https://github.com/msolefonte/ptolomeo>
- [4] The Apache Software Foundation. (2019). Apache License, Version 2.0. *The Apache Software Foundation* - <https://www.apache.org/licenses/LICENSE-2.0>
- [5] Perez, S. (2017). Amazon Lex, The Technology Behind Alexa Opens up to Developers. *Tech Crunch*.
- [6] Freaney, N. (2015). Facebook Acquires Voice Recognition Startup Wit.ai. *Time Magazine* - <http://time.com/3655070/facebook-voice-recognition-wit-ai/>
- [7] Amazon.com, Inc. (2006). Amazon Web Services Launches “Amazon S3”. *The Amazon blog dayone* - <http://phx.corporate-ir.net/phoenix.zhtml?c=176060&p=irol-newsArticle&ID=830816>
- [8] Handy, A. (2014). Amazon introduces Lambda, Containers at AWS re: Invent. *BZ Media LLC, 14*.
- [9] Evans, B. (2017). The Top 5 Cloud-Computing Vendors: #1 Microsoft, #2 Amazon, #3 IBM, #4 Salesforce, #5 SAP. *Forbes* - <https://www.forbes.com/sites/bobevans1/2017/11/07/the-top-5-cloud-computing-vendors-1-microsoft-2-amazon-3-ibm-4-salesforce-5-sap/>
- [10] ZDNet Staff. (2008). Ray Ozzie announces Windows Azure. *ZDNet* - <https://www.zdnet.com/>
- [11] Malawski, M., Figiela, K., Gajek, A., & Zima, A. (2017, August). Benchmarking heterogeneous cloud functions. *European Conference on Parallel Processing, Singer, Cham, 415-426*.
- [12] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2016). Borg, omega, and kubernetes.
- [13] Li, Z., Zhang, Y., & Liu, Y. (2017). Towards a full-stack devops environment (platform-as-a-service) for cloud-hosted applications. *Tsinghua Science and Technology, 22(01), 1-9*.

-
- [14] The HFT Guy. (2016). Google Cloud is 50% cheaper than AWS. *The HFT Guy: A developer in London* - <https://thehftguy.com/2016/11/18/google-cloud-is-50-cheaper-than-aws/>
- [15] Loukides, M. (2012). What is DevOps? *O'Reilly Radar* - <http://radar.oreilly.com/2012/06/what-is-devops.html>
- [16] Baldini, I., Cheng, P., J. Fink, S., Mitchell, N., Muthusamy, V., Rabbah, R., Suter, P. & Tardieu, O. (2017). The serverless trilemma: function composition for serverless computing. *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. ACM, New York, NY, USA, 89-103.
- [17] Amazon.com, Inc. (2019). Managing Concurrency - AWS Lambda. *AWS Documentation* - <https://docs.aws.amazon.com/lambda/latest/dg/concurrent-executions.html>
- [18] Giménez-Alventosa, V., Moltó, G., & Caballer, M. (2019). A framework and a performance assessment for serverless MapReduce on AWS Lambda. *Future Generation Computer Systems*, Mar 2019, 11-19.
- [19] Pérez, A., Moltó, G., Caballer, M., & Calatrava, A. (2018). Serverless computing for container-based architectures. *Future Generation Computer Systems*, 83, 50-59.
- [20] Shilkov, M. (2018). Serverless: Cold Start War. *Mikhail Shilkov* - <https://mikhail.io/2018/08/serverless-cold-start-war/>