Universidad de las Ciencias Informáticas

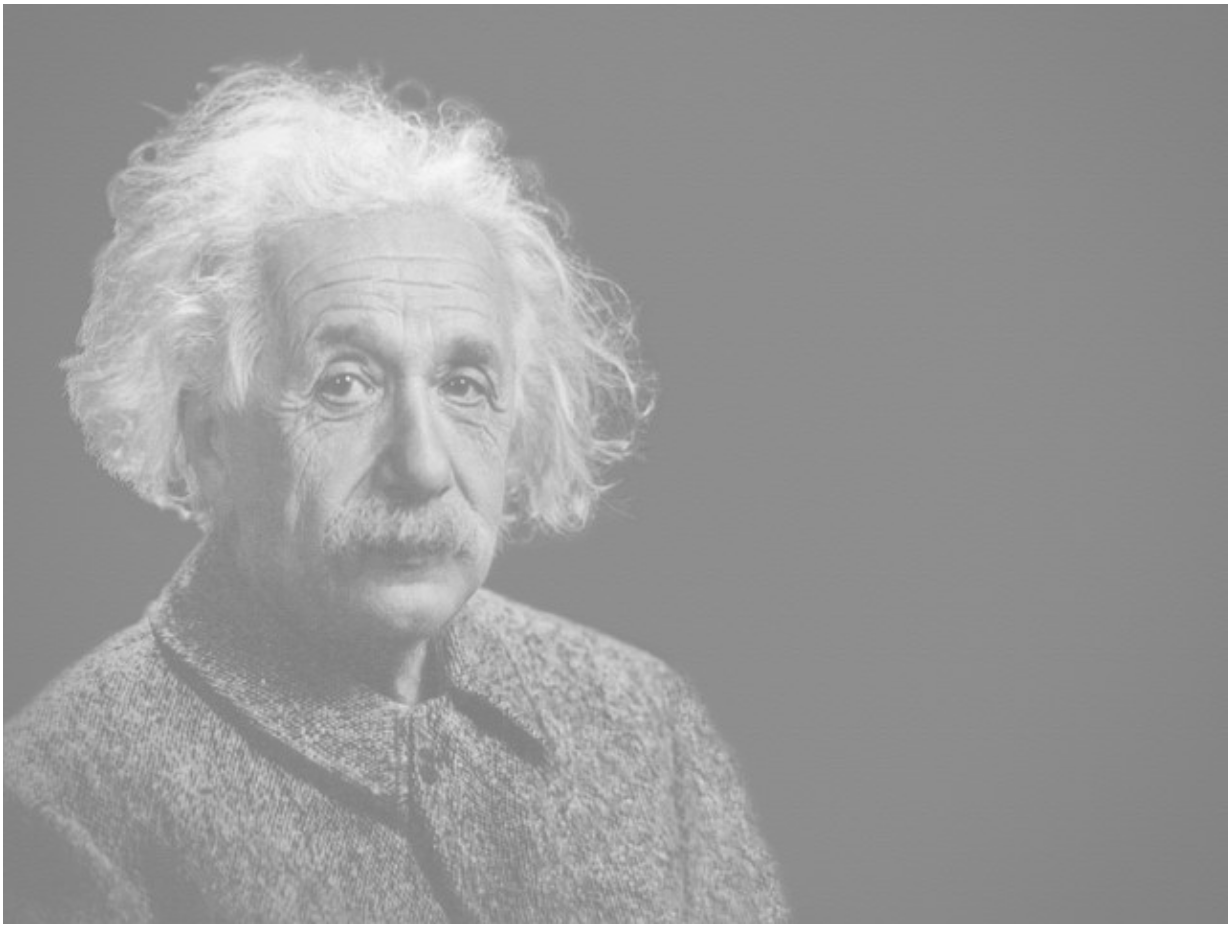
Facultad CITEC

Arquitectura base de despliegue el para sistema decidimOS

Trabajo de Diploma para optar por el título de Ingeniero en Ciencias Informáticas

Autor(es): Ariel Rodríguez Lau
Tutor(es): Ing. Lester Collado Rolo
La Habana, 17 de noviembre de 2022

Arquitectura base de despliegue el para sistema decidimOS



Nunca consideres un estudio como una obligación,
sino como una oportunidad para penetrar
en el bello y maravilloso mundo del saber

(Albert Einstein)

Arquitectura base de despliegue el para sistema decidimOS

Autor: Ariel Rodríguez Lau
Tutor(es): Ing. Lester Collado Rolo

17 de noviembre de 2022

Universidad de las Ciencias Informáticas

Declaración de Autoría

Declaro que soy el único autor del trabajo de diploma “Arquitectura Base de Despliegue para decidimOS” y se autoriza a la Facultad CITEC de la Universidad de las Ciencias Informáticas a hacer uso del mismo en su beneficio.

Para que así conste firmo la presente a los ____ días del mes de _____ del año 2022.

Autor:	**Tutor:**
_____	_____
Ariel Rodríguez Lau	Ing. Lester Collado Rolo

Dedicatoria

A mis padres por ser los motores de mi vida les dedico este trabajo, por todo el esfuerzo durante todo el proceso estudiantil de mi vida.
A mi hermano por ser un ejemplo a seguir en mi vida y ayudarme siempre que lo necesito.
A la claudiña de mi alma, gracias por llegar a mi vida y ser un apoyo más.
A mi vida Milo por ser la alegría de la familia.
A todos mis amigos por ser la mejor familia que se puede elegir.
A todos los que creyeron en mi siempre. A todos GRACIAS, MUCHAS GRACIAS. Los quiero mucho.

Agradecimientos

Me gustaría agradecer primero que nada a mi familia por quererme y apoyarme incondicionalmente en todas las facetas de mi vida, en los momentos más duros.

Mamá, Papi, Tata, Claudiña, nunca les podré devolver todo el amor y la confianza que me han dado, ustedes son mi historia.

A mis tíos/as en especial a mi tía Deysi, mis primos/as y mi abuela Iluminada por siempre preocuparse por mí.

Agradecer al profesor Lester quien me ha dado la confianza y en el empuje necesario en el desarrollo de la investigación, por los ánimos que me dio cuando sentía que no podía.

Al profesor Luna por ser un amigo más y aconsejarme durante el ciclo de vida de este proyecto de tesis.

Mención especial al Señor Ingeniero Víctor Daniel Vela Cuevas por ser un referente en LaTeX.

A mis amigos de siempre: Alejandro García, Alejandro Núñez y Dariel López, ustedes han sido mi pequeña familia en esta nuestra nueva casa, gracias además a mis demás compañeros con los cuales ha sido un orgullo compartir durante estos 5 años: Marcelo, Raikol, Anás, Jose, Yosley, Oscar, Enmanuel el pina, Brian el Calvo, Ernesto, Susana, Reynier, Marlon, Hassam, Rey, Manolo gracias por tantas noches de estudio, de sufrimiento, de alegría, estoy orgulloso de ustedes también.

Gracias a mi amigo Rey Marante por ser un pilar en estos últimos meses, a las demás personas que me han apoyado y me han mantenido fuerte. Gracias a los muchachos del centro de desarrollo CREAD por permitirme ser uno más durante estos meses de arduo trabajo.

RESUMEN

La tarea de investigación emerge en el marco de trabajo del grupo de desarrollo del Centro de Análisis y Representación de Datos (CREAD) para la gestión de un sistema electoral (decidimOS) derivado de las necesidades del grupo de proyecto de integrar los módulos para obtener componentes altamente integrados y propiciar soluciones escalables y confiables.

La arquitectura definida en el transcurso de la investigación detalla los elementos necesarios para satisfacer el correcto desarrollo de software de CREAD. El éxito de la misma se debe por las decisiones arquitectónicas tomadas a cabo a lo largo del ciclo de vida del proyecto, la definición de las herramientas necesarias para su realización y de la plataforma tecnológica para su construcción, así como los mecanismos arquitectónicos y la descripción de la arquitectura por medio de las vistas arquitectónicas.

Para la selección de dichas herramientas se tuvo en cuenta que fueran de software libre, libre de pago y bajo licencia libre. Se aplicaron técnicas de evaluación de arquitectura como Software Architecture Analysis Method (SAAM) y Architecture Trade-off Analisys Method (ATAM) incluyendo sus procedimientos de Árbol de Utilidad y las técnicas basadas en escenarios.

El presente trabajo constituye una guía para arquitectos de desarrollo de software utilizando la herramienta Docker y definiendo una arquitectura.

PALABRAS CLAVE

Arquitectura de Software, Plataforma Tecnológica, Desarrollo de Software, Técnicas, Herramientas.

ABSTRACT

The research task emerges in the framework of the development group of the Center for Data Analysis and Representation (CDAR) for the management of an electoral system (decidimOS) derived from the needs of the project group to integrate the modules to obtain components highly integrated and enable scalable and reliable solutions.

The architecture defined in the course of the investigation details the necessary elements to satisfy the correct development of CREAD software. Its success is due to the architectural decisions made throughout the life cycle of the project, the definition of the tools necessary for its realization and the technological platform for its construction, as well as the architectural mechanisms and the description of architecture by means of architectural views.

For the selection of these tools, it was taken into account that they were free software, free of payment and under a free license. Architecture evaluation techniques such as Software Architecture

Analysis Method (SAAM) and Architecture Trade-off Analysis Method (ATAM) were applied, including their Utility Tree procedures and scenario-based techniques.

This work constitutes a guide for software development architects using the Docker tool and defining an architecture.

KEYWORDS

Software Architecture, Technological Platform, Software Development, Techniques, Tools.

Índice general

Agradecimientos **11**

1 Introducción **16**

2 Fundamentación teórica **20**

2.1 Conceptos asociados 20

2.1.1 Arquitectura de *Software* 20

2.1.2 Interoperabilidad 20

2.1.3 Docker 21

2.2 Arquitectura de *Software* 21

2.3 Estado de la cuestión 22

2.3.1 Despliegue e Ingeniería de *Software* 22

2.3.2 Metodología de Desarrollo de *Software* 27

2.3.3 Lenguajes 29

2.3.4 Marco de trabajo 30

2.3.5 Herramientas utilizadas para el desarrollo del sistema 31

2.3.6 Entorno de Desarrollo Integrado 33

2.3.7 Herramientas de gestión de base de datos 34

2.4 Conclusiones parciales 37

3 DISEÑO DE LA SOLUCIÓN PROPUESTA AL PROBLEMA CIENTÍFICO **39**

3.1 Plan de Despliegue 40

3.2 Requisitos: 41

3.2.1 Definición de Requisito: 41

3.2.2 Requisitos Funcionales: 42

3.2.3 Requisitos No Funcionales: 42

3.2.4 Historias de Usuario: 44

3.2.5 Vista de Despliegue: 46

3.2.6 Diagrama de Flujo: 48

3.3 Patrones GRASP: 52

3.4 Modelo de Vistas Arquitectónicas 54
3.4.1 Arquitectura de Infraestructura: 54
3.5 Conclusiones Parciales: 54

4 VALIDACIÓN DE LA SOLUCIÓN PROPUESTA 56
4.1 Necesidad de evaluar una arquitectura 56
4.2 Atributos de Calidad 58
4.3 Modelos de Calidad 59
4.4 Técnicas de evaluación de arquitectura 61
4.5 Métodos de evaluación: 66
4.6 Validación de la arquitectura: 70
4.7 Conclusiones parciales: 73

5 Conclusiones Generales 75

6 Aportes Realizados 76

Bibliografía 79

1 Introducción

Las llamadas Tecnologías de la Información y la Comunicación son los recursos y herramientas que se utilizan para la administración y distribución de la información a través de elementos tecnológicos, como: ordenadores, teléfonos, televisores, etc. A través del paso del tiempo la utilización de este tipo de recursos se ha incrementado y actualmente presta servicios de utilidad como el correo electrónico, la búsqueda y el filtro de la información, descarga de materiales, comercio en línea, entre otras (Costa Rica, 2020).

Las Tecnologías de la Información y las Comunicaciones se encuentran profundamente arraigadas en el tejido social y forman parte de la manera en la que los negocios y sistemas son conducidos en la actualidad, la forma en la que los seres humanos se relacionan, se aprende e incluso influye en la alimentación. Las TIC cada día cobran mayor importancia, pese a su ubicuidad casi total, los beneficios de estas tecnologías siguen siendo desiguales, es decir, el acceso a la “sociedad mundial de la información” no otorga de inmediato la condición de miembro.

Esto obedece a muchas razones. Los países de recursos limitados se esfuerzan por suministrar electricidad y conectividad a las aldeas alejadas y a zonas urbanas de rápida expansión; la provisión de TIC para escuelas y oficinas sigue afectada por una grave escasez de fondos, los cuales provienen de gobiernos locales que se esfuerzan por prestar los servicios básicos; y por atractivos que resulten los últimos modelos de computadoras personales, tabletas y teléfonos inteligentes, simplemente no son asequibles para muchas familias y negocios. Pero el factor económico y el nivel de los precios no son los únicos responsables (Cutrell, 2022).

El sistema electoral y las organizaciones políticas forman parte, tanto en el ámbito dogmático como en el ámbito orgánico, de la Constitución Política del Estado; pues, de un lado tienen relación directa con los derechos políticos y electorales, y de otro lado, constituyen elementos fundamentales de la organización y funcionamiento del poder político, es decir, de la organización del Estado y del gobierno.

La presencia de organizaciones políticas y su manifestación en el Estado, reguladas a través del sistema electoral, permiten que los órganos políticos definidos por la Constitución adquieran vida política, perduren, evolucionen, se desarrollen, etc. Y esta actividad política se produce conforme a los principios que sustentan al Estado democrático. En consecuencia, el sistema electoral de sí es muy importante y se instituye con el elemento legitimador de mayor significación del sistema político (Ojeda, 2019).

Contexto del problema:

El despliegue de software son todas las actividades que hacen que un sistema de software esté disponible para su uso. El proceso de implementación general consiste en varias actividades interrelacionadas con posibles transiciones entre ellas. El Proyecto decidimOS del Centro de Representación y Análisis de Datos ha identificado como un proceso clave en el proceso de desarrollo de software, la fase de despliegue del software. Esta fase de despliegue en ocasiones, ha aumentado los tiempos de entrega de los productos afectando los acuerdos con clientes y atrasando los cronogramas acordados.

El Centro de Representación y Análisis de Datos de la UCI ha garantizado desde 2015 los procesos de elecciones, su soporte tecnológico y gestión del centro de información del hoy Consejo Electoral Nacional. Existen varias soluciones de software para el desarrollo de los principales procesos electorales: elecciones municipales, nacionales y referendos.

Dichas soluciones requieren de una evolución tecnológica para ser más eficientes, flexibles ante solicitudes de cambios. Igualmente, al formarse el Consejo Electoral Nacional con estructuras permanentes de gestión se han consolidado todos los procesos que debe gestionar el órgano. Sus funciones están definidas en la Constitución de la República y detalladas por la Ley no 72.

El proyecto decidimOS tiene como objetivo desarrollar un ecosistema de software a nivel nacional para la gestión de procesos electorales basado en una arquitectura orientada a componentes, desacoplada y con buenas prácticas tecnológicas para garantizar la escalabilidad y la funcionalidad de las soluciones de acuerdo a los procesos electorales a ejecutarse.

El ecosistema de software decidimOS consiste en una plataforma formada por varios componentes para la gestión de los procesos electorales en la que se han definido componentes bases tales como la seguridad, la gestión de las personas, las estructuras electorales y algunos tecnológicos para la gestión de trazas, caché y notificaciones.

La problemática de este trabajo está dada además, de las carencias que posee la entidad, la falta de recursos, la falta de personal y de personal cualificado para resolver o gestionar dicha tarea.

Problema científico: ¿Cómo contribuir al despliegue de los productos del proyecto decidimOS?

Objeto de estudio: Arquitecturas para despliegue de *softwares*.

Campo de acción: Arquitecturas para despliegue de productos del proyecto decidimOS.

Objetivo general: Desarrollar arquitectura para el despliegue de los productos del proyecto decidimOS.

Objetivos específicos:

1. Análisis de fundamentos teóricos para el desarrollo de arquitecturas de despliegue de productos de software.
2. Desarrollo de la arquitectura de despliegue.
3. Validar a través de técnicas de evaluación de la arquitectura la solución desarrollada.

Posibles resultados:

1. Arquitectura base para el despliegue de productos del proyecto decidimOS.
2. Artefactos de ingeniería para la documentación del desarrollo.

Los Métodos teóricos utilizados para cumplir las tareas son los siguientes:

- Histórico-Lógico: Permitió realizar un estudio de los principales conceptos asociados a la utilización de arquitecturas de despliegue en la informática actual
- Hipotético-deductivo: se utilizó para guiar la investigación desde el planteamiento del problema hasta la verificación de la solución a partir de las validaciones, orientando la secuencia lógica de las tareas que se realizaron.
- Análisis y Síntesis: Se utiliza para identificar y analizar las diversas funcionalidades de las empresas desarrolladoras que despliegan *softwares* a nivel internacional, que pueden ser aplicadas en la solución.
- Modelación: Permitió la creación del modelo conceptual para entender el contexto en el que se enmarca la investigación.

Los Métodos empíricos utilizados para cumplir las tareas fueron:

- Entrevista: Permitió obtener la información necesaria relacionada con los problemas presentes en los centros donde se desea implementar dicha arquitectura de software

Capitulo 1. Fundamentación Teórica: En este capítulo se definen los conceptos más importantes para un mejor entendimiento de la investigación a realizar, se explican las herramientas, metodologías, tecnologías y lenguajes que serán utilizados para dar solución a la problemática existente.

Capítulo 2. Análisis y diseño de la arquitectura de despliegue de la plataforma decidimOS. En este capítulo se definió la solución que se propone y se realizó la selección de los requerimientos del sistema que se pretende implementar. Se realizó el estudio y análisis de la solución que se propone, se modelan y describen los recursos ingenieriles necesarios acorde a la metodología DevOps que representan las funcionalidades del sistema, aplicando los patrones de arquitectura y diseño seleccionados.

2 Fundamentación teórica

En este capítulo se expone la fundamentación teórica del trabajo, incluyéndose en el mismo, el estudio de otras soluciones informáticas existentes en la actualidad, de las metodologías, modelos de desarrollo de *software* y herramientas a utilizar.

2.1. Conceptos asociados

Los conceptos asociados son de vital importancia para una mejor comprensión del negocio, para ello se tuvo en cuenta los conceptos de Arquitectura de *Software*, Interoperabilidad, Docker.

2.1.1. Arquitectura de *Software*

A medida que aumentan el tamaño y la complejidad de los sistemas de *software*, el diseño, la especificación y el análisis del sistema general estructura se convierte en un tema crítico. Los problemas estructurales incluyen la organización de un sistema como una composición de componentes; estructuras de control global, los protocolos de comunicación, sincronización y acceso a datos; la asignación de funcionalidad para diseñar elementos; la composición del diseño elementos; distribución física; escalado y rendimiento, y dimensiones de la evolución (Garlan, 2001).

2.1.2. Interoperabilidad

La interoperabilidad es la capacidad de dos o más componentes de *software* para cooperar a pesar de las diferencias de lenguaje, interfaces y plataforma de ejecución. Es una forma escalable de reutilización, siendo preocupado por la reutilización de recursos del servidor fuentes por los clientes cuyos mecanismos de acceso pueden ser incompatibles con el enchufe con sockets del servidor (Wegner, 1996).

2.1.3. Docker

Docker es una tecnología de virtualización de contenedores. Entonces, es como una máquina virtual [VM] muy liviana. Además de crear contenedores, proporcionamos lo que llamamos un flujo de trabajo de desarrollador, que en realidad se trata de ayudar a las personas a crear contenedores y aplicaciones dentro de contenedores y luego compartirlos con sus compañeros de equipo (Anderson, 2015).

2.2. Arquitectura de *Software*

No es, sino hasta 1989 donde Mary Shaw con la publicación de su libro "Sistemas de mayor escala que requieren abstracciones de mayor nivel" ("Larger Scale Systems Require Higher-Level Abstractions") define el concepto "arquitectura de *software*" de la siguiente manera:

"[. . .][Arquitectura de *software* es el estudio de la estructura a gran escala y el rendimiento de los módulos de los sistemas de *software*. Aspectos importantes de la arquitectura de un sistema; incluyendo la división de funciones entre los módulos del sistema, los medios de comunicación entre módulos, y la representación de la información compartida.]" Shaw, 1989

En 1994 Mary Shaw y David Garlan publican el libro "Una introducción a la Arquitectura de *Software*" ("An Introduction to *Software* Architecture") donde se introduce la necesidad de la creación de la arquitectura de *software* como disciplina científica, cuyo objeto de estudio es la determinación de un conjunto de paradigmas que establezcan una organización del sistema a alto nivel, la interrelación entre los distintos componentes que lo conforman y los principios que orientan su diseño y evolución (Garlan, y otros, 1994).

Según Booch la Arquitectura de Software es: "[. . .][Toda la arquitectura es diseño, pero no todo el diseño es arquitectura. La arquitectura representa las decisiones de diseño significativas que le dan forma a un sistema, donde lo significativo puede ser medido por el costo del cambio.]" Booch y col., 1999

El objetivo final de la arquitectura es identificar los requisitos que producen un impacto en la estructura del *software* y reducir los riesgos asociados con la construcción del mismo.

Por tanto, la arquitectura de *software* se define como la infraestructura de *software* dentro de la cual se pueden especificar, implementar y ejecutar los componentes de la aplicación que brindan funcionalidad al usuario. Se argumenta que una descripción de la arquitectura del *software* debe incluir, a través de los niveles de granularidad, los conceptos básicos y las restricciones

dentro de las cuales se especificarán los componentes de la aplicación, los componentes arquitectónicos que abordan las preocupaciones técnicas, la infraestructura de integración y las estrategias arquitectónicas utilizadas para abordar de manera concreta la calidad de requisitos (Solms, 2012).

En el proceso de desarrollo de la arquitectura de *software* se definen los siguientes aspecto:

- Criterios de selección para evaluar las alternativas de solución
- Informe de nuevas tecnologías
- Soluciones alternativas
- Criterios para la selección final
- Informe de evaluación de los activos de productos
- Relación entre componentes de producto y requisitos establecidos
- Soluciones, evaluaciones y fundamentaciones
- Arquitectura del producto
- Diseño del componente de producto
- Paquete de datos técnicos

2.3. Estado de la cuestión

2.3.1. Despliegue e Ingeniería de *Software*

El despliegue de *software* suele ser un proceso intensivo, en algunos casos, repetitivo y con elevadas probabilidades de cometer errores, debido a que en la mayoría de los casos se realiza de forma manual por los operadores de sistemas y el número acciones a desarrollar es elevado. Si no se ejecutan todos los elementos involucrados en el despliegue de forma precisa, la aplicación no funcionará de forma satisfactoria (Hernández Yeja y col., 2016a).

La seguridad de la información se encarga de proteger la confidencialidad, integridad y disponibilidad de los activos de información, ya sea en el almacenamiento, procesamiento o transmisión. Se alcanza con la aplicación de políticas, educación, entrenamiento, conciencia y tecnología (Whithman, 2011). Otra definición establece la protección de los sistemas frente al acceso, uso, divulgación, alteración, modificación o destrucción de la información (Jason, 2014). En este sentido, para ser más efectivo, la seguridad de la información debe integrarse en todo el Ciclo de Desarrollo del *Software* (SDLC por sus siglas en inglés) (Kissel, 2008), por lo que el *software* debe ser diseñado, desarrollado y desplegado con una mentalidad segura y la aplicación de estrategias que minimicen la probabilidad de la exposición e impacto a las amenazas.

En el caso de la etapa de despliegue de *software*, la misma constituye el momento en que el código se ha completado y la aplicación está lista para pruebas rigurosas y de validación que confirmen que no hay vulnerabilidades conocidas de seguridad; además, se configura de forma tal que no sea objeto de ningún comprometimiento de la información. Se deben planificar cuidadosamente las actividades necesarias desde la perspectiva de seguridad en torno al despliegue del *software* .(Hernández Yeja y col., 2016b)

Tradicionalmente el diseño de software se ha realizado con arquitectura monolítica, en la que el software se estructura de forma que todos los aspectos funcionales quedan acoplados y sujetos en un mismo programa. En este tipo de sistema, toda la información está alojada en un servidor, por lo que no hay separación entre módulos y las diferentes partes de un programa están muy acopladas. Esto genera un problema a largo plazo, ya que se trata de un sistema no escalable de manera sencilla. Por eso aparece la arquitectura de **microservicios**.

Detectada la necesidad por parte de la empresa de realizar cambios en el software e implementarlo de forma fácil y rápida nacen los microservicios. La idea es dividir los sistemas en partes individuales, permitiendo que se puedan tratar y abordar los problemas de manera independiente sin afectar al resto. La **arquitectura** de microservicios funciona con un conjunto de pequeños servicios que se ejecutan de manera autónoma e independiente. (Ver figura 2.1)

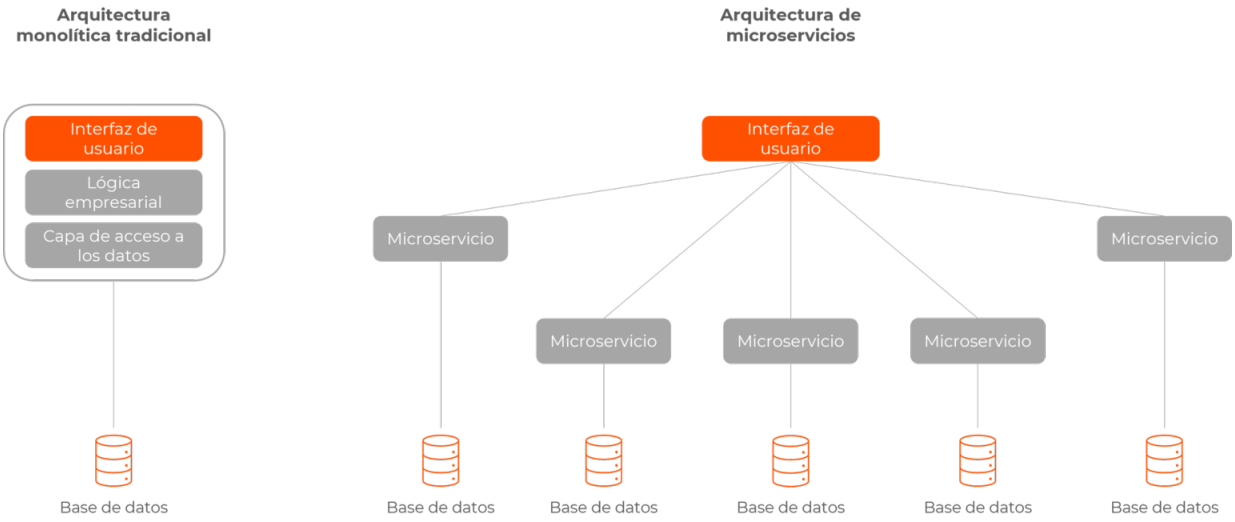


Figura 2.1: Arquitectura de microservicios

En esta arquitectura cada microservicio es un código que puede estar en un lenguaje de programación diferente, y que desempeña una función específica. Los microservicios se comunican entre sí a través de una API (Interfaz de Programación de Aplicaciones) y cuentan con sistemas de almacenamiento propios, lo que evita la sobrecarga y la caída de la aplicación.

Los microservicios han creado estructura IT más adaptables y flexibles porque si se quiere modificar solamente un servicio no es necesario alterar el resto de la infraestructura. Cada uno de los servicios se puede desplegar y modificar sin que ello afecte a otros servicios o aspectos funcionales de la aplicación.

A continuación se detallan ventajas y desventajas de esta arquitectura para su mejor comprensión. (Services, 2022)

Ventajas:

- Modularidad: Al tratarse de servicios autónomos, se pueden desarrollar y desplegar de forma independiente. Además un error en un servicio no debería afectar la capacidad de otros servicios para seguir trabajando según lo previsto.
- Escalabilidad: Como es una aplicación modular, se puede escalar horizontalmente cada parte según sea necesario, aumentando el escalado de los módulos que tengan un procesamiento más intensivo.
- Versatilidad: Se pueden usar diferentes tecnologías y lenguajes de programación. Lo que permite adaptar cada funcionalidad a la tecnología más adecuada y rentable.

- Implementación sencilla: Los microservicios permiten la integración y la entrega continuas, lo que facilita probar nuevas ideas y revertirlas si algo no funciona. El bajo costo de los errores permite experimentar, facilita la actualización del código y acelera el tiempo de comercialización de las nuevas características.
- Rapidez de actuación: El reducido tamaño de los **microservicios** permite un desarrollo menos costoso, así como el uso de "contenedores de software" permite que el despliegue de la aplicación se pueda llevar a cabo rápidamente.
- Mantenimiento simple y barato: Al poder hacerse mejoras de un solo módulo y no tener que intervenir en toda la estructura, el mantenimiento es más sencillo y barato que en otras arquitecturas.
- Agilidad: Se pueden utilizar funcionalidades típicas (autenticación, trazabilidad, etc.) que ya han sido desarrolladas por terceros, no hace falta que el desarrollador las cree de nuevo.

Desventajas:

- Alto consumo de memoria: al tener cada microservicio sus propios recursos y bases de datos, consumen más memoria y CPU.
- Inversión de tiempo inicial: al crear la arquitectura, se necesita más tiempo para poder fragmentar los distintos microservicios e implementar la comunicación entre ellos.
- Complejidad en la gestión: si contamos con un gran número de microservicios, será más complicado controlar la gestión e integración de los mismos. Es necesario disponer de una centralización de trazas y herramientas avanzadas de procesamiento de información que permitan tener una visión general de todos los microservicios y orquesten el sistema.
- Perfil de desarrollador: los microservicios requieren desarrolladores experimentados con un nivel muy alto de experiencia y un control exhaustivo de las versiones. Además de conocimiento sobre solución de problemas como latencia en la red o balanceo de cargas.
- No uniformidad: aunque disponer de un equipo tecnológico diferente para cada uno de los servicios tiene sus ventajas, si no se gestiona correctamente, conducirá a un diseño y arquitectura de aplicación poco uniforme.
- Dificultad en la realización de pruebas: debido a que los componentes de la aplicación están distribuidos, las pruebas y test globales son más complicados de realizar.
- Coste de implantación alto: una arquitectura de microservicios puede suponer un alto coste de implantación debido a costes de infraestructura y pruebas distribuidas.

Actualmente gracias a los contenedores como Docker y orquestadores como Kubernetes, se han podido cambiar los servidores web tradicionales por contenedores virtuales mucho más pequeños y adaptables.

Contexto Internacional: La concepción de una Arquitectura de Referencia para Nubes Privadas constituye una herramienta para describir y desarrollar un sistema o arquitectura específica partiendo de las necesidades de las TIC de una institución y sus restricciones (UIT-T, 2012b). No obstante, actualmente las arquitecturas propuestas son genéricas para la Nube, no particularizando en las características propias de cada uno de los modelos de despliegue. En pos de definir entonces, una Arquitectura de Referencia para Nubes Privadas, se analizaron críticamente desde la perspectiva de dicho modelo, las Arquitecturas de Referencias emitidas por las organizaciones líderes en los procesos de estandarización de la Computación en la Nube (UIT-T, 2012a): el NIST, la UIT y la DMTF, para determinar las semejanzas y particularidades de sus propuestas, y cómo estas se ajustan a una Nube Privada gestionada por la entidad en sus propias instalaciones con soporte para IaaS.

La Arquitectura de Referencia abarca su Ecosistema del Negocio y su Arquitectura de Referencia Funcional. El Ecosistema define los actores y sus roles. Se encuentra formado de manera general por el proveedor, el usuario y el contribuidor, los cuales poseen su ubicación y rol en la Nube Privada, destacando la DMTF por su detallada descripción de los actores (DMFT, 2010), convirtiéndose en la base del Ecosistema de la Nube Privada en la que se centró el presente trabajo. La segunda tarea fundamental realizada para la obtención de una Arquitectura de Referencia Funcional para una Nube Privada con soporte para IaaS, fue analizar cómo los principales líderes en la rama despliegan las capas de la Arquitectura de Referencia Funcional de la UIT en sus soluciones de Nube Privada. Las soluciones propietarias examinadas fueron las de Microsoft, Vmware e IBM por sus experiencias desarrolladas en el diseño e implementación de Nubes Privadas (García Perellada, 2014), aunque el principal interés para cumplir los objetivos del proyecto lo constituyeron las soluciones de SLCA. Las soluciones de SLCA estudiadas fueron OpenNebula, Eucalyptus y CloudStack, seleccionadas por su empleo en entidades a nivel internacional, las referencias obtenidas en relación a las capacidades para efectuar la gestión de infraestructuras en la Nube y la bibliografía disponible (García Perellada, 2014).

Contexto Nacional

Actualmente en nuestro país, específicamente en la Universidad de Ciencias Informáticas (UCI), se están desarrollando nuevos proyectos, entre ellos se encuentra el “Sistema de Gestión Académica para el MININT” (Sauce Akademos), una APK con una arquitectura orientada a microservicios. Debido a todo el engorroso proceso que este requería del levantamiento de información,

de requisitos, de la disponibilidad que tenía dicho proyecto para con sus clientes se ha decidido utilizar Docker. Esto debido a los problemas existentes para este proyecto se determinó que la tecnología Docker resolvía tales problemas como son: falta de personal, conectividad en los centros, escuelas, etc. Dicho proyecto cuenta con una aplicación web, una APK y una aplicación Desktop. Una de las ventajas de Docker es que permite usar varios lenguajes, pues para frontend se pueden usar lenguajes como: Angular o React y para el backend: Django o Laravel. Docker facilita el manejo de estos lenguajes al ser posible integrarlos en un solo archivo "Docker-Compose".

2.3.2. Metodología de Desarrollo de *Software*

En el presente trabajo se realizó un estudio para determinar que metodología de desarrollo de *software* es la más eficaz a utilizar para hacer más preciso y eficiente el despliegue de la aplicación.

Las metodologías de desarrollo de *software* son indispensables para crear o actualizar *software* de calidad que cumpla con los requisitos de los usuarios; son una parte fundamental de la Ingeniería de *Software* la cual denomina metodología a un conjunto de métodos coherentes y relacionados por unos principios comunes (Rivas, 2015).

Debido a que el despliegue de la aplicación precisa de entregar continuamente infraestructura como código (*continue delivery*) la cual ayuda a limitar el trabajo en curso, al tiempo que la automatización de la implementación ayuda a erigir restricciones y una integración continua, se ha decidido utilizar una metodología DevOps.

DevOps fue acuñado en 2009 por Patrick Debois, que se ha convertido desde entonces en uno de los gurús dentro de la comunidad. El término se conforma de combinar las palabras 'desarrollo' y 'operaciones', del inglés "Development & Operations", y puede servir como punto de partida para entender qué significa exactamente el término DevOps. Esta nueva cultura no es un proceso, una tecnología concreta o un estándar, sino un conjunto de técnicas, pensamientos, y modelos de trabajo. También se utiliza el término "movimiento DevOps" cuando se habla de temas acerca de la adopción de nuevos ratios e indicadores y tendencias de futuro y "entorno DevOps" para referirse a la estrategia organizativa que sugiere la cultura DevOps (Walls, 2013).

DevOps consiste en algo más que automatizar el canal de despliegue. En palabras de John Allspaw, DevOps consiste en "Operaciones que piensan como Desarrollo y Desarrollo que piensa como Operaciones".

Tabla 2.1: Modos y principios de DevOps

Modo	Principio
Primer Modo	Pensamiento Sistémico
Segundo Modo	Aumentar los ciclos de *feedback*
Tercer Modo	Cultura de la experimentación y el aprendizaje continuo

1. El pensamiento sistémico trata sobre la importancia de hacer hincapié en el rendimiento del sistema completo, en lugar del rendimiento de un compartimento aislado del trabajo o un departamento en concreto (tan grande como una división o tan pequeño como un colaborador individual).

2. El segundo modo se centra en la creación de ciclos de *feedback* de derecha a izquierda. El objetivo de casi cualquier iniciativa de mejora en los procesos es reducir y aumentar los ciclos de *feedback* para poder realizar las correcciones necesarias de forma continua.

3. El tercer modo consiste en la creación de una cultura que fomente dos cosas: la experimentación continua, asumiendo riesgos y aprendiendo de los errores; y la comprensión de que la repetición y la práctica es el requisito previo de la excelencia.

4. La entrega contínua se centra en el primer modo: automatizar el flujo de desarrollo a operaciones. La automatización desempeña una función obvia en ayudar a acelerar el sistema de despliegue. Pero hay más pensamientos sistémicos aparte de la automatización. (Buchanan, [s.f.])

En 2012 Debois elaboró cuatro áreas claves para indicar los aspectos relevantes en DevOps: (Marco, 2016)

- Área 1: Extender la entrega a producción: Los equipos de desarrollo y operaciones colaboran para mejorar el proceso de entrega de un proyecto al entorno de producción.

- Área 2: Extender la respuesta del sistema en operaciones al proyecto: Hacer extensible toda la información relevante en producción al equipo de desarrollo.

- Área 3: Compartir todo el conocimiento del proyecto al equipo de operaciones: El equipo de desarrollo comparte la responsabilidad de todo lo que ocurre en el entorno de producción.

- Área 4: Integrar el conocimiento de operaciones en el equipo de desarrollo: Operaciones debe involucrarse en el proyecto desde el inicio.

Ejemplo de tecnologías que usan esta metodología es precisamente Docker, en el siguiente código se puede observar como funciona utilizando un archivo Jenkins.

```
pipeline {
        agent {
                docker { image 'node:16.13.1-alpine' }
        }
        stages {
                stage('Test') {
                        steps {
                                sh 'node --version'
                        }
                }
        }
}
```

En la codificación anterior se pone de manifiesto que se utiliza una imagen de *node* en su versión *alpine* para hacer la aplicación más ligera, luego el código para por diferentes etapas. La ventaja es que si una etapa padre falla simplemente se puede reestablecer desde ese punto sin tener que cargar o configurar nada previamente, también sucede si una etapa tiene una condición. No obstante, si al momento de cambiar de etapas ocurre un error previo no se puede reiniciar desde ese punto. Este código posee una etapa llamada *Test* en la cual se verifica la versión de *node* utilizando el comando *sh*.

La principal diferencia de la metodología DevOps con las metodologías ágiles es que ha conseguido unir los equipos de desarrollo con los de sistemas.

2.3.3. Lenguajes

Lenguaje de Modelado

El Lenguaje Unificado de Modelado (UML) fue creado para forjar un lenguaje de modelado visual común y semántica y sintácticamente rico para la arquitectura, el diseño y la implementación de sistemas de *software* complejos, tanto en estructura como en comportamiento. UML tiene aplicaciones más allá del desarrollo de *software*, por ejemplo, en el flujo de procesos en la fabricación.

Es comparable a los planos usados en otros campos y consiste en diferentes tipos de diagramas. En general, los diagramas UML describen los límites, la estructura y el comportamiento del sistema y los objetos que contiene.

UML no es un lenguaje de programación, pero existen herramientas que se pueden usar para generar código en diversos lenguajes usando los diagramas UML. UML guarda una relación directa con el análisis y el diseño orientados a objetos (Lucidchart, [s.f.(b)]).

Actualizaciones a UML 2.0:

El UML se perfecciona continuamente. UML 2.0 extiende las especificaciones de UML para cubrir más aspectos de desarrollo, incluido Agile. La meta era reestructurar y perfeccionar UML de forma que la facilidad de uso, la implementación y la adaptación se simplificaran. Estas son algunas de las actualizaciones de los diagramas UML:

1. Mayor integración entre modelos estructurales y de comportamiento.
2. Capacidad de definir jerarquía y desglosar un sistema de *software* en componentes y subcomponentes.
3. UML 2.0 eleva el número de diagramas de 9 a 13 (Lucidchart, [s.f.(a)]).

2.3.4. Marco de trabajo

Django es un *framework* web de alto nivel que permite el desarrollo rápido de sitios web seguros y mantenibles. Desarrollado por programadores experimentados, Django se encarga de gran parte de las complicaciones del desarrollo web, por lo que puedes concentrarte en escribir tu aplicación sin necesidad de "reinventar la rueda". Es gratuito y de código abierto, tiene una comunidad próspera y activa, una gran documentación y muchas opciones de soporte gratuito (MDN, 2021).

Django es un poderoso *framework*, es uno de los más usados en la actualidad. Esto lo justifica las numerosas ventajas que posee, entre ellas se manifiestan: (MDN, 2021)

- Versatilidad: Django puede ser (y ha sido) usado para construir casi cualquier tipo de sitio web: desde sistemas manejadores de contenidos y wikis, hasta redes sociales y sitios de noticias. Puede funcionar con cualquier *framework* en el lado del cliente, y puede devolver contenido en casi cualquier formato (incluyendo HTML, RSS *feeds*, JSON, XML, etc).

- Seguridad: Django ayuda a los desarrolladores evitar varios errores comunes de seguridad al proveer un *framework* que ha sido diseñado para “hacer lo correcto” para proteger el sitio web automáticamente.
- Escalable: Django usa un componente basado en la arquitectura. “shared-nothing” (cada parte de la arquitectura es independiente de las otras, y por lo tanto puede ser reemplazado o cambiado si es necesario). Teniendo en cuenta una clara separación entre las diferentes partes significa que puede escalar para aumentar el tráfico al agregar *hardware* en cualquier nivel: servidores de cache, servidores de bases de datos o servidores de aplicación.
- Mantenible: El código de Django está escrito usando principios y patrones de diseño para fomentar la creación de código mantenible y reutilizable. En particular, utiliza el principio No te repitas “Don’t Repeat Yourself” (DRY) para que no exista una duplicación innecesaria, reduciendo la cantidad de código.
- Portable: Django está escrito en Python, el cual se ejecuta en muchas plataformas. Lo que significa que no está sujeto a ninguna plataforma en particular, y puede ejecutar sus aplicaciones en muchas distribuciones de Linux, Windows y Mac OS X.

2.3.5. Herramientas utilizadas para el desarrollo del sistema

Herramienta del modelado

Muchos pueden pensar que la etapa más importante en el desarrollo de *software* es la implementación. Y aunque es el momento en el cual se construye el producto final, es imprescindible que se realice un minucioso modelado de *software* previamente.

El modelado de *software* es una técnica de ingeniería donde se visualiza el *software* antes de ser desarrollado.

El nivel de detalle a aplicar depende de la experiencia del analista funcional que realice el modelado y la metodología que utilice.

Mediante el modelado se puede especificar la arquitectura de *software*, los componentes y sus relaciones, el modelo de base de datos, las actividades y el flujo que deben seguir, procesos... en fin, una gran variedad de artefactos y diagramas que organizan incluso el código fuente, los nombres de las funcionalidades y los patrones de diseño que se incorporan.

Un modelado de *software* correcto y específico deja listo el terreno para que el desarrollador pueda programar el sistema desde una base sólida, asegurando tener en cuenta la seguridad

del producto, la reutilización de código y especialmente, la organización de cada clase y método (Lago, 2022a).

Aunque existen al menos 5 otras herramientas que son altamente recomendadas por su disponibilidad y características (Lago, 2022b), se usará *Visual Paradigm*.

Visual Paradigm se clasifica también dentro de las herramientas CASE, siglas que corresponden al nombre de Ingeniería de *Software* Asistida por Computadora en inglés y se conoce como una herramienta de modelado, concebida especialmente para el desarrollo de *software*.

Por este motivo contiene un amplio grupo de diagramas, estereotipos y otros componentes que abarcan cada etapa del ciclo de desarrollo de un *software*.

Sin embargo, mediante esta herramienta también es posible lograr la descripción de procesos complejos, incluyendo sus subprocesos. De ahí que se haya convertido en una plataforma, o suite como también se le conoce, del modelado.

A continuación, se muestran algunas de sus características:

- Una vez hecho, no hay que escribirlo: permite generar código fuente, una imagen, incluso los datos en un Excel a partir de los diagramas modelados.
- Ingeniería en cualquier dirección: soporta ingeniería inversa.
- Ágil o robusto: incluye artefactos para metodologías tradicionales, metodologías ágiles y para la Gestión de Servicios de Tecnologías de la Información (ITSM).

Además, permite el modelado en tiempo real, la documentación de proyectos de desarrollo entre otras ventajas. Como aspecto negativo puede señalarse que al ser una plataforma con tantas opciones es compleja de entender en su totalidad.

Es una herramienta que posee distribución de pago y una opción libre. La diferencia radica en que la de pago permite generar código fuente a partir de los diagramas y la gratis no. Además, al exportar los diagramas como imagen bajo la distribución libre, se añade una marca de agua.(Lago, 2022c)

2.3.6. Entorno de Desarrollo Integrado

Un entorno de desarrollo integrado (IDE por sus siglas en inglés) es un *software* para crear aplicaciones que combina herramientas de desarrollo comunes en una única interfaz gráfica de usuario (GUI). Un IDE normalmente consta de:

- Editor de código fuente: un editor de texto que puede ayudar a escribir código de *software* con características como el resaltado de sintaxis con señales visuales, que proporciona auto-completado específico del idioma y la verificación de errores a medida que se escribe el código.
- Automatización de compilación local: utilidades que automatizan tareas simples y repetibles como parte de la creación de una compilación local del *software* para que la use el desarrollador, como compilar el código fuente de la computadora en código binario, empaquetar código binario y ejecutar pruebas automatizadas.
- Depurador: un programa para probar otros programas que pueden mostrar gráficamente la ubicación de un error en el código original (Hat, 2019).

Visual Studio Code:

Se decidió utilizar Visual Studio Code el cual es un editor de código fuente ligero pero potente que se ejecuta en su escritorio desarrollada por Microsoft y está disponible para Windows, macOS y Linux. Viene con soporte incorporado para JavaScript, TypeScript y Node.js y tiene un rico ecosistema de extensiones para otros lenguajes y tiempos de ejecución (como .NET y Unity) (Microsoft, [s.f.(a)]).

Visual Studio Code combina la simplicidad de un editor de código fuente con potentes herramientas para desarrolladores, como la finalización y depuración de código de IntelliSense.

En primer lugar, es un editor que se quita de en medio. El ciclo de edición-construcción-depuración deliciosamente fluido significa menos tiempo jugando con su entorno y más tiempo ejecutando sus ideas.

En esencia, Visual Studio Code presenta un editor de código fuente ultrarrápido, perfecto para el uso diario. Con soporte para cientos de idiomas, VS Code lo ayuda a ser productivo al instante con resaltado de sintaxis, coincidencia de corchetes, sangría automática, selección de

cuadros, fragmentos y más. Los atajos de teclado intuitivos, la fácil personalización y las asignaciones de atajos de teclado aportadas por la comunidad le permiten navegar por su código con facilidad.

Para la codificación seria, a menudo se beneficiará de herramientas con más comprensión del código que solo bloques de texto. Visual Studio Code incluye soporte integrado para la finalización de código de IntelliSense, comprensión y navegación de código semántico enriquecido y refactorización de código.

Y cuando la codificación se vuelve difícil, los difíciles se vuelven depuradores. La depuración suele ser la característica que más extrañan los desarrolladores en una experiencia de codificación más eficiente, así que lo hicimos realidad. Visual Studio Code incluye un depurador interactivo, por lo que puede recorrer el código fuente, inspeccionar variables, ver pilas de llamadas y ejecutar comandos en la consola.

VS Code también se integra con herramientas de compilación y secuencias de comandos para realizar tareas comunes que agilizan los flujos de trabajo diarios. VS Code es compatible con Git, por lo que puede trabajar con control de código fuente sin salir del editor, incluida la visualización de diferencias de cambios pendientes (Microsoft, [s.f.(b)]).

2.3.7. Herramientas de gestión de base de datos

Un sistema gestor de base de datos o SGBD (del inglés: data management system o DBMS) es un *software* que permite administrar una base de datos. Esto significa que mediante este programa se puede utilizar, configurar y extraer información almacenada. Los usuarios pueden acceder a la información usando herramientas específicas de consulta y de generación de informes, o bien mediante aplicaciones al efecto.

Estos sistemas también proporcionan métodos para mantener la integridad de los datos, para administrar el acceso de usuarios a los datos y para recuperar la información si el sistema se corrompe. Permiten presentar la información de la base de datos en variados formatos. La mayoría incluyen un generador de informes. También pueden incluir un módulo gráfico que permita presentar la información con gráficos y tablas.

Generalmente se accede a los datos mediante lenguajes de consulta, lenguajes de alto nivel que simplifican la tarea de construir las aplicaciones. También simplifican las consultas y la presentación de la información. Un SGBD permite controlar el acceso a los datos, asegurar su integridad, gestionar el acceso concurrente a ellos, recuperar los datos tras un fallo del sistema y hacer

copias de seguridad. Las bases de datos y los sistemas para su gestión son esenciales para cualquier área de negocio, y deben ser gestionados con esmero (IONOS, 2020).

PostgreSQL

El gestor de base de datos seleccionado por grupo de desarrollo y el líder de proyecto del ecosistema decidimOS es PostgreSQL el cual es un poderoso sistema de base de datos relacional de objetos de código abierto con más de 30 años de desarrollo activo que le ha valido una sólida reputación por su confiabilidad, robustez de funciones y rendimiento (PGDG, 2022).

PostgreSQL se ha ganado una sólida reputación por su arquitectura comprobada, confiabilidad, integridad de datos, conjunto sólido de funciones, extensibilidad y la dedicación de la comunidad de código abierto detrás del *software* para ofrecer soluciones innovadoras y de alto rendimiento. PostgreSQL se ejecuta en todos los principales sistemas operativos, cumple con ACID desde 2001 y tiene potentes complementos, como el popular extensor de base de datos geo-espacial PostGIS. No sorprende que PostgreSQL se haya convertido en la base de datos relacional de código abierto elegida por muchas personas y organizaciones.

A continuación, se muestra una lista exhaustiva de varias funciones que se encuentran en PostgreSQL, y se agregan más en cada versión principal:(The PostgreSQL Global Development, 2022)

1. Tipos de datos
    - Primitivas: entero, numérico, cadena, booleano.
    - Estructurado: Fecha/Hora, Matriz, Rango/Multirango, UUID.
    - Documento: JSON/JSONB, XML, clave-valor(Hstore).
    - Geometría: punto, línea, círculo, polígono.
    - Personalizaciones: Compuesto, Tipos personalizados.
2. Integridad de los datos:
    - Único, no nulo.
    - Claves primarias.
    - Claves foráneas.

- Restricciones de exclusión.
- Bloqueos explícitos, bloqueos de aviso .

3. Concurrencia, Rendimiento:
    - Indexación: árbol B, varias columnas, expresiones, parcial.
    - Indexación avanzada: GiST, SP-Gist, KNN Gist, GIN, BRIN, índices de cobertura, filtros Bloom.
    - Planificador/optimizador de consultas sofisticado, escaneos de solo índice, estadísticas de varias columnas.
    - Transacciones, transacciones anidadas (a través de puntos de guardado).
    - Control de concurrencia de múltiples versiones (MVCC).
    - Paralelización de consultas de lectura y creación de índices de árbol B.
    - Partición de tablas.
    - Todos los niveles de aislamiento de transacciones definidos en el estándar SQL, incluido Serializable.
    - Compilación de expresiones justo a tiempo (JIT).
4. Confiabilidad, Recuperación de Desastres:
    - Registro de escritura anticipada (WAL).
    - Replicación: asíncrona, síncrona, lógica.
    - Recuperación de un punto en el tiempo (PITR), esperas activas.
    - Espacios de tablas.
5. Seguridad:
    - Autenticación: GSSAPI, SSPI, LDAP, SCRAM-SHA-256, Certificado y más.
    - Sistema robusto de control de acceso.
    - Seguridad a nivel de columna y fila.
    - Autenticación multifactor con certificados y un método adicional.
6. Extensibilidad:

- Funciones y procedimientos almacenados.
- Lenguajes de procedimiento: PL/PGSQL, Perl, Python (y muchos más).
- Expresiones de ruta SQL/JSON.
- Envoltorios de datos externos: conéctese a otras bases de datos o flujos con una interfaz SQL estándar.
- Interfaz de almacenamiento personalizable para tablas.
- Muchas extensiones que brindan funcionalidad adicional, incluido PostGIS.

7. Internacionalización, Búsqueda de Texto:
   - Compatibilidad con conjuntos de caracteres internacionales, a través de colaciones de la UCI.
   - Intercalaciones que no distinguen entre mayúsculas y minúsculas ni acentos.
   - Búsqueda de texto completo.

2.4. Conclusiones parciales

En este proyecto se parte de la idea de hacer un despliegue de *software* a un sistema web con unos requisitos que incluían, entre otros, despliegue automático y continuo de las diferentes partes de la plataforma.

Se ha decidido utilizar una arquitectura de microservicios donde cada servicio tiene una determinada función. Este primer capítulo ha arrojado excelentes resultados los cuales permitieron:

- Analizar los conceptos fundamentales de la tecnología Docker asociados al proceso de despliegue de *software*s.
- El estudio comparativo de soluciones similares para el despliegue de *softwares*.

- La identificación de las características propuestas para la solución, el análisis de las tecnologías que distinguen a soluciones similares para el despliegue de *software*s, contribuyó a la selección de las herramientas y tecnologías más adecuadas para desarrollar la propuesta de solución, destacando como metodología de desarrollo DevOps, como marco de trabajo Django, para construcción de *frontend* Angular y para administrar base de datos postgreSQL y como herramienta de modelado *Visual Paradigm* usando como lenguaje UML.

3 DISEÑO DE LA SOLUCIÓN PROPUESTA AL PROBLEMA CIENTÍFICO

En este capítulo se aborda la propuesta de solución a la problemática planteada. Se describen los requisitos funcionales y no funcionales para dar cumplimiento a los objetivos planteados. Se realizan las historias de usuario, diagrama de despliegue y diagrama de actividades así como el modelado de las vistas y modelos arquitectónicos exigidos por la Metodología DevOps.

El proceso de despliegue se define como el conjunto de instrucciones o actividades que deben realizarse de manera ordenada y correcta para una puesta en marcha eficiente del software que se desarrolla, adapta o mantiene en el contexto específico en que se despliega que requiere una infraestructura tecnológica.

El objetivo de esta actividad es planificar, programar y controlar las *Releases* de un producto. *Release & deploy* es el origen de muchas de las prácticas que se adoptan en DevOps. *Continuous Release* y *continuous deployment* complementan el *continuous integration* y un paso más allá. La práctica que permite la liberación y despliegue de *Releases* también permite la creación de una línea de entrega.

Esa línea de entrega facilita el despliegue continuo de software manteniendo un control de calidad y una entrega a producción de forma eficiente. El objetivo de la liberación continua de *Releases*, así como del despliegue continuo, es liberar y/o mejorar la funcionalidad de negocio para usuarios y clientes tan pronto como sea posible.

La mayoría de herramientas y procesos que conforman el núcleo de la tecnología DevOps se han creado para facilitar la integración continua, liberación contínua de *Releases* y el despliegue continuo.

3.1. Plan de Despliegue

Los planes de despliegue contienen el conjunto de actividades a realizar sobre las unidades de software en un entorno concreto. El gestor de cambios de entorno debe ser responsable de que los planes generados sean válidos, evitando que se intenten desplegar unidades sobre nodos sin los recursos requeridos. Para ello, debe analizar las características de cada unidad a desplegar, considerando sus dependencias y restricciones, y el estado del entorno sobre el que se desea llevar a cabo del despliegue. Además, es posible que desde la creación de un plan hasta su ejecución, el entorno varíe, siendo necesario comprobar que las operaciones contenidas en las actividades se puedan cursar, debido a que todos los requisitos y dependencias previos se siguen cumpliendo y que además no entran en conflicto con las unidades desplegadas actualmente en el entorno (Diaz Casillas y col., 2010).

Los datos relevantes que se deben incluir en un plan de despliegue son:

- Seguimiento de la cuestión
- Funciones y responsabilidades (antes, durante y después de la implantación)
- Soporte del sistema
- Procesos de escalada

La metodología *Development Operations* (DevOps) permite acortar el ciclo de vida del desarrollo y la entrega continua de software con alta calidad. La contenerización es la tecnología que nos facilita el seguimiento en la práctica **DevOps**. Es aquel proceso de “empaquetar una aplicación junto con sus bibliotecas, marcos y archivos de configuración necesarios para que pueda ejecutarse en varios entornos informáticos de manera eficiente”. La contenerización es la encapsulación de una aplicación y de todo su entorno en un “contenedor”.

Docker es uno de las herramientas más populares en esta industria. Es una plataforma de código abierto que simplifica la construcción, prueba, protección e implementación de aplicaciones dentro de contenedores y promueve también la colaboración con sistemas operativos en la nube.

Antes, el desarrollo, pruebas y despliegue se llevaban a cabo en fases seguidas, donde cada una se realizaba cuando finalizaba la anterior. Ahora, gracias a la administración de imágenes de DevOps y Docker, se han facilitado las operaciones de TI. Compartir software y colaborar entre sí han mejorado la productividad, además de alentar un ambiente más colaborativo, eliminando los conflictos entre entornos de trabajo que afectaban a la aplicación.

3.2. Requisitos:

3.2.1. Definición de Requisito:

Sommerville (2003, p. 82) identifica dos puntos de vistas diferentes sobre el término requisito. Este puede ser “una visión abstracta, de alto nivel, de una función que el sistema debe suministrar o de una restricción del sistema” o, por otro lado, “una definición detallada , matemáticamente formal, de una función del sistema”.

El autor también pone de relieve los diferentes niveles de detalle de los requisitos: (Ramos Cardozzo, 2016)

- Requisitos del usuario: Sentencias en lenguaje natural acerca de las funciones que el sistema debe proporcionar.
- Requisitos del sistema: Detalles de las funciones y restricciones del sistema, pudiendo ser utilizados en el contrato de desarrollo de software.
- Especificación de proyecto de software: Descripción abstracta del proyecto de software, añadiendo más detalles acerca de la solución a los requisitos del sistema.

Los requisitos son la base para el desarrollo de una solución al problema. Los requisitos funcionales son aquellos directamente relacionados con las funciones o las reacciones que el sistema debe proporcionar. Los requisitos no funcionales son restricciones impuestas al funcionamiento del sistema, tales como las limitaciones de tiempo, presupuesto, proceso de desarrollo, las políticas de la organización, normas que deben adoptarse, entre otros. (Sommerville, 2003, p. 83)

3.2.2. Requisitos Funcionales:

Tabla 3.1: Tabla de requisitos funcionales

No.	Nombre	Descripción	Prioridad	Complejidad
RF1	Gestionar Base de Datos	Este requisito funcional es de suma importancia ya que es un CRUD Total y su correcto funcionamiento tiene estrecha relación con el resultado final del proyecto	Alta	Media
RF2	Configurar y crear archivo Python	La resolución correcta de dicho requisito es de vital importancia para la sincronización correcta entre los servicios de Python, Angular y PostgreSQL	Alta	Media
RF3	Configurar y crear archivo Angular	La resolución correcta de dicho requisito es de vital importancia para la sincronización correcta entre los servicios de Python, Angular y PostgreSQL	Alta	Media
RF4	Configurar Base de Datos	La configuración correcta de la base de datos en el archivo `settings` de la aplicación permite el uso de dicha base de datos	Alta	Baja
RF5	Sincronizar aplicación decidimOS	La sincronización del módulo de despliegue junto con la aplicación es de máxima prioridad	Alta	Baja

3.2.3. Requisitos No Funcionales:

- RNF 1. Hardware:
  - RNF 1.1. El sistema debe correr sobre cualquier distribución del sistema operativo Linux, MacOS y/o superior a Windows 8 64-bit.
  - RNF 1.2. La PC Cliente debe tener las siguientes propiedades:

    - Un mínimo de 30 GB de espacio libre en disco duro.
    - Un mínimo de 2 GB de memoria RAM.
    - Un procesador i3-5015U o superior.
    - Velocidad de núcleo superior a 2 GHz.

- RNF 2. Confiabilidad:
    - RNF 2.1. Asignar los permisos de acceso, lectura y escritura al personal adecuado para la gestión del despliegue.
    - RNF 2.2. El sistema debe permitir variación en los campos de usuario y contraseña de la base de datos.
    - RNF 2.3. En caso de fallas en el despliegue no es necesario iniciar desde el principio el código.
    - RNF 2.4. Permite la sincronización de varios servicios simultáneamente.
    - RNF 2.5. Poseer un alto grado de escalabilidad para agregar nuevas funciones, volúmenes y servicios.
- RNF 3. Seguridad:
    - RNF 3.1. La información de usuario y contraseña de la base de datos estarán protegidas en un archivo específico para ello.
    - RNF 3.2. El acceso a la aplicación solo será permitida en el puerto que decida el administrador o líder del proyecto.
- RNF 4. Usabilidad:
    - RNF 4.1. Lograr una correcta estructura en la sincronización de los servicios para el correcto despliegue de la aplicación.
    - RNF 4.2. Para el uso correcto de dicho módulo de despliegue es necesario conocimientos a nivel medio de Arquitecturas de Despliegue y Docker.
    - RNF 4.3. Tener instaladas las extensiones necesarias en la PC Cliente.
- RNF 5. Interfaz:
    - RNF 5.1. El puerto IP asignado debe ser posible acceder a el desde cualquier navegador.

- RNF 6. Rendimiento:
  - RNF 6.1. Garantizar mejora en velocidad de respuesta y despliegue de la aplicación.
  - RNF 6.2. Permitir la conexión simultánea de varios puertos ip a dicha aplicación decidimOS.
  - RNF 6.3. El sistema debe ser capaz de mantener tiempos de respuestas en un marco razonable a pesar del aumento de información.
- RNF 7. Restricciones del diseño e implementación:
  - RNF 7.1. Se utilizaría para el modelado la herramienta CASE Visual Paradigm for UML 8.0
  - RNF 7.2. Se utilizará para la configuración de Base de Datos el Sistema de Gestión de Base de Datos PostgreSQL.
  - RNF 7.3. Se utiliza el orquestador Docker Compose en su versión 3.

3.2.4. Historias de Usuario:

Siguiendo con la definición de requerimientos de software de la IEEE (*Standards Coordinating Committee of the Computer Society of the IEEE*, 1990) citada en la sección Introducción a los Requerimientos de Software, podemos concluir que las historias de usuario son requerimientos ya que expresan el problema que el sistema o producto software debe resolver.

Numero: RF1	Requisito: Configurar Base de Datos
Programador: Ariel Rodríguez Lau	Iteración asignada: 1
Prioridad: Alta	Tiempo estimado: 54 h
Riesgo en Desarrollo: Uso de caracteres no permitidos o mal uso de información en su configuración	Tiempo Real 1 h
Descripción: La configuración correcta de la base de datos en el archivo `settings` de la aplicación permite el uso de dicha base de datos	Observaciones: Debe presentar las configuraciones mínimas necesarias como son: usuario, contraseña y puerto.

Prototipo elemental de interfaz gráfica de usuario 3.1:

Figura 3.1: Imagen de configuración Base de Datos

Numero: RF2	Configurar y crear archivo python
Programador: Ariel Rodríguez Lau	Iteración asignada: 1
Prioridad: Alta	Tiempo estimado: 48hrs
Riesgo en Desarrollo: El no conocimiento adecuado de la tecnología o lenguaje de programación	Tiempo Real: 1h
Descripción: La resolución correcta de dicho requisito es de vital importancia para la sincronización correcta entre los servicios de Python, Angular y PostgreSQL	Observaciones: Esta configuración contiene el nombre de la carpeta del proyecto así como sus hilos de trabajo.

Prototipo elemental de interfaz gráfica de usuario 3.2:

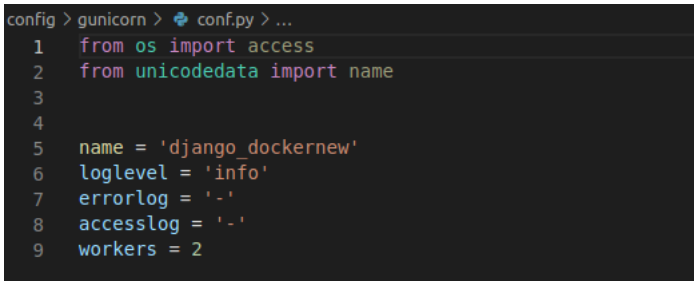


Figura 3.2: Imagen de configuración de python

Numero: RF3	Configurar y crear archivo angular
Programador: Ariel Rodríguez Lau	Iteración asignada: 1
Prioridad: Alta	Tiempo estimado: 50 h
Riesgo en Desarrollo: El no conocimiento adecuado de la tecnología o lenguaje de programación	Tiempo Real: 1 h
Descripción: La resolución correcta de dicho requisito es de vital importancia para la sincronización correcta entre los servicios de Python, Angular y PostgreSQL	Observaciones: Esta configuración contiene la declaración del puerto de escucha fuera del contenedor, en la web, contiene la declaración del servicio Django así como los proxy.

Figura 3.3: Imagen de configuración de angular

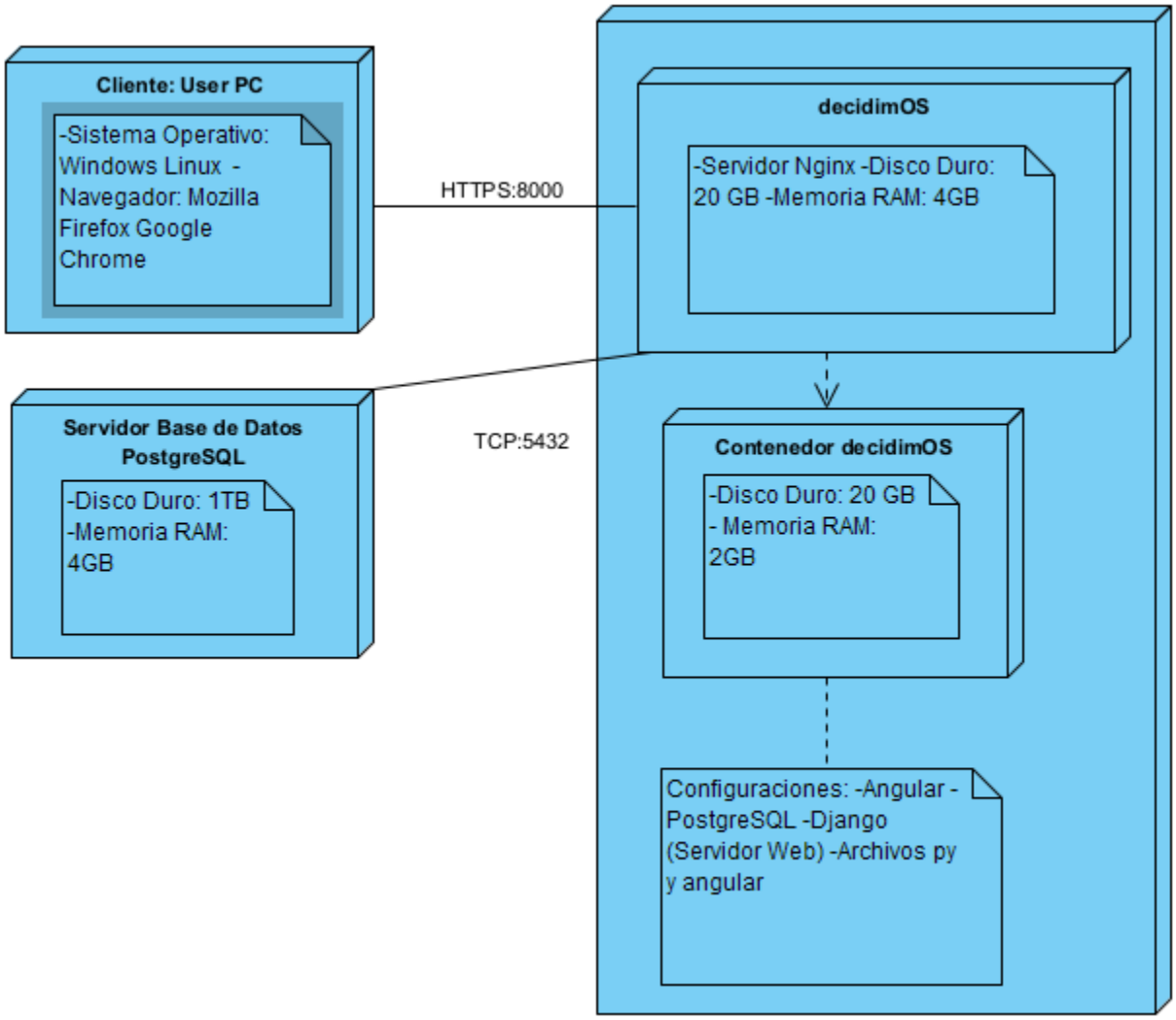
3.2.5. Vista de Despliegue:

A continuación, se representa la Vista de Despliegue, la cual, es un tipo de diagrama del Lenguaje Unificado de Modelado que se utiliza para modelar la disposición física de los artefactos software en nodos. Muestra la arquitectura del sistema como el despliegue de los artefactos de software a los objetivos de despliegue.

En la imagen 3.4 se pueden apreciar un conjunto de 4 nodos los cuales poseen las características esenciales para realizar el despliegue. En el primer nodo se muestra la $PC\_Cliente$ la

cual va a ser el propio administrador de dicha aplicación. Esta $PC\_Cliente$ debe poseer las características que se muestran en la nota que posee para poder realizar el despliegue. Este primer nodo está asociado al nodo decidimOS el cual se realiza sobre un servidor Apache, debe tener un espacio en disco para dicha aplicación de 20 GB y una memoria RAM de 4 GB y va a estar conectado a su base de datos la cual controla el administrador la cual se necesita 1 TB de almacenamiento debido a que es una aplicación a nivel nacional y memoria RAM de 4 GB para su consumo. El administrador además será quien controle el contenedor de la aplicación para hacer los ajustes correspondientes en cada momento, dentro de este contenedor se encuentran todas las configuraciones para realizar dicho despliegue, para esto se necesita un mínimo de 2 GB de memoria RAM y almacenamiento de 20 GB para controlar todos los contenedores.

Figura 3.4: Imagen de Diagrama de Despliegue



3.2.6. Diagrama de Flujo:

El diagrama de flujo o flujograma o diagrama de actividades es la representación gráfica de un algoritmo o proceso. En Lenguaje Unificado de Modelado (UML) es un diagrama de actividades que representa los flujos de trabajo paso a paso. A continuación se muestra una representación de dicho diagrama del sistema.

En la imagen 3.5 se puede apreciar el proceso de realizar el despliegue, la primera acción a realizar es la preparación de despliegue, luego se pregunta si están creados los servicios, si no están creados se procede a realizar dicha acción, se sincronizan y se repite este proceso hasta que ya estén creados, luego se configuran los archivos `*.py` y angular, así como la base de datos en `settings.py` y se agrega la aplicación, luego de realizado este proceso se realiza el despliegue.

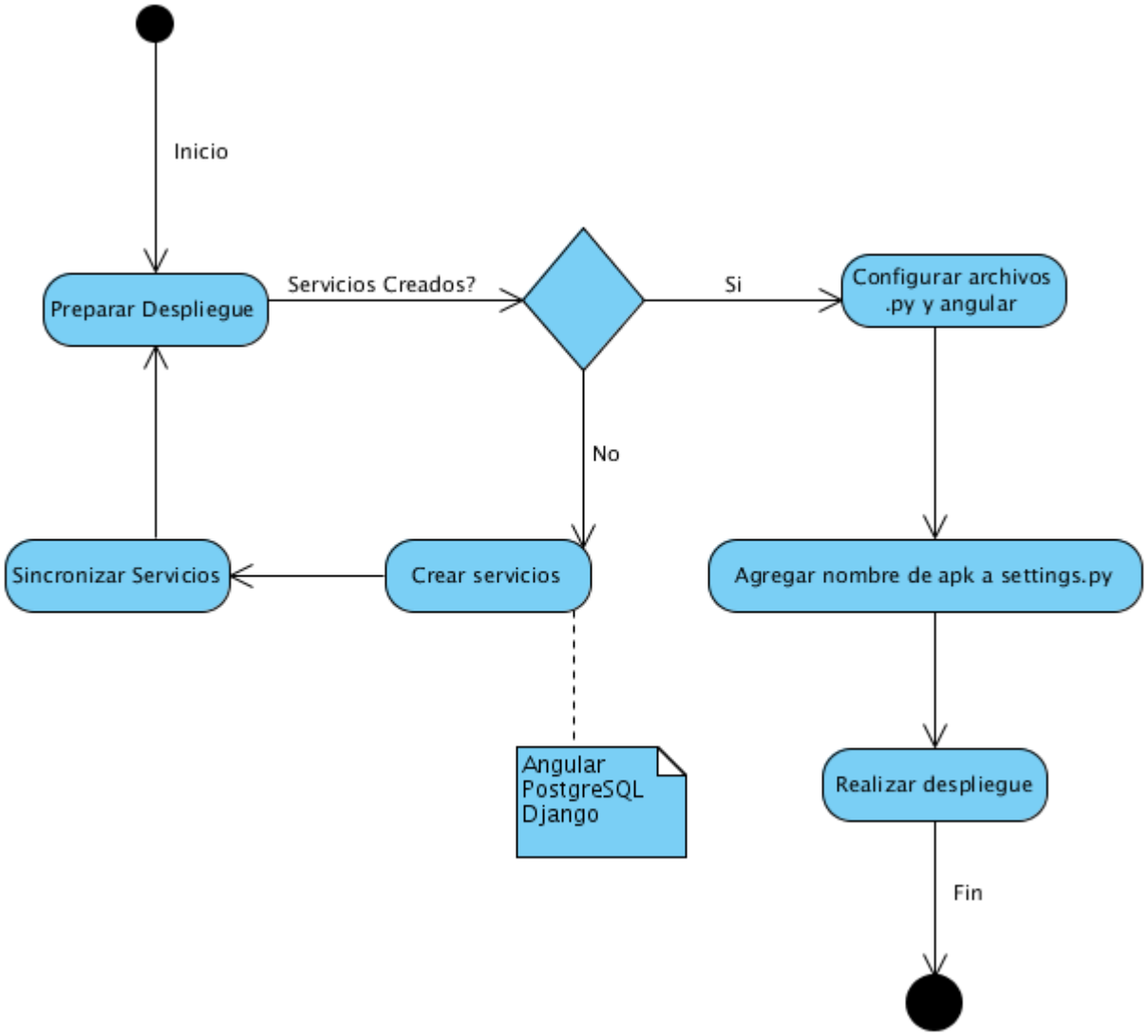


Figura 3.5: Imagen de Diagrama de Actividades o flujograma del sistema.

Figura 3.6: Se aprecia la codificación del dockerfile de Django con Nginx y Gunicorn

En la imagen 3.6 se puede observar detalladamente la codificación del dockerfile para el correcto despliegue de la aplicación donde se pueden apreciar aspectos fundamentales del código como son:

- FROM: Este comando se usa para decirle a Docker que base se usará para la imagen.
- ENV: Se utiliza para crear variables de entorno.
- RUN: El comando RUN es donde se interactúa con la imagen para instalar software y ejecutar guiones, comandos y otras tareas.
- WORKDIR: Se utiliza para establecer un directorio de trabajo.
- COPY: Se utiliza para copiar archivos de un lugar a otro.
- CMD: Se utiliza para ejecutar un comando por defecto cuando se crea el contenedor. (McKendrick, 2020)

En la presente imagen 3.6 se puede apreciar la codificación de un archivo Dockerfile para el servicio de Django el cual a groso modo lo que hace es:

- FROM python:3.7 este comando utilizará como base para su imagen, python en su versión 3.7
- ENV PYTHONUNBUFFERED 1 permite que los mensajes de registro se descarguen inmediatamente en la transmisión en lugar de almacenarse en el búfer.

- RUN mkdir /code crea un directorio o carpeta code en el lugar donde se está trabajando.
- WORKDIR /code se establece como directorio de trabajo la carpeta code.
- COPY decidimos /code/ este código permite copiar todo el contenido de la carpeta decidimos hacia la nueva carpeta de trabajo code.
- ENV PIP_CONFIG_FILE pip.conf esta línea de código permite instalar la configuración del paquete pip de python.
- RUN pip install -r requirements.txt este comando utiliza el sistema de gestión de paquetes pip que usa por defecto Python para instalar recursivamente las dependencias que posee el archivo requirements.txt
- CMD [“gunicorn”, “-c”, “config/gunicorn/conf.py”, “–bind”, “:8000”, “–chdir”, “decidimos”, “decidimos.wsgi:application”] se utiliza para iniciar gunicorn, especificar las configuraciones que son las que se encuentran en `conf.py`, luego se vincula o enlaza al puerto 8000 la aplicación y luego se especifica la dirección del archivo wsgi de la aplicación.

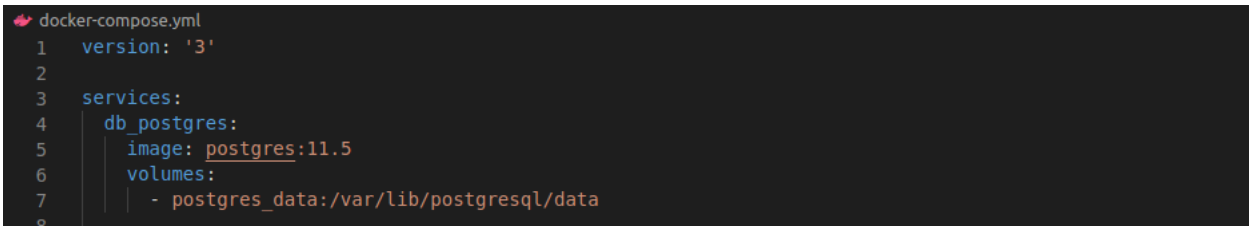


Figura 3.7: Se aprecia la codificación del servicio PostgreSQL

En la imagen 3.7 se presenta la codificación del servicio de postgreSQL donde se puede observar que se utilizará una imagen base de Postgres en su versión 11.5 la cual tiene instalada todas las dependencias necesarias para la base de datos. Además, se define el volumen necesario para la persistencia de la información el cual es el volumen por defecto que se usa internacionalmente.

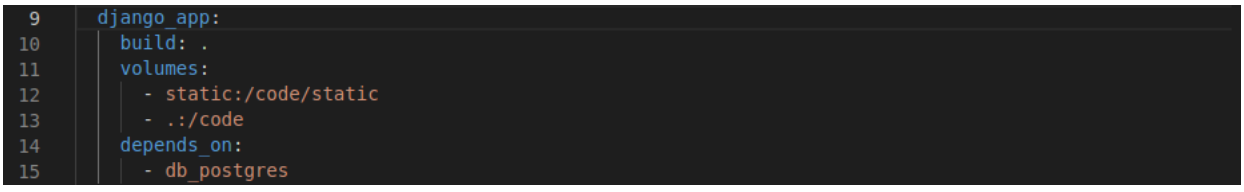


Figura 3.8: Se aprecia la codificación del servicio web

En la imagen 3.8 se muestra el servicio de Django (servidor) se aprecia la utilización del comando “build” , el cual se usa para ejecutar el dockerfile perteneciente a dicho servicio en la carpeta root de la aplicación, luego se usan dos volúmenes y el comando “depends-on:” , que se usa para decirle a dicho servicio que se construya obligatoriamente solo cuando el servicio de base de datos (Postgres) se haya ejecutado.

```
17    nginx:
18      image: nginx:1.13
19      ports:
20        - 8000:80
21      volumes:
22        - ./config/nginx/conf.d:/etc/nginx/conf.d
23        - static:/code/static
24      depends_on:
25        - django_app
```

Figura 3.9: Se aprecia el código del servicio Angular

En la imagen 3.9 se puede observar que se utilizará el servicio de angular que posee una imagen base de nginx en su versión 1.13 la cual tiene instalada todas las dependencias necesarias para la visualización de la aplicación, lo que observa el cliente o usuario. Luego se define el puerto de escucha el cual es el 8000 y el puerto del contenedor que es 80, usan dos volúmenes en el cual el primero mantiene la codificación de nginx en el archivo conf-d y el comando “depends-on:” que se usa para decirle a dicho servicio que se construya obligatoriamente solo cuando el servicio de backend o servidor (django-app) se haya ejecutado.

```
27  volumes:
28    .:
29    postgres_data:
30    static:
```

Figura 3.10: Se puede apreciar la declaración de los volúmenes usados

En esta imagen 3.10 se observa la declaración de tres volúmenes ya que sin esta codificación no pueden funcionar.

3.3. Patrones GRASP:

Los patrones GRASP son guías o principios que sirven para asignar responsabilidades a las clases. (Kord, 2011)

Tabla 3.2: Tabla de Patrones GRASP

Nombre del patrón	Problema	Solución
Experto	¿Cuál es un principio general para asignar responsabilidades a los objetos?	Asignar una responsabilidad al experto en información: la clase que tiene la información necesaria para la realización de la asignación.
Creador	¿Quién debería ser el responsable de la creación de una nueva instancia de alguna clase?	Asignar a la clase B la responsabilidad de crear una instancia de clase A si se cumple uno o más de los casos siguientes: ▪ B agrega objetos de A. ▪ B contiene objetos de A. ▪ B registra instancias de objetos de A. ▪ B utiliza más estrechamente objetos de A. ▪ B tiene datos de inicialización que se pasarán a un objeto de A cuando sea creado (por tanto, B es un experto con respecto a la creación de A). ▪ B es un creador de los objetos A.
Bajo Acoplamiento	¿Cómo soportar bajas dependencias, bajo impacto del cambio e incremento de la reutilización?	Asignar una responsabilidad de manera que el acoplamiento permanezca bajo.
Alta cohesión	¿Cómo mantener la complejidad manejable?	Asignar una responsabilidad de manera que la cohesión permanezca alta.
Controlador	¿Quién debería ser el responsable de gestionar un evento de entrada al sistema?	Asignar una responsabilidad de recibir o manejar un mensaje de evento del sistema a una clase que representa una de las opciones siguientes: Representa el sistema global, dispositivo o subsistema. Representa un caso de uso en el que tiene lugar el evento del sistema a menudo denominado <nombre del caso de uso> Manejador, <nombre del caso de uso> coordinador, <nombre del caso de uso> Sesión. ▪ Utilice la misma clase controlador para todos los eventos del sistema en el mismo escenario de caso de uso. ▪ Informalmente, una sesión es una instancia de una conversación con un actor. Las sesiones pueden tener cualquier duración, pero se organizan a menudo en función de casos de uso.

3.4. Modelo de Vistas Arquitectónicas

3.4.1. Arquitectura de Infraestructura:

La Arquitectura de Infraestructura de Tecnologías de la Información, o Arquitectura de Despliegue, es la capa final de la Arquitectura Empresarial donde todas las definiciones y acuerdos definidos en las otras capas se deben concretar en plataformas de *hardware* y software específicas (Orellana, 2017).

Es además, el diseño de la infraestructura incluyendo servidores, almacenamiento, *middleware*, software no de aplicación, redes y facilidades físicas que soportan las aplicaciones y los procesos de negocio requeridos por la empresa.

En la siguiente imagen 3.11 se puede apreciar la infraestructura de un negocio que tiene como elementos principales los roles de los usuarios, las bases de datos, los servicios de aplicación, los servicios de negocio y el sistema manteniendo así un flujo.

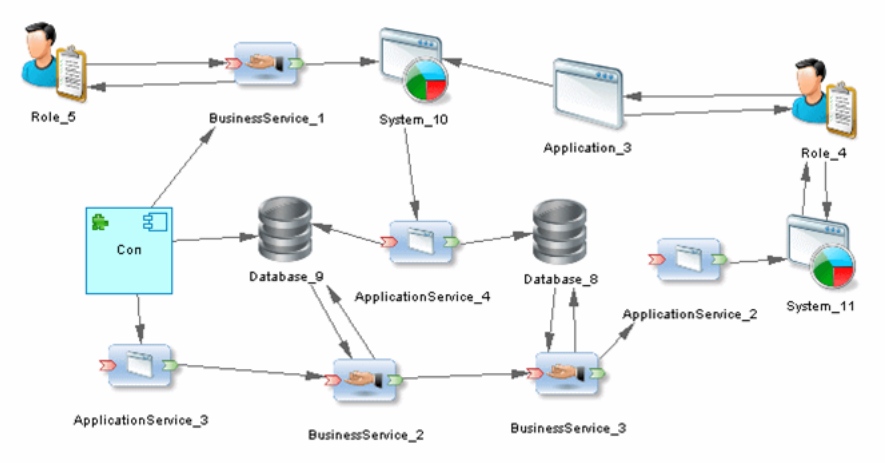


Figura 3.11: Imagen que representa una arquitectura de infraestructura

3.5. Conclusiones Parciales:

La arquitectura de despliegue juega un papel fundamental en la eficacia de una aplicación o software, entre las principales ventajas de usar una arquitectura es que mejora el rendimiento de la aplicación a desplegar y aumenta la velocidad de su despliegue en la web. En este capítulo

se ha argumentado como hacer un correcto plan de despliegue, la importancia del uso de los requisitos funcionales y más relevante aún:

- La configuración correcta de los servicios angular, postgreSQL y Django.
- La configuración correcta de los requisitos necesarios para realizar un despliegue con dichos servicios utilizando diferentes lenguajes de programación.
- Se ha realizado una mejora en la seguridad de las variables (configuraciones) sobre la base de datos del sistema.
- Se realizó una mejora en el sistema, de manera tal que, en cualquier sistema operativo se pueda usar decidimOS ya que se instalan las dependencias necesarias para ejecutarlo.
- Se manifiesta el uso de la metodología DevOps con dicha herramienta para potenciar la arquitectura de despliegue.

4 VALIDACIÓN DE LA SOLUCIÓN PROPUESTA

El objetivo del presente capítulo es realizar la validación de la arquitectura propuesta, a partir del análisis de los principales métodos y técnicas que permiten evaluar arquitecturas de software. Los mismos se enfocan en el cumplimiento de determinados atributos de calidad, que son evaluados cualitativamente en distintas etapas del proceso de desarrollo de software. A partir del análisis se realizará la selección del método más adecuado para realizar la evaluación.

4.1. Necesidad de evaluar una arquitectura

En el contexto actual, se torna imperativo hablar de calidad en el ámbito de desarrollo de software, ya que la misma garantiza poder competir en el mercado internacional con mayores posibilidades de éxito. Diversas definiciones del término han sido ofrecidas, una de ellas plantea que es un “proceso eficaz de software que se aplica de manera que crea un producto útil que proporciona valor medible a quienes lo producen y a quienes lo utilizan” (R. S. Pressman, 2010), la ISO/IEC (Puerto, 2009) la define como “la totalidad de rasgos y atributos de un producto de software que le apoyan en su capacidad de satisfacer sus necesidades explícitas o implícitas”. Sin embargo, los autores del presente trabajo asumen la definición ofrecida por la IEEE, la cual afirma que “la calidad de un software es el grado en el cual el software posee una combinación deseada de factores”. Sin embargo, el autor del presente trabajo asumen la definición ofrecida por la IEEE, la cual afirma que “la calidad de un software es el grado en el cual el software posee una combinación deseada de factores”.

La detección y corrección de errores en etapas tempranas del desarrollo de software es una tarea de suma importancia que puede evitar repercusiones negativas en cuanto a costos y recursos. Realizar una evaluación de la arquitectura de software permite analizar e identificar

riesgos potenciales en su estructura y sus propiedades, que puedan afectar al sistema de software resultante, verificar que los requerimientos no funcionales estén presentes en la arquitectura, así como determinar en qué grado se satisfacen los atributos de calidad (CuriositySec, 2014).

Generalmente, la evaluación de la arquitectura se realiza después de su especificación y antes de su implementación, en un proceso iterativo y/o incremental que puede realizarse al concluir cada ciclo. Sin embargo, no existe una regla que establezca cuál es el mejor momento, dando flexibilidad para efectuar la evaluación en cualquier etapa del ciclo de vida de una arquitectura. En particular, existen dos etapas: temprana y tardía (Dávila y col., 2020) (Clements y col., 2004).

- Evaluación temprana: No es necesario que la arquitectura esté completamente especificada para efectuar la evaluación, y esto abarca desde las fases tempranas de diseño y a lo largo del desarrollo. En esta fase se puede usar un prototipo del software a evaluar.
- Evaluación tardía: Cuando ésta se encuentra establecida y la implementación de la solución, ya se ha completado.

Las evaluaciones pueden ser planeadas o no planeadas. La primera es aquella que ha sido planificada por el ciclo de vida de desarrollo, por lo que es parte de las actividades del proyecto. Por el contrario, las no planeadas se presentan cuando la arquitectura contiene varios defectos que han sido detectados en las etapas tardías del desarrollo. Realizar una evaluación no planeada, representa retrasos en los tiempos de entrega, así como un incremento en los costos de un proyecto, por lo que es recomendable que en los proyectos se contemple realizar una o varias evaluaciones a la arquitectura.

La evaluación de la arquitectura permite determinar si la misma es adecuada para el sistema para el cual fue diseñada. Se asume el cumplimiento de esta condición cuando el sistema resultante cumple con los objetivos de calidad y puede ser construido con los recursos disponibles. Este proceso no produce un resultado cuantitativo ya que no resulta de interés, puesto que el sistema no está construido aún. El objetivo que se persigue es aprender cómo un atributo de calidad es afectado por una decisión de diseño arquitectónico, para que de esta manera se pueda estudiar con cuidado dicha decisión.

El primer paso para evaluación de la arquitectura de *software* es conocer qué es lo que se quiere probar para poder establecer la base para la evaluación, puesto que la intención es saber qué se puede evaluar y qué no. Si las decisiones que se toman la sobre arquitectura de

software determinan los atributos de calidad del sistema, entonces es posible evaluar las decisiones arquitectónicas con respecto al impacto sobre dichos atributos. La garantía de una arquitectura correcta cumple un papel fundamental en el éxito general del proceso de desarrollo de software, además del cumplimiento de los atributos de calidad del sistema. (Eeles, 2006)

Evaluar una arquitectura de software sirve para prevenir todos los posibles problemas de un diseño que no cumple con los requerimientos de calidad y para saber que tan adecuada es la arquitectura de software definida para el sistema. De la evaluación de una arquitectura de software no se obtienen respuestas del tipo si, no, bueno o malo, sino que explica cuáles son los puntos de riesgo del diseño evaluado, es decir, fortalezas y debilidades identificadas de la arquitectura de software.

El propósito de realizar evaluaciones a la arquitectura, es para analizar e identificar riesgos potenciales en su estructura y sus propiedades, que puedan afectar al sistema de software resultante, verificar que los requerimientos no funcionales estén presentes en la arquitectura, así como determinar en qué grado se satisfacen los atributos de calidad.

4.2. Atributos de Calidad

Los factores o atributos de calidad, como también se les conoce, representan características o requerimientos adicionales, independientes de los requisitos funcionales, que el sistema debe satisfacer. También pueden ser definidos como las propiedades de un servicio que presta el sistema a sus usuarios(Camacho y col., 2004). Los mismos son la base para la evaluación de una arquitectura, pero por si solos no son suficiente para juzgar la adecuabilidad de la misma, ya que sus descripciones pueden ser ambiguas y estar sujetas a diferentes interpretaciones. Es decir, que los atributos de calidad no son cantidades absolutas, sino que dependen de un contexto en particular (55). Estos atributos pueden dividirse en dos categorías:

- Observables vía ejecución: aquellos atributos que se determinan del comportamiento del sistema en tiempo de ejecución.
- No observables vía ejecución: aquellos atributos que se establecen durante el desarrollo del sistema.

Los atributos de calidad por los que se puede evaluar son: desempeño (performance), mantenibilidad (maintainability), confiabilidad (reliability), seguridad externa (safety) y seguridad interna (security). La descripción de estos atributos se pueden observar en el anexo 3 6.1.

4.3. Modelos de Calidad

Los atributos de calidad pueden organizarse y descomponerse de maneras diferentes, en lo que se conoce como modelos de calidad. La ventaja de los mismos es que permiten definir y medir la calidad, además, contribuyen a comprender las relaciones existentes entre diferentes características de un producto software.

Modelo de McCall

El modelo de McCall, Richards y Walters describe la calidad como un concepto elaborado mediante relaciones jerárquicas entre factores de calidad, en base a criterios y métricas de calidad. Este enfoque es sistemático, y permite cuantificar la calidad a través de las siguientes fases (Camacho y col., 2004):

- Determinación de los factores que influyen sobre la calidad del software.
- Identificación de los criterios para juzgar cada factor.
- Definición de las métricas de los criterios y establecimiento de una función de normalización que define la relación entre las métricas de cada criterio y los factores correspondientes.
- Evaluación de las métricas.
- Correlación de las métricas a un conjunto de guías que cualquier equipo de desarrollo podría seguir.
- Desarrollo de las recomendaciones para la colección de métricas.
- Los factores de calidad se concentran en tres aspectos importantes de un producto de software: características operativas, capacidad de cambios y adaptabilidad a nuevos entornos.

Los factores propuestos por el modelo se describen a continuación (R. S. Pressman, 2010):

- Corrección: grado en el que un programa satisface sus especificaciones y cumple con los objetivos de la misión del cliente.
- Confiabilidad: grado en el que se espera que un programa cumpla con su función y con la precisión requerida.
- Eficiencia: cantidad de recursos de cómputo y de códigos requeridos por un programa para llevar a cabo su función.

- Integridad: grado en el que es posible controlar el acceso de personas no autorizadas al software o a los datos.
- Usabilidad: esfuerzo que se requiere para aprender, operar, preparar las entradas e interpretar las salidas de un programa.
- Facilidad de recibir mantenimiento: esfuerzo requerido para detectar y corregir un error en un programa.
- Flexibilidad: esfuerzo necesario para modificar un programa que ya opera.
- Susceptibilidad de someterse a pruebas: esfuerzo que se requiere para probar un programa a fin de garantizar que realiza la función que se pretende.
- Portabilidad: esfuerzo que se necesita para transferir el programa de un ambiente de sistema de hardware o software a otro.
- Reusabilidad: grado en el que un programa, o sus partes, pueden reutilizarse en otras aplicaciones.
- Interoperabilidad: esfuerzo requerido para acoplar un sistema con otro.

Modelo ISO/IEC 9126

Este estándar fue desarrollado para identificar los atributos clave de calidad para un producto de software. Es una simplificación del Modelo de McCall e identifica seis características básicas de calidad que pueden estar presentes en cualquier producto de software (Camacho y col., 2004). El estándar provee una descomposición de las características en subcaracterísticas.

Modelo ISO/IEC 9126 adaptado para arquitectura de software

Es una adaptación del modelo ISO/IEC 9126 de calidad para efectos de la evaluación de arquitecturas de software (Camacho y col., 2004). A continuación se exponen cómo los requisitos de calidad son refinados y adaptados a la arquitectura de software, teniendo en cuenta que las características y sub-características se consideran independientes (Vigil Regalado, 2009).

Funcionalidad Adecuación: Esta característica se basa en que el producto posea funciones necesarias para las tareas que debe desempeñar. Se identifican todas las funcionalidades del sistema y cada una de ellas se refina en un atributo cuyo valor es sí (1) o no (0) expresando sentido de presencia o ausencia del atributo.

Interoperabilidad: Evidencia la capacidad del producto de establecer una interacción entre sus componentes y/o con otros sistemas externos. Identifica los conectores de comunicación con sistemas externos y es refinada en un atributo cuyo valor es sí o no.

Seguridad: La capacidad para prevenir el acceso no autorizado a programas o datos. A nivel arquitectónico significa contar con un mecanismo o dispositivo (software o hardware) para llevar a cabo esta tarea de forma explícita. Puede ser un componente o una funcionalidad integrada en un componente. Es refinada en un atributo cuyo valor es sí o no, dependiendo de la presencia o no del mecanismo o dispositivo.

Fiabilidad Tolerancia a fallos y Recuperación: Refleja la capacidad de mantener un nivel determinado de rendimiento en caso de fallos de software o de vulnerabilidad en su interfaz especificada. Significa contar con un mecanismo de software que puede ser bien un componente independiente o una funcionalidad integrada. Es refinada en un atributo cuyo valor es sí o no, dependiendo de la presencia o no del mecanismo.

Mantenibilidad Facilidad de adaptación al cambio: Capacidad del producto de software para permitir una modificación específica que debe aplicarse.

4.4. Técnicas de evaluación de arquitectura

A pesar de que los modelos de calidad logren medir el grado de cumplimiento de los factores de calidad en un producto específico, la evaluación de la arquitectura de software no abarca solamente ese marco de medición. Evaluar una arquitectura implica además, llevar a cabo un proceso de priorización de características de calidad y valorar mediante escenarios que elemento arquitectónico debe ser cambiado para mejorar atributos de rendimiento o mantenibilidad (Vigil Regalado, 2009). Por estas razones, a continuación se analizan un conjunto de técnicas y métodos que separan el concepto de calidad de la evaluación arquitectónica y va más enfocado a la arquitectura de software como disciplina.

Con el objetivo de tomar decisiones de tipo arquitectónico en las fases tempranas del desarrollo, son necesarias técnicas que requieran poca información detallada y puedan conducir a resultados relativamente precisos. En este sentido, existen un grupo de técnicas para la evaluación de la calidad, las cuales se dividen en dos grupos: cualitativas y cuantitativas. Por lo general, las técnicas de evaluación cualitativas son utilizadas cuando la arquitectura está en construcción, mientras que las cuantitativas, se usan cuando la arquitectura ya ha sido implantada (CuriositySec, 2014).

Tabla 4.1: Tabla de técnicas de evaluación y sus instrumentos asociados

Técnicas de Evaluación	Instrumentos de Evaluación
Basada en Escenarios	■ Profiles. ■ Utility Tree.
Basada en Simulación	■ ADLs. ■ Modelos de colas.
Basada en Modelos Matemáticos	■ Cadenas de Markov. ■ *Reliability* y *Block Diagrams*.
Basada en Experiencia	■ Intuición y experiencia. ■ Tradición. ■ Proyectos similares.

Esta medición contiene diferentes técnicas de evaluación: (Ver anexo 4.1)

- Evaluación Basada en Escenarios.
- Evaluación Basada en Simulación.
- Evaluación Basada en Modelos Matemáticos.
- Evaluación Basada en Experiencia.

Escenarios: Un escenario es una breve descripción de la interacción de alguno de los involucrados en el desarrollo del sistema. Un escenario consta de tres partes: el estímulo, el contexto y la respuesta. El estímulo es la parte del escenario que explica o describe lo que el involucrado en el desarrollo hace para iniciar la interacción con el sistema. Puede incluir ejecución de tareas, cambios en el sistema, ejecución de pruebas, configuración, etc. El contexto describe qué sucede

en el sistema al momento del estímulo. La respuesta describe, a través de la arquitectura, cómo debería responder el sistema ante el estímulo. Este último elemento es el que permite establecer cuál es el atributo de calidad asociados (Camacho y col., 2004).

Entre las ventajas de su uso están:

- Son simples de crear y fáciles de entender.
- Son poco costosos y no requieren mucho entrenamiento.
- Son efectivos.

Actualmente las técnicas basadas en escenarios cuentan con dos instrumentos de evaluación relevantes, a saber: el Utility Tree propuesto por Kazman et al. (2001), y los Profiles, propuestos por Bosch (2000).

Utility Tree (Árbol de Utilidad):

Es un esquema en forma de árbol que presenta los atributos de calidad de un sistema de software, refinados hasta el establecimiento de escenarios que especifican con suficiente detalle el nivel de prioridad de cada uno. La intención del uso del Utility Tree 4.6 es la identificación de los atributos de calidad más importantes para un proyecto particular. El Utility Tree contiene como nodo raíz la utilidad general del sistema. Los atributos de calidad asociados al mismo conforman el segundo nivel del árbol los cuales se refinan hasta la obtención de un escenario lo suficientemente concreto para ser analizado y otorgarle prioridad a cada atributo considerado. Cada atributo de calidad perteneciente al árbol contiene una serie de escenarios relacionados, y una escala de importancia y dificultad asociada, que será útil para efectos de la evaluación de la arquitectura.

Para especificar la calidad se hace uso de un árbol de utilidad, el cual tiene en la raíz la bondad o utilidad del sistema, en el segundo nivel del árbol se encuentran los atributos de calidad y las hojas del árbol son los escenarios, los cuales representan mecanismos mediante los cuales extensas declaraciones de cualidades son hechas específicas y posibles de evaluar. La generación del árbol de utilidad permite establecer prioridades.(Kazman y col., 2000)

Perfiles (Profiles)

Un perfil (Profile) es un conjunto de escenarios, generalmente con alguna importancia relativa asociada a cada uno de ellos. El uso de perfiles permite hacer especificaciones más precisas del requerimiento para un atributo de calidad. Los perfiles tienen asociados dos formas de especificación: perfiles completos y perfiles seleccionados.

Los perfiles completos definen todos los escenarios relevantes como parte del perfil. Esto permite al ingeniero de software realizar un análisis de la arquitectura para el atributo de calidad estudiado de una manera completa, puesto que incluye todos los casos posibles. Su uso se reduce a sistemas relativamente pequeños y sólo es posible predecir conjuntos de escenarios completos para algunos atributos de calidad.

Los perfiles seleccionados se asemejan a la selección de muestras sobre una población en los experimentos estadísticos. Se toma un conjunto de escenarios de forma aleatoria, de acuerdo a algunos requerimientos. La aleatoriedad no es totalmente cierta por limitaciones prácticas, por lo que se fuerza la realización de una selección estructurada de elementos para el conjunto de muestra. Si bien es informal, permite hacer proposiciones científicamente válidas.

La tabla 4.2 presenta para cada atributo de calidad, el perfil asociado, la forma en que se definen las categorías, el significado de los "pesos" y posibles métricas de evaluación, de acuerdo al planteamiento de Bosch (2000).

Tabla 4.2: Tabla de perfiles, categorías y pesos asociados a atributos de calidad según Bosch (2000)

Atributo	**Perfil**	**Categorías**	**Pesos**
Mantenibilidad	Perfil de mantenimiento	Se organizan alrededor de las interfaces del sistema (sistema operativo, interfaces con otros sistemas). Los escenarios de cambio describen modificaciones en los requerimientos.	Indican probabilidad de ocurrenc cambio de escenario en un perío tiempo.
Desempeño	Perfil de uso	Descompone los escenarios de uso basado en los tipos de usuarios y/o interfaces del sistema.	Representan la frecuencia relativa c cenario.
Confiabilidad	Perfil de uso	Confiabilidad de los componentes, genera la confiabilidad de los escenarios de uso.	Indican la robustez del sistema.
Seguridad Interna	Perfil de seguridad	Basada en todas las interfaces del sistema.	Indican la probabilidad de falla.
Seguridad Externa	Perfil de peligro	Se organizan de acuerdo a documentos de certificación.	Indican la probabilidad de falla u oc cia de consecuencias desastrosas.

Evaluación basada en modelos matemáticos: Se utiliza para evaluar atributos de calidad operacionales, permite una evaluación estática de los modelos de diseño arquitectónico, se pre-

sentan como alternativa a la simulación, dado que evalúan el mismo tipo de atributos. Ambos enfoques pueden ser combinados, utilizando los resultados de uno como entrada para el otro. Entre los instrumentos que se cuentan para las técnicas de evaluación de arquitecturas de software basada en modelos matemáticos, se encuentran las Cadenas de Markov y los Reliability Block Diagrams.

Entre las desventajas que presenta esta técnica se encuentra la inexistencia de modelos matemáticos apropiados para los atributos de calidad relevantes y el hecho de que el desarrollo de un modelo de simulación completo puede requerir esfuerzos sustanciales.

Evaluación basada en experiencia: Existen dos tipos de evaluación basada en experiencia: la evaluación informal, que es realizada por los arquitectos de software durante el proceso de diseño, y la realizada por equipos externos de evaluación de arquitecturas.

Evaluación basada en simulación: En el ámbito de las simulaciones, se encuentra la técnica de implementación de prototipos (prototyping). Esta técnica implementa una parte de la arquitectura de software y la ejecuta en el contexto del sistema. Es utilizada para evaluar requerimientos de calidad operacional, como desempeño y confiabilidad. Para su uso se necesita mayor información sobre el desarrollo y disponibilidad del hardware, y otras partes que constituyen el contexto del sistema de software. Se obtiene un resultado de evaluación con mayor exactitud (Bosch, 2000).

La evaluación basada en simulación utiliza una implementación de alto nivel de la arquitectura de software. La finalidad es evaluar el comportamiento de la arquitectura bajo diversas circunstancias. En el ámbito de las simulaciones se encuentra la técnica de implementación de prototipos, es utilizada para evaluar requerimientos de calidad operacional como desempeño y disponibilidad. Para su uso se necesita mayor información sobre el desarrollo y disponibilidad del hardware y otras partes que constituyen el contexto del sistema de software por lo que se obtiene un resultado de evaluación con mayor exactitud (Bosch, 2000).

Utiliza una implementación de alto nivel de la arquitectura de software. La finalidad es evaluar el comportamiento de la arquitectura bajo diversas circunstancias.

Los pasos para utilizar esta técnica son:

- 1) Definición e implementación del contexto.
- 2) Implementación de los componentes arquitectónicos.
- 3) Implementación del perfil.
- 4) Simulación del sistema e inicio del perfil.

4.5. Métodos de evaluación:

Cuando se trata de garantizar calidad en una etapa temprana, un punto de partida importante es la arquitectura del software. Ésta permite propiciar o inhibir la mayoría de los atributos de calidad esperados en él.

Un método de evaluación sirve de guía a los involucrados en el desarrollo del sistema, en la búsqueda de conflictos que puede presentar una arquitectura, y sus soluciones. Por esta razón, resulta conveniente estudiar los métodos de evaluación de arquitecturas de software propuestos hasta el momento. (**KazmanTec**)

Software Architecture Analysis Method (SAAM)

El método fue originalmente creado para el análisis de la modificabilidad de una arquitectura, pero en la práctica ha demostrado ser muy útil para evaluar de forma rápida distintos atributos de calidad, tales como modificabilidad, portabilidad, escalabilidad e integrabilidad.

El método de evaluación SAAM se enfoca en la enumeración de un conjunto de escenarios que representan los cambios probables a los que estará sometido el sistema en el futuro. Como entrada principal, es necesaria alguna forma de descripción de la arquitectura a ser evaluada y como principales salidas de la evaluación del método SAAM son las siguientes:

- Una proyección sobre la arquitectura de los escenarios que representan los cambios posibles ante los que puede estar expuesto el sistema.
- Entendimiento de la funcionalidad del sistema, e incluso una comparación de múltiples arquitecturas con respecto al nivel de funcionalidad que cada una soporta sin modificación.

Con la aplicación de este método, si el objetivo de la evaluación es una sola arquitectura, se obtienen los lugares en los que la misma puede fallar, en términos de los requerimientos de modificabilidad. Para el caso en el que se cuenta con varias arquitecturas candidatas, el método produce una escala relativa que permite observar qué opción satisface mejor los requerimientos de calidad con la menor cantidad de modificaciones.

Architecture Trade-off Analysis Method (ATAM)

El Método de Análisis de Acuerdos de Arquitectura (ATAM por sus siglas en inglés), es un método de evaluación de arquitectura de software desarrollado e impulsado por el Instituto de Ingeniería de Software (Software Engineering Institute, SEI), este centra su actividad de evaluación en la interacción entre los diferentes atributos de calidad arquitectónica y basa sus

evaluaciones sobre los escenarios desarrollados por los involucrados y un equipo de evaluación.

El propósito de ATAM es evaluar las consecuencias de las decisiones arquitectónicas sobre los atributos de calidad necesarios. Es un método de identificación de riesgos, o sea detecta las áreas de riesgos potenciales en la arquitectura de un sistema. Puede hacerse en una fase temprana en el ciclo de vida del desarrollo de software. Evalúa de una forma temprana los artefactos del diseño arquitectónico. No es necesario el análisis detallado sobre los atributos de calidad, sino una identificación de tendencias; no se trata de predecir con precisión el comportamiento de un atributo de calidad, ya que es imposible en una etapa temprana del diseño tener suficiente información. El interés está en conocer donde un atributo de interés es afectado por las decisiones del diseño arquitectónico.

Por tanto lo que se pretende hacer con ATAM a demás de mejorar la documentación, es registrar los posibles `riesgos`, los `no riesgos`, los puntos de `sensibilidad` y los puntos de `desventajas` que encontramos en el análisis de la arquitectura.

Riesgos: Las decisiones de arquitectura que podría crear problemas en el futuro para algunos atributos de calidad.

No riesgos: Las decisiones arquitectónicas que sean adecuadas al atributo de calidad que afectan.

Desventajas: Las decisiones de arquitectura que tienen un efecto en más de un atributo de calidad.

Puntos de sensibilidad: Una propiedad de uno o más componentes, y/o las relaciones entre componentes, fundamental para el logro de un determinado requisito de atributo de calidad.

Tabla 4.3: Pasos y fases del método ATAM

Fase 1: Presentación.	
Paso 1: Presentación del ATAM.	El equipo de evaluación presenta un panorama general de los pasos de ATAM trata de establecer las expectativas, las técnicas utilizadas y de los resultados del proceso.
Paso 2: Presentación de las metas del negocio.	Se realiza la descripción de las metas del negocio que motivan el esfuerzo, y aclara que se persiguen objetivos de tipo arquitectónico. Se presenta brevemente el negocio y el contexto de la arquitectura.
Paso 3: Presentación de la arquitectura.	El arquitecto presenta un panorama de la arquitectura, describe la arquitectura, enfocándose en cómo ésta cumple con los objetivos del negocio.
Fase 2: Investigación y Análisis.	
Paso 4: Identificación de los enfoques arquitectónicos.	El equipo de evaluación y el arquitecto deben detallar los planteamientos arquitectónicos descubiertos en el paso anterior. Estos elementos son detectados, pero no analizados.
Paso 5: Generación del Utility Tree.	Se identifican, priorizan, definen y refinan los atributos de calidad más importantes en un formato de árbol de utilidad y que engloban la “utilidad” del sistema, especificados en forma de escenarios. Se anotan los estímulos y respuestas, así como se establece la prioridad entre ellos.
Paso 6: Análisis de los enfoques arquitectónicos.	Con base en los resultados del establecimiento de prioridades del paso anterior, se analizan los elementos del paso 4. Se identifican riesgos arquitectónicos, puntos de sensibilidad y puntos de balance. Se utilizan las preguntas similares a las presentadas en el paso 5.
Fase 3: Pruebas	
Paso 7: Lluvia de ideas y establecimiento de prioridad de escenarios	Con la colaboración de todos los involucrados, se complementa el conjunto de escenarios y se le da prioridad a los escenarios sobre la base de su importancia relativa.
Paso 8: Análisis de los enfoques arquitectónicos.	Este paso repite las actividades del paso 6, haciendo uso de los resultados del paso 7. Los escenarios son considerados como casos de prueba para confirmar el análisis realizado hasta el momento.
Fase 4: Reporte.	
Paso 9: Presentación de los resultados.	Basado en la información recolectada a lo largo de la evaluación del ATAM, se presentan los hallazgos a los participantes. El equipo de evaluación recapitula los pasos de ATAM.

Active Reviews for Intermediate Designs (ARID)

De acuerdo con Kazman et al. (2001) el método ARID es conveniente para realizar la evaluación de diseños parciales en las etapas tempranas del desarrollo. El método ARID consta de nueve pasos recogidos en dos fases:

Tabla 4.4: Tabla: Pasos para la evaluación del método ARID

Fase 1: Actividades previas	
Paso 1: Identificación de los encargados de la revisión	Los encargados de la revisión son los ingenieros de software que se espera que usen el diseño, y todos los involucrados en el diseño.
Paso 2: Preparar el informe de diseño	El diseñador prepara un informe que explica el diseño. Se incluyen ejemplos del uso del mismo para la resolución de problemas reales. Esto permite al facilitador anticipar el tipo de preguntas posibles, así como identificar áreas en las que la presentación puede ser mejorada.
Paso 3: Preparar los escenarios base	El diseñador y el facilitador preparan un conjunto de escenarios base. De forma similar a los escenarios del ATAM y el SAAM.
Paso 4: Preparar los materiales	Se reproducen los materiales preparados para ser presentados en la segunda fase.
Fase 2: Revisión	
Paso 5: Presentación del ARID	Se explica los pasos del ARID a los participantes.
Paso 6: Presentación del diseño	El líder del equipo de diseño realiza una presentación, con ejemplos incluidos. El objetivo es verificar que el diseño es conveniente.
Paso 7: Lluvia de ideas y establecimiento de prioridad de escenarios	Se establece una sesión para la lluvia de ideas sobre los escenarios y el establecimiento de prioridad de escenarios. Se proponen escenarios, los cuales son sometidos a votación, y se utilizan los que resultan ganadores para hacer pruebas sobre el diseño.
Paso 8: Aplicación de los escenarios	Comenzando con el escenario que contó con más vot os, el facilitador solicita pseudo-código que utiliza el diseño para proveer el servicio, y el diseñador no debe ayudar en esta tarea.
Paso 9: Resumen	Al final, el facilitador recuenta la lista de puntos tratados, pide opiniones de los participantes sobre la eficiencia del ejercicio de revisión, y agradece por su participación.

Para la selección del método basado en arquitectura se realizó una comparación entre los métodos estudiados: SAAM, ATAM y ARID analizando aspectos relevantes como los atributos de calidad que contempla cada uno, los objetivos que analizan, las etapas o fases del proyecto en las que se aplican, así como sus principales enfoques.

Tabla 4.5: Tabla: Comparación entre los métodos basados en la arquitectura (Kazman 2001)

	ATAM	**SAAM**
Atributos de calidad	■ Modificabilidad ■ Seguridad ■ Confiabilidad ■ Desempeño	■ Modificabilidad ■ Funcionalidad
Objetivos Analizados	■ Estilos arquitectónicos ■ Documentación ■ Flujo de datos	■ Documentación
Etapas del proyecto en las que se aplica	Luego de que la arquitectura ha sido establecida	Luego de que la arquitectura con funcionalidad ubicada en módulos.
Enfoques utilizados	■ Utility Tree y lluvia de ideas para articular los requerimientos de calidad. ■ Análisis arquitectónico que detecta puntos sensibles, puntos de balance y riesgos.	■ Lluvia de ideas para escenarios y articular los requerimientos de calidad. ■ Análisis de los escenarios para verificar funcionalidad o estimar el costo de los cambios.

4.6. Validación de la arquitectura:

Evaluar una arquitectura de software sirve para prevenir todos los posibles riesgos de un diseño que no cumple con los requerimientos de calidad y para saber que tan adecuada es la arquitectura de software diseñada para el sistema. Una buena práctica para decidir cuando hay que realizar una evaluación de una arquitectura de software es —cuando el equipo de desarrollo comience a tomar decisiones que dependan de la arquitectura definida.

La arquitectura es, sin embargo, un aspecto tan importante dentro del desarrollo que es conveniente realizar actividades de verificación de la misma de forma temprana, con el fin de identificar problemas que podría resultar muy costoso eliminar posteriormente. La evaluación de la arquitectura de software permite justamente realizar la verificación del diseño y es una categoría de actividades que cubre el conjunto de aspectos relacionados con el desarrollo de arquitectura de software.

En el momento de evaluar la arquitectura de software definida por CREAD sería útil realizarse las siguientes interrogantes:

- ¿Se ha creado una línea base de arquitectura?
- ¿Es robusta y adaptable?
- ¿Es escalable?
- ¿Es auditable y testable?
- ¿Se ajusta a la construcción y el uso de la base de datos?

1

La selección correcta de la arquitectura permite la partición del sistema y que éstas colaboren entre sí permitiendo una solidificación de las interfaces las cuales mejoran la distribución del trabajo entre los equipos de desarrollo.

Actividades de revisión de arquitectura:

- Revisión inicial y documentación de arquitectura física, lógica, de procesos, sus casos de uso principales y su estructura, marco de implementación.
- Una vez obtenida una representación básica del sistema, realizar un análisis del mismo con el fin de detectar oportunidades de mejora.
- Revisión de procesos de desarrollo, guías.
- Recolección de mediciones del comportamiento del sistema actual tales como: métricas de uso de recursos de los componentes del sistema: base de datos, aplicaciones, procesos.
- Análisis de los escenarios que apunten a validar requerimientos tales como escalabilidad, performance, portabilidad, usabilidad.

[1] Evitar tener dos herramientas que resuelvan la misma problemática.

- Detección de posibilidades de mejora en aspectos como riesgos, arquitectura general, procesos.

Para la correcta y eficiente evaluación de la arquitectura se tomaron en cuenta diferentes atributos de calidad según la norma ISO/IEC 9126 e incluso se incorporaron otros que están relacionados directamente con la arquitectura de software: seguridad, rendimiento, funcionalidad, eficiencia, usabilidad y mantenibilidad. Además se decidió hacer uso de las técnicas de evaluación basadas en escenarios con el instrumento de evaluación 'Utility Tree', el cual a pesar de ser incluido en el método ATAM es muy útil porque ayuda a entender las consecuencias de las decisiones arquitectónicas respecto a los atributos de calidad.

Tabla 4.6: Árbol de Utilidad CREAD

Árbol de Utilidad			
Atributo de Calidad	Subcaracterística	Escenario	Prioridad
Funcionalidad	Seguridad	El sistema debe restringir el acceso a los datos.	A
		Un usuario no puede realizar una modificación de los datos si no posee la última versión de los mismos.	A
	Adecuación	El sistema cumple los RF y RNF solicitudes por el cliente.	A
	Interoperabilidad	El sistema debe ser capaz de interactuar con otros sistemas desplegados en el centro CREAD	A
Fiabilidad	Tolerancia a fallos & Recuperación	Cuando se compila el código y empieza a construirse la si ocurren fallos se vuelve a reconstruir desde ese mismo paso sin tener que construir todo el sistema desde el principio	A
Usabilidad	Comprensibilidad, Aprendibilidad & Atractividad.	El software debe ser legible para el usuario	A
Eficiencia	Usabilidad de Recursos & Conectividad	El sistema debe ser capaz de actualizar la información bidireccionalmente durante el tiempo sin conexión.	A
	Tiempo de Respuesta	Los tiempos de respuesta a las peticiones realizadas por los usuarios deben ser mínimos.	A
Mantenibilidad	Cambiabilidad & Estabilidad	El sistema brida la posibilidad de cambiar de SGBD con facilidad y de emigrar hacia otro ORM dentro del mismo framework de desarrollo.	M
	Cambiabilidad & Facilidad de adaptación al cambio	El sistema debe permitir que se le puedan adicionar componentes, subsistemas, módulos o nuevas funcionalidades	A
	Facilidad de adaptación al cambio.	Los componentes pueden instalarse fácilmente en todos los ambientes donde debe funcionar, básicamente en Windows y Linux.	A
Portabilidad		Es fácil instalar las dependencias del sistema.	M

ATAM es un método utilizado para identificar riesgos, Es sumamente importante en ATAM registrar cualquier riesgo, punto sensible y puntos de intercambio, esto significa que se detecta áreas con potenciales riesgos dentro de la arquitectura de un sistema de software complejo por lo que este método tiene algunas carácterísticas a tener en cuenta:

- Debe ser ejecutado tempranamente en el ciclo de desarrollo del software.

- Debe ejecutarse relativamente con un bajo costo y rápido, ya que evalúa los artefactos del diseño arquitectónico.
- Produce un análisis en proporción con el nivel de detalle de la especificación de la arquitectura.

Es sumamente importante en ATAM registrar cualquier riesgo, punto sensible y puntos de intercambio

Los riesgos son decisiones arquitecturalmente importantes, que no han sido tomadas o decisiones que han sido tomadas pero las consecuencias no han sido entendidas a plenitud.

Los puntos sensibles son parámetros en la arquitectura en los cuales la respuesta medible de algunos atributos de calidad es altamente correlacionada.

Un punto de intercambio (trade-off) es descubierto en la arquitectura cuando un parámetro de construcción arquitectural es un anfitrión para más de un punto sensible donde los puntos de calidad medibles son afectados indistintamente por cambio en el parámetro.

Una vez identificados los riesgos presentes en la arquitectura resulta necesario efectuar la toma de decisiones por parte de los stakeholders y el equipo que intervino en el proceso de ejecución de las pruebas de concepto de la arquitectura: arquitecto de software, líder de proyecto y algunos desarrolladores.

La toma de decisiones se ejecuta con el objetivo de mitigar los riesgos en la arquitectura desde el punto de vista de los atributos de calidad analizados y dar paso a la construcción del sistema.

4.7. Conclusiones parciales:

En el presente capítulo se han analizado los principales elementos asociados a la evaluación de la arquitectura, así como sus métodos y técnicas, propiciando una correcta y adecuada selección de los atributos de calidad para el análisis de la arquitectura de software del proyecto decidimOS según la norma ISO/IEC 9126: funcionalidad, fiabilidad, usabilidad, eficiencia, mantenibilidad y portabilidad; todos relacionados estrechamente con la arquitectura de software.

En este análisis de la arquitectura se aplicó el método ATAM como parte del proceso de evaluación temprana así como la técnica de evaluación basada en escenarios y el procedimiento

de árbol de utilidad. Después de aplicados estos procedimientos se obtuvo un resultado de 12 escenarios clasificados con un prioridad Alta (A), Media (M) y Baja (B) dando como resultado 10 puntos sensibles y 2 puntos de riesgos por lo que se ha decidido en conjunto con el equipo de desarrollo del centro, implementar el sistema llegando a la conclusión de que el sistema es funcional.

5 Conclusiones Generales

Con el objetivo de definir la arquitectura de software se analizaron un conjunto de herramientas, arquitecturas, metodologías para utilizar y llevar a cabo en el proyecto. En el soporte técnico se utilizó la herramienta Docker con el orquestador Docker-Compose el cual propicia una escalabilidad horizontal poniendo en valor las buenas prácticas de trabajo de la arquitectura de software contribuyendo en conjunto con la metodología DevOps a un mejor flujo de trabajo entre los desarrolladores y propiciando un entorno de desarrollo ágil seguro y confiable donde el sistema cumpliera con las expectativas relacionadas con la estandarización en el desarrollo de las aplicaciones.

Luego de este estudio se construyeron los lineamientos base y mecanismos arquitectónicos a seguir los cuales fueron el punto de partida para realizar la descripción de la arquitectura a través de diagramas de despliegue y diagrama de actividades, los cuales proporcionaron una visión común y propuesta de solución desde diferentes perspectivas del sistema. Además se analizó la vista Lógica del proyecto decidimOS.

Como parte de la evaluación de la arquitectura diseñada se aplicó el método de evaluación de la arquitectura ATAM, el cual contribuyó a que se alcanzara un sistema funcional y se diera paso a la implementación del sistema. Con la evaluación de la arquitectura quedó evidenciada la forma en que la arquitectura propuesta hizo tratamiento de los diferentes atributos de calidad analizados.

6 Aportes Realizados

En el presente Trabajo de Diploma se ha investigado e implementado soluciones modernas con respecto a las **Arquitecturas de Despliegue** utilizando soluciones complejas como Nginx usando un *proxy inverso* y balanceador de carga, así como también se ha utilizado Gunicorn para automatizar el proceso de despliegue y aumentar el rendimiento de la aplicación, se han realizado las configuraciones pertinentes a los archivos de configuración python y angular para su automatización en el proceso, además también se ha otorgado seguridad a las variables o configuraciones de la base de datos del sistema.

Se ha realizado un estudio exhaustivo en la búsqueda de la solución más óptima para la implementación de la arquitectura así como la herramienta. Se desarrolló la validación de la arquitectura con herramientas novedosas como JMeter donde se ha expuesto a pruebas de rendimiento lo cual aporta confiabilidad al sistema y a los métodos empleados.

Recomendaciones

- Profundizar sobre las vistas arquitectónicas.
- Investigar sobre otros orquestadores diferentes a Docker-Compose como son Docker Swarm y Kubernetes los cuáles presentan diferentes ventajas con respecto al utilizado en este proyecto.
- Desarrollar un nexus local para la utilización de dependencias o paquetes ya usados previamente para mayor agilidad.
- Actualizar las dependencias y los servicios para futuro soporte técnico.

*

Bibliografía

ANDERSON, C., 2015. Docker [software engineering]. *Ieee Software*. Vol. 32, n.º 3, págs. 102-c3 (vid. pág. 21).

BOOCH, Grady; RUMBAUGH, James y JACOBSON, Jvor, 1999. *El lenguaje unificado de modelado*. España Pearson Educación (vid. pág. 21).

BUCHANAN, I., [s.f.]. Las metodologías ágiles y DevOps: amigos o enemigo? **urlalso**: https://www.atlassian.com/es/agile/devops (vid. pág. 28).

CAMACHO, Erika; CARDESO, Fabio y NUÑEZ, Gabriel., 2004. Arquitecturas de Software. (Vid. págs. 58-60, 63).

CLEMENTS, Paul; KAZMAN, Rick y KLEIN, Mark., 2004. Evaluating Software Architectures: Methods and Case Studies. *ISBN* (vid. pág. 57).

COSTA RICA, Universidad Latina de, 2020. Qué son las TIC y para qué sirven? *Arizona State University* (vid. pág. 16).

CURIOSITYSEC, 2014. Procedimiento para la evaluación de arquitecturas de software basada en componentes. *CuriositySec*. **urlalso**: http://curiositysec.com/procedimiento-para-la-evaluacion-de-arquitecturas-de-software-basadas-en-24%20componentes/ (vid. págs. 57, 61).

CUTRELL, Ed, 2022. El contexto y el diseño de TIC para el desarrollo mundial. *Naciones Unidas* (vid. pág. 16).

DATA, Power, 2022. Cloud: definiciones, servicios, despliegue, su seguridad y privacidad. En: **urlalso**: https://www.powerdata.es/cloud.

DÁVILA, Mauricio; GERMÁN, Martín; CRUTAS, Diego y GARCÍA, Andrés., 2020. Evaluación de arquitecturas de software. *studocu* (vid. pág. 57).

DIAZ CASILLAS, L. y BLANCO, Francisco J., 2010. Sistema basado en reglas para la validación del despliegue de servicios. *Revista Iberoamericana de Inteligencia Artificial*. Vol. 14, págs. 54, 55, 56, 57 (vid. pág. 40).

DMFT, 2010. *Use Cases and Interactions for Managing Clouds*. DMFT. **urlalso**: http://dmtf.org/sites/default/files/standards%20/documents/DSP-IS0103_1.0.0.pdf. (vid. pág. 26).

EELES, P., 2006. Characteristics of a Software Architect. **urlalso**: http://www.ibm.com/developerworks/rational/library/mar06/eeles/ (vid. pág. 58).

ENATEC, 2019. Nube privada: ventajas y desventajas.

GARCÍA PERELLADA, L. R., 2014. Propuesta de arquitectura para una Nube Privada con soporte para Infraestructura como Servicio. *Instituto Superior Politécnico José Antonio Echeverría (ISPJAE)*. Vol. DSP-IS0103 (vid. pág. 26).

GARLAN, D., 2001. Software architecture (vid. pág. 20).

GROWTH PARTNER, I. P. N. E. T., 2020. Nube privada: qué es, tipos y funciones.

HAT, Red, 2018. Qué es la nube híbrida? **urlalso**: https://www.redhat.com/es/topics/cloud-computing/what-is-hybrid-cloud.

HAT, Red, 2019. WHAT is an IDE? **urlalso**: https://www.redhat.com/en/topics/middleware/what-is-ide (vid. pág. 33).

HERNÁNDEZ YEJA, A. y PORVEN RUBIER, J., 2016a. Procedimiento para la seguridad del proceso de despliegue de aplicaciones web. *RCCI*. Vol. 10 (vid. pág. 22).

HERNÁNDEZ YEJA, A. y PORVEN RUBIER, J., 2016b. Procedimiento para la seguridad del proceso de despliegue de aplicaciones web. *RCCI*. Vol. 10 (vid. pág. 23).

IONOS, 2020. Introducción al sistema gestor de base de datos (SGBD). **urlalso**: https://www.ionos.mx/digitalguide/hosting/cuestiones-tecnicas/sistema-gestor-de-base-de-datos-sgbd/ (vid. pág. 35).

JETBRAINS, 2022. Introduction to PyCharm. **urlalso**: https://www.jetbrains.com/help/pycharm/quick-start-guide.html.

KAZMAN, C. Paul; MARK, R. y MARK, P., 2000. *Evaluating Software Architectures: Methods and Case Studies*. Adison Wesley (vid. pág. 63).

KONTIO, M., 2008. *Architectural manifesto: Designing software architectures, Part 5*.

KORD, M., 2011. Patrones GRASP. Patrones GoF. Diferencia entre GRASP y GoF. *TuxNots* (vid. pág. 52).

KRUCHTEN, P., [s.f.]. *Architectural Blueprints–The 4+1 View Model of Software Architecture*.

LAGO, N., 2022a. 6 herramientas del modelado de software: visualiza antes, desarrolla después. **urlalso**: https://saasradar.net/modelado-de-software/ (vid. pág. 32).

LAGO, N., 2022b. Herramientas más usadas en el modelado de software. **urlalso**: https://saasradar.net/modelado-de-software/#Herramientas_de_modelado (vid. pág. 32).

LAGO, N., 2022c. Introducción a Visual Paradigm. **urlalso**: https://saasradar.net/modelado-de-software/#Visual_Paradigm (vid. pág. 32).

LIZAMA, Oscar; LIZAMA, Kindley; GEORDY; MORALES, JeriaJ. I. y GONZALES, Agustín, 2016. Redes de Computadores: Arquitectura Cliente-Servidor. *Universidad Tecnica Federico Santa Maria*, págs. 1-8.

LUCIDCHART, [s.f.(a)]. Actualización a UML 2.0. **urlalso**: https://www.lucidchart.com/pages/es/que-es-el-lenguaje-unificado-de-modelado-uml#section_6 (vid. pág. 30).

LUCIDCHART, [s.f.(b)]. Qué es el lenguaje unificado de modelado? **urlalso**: https://www.lucidchart.com/pages/es/que-es-el-lenguaje-unificado-de-modelado-uml#section_0 (vid. pág. 30).

MARCO, G. Jímenez, 2016. DevOps, la nueva tendencia en el desarollo de sistemas TI, un caso práctico en el análisis de incidencias de software (vid. pág. 28).

MCKENDRICK, R., 2020. *Mastering Docker*. Packt Publishing Ltd. (vid. pág. 50).

MDN, 2021. Introducción a Django. **urlalso**: https://developer.mozilla.org/es/docs/Learn/Server-side/Django/Introduction (vid. pág. 30).

MICROSOFT, [s.f.(a)]. Introduction to Visual Studio Code. **urlalso**: https://code.visualstudio.com/docs (vid. pág. 33).

MICROSOFT, [s.f.(b)]. Why Visual Studio Code? **urlalso**: https://code.visualstudio.com/docs/editor/whyvscode (vid. pág. 34).

OJEDA, M. G., 2019. EL SISTEMA ELECTORAL. *Ius Inkarri*. N.º 8, págs. 91-115 (vid. pág. 17).

ORELLANA, A. Cruz, 2017. *Cloud Solutions SRE Architect* (vid. pág. 54).

PGDG, 2022. PostgreSQL: The World's Most Advanced Open Source Relational Database. **urlalso**: https://www.postgresql.org/ (vid. pág. 35).

PRESSMAN, R. S., 2010. Ingeniería del Software. Un enfoque práctico. *McGraw Hill*. Vol. 7 (vid. págs. 56, 59).

PRESSMAN, Roger S., 2010. *Ingeniería del Software. Un enfoque práctico.* Ed. por HILL, McGraw.

PUERTO, Ordaz, 2009. Metodologías de Desarrollo para Sistemas de tiempo real. Un estudio comparativo. *SCIELO*. Vol. 13, n.º 50 (vid. pág. 56).

PURRIER, J., 2022. What is Rocket and how it's different than Docker. **urlalso**: https://www.ctl.io/developers/blog/post/what-is-rocket-and-how-its-different-than-docker/.

QUIJANO, C. Arjona, 2020. Docker: Creando entorno de desarrollo seguro. *GITCE*.

RAMOS CARDOZZO, D., 2016. *Desarrollo de Software: Requisitos, Estimaciones y Análisis* (vid. pág. 41).

RIVAS, C. I., 2015. Metodologías actuales de desarrollo de software. **urlalso**: https://www.ecorfan.org/bolivia/researchjournals/Tecnologia_e_innovacion/vol2num5/Tecnologia_e_Innovacion_Vol2_Num5_6.pdf (vid. pág. 27).

SERVICES, Amazon Web, 2022. Microservicios. **urlalso**: https://aws.amazon.com/es/microservices/ (vid. pág. 24).

SHAW, M., 1989. Larger Scale Systems Require Higher-Level Abstractions. *IEEE Computer Society*. Vol. 14, n.º 3, págs. 143-146 (vid. pág. 21).

SOLMS, F., 2012. What is software architecture?, págs. 363-373 (vid. pág. 22).

THE POSTGRESQL GLOBAL DEVELOPMENT, G., 2022. Why use PostgreSQL? **urlalso**: https://www.postgresql.org/about/ (vid. pág. 35).

UIT-T, 2012a. *Overview of SDOs involved in cloud computing*. ITU-T (vid. pág. 26).

UIT-T, 2012b. *Requirements and framework architecture of cloud infrastructure*. ITU-T (vid. pág. 26).

VIGIL REGALADO, Ing. Yamila, 2009. *Metodología de evaluación de arquitecturas de software.* Tesis de mtría. Universidad de Las Ciencias Informáticas (vid. págs. 60, 61).

WALLS, M., 2013. Building a DevOps Culture. *Sebastopol: O'Reilly* (vid. pág. 27).

WEGNER, P., 1996. Interoperability. *ACM Computing Surveys (CSUR)*. Vol. 28, n.º 1, págs. 285-287 (vid. pág. 20).

WU, Yiwen; ZHANG, Yang; WANG, Tao y WANG, Huaimin, 2020. Characterizing the occurrence of dockerfile smells in open-source software: An empirical study. *IEEE Access*. Vol. 8, págs. 34127-34139.

ZAPATA SÁNCHEZ, J., 2013. Metodología de Pruebas. **urlalso**: https://pruebasdelsoftware.wordpress.com/2013/01/13/metodologia-de-pruebas/#comments.

ANEXOS

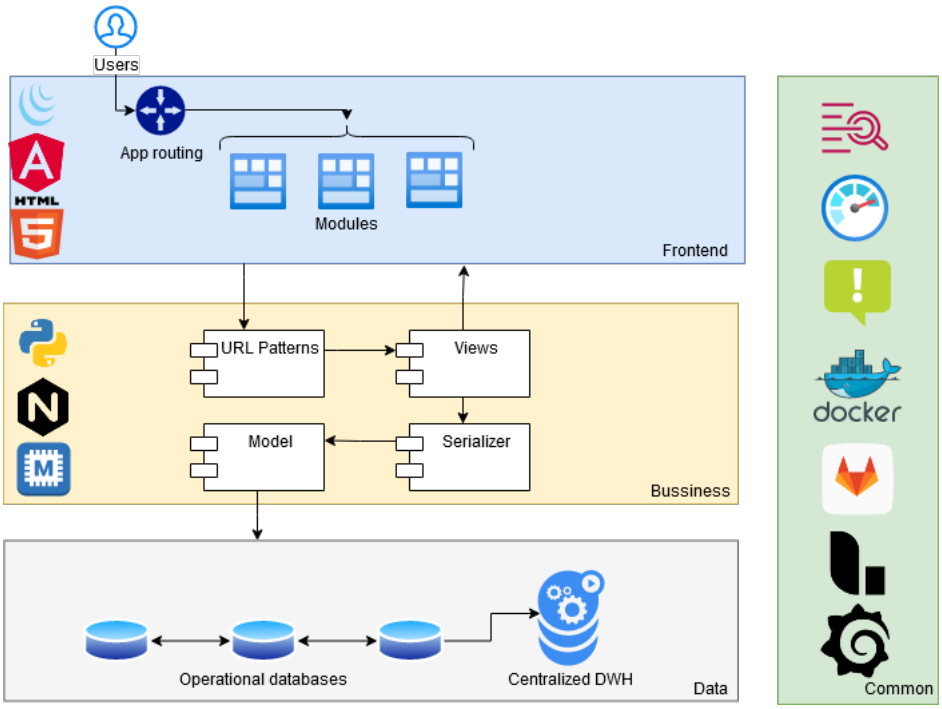


Figura 6.1: Vista de alto nivel de la arquitectura del proyecto decidimOS

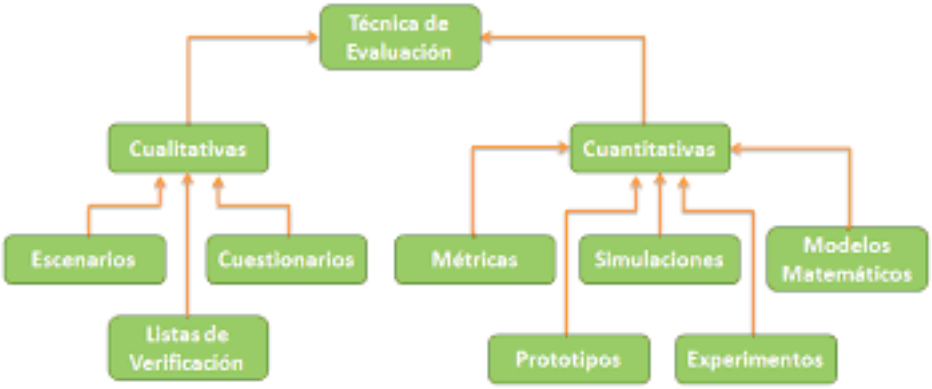


Figura 6.2: Diagrama de las técnicas de evaluación de la arquitectura

Tabla 6.1: Tabla. Descripción de atributos de calidad observables vía ejecución.

Atributo de Calidad	**Descripción**
Disponibilidad	Es la medida de disponibilidad del sistema para el uso.
Confidencialidad	Es la ausencia de acceso no autorizado a la información.
Funcionalidad	Habilidad del sistema para realizar el trabajo para el cual fue concebido.
Desempeño	Es el grado en el cual un sistema o componente cumple con sus funciones designadas, dentro de ciertas restricciones dadas, como velocidad, exactitud o uso de memoria.
Confiabilidad	Es la medida de la habilidad de un sistema para mantenerse operativo a lo largo del tiempo.
Seguridad externa	Ausencia de consecuencias catastróficas en el ambiente. Es la medida de ausencia de errores que generan pérdidas de información.
Seguridad interna	Es la medida de la habilidad del sistema para resistir a intentos de uso no autorizados y negación del servicio, mientras se sirve a usuarios legítimos.