Pontificia Universidad Católica del Ecuador

Facultad De Ingeniería

Escuela de Sistemas

TEMA:

ANÁLISIS COMPARATIVO DE PATRONES DE ARQUITECTURA PARA EL DESARROLLO DE SOFTWARE WEB.

AUTOR:

ALAN ANDRE GUIJARRO POMBOZA

TRABAJO PREVIO A LA OBTENCIÓN DEL TÍTULO DE INGENIERO DE SISTEMAS DE LA INFORMACIÓN

QUITO, MAYO 2023

DEDICATORIA

Dedico este trabajo de titulación a mis padres, por todo el apoyo y comprensión que me han brindado durante todo el transcurso de la carrera. A mis hermanos, con los que he podido crecer personalmente y se han mostrado presentes en todo momento brindando su apoyo incondicional, además de ser una de mis principales fuentes de inspiración.

AGRADECIMIENTO

Quiero agradecer a todos los docentes que me acompañaron durante el transcurso de la carrera, en especial a mi directora de titulación la Dra. Susana Masapanta, que ha sabido guiarme durante todo el proceso y ha sido de gran ayuda para mi formación, no solo profesional, sino también personal.

Agradezco a mis amigos, por compartir grandes momentos llenos de aprendizaje tanto de forma académica como personal. Además, quisiera agradecer a Adriana Salazar, Sergio Barba, Cristian López, Roberto Armas, Jonathan López, Emilia Armijos, Mario Núñez, Camila Abeda, Carlos Araujo y Ariana Cuesta que me brindaron su apoyo en el desarrollo de las encuestas.

RESUMEN

El presente trabajo tiene como objetivo analizar y comparar a detalle los cinco patrones de arquitectura más populares utilizados en el desarrollo de *software web*. Se toman en consideración elementos como el grado de acoplamiento, cohesión y complejidad, tanto en un sentido general como en escenarios específicos contemplados en casos de proyectos de desarrollo, que abarcan diversas situaciones en donde se requieren enfoques diferentes y específicos.

Se utilizan distintos métodos de investigación, entre ellos, la metodología aplicada y el método Delphi dentro de la prospectiva de la información. Con estas técnicas se garantiza la recopilación de datos precisos y la obtención de resultados sólidos y confiables.

Como resultado de este estudio comparativo, se presenta una tabla comparativa que permite evaluar la idoneidad de cada patrón de arquitectura en base a los casos propuestos. Este trabajo puede ser utilizado como soporte para que los desarrolladores de *software* tomen decisiones informadas sobre la selección del patrón de arquitectura más adecuado para diferentes escenarios.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

CAPÍTULO I: INTRODUCCIÓN

1. MARCO DE REFERENCIA

1.1. Justificación

Los patrones de arquitectura de *software web* nacen como resultado de la necesidad de estructurar y diseñar sistemas *web* escalables. Según Brown (2017) los patrones de arquitectura son soluciones probadas a problemas comunes que surgen en el diseño de *software*.

Con el creciente aumento en la demanda de aplicaciones *web* y la necesidad de *software* escalable y eficiente, es fundamental para los desarrolladores contar con conocimientos y habilidades sólidas sobre los patrones de arquitectura de *software web*.

Realizar un análisis comparativo de estos patrones ayuda a tomar una elección adecuada del patrón de arquitectura para cada situación, resultando en la generación de código reutilizable, escalable y preparado para pruebas, de igual forma permite una reducción de costos y optimización de rendimiento.

Para un desarrollador, es importante tener conocimiento de una amplia variedad de patrones de arquitectura de *software*. El dominio de estos patrones le permitirá obtener una ventaja competitiva y crear *software* de alta calidad, esta calidad es esencial para las empresas, ya que les permite cumplir con sus procesos y objetivos de manera eficiente.

Cabe mencionar que existe una gran cantidad de patrones de arquitectura, por lo que es importante que el desarrollador tenga un amplio conocimiento de los más utilizados y relevantes para su proyecto. A pesar de que no es posible conocer todos los patrones existentes en su totalidad, es importante estar familiarizado con ellos para poder elegir el adecuado en cada situación.

Antes de definir el problema a resolver con el presente trabajo se presenta la definición que se ha adoptado para el concepto de patrón de arquitectura. Según Richards (2017), un patrón de arquitectura se refiere a un enfoque general que permite describir la estructura macro de un sistema.

Dicho esto, el trabajo se encuentra enfocado hacia los patrones de arquitectura de alto nivel, definiendo la estructura general de la aplicación y cómo se organizan sus componentes, sin tomar en cuenta a patrones de arquitectura de bajo nivel. Algunos de los patrones de arquitectura que se incluirán son la arquitectura en n capas, cliente-servidor y la arquitectura de microservicios.

2. Planteamiento del problema

"La industria del desarrollo de *software* está experimentando un crecimiento exponencial a medida que cada vez más empresas se digitalizan y buscan soluciones tecnológicas innovadoras para mantenerse competitivas en el mercado global" (Kokol, 2020). Esto ha llevado a un aumento en el número de proyectos de desarrollo *web*, pero a menudo se descuida la estructuración adecuada del *software*, lo que puede afectar negativamente la calidad del producto y reducir la productividad de las organizaciones a las que se dirige.

El desconocimiento sobre el patrón de arquitectura de *software web* adecuado para las necesidades del proyecto o directamente el desarrollo de *software* sin un patrón de arquitectura como base conlleva a un producto pobre en funciones de rendimiento, escalabilidad y estructuración, pudiendo llegar a generar altos costos en el mantenimiento o actualización sobre este.

Existe un desconocimiento sobre los patrones de arquitectura por una gran parte de desarrolladores de *software*, por lo que el proceso de selección del patrón de arquitectura al momento de empezar a desarrollar *software* resulta confuso, en algunas ocasiones se opta

por un patrón de arquitectura erróneo en base a las necesidades del proyecto o incluso se desarrolla el *software* directamente sin la aplicación de estos.

2.1. Objetivo General

Realizar una comparación entre los patrones de arquitectura de *software web* más populares, para identificar y detallar el patrón adecuado para cada situación en el desarrollo de *software web*.

2.2. Objetivos Específicos

- Identificar los patrones de arquitectura de desarrollo de *software web* más populares en la actualidad.
- Analizar el acoplamiento, cohesión y complejidad de los patrones de arquitectura identificados.
- Analizar la escalabilidad de los patrones de arquitectura según el tipo de proyecto.
- Determinar los factores críticos para la implementación de cada patrón de arquitectura identificado.

2.3. Antecedentes

A lo largo de los años, numerosos estudios han analizado y comparado los patrones de arquitectura de *software* para el desarrollo *web*, incluidos la arquitectura cliente-servidor, arquitectura en capas, arquitectura orientada a servicios, arquitectura de microservicios y arquitectura *Serverless*.

En el caso de la arquitectura cliente-servidor, Pautasso et al. (2017) analizaron el impacto de este patrón en el rendimiento y la escalabilidad en un entorno de aplicaciones *web* distribuidas. El estudio de Bass et al. (2015) investigó los desafíos y soluciones asociados con la arquitectura en capas, especialmente en relación con la separación de responsabilidades y la reutilización de componentes.

Newman (2015) comparó la arquitectura orientada servicios con la arquitectura de microservicios, analizando las ventajas y desventajas de cada patrón en relación con la escalabilidad y la resiliencia. Roberts (2016) profundizó en el análisis de la arquitectura de microservicios, evaluando su efecto en la eficiencia del desarrollo y la capacidad de respuesta a los cambios en los requisitos del negocio.

A pesar de que las investigaciones previas contienen información valiosa, no proporcionan una orientación específica para la selección o aplicación de patrones de arquitectura en proyectos reales. Contrastando con el presente proyecto, que mantiene este enfoque.

En conclusión, mientras que investigaciones previas se han centrado en el análisis y comparación de forma individual o en la discusión de las ventajas y desventajas en ciertos enfoques, el presente trabajo abordará un análisis comparativo exhaustivo con respecto a todos los patrones de arquitectura a analizar, profundizando en aspectos específicos de los patrones de arquitectura, tales como el acoplamiento, cohesión, complejidad y escalabilidad en función al tipo de proyecto.

2.4. Alcance

El presente proyecto tiene como alcance la creación de un informe detallado que describa y compare los patrones de arquitectura de desarrollo de *software web* más populares en la actualidad, analizando su eficacia, eficiencia y escalabilidad en diferentes tipos de proyectos. El informe también proporcionará una base para los desarrolladores de *software web* en la elección del patrón de arquitectura más adecuado para sus proyectos, considerando los factores críticos para su implementación.

CAPÍTULO II: FUNDAMENTACIÓN TEÓRICA

3. Marco Teórico

3.1. Metodología aplicada

"La metodología aplicada se refiere al conjunto de técnicas, herramientas y enfoques metodológicos que se seleccionan y adaptan para llevar a cabo un proyecto de investigación específico, en función de los objetivos de la investigación y el tipo de datos que se recopilan" (Creswell, 2018).

Se seleccionó a la revisión sistémica de literatura y encuestas como técnicas en base a la metodología aplicada. La revisión sistemática de literatura ofrece una manera rigurosa de recopilar la información existente sobre el tema de estudio. Por otro lado, las encuestas proporcionaron una forma directa de obtener datos de los expertos en el área.

3.2. Investigación cualitativa

Según Cresswell:

> El enfoque de la investigación cualitativa se centra en comprender la experiencia humana, ya que se basa en la recopilación de datos que se relacionan con la interpretación y la comprensión del significado de la vida cotidiana de las personas, sus interacciones sociales y sus perspectivas únicas. (Cresswell, 2018)

Permite una exploración detallada de las experiencias y perspectivas de los participantes con relación a la utilización de patrones de arquitectura en el desarrollo de *software web*. A través de técnicas como entrevistas y observación participante, se puede obtener información descriptiva sobre cómo los desarrolladores perciben y utilizan estos patrones en su trabajo diario.

Se optó por la aplicación de la investigación cualitativa debido a que esta permite una exploración detallada sobre temas complejos que son difíciles de abordar con métodos cuantitativos tales como las características de los patrones de arquitectura, ya que estos datos no pueden ser estadísticamente mesurados.

3.3. Prospectiva de la información

La prospectiva de la información es un enfoque estratégico qué permite analizar y planificar el futuro en el ámbito de la información y la comunicación. Implica la aplicación de diversos métodos que permiten identificar tendencias emergentes, tecnologías y cambios sociales que pueden afectar el uso y el valor de la información.

Según González (2012), la prospectiva se enfoca en "describir escenarios futuros posibles y preferibles a partir del análisis de las tendencias del presente, en función de las necesidades e intereses de una organización o de un sector de la sociedad".

Mediante la prospectiva de la información, se pueden realizar análisis y estudios que permitan tomar decisiones estratégicas enfocadas a la gestión de la información y la tecnología.

Una de las técnicas de la prospectiva de la información es el método Delphi, la aplicación de este método podría generar información valiosa para el proyecto, debido al análisis estructurado y su capacidad para recopilar opiniones dentro un grupo de expertos y en base a ello generar conocimiento.

3.3.1. Método Delphi

Según Saaty:

El método Delphi es una técnica de consenso que se utiliza para obtener la opinión de un grupo de expertos sobre un tema particular. Consiste en hacer preguntas repetidas a un

panel de expertos con el objetivo de recopilar y destilar opiniones para producir un pronóstico o estimación confiable e imparcial. (Saaty & Peniwati, 2008)

La aplicación de este método sobre el proyecto ayudará a identificar y priorizar las variables o factores más relevantes a considerar en el análisis. Al solicitar aportes de un grupo diverso de expertos, el método puede ayudar a descubrir variables que de otro modo no se habrían considerado, o aclarar la importancia de variables ya identificadas.

Para aplicar el método Delphi en el proyecto, se empezará por la selección un grupo de expertos en desarrollo de *software* a quienes se encuestará. Posteriormente, se realizará el diseño y aplicación de la encuesta sobre los expertos, para así, recoger los resultados obtenidos y, en base a estos, se elaborará un resumen que destaque las principales ideas y perspectivas expresadas por los especialistas.

3.4. Desarrollo web

El desarrollo *web* se enfoca en la creación y mantenimiento de sitios y aplicaciones *web* que se ejecutan en un navegador. Según Kaur y Singh (2020), “el desarrollo *web* se ha vuelto cada vez más importante debido al aumento en la demanda de aplicaciones y servicios en línea”.

Contempla desde páginas *web* estáticas hasta la construcción de aplicaciones *web* complejas con múltiples funcionalidades y características interactivas. Implica el uso de diferentes lenguajes de programación, *frameworks* y herramientas para el desarrollo, dependiendo de las necesidades específicas del proyecto.

3.5. Calidad de software

Garousi afirma lo siguiente:

La calidad del *software* se refiere a un conjunto de propiedades o atributos que se utilizan para medir qué tan bien satisface un *software* las necesidades de sus usuarios y clientes.

Algunos de los atributos de calidad más importantes incluyen la funcionalidad, la confiabilidad, la usabilidad, la eficiencia, la seguridad y la mantenibilidad. (Garousi, 2018)

La relevancia de la calidad de *software* afecta de forma directa en la satisfacción y la productividad del usuario final, así como la reputación y los resultados de la organización. “El *software* de mala calidad puede provocar errores, fallas y violaciones de la seguridad, lo que genera pérdida de productividad, pérdida de ingresos y daño a la reputación de la organización.” (Babok, 2015)

"Los costos de corregir los errores aumentan exponencialmente a medida que avanzan las etapas del ciclo de vida del *software*” (Boehm y Basili, 2001). Actualmente, con el gran crecimiento de las aplicaciones de *software* en todos los ámbitos profesionales y cotidianos, la calidad del *software* es una preocupación crítica tanto para las organizaciones como para las personas.

"La calidad del *software* se logra mediante un proceso sistemático y disciplinado de evaluación y mejora, que debe comenzar en las primeras etapas del ciclo de vida del *software* y continuar hasta después del despliegue del producto" (Pressman, 2015). Integrando a la calidad en el producto desde el principio y no de forma tardía, además de incluir procesos tales como la recopilación de requisitos, el diseño, la implementación, las pruebas, el despliegue y el mantenimiento.

3.5.1. Escalabilidad

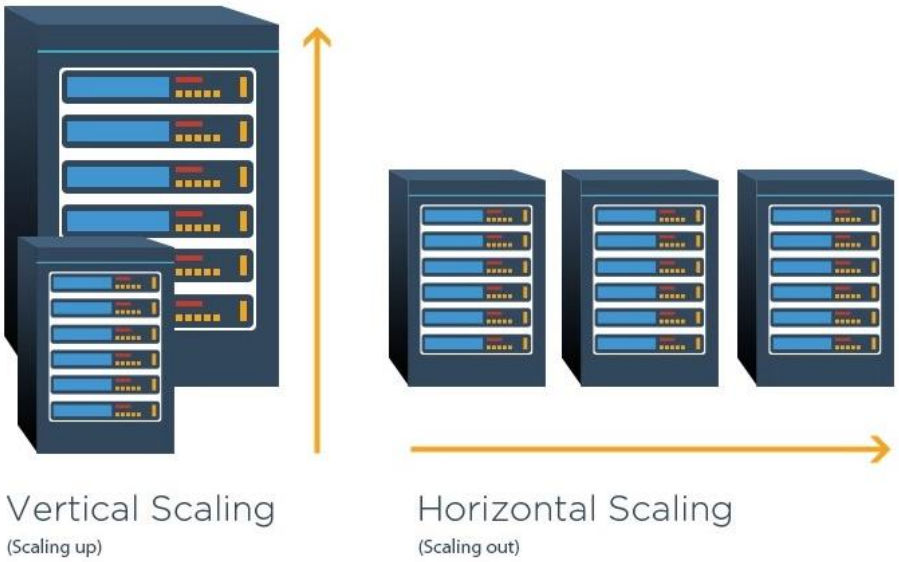
La escalabilidad puede definirse como "la capacidad de un sistema de adaptarse y manejar un incremento en la carga de trabajo mediante la adición de recursos" (Armbrust et al., 2010). Un sistema escalable se ajusta fácilmente al crecimiento de demanda sin necesidad de cambios si gnificativos en su diseño o infraestructura.

Una arquitectura escalable permite que un sistema se adapte fácilmente al aumento del uso, sin necesidad de cambios importantes en el diseño del sistema o en su infraestructura. Esto es importante precisamente para las aplicaciones *web* que suelen experimentar picos repentinos en su tráfico, debido a que garantiza que el sistema pueda manejar el aumento de carga y continuar funcionando de manera eficiente.

Algunas de las técnicas más comunes para lograr la escalabilidad son el escalado horizontal y el escalado vertical. "El escalado horizontal implica agregar más instancias de los mismos componentes, mientras que el escalado vertical implica agregar más recursos a los componentes existentes" (Hoffman, 2014).

Figura 1

Escalamiento vertical y horizontal



Nota. Adaptado de Horizontal Vs. Vertical Scaling: How Do They Compare?, por CloudZero Team, 2023, CloudZero (https://www.cloudzero.com/blog/horizontal-vs-vertical-scaling)

La escalabilidad es un componente fundamental que se encuentra presente en el *software* de alta calidad, dado que este garantiza que el sistema satisfaga las necesidades de los usuarios y de la organización a medida que el aplicativo crece y evoluciona con el tiempo.

3.5.2. Acoplamiento

El acoplamiento es "la medida en que un componente de *software* depende de otros componentes para realizar sus funciones" (Bass et al., 2015). Esta describe la dependencia que un componente presenta sobre otro para realizar sus funciones. Dicho de otra forma, el acoplamiento mide qué tan estrechamente relacionados están dos o más componentes en un sistema.

“Un sistema bien diseñado debe tener componentes que estén lo menos acoplados posible, lo que permite que sean más independientes y puedan ser modificados con menor impacto en el resto del sistema" (Sommerville, 2016). En el caso de realizar cambios sobre un componente que se encuentra estrechamente acoplado entonces los cambios realizados pueden tener un gran impacto en otros componentes, generando la posibilidad de causar consecuencias no deseadas, además de dificultar la modificación y el mantenimiento del sistema.

En conclusión, un bajo acoplamiento permite mejorar la calidad del *software* y mejorar la capacidad de reutilización de los componentes en otros sistemas. Por lo tanto, reducir el acoplamiento sobre los componentes es fundamental para lograr una arquitectura de *software* bien diseñada que sea flexible, escalable y mantenible en el tiempo.

3.5.3. Cohesión

La cohesión se refiere a "cuán estrechamente relacionadas están las funcionalidades dentro de un módulo o componente" (Pressman & Maxim, 2015). La alta cohesión indica que

los elementos dentro de un módulo están estrechamente relacionados y trabajan de forma conjunta para lograr un solo objetivo.

En el diseño de *software*, se suele preferir una alta cohesión, debido a que conduce a sistemas más modulares, fáciles de mantener y escalables. Una arquitectura con alta cohesión suele permitir una mayor reutilización del código y una depuración más sencilla para la aplicación.

Algunas de las técnicas más comunes para lograr una alta cohesión en la arquitectura de *software* abarcan el uso de una convención de nomenclatura clara y consistente, mantener el código relacionado en el mismo módulo y usar niveles apropiados de abstracción para garantizar que cada módulo se centre en una única responsabilidad.

3.5.4. Complejidad

La complejidad se suele asociar a "la cantidad de esfuerzo necesario para comprender, modificar y mantener un sistema de *software*" (Booch, Maksimchu et al., 2007). Suele encontrarse influenciada por varios factores, tales como el número de componentes o módulos, sus interdependencias, el grado de acoplamiento y cohesión entre ellos y el nivel de abstracción en el sistema.

La alta complejidad puede incrementar los tiempos de desarrollo, costos de mantenimiento y un mayor riesgo de errores y fallas. En consecuencia, los arquitectos de *software* suelen administrar y equilibrar cuidadosamente la complejidad de un sistema para garantizar que este sea comprensible, mantenible y escalable.

3.6. Principios SOLID

Según Martin:

> Los principios SOLID son un conjunto de cinco principios de diseño de *software* orientado a objetos que promueven el desarrollo de sistemas flexibles, mantenibles y escalables.

> Cada principio se enfoca en un aspecto particular de la estructura y organización del código, y juntos proporcionan un marco para crear código modular, fácil de entender y mantener. (Martin, 2003)

3.6.1. Principio de responsabilidad simple (S)

"Una clase debe tener solo una razón para cambiar" (Martin, 2003). Este principio se centra en la idea de que cada clase debe enfocarse en desarrollar una sola responsabilidad y no involucrarse en múltiples tareas dentro del sistema, permitiendo mejorar el mantenimiento y escalabilidad.

Este principio permite mejorar la capacidad de mantenimiento y escalabilidad del *software*, ya que genera código más fácil de entender, probar y modificar. Al modificar una clase los cambios solo afectan a una responsabilidad, en lugar de varias. De la misma forma, entender el código por responsabilidades resulta más fácil que entender el sistema como un todo.

3.6.2. Principio abierto-cerrado (O)

"Un módulo debe ser abierto para extensión, pero cerrado para modificación" Martin (2003). Esto hace referencia a que cuando se requiere una nueva funcionalidad, esta debe agregarse creando un código nuevo en lugar de cambiar el código existente.

Para lograr flexibilidad y escalabilidad dentro del sistema este principio utiliza la abstracción y herencia. Según Martin (2003), "permite a un programador agregar nuevas funcionalidades sin cambiar el código existente". Facilitando las pruebas y mantenimiento, y minimizando el riesgo de generar errores en el código base existente.

3.6.3. Principio de sustitución de Liskov (L)

Joshi & Shab establecen lo siguiente:

> El Liskov Substitution Principle (LSP) es un principio fundamental de la programación orientada a objetos que establece que los objetos de una clase derivada deben poder ser

usados en lugar de los objetos de su clase base sin que el comportamiento del programa sea alterado. (Joshi & Shah, 2015).

Dicho de otra forma, un programa que se encuentra diseñado para trabajar con objetos de una clase padre también debe poder trabajar con objetos de las clases hijas, sin que el funcionamiento del programa se altere, esto hace que el código sea genérico, lo cual facilita el mantenimiento y la evolución del sistema.

3.6.4. Principio de segregación de interfaces (I)

El principio de segregación de interfaces es un principio de diseño de *software* que se enfoca en la creación de interfaces de usuario y clases con un conjunto cohesivo de funciones relacionadas, "Una interfaz es un contrato entre el proveedor de la interfaz y el consumidor de la interfaz" (Martin, 2017). Este principio "cada componente del *software* debe tener una sola responsabilidad y encapsular un conjunto cohesivo de funcionalidades relacionadas" (Balaji, 2017).

"Una clase o componente que tiene demasiadas responsabilidades es difícil de entender y modificar, y puede ser una fuente de errores" (Freeman, 2014). Al aplicar este principio, se mejora la claridad y la simplicidad de la estructura del *software*, facilitando el trabajo en equipo y la colaboración.

3.6.5. Principio de inversión de dependencias (D)

"Los módulos de alto nivel no deben depender directamente de los módulos de bajo nivel, sino que deben depender de abstracciones" (Sommerville, 2016). Dicho de otra forma, la implementación debe depender de abstracciones, reduciendo el acoplamiento entre los diferentes módulos.

En palabras de Martin (2017), "la inversión de dependencias es la clave para desacoplar los componentes del sistema". Al depender de abstracciones en lugar de detalles de

implementación, se permite la creación de un código más flexible y adaptable a futuros cambios.

3.7. Arquitectura de software

La arquitectura de *software* abarca los componentes, propiedades, interacciones y relaciones del sistema. Según Shaw y Garlan (1996), la arquitectura de *software* "se preocupa por las propiedades globales del sistema: la forma en que se estructura y se compone, cómo se comporta, cómo se mantiene a lo largo del tiempo y cómo se adapta a los cambios en el entorno"

Según Sommerville:

> La arquitectura del *software* es un modelo de alto nivel que describe la estructura del *software* en términos de sus componentes, módulos y sus interrelaciones. El modelo también describe la forma en que los componentes del *software* interactúan entre sí y con el usuario final. (Sommerville, 2016)

La arquitectura de *software* incluye elementos tales como componentes, módulos, capas, patrones de diseño, interfaces y protocolos de comunicación, que permiten a los desarrolladores comprender cómo estos se integran y relacionan para brindar funcionalidad a los usuarios.

"La arquitectura de *software* se ha convertido en una parte fundamental del desarrollo de *software*, ya que puede garantizar que los requisitos de calidad, rendimiento y escalabilidad se cumplan antes de comenzar el desarrollo" (Bass et al., 2015). Así mismo, una buena arquitectura de *software* puede facilitar la colaboración entre los equipos de desarrollo y promover la reutilización de componentes.

3.7.1. Arquitectura cliente-servidor

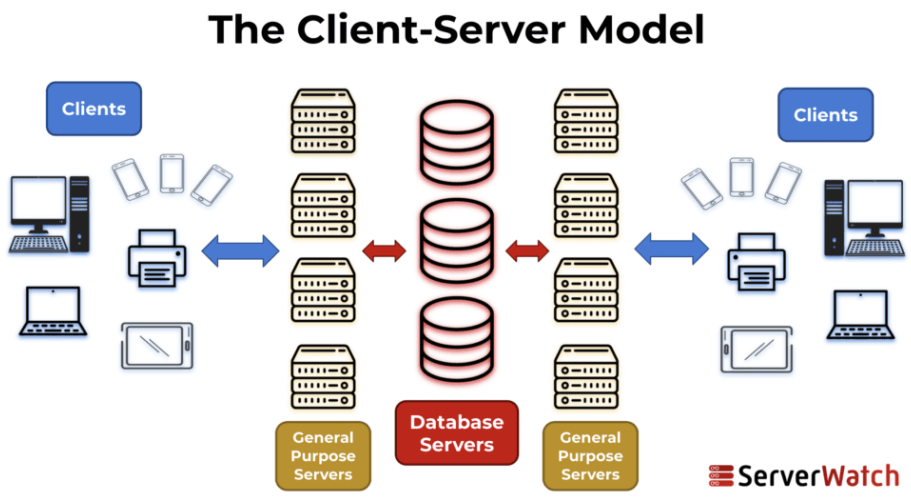
Es un estilo arquitectónico que divide la carga de trabajo entre dos entidades separadas, un cliente y un servidor, "la arquitectura cliente-servidor permite que las máquinas cliente y servidor provengan de diferentes proveedores y requieran diferentes recursos de hardware y *software*" (Tanenbaum & Steen, 2017).

Según Bass et al. (2015), "en una arquitectura cliente-servidor, el cliente es una interfaz de usuario que solicita servicios o datos del servidor. El servidor procesa las solicitudes del cliente y devuelve la información necesaria".

Esta arquitectura presenta una distribución más eficiente, ya que el servidor maneja tareas de procesamiento, generalmente con grandes cargas de trabajo mientras que el cliente es responsable de manejar la interfaz de usuario y la presentación de la información. Los equipos cliente y servidor normalmente requieren diferentes recursos de *hardware* y *software*.

Figura 2

Arquitectura cliente-servidor



Nota. Adaptado de What Is a Client-Server Model? A Guide to Client-Server Architecture, por Sam Ingalls, 2023, ServerWatch (https://www.serverwatch.com/guides/client-server-model/)

3.7.2. Arquitectura en capas

Según Sommerville:

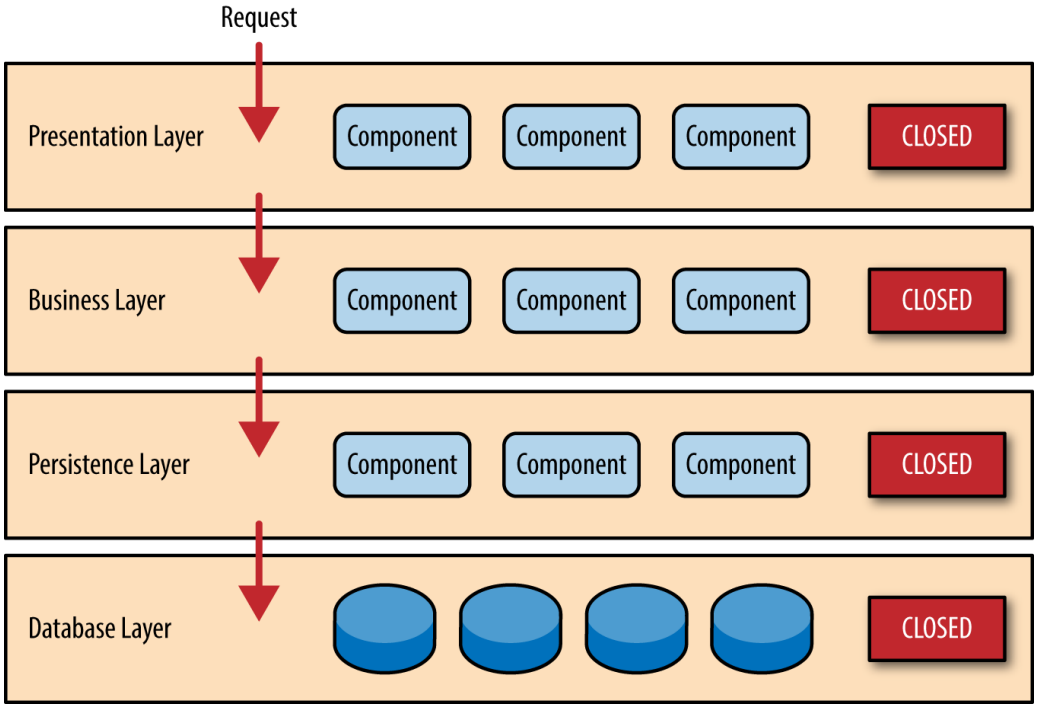
La arquitectura en capas es una técnica de diseño que se utiliza para separar las preocupaciones y la funcionalidad de un sistema de *software* en diferentes niveles lógicos. Esto proporciona una mayor claridad y estructura al sistema, permitiendo una mayor facilidad de gestión y mantenimiento en el tiempo. (Sommerville, 2016)

El modelo OSI (Open Systems Interconnection) en redes de computadoras es un ejemplo para la arquitectura en capas, divide la funcionalidad de la comunicación en red en siete capas. Cada capa se encarga de un aspecto particular de la comunicación, desde el transporte de bits a través del medio físico hasta la gestión de sesiones y la presentación de datos para las aplicaciones de usuario.

La arquitectura en capas resulta útil en sistemas complejos y distribuidos, donde se busca mantener una estructura clara y organizada para facilitar la comprensión, evolución y mantenimiento del sistema. Así mismo, es adecuada en sistemas que requieren una mayor flexibilidad y adaptabilidad, debido a que capa puede modificarse, ampliarse o reemplazarse de forma independiente sin afectar a las demás capas del sistema.

Figura 3

Capas cerradas y solicitud de acceso



Nota. Adaptado de Software Architecture Patterns (p.3), por Mark Richards, 2015, O'Reilly Media.

Características principales de la arquitectura en capas:

- Facilita la separación de responsabilidades y la organización del sistema, mejorando la comprensión y el mantenimiento de la estructura en su conjunto.
- Promueve la reutilización de componentes y la posibilidad de sustituir o modificar capas específicas sin afectar al resto del sistema.
- Mejora la flexibilidad y adaptabilidad a cambios en los requisitos del negocio o en las tecnologías subyacentes, permitiendo una evolución más ágil del sistema.
- Es especialmente adecuada para sistemas complejos y distribuidos que requieren una estructura clara y bien organizada para su correcto funcionamiento y mantenimiento.

3.7.3. Arquitectura orientada a servicios

Según Melzer:

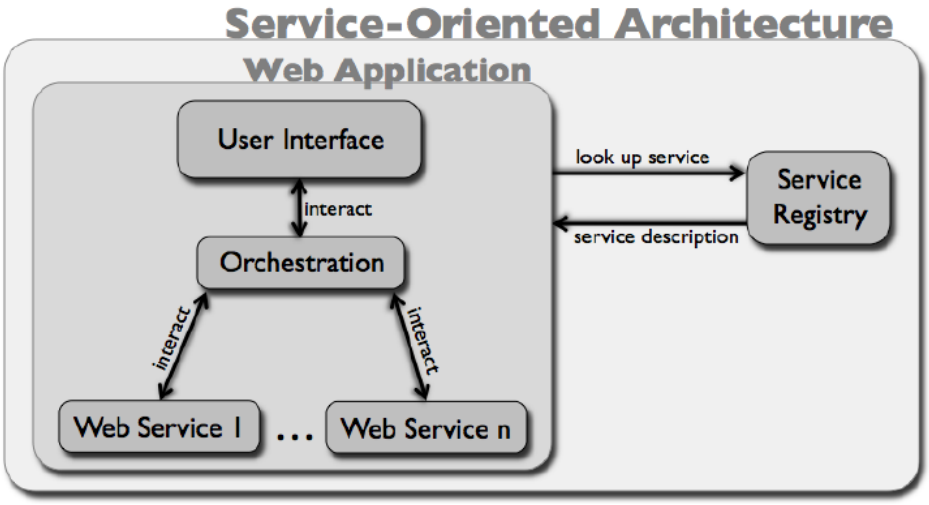
> Una arquitectura orientada a servicios es una arquitectura de sistema, que representa métodos o aplicaciones diversas, diferentes y posiblemente incompatibles como servicios reutilizables y de acceso público y, por lo tanto, ofrece un uso y una reutilización independientes de la plataforma y el idioma. (Melzer, 2010)

Se fundamenta en que los distintos servicios pueden ser reutilizados y combinados para crear aplicaciones complejas y funcionales. Los servicios pueden ser desarrollados bajo distintos lenguajes de programación y *frameworks*, permitiendo una mayor diversidad y adaptabilidad en el proceso de desarrollo.

Se puede encontrar implementada especialmente en entornos empresariales y de integración, donde los distintos sistemas y aplicaciones deben comunicarse e interoperar entre sí. La implementación de esta arquitectura permite la creación de soluciones más flexibles y escalables, que permiten adaptarse a las cambiantes necesidades del negocio y a la evolución de las tecnologías.

Figura 4

Arquitectura orientada a servicios



Nota. Adaptado de Service-oriented architectures: from desktop tools to web services and web applications (p. 73) por Henrich et al., 2010, ResearchGate.

Características principales de la arquitectura orientada a servicios:

- Facilita la integración y la interoperabilidad entre distintos sistemas y aplicaciones.
- Fomenta la reutilización y el desacoplamiento de componentes, facilitando la adaptación a cambios en los requisitos del negocio.
- Mejora la escalabilidad y el rendimiento, debido a que los servicios son implementados de forma independiente y pueden distribuirse en diferentes servidores o infraestructuras, mediante distintos lenguajes de programación o *frameworks*.
- Permite la modificación de un servicio sin interrumpir el funcionamiento general del sistema, facilitando el mantenimiento y actualización de la aplicación.

3.7.4. Arquitectura de microservicios

La arquitectura de microservicios "consiste en la creación de aplicaciones a partir de una serie de servicios pequeños, independientes y en ejecución libre que se comunican entre sí a través de *APIs* ligeras y bien definidas" (Newman, 2015).

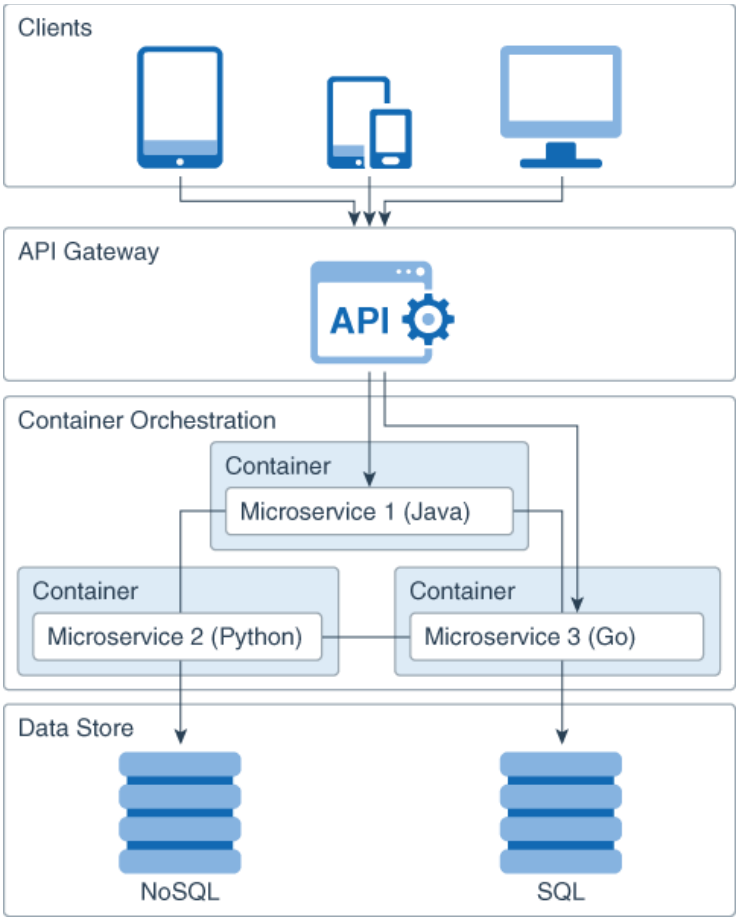
Según Newman (2015), "un microservicio es un servicio pequeño, autónomo y enfocado en una tarea única que se ejecuta en su propio proceso y se comunica con otros servicios a través de una interfaz bien definida". Los microservicios se desarrollan e implementan de forma independiente, lo cual permite responder eficientemente a los cambios en los requerimientos del negocio.

Una importante ventaja en el ámbito del desarrollo es que esta arquitectura permite que el equipo de desarrollo trabaje de forma más autónoma, haciendo que cada equipo pueda ser responsable sobre un conjunto específico de microservicios, lo cual facilita la colaboración y mejora la entrega continua.

Se suele encontrar implementada en aplicaciones distribuidas y en la nube, donde la escalabilidad y la resiliencia son cruciales para el éxito del proyecto. Esta suele facilitar la implementación de prácticas de desarrollo ágil y *DevOps*, mejorando la eficiencia y la velocidad de entrega.

Figura 5

Arquitectura de microservicios



Nota. Adaptado de Aprender acerca de la arquitectura de microservicios, s.f., Oracle (https://docs.oracle.com/es/solutions/learn-architect-microservice/index.html#GUID-1A9ECC2B-F7E6-430F-8EDA-911712467953)

Características principales de la arquitectura de microservicios:

• Facilita la organización y la colaboración entre equipos de desarrollo, al momento de distribuir la carga de trabajo.

• Mejora la capacidad de respuesta a cambios en los requisitos del negocio, debido a que cada microservicio puede actualizarse o reemplazarse sin afectar a todo el sistema.

• Permite una mayor escalabilidad y distribución de la carga de trabajo entre diferentes servidores o infraestructuras.

• Facilita la implementación de prácticas de desarrollo ágil y *DevOps*.

3.7.5. Arquitectura basada en eventos

"La arquitectura basada en eventos es una arquitectura de *software* que se enfoca en la generación, detección, y reacción a eventos y en la propagación de eventos a través de un sistema" (Newman, 2015).

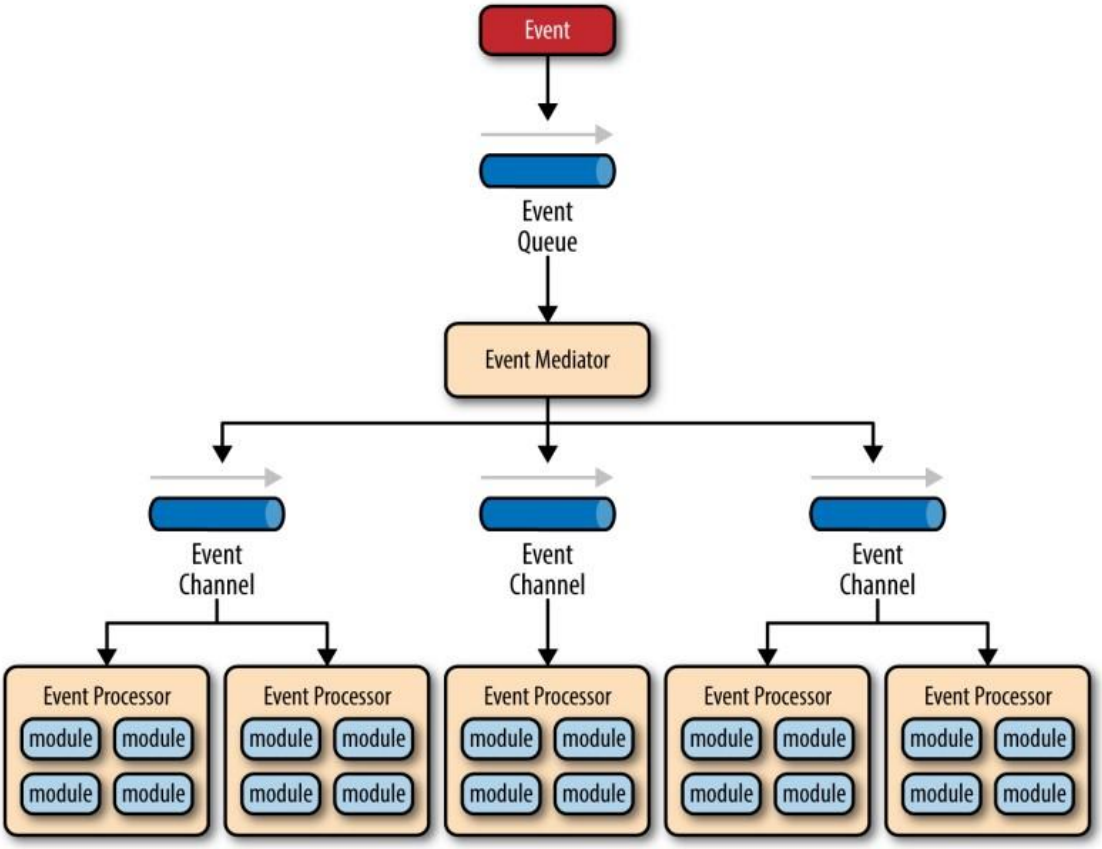
“Es un patrón de arquitectura asíncrono distribuido que se utiliza para la creación de aplicaciones altamente escalable” (Richards, 2017). La arquitectura basada en eventos se conforma de componentes de procesamiento de eventos de propósito único altamente desacoplados que reciben y procesan eventos de manera asíncrona.

Los componentes no se encuentran directamente acoplados y no necesitan conocer la existencia de otros componentes en el sistema. Cada componente se enfoca en producir o consumir eventos a través de canales de eventos compartidos, permitiendo una mayor flexibilidad y escalabilidad en el diseño del sistema.

Algunos de sus principales beneficios son la capacidad para integrar diferentes sistemas y tecnologías, la mejora en la resiliencia del sistema, la reducción de la complejidad y la capacidad para diseñar sistemas más flexibles, permitiendo adaptarse a los cambios en los requerimientos del negocio.

Figura 6

Arquitectura basada en eventos



Nota. Adaptado de Software Architecture Patterns (p.12), por Mark Richards, 2015, O'Reilly media.

3.7.6. Arquitectura de microkernel

Tanenbaum afirma lo siguiente:

> La arquitectura de *microkernel* se enfoca en proporcionar una base mínima de servicios esenciales, tales como la comunicación entre procesos, la gestión de memoria, y la planificación de procesos, mientras que otros servicios, como los sistemas de archivos y los controladores de dispositivos, se implementan como procesos en el espacio de usuario. (Tanenbaum, 2017)

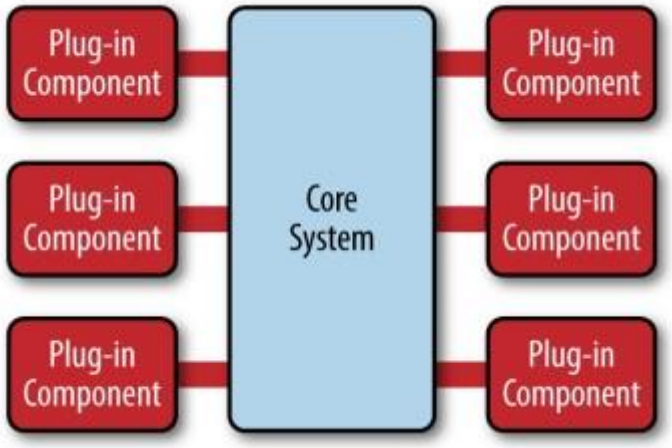
“El patrón de arquitectura de microkernel le permite agregar funciones de aplicación adicionales como complementos a la aplicación principal, proporcionando extensibilidad y separación de funciones.” (Richards, 2017).

La arquitectura de *microkernel* ofrece diversas ventajas en el desarrollo *web*, especialmente en la implementación de aplicaciones basadas en productos. Una de las principales ventajas radica en la capacidad de crear aplicaciones empaquetadas disponibles para su descarga en versiones.

Su enfoque facilita la modularidad y extensibilidad de las aplicaciones, lo que permite una fácil actualización y mantenimiento de cada componente individual. De esta manera, las empresas pueden adaptar rápidamente sus aplicaciones a las necesidades cambiantes del mercado y ofrecer una experiencia de usuario mejorada.

Figura 7

Arquitectura de *microkernel*



Nota. Adaptado de *Software* Architecture Patterns (p. 22), por Mark Richards, 2015, O'Reilly media.

CAPÍTULO III: METODOLOGÍA

4. Metodología de desarrollo del plan de tesis

4.1. Uso metodología aplicada

En el presente proyecto, se utilizaron dos técnicas principales en base a la metodología aplicada que permitieron que los resultados obtenidos sean precisos y confiables, los cuáles son la revisión sistemática de literatura y las encuestas.

La revisión sistemática de literatura se aplica en la recopilación de datos mediante la búsqueda exhaustiva de libros y artículos seleccionados. La cadena de búsqueda incluyó términos como "*software architecture patterns*", "escalabilidad en arquitectura *web*", "*comparative análisis*", "*web application architecture*", *"web development*" y "*best architecture practices*".

Se seleccionaron 12 artículos y 7 libros para la realización del trabajo, recopilados en fuentes como bases de datos académicas tales como IEEE Xplore, ACM Digital Library y ScienceDirect, además de motores de búsqueda como Google Scholar y Bing. Aunque algunos de los libros se encuentran en sus versiones digitales todavía se clasifican como libros y no artículos.

El libro *Software Architecture for Developers (Brown, 2017)* permitió obtener una comprensión sobre los fundamentos de los patrones de arquitectura y sus conceptos básicos. Los libros *Software Architecture Patterns* (Richards, 2017) y *Clean Architecture (Martin, 2017)* proporcionan una comprensión sólida sobre los patrones y principios de arquitectura de *software*, en estos libros se detallan varias características de los patrones de arquitectura utilizados para el análisis.

El libro *Building microservices (Newman, 2015)* proporcionó una comprensión detallada de cómo diseñar aplicaciones con microservicios, lo cual es importante para el desarrollo de aplicaciones *web* modernas y escalables. El libro Ingeniería del *software*: un enfoque práctico, proporciona una introducción general a la ingeniería de *software*, incluyendo la arquitectura de *software*.

Por último, *Software architecture in practice* (Bass et al., 2015) y *Scalability rules: Principles for scaling web sites* (Hoffman, 2014) son libros importantes para comprender cómo la arquitectura de *software* se aplica en la práctica y cómo se pueden escalar aplicaciones *web* para manejar cargas de trabajo más grandes. En conjunto, estos libros permiten obtener una base sólida de la arquitectura de *software* necesaria para realizar el análisis comparativo.

A continuación, se detallan los sitios *web* que se utilizaron para identificar los patrones de arquitectura más populares. TechBeacon es un sitio *web* que se enfoca en temas de tecnología y desarrollo de *software*, con un equipo de expertos en la materia. Apiumhub, por su parte, es una empresa especializada en consultoría de tecnología y desarrollo de *software*.

MilesWeb es un proveedor de servicios de *hosting* y tiene una sección de blogs que ofrece contenido sobre diferentes aspectos tecnológicos. Finalmente, Decipher Zone *Softwares* es una compañía de desarrollo de *software* que también tiene una página que cubre temas relacionados con la arquitectura de *software*.

Todas estas fuentes tienen experiencia y conocimiento en el campo de la tecnología y el desarrollo de *software*, lo que las hace confiables para identificar los patrones de arquitectura más importante. En base a ellas, se identificaron a los patrones de arquitectura más utilizados en los últimos diez años, y que se utilizarán para el análisis comparativo: arquitectura en n capas, arquitectura cliente-servidor, arquitectura basada en eventos, arquitectura de *microkernel*, arquitectura orientada a servicios y arquitectura de microservicios.

Se escogió tomar un periodo de diez años debido a que un ciclo de desarrollo suele abarcar este rango, otro factor para tomar en consideración es cuenta es la amplia información sobre proyectos exitosos y maduros, además de que se pueden evidenciar las tendencias de los patrones de arquitectura de forma clara.

4.2. Aplicación del Método Delphi

Las encuestas son útiles para recopilar información de valor sobre la percepción de los desarrolladores de *software* acerca de los patrones de arquitectura identificados, que nos permitirán comprender de mejor manera las necesidades y preferencias de los desarrolladores de *software*, el método Delphi permitió realizar el diseño de las encuestas y la recopilación de resultados.

Se escogió a Google Forms como herramienta para el desarrollo de las encuestas, dado que posee grandes ventajas, entre las cuales se incluyen una personalización amplia, facilidad de uso, accesibilidad y presentación de resultados mediante gráficos estadísticos. Se presenta el diseño de las encuestas y sus resultados en la sección de anexos.

4.2.1. Selección de expertos

Se optó por seleccionar a un grupo de 10 expertos en el desarrollo de *software* debido a que este es el número de personas recomendable para garantizar que se obtengan opiniones representativas y diversas según el método Delphi. El grupo de expertos seleccionado posee una amplia experiencia en el desarrollo de aplicaciones *web*, habiendo implementado diversos patrones de arquitectura en distintos tipos de proyectos. Los miembros del grupo desempeñan cargos de Lead Engineer, Senior Engineer, QA Lead y Auditor informático, dentro de las empresas Verndale, Bayer y Banco del Pichincha.

4.2.2. Primera ronda de encuestas

La encuesta permitió a los expertos ofrecer sus opiniones sobre los factores críticos en la implementación de cada patrón de arquitectura tales como el modularidad y la separación de responsabilidades en el caso de la arquitectura en n capas o el rendimiento y balanceo de carga en la arquitectura Cliente-Servidor.

La encuesta consta de 6 preguntas de selección múltiple en las que se solicitó a los expertos calificar la importancia de cada factor descrito mediante una escala de Likert de 1 a 5, de la siguiente manera:

1: Nada importante

2: Poco importante

3: Algo importante

4: Bastante importante

5: Extremadamente importante

Así mismo, se incluyó una pregunta abierta, en las que los expertos podían sugerir factores críticos adicionales a los propuestos, permitiendo recabar más información.

4.2.3. Resumen de las respuestas primera encuesta

La primera encuesta proporcionó una visión general sobre las opiniones de los expertos con respecto a los factores críticos en la implementación de los patrones de arquitectura de *software web*, según sus experiencias y opiniones.

Aunque se identificaron áreas de consenso entre los expertos como en el caso del modularidad en la arquitectura en n capas, o la integración y reutilización de servicios en la arquitectura orientada a servicios, también se evidenciaron algunas discrepancias, tales como

la latencia en la arquitectura basada en eventos o el balanceo de carga en la arquitectura cliente-servidor.

4.2.4. Segunda ronda de encuestas

Los resultados de la primera encuesta sirvieron como base para la segunda ronda, según lo que promulga el método Delphi, se incluyó la puntuación media de las respuestas obtenidas en la primera encuesta sobre cada factor crítico, además de aumentar los factores extra sugeridos por los encuestados.

La encuesta se conformó de 6 preguntas cerradas, buscando conseguir un mayor consenso entre los expertos y obtener una comprensión más refinada de los factores críticos en la implementación de cada patrón de arquitectura, permitiéndoles a los expertos reevaluar sus respuestas.

4.2.5. Resumen de la segunda ronda

Los resultados de la segunda encuesta evidencian un alto nivel de consenso entre los expertos, permitiendo identificar claramente los factores críticos que deben ser considerados al implementar cada patrón de arquitectura en el desarrollo *web* de arquitectura en el desarrollo de *software web*.

4.2.6. Resultados y consenso

De los resultados obtenidos en la segunda encuesta se tomaron en consideración los factores críticos que obtuvieron una puntuación entre 4 y 5 en la calificación de su importancia, debido a que, en base a nuestra escala de Likert, una calificación entre ese rango se considera bastante importante.

A continuación, se detallan los factores críticos para la implementación de cada patrón de arquitectura en base a los resultados de la encuesta:

Arquitectura en N capas

- Modularidad
- Separación de responsabilidades
- Comunicación entre capas
- Facilidad de mantenimiento

Arquitectura cliente-servidor

- Rendimiento
- Balanceo de carga
- Seguridad
- Tolerancia a fallos

Arquitectura Basada en Eventos

- Manejo de eventos
- Escalabilidad
- Desacoplamiento

Arquitectura de Microkernel

- Abstracción
- Gestión de *plugins*
- Documentación y soporte

Arquitectura Orientada a Servicios

- Integración
- Reutilización
- Interoperabilidad

Arquitectura de Microservicios

- Desacoplamiento
- Escalabilidad
- Monitoreo
- Resiliencia

4.3. Aplicación investigación cualitativa

Se definieron varias categorías para el desarrollo de la investigación, con el fin de describir el nivel de acoplamiento, cohesión y complejidad de cada patrón de arquitectura. De la misma manera, para evaluar la idoneidad de cada patrón de arquitectura en base a distintos tipos de proyecto, además de su escalabilidad.

Las categorías creadas para describir el nivel de acoplamiento, cohesión y complejidad de cada patrón de arquitectura son: muy bajo, bajo, moderado, alto y muy alto. Se basaron en la clasificación asignada a los patrones de arquitectura dentro del libro *Software Architecture Patterns*. A continuación, se define a qué se refiere cada una de ellas.

Acoplamiento:

- **Muy bajo:** Los componentes están altamente desacoplados y muestran una dependencia insignificante entre sí, facilitando la implementación y el mantenimiento del sistema, haciéndolo modular y escalable.
- **Bajo:** Los componentes están acoplados, sin embargo, la dependencia entre ellos es limitada. Este nivel de acoplamiento permite que el patrón sea relativamente fácil de implementar y mantener en cierta medida.
- **Moderado**: Esto significa que los componentes están acoplados, pero no de forma completa y que se requiere un esfuerzo adicional para implementar y mantener el patrón.

- **Alto**: Los componentes están altamente acoplados y la dependencia entre ellos es significativa, implicando en que la implementación y el mantenimiento del patrón sean complejos y que se requiera un esfuerzo y recursos significativos.
- **Muy alto:** Los componentes están altamente acoplados y la dependencia entre ellos es crítica, esto hace que la implementación y el mantenimiento del patrón sean muy complejos, lo que resulta en una cantidad considerable de recursos y tiempo para hacerlo correctamente.

Cohesión:

- **Muy bajo:** Los componentes no están centrados y enfocados en tareas específicas, dificultando la comprensión y el mantenimiento del sistema.
- **Bajo:** Se refiere a que los componentes se encuentran con un enfoque poco claro en las responsabilidades que debe desempeñar cada componente, lo que puede presentar dificultades para el mantenimiento del sistema.
- **Moderado:** Las responsabilidades de los componentes se definen de forma coherente pero no completamente definidas, dificultando el mantenimiento del sistema.
- **Alto**: Las responsabilidades de los componentes se encuentran altamente definidas, resultando en una aplicación fácil de mantener, con escalabilidad alta y baja complejidad.
- **Muy alto:** Los componentes se enfocan directamente en tareas específicas, sin lugar a ambigüedades, manteniendo una responsabilidad coherente y resultando en un sistema muy fácil de mantener.

Complejidad:

- **Muy bajo**: No requiere de conocimientos técnicos avanzados para la implementación del patrón de arquitectura.
- **Bajo:** Puede requerir una cantidad manejable de trabajo adicional para entender, implementar y mantener la aplicación.
- **Moderado**: Requiere una cantidad de trabajo significativa para entender la estructura de la aplicación.
- **Alto:** Genera una dificultad grande para entender, implementar y mantener, requiriendo una cantidad de conocimientos técnicos avanzados para su aplicación.
- **Muy alto**: Resulta muy difícil de entender, implementar y mantener, por lo que involucra vastos recursos y conocimientos técnicos para su aplicación.

Las categorías creadas para describir la idoneidad de los patrones de arquitectura en base al tipo de proyecto son: patrón no adecuado, presenta limitaciones y patrón adecuado. A continuación, se define a qué se refiere cada una de ellas.

- **Patrón adecuado:** se refiere a que el patrón de arquitectura seleccionado es el más apropiado para la situación o caso en particular, cumpliendo con los requisitos y necesidades específicas del proyecto.
- **Presenta limitaciones:** se refiere a que, el patrón de arquitectura se puede aplicar al proyecto, sin embargo, puede incluir ciertos inconvenientes como la falta de flexibilidad o la necesidad de adaptar el patrón para cumplir con ciertos requisitos específicos.
- **Patrón no adecuado:** se refiere a que el patrón de arquitectura no cumple con los requisitos específicos, por lo que, se deben explorar otras opciones de patrones de arquitectura más adecuados.

Por último, las categorías creadas para describir la escalabilidad de los patrones de arquitectura en base al tipo de proyecto son: buena escalabilidad, escalabilidad moderada, baja escalabilidad.

- **Buena escalabilidad:** El patrón de arquitectura se adapta fácilmente a proyectos de diferentes tamaños y complejidades.
- **Escalabilidad moderada:** A pesar de que el patrón puede adaptarse de cierta forma a un incremento de carga, puede presentar limitaciones sobre la adaptabilidad a proyectos más grandes y complejos.
- **Baja escalabilidad:** Presenta una capacidad muy limitada de adaptación a proyectos más grandes, afectando negativamente a su rendimiento.

5. Desarrollo

5.1. Análisis comparativo

Se analiza el acoplamiento, cohesión y complejidad de cada patrón de arquitectura identificado en base a las categorías creadas anteriormente entre muy bajo, bajo, moderado, alto y muy alto.

Arquitectura en N capas

- **Acoplamiento**: El acoplamiento en la arquitectura en N capas es moderado, ya que, aunque las capas tienen dependencias bien definidas entre sí, estas dependencias se limitan a las capas adyacentes. Por lo tanto, cualquier cambio en una capa podría afectar a las capas directamente relacionadas, pero no necesariamente a toda la aplicación.
- **Cohesión**: La cohesión es alta debido a la división de responsabilidades en capas específicas, como la capa de presentación, la capa de lógica de negocio y la capa de acceso a datos. Esta separación permite que cada capa tenga responsabilidades bien definidas y únicas.
- **Complejidad**: La complejidad moderada proviene de la necesidad de administrar y coordinar las interacciones entre las diferentes capas, así como de garantizar la consistencia y la integridad de los datos a medida que fluyen a través de las capas.

Arquitectura Cliente-Servidor

- **Acoplamiento**: El acoplamiento es moderado dado que el cliente y el servidor están interconectados y dependen el uno del otro para funcionar. Sin embargo, las responsabilidades están divididas entre el cliente y el servidor, lo cual limita las dependencias.

- **Cohesión**: La cohesión es baja a razón de la separación de responsabilidades entre el cliente y el servidor, ya que las funcionalidades pueden estar distribuidas en ambos extremos.
- **Complejidad**: La complejidad es baja debido a su simplicidad inherente, ya que contiene únicamente dos componentes principales (cliente y servidor), lo cual resulta fácil de entender y manejar.

Arquitectura basada en eventos

- **Acoplamiento**: El acoplamiento en la arquitectura basada en eventos es bajo no solo debido a que los componentes interactúan a través de eventos, generando una comunicación indirecta y asíncrona entre ellos, sino también porque los componentes no necesitan conocer detalles de implementación de otros componentes, reduciendo las dependencias directas.
- **Cohesión**: La cohesión en la arquitectura basada en eventos es alta, ya que los componentes se encuentran bien organizados y divididos en base a sus responsabilidades.
- **Complejidad**: La complejidad en la arquitectura basada en eventos es moderada debido a que la coordinación y el manejo de los eventos puede ser complejo, especialmente en sistemas de gran escala.

Arquitectura de Microkernel

- **Acoplamiento**: Se identificó un bajo acoplamiento debido a la separación de las funcionalidades en *plugins* y un núcleo pequeño. Los *plugins* son independientes entre sí y se comunican con el núcleo mediante interfaces bien definidas, lo que minimiza las dependencias directas entre ellos.

- **Cohesión**: La cohesión es alta en vista de que cada *plugin* tiene una funcionalidad bien definida y separada del núcleo, lo cual permite una organización clara de los componentes y facilita el mantenimiento y la extensibilidad del sistema.
- **Complejidad**: La alta complejidad en la arquitectura de *microkernel* proviene de la interacción entre el núcleo y los múltiples *plugins*, así como de la necesidad de coordinar y gestionar estos *plugins*. Desarrollar nuevos *plugins* puede requerir un conocimiento profundo del núcleo y las interfaces, generando una mayor complejidad.

Arquitectura Orientada a Servicios (SOA)

- **Acoplamiento**: El acoplamiento es moderado, ya que, aunque los servicios están diseñados para ser independientes, pueden existir dependencias entre ellos a través de la comunicación e interacción entre servicios, así mismo, la reutilización de servicios puede llevar a dependencias entre ellos.
- **Cohesión**: La cohesión es alta, debido a que cada servicio tiene una responsabilidad específica y bien definida. Los servicios se crean para realizar tareas específicas, lo que permite una separación clara de responsabilidades y una organización más sencilla.
- **Complejidad**: La complejidad es alta, esto en base a la gestión de servicios y la comunicación entre ellos, ya que, el descubrimiento de servicios, la orquestación y la gestión de políticas de seguridad, entre otros aspectos, aumentan la complejidad del sistema.

Arquitectura de Microservicios

- **Acoplamiento**: El acoplamiento es bajo debido a que cada microservicio es independiente y se comunica a través de *APIs*. Permitiendo que los microservicios evolucionen y cambien sin afectar a otros microservicios, lo que reduce las dependencias directas.

- **Cohesión**: La cohesión es muy alta debido a su enfoque en servicios pequeños y especializados, cada uno de ellos tiene una responsabilidad muy específica y bien definida. Facilitando la organización y el mantenimiento del sistema, ya que cada microservicio encapsula una parte específica de la lógica de negocio.
- **Complejidad**: La complejidad es muy alta, a razón de la gran cantidad de componentes independientes y de la necesidad de coordinar la comunicación y la orquestación entre ellos. Otros factores que pueden contribuir a la complejidad son la gestión de la infraestructura, el monitoreo y la resiliencia.

La siguiente tabla da un vistazo general al nivel de acoplamiento, cohesión y complejidad encontrados en cada patrón de arquitectura.

Tabla 1

Comparación general de patrones de arquitectura

Patrón de Arquitectura	Acoplamiento	Cohesión	Complejidad
Arquitectura en N capas	Moderado	Alta	Moderada
Arquitectura Cliente-Servidor	Moderado	Baja	Baja
Arquitectura Basada en Eventos	Bajo	Alta	Moderada
Arquitectura de Microkernel	Bajo	Alta	Alta
Arquitectura Orientada a Servicios	Moderado	Alta	Alta
Arquitectura de Microservicios	Bajo	Muy Alta	Muy Alta

5.2. Evaluación crítica del análisis comparativo

A pesar de tener claras las características de cada patrón de arquitectura, estos datos por sí solos no proporcionan una base sólida para tomar decisiones informadas sobre qué patrón de arquitectura podría ser más adecuado en función de las características y requisitos específicos de un proyecto.

Se realizará un análisis crítico en base a distintos casos que fueron elegidos para abarcar una variedad de escenarios de desarrollo de *software web* que podrían enfrentar desafíos diferentes y requerir enfoques distintos en términos de patrones de arquitectura. El objetivo del planteamiento de los siguientes casos es cubrir una gama de casos de uso y requisitos, que incluyen:

- Aplicaciones *web* colaborativas: Enfocadas en la interacción y la colaboración en tiempo real entre usuarios.
- Aplicaciones *web* para análisis de datos: Enfocadas en el procesamiento y análisis de datos a gran escala y en la presentación de resultados en diferentes formatos.
- Redes sociales en tiempo real: Requieren interacción en tiempo real entre usuarios y una alta capacidad de respuesta a eventos.
- Análisis y visualización de datos: Enfocadas en el procesamiento intensivo de datos y la personalización de informes.
- Gestión de proyectos distribuidos: Dirigidas a equipos distribuidos que necesitan colaborar en tiempo real y garantizar alta disponibilidad.
- Comercio electrónico escalable: Orientado a empresas en crecimiento que requieren integración con sistemas externos y gestión de inventario en tiempo real.

Los casos son:

- Caso 1: Desarrollo de una plataforma de comercio electrónico para una empresa mediana con requisitos de escalabilidad y modularidad.
- Caso 2: Desarrollo de un sistema de gestión de contenidos (CMS) para una pequeña empresa con un equipo de desarrollo limitado y un presupuesto ajustado.
- Caso 3: Desarrollo de una página *web* de red social con requisitos de rendimiento en tiempo real y alta interacción entre usuarios.
- Caso 4: Desarrollo de una aplicación *web* de análisis de datos y visualización para una organización que requiere procesamiento intensivo de datos y personalización de informes,
- Caso 5: Desarrollo de una aplicación *web* de gestión de proyectos para un equipo distribuido con requisitos de colaboración en tiempo real y alta disponibilidad.
- Caso 6: Desarrollo de un comercio electrónico escalable para una empresa en crecimiento con requisitos de integración de sistemas externos y gestión de inventario en tiempo real.

Caso 1: Desarrollo de una plataforma de comercio electrónico para una empresa mediana con requisitos de escalabilidad y modularidad.

Arquitectura en N capas:

La arquitectura en N capas sería adecuada debido a su alta cohesión y acoplamiento moderado, permitiendo una estructura modular y escalable, lo que ayuda a manejar el crecimiento en el número de usuarios y la incorporación de nuevas funcionalidades en la plataforma de comercio electrónico.

Arquitectura Cliente-Servidor:

La arquitectura cliente-servidor puede manejar cierto grado de modularidad y tiene un nivel moderado de acoplamiento, sin embargo, podría enfrentar problemas de escalabilidad cuando la plataforma de comercio electrónico crezca y experimente un aumento en la demanda, debido a que el servidor podría convertirse en un cuello de botella, afectando el rendimiento del sistema.

Arquitectura basada en eventos:

Debido a su bajo acoplamiento y alta cohesión esta arquitectura resulta útil en el apartado de notificaciones en tiempo real y actualizaciones de estado dentro de la aplicación. Sin embargo, no sería la opción más adecuada para todo el sistema, ya que podría aumentar la complejidad y dificultar la gestión de eventos en la plataforma.

Arquitectura de Microkernel:

La arquitectura de *microkernel* ofrece un bajo acoplamiento y un nivel moderado de cohesión, lo que le permite destacar en flexibilidad y extensibilidad, sin embargo, su alta complejidad podría dificultar su implementación y mantenimiento en una plataforma de comercio electrónico, por lo que su aplicación no sería la mejor opción en este caso.

Arquitectura Orientada a Servicios (SOA):

La arquitectura SOA puede ser adecuada en este caso, debido a que su bajo acoplamiento y alto nivel de cohesión permite la reutilización de servicios y una mejor organización de componentes. Sin embargo, la orquestación de servicios puede afectar el rendimiento del sistema.

Arquitectura de Microservicios:

La arquitectura de microservicios resulta ser la mejor opción por optar en este caso, debido a que ofrece la escalabilidad y el modularidad necesarias para una plataforma de comercio electrónico en crecimiento gracias a su bajo acoplamiento y alto nivel de cohesión, además de facilitar el despliegue y la actualización de componentes independientes sin afectar al resto del sistema. No obstante, hay que tener en cuenta que el alto nivel de complejidad podría implicar un mayor consumo de recursos.

Arquitecturas recomendadas para el caso 1:

Para el desarrollo de una plataforma de comercio electrónico con requisitos de escalabilidad y modularidad, la arquitectura de microservicios sería la mejor opción, ya que ofrece la flexibilidad y escalabilidad necesarias para manejar un gran crecimiento de la plataforma y satisfacer las necesidades de la empresa.

Sin embargo, también se considera que la arquitectura en n capas y la arquitectura orientada a servicios son opciones adecuadas en este caso. La arquitectura en N capas ofrece una estructura modular y escalable, mientras que la arquitectura orientada a servicios permite la reutilización de servicios y una mejor organización de componentes

Caso 2: Desarrollo de un sistema de gestión de contenidos (CMS) para una pequeña empresa con un equipo de desarrollo limitado y un presupuesto ajustado

Arquitectura en N capas:

La arquitectura en N capas proporciona una estructura modular y escalable, sin embargo, puede ser más complicada de implementar y mantener para un equipo pequeño y un presupuesto ajustado. Por lo que, su complejidad moderada y la infraestructura necesaria podrían ser desafiantes en este contexto.

Arquitectura Cliente-Servidor:

La arquitectura cliente-servidor sería una opción viable para el caso propuesto, ya que proporciona una clara división entre la interfaz de usuario y el procesamiento de datos, es más simple de implementar y mantener que otras arquitecturas, lo cual es adecuado para un equipo pequeño y un presupuesto limitado.

Arquitectura basada en eventos:

Si bien la arquitectura basada en eventos podría ser útil en ciertas partes del CMS, como notificaciones y actualizaciones en tiempo real, no sería la opción más adecuada para todo el sistema debido a la complejidad moderada que presenta y sus dificultades en la gestión de eventos.

Arquitectura de Microkernel:

La arquitectura de *microkernel* no sería la mejor opción en este contexto, ya que su alta complejidad dificulta la implementación y el mantenimiento del CMS para un equipo pequeño y con un presupuesto ajustado.

Arquitectura Orientada a Servicios (SOA):

La arquitectura SOA ofrece ciertos beneficios, como la reutilización de servicios y una mejor organización de componentes, sin embargo, la construcción de servicios presenta una alta complejidad y costos de desarrollo elevados en comparación con otros patrones de arquitectura.

Arquitectura de Microservicios:

La arquitectura de microservicios presenta una complejidad muy alta y puede resultar costosa para un equipo pequeño y un presupuesto ajustado, sin contar que la gestión de la

infraestructura y la coordinación entre microservicios podrían resultar desafiantes en este contexto.

Arquitectura recomendada para el caso 2:

Para el desarrollo de un sistema de gestión de contenidos (CMS) para una pequeña empresa con un equipo de desarrollo limitado y un presupuesto ajustado, la arquitectura cliente-servidor sería la mejor opción, debido a que proporciona una separación clara entre la interfaz de usuario y el procesamiento de datos, sumado a ello, es más fácil de implementar y mantener en comparación con otras arquitecturas más complejas.

Caso 3: Desarrollo de una página *web* de red social con requisitos de rendimiento en tiempo real y alta interacción entre usuarios

Arquitectura en N capas:

La arquitectura en N capas puede no ser la mejor opción para este contexto debido a los posibles problemas de rendimiento al manejar la alta interacción entre usuarios y las necesidades en tiempo real de la página *web*.

Arquitectura Cliente-Servidor:

La arquitectura cliente-servidor no es un patrón adecuado para este caso debido a que la carga en el servidor generaría un cuello de botella, lo que afecta de forma negativa al rendimiento de la página *web* y empeora la experiencia del usuario.

Arquitectura basada en eventos:

La arquitectura basada en eventos sería una opción adecuada, ya que es especialmente útil para manejar interacciones en tiempo real y notificaciones en sistemas de alta concurrencia, permitiendo a la página *web* de red social responder rápidamente a eventos y mantener a los usuarios informados en tiempo real.

Arquitectura de Microkernel:

La arquitectura de *microkernel* no sería la mejor opción en este contexto, ya que su implementación generaría una complejidad innecesaria al no utilizar sus principales ventajas, lo cual afecta no solo a la implementación sino también el mantenimiento de la página *web*.

Arquitectura Orientada a Servicios (SOA):

La arquitectura SOA tiene beneficios en este caso en términos de reutilización de servicios y organización de componentes, sin embargo, la orquestación entre servicios podría generar latencia y afectar el rendimiento en tiempo real de la página *web* de red social.

Arquitectura de Microservicios:

La arquitectura de microservicios se podría considerar una buena opción en este caso, debido a su escalabilidad y modularidad. Su bajo acoplamiento permite la separación de la funcionalidad en microservicios independientes, permitiendo que la aplicación se adapte y escale en base a las necesidades de la página *web*.

Arquitecturas recomendadas para el contexto 3:

En este caso las mejores opciones son la arquitectura basada en eventos y la arquitectura de microservicios, entre sus principales características se presentan: la arquitectura basada en eventos permite una rápida respuesta a eventos y mantener una actualización en tiempo real para los usuarios, mientras que la arquitectura de microservicios por su lado proporcionaría principalmente la escalabilidad necesaria para adaptarse al tráfico de la página *web.*

Caso 4: Desarrollo de una aplicación *web* de análisis de datos y visualización para una organización que requiere procesamiento intensivo de datos y personalización de informes

Arquitectura en N capas:

La arquitectura en N capas sería una opción adecuada en este caso, ya que permite separar la lógica de negocio y la presentación. Sin embargo, podría enfrentar desafíos de rendimiento al manejar grandes volúmenes de datos y operaciones intensivas de procesamiento.

Arquitectura Cliente-Servidor:

La arquitectura cliente-servidor no sería una opción adecuada para este caso, a razón de los problemas potenciales de rendimiento y escalabilidad al manejar grandes volúmenes de datos y operaciones intensivas de procesamiento.

Arquitectura basada en eventos:

La arquitectura basada no sería la opción más adecuada para todo el sistema, ya que el enfoque principal de la aplicación es el procesamiento intensivo de datos y la personalización de informes.

Arquitectura de Microkernel:

La arquitectura de *microkernel* podría ser una opción viable para este caso, ya que permite la flexibilidad necesaria para adaptarse a las necesidades de procesamiento de datos y personalización de informes. No obstante, la complejidad alta de este patrón podría dificultar su implementación.

Arquitectura Orientada a Servicios (SOA):

La arquitectura SOA sería una opción adecuada en este contexto, debido a que permite la reutilización de servicios y una mejor organización de componentes, de la misma forma, los servicios pueden adaptarse y escalarse según las necesidades de procesamiento de datos y personalización de informes. No obstante, la comunicación entre servicios podría generar latencia y afectar el rendimiento en tiempo real.

Arquitectura de Microservicios:

La arquitectura de microservicios sería la mejor opción para este caso, ya que ofrece escalabilidad y modularidad para adaptarse a las demandas de procesamiento intensivo de datos y personalización de informes. La separación de la funcionalidad en microservicios independientes puede optimizar el rendimiento y adaptarse a las necesidades específicas de la organización.

Arquitectura recomendada para el caso 4:

En este caso, tomando en cuenta los requerimientos del proyecto, la arquitectura de microservicios sería la mejor opción, debido a que esta arquitectura ofrece la escalabilidad y modularidad necesarias para adaptarse a las demandas de la aplicación, permitiendo un rendimiento óptimo y una personalización eficiente de los informes.

Caso 5: Desarrollo de una aplicación *web* de gestión de proyectos para un equipo distribuido con requisitos de colaboración en tiempo real y alta disponibilidad

Arquitectura en N capas:

La aplicación de este patrón de arquitectura podría tener ventajas para este caso, puesto que nos ayuda a separar la lógica de negocio y la presentación. No obstante, en base a los

requerimientos del caso se descarta la implementación de este patrón, debido a los desafíos que presenta al manejar la colaboración en tiempo real y la alta disponibilidad.

Arquitectura Cliente-Servidor:

La arquitectura cliente-servidor puede proporcionar como ventaja una clara división entre la interfaz de usuario y el procesamiento de datos, no obstante, como hemos observado en otros casos podría enfrentar problemas de escalabilidad y rendimiento al manejar la colaboración en tiempo real y la alta disponibilidad requerida en este contexto.

Arquitectura basada en eventos:

La arquitectura basada en eventos sería adecuada en este contexto, ya que permite una rápida respuesta a eventos y notificaciones en tiempo real, lo cual es crucial para una aplicación de gestión de proyectos que requiere colaboración en tiempo real entre miembros del equipo distribuido.

Arquitectura de Microkernel:

Aunque la arquitectura de *microkernel* ofrece flexibilidad y extensibilidad, su complejidad adicional podría no ser necesaria en una aplicación de gestión de proyectos y podría dificultar la implementación y el mantenimiento del sistema.

Arquitectura Orientada a Servicios (SOA):

La arquitectura SOA podría ser adecuada para este caso, ya que permite la reutilización de servicios y una mejor organización de componentes. No obstante, la comunicación entre servicios y la orquestación podrían generar latencia y afectar el rendimiento en tiempo real de la colaboración entre miembros del equipo distribuido.

Arquitectura de Microservicios:

La arquitectura de microservicios sería una buena opción en este contexto, ya que ofrece escalabilidad y modularidad para adaptarse a las demandas de colaboración en tiempo real y alta disponibilidad. Al separar la funcionalidad en microservicios independientes, la aplicación puede optimizar el rendimiento y garantizar una alta disponibilidad.

Arquitecturas recomendadas para el caso 5:

Para el caso de una aplicación *web* de gestión de proyectos para un equipo distribuido con requisitos de colaboración en tiempo real y alta disponibilidad, la arquitectura basada en eventos y la arquitectura de microservicios serían las más adecuadas. Las características principales de cada una son: la arquitectura basada en eventos permitiría a la aplicación responder rápidamente a eventos y notificaciones en tiempo real para los usuarios, mientras que la arquitectura de microservicios permitiría la escalabilidad necesaria para garantizar un rendimiento óptimo y alta disponibilidad.

Caso 6: Desarrollo de un comercio electrónico escalable para una empresa en crecimiento con requisitos de integración de sistemas externos y gestión de inventario en tiempo real

Arquitectura en N capas:

La arquitectura en N capas podría ser adecuada en este contexto, ya que permite facilitar la escalabilidad debido a su moderado acoplamiento y alta cohesión. No obstante, enfrentaría problemas al manejar la integración de sistemas externos y la gestión de inventario en tiempo real.

Arquitectura Cliente-Servidor:

La arquitectura cliente-servidor no sería una buena opción en este caso debido a los potenciales problemas de escalabilidad y rendimiento que se presentarían al gestionar la integración de sistemas externos.

Arquitectura basada en eventos:

La arquitectura basada en eventos sería adecuada en este contexto, especialmente para la gestión de inventario en tiempo real y las notificaciones, debido a que esta arquitectura permite una respuesta rápida a eventos y actualizaciones en tiempo real, lo cual es fundamental para un comercio electrónico en crecimiento.

Arquitectura de Microkernel:

Aunque la arquitectura de *microkernel* ofrece flexibilidad y extensibilidad, su complejidad alta podría no ser necesaria en un comercio electrónico y generaría problemas en la implementación y el mantenimiento del sistema.

Arquitectura Orientada a Servicios (SOA):

La arquitectura SOA sería adecuada en este contexto, ya que facilita la integración de sistemas externos y la reutilización de servicios. Permite una mejor organización de componentes y una comunicación más eficiente entre servicios, lo cual es crucial para la integración de sistemas externos en un comercio electrónico.

Arquitectura de Microservicios:

La arquitectura de microservicios sería una excelente opción en este contexto, ya que ofrece escalabilidad y modularidad para adaptarse a las demandas de un comercio electrónico en crecimiento. Al separar la funcionalidad en microservicios independientes, la aplicación

puede optimizar el rendimiento, la gestión de inventario en tiempo real y la integración de sistemas externos.

Arquitecturas recomendadas para el caso 6:

Para el desarrollo de un comercio electrónico escalable con requisitos de integración de sistemas externos y gestión de inventario en tiempo real, los patrones de arquitectura adecuados son la arquitectura basada en eventos, arquitectura orientada a servicios y arquitectura de microservicios.

Las principales ventajas de las arquitecturas en base al contexto son: la arquitectura basada en eventos permite una rápida respuesta a eventos y actualizaciones en tiempo real, la arquitectura orientada a servicios facilita la integración de sistemas externos y la reutilización de servicios, finalmente, la arquitectura de microservicios proporciona la escalabilidad y modularidad necesarias para un comercio electrónico en crecimiento.

Presentación de resultados

El patrón de arquitectura adecuado para cada caso en base a lo descrito anteriormente se puede resumir en la tabla que se presenta a continuación, esta permite que un desarrollador identifique el caso práctico que mejor describa su proyecto, y usarla como referencia para la elección del patrón de arquitectura en su proyecto de forma informada.

Símbolos:

✓: Patrón adecuado para el contexto

⚠: Presenta limitaciones

✗: Patrón no adecuado

Tabla 2

Comparación patrones de arquitectura en base al contexto

Caso/Patrón de arquitectura	N Capas	Cliente-Servidor	Basada en eventos	Microkernel	Orientada a Servicios	Microservicios
Caso 1: Aplicación web colaborativa	✓	⚠	⚠	⚠	✓	✓
Caso 2: Aplicación web para análisis de datos	⚠	✓	⚠	✗	✗	✗
Caso 3: Red social en tiempo real	⚠	✗	✓	✗	⚠	✓
Caso 4: Análisis y visualización de datos	⚠	✗	⚠	⚠	⚠	✓
Caso 5: Gestión de proyectos distribuidos	✗	✗	✓	⚠	⚠	✓
Caso 6: Comercio electrónico escalable	⚠	✗	✓	✗	✓	✓

A continuación, se analizará la escalabilidad para cada tipo de proyecto en los casos planteados anteriormente.

Arquitectura en N capas

Esta arquitectura ofrece una escalabilidad moderada o limitada en la mayoría de los casos propuestos, a pesar de que permite separar la lógica de negocio, presentación y acceso a datos, su capacidad para escalar puede verse limitada debido a la dependencia entre las capas y la posible falta de modularidad.

Arquitectura Cliente-Servidor

Esta arquitectura generalmente no se adapta de forma correcta a la escalabilidad en los casos mencionados debido a cuellos de botella que pueda generar y sus limitaciones de rendimiento cuando el número de clientes y solicitudes aumenta.

Arquitectura basada en eventos

Este patrón de arquitectura presenta una gran escalabilidad en la mayoría de los casos, a razón de que esta permite a las aplicaciones responder rápidamente a eventos y actualizaciones en tiempo real, siendo fundamental para aplicaciones con interacciones en tiempo real y alta capacidad de respuesta a eventos.

Arquitectura de Microkernel

La arquitectura de *microkernel* no es escalable en la mayoría de los casos descritos, debido a su enfoque en la extensibilidad y la flexibilidad en lugar de la escalabilidad, una consideración importante es la alta complejidad presenta este patrón, ya que esto puede influir en su implementación.

Arquitectura Orientada a Servicios (SOA)

La arquitectura SOA muestra una escalabilidad limitada en algunos de los casos propuestos y buena en otros, tales como en el comercio electrónico escalable o la aplicación *web* colaborativa. Esta permite la integración de sistemas externos y la reutilización de servicios, lo cual puede ser útil para ciertos casos de uso, pero no es suficiente para garantizar una escalabilidad en otros contextos.

Arquitectura de Microservicios

Este patrón de arquitectura demuestra una buena escalabilidad para todos los casos propuestos, debido a que permite la separación de funcionalidades en servicios independientes, lo que facilita la adaptación y el crecimiento de una aplicación a medida que cambian sus necesidades y requisitos. Sin embargo, se debe tomar en cuenta ciertas consideraciones al momento de implementar este patrón de arquitectura, tales como la complejidad en la gestión de arquitectura y costo de recursos.

Además de tomar una decisión informada sobre el patrón de arquitectura a utilizar, es importante considerar la escalabilidad que ofrece cada patrón de arquitectura en cada caso e identificar si esta se adapta a las necesidades del proyecto a desarrollar, por lo que a continuación se muestra una tabla que resume la escalabilidad de cada patrón de arquitectura en base al caso propuesto.

Símbolos:

✓: Buena escalabilidad

⚠: Escalabilidad moderada o limitada

✗: Baja escalabilidad o no escalable

Tabla 3

Comparación de escalabilidad en los patrones de arquitectura

Caso/Patrón de arquitectura	N Capas	Cliente-Servidor	Basada en eventos	Microkernel	Orientada a Servicios	Microservicios
Caso 1: Aplicación web colaborativa	⚠	✗	✓	✗	✓	✓
Caso 2: Aplicación web para análisis de datos	⚠	✗	⚠	✗	⚠	✓
Caso 3: Red social en tiempo real	⚠	✗	✓	✗	⚠	✓
Caso 4: Análisis y visualización de datos	⚠	✗	⚠	✗	⚠	✓
Caso 5: Gestión de proyectos distribuidos	⚠	✗	✓	✗	⚠	✓
Caso 6: Comercio electrónico escalable	⚠	✗	✓	✗	✓	✓

CONCLUSIONES Y RECOMENDACIONES

Conclusiones

- La arquitectura de microservicios se mostró como el patrón más apropiado para la mayoría de los casos planteados, sus principales ventajas incluyen la escalabilidad, flexibilidad, mantenimiento simplificado y adaptabilidad a diferentes tecnologías. Sin embargo, es importante considerar que, para proyectos de pequeña escala y simples, su implementación puede resultar excesiva y generar una sobrecarga de recursos innecesaria.
- La arquitectura cliente-servidor es la menos adecuada para la mayoría de los casos planteados, debido a sus grandes limitaciones para los proyectos actuales, tales como su escalabilidad limitada o los cuellos de botella en su rendimiento, dificultando la capacidad de mantenimiento y evolución del sistema.
- La arquitectura basada en eventos se mostró como una opción adecuada para proyectos de mediana escala y que no requieran de procesamiento de grandes volúmenes de datos, esto debido a su bajo acoplamiento y flexibilidad, sin presentar una alta complejidad y sin la necesidad de contar con una infraestructura costosa para la escala del proyecto.
- Se evidenció que no existe una solución global en cuanto a la selección de un patrón de arquitectura, cada uno de ellos presentó distintas ventajas según el tipo de proyecto a desarrollar, por lo que se tomó en cuenta las limitaciones sobre cada uno de los patrones analizados.
- El análisis comparativo de cada patrón de arquitectura en base al tipo de proyecto permitió crear una tabla comparativa para que los desarrolladores puedan tomar

decisiones informadas sobre qué patrón utilizar en sus proyectos, tomando en cuenta el acoplamiento, cohesión, complejidad y escalabilidad en cada uno de ellos.

- La aplicación del método Delphi brindó una visión integral de los factores que influyen en la implementación de los patrones de arquitectura, tales como el rendimiento en la arquitectura cliente-servidor o la reutilización de servicios en la arquitectura orientada a servicios. Al involucrar a un grupo diverso de expertos, se logró obtener una amplia gama de opiniones y conocimientos.

Recomendaciones

- Sustentar los argumentos e información a utilizar en trabajos académicos mediante fuentes confiables y verificables que sean respaldadas por expertos en el campo de estudio para garantizar la calidad y validez de la información
- Especificar el significado de los términos a utilizar en trabajos académicos, especialmente en el ámbito de tecnología, para evitar caer en términos erróneos debido a la gran ambigüedad que se encuentra en las diversas fuentes de información.
- Investigar cómo los patrones de arquitectura pueden integrarse a las nuevas herramientas y *frameworks* emergentes, tales como *progressive web apps* (PWA) o *single page applications* (SPA).
- Aplicar la tabla resultante del análisis comparativo para la creación de un nuevo proyecto de desarrollo *web*, identificando el caso propuesto en el análisis más similar al proyecto a desarrollar para evidenciar las ventajas y desventajas de cada patrón de arquitectura con relación al proyecto.

BIBLIOGRAFÍA

Brown, S. (2017). Software Architecture for Developers. https://leanpub.com/software-architecture-for-developers

Kokol, P. (2021). Software quality: A Historical and Synthetic Content Analysis. https://arxiv.org/ftp/arxiv/papers/2106/2106.14598.pdf

Gharbi, M., Koschel, A. & Rausch, A. (2019). Software Architecture Fundamentals: A Study Guide for the Certified Professional for Software Architecture® – Foundation Level – iSAQB compliant. Beltz Verlag.

Huaman, W. (2018). Los patrones comunes de arquitectura de software. Medium; Medium. https://medium.com/@maniakhitoccori/los-patrones-comunes-de-arquitectura- de-software-d8b9047edf0b

Ingalls, S. (2021, November 17). What is the Client-Server Model? ServerWatch. https://www.serverwatch.com/guides/client-server-model/

Hernández, R., Fernández, C., & Baptista, P. (2010). Metodología de la investigación (5ª ed.). México: McGraw-Hill.

Denzin, K., & Lincoln, S. (Eds.). (2018). The SAGE handbook of qualitative research. Sage publications.

Babok, I. (2015). Guide to the Business Analysis Body of Knowledge (BABOK® Guide). International Institute of Business Analysis.

Oracle. (s.f.). Learn to Architect Microservices. https://docs.oracle.com/es/solutions/learn-architect-microservice/index.html

Cloudflare. (n.d.). Modelo de interconexión de sistemas abiertos (OSI). https://www.cloudflare.com/es-es/learning/ddos/glossary/open-systems-interconnection-model-osi/

Newman, S. (2015). Building microservices: designing fine-grained systems. O'Reilly Media.

Pautasso, C., Zimmermann, O., & Leymann, F. (2017). Restful web services vs. big web services: making the right architectural decision. In Proceedings of the 17th international conference on World Wide Web. https://dl.acm.org/doi/10.1145/1367497.1367606

CloudZero (2023). Horizontal vs Vertical Scaling: Which Is Best for Your Application? https://www.cloudzero.com/blog/horizontal-vs-vertical-scaling

Richards, M. (2017). Software Architecture Patterns. O'Reilly Media. https://isip.piconepress.com/courses/temple/ece_1111/resources/articles/20211201_software_architecture_patterns.pdf

Henrich, V., Hinrichs, E., Hinrichs, M. y Zastrow, T. (2010). Service-oriented architectures: from desktop tools to web services and web applications. ResearchGate. https://www.researchgate.net/publication/267849988_SERVICE-ORIENTED_ARCHITECTURES_FROM_DESKTOP_TOOLS_TO_WEB_SERVICES_AND_WEB_APPLICATIONS

González, J. A. (2012). Prospectiva de la información: conceptos y herramientas. Ediciones Universidad de Salamanca.

IEEE. (1991). IEEE standard glossary of software engineering terminology. IEEE Standards Board.

Dutonde P., Mamidwar S., & Korvate, M. (2022). Web development technologies: A review. International Journal of Advanced Science and Technology. https://doi.org/10.22214/ijraset.2022.39839

Boehm, B., & Basili, V. (2001). Software defect reduction top 10 list. Computer. https://www.cs.umd.edu/projects/SoftEng/ESEG/papers/82.78.pdf

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R. H., Konwinski, A. & Zaharia, M. (2010). A view of cloud computing. https://dl.acm.org/doi/10.1145/1721654.1721672

Booch, G., Maksimchuk, R. A., Engle, M. W., Young, B. J., Conallen, J., & Kustanovich, A. (2007). Object-Oriented Analysis and Design with Applications. Addison-Wesley Professional.

Martin, R. C. (2003). Agile Software Development: Principles, Patterns, and Practices. Prentice Hall.

Shaw, M., & Garlan, D. (1996). Software architecture: Perspectives on an emerging discipline. Prentice Hall.

Joshi, N., & Shah, N. (2015). Liskov substitution principle (LSP) in object-oriented programming (OOP). International Journal of Computer Applications. https://doi.org/10.5120/21548-1589

Sommerville, I. (2016). Software engineering. Pearson Education Limited.

Freeman, E., & Robson, E. (2014). Head first design patterns: A brain-friendly guide. O'Reilly Media.

Melzer I. (2010). Service-orientierte Architekturen mit Web Services: Konzepte Standards Praxis, 4th ed., Spektrum Akademischer Verlag Heidelberg.

Martin, R. C. (2017). Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall.

Pressman, R. S., & Maxim, B. R. (2015). Ingeniería del software: un enfoque práctico. McGraw-Hill Education.

Tanenbaum, A. S., & Steen, M. V. (2017). Distributed systems: principles and paradigms. Pearson Education.

Bass, L., Clements, P., & Kazman, R. (2015). Software architecture in practice (3ra ed.). Addison-Wesley.

Saaty, T. L., & Peniwati, K. (2008). Group decision making: Drawing out and reconciling differences. RWS Publications.

Hoffman, T. (2014). Scalability rules: Principles for scaling web sites. Addison-Wesley Professional.

Creswell, J. W., & Creswell, J. D. (2018). Research design: Qualitative, quantitative, and mixed methods approaches (5ta ed.). Sage publications.

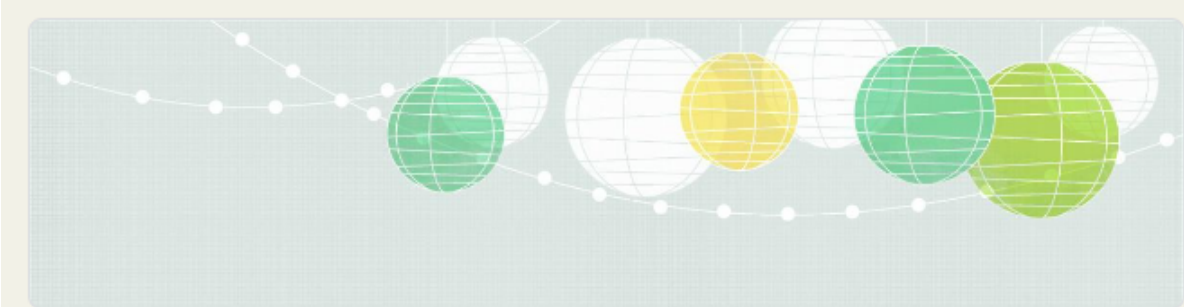
TechBeacon. (2021). Top 5 software architecture patterns: How to make the right choice. https://techbeacon.com/app-dev-testing/top-5-software-architecture-patterns-how-make-right-choice

Apiumhub. (2020). Major software architecture patterns. https://apiumhub.com/tech-blog-barcelona/major-software-architecture-patterns/

MilesWeb. (2022). Software architecture patterns: An overview. https://www.milesweb.in/blog/technology-hub/software-architecture-patterns/?utm_source=LinkedIn-InternalLinking&utm_medium=LinkedIn-Article&utm_campaign=Sarang-LinkedIn-Software-architecture-110522

Decipher Zone Softwares. (2022, December 13). Software Architecture Patterns. https://www.decipherzone.com/blog-detail/software-architecture-patterns-type

1. Primera encuesta



Encuesta sobre Factores Críticos para Implementar Patrones de Arquitectura en Desarrollo de Software Web

Estamos realizando una investigación para identificar los factores críticos que influyen en la implementación exitosa de patrones de arquitectura en proyectos de desarrollo de software web. Su participación en esta encuesta es muy valiosa para nosotros. La encuesta consta de dos rondas y le tomará aproximadamente 15 minutos completarla. Toda la información que proporcione será confidencial y se utilizará únicamente con fines de investigación. Muchas gracias por su tiempo y contribución.

1. Arquitectura en N capas
2. Arquitectura Cliente-Servidor
3. Arquitectura Basada en Eventos
4. Arquitectura de Microkernel
5. Arquitectura Orientada a Servicios (SOA)
6. Arquitectura Basada en Microservicios

Para cada patrón de arquitectura, se proporciona una lista de posibles factores críticos. Por favor, califique la importancia de cada factor en una escala del 1 al 5, siendo 1 "No importante" y 5 "Extremadamente importante". Si considera que hay algún factor crítico adicional que no esté en la lista, por favor, añádalo al final de cada sección.

Acceder a Google para guardar el progreso. Más información

Siguiente

Borrar formulario

1. Arquitectura en N capas

Califique la importancia de cada factor *

1= Nada importante 5= Extremadamente importante

	1	2	3	4	5
Modularidad	○	○	○	○	○
Separación de responsabilidades	○	○	○	○	○
Comunicación entre capas	○	○	○	○	○
Facilidad de mantenimiento	○	○	○	○	○

Factor adicional (si lo considera necesario)

Escriba el grado de importancia después de su respuesta. Por ejemplo: Modularidad 3

Tu respuesta

Atrás Siguiente Borrar formulario

2. Arquitectura Cliente-Servidor

Califique la importancia de cada factor *

1= Nada importante 5= Extremadamente importante

	1	2	3	4	5
Rendimiento	○	○	○	○	○
Balanceo de carga	○	○	○	○	○
Seguridad	○	○	○	○	○
Tolerancia a fallo	○	○	○	○	○

Factor adicional (si lo considera necesario)

Escriba el grado de importancia después de su respuesta. Por ejemplo: Modularidad 3

Tu respuesta

Atrás Siguiente Borrar formulario

3. Arquitectura Basada en Eventos

Califique la importancia de cada factor *

1= Nada importante 5= Extremadamente importante

	1	2	3	4	5
Manejo de eventos	○	○	○	○	○
Escalabilidad	○	○	○	○	○
Desacoplamiento	○	○	○	○	○
Latencia	○	○	○	○	○

Factor adicional (si lo considera necesario)

Escriba el grado de importancia después de su respuesta. Por ejemplo: Modularidad 3

Tu respuesta

Atrás Siguiente Borrar formulario

4. Arquitectura de Microkernel

Califique la importancia de cada factor *

1= Nada importante 5= Extremadamente importante

	1	2	3	4	5
Extensibilidad	○	○	○	○	○
Abstracción	○	○	○	○	○
Complejidad	○	○	○	○	○
Portabilidad	○	○	○	○	○

Factor adicional (si lo considera necesario)

Escriba el grado de importancia después de su respuesta. Por ejemplo: Modularidad 3

Tu respuesta

Atrás Siguiente Borrar formulario

5. Arquitectura Orientada a Servicios (SOA)

Califique la importancia de cada factor *

1= Nada importante 5= Extremadamente importante

	1	2	3	4	5
Integración	○	○	○	○	○
Reutilización	○	○	○	○	○
Descubrimiento de servicios	○	○	○	○	○
Interoperabilidad	○	○	○	○	○

Factor adicional (si lo considera necesario)

Escriba el grado de importancia después de su respuesta. Por ejemplo: Modularidad 3

Tu respuesta

Atrás Siguiente Borrar formulario

6. Arquitectura Basada en Microservicios

Califique la importancia de cada factor *

1= Nada importante 5= Extremadamente importante

	1	2	3	4	5
Desacoplamiento	○	○	○	○	○
Escalabilidad	○	○	○	○	○
Monitoreo	○	○	○	○	○
Resilencia	○	○	○	○	○

Factor adicional (si lo considera necesario)

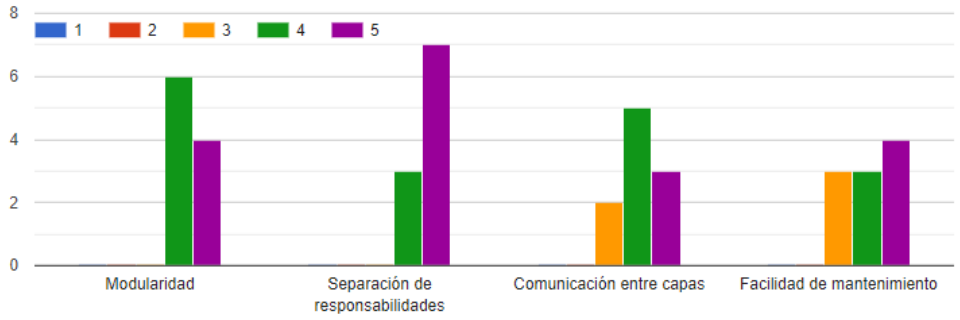
Escriba el grado de importancia después de su respuesta. Por ejemplo: Modularidad 3

Tu respuesta

Atrás Enviar Borrar formulario

1.1. Resultados primera encuesta

1. Arquitectura en N capas



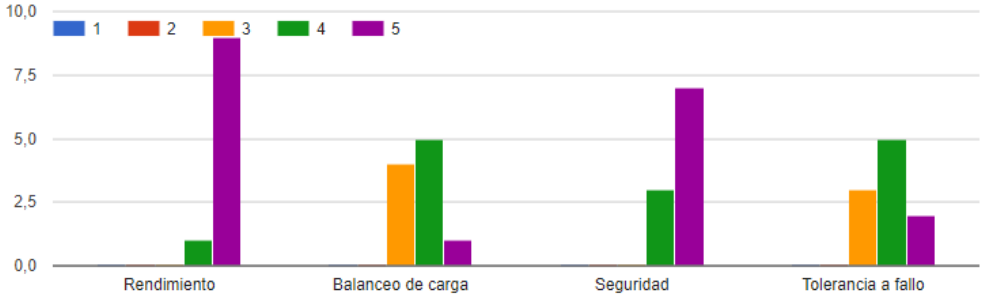
Factor adicional (si lo considera necesario)

2 respuestas

Gestión de dependencias 3

Uso de frameworks 3

2. Arquitectura Cliente-Servidor



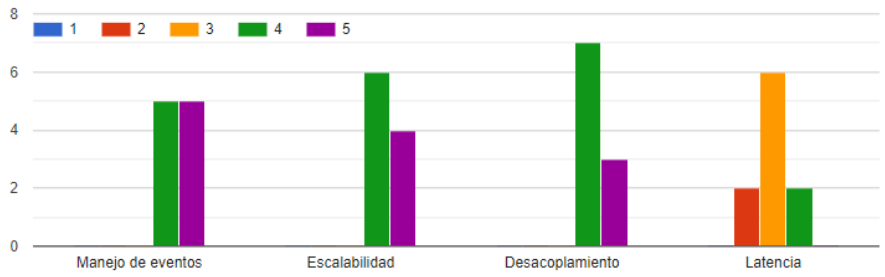
Factor adicional (si lo considera necesario)

2 respuestas

Protocolos de comuicación 4

Distribución geográfica 4

3. Arquitectura Basada en Eventos



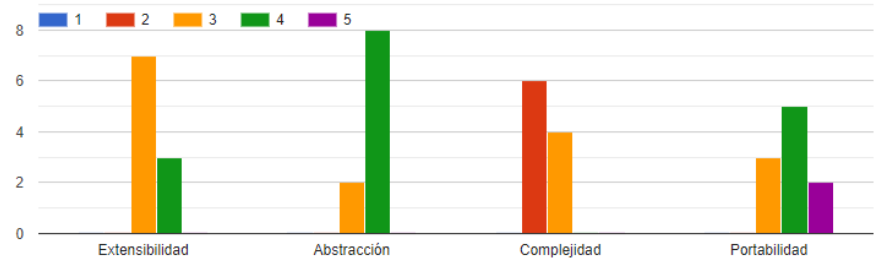
Factor adicional (si lo considera necesario)

2 respuestas

Persistencia de eventos 3

Priorización de eventos 4

4. Arquitectura de Microkernel



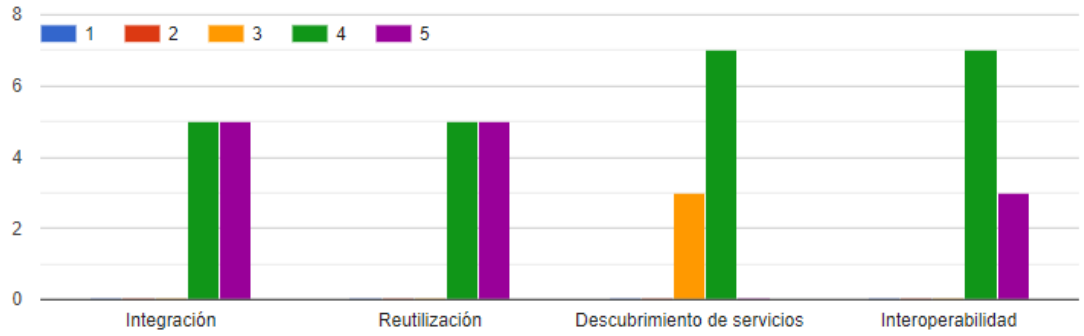
Factor adicional (si lo considera necesario)

2 respuestas

Gestión de plugins 4

Documentación y soporte 4

5. Arquitectura Orientada a Servicios (SOA)

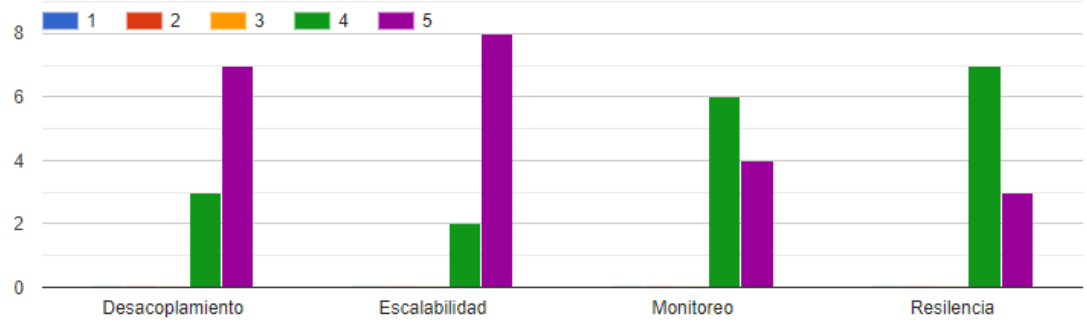


Factor adicional (si lo considera necesario)

1 respuesta

Estándares y protocolos 4

6. Arquitectura Basada en Microservicios



Factor adicional (si lo considera necesario)

0 respuestas

Aún no hay respuestas para esta pregunta.

2. Segunda encuesta

1. Arquitectura en N capas

Califique la importancia de cada factor *

1= Nada importante 5= Extremadamente importante

Resultados de la primera encuesta:

- Modularidad (Promedio: 4.4)
- Separación de responsabilidades (Promedio: 4.7)
- Comunicación entre capas (Promedio: 4.1)
- Facilidad de mantenimiento (Promedio: 4.1)
- Gestión de dependencias (Promedio: 3)
- Uso de frameworks (Promedio: 3)

	1	2	3	4	5
Modularidad	○	○	○	○	○
Separación de responsabilidades	○	○	○	○	○
Comunicación entre capas	○	○	○	○	○
Facilidad de mantenimiento	○	○	○	○	○
Gestión de dependencias	○	○	○	○	○
Uso de frameworks	○	○	○	○	○

Atrás Siguiente Borrar formulario

2. Arquitectura Cliente-Servidor

- Rendimiento (Promedio: 4.9)
- Balanceo de carga (Promedio: 3.7)
- Seguridad (Promedio: 4.7)
- Tolerancia a fallos (Promedio: 3.9)
- Protocolos de comunicación (Promedio: 4)
- Distribución geográfica (Promedio: 4)

Califique la importancia de cada factor *

1= Nada importante 5= Extremadamente importante

	1	2	3	4	5
Rendimiento	○	○	○	○	○
Balanceo de carga	○	○	○	○	○
Seguridad	○	○	○	○	○
Tolerancia a fallo	○	○	○	○	○
Protocolos de comunicación	○	○	○	○	○
Distribución geográfica	○	○	○	○	○

Atrás Siguiente Borrar formulario

3. Arquitectura Basada en Eventos

- Manejo de eventos (Promedio: 4.5)
- Escalabilidad (Promedio: 4.4)
- Desacoplamiento (Promedio: 4.3)
- Latencia (Promedio: 3)
- Persistencia de eventos (Promedio: 3)
- Priorización de eventos (Promedio: 4)

Califique la importancia de cada factor *

1= Nada importante 5= Extremadamente importante

	1	2	3	4	5
Manejo de eventos	○	○	○	○	○
Escalabilidad	○	○	○	○	○
Desacoplamiento	○	○	○	○	○
Latencia	○	○	○	○	○
Persistencia de eventos	○	○	○	○	○
Priorización de eventos	○	○	○	○	○

Atrás Siguiente Borrar formulario

4. Arquitectura de Microkernel

- Extensibilidad (Promedio: 3.3)
- Abstracción (Promedio: 3.8)
- Complejidad (Promedio: 2.4)
- Portabilidad (Promedio: 3.9)
- Gestión de plugins (Promedio: 4)
- Documentación y soporte (Promedio: 4)

Califique la importancia de cada factor *

1= Nada importante 5= Extremadamente importante

	1	2	3	4	5
Extensibilidad	○	○	○	○	○
Abstracción	○	○	○	○	○
Complejidad	○	○	○	○	○
Portabilidad	○	○	○	○	○
Gestión de plugins	○	○	○	○	○
Documentación y soporte	○	○	○	○	○

Atrás Siguiente Borrar formulario

5. Arquitectura Orientada a Servicios (SOA)

Califique la importancia de cada factor *

1= Nada importante 5= Extremadamente importante

- Integración (Promedio: 4.5)
- Reutilización (Promedio: 4.5)
- Interoperabilidad (Promedio: 4.3)
- Estándares y protocolos (Promedio: 4)

	1	2	3	4	5
Integración	○	○	○	○	○
Reutilización	○	○	○	○	○
Interoperabilidad	○	○	○	○	○
Estándares y protocolos	○	○	○	○	○

Atrás Siguiente Borrar formulario

6. Arquitectura Basada en Microservicios

- Desacoplamiento (Promedio: 4.7)
- Escalabilidad (Promedio: 4.8)
- Monitoreo (Promedio: 4.4)
- Resiliencia (Promedio: 4.3)

Califique la importancia de cada factor *

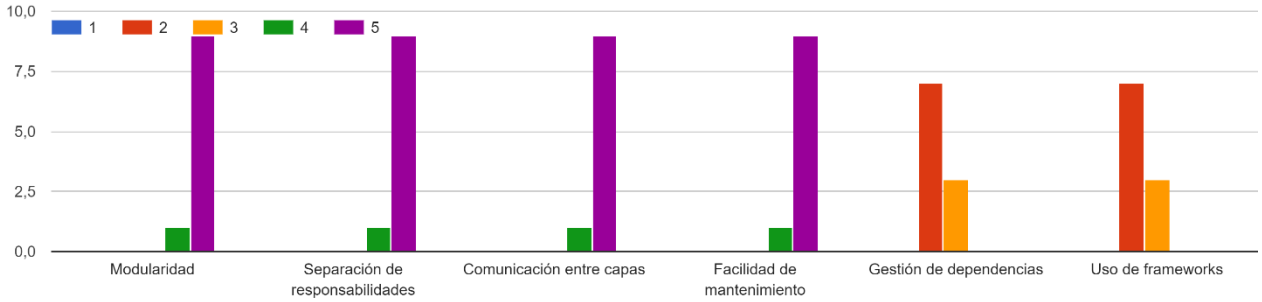
1= Nada importante 5= Extremadamente importante

	1	2	3	4	5
Desacoplamiento	○	○	○	○	○
Escalabilidad	○	○	○	○	○
Monitoreo	○	○	○	○	○
Resilencia	○	○	○	○	○

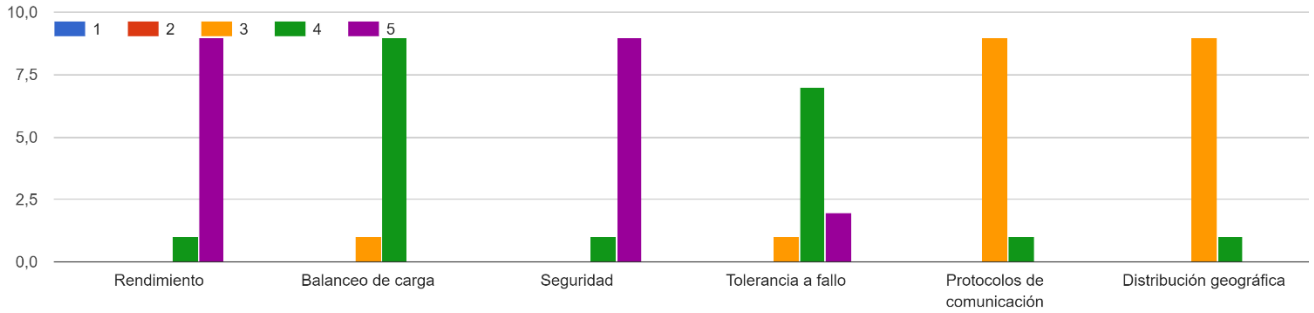
Atrás Enviar Borrar formulario

2.1. Resultados segunda encuesta

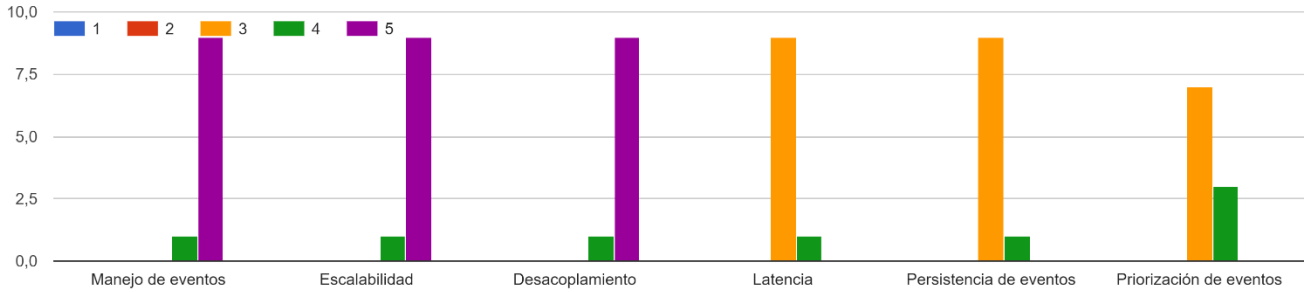
1. Arquitectura en N capas



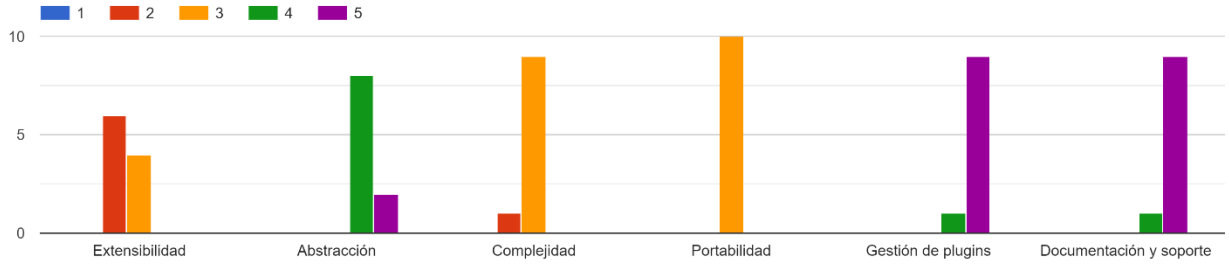
2. Arquitectura Cliente-Servidor



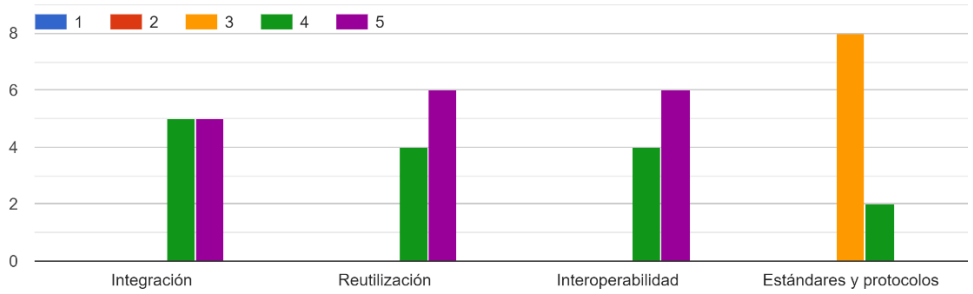
3. Arquitectura Basada en Eventos



4. Arquitectura de Microkernel



5. Arquitectura Orientada a Servicios (SOA)



6. Arquitectura Basada en Microservicios