Estrategia de trazabilidad de elementos de seguridad a través del ciclo de vida del software usando el modelo cascada

DAVID RICARDO ZULUAGA OSSA

Trabajo de grado

Asesores
Paola Andrea Vallejo Correa
Daniel Correa Botero

UNIVERSIDAD EAFIT

ESCUELA DE CIENCIAS APLICADAS E INGENIERÍA

MAESTRÍA EN INGENIERÍA

MEDELLÍN

2025

TABLA DE CONTENIDO

1 INTRODUCCIÓN 10

2 PLANTEAMIENTO DEL PROBLEMA 12

3 JUSTIFICACIÓN 13

4 OBJETIVOS 14

4.1 Objetivo General 14

4.2 Objetivos Específicos 14

5 MARCO TEÓRICO 15

5.1 Conceptos relacionados con seguridad 15

5.1.1 Seguridad 15

5.1.2 Vulnerabilidad 15

5.1.3 Herramienta de análisis SonarQube 16

5.2 Conceptos relacionados con trazabilidad 17

5.2.1 Trazabilidad 17

5.2.2 Trazabilidad en modelos secuenciales de desarrollo 17

5.2.3 Trazabilidad aplicada a la seguridad 17

5.3 Conceptos relacionados con el ciclo de vida del desarrollo de software 18

5.3.1 Ciclo de vida del desarrollo de software 18

5.3.2 Modelos aplicables al ciclo de vida del desarrollo de software 18

5.3.3 Modelo en cascada 19

5.4 Conceptos relacionados con modelado 21

5.4.1 Modelado 21

5.4.2 Diagrama de clases 21

5.4.3 Diagrama de despliegue 22
5.4.4 Diagrama de secuencia 22
6 METODOLOGÍA 23
6.1 Identificación y motivación del problema 23
6.2 Definición de objetivos 24
6.3 Diseño y desarrollo 24
6.4 Demostración 25
6.5 Evaluación 25
6.6 Comunicación 25
7 DESARROLLO DEL TRABAJO 26
7.1 Identificación del proyecto 28
7.2 Definición de Requisitos 29
7.2.1 Definición de requisitos funcionales 29
7.2.2 Definición de requisitos no funcionales de seguridad 30
7.3 Modelado del sistema y del software 31
7.3.1 Diagrama de clases 32
7.3.2 Diagrama de secuencia 35
7.3.3 Diagrama de despliegue 37
7.4 Implementación y pruebas unitarias 39
7.4.1 Implementación 40
7.4.2 Pruebas unitarias 42
7.5 Integración y pruebas del sistema 44
7.5.1 Integración 44
7.5.2 Pruebas del sistema 45

7.5.3 Pipeline de despliegue 46
7.6 Operación y mantenimiento..48
7.6.1 Operación 48
7.6.2 Mantenimiento 49
8 RESULTADOS DE LA APLICACIÓN DE LA ESTRATEGIA DE TRAZABILIDAD 51
8.1 Identificación del proyecto 51
8.2 Definición de requisitos 52
8.2.1 Definición de requisitos funcionales..52
8.2.2 Definición de requisitos no funcionales de seguridad 53
8.3 Modelado del sistema y del software 53
8.3.1 Diagrama de clases 54
8.3.2 Diagrama de secuencia 57
8.3.3 Diagrama de despliegue 59
8.4 Implementación y pruebas unitarias..61
8.4.1 Implementación 61
8.4.2 Pruebas unitarias..64
8.5 Integración y pruebas del sistema..67
8.5.1 Integración 67
8.5.2 Pruebas del sistema 69
8.5.3 *Pipeline* de despliegue..71
8.6 Operación y mantenimiento..76
8.6.1 Operación 76
8.6.2 Mantenimiento 76
9 EVALUACIÓN DE LA ESTRATEGIA DE TRAZABILIDAD..78

9.1 Comparación cualitativa 78

9.2 Comparación cuantitativa 80

10 CONCLUSIONES 81

REFERENCIAS 84

LISTA DE TABLAS *pág.*

Tabla 1. Guía para la identificación del proyecto con sus ventajas para la trazabilidad en seguridad 28

Tabla 2. Guía para la redacción de Requisitos Funcionales 30

Tabla 3. Guía para la redacción de Requisitos No Funcionales de Seguridad 31

Tabla 4. Guía para la trazabilidad de requisitos a elementos del diagrama de clases 35

Tabla 5. Guía para la trazabilidad de requisitos a los archivos del sistema 41

Tabla 6. Guía para la matriz de verificación de requisitos con las pruebas unitarias 42

Tabla 7. Guía para la trazabilidad de requisitos con los archivos del sistema y la ubicación de las pruebas unitarias 43

Tabla 8. Guía para la trazabilidad de requisitos a pruebas de integración 46

Tabla 9. Guía donde se muestra un ciclo de despliegue automatizado en entornos de integración continua 47

Tabla 10. Identificación del proyecto 52

Tabla 11. Trazabilidad de requisitos al diagrama de clases 56

Tabla 12. Trazabilidad de Requisitos a los archivos del sistema 63

Tabla 13. Matriz de verificación de requisitos con las pruebas unitarias 65

Tabla 14. Trazabilidad de los requisitos con los archivos del sistema y la ubicación de las pruebas unitarias en los archivos del sistema 66

Tabla 15. Trazabilidad de requisitos a pruebas de integración 70

Tabla 16. Comparación cualitativa entre el modelo en cascada con trazabilidad y el modelo en cascada tradicional. 78

Tabla 17. Comparación cuantitativa entre el modelo en cascada con trazabilidad y el modelo en cascada tradicional. 80

LISTA DE FIGURAS

pág.

Figura 1. Etapas del modelo en cascada 19

Figura 2. Metodología Design Science Research 23

Figura 3. Estructura secuencial para la trazabilidad de requisitos de seguridad del software 26

Figura 4. Ejemplo de un diagrama de clases representando los métodos que realizan las acciones de los requisitos en cada clase 34

Figura 5. Ejemplo de un diagrama de secuencia 37

Figura 6. Ejemplo de un diagrama de despliegue 39

Figura 7. Ejemplo de la estructura de un árbol de directorios 41

Figura 8. Ejemplo de la representación de la arquitectura de integración de componentes 45

Figura 9. Diagrama de clases representando las interacciones entre Controladores y Modelos.... 55

Figura 10. Diagrama de secuencia de la creación de un post 58

Figura 11. Diagrama de despliegue de la aplicación 60

Figura 12. Árbol de directorios que contienen el proyecto 62

Figura 13. Representación de la arquitectura de integración de componentes 68

Figura 14. Proceso despliegue: Pipeline configurado en AWS 72

Figura 15: Proceso despliegue: Configuración de Jenkins 73

Figura 16. Proceso despliegue: Reglas preconfiguradas para el análisis de seguridad del código en SonarQube 74

Figura 17. Proceso despliegue: Resultado luego de la ejecución del pipeline en Jenkins 74

Figura 18. Proceso despliegue: Proceso final luego de la ejecución del pipeline en Jenkins 75

Resumen

En el desarrollo de software, la seguridad es una preocupación crítica que debe abordarse en todas las etapas del ciclo de vida del desarrollo de software. Especialmente cuando se trata de llevar un control riguroso sobre los requisitos de seguridad, ya que la adecuada integración de prácticas de seguridad permite proteger la integridad, confidencialidad y disponibilidad de los sistemas.

Actualmente, existen metodologías como el desarrollo seguro (*Secure SDLC*), y estándares como *OWASP* e ISO/IEC 27034 que proponen incorporar seguridad en todo el ciclo de vida del software. Sin embargo, en muchos casos, no se enfatiza suficientemente en el uso de herramientas de trazabilidad que permitan gestionar de manera continua el avance respecto a estos requisitos. En especial en el modelo cascada, donde por su naturaleza secuencial, la retroalimentación entre etapas es limitada.

Este trabajo propone una estrategia de trazabilidad para los elementos de seguridad en el contexto del modelo en cascada de Sommerville. La estrategia consiste en trazar cada requisito de seguridad, desde la etapa de definición de requisitos hasta el despliegue, relacionándolo con los elementos de diseño, implementación y pruebas correspondientes, y mantener esa relación actualizada a medida que evoluciona el proyecto de software.

De este modo, la estrategia permite detectar tempranamente omisiones o desviaciones en la aplicación de los requisitos de seguridad, lo cual no solo facilita procesos de auditoría y certificación, sino que también mejora la calidad y consistencia del software entregado. Al mantener un seguimiento estructurado y verificable, se reducen los costos derivados de correcciones tardías y se fortalece la confianza de las partes interesadas, al demostrar un compromiso con un desarrollo seguro, ordenado y alineado con los principios del modelo en cascada desde el inicio hasta el cierre del proyecto.

Abstract

Security in software development is a critical concern that must be addressed throughout the entire software development lifecycle (SDLC). Effective integration of security practices is essential to ensure system integrity, confidentiality, and availability.

Existing frameworks such as Secure SDLC, OWASP, and ISO/IEC 27034 promote the incorporation of security from the early stages of development. However, these methodologies often overlook the continuous traceability of security requirements, particularly within sequential models like the waterfall model, where the lack of iterative feedback limits visibility and adaptability across phases.

This work proposes a traceability strategy tailored to the waterfall model as described by Sommerville, focusing on linking security requirements from their initial specification through design, implementation, and testing.

The proposed approach facilitates early detection of omissions and deviations, enhances consistency and quality in deliverables, supports auditing and compliance verification, and reduces the costs associated with late-stage security fixes. Furthermore, it fosters stakeholder confidence by providing transparent evidence of secure and structured development practices across the project lifecycle.

1 INTRODUCCIÓN

El avance acelerado de la digitalización, el incremento de la conectividad global y la evolución de los ciberataques han transformado profundamente el entorno en el que se desarrolla el software. Esta transformación ha generado nuevas exigencias en la forma como se concibe, diseña, implementa y se mantiene el software. En este nuevo panorama, la protección de la información, definida como la garantía de su integridad, disponibilidad y confidencialidad, ha adquirido una relevancia central [1].

Las organizaciones enfrentan cada vez más desafíos para mantener la seguridad de sus sistemas, incluso en los escenarios en los que se definen de antemano los requisitos del sistema.

No hacer un seguimiento exhaustivo de la seguridad a lo largo del ciclo de vida del desarrollo de software, puede generar impactos negativos significativos: altos costos de remediación, poca eficacia técnica, dificultades para integrar medidas correctivas en sistemas complejos ya avanzados y la falta de visibilidad, que puede llevar a inconsistencias entre lo que se planifica y lo que finalmente se entrega, así como a la omisión de medidas de protección críticas. Además, detectar errores de seguridad en etapas avanzadas, implica costos elevados de corrección, retrabajo y pérdida de confianza.

Resolver el problema del seguimiento de la seguridad ha sido un caso de estudio durante algún tiempo. En la actualidad, existen marcos que ayudan a mantener el desarrollo seguro como *Secure SDLC* y estándares como ISO/IEC 27034, que promueven la incorporación de controles de seguridad desde etapas tempranas del desarrollo. Asimismo, prácticas de desarrollo como *DevSecOps* y métodos iterativos como Scrum, han permitido introducir prácticas de seguridad de forma continua, favoreciendo revisiones permanentes y detección temprana de riesgos. Por otro lado, empresas líderes han desarrollado metodologías propias: Google aplica *Security by Design* [2], Microsoft institucionalizó el *Security Development Lifecycle* (SDL) [3], y Netflix automatiza la supervisión de entornos con herramientas como *Security Monkey* [4].

Aun existiendo estas metodologías, existen proyectos y compañías que usan el modelo en cascada que, aunque a menudo criticado por su rigidez, sigue siendo ampliamente utilizado en sectores como el médico, contable y aeroespacial, donde la trazabilidad, la validación formal y el control documental son indispensables. En estos entornos donde se emplea el modelo en cascada, la trazabilidad de los elementos de seguridad a lo largo del ciclo de vida del software

se ve limitada por su estructura secuencial. Esta limitación se debe a que no existen mecanismos inherentes que garanticen el seguimiento continuo de los requisitos de seguridad entre etapas. Esto puede provocar que dichos requisitos se diluyan o pierdan visibilidad conforme avanza el ciclo de vida, dificultando su verificación posterior y comprometiendo la consistencia de las decisiones técnicas relacionadas con la seguridad.

Para abordar esta problemática, se propone una estrategia de trazabilidad para los requisitos de seguridad dentro de ciclo de vida del desarrollo de software cuando se usa el modelo en cascada, que permita documentar y verificar las decisiones relacionadas con la protección del sistema en cada una de sus etapas. La estrategia contempla el registro de cada requisito de seguridad desde la etapa de definición hasta el despliegue.

Esta trazabilidad facilita la detección temprana de omisiones o errores antes de avanzar de una etapa a otra, lo que permite realizar correcciones en momentos menos costosos y más seguros. Además, posibilita documentar cada decisión relacionada con la seguridad, fortaleciendo así la preparación técnica ante auditorías, procesos de certificación como los establecidos por normas o el cumplimiento de marcos regulatorios.

En resumen, esta propuesta complementa el uso del modelo cascada, aportando visibilidad, orden y trazabilidad de los elementos de seguridad, sin comprometer la estructura ni los procesos existentes.

El documento se estructura de la siguiente manera: La Sección 2 presenta el planteamiento del problema, junto con la formulación de la pregunta de investigación. La Sección 3 expone la justificación del estudio. La Sección 4 establece los objetivos generales y específicos. La Sección 5 desarrolla el marco teórico, compuesto por el marco conceptual. La Sección 6 describe la metodología adoptada para la investigación. La Sección 7 detalla, de manera secuencial, la estrategia diseñada para implementar trazabilidad de elementos de seguridad en el modelo en cascada. La Sección 8 muestra los resultados de la aplicación práctica de la estrategia propuesta. La Sección 9 presenta la evaluación de la estrategia. Finalmente, La Sección 10 presenta las conclusiones.

2 PLANTEAMIENTO DEL PROBLEMA

En la actualidad, las organizaciones dependen cada vez más del software para sus operaciones críticas y sus objetivos estratégicos, y un número creciente de esas soluciones se ejecuta sobre servicios expuestos a internet. Esta dependencia ha incrementado la frecuencia e impacto de los ciberataques, exponiendo vulnerabilidades críticas en el software.

Aunque se reconoce la necesidad de incorporar la seguridad desde las etapas tempranas del desarrollo, uno de los problemas más persistentes es la ausencia de mecanismos de trazabilidad que permitan hacer un seguimiento claro y de manera sistemática de los requisitos de seguridad a lo largo del ciclo de vida del desarrollo del software, particularmente en el modelo en cascada. En consecuencia, las decisiones tomadas en las etapas iniciales del ciclo de vida se diluyen, complicando su verificación posterior y comprometiendo la coherencia de las decisiones técnicas relacionadas con la protección del sistema.

Si bien metodologías como Scrum o prácticas como *DevSecOps* han logrado integrar la seguridad en ciclos iterativos y automatizados, en sectores regulados con procesos de certificación estrictos, ciclos de aprobación extensos y contratos basados en entregables fijos, adoptar estas prácticas resulta complejo, ya que su énfasis en el cambio constante y el despliegue rápido interfiere con la necesidad de documentación exhaustiva y planificación anticipada.

En estos contextos, el modelo en cascada sigue siendo ampliamente utilizado en ciertos sectores, ya que su estructura secuencial favorece la generación de entregables verificables por etapa, la trazabilidad técnica, y el cumplimiento normativo, aspectos fundamentales donde la seguridad debe estar documentada, validada y visible a lo largo de todo el ciclo de vida. No obstante, dicho modelo no contempla mecanismos explícitos para mantener una trazabilidad de los requisitos de seguridad entre etapas, lo que representa un punto crítico.

En este contexto, el trabajo plantea la siguiente pregunta de investigación:

¿Es posible implementar una estrategia de trazabilidad de los elementos de seguridad en cada etapa del ciclo de vida del software utilizando el modelo en cascada?

3 JUSTIFICACIÓN

La naturaleza secuencial del modelo en cascada tiene como particularidad, que cada etapa del desarrollo debe completarse antes de avanzar a la siguiente, limitando la retroalimentación continua entre etapas. Como resultado, los requisitos de seguridad, aunque suelen definirse en las etapas iniciales, se documentan y dan por concluidos una vez finalizada la etapa de definición, sin incluir implícitamente mecanismos inherentes que promuevan su revisión o trazabilidad posterior. Esto dificulta el seguimiento de dichos requisitos a lo largo del ciclo de vida del software. Esta característica ocasiona que, en muchos casos, los requisitos de seguridad se pierdan o se diluyan a lo largo del ciclo de vida, lo que puede generar vulnerabilidades. Se hace necesario contar con mecanismos para rastrear estos requisitos desde su definición hasta el despliegue del software.

Para abordar esta necesidad de darle un seguimiento a los requisitos, se propone una estrategia que habilite la trazabilidad de los requisitos de seguridad definidos, permitiendo su identificación, documentación y seguimiento a lo largo de todas las etapas del ciclo de vida del software, sin alterar la secuencia del modelo en cascada. Esta estrategia incorpora controles desde la etapa de definición de requisitos y asocia cada decisión técnica con su respectivo requisito, fortaleciendo la seguridad en el diseño, la implementación, y la integración del sistema.

Implementar esta estrategia permite fortalecer la calidad del producto final y potencialmente reducir los costos derivados de correcciones tardías, especialmente en términos de seguridad. Además, puede contribuir a facilitar la validación ante entes reguladores, a mejorar la documentación técnica y a reforzar la confianza de las partes interesadas, al demostrar un enfoque activo en pro del seguimiento riguroso hacia la seguridad a lo largo del ciclo de vida del desarrollo del software. Así, la propuesta busca trazar los elementos de seguridad, donde más que anticipar todas las amenazas posibles, se pueda dar seguimiento a que cada requisito de seguridad sea documentado y verificado desde los requisitos hasta el despliegue, contribuyendo así a un desarrollo seguro y ordenado.

4 OBJETIVOS

Esta sección presenta el objetivo general del trabajo y posteriormente los objetivos específicos del mismo.

4.1 Objetivo General

Implementar una estrategia de trazabilidad de los elementos de seguridad en cada etapa del ciclo de vida del desarrollo de software, usando el modelo en cascada, que facilite la documentación y el seguimiento de decisiones técnicas orientadas a la protección del sistema desde la definición de los requisitos hasta el despliegue.

4.2 Objetivos Específicos

- Determinar los momentos clave de cada etapa del ciclo de vida del desarrollo de software en los que deben incorporarse los elementos de seguridad, integrándolos de forma sistemática en cada etapa del modelo en cascada. Identificando las actividades propias del modelo y definiendo mecanismos específicos que permitan reforzar la protección del sistema desde la etapa de requisitos hasta su despliegue.
- Diseñar una estrategia que permita trazar los requisitos de seguridad con las decisiones técnicas tomadas a lo largo del ciclo de vida del desarrollo de software.
- Implementar la estrategia de trazabilidad propuesta en un caso práctico, mediante el desarrollo de un software que sirva como entorno controlado para evidenciar la vinculación de los requisitos de seguridad con sus correspondientes elementos de diseño, implementación, pruebas y despliegue.
- Evaluar la efectividad de la estrategia de trazabilidad de elementos de seguridad propuesta, mediante la comparación entre el modelo en cascada sin trazabilidad explícita y el modelo en cascada con la estrategia de trazabilidad propuesta en este trabajo.

5 MARCO TEÓRICO

Esta sección expone conceptos relevantes usados en el desarrollo de este trabajo, abordando aspectos esenciales relacionados con la seguridad en el desarrollo de software, la trazabilidad de requisitos, los modelos del ciclo de vida del desarrollo de software, el modelo en cascada y diferentes diagramas de modelado.

5.1 Conceptos relacionados con seguridad

Esta sección presenta los fundamentos conceptuales sobre seguridad informática, algunos tipos de vulnerabilidades que pueden comprometer un sistema y una herramienta utilizada para su detección y control.

5.1.1 Seguridad

La seguridad informática, en el contexto del desarrollo de software, es la disciplina enfocada en diseñar, construir y mantener software que funcione correctamente incluso bajo ataques maliciosos. La seguridad del software se considera una propiedad emergente del sistema, lo que significa que no basta con agregar funciones de seguridad, se necesita un diseño robusto, análisis de riesgos, pruebas orientadas a amenazas, revisiones de código y formación del equipo de desarrollo. Todo esto debe integrarse de forma proactiva y transversal en el ciclo de vida del desarrollo, para lograr sistemas más resilientes y menos expuestos a vulnerabilidades [5]. En este trabajo, se plantea una estrategia de trazabilidad que permite identificar, documentar y verificar los elementos de seguridad desde su definición hasta su validación, asegurando así un control técnico riguroso y continuo en cada etapa del desarrollo.

5.1.2 Vulnerabilidad

Una vulnerabilidad de seguridad es un defecto en el software que puede tener un impacto negativo en la seguridad del sistema. Las vulnerabilidades pueden encontrarse tanto en código desarrollado internamente como en software de terceros, y su gravedad varía desde baja hasta crítica [6].

A continuación, se presentan dos vulnerabilidades que se usarán en este trabajo, con el propósito de servir como ejemplo en la Sección 8.

5.1.2.1 Inyección SQL.

La inyección SQL se manifiesta cuando un atacante introduce comandos ilegales, a través de formularios o interfaces algorítmicas expuestas. Si el sistema no impone límites o controles adecuados sobre las entradas del usuario, un tercero podría acceder, modificar o eliminar información en la base de datos [7].

Esta vulnerabilidad se explota principalmente en formularios, campos de búsqueda o cualquier entrada que permita al atacante introducir código malicioso que será ejecutado directamente en la base de datos, por ejemplo, en una aplicación que utiliza sentencias SQL construidas directamente a partir de entradas del usuario, si alguien ingresa `1' OR '1'='1` en un campo de búsqueda, podría alterar la consulta y acceder a todos los registros de la base de datos sin autorización.

5.1.2.2 Ataque por XSS (*Cross Site Scripting*).

Una vulnerabilidad de tipo XSS (*Cross-Site Scripting*) ocurre cuando un sistema permite que el usuario ingrese datos que luego se muestran directamente en la interfaz sin aplicar validaciones ni filtros, lo que permite ejecutar código malicioso en el navegador de otros usuarios [7].

Esta vulnerabilidad con frecuencia se explota en formularios que no se validan y que ejecutan código, por ejemplo, si un atacante envía `<script>alert('XSS')</script>` en un formulario expuesto a esta vulnerabilidad, permitirá inyectar código malicioso, que podría ser usado a voluntad del atacante y se ejecutará cada vez que otro usuario visite la página que lo contenga.

5.1.3 Herramienta de análisis SonarQube

SonarQube es una herramienta de análisis estático de código fuente, que permite identificar errores, vulnerabilidades, *bugs*, *code smells* (malas prácticas) y problemas de mantenibilidad en proyectos de software [8]. Su utilización, especialmente en esta propuesta, permite

automatizar la verificación de condiciones técnicas vinculadas a los elementos de seguridad, facilitando la documentación objetiva de hallazgos y reforzando la trazabilidad entre lo especificado, lo implementado y lo validado.

5.2 Conceptos relacionados con trazabilidad

En esta sección se abordan algunos conceptos de trazabilidad desde una perspectiva general y su aplicación específica dentro del modelo en cascada y en su importancia para el tratamiento de la seguridad.

5.2.1 Trazabilidad

La trazabilidad en ingeniería de software se refiere a la capacidad de establecer y mantener relaciones explícitas entre los diferentes artefactos que componen un sistema software, tales como requisitos, modelos de diseño, código fuente, casos de prueba y documentación técnica, a lo largo de todo su ciclo de vida [9]. La trazabilidad es crucial en el desarrollo de este trabajo, ya que permite estructurar el seguimiento de los elementos de seguridad mediante una vinculación verificable entre las decisiones técnicas y los requisitos previamente definidos.

5.2.2 Trazabilidad en modelos secuenciales de desarrollo

La trazabilidad en modelos secuenciales es la capacidad de describir y seguir el rastro de un requisito en ambas direcciones, tanto desde su origen hasta su implementación (trazabilidad hacia adelante), como desde la implementación hasta su justificación original (trazabilidad hacia atrás). Esta práctica es fundamental, porque en el desarrollo secuencial, donde las etapas están claramente delimitadas, cualquier pérdida o desviación de requisitos, en especial los de seguridad, puede comprometer la coherencia y calidad del sistema final [10]. En este trabajo, la trazabilidad en modelos secuenciales, como en cascada, permite validar que los requisitos de seguridad se mantengan íntegros y comprobables en cada etapa, fortaleciendo la calidad técnica y facilitando auditorías y mantenimiento.

5.2.3 Trazabilidad aplicada a la seguridad

La trazabilidad aplicada a la seguridad es la capacidad de establecer y mantener relaciones verificables entre las prácticas de seguridad y los elementos del sistema que los implementan,

como modelos, código y pruebas. Esto es fundamental porque permite vincular las decisiones tomadas desde la etapa de definición de requisitos hasta su implementación efectiva [11]. En este trabajo es importante trazar los requisitos de seguridad, en especial en modelos secuenciales como el de cascada, cuya secuencia estricta vuelve costoso y complejo regresar a etapas anteriores: si los requisitos de seguridad no se rastrean desde el inicio, las evidencias de su cumplimiento pueden perderse a medida que el proyecto avanza, lo que puede dejar vulnerabilidades latentes y dificultar ajustes posteriores.

5.3 Conceptos relacionados con el ciclo de vida del desarrollo de software

Esta sección describe las etapas generales que conforman el ciclo de vida del desarrollo del software, se detalla el modelo en cascada y se abordan las etapas de dicho modelo.

5.3.1 Ciclo de vida del desarrollo de software

El ciclo de vida del desarrollo de software es descrito como un conjunto de actividades fundamentales que llevan a la producción de un sistema de software, comprendiendo cinco momentos claves: especificación, diseño, desarrollo, validación y evolución [12]. Entender el ciclo de vida y su comprensión de los diferentes modelos es esencial para identificar la importancia de integrar los elementos de seguridad a lo largo del ciclo de vida y cómo su trazabilidad puede mantenerse a través de los distintos artefactos generados en cada etapa.

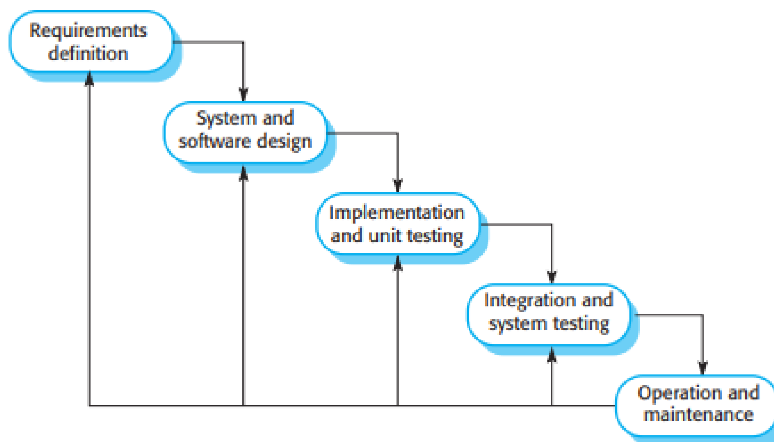
5.3.2 Modelos aplicables al ciclo de vida del desarrollo de software

Los modelos aplicables al ciclo de vida del desarrollo de software representan diferentes enfoques para organizar y gestionar las actividades fundamentales del desarrollo. Los principales modelos incluyen: el modelo en cascada (*Waterfall Model*), el desarrollo incremental, el desarrollo basado en reutilización (*Reuse-oriented Development* o *Integration and Configuration Model*) y los métodos ágiles (*Agile Methods*) [12]. El modelo de cascada proporciona la base estructural sobre la cual se implementará la estrategia de trazabilidad propuesta en este trabajo, permitiendo registrar y relacionar elementos de seguridad en cada una de estas etapas explicadas en la sección siguiente.

5.3.3 Modelo en cascada

El modelo en cascada es un enfoque clásico en el desarrollo de software estructurado y secuencial que organiza las actividades del ciclo de vida, como la especificación de requisitos, el diseño del sistema, la implementación y las pruebas, la integración y las pruebas del sistema, y el mantenimiento, en etapas bien definidas que deben completarse una tras otra, sin solapamientos [12]. La Figura 1, presenta las etapas del modelo en cascada.

Figura 1. Etapas del modelo en cascada



Fuente: Modelo en cascada propuesto en [12].

En el modelo en cascada se parte del supuesto de que todos los requisitos pueden definirse completamente desde el inicio y que es posible planificar el desarrollo con precisión, por lo que es especialmente adecuado para sistemas críticos o altamente regulados, como los embebidos o los de uso aeroespacial, donde la trazabilidad, la validación formal y la documentación exhaustiva son fundamentales.

Sin embargo, este modelo es poco flexible frente a cambios, ya que no permite la retroalimentación temprana del usuario ni facilita la adaptación de requisitos durante el desarrollo, lo cual puede generar costos elevados por retrabajo en etapas posteriores. Por ejemplo, si en la etapa de requisitos no se especifica que las consultas a la base de datos deben usar mecanismos seguros, como consultas preparadas, es posible que en etapas finales se detecten vulnerabilidades por inyección SQL. En el modelo en cascada, corregir esto implicaría modificar el diseño, el código y volver a probar, lo que genera retrasos y altos costos.

A pesar de sus limitaciones, el modelo en cascada sigue siendo útil en proyectos donde se requiere una planificación rigurosa, donde los requisitos son estables y bien entendidos desde el comienzo, y donde los estándares de calidad y cumplimiento normativo exigen una documentación formal en cada etapa del proceso.

5.3.3.1 Etapas del modelo en cascada

Las etapas del modelo en cascada reflejan directamente las actividades fundamentales del desarrollo de software, según [12] son:

- **Definición de requisitos**: En esta etapa se determinan, en colaboración con los usuarios del sistema, los servicios, restricciones y objetivos que debe cumplir. Luego, estos elementos se detallan con precisión y se documentan como la especificación del sistema.
- **Diseño del sistema y del software**: En esta etapa, se asignan los requisitos identificados al hardware o al software, y se establece la arquitectura general del sistema. El diseño del software consiste en identificar y describir las abstracciones fundamentales del sistema y cómo se relacionan entre sí.
- **Implementación y pruebas unitarias**: En esta etapa, el diseño del software se transforma en programas o unidades de programa concretas. Las pruebas unitarias tienen como objetivo verificar que cada unidad individual cumpla con su especificación.
- **Integración y pruebas del sistema**: En esta etapa, las distintas unidades de programa se integran para formar el sistema completo, que luego se somete a pruebas para asegurar que cumple con los requisitos establecidos. Una vez superadas estas pruebas, el sistema se entrega al cliente.
- **Operación y mantenimiento**: Esta suele ser la etapa más extensa del ciclo de vida. El sistema se instala y se utiliza en un entorno real. El mantenimiento abarca la corrección de errores no detectados en etapas anteriores, la mejora de las unidades implementadas y la incorporación de nuevas funcionalidades a medida que surgen nuevos requisitos.

En este trabajo, se adopta el modelo en cascada por su enfoque estructurado y secuencial, el cual permite una planificación clara y una documentación detallada en cada etapa del desarrollo. Esta organización facilita la revisión formal de los entregables, lo que resulta especialmente útil cuando se abordan requisitos críticos como los de seguridad. Sin embargo, el modelo carece de mecanismos formales que aseguren el enlace y la trazabilidad de los requisitos de seguridad a lo largo de sus distintas etapas, por lo cual se desarrolla la estrategia presentada en este trabajo.

5.4 Conceptos relacionados con modelado

Esta sección presenta distintos tipos de diagramas utilizados para representar la estructura, el comportamiento y la distribución de un sistema, incluyendo el diagrama de clases, el diagrama de despliegue y el diagrama de secuencia.

5.4.1 Modelado

El modelado en el contexto del software es la representación estructurada de los elementos de un sistema. El propósito del modelado es permitir abstraer detalles y enfocarse en la estructura general del sistema, sus componentes, conectores y sus interacciones, facilitando así el análisis, evolución y reutilización del sistema a lo largo del ciclo de vida del desarrollo de software [13]. Para este trabajo se usará el diagrama de clases, diagrama de despliegue y el diagrama de secuencia, estas representaciones contribuyen a establecer las relaciones explícitas entre los requisitos funcionales y no funcionales de seguridad y los componentes técnicos, fortaleciendo la trazabilidad entre el diseño del sistema y las decisiones orientadas a su protección.

5.4.2 Diagrama de clases

El diagrama de clases UML es una representación estructural que describe las clases dentro de un sistema y las relaciones entre ellas. Este diagrama representa atributos, operaciones (métodos) y asociaciones que existen entre diferentes clases, permitiendo capturar la estructura estática del software [12]. El diagrama de clases es usado dentro de esta estrategia de trazabilidad propuesta, debido a que permite establecer una representación estructurada

de las entidades del sistema y sus relaciones, facilitando la vinculación directa entre los requisitos funcionales y de seguridad con los componentes del diseño técnico.

5.4.3 Diagrama de despliegue

El diagrama de despliegue UML es la representación gráfica que define la distribución física de los componentes del sistema sobre recursos computacionales concretos, tales como servidores, contenedores o plataformas en la nube [14]. En el contexto de este trabajo, este diagrama sirve para validar que las decisiones tomadas en la etapa de diseño sigan presentes y sigan siendo coherentes al momento de implementar el sistema en un entorno desplegado.

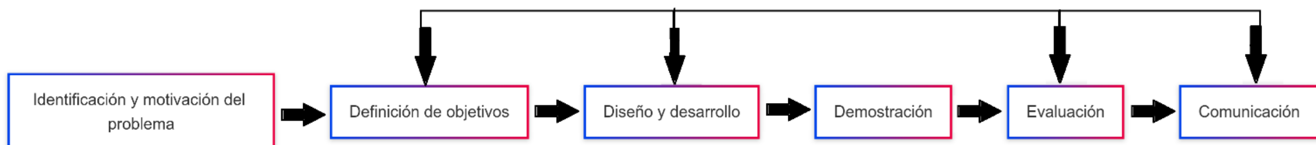
5.4.4 Diagrama de secuencia

Un diagrama de secuencia UML es usado para modelar de manera secuencial, las interacciones entre los objetos y los actores en un sistema. Se usa principalmente para representar cómo se comunican los distintos componentes o actores del sistema a lo largo del tiempo, en respuesta a un escenario o caso de uso [12]. En este trabajo, el diagrama de secuencia permite evidenciar cómo los requisitos funcionales se ejecutan dentro de un flujo controlado y verificable, y cómo los requisitos no funcionales de seguridad se integran de forma precisa en los momentos clave del proceso, como la validación de entradas o la protección contra accesos indebidos.

6 METODOLOGÍA

Para realizar el presente trabajo se adopta la metodología *Design Science Research* (DSR), propuesta por Peffers *et al.* [15], la cual se organiza en etapas secuenciales que permiten generar una solución a problemas identificados en el campo de la ingeniería de software. La estructura que ofrece este método proporciona mecanismos para construir, analizar y validar un artefacto tecnológico orientado a garantizar la trazabilidad de elementos de seguridad en cada etapa del ciclo de desarrollo del software basado en el modelo en cascada. La metodología seleccionada consta de las siguientes etapas representadas en la Figura 2.

Figura 2. Metodología Design Science Research



Fuente: Tomado y adaptado de [15]

6.1 Identificación y motivación del problema

Esta etapa consiste en reconocer una situación problemática que afecta un entorno determinado. Se identifican causas, consecuencias y posibles implicaciones técnicas, organizativas o sociales. El objetivo es delimitar con claridad la necesidad de intervención, justificando la pertinencia de generar una solución tecnológica.

En entornos donde se adopta el modelo en cascada para el desarrollo de software, la ausencia de mecanismos que aseguren la trazabilidad de los requisitos de seguridad a lo largo del ciclo de vida del desarrollo de software representa un riesgo significativo. Debido a la naturaleza secuencial del modelo, los requisitos definidos en etapas tempranas tienden a quedar fijos y aislados, lo que dificulta su seguimiento, revisión o validación en etapas posteriores como el diseño, la implementación o las pruebas. Esta falta de trazabilidad puede dar lugar a la omisión de controles críticos, la aparición de vulnerabilidades, y la dificultad posterior de justificar decisiones técnicas en auditorías o procesos de certificación.

La identificación y la justificación del trabajo se exponen en la Sección 2 y en la Sección 3 respectivamente.

6.2 Definición de objetivos

Esta etapa tiene como propósito establecer los objetivos del trabajo, tanto los generales como los específicos, donde se oriente el diseño, la implementación y la validación del sistema propuesto. Cada uno de estos objetivos debe responder de manera directa a la necesidad identificada y ofrecer un criterio de referencia para evaluar los resultados alcanzados.

El objetivo general de este trabajo es implementar una estrategia de trazabilidad de los elementos de seguridad para cada etapa del ciclo de vida del desarrollo de software, usando el modelo en cascada, que facilite la documentación y el seguimiento de decisiones técnicas orientadas a la protección del sistema desde la definición de los requisitos hasta el despliegue.

Cabe destacar que los objetivos (ver Sección 4) se depuraron y ajustaron conforme avanzaron las actividades de diseño y desarrollo. Este refinamiento continuo, posible gracias a la metodología adoptada, permitió responder con mayor precisión a los hallazgos emergentes y asegurar la pertinencia de cada meta frente a los desafíos identificados.

6.3 Diseño y desarrollo

Esta etapa corresponde a la construcción de la solución propuesta. Se definen los componentes técnicos, modelos, procesos, herramientas y procedimientos que conforman el trabajo. También se establecen relaciones entre dichos elementos y las etapas del proyecto, asegurando su coherencia con los objetivos definidos.

En este trabajo se propone trazar cada requisito, en especial los de seguridad, en cada etapa del ciclo de vida del desarrollo de software siguiendo el modelo en cascada, permitiendo hacer seguimiento de las decisiones relacionadas con la seguridad a lo largo del ciclo de vida.

El diseño y el desarrollo de la propuesta se fueron mejorando conforme a los hallazgos obtenidos durante las etapas de Demostración y Evaluación, esto gracias al refinamiento progresivo, propio de esta metodología iterativa empleada. El desarrollo de este trabajo se expone a profundidad en la Sección 7.

6.4 Demostración

Esta etapa busca poner en funcionamiento la solución en condiciones controladas. El propósito es mostrar que el sistema diseñado cumple con las funciones esperadas y puede operar dentro de escenarios realistas. La demostración permite verificar la aplicabilidad del mecanismo en contextos similares al que motivó su diseño.

En este trabajo se realiza una demostración de la aplicación de la estrategia que presenta etapa por etapa, cómo se pone en práctica la metodología propuesta. Esta aplicación se encuentra en la Sección 8.

6.5 Evaluación

En esta etapa se analiza el desempeño de la solución propuesta. Se recogen evidencias, se aplican métricas o criterios de análisis y se examina la relación entre los objetivos formulados y los resultados obtenidos. El propósito es identificar el grado de utilidad, consistencia y aplicabilidad del sistema.

La evaluación de la estrategia propuesta se realiza mediante la comparación entre el modelo en cascada sin trazabilidad explícita y el modelo en cascada con la estrategia de trazabilidad propuesta en este trabajo. Gracias al uso de esta metodología iterativa se permitió refinar detalles relacionados con la evaluación, a medida que se avanzaban en las anteriores etapas. Esta evaluación se presenta a detalle en la Sección 9 del documento.

6.6 Comunicación

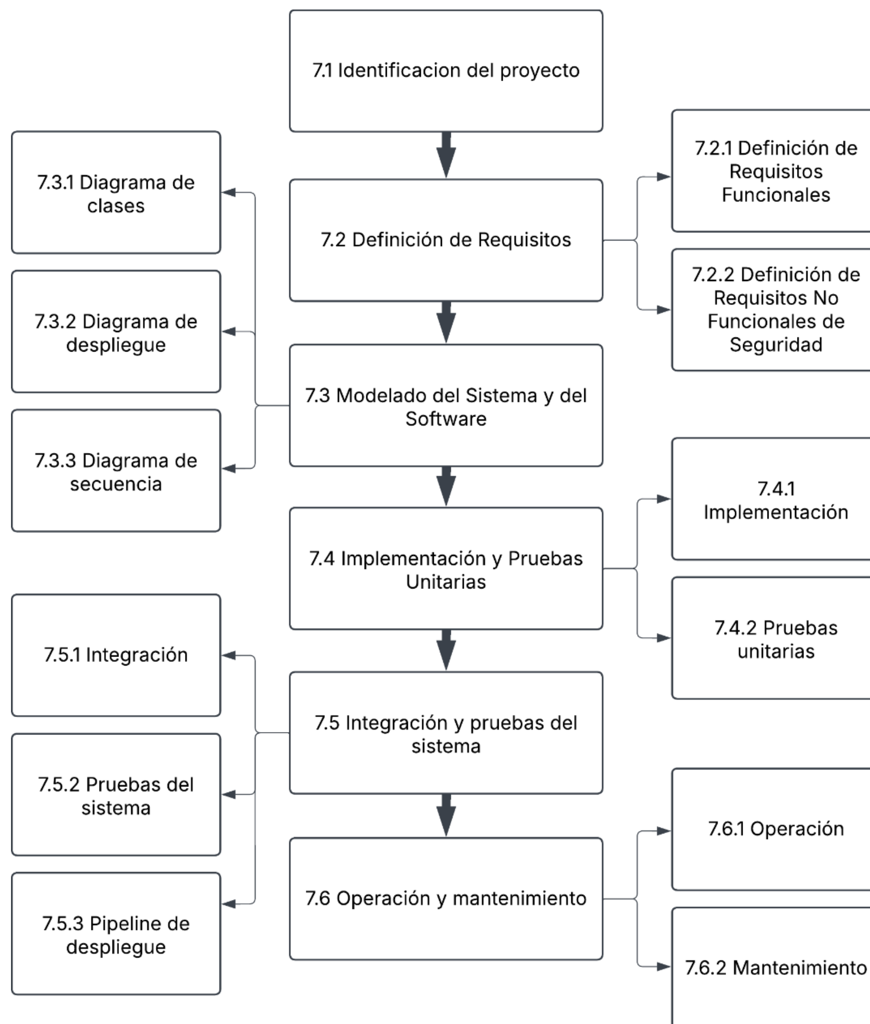
Esta etapa tiene como finalidad organizar y presentar los resultados obtenidos. La información se estructura en un documento que permite entender cómo se construyó, aplicado y evaluado el sistema. Se espera que esta documentación sea útil para replicar o adaptar la solución en contextos diferentes.

En este trabajo, la documentación completa del proceso desarrollado se presenta en este documento. Se incluyen las etapas del proceso, los fundamentos teóricos, las decisiones técnicas adoptadas y los resultados observados, con el fin de facilitar su consulta en proyectos similares.

7 DESARROLLO DEL TRABAJO

En esta sección se presenta la estrategia que permite incorporar trazabilidad de los elementos de seguridad a lo largo del ciclo de vida del software, siguiendo la lógica secuencial del modelo en cascada. En la Figura 3 se presenta cada etapa organizada de manera jerárquica para permitir una ejecución ordenada y controlada.

Figura 3. Estructura secuencial para la trazabilidad de requisitos de seguridad del software



Fuente: elaboración propia.

Se adopta la estructura secuencial definida en la Figura 3 para garantizar que los requisitos definidos, en especial los no funcionales de seguridad, se alineen con los entregables definidos en cada etapa del modelo en cascada. En primer lugar, la **identificación del proyecto** permite anticipar necesidades generales específicas, tanto funcionales como de seguridad según el

tipo de sistema, su entorno de ejecución y sus actores relevantes. Posteriormente, en la **definición de requisitos**, se establecen los requisitos funcionales y no funcionales, especialmente aquellos relacionadas con la seguridad, que orientan las decisiones desde el inicio del desarrollo. En la etapa de **modelado del sistema y del software**, los requisitos se vinculan con componentes específicos del sistema, utilizando representaciones como diagramas de clases, secuencia y despliegue que permiten trazar la lógica funcional y las medidas de protección previstas. Durante la etapa de **implementación y pruebas unitarias**, se integran directamente los mecanismos de seguridad en el código fuente, y se diseñan pruebas que permiten verificar individualmente la funcionalidad y la eficacia de los controles definidos, en la integración y pruebas del sistema, se valida que todos los componentes trabajen de forma coordinada y segura, garantizando que la interacción entre componentes no comprometa los objetivos de seguridad establecidos. Finalmente, en la etapa de **operación y mantenimiento** se contempla la ejecución segura del sistema en un entorno real, con monitoreo continuo, aplicación de actualizaciones, análisis de vulnerabilidades y revisión periódica de los mecanismos implementados, asegurando que la trazabilidad de seguridad se mantenga vigente a lo largo del tiempo.

Además de las etapas anteriormente nombradas, se encuentra la etapa **identificación del proyecto** que no es explicita en el modelo en cascada. Esta se incorpora en la estrategia, con el fin de ofrecer un entendimiento previo del entorno, los actores involucrados y las condiciones operativas, sirviendo como base para la definición de requisitos y facilitando una trazabilidad más precisa a lo largo de todo el ciclo de vida del desarrollo de software.

A continuación, se detallará cada etapa de la estrategia propuesta, mostrando cómo se asegura la trazabilidad de los elementos clave que las integran. Pasando por cada una de las etapas se propondrán, seleccionarán y definirán elementos claves propios de cada etapa para ser usadas a favor del seguimiento de lo decidido en las etapas de identificación y de requisitos, en especial aquellas decisiones relacionadas con la seguridad.

7.1 Identificación del proyecto

El proyecto sobre el cual se implementa esta estrategia debe definirse y contextualizarse desde antes de la definición de requisitos, con el fin de establecer una base que oriente adecuadamente el ciclo de vida del desarrollo. Este proyecto debe tener en consideración las particularidades propias de un proyecto que se desarrolle bajo el modelo cascada, como la necesidad de una documentación exhaustiva, validaciones estrictas, cumplimiento de normas o leyes, y una necesaria planificación a largo plazo, donde los cambios durante el desarrollo resulten altamente costosos.

En esta etapa, de identificación del proyecto, se establecen aspectos clave como el tipo de aplicación, sus funcionalidades generales, los actores involucrados, el entorno tecnológico y operativo, así como algunas de las condiciones específicas que puedan influir en decisiones técnicas y de seguridad. Esta definición inicial no solo delimita el alcance del sistema, sino que también permite anticipar necesidades críticas de seguridad y preparar el terreno para una formulación estructurada, verificable y trazable de los requisitos funcionales y no funcionales. En la Tabla 1 se presenta una guía básica que puede emplearse para replicar en proyectos similares, mostrando su respectivo aporte a la trazabilidad de la seguridad del proyecto y del sistema.

Tabla 1. Guía para la identificación del proyecto con sus ventajas para la trazabilidad en seguridad

Elemento por definir	**Actividad**	**Aportes a la trazabilidad de seguridad**
Tipo de aplicación	Identificar si es web, móvil, embebida, IoT, etc.	Ayuda a prever vulnerabilidades (ej. Inyecciones SQL en formularios web)
Funcionalidades principales	Determinar funciones clave del proyecto	Permite anticipar puntos críticos de validación de entrada y control de acceso
Tecnologías seleccionadas	Definir *stack* tecnológico e infraestructura	Identifica herramientas de análisis compatibles (ej. *SonarQube*) y posibles configuraciones seguras
Entorno operativo	Establecer si es local, en la nube, híbrido, con qué servicios externos se conecta	Delimita superficie de exposición y dependencias externas
Actores involucrados	Listar perfiles de usuario, servicios externos, administradores	Informa sobre necesidades de autenticación/autorización y *logging*
Amenazas preliminares identificadas	Identificar posibles amenazas comunes para el tipo de aplicación	Permite definir requisitos no funcionales de seguridad específicos

Fuente: elaboración propia

7.2 Definición de Requisitos

En esta etapa se lleva a cabo la identificación, clasificación y documentación de los requisitos necesarios para el funcionamiento del sistema. Estos se dividen en dos categorías principales: los requisitos funcionales, que describen las acciones específicas que debe realizar el sistema, y los requisitos no funcionales, que establecen las propiedades y restricciones del sistema, definiendo las condiciones bajo las cuales debe operar. En el marco de esta estrategia, en los requisitos no funcionales, los que se abordarán son los de seguridad, los cuales especifican las condiciones necesarias para proteger la información, prevenir vulnerabilidades y asegurar un comportamiento confiable frente a posibles amenazas.

7.2.1 Definición de requisitos funcionales

En esta parte de la etapa de definición se identifican las funcionalidades esenciales que el sistema debe ofrecer para cumplir con sus objetivos operativos. Cada requisito funcional describe una acción concreta esperada del sistema. Cada funcionalidad se formula de manera específica y medible, lo que facilita su implementación, validación técnica y seguimiento a lo largo del ciclo de vida del software.

Estos requisitos se deben basar en lo que se definió en la Tabla 1, en la sección de funcionalidades principales. De esta forma se podrá hacer seguimiento de lo que se determinó en la etapa de identificación.

Cada requisito se identifica mediante un código único y se secuencia en el orden en que se define, lo que permite rastrear su cumplimiento y facilita la verificación técnica. Para la definición de requisitos se sugiere usar la estructura que se encuentra en la Tabla 2, que está inspirada en lo que establece [16], donde el autor propone una plantilla base para escribir requisitos funcionales desde las funcionalidades del sistema.

Tabla 2. Guía para la redacción de Requisitos Funcionales

Elemento	Descripción	Ejemplo
Código único del requisito	Identificador corto y específico que permite distinguir cada requisito de los demás. Se recomienda usar el formato RF-#, donde # es un número consecutivo.	RF-1
Nombre del requisito	Título breve que resume la función que el sistema debe realizar. Debe ser representativo y permitir una lectura rápida.	Obtener todas las publicaciones
Descripción detallada del requisito	Explica qué debe hacer el sistema. El sistema debe permitir a [el actor que interviene] [acción que se ejecuta] [elemento sobre el cual se ejecuta] [reglas o condiciones que se aplican] **Nota:** El campo que esté entre corchetes se debe reemplazar con la información real del requisito	El sistema debe permitir al usuario obtener todas las publicaciones, ordenando las publicaciones por fecha de creación en orden descendente.

Fuente: elaboración propia

Después de aplicar la estructura de la Tabla 2, tendríamos como resultado un requisito de esta forma: **RF-1 (Obtener todas las publicaciones):** El sistema debe permitir a los usuarios obtener todas las publicaciones, ordenando las publicaciones por fecha de creación en orden descendente.

7.2.2 Definición de requisitos no funcionales de seguridad

En esta etapa se identifican los requisitos no funcionales que el sistema debe cumplir para garantizar la seguridad de la información y la estabilidad operativa. Se definen condiciones técnicas específicas que permitan prevenir ataques, proteger datos sensibles y asegurar la disponibilidad continua del servicio.

Los requisitos no funcionales de seguridad pueden incluir prácticas como la sanitización de entradas, la validación de datos y el manejo adecuado de errores. Cada medida se describe de manera clara para facilitar su implementación y verificación a lo largo del ciclo de vida del desarrollo. Estos requisitos deberían ser basados en lo que se definió en la Tabla 1, en la sección de amenazas preliminares identificadas. De esta forma se podrá hacer seguimiento de lo que se determinó en la etapa de identificación.

Cada requisito no funcional de seguridad se identifica mediante un código único y se secuencia en el orden en que se define, lo que permite rastrear su cumplimiento en todas las etapas del desarrollo y mantener su vinculación directa con los controles implementados. La

estructura para la definición de requisitos no funcionales de seguridad presentada en la Tabla 3 es una guía, inspirada en la plantilla de requisitos de seguridad sugerida por [17].

Tabla 3. Guía para la redacción de Requisitos No Funcionales de Seguridad

Elemento	**Descripción**	**Ejemplo**
Código único del requisito de seguridad	Identificador breve y único que diferencia este requisito de los demás. Se recomienda el uso del formato RNS-#, donde # representa un número secuencial. Este código permite localizar el requisito en pruebas, archivos de implementación o matrices de verificación.	RNS-1
Nombre del requisito de seguridad	Título que resume la medida de protección requerida. Debe describir de forma breve qué aspecto del sistema será protegido o validado.	Sanitización del filtrado de las publicaciones
Descripción del requisito	Esta estructura indica cómo se protege el sistema. El sistema debe asegurar que [elemento a asegurar] [mecanismo] [objetivo qué se busca al aplicar este mecanismo]. **Nota:** El campo que esté entre corchetes se debe reemplazar con la información real del requisito	El sistema debe asegurar que los parámetros de entrada del filtrado de las publicaciones sean sanitizados, para prevenir inyecciones SQL o XSS.

Fuente: elaboración propia

Después de aplicar la estructura de la Tabla 3, tendríamos como resultado un requisito de esta forma: **RNS-1 (Sanitización del filtrado de las publicaciones):** El sistema debe asegurar que los parámetros de entrada del filtrado de las publicaciones sean sanitizados, para prevenir inyecciones SQL o XSS.

7.3 Modelado del sistema y del software

El modelado del sistema y del software permite representar de manera estructurada la arquitectura lógica del sistema, facilitando la vinculación de cada requisito con los elementos técnicos que lo implementan.

Para el desarrollo de esta propuesta se usará el diagrama de clases, que representa las estructuras lógicas y relaciones internas del sistema. Este diagrama es útil para validar que lo decidido en los requisitos siga estando presente y coherente, ya que permite asociar funcionalidad y controles de seguridad directamente con clases específicas del diseño. Además, se usará el diagrama de secuencia, que ilustra dinámicamente la interacción entre los objetos a lo largo del tiempo durante la ejecución de funcionalidades clave. Finalmente, se usará el diagrama de despliegue, que muestra cómo se distribuyen físicamente los componentes en la infraestructura.

Estos diagramas cumplen una doble función en la propuesta: primero, proporcionan una representación estructurada de la arquitectura del sistema; segundo, facilitan la trazabilidad al vincular cada requisito, en especial los de seguridad, con los elementos que lo implementan.

7.3.1 Diagrama de clases

El diagrama de clases describe la estructura de un sistema mostrando las clases del sistema, sus atributos, operaciones y las relaciones entre las clases. Su uso es especialmente útil en esta propuesta para la trazabilidad de requisitos, ya que permite vincular cada requisito funcional o de seguridad con una o más clases responsables de su implementación.

Para construir el diagrama de clases, se recomienda seguir este orden:

1. **Se definen las clases:** Basado en la Tabla 2, para cada requisito funcional localice el fragmento entre corchetes identificado como “elemento sobre el cual se ejecuta” (la acción), este elemento puede ser una clase.

   Nota: Los nombres de las clases se definen en singular. Si varias clases comparten el mismo elemento sobre el cual se ejecuta, es decir, tienen el mismo nombre, estas son la misma clase.

2. Se define una **clase de seguridad** que se puede llamar *SecurityManager*.
3. **Se definen los métodos:**
   a. Basado en la Tabla 2, para cada requisito funcional, localice el fragmento entre corchetes identificado como “acción que se ejecuta”, posiblemente esta acción sea un método de la clase que se definió en ese requisito funcional.

      Nota: Los nombres de los métodos deben estar en infinitivo.

   b. Basado en la Tabla 3, para cada requisito no funcional de seguridad, localice el fragmento entre corchetes identificado como “mecanismo de seguridad”, posiblemente este mecanismo sea un método de la clase de seguridad.

Nota: Si un mecanismo de seguridad se repite en distintos requisitos de seguridad, basta con definir un único método y reutilizarlo para los otros requisitos no funcionales que lo usen, con el fin de no repetir lógica.

4. **Se definen los atributos:** A partir de la Tabla 2, para cada requisito funcional, localice el fragmento entre corchetes identificado como "reglas o condiciones que se aplican". Ese fragmento puede interpretarse de dos maneras:
    a. **Atributos de la clase:** cuando las reglas o condiciones describen características, calificativos o adjetivos del elemento, estas se modelan como atributos de la clase definida en el requisito.
    b. **Lógica de los métodos:** cuando las reglas o condiciones corresponden a acciones o restricciones de comportamiento, se plasman en la lógica de los métodos asociados a la acción especificada en el requisito.
5. Por cada clase definida el paso 1 se crea otra clase con el mismo nombre, añadiéndole el sufijo *Controller*, a esta clase se le añaden los mismos métodos que se definieron en la clase anterior.
   Nota: Los nombres de los métodos en estas nuevas clases pueden cambiar a discreción del diseñador de este diagrama.
6. **Se definen las relaciones entre las clases:** Primero se enlazan las clases definidas en el paso 1, con las clases que llevan el sufijo *Controller* creadas en el paso 5; después, cada clase *Controller* se conectan con la clase de seguridad *SecurityManager*.
   Nota: Las relaciones pueden cambiar a discreción del diseñador de este diagrama.

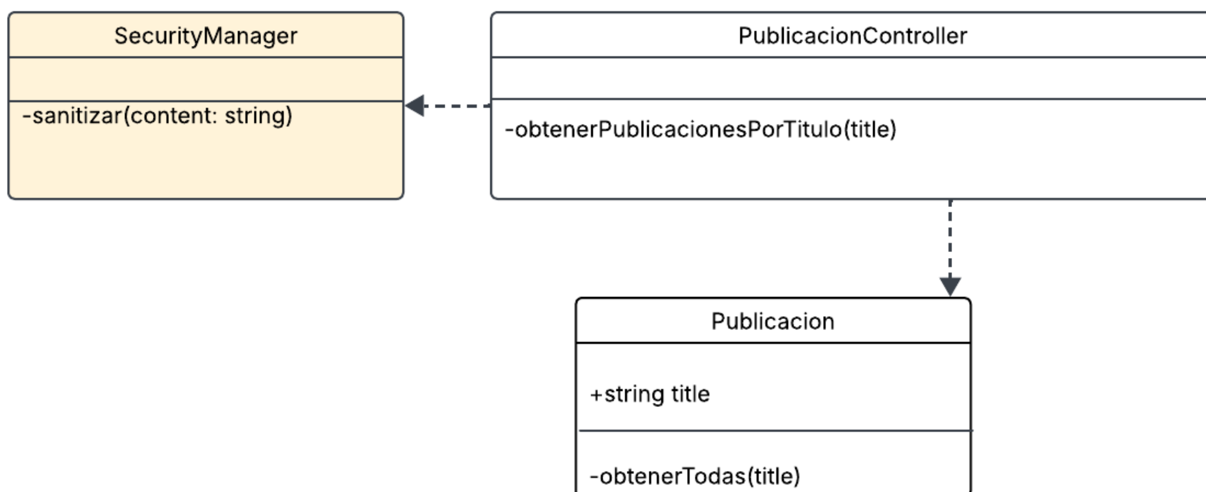
Para esta propuesta, se usarán las capas Modelo y Controlador, inspirado del patrón Modelo-Vista-Controlador (MVC) descrito por [12], con el propósito facilitar la organización del sistema al separar claramente la lógica de interacción, el manejo de datos y los aspectos de seguridad. Las clases Controlador se encargan de la lógica de interacción del usuario y las clases Modelo se encargan del manejo de los datos. Siguiendo este patrón, llamaremos a las clases nombradas en el paso 1 del proceso de construcción del diagrama, clases Modelo, las

definidas en el paso 5, clases Controlador y la clase *SecurityManager* como una clase Controlador.

En la Figura 4 se presenta un ejemplo si se quisiera crear un diagrama basado un requisito funcional y un requisito no funcional de seguridad:
RF-1 (Obtener todas las publicaciones filtradas por título): El sistema debe permitir al usuario filtrar las publicaciones por título, ordenando las publicaciones por fecha de creación en orden descendente.

RNS-1 (Sanitización del filtrado de las publicaciones): El sistema debe asegurar que los parámetros de entrada del filtrado de las publicaciones sean sanitizados, para prevenir inyecciones SQL o XSS.

Figura 4. Ejemplo de un diagrama de clases representando los métodos que realizan las acciones de los requisitos en cada clase



Fuente: elaboración propia.

La Tabla 4 proporciona una referencia de trazabilidad entre los requisitos y los elementos del Controlador y del Modelo, permitiendo identificar con precisión las clases donde se implementan estas funcionalidades y sus respectivos controles de seguridad. Esta tabla permite asegurar que lo definido en la etapa de requisitos se haya reflejado de forma concreta en el diagrama de clases, debido a que permite trazar el requisito con las clases Controlador y Modelo que intervienen en su cumplimiento.

Tabla 4. Guía para la trazabilidad de requisitos a elementos del diagrama de clases

Id requisito	**Tipo de requisito**	**Elemento del diagrama de clases**
Código único del requisito (Nombre del requisito)	Funcional	Clase: Clase controlador donde se encuentra la funcionalidad del requisito. Método: El método especifico dentro de la clase que realiza la acción del requisito. Clase: Clase modelo que está relacionada con este requisito. Método: Método que usa la clase modelo para la acción especifica.
Código único del requisito no funcional de seguridad (Nombre del requisito no funcional)	Funcional de Seguridad	Clase: Clase controlador donde se encuentra la funcionalidad del requisito al que se va a aplicar el método de seguridad. Método: El método especifico dentro de la clase que realiza la acción del requisito al que se va a aplicar el método de seguridad. Clase: Clase *SecurityManager* que interviene en la seguridad del requisito Método: El método especifico dentro de la clase que interviene en la seguridad del requisito

Fuente: elaboración propia.

7.3.2 Diagrama de secuencia

Un diagrama de secuencia muestra las interacciones de los procesos ordenadas en el tiempo. Este diagrama representa los procesos y objetos involucrados, así como la secuencia de mensajes intercambiados según sea necesario para llevar a cabo la función. En esta propuesta adquiere un valor adicional al facilitar la trazabilidad de los requisitos y en especial los requisitos no funcionales de seguridad ya que permite observar de manera explícita cómo se integran los mecanismos de seguridad dentro del flujo funcional, destacando las interacciones entre clases que aseguran el cumplimiento de medidas de seguridad.

Este diagrama se desarrolla usando como base las clases, métodos, atributos y relaciones definidos en el diagrama de clases anterior. Este diagrama ilustra el flujo de los requisitos funcionales y cómo cada clase se relaciona con las demás para realizar las funcionalidades que se especificaron en el requisito. Al mismo tiempo, se muestra en qué punto del flujo se invocan los métodos de seguridad definidos en la clase de seguridad.

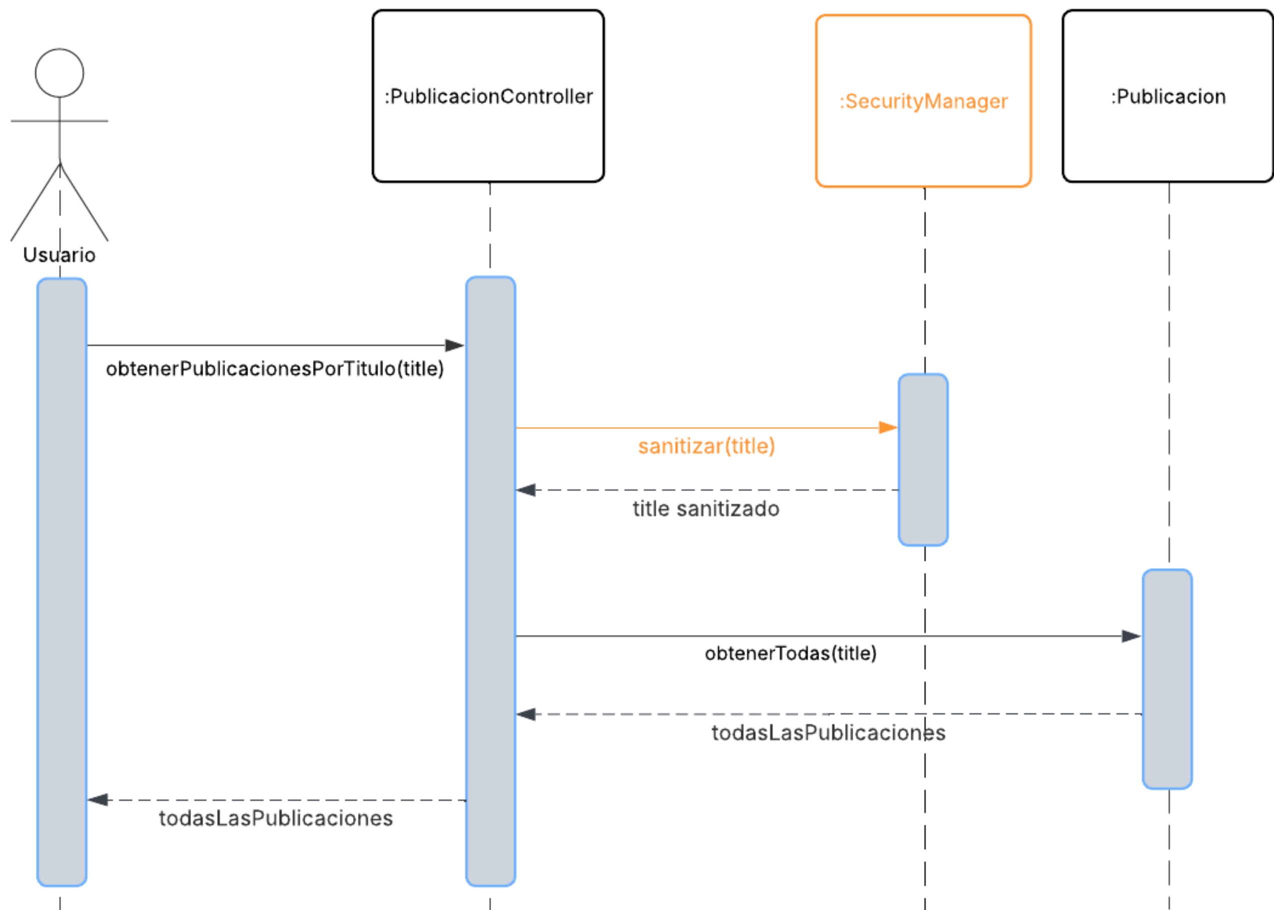
Basado en la Tabla 2, donde se estableció la guía para los requisitos funcionales, localice el fragmento de la estructura situado entre corchetes, identificado como el actor que interviene. Este actor ejecuta una acción que será la que comienza el flujo de este diagrama, a partir de

esta acción se ilustra cómo la clase controladora interviene con el método que se invocó. Posterior a esta primera acción, se traza la secuencia de mensajes que se propagan desde el controlador hacia las demás clases necesarias para cumplir con el requisito (por ejemplo, la clase de seguridad *SecurityManager*, otro controlador o la clase modelo que administra el acceso a la información), representando cada participante con una línea vertical. Emplee flechas sólidas para las invocaciones y flechas discontinuas para las respuestas, etiquetando cada interacción con el nombre del método y los parámetros relevantes. Si hay reglas o validaciones que aplicar, añada anotaciones junto a los mensajes. Finalmente, muestra el retorno de la información al actor en orden inverso, de modo que el diagrama refleje de manera clara y comprensible todo el flujo descrito por el requisito funcional, sin depender de nombres concretos de clases o métodos.

La Figura 5 se presenta un ejemplo si se quisiera crear un diagrama basado un requisito funcional y un requisito no funcional de seguridad:
RF-1 (Obtener todas las publicaciones filtradas por título): El sistema debe permitir al usuario filtrar las publicaciones por título, ordenando las publicaciones por fecha de creación en orden descendente.

RNS-1 (Sanitización del filtrado de las publicaciones): El sistema debe asegurar que los parámetros de entrada del filtrado de las publicaciones sean sanitizados, para prevenir inyecciones SQL o XSS.

Figura 5. Ejemplo de un diagrama de secuencia



Fuente: elaboración propia.

La Figura 5 muestra en cómo un usuario invoca a PublicacionController, solicitando las publicaciones cuyo título coincide con el texto ingresado; antes de consultar al modelo Publicacion, el controlador envía el parámetro a *SecurityManager* para sanitizarlo y prevenir ataques (por ejemplo, inyección SQL o XSS); una vez devuelto el título depurado, el controlador invoca a Publicacion para recuperar las entradas correspondientes, recibe la lista resultante y se la entrega al usuario.

7.3.3 Diagrama de despliegue

El diagrama de despliegue representa la distribución física de los componentes del sistema dentro de sus diferentes entornos de ejecución. Este diagrama especifica cómo los módulos de software se instalan sobre la infraestructura física y cómo se interconectan entre sí,

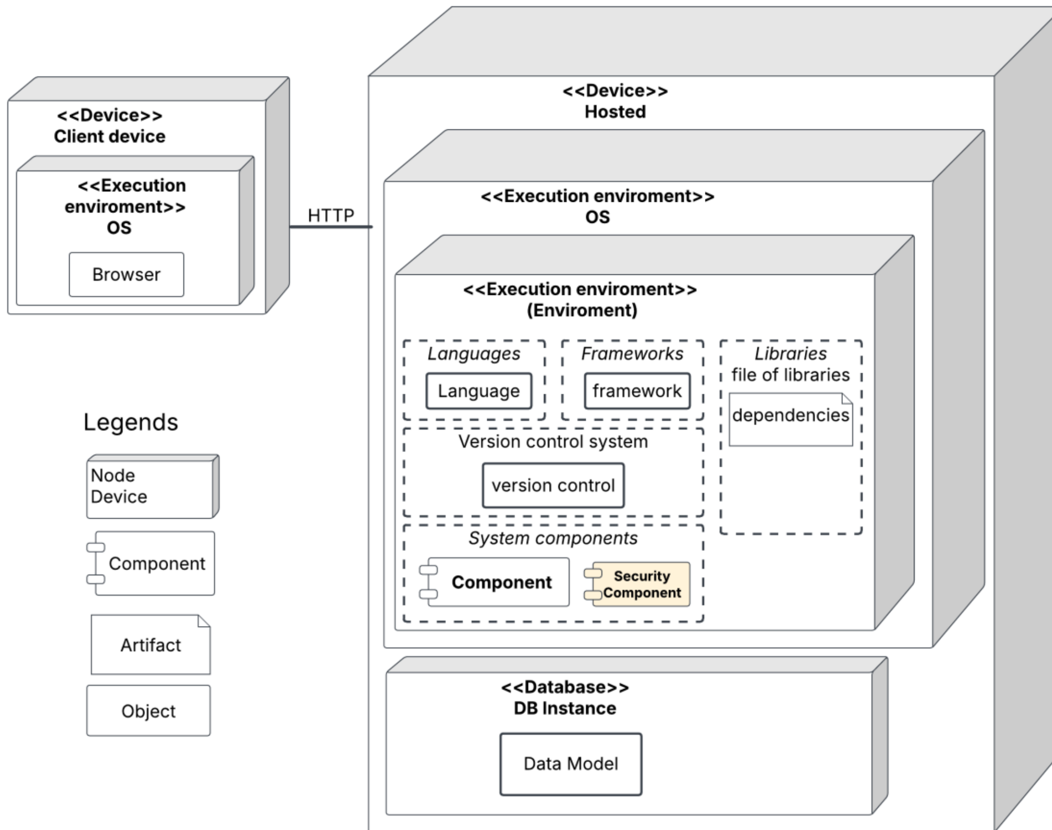
reflejando el entorno operativo establecido durante la etapa de identificación del proyecto y de definición de requisitos.

Este diagrama es relevante para la trazabilidad de los requisitos, ya que permite identificar dónde y cómo se despliegan las funcionalidades del sistema, asegurando la correspondencia entre diseño y la ejecución. Además, facilita la trazabilidad de los elementos de seguridad, al permitir mapear los controles definidos.

Basado en la Tabla 1 de la primera etapa, identificación del proyecto, donde se definieron los elementos principales que tendrá el proyecto, se usarán cada uno de estos elementos para definir la arquitectura de despliegue. Comenzando por el tipo de aplicación, se podrá definir que tecnologías se usarán, y cuáles son los dispositivos que intervendrán, tanto del lado del cliente, como del entorno de ejecución. Posteriormente, basado en el diagrama de clases elaborado en los pasos anteriores, se establecerán los modelos para el manejo de datos y los componentes del sistema, estos últimos basados en las clases controladoras.

La Figura 6 presenta esta distribución garantizando que las decisiones tomadas durante la identificación del proyecto, la definición de requisitos y el diseño queden reflejadas en el entorno de ejecución real. Se muestran dos dispositivos principales: el cliente, desde donde el usuario realiza las solicitudes, y el entorno alojado de la aplicación, que ejecuta la lógica de negocio del sistema. Este entorno incluye un sistema operativo anfitrión sobre el cual se despliega el entorno de ejecución, compuesto por lenguajes, *frameworks*, bibliotecas y herramientas necesarias, entre las cuales se incorpora un componente de seguridad, que contiene las clases ya definidas, de un color distintivo para facilitar su trazabilidad. Finalmente, en la parte inferior se encuentra la instancia de base de datos, responsable de la persistencia y el acceso seguro a la información, reforzando la continuidad de la trazabilidad al vincular los modelos de datos con los controles implementados en el sistema.

Figura 6. Ejemplo de un diagrama de despliegue



Fuente: elaboración propia

Nota: Este diagrama puede variar de acuerdo con la arquitectura del proyecto a discreción del diseñador.

7.4 Implementación y pruebas unitarias

La etapa de implementación constituye un punto crítico dentro del ciclo de vida del desarrollo de software, ya que implica la transición del diseño a un sistema desarrollado. En esta etapa, las estructuras y componentes definidos en el diseño son convertidos en código funcional, utilizando herramientas, lenguajes de programación y entornos previamente seleccionados.

Esta etapa no solo busca construir el sistema, sino también garantizar que su comportamiento cumpla con los requisitos funcionales y no funcionales establecidos, para esto se hace crucial desarrollar pruebas unitarias que sirvan para verificar estos requisitos.

7.4.1 Implementación

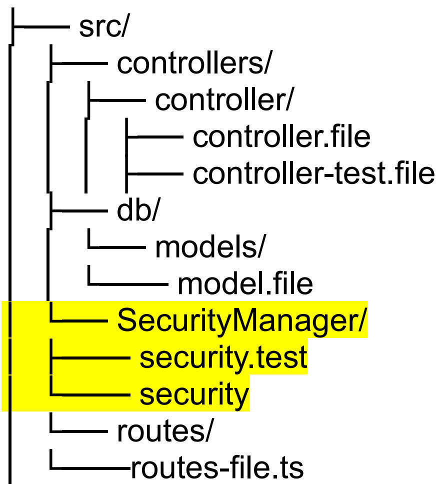
Durante la etapa de implementación, se procede a escribir el código del sistema conforme a las decisiones técnicas tomadas en las etapas previas. La alineación entre lo que fue planificado y lo que se implementa debe ser estricta, de modo que cada componente cumpla con su propósito y pueda ser trazado de forma clara.

Esta etapa se desarrolla en concordancia a lo diseñado en el diagrama de clases y el diagrama de secuencia. Para ello, el sistema se organiza en una jerarquía de carpetas estructuradas de acuerdo con la responsabilidad funcional de cada requisito.

Tomando como inspiración el patrón MVC y la organización de los archivos que se usa en [18], se definen los directorios de esta forma: Los directorios que agrupan la lógica de procesamiento de solicitudes del usuario, es donde estarían los controladores. Los directorios encargados de la manipulación directa de los datos, es donde se ubicarían los modelos. El directorio específico dedicado a la implementación de controles de seguridad es donde alojaría los métodos relacionados con el *SecurityManager*. Esta división no solo aporta claridad al diseño, sino que también permite asociar cada requisito a su respectivo archivo.

En la Figura 7 se muestra el árbol de directorios de ejemplo. La estructura permite ubicar cada método implementado; los métodos de seguridad están resaltados con un color diferente, esto facilita la trazabilidad de los componentes vinculados a los requisitos no funcionales de seguridad. En esta figura también se encuentran las pruebas unitarias que fueron definidas para verificar el cumplimiento de los requisitos, estas se encuentran en los directorios *.test*.

Figura 7. Ejemplo de la estructura de un árbol de directorios



Fuente: elaboración propia.

Para asegurar la trazabilidad, cada archivo que interviene en la implementación de un requisito debe ser identificado y documentado. La Tabla 5 facilita este proceso al vincular el código único del requisito con las clases, métodos y archivos donde se implementa su funcionalidad. De esta manera, se establece una correspondencia directa entre los componentes lógicos del sistema y su ubicación en la estructura de archivos, lo que permite rastrear y verificar técnicamente cada requisito en el código fuente.

Tabla 5. Guía para la trazabilidad de requisitos a los archivos del sistema

Id requisito	**Elemento del diagrama de clases**	**Directorio o archivo que lo implementa**
Código único del requisito (Nombre del requisito)	Clase: Clase controlador donde se encuentra la funcionalidad del requisito. Método: El método especifico dentro de la clase que realiza la acción del requisito. Clase: Clase modelo que está relacionada con este requisito. Método: Método que usa la clase modelo para la acción especifica.	Archivo: Lugar donde se encuentra el controlador de la funcionalidad Archivo: Lugar donde se encuentra el Modelo que representa el esquema en la base de datos.
Código único del requisito no funcional de seguridad (Nombre del requisito no funcional)	Clase: Clase controlador donde se encuentra la funcionalidad del requisito al que se va a aplicar el método de seguridad. Método: El método especifico dentro de la clase que realiza la acción del requisito al que se va a aplicar el método de seguridad. Clase: Clase de seguridad que interviene en la seguridad del requisito. Método: El método especifico dentro de la clase que interviene en la seguridad del requisito	Archivo: Lugar donde se encuentra el controlador de la funcionalidad al que se va a aplicar el método de seguridad Archivo: Lugar donde se encuentra la clase de seguridad que interviene en la seguridad del requisito

Fuente: elaboración propia.

7.4.2 Pruebas unitarias

Esta etapa tiene como propósito asegurar que el sistema cumple con los requisitos definidos durante el análisis y diseño. Esta se desarrolla mediante la ejecución de pruebas específicas orientadas a validar tanto los requisitos funcionales como los requisitos no funcionales, con especial énfasis en aquellos relacionados con la seguridad.

Una vez implementado cada componente del sistema, se aplican pruebas controladas para verificar su correcto comportamiento. Estas pruebas no se limitan a confirmar que la funcionalidad responde según lo esperado, sino que también comprueban la efectividad de los controles de seguridad integrados. Por ejemplo, se diseñan escenarios en los que se inyectan datos maliciosos para comprobar que sean rechazados, o se intenta acceder a recursos protegidos con el fin de verificar que se bloquea el acceso no autorizado.

Con el fin de garantizar la trazabilidad y facilitar auditorías o revisiones posteriores, cada prueba debe ser organizada y documentada de forma rigurosa. Esto incluye registrar la funcionalidad evaluada, el procedimiento de prueba utilizado, los datos de entrada, el resultado esperado y el resultado obtenido. Esta documentación constituye la base de una verificación transparente y reproducible.

Para reforzar la trazabilidad entre requisitos y pruebas, se elabora una tabla de verificación, en la que se establece una correspondencia directa entre cada requisito del sistema y su prueba asociada. La Tabla 6, permite identificar qué elementos del sistema han sido validados, mediante qué pruebas, y con qué resultado. En el caso de los requisitos funcionales, se comprueba que el método implementado ejecuta la lógica definida de forma precisa. Para los requisitos no funcionales de seguridad, la validación se enfoca en determinar si las medidas aplicadas mitigan efectivamente el riesgo o necesidad planteada en el diseño.

Tabla 6. Guía para la matriz de verificación de requisitos con las pruebas unitarias

Id requisito	**Pruebas para realizar**
Código único del requisito (Nombre del requisito)	Verificar que el método actúa de acuerdo con el requisito
Código único del requisito no funcional de seguridad (Nombre del requisito no funcional)	Verificar que el método que interviene en la seguridad soluciona de manera eficaz el problema expuesto en el requisito no funcional de seguridad

Fuente: elaboración propia.

A continuación, se presenta la tabla de trazabilidad actualizada: La relación entre los requisitos, su implementación y las pruebas asociadas se debe establecer claramente para garantizar la trazabilidad. Se detalla qué clase y método implementan cada requisito, así como la ubicación de los archivos correspondientes y los puntos específicos donde se realizan las verificaciones. Esta trazabilidad incluye tanto los requisitos funcionales como los no funcionales de seguridad, permitiendo identificar de manera precisa no solo dónde se encuentra la lógica del sistema, sino también dónde se valida su correcto funcionamiento. La Tabla 7 muestra esta correspondencia entre requisitos, código fuente y archivos de prueba, facilitando el seguimiento de cada funcionalidad desde su definición hasta su comprobación.

La Tabla 7 se relaciona estrechamente con la Tabla 6 al permitir evidenciar la utilidad de cada prueba unitaria y su relación con los requisitos que la originaron, y permitir hacer un seguimiento directo de la trazabilidad de esta prueba desde su requisito, pasando por el diseño, el directorio donde se implementó su funcionalidad y finalmente el directorio donde se encuentra la prueba.

Tabla 7. Guía para la trazabilidad de requisitos con los archivos del sistema y la ubicación de las pruebas unitarias

Id requisito	**Elemento del diagrama de clases**	**Directorio o archivo que implementa la funcionalidad**	**Directorio o archivo donde se implementa la prueba unitaria**
Código único del requisito (Nombre del requisito)	Clase: Clase controlador donde se encuentra la funcionalidad del requisito. Método: El método especifico dentro de la clase que realiza la acción del requisito Clase: Clase modelo que representa un esquema en la base de datos. Método: Método que usa la clase modelo para la acción especifica	Archivo: Lugar donde se encuentra el controlador de la funcionalidad Archivo: Lugar donde se encuentra el Modelo que representa el esquema en la base de datos	Archivo: Lugar donde se encuentran las pruebas que verifican la funcionalidad del requisito
Código único del requisito no funcional de seguridad (Nombre del requisito no funcional)	Clase: Clase controlador donde se encuentra la funcionalidad del requisito al que se va a aplicar el método de seguridad. Método: El método especifico dentro de la clase que realiza la acción del requisito al que se va a aplicar el método de seguridad. Clase: Clase de seguridad que interviene en la seguridad del requisito Método: El método especifico dentro de la clase que interviene en la seguridad del requisito	Archivo: Lugar donde se encuentra el controlador de la funcionalidad al que se va a aplicar el método de seguridad Archivo: Lugar donde se encuentra la clase de seguridad que interviene en la seguridad del requisito	Archivo: Lugar donde se encuentran las pruebas que verifica el método que interviene en la seguridad del requisito

Fuente: elaboración propia.

7.5 Integración y pruebas del sistema

La etapa de integración y pruebas del sistema representa un paso crucial en el ciclo de vida del desarrollo de software, ya que permite validar el comportamiento del sistema como un todo, asegurando que la integración de sus distintos componentes preserve la funcionalidad, estabilidad y seguridad previstas en el diseño. A diferencia de las pruebas unitarias, que evalúan componentes individuales de forma aislada, esta etapa se enfoca en verificar la interacción entre módulos y su funcionamiento conjunto en un entorno que simula condiciones reales de operación.

Estas pruebas permiten evaluar el cumplimiento de los requisitos funcionales y no funcionales en situaciones de uso completas, garantizando que los módulos operen de manera coordinada y sin generar conflictos. Además, se desarrollan pruebas a nivel de sistema, que validan el comportamiento integral del software ante escenarios tanto esperados como imprevistos. Esta validación incluye aspectos críticos como la correcta gestión de excepciones, el manejo seguro de datos, y la respuesta del sistema ante condiciones límite o fallos externos.

El éxito en esta etapa es determinante para asegurar que el sistema final se comporta de acuerdo con las especificaciones, ofreciendo confiabilidad y seguridad antes de su liberación o puesta en producción.

7.5.1 Integración

Esta parte de la etapa permite validar la interacción entre los módulos desarrollados, asegurando que funcionen de manera coordinada y sin comprometer los objetivos del sistema, especialmente en lo referente a funcionalidad, estabilidad y seguridad.

La Figura 8 muestra un diagrama de arquitectura en la nube, creada usando la guía que entrega AWS [19], de la cual está inspirada esta figura, sin embargo, se recomienda usar el diagrama de arquitectura según la nube donde se decida integrar y desplegar el proyecto. Esta figura agrupa los artefactos que intervienen en la aplicación: desde el punto de entrada hasta los componentes internos de procesamiento y de datos. Esta visualización permite comprender la secuencia lógica de interacción entre los módulos y facilita el análisis de su comportamiento en condiciones reales de operación, especialmente en lo que respecta a la aplicación de controles de seguridad.

Con el propósito de darle continuidad a la trazabilidad de los elementos de seguridad, es importante tener en cuenta lo que se definió en la etapa de identificación del proyecto, principalmente con lo relacionado al entorno operativo, tipo de aplicación, actores involucrados y las amenazas identificadas. Estos elementos, son claves para determinar qué tipo de arquitectura y que características de seguridad se deben tener en cuenta para elaborar este diagrama y su posterior implementación. Aunque este diagrama también refleja una distribución física como el diagrama de despliegue que se definió en la etapa de modelado del sistema, el diagrama de arquitectura en la nube refleja de manera general el ambiente seleccionado para desplegar el proyecto, con las características y los artefactos específicos de la nube.

Figura 8. Ejemplo de la representación de la arquitectura de integración de componentes

Fuente: elaboración propia.

7.5.2 Pruebas del sistema

Esta parte de la etapa es esencial para validar que el software completo, una vez integrado, satisface los requisitos funcionales y no funcionales implementados durante el proceso de desarrollo. A diferencia de las pruebas unitarias y de integración que verifican módulos específicos o su interacción, las pruebas del sistema se centran en evaluar el comportamiento del software como un todo dentro de un entorno operativo controlado.

Estas pruebas permiten comprobar que el sistema responde adecuadamente ante escenarios reales de uso, incluyendo tanto condiciones normales como situaciones excepcionales. Se examinan aspectos clave como la ejecución de funcionalidades completas, el flujo correcto de datos, el manejo adecuado de errores, la integridad de las operaciones y la correcta aplicación de las políticas de seguridad.

Este enfoque asegura que el sistema no solo cumpla con lo especificado, sino que también sea robusto, seguro y confiable antes de su puesta en producción, en línea con los principios de verificación y validación propuestos por [12]. De esta forma, podemos hacer un seguimiento estricto a la trazabilidad de los requisitos definidos desde las etapas iniciales y garantizar su cumplimiento hasta su despliegue.

En la Tabla 8 se presenta una guía que establece la trazabilidad correspondiente entre los requisitos y sus respectivas pruebas de sistema, indicando los elementos clave para su validación, donde usando herramientas como *Postman* o *Selenium,* se obtienen reportes que pueden ser usados para trazar los requisitos con sus respectivas funcionalidades y capturar la evidencia de su cumplimiento. Esta tabla presenta un mecanismo de trazabilidad orientado a la etapa de pruebas del sistema, en la cual cada fila de la tabla establece una relación explícita entre un requisito previamente definido y el caso de prueba que permite verificar su implementación dentro del entorno ya integrado. Se documentan los resultados esperados, los resultados obtenidos durante la ejecución, y la evidencia correspondiente, la cual puede incluir archivos de log, reportes automáticos o capturas de validación.

Tabla 8. Guía para la trazabilidad de requisitos a pruebas de integración

ID del Requisito	**Descripción del Requisito**	**Caso de Prueba del Sistema**	**Resultado Esperado**	**Resultado Obtenido**	**Evidencia**
Código único del requisito (Nombre del requisito)	Descripción general del comportamiento funcional que debe cumplir el sistema.	Escenario de validación de la funcionalidad bajo condiciones válidas.	El sistema ejecuta la acción correctamente y responde según lo especificado.	La funcionalidad respondió conforme a lo definido en el requisito.	Archivos de log, reportes automáticos o capturas de validación
Código único del requisito no funcional de seguridad (Nombre del requisito no funcional)	Descripción del control de seguridad esperado.	Prueba de integración con validación de seguridad.	El sistema mantiene el control de seguridad y responde de forma segura ante condiciones adversas.	El sistema validó y protegió el flujo de forma adecuada.	Archivos de log, reportes automáticos o capturas de validación

Fuente: elaboración propia.

7.5.3 Pipeline de despliegue

En el contexto del desarrollo de software moderno, el uso de *pipelines* de despliegue automatizado constituye una estrategia esencial para soportar la entrega controlada de los componentes del sistema. Esta práctica, permite estandarizar y automatizar las actividades

necesarias para validar, ensamblar y desplegar el software hacia entornos de prueba o producción sin intervención manual.

El *pipeline* de despliegue se configura como una secuencia estructurada de etapas que aseguran que cualquier cambio realizado sobre el código pase por múltiples puntos de verificación antes de ser liberado. Cada etapa está diseñada para comprobar un aspecto crítico del sistema, incluyendo la calidad del código, la ejecución de pruebas automatizadas, la activación de controles de seguridad y la validación en entornos controlados.

De esta forma, garantizar que la trazabilidad y el seguimiento de los requisitos, en especial los de seguridad, se apliquen, debido a los controles que son aplicados al iniciar un *pipeline*, ya que cada vez que se registra una modificación en el repositorio del proyecto, este se activa automáticamente. Este se encarga de ejecutar una serie de validaciones que permiten asegurar que los nuevos cambios no introducen fallos de funcionamiento, ni vulnerabilidades de seguridad. De esta forma, se establece una barrera de calidad previa al despliegue, evitando la propagación de errores y potenciales vulnerabilidades hacia entornos reales de operación.

La Tabla 9 describe un ciclo completo de despliegue automatizado, estructurado en diez etapas, definido así típicamente al usar el servidor de Jenkins [20]. Este ciclo inicia con la detección automática de cambios en el repositorio del código fuente. Posteriormente, se realiza una revisión sintáctica y semántica del código mediante herramientas de análisis estático. Se ejecutan pruebas unitarias y de integración para verificar el comportamiento correcto de cada componente, así como la activación de los mecanismos de seguridad previamente definidos.

Tabla 9. Guía donde se muestra un ciclo de despliegue automatizado en entornos de integración continua

Paso	**Descripción**
1. Detección del cambio	El pipeline se activa automáticamente cuando se registra un cambio en el repositorio de código.
2. Validación del código	Se ejecutan herramientas de análisis estático para detectar errores, malas prácticas o posibles vulnerabilidades.
3. Ejecución de pruebas unitarias	Se validan las funciones individuales del sistema para asegurar que se comporten según lo esperado.
4. Ejecución de pruebas de seguridad	Se comprueba que los controles de seguridad sigan activos y funcionando correctamente.
5. Empaquetado de la aplicación	Se genera una versión ejecutable del sistema, preparada para desplegarse.
6. Despliegue en entorno de pruebas (*staging*)	La nueva versión se instala en un entorno intermedio donde se puede validar sin afectar al entorno real.

7. Pruebas de integración y sistema	Se verifica que todos los componentes funcionen correctamente en conjunto.
8. Aprobación manual o automática	El despliegue requiere una validación final o se ejecuta automáticamente si todas las pruebas fueron exitosas.
9. Despliegue en producción	La versión aprobada se instala en el entorno real, reemplazando la versión anterior.
10. Notificación y monitoreo	Se informa a los responsables del despliegue y se activa el monitoreo en tiempo real del sistema.

Fuente: elaboración propia.

7.6 Operación y mantenimiento

Una vez finalizadas las etapas de desarrollo, prueba y despliegue, el sistema entra en una etapa crítica para asegurar su futuro para la operación y el mantenimiento. Esta sección aborda los conceptos y prácticas necesarios para asegurar que el sistema funcione de manera estable, segura y adaptable en un entorno real de producción.

La operación se enfoca en garantizar la disponibilidad y la protección continua del sistema mediante procedimientos definidos para su ejecución controlada, monitoreo y recuperación ante fallos. Por su parte, el mantenimiento comprende las acciones que permiten corregir errores, actualizar componentes, adaptar el sistema a nuevos requerimientos y prevenir la obsolescencia.

7.6.1 Operación

Una vez desplegado el sistema, se deben establecer los procedimientos necesarios para mantener su disponibilidad, seguridad y capacidad de evolución a lo largo del tiempo. Se deben identificar y definir las acciones específicas que permitan gestionar correctamente las operaciones de arranque y cierre controlados, garantizando que el sistema pueda iniciarse y detenerse sin generar condiciones de error ni comprometer su seguridad ante situaciones imprevistas. Se debe planificar cuidadosamente la existencia de entornos de entornos de pruebas de producción (*staging)* y planes de contingencia. Estos mecanismos permiten realizar pruebas seguras antes de aplicar cambios y aseguran la reanudación rápida de operaciones en caso de caídas o incidentes de seguridad, sin afectar los datos ni la estabilidad del sistema.

En esta etapa se organizan todos los pasos necesarios para el funcionamiento seguro y continuo del sistema una vez en producción. Esto incluye la definición de procedimientos para el encendido y apagado controlado, la planificación de pruebas en entornos de pruebas antes

de actualizar versiones, la implementación de copias automáticas de seguridad de la información y la configuración de registros y alertas para detectar fallos o accesos indebidos en tiempo real. Por ejemplo, en una aplicación web de publicaciones y comentarios, esta etapa debe contemplar la programación de copias de seguridad periódicas de la base de datos, la creación de un entorno de pruebas para validar cambios, la definición de protocolos seguros para iniciar y detener el sistema, y la activación de mecanismos de monitoreo mediante registros y alertas automáticas.

7.6.2 Mantenimiento

Estrategias de mantenimiento a largo plazo: Para garantizar que el sistema evolucione al ritmo de la industria, Sommerville [12] señala que el mantenimiento debe concebirse como un proceso continuo que anticipe los cambios tecnológicos y de negocio, reduciendo así los costos de adaptación futuros. En este contexto, el plan de mantenimiento debe incluir (1) Revisiones periódicas de dependencias (librerías de acceso a datos, frameworks, SDK), (2) Análisis estático recurrente para detectar código defectuoso o inseguro, y (3) Aplicación sistemática de parches y actualizaciones que corrijan vulnerabilidades tan pronto como aparezcan en los boletines de seguridad.

Reingeniería de Software: Cuando la base instalada se vuelve obsoleta, se propone la reingeniería como alternativa de menor riesgo y costo frente a la reescritura completa. Este proceso comprende la migración a librerías o *frameworks* modernos, por ejemplo, capas de persistencia que gestionen sentencias SQL de forma segura, y el mejoramiento estructural para que la arquitectura cumpla con los estándares de seguridad actuales. Con ello se reduce la superficie de ataque y se prolonga la vida útil del sistema aun cuando su diseño original no contemplara prácticas contemporáneas de protección.

Refactorización como mantenimiento preventivo: Por su parte, la refactorización constituye mantenimiento preventivo que mejora la estructura, disminuye la complejidad y hace el código más comprensible sin añadir funcionalidad nueva. Al reorganizar clases y métodos, resulta más sencillo incorporar controles de seguridad, sanitización de entradas, validación de ORM, comprobaciones de acceso, sin que el código se degrade. Mantener una base limpia y

modular agiliza la localización y actualización de puntos críticos donde pudieran emerger vulnerabilidades.

En conjunto, estas tres líneas de acción, plan de mantenimiento proactivo, reingeniería selectiva y refactorización continua, ofrecen un marco sólido para preservar la calidad y la seguridad del sistema a lo largo del tiempo, alineándose con las recomendaciones de Sommerville [12] sobre la importancia de diseñar pensando en la evolución y la mantenibilidad.

8 RESULTADOS DE LA APLICACIÓN DE LA ESTRATEGIA DE TRAZABILIDAD

Esta sección muestra los resultados obtenidos al aplicar la estrategia de trazabilidad previamente desarrollada, presentando etapa por etapa cómo se pone en práctica la metodología propuesta mediante un caso de estudio conservan la trazabilidad de los elementos de seguridad.

8.1 Identificación del proyecto

Como escenario práctico para aplicar la estrategia de trazabilidad, se definió una aplicación web que permite la gestión de publicaciones y comentarios. Esta aplicación ofrece funcionalidades como obtener todas las publicaciones, obtener todas las publicaciones filtradas por título, crear una publicación, editar una publicación y crear un comentario, lo que la convierte en un entorno adecuado para simular un ciclo completo de desarrollo con consideraciones reales de seguridad.

Aunque se trata de una solución de baja complejidad funcional, su estructura, compuesta por un servidor *backend* que procesa las peticiones y una base de datos relacional, permite representar de forma clara los distintos momentos del ciclo de vida del desarrollo de software y asociar cada uno de ellos con controles de seguridad específicos. Una aplicación de esta naturaleza puede ser desarrollada con varios modelos del desarrollo de software, pero para los efectos de la aplicación de la estrategia de trazabilidad propuesta, se realizará usando el modelo en cascada.

En la aplicación web de ejemplo desde las etapas iniciales, se incorporan estrategias para identificar vulnerabilidades a lo largo del ciclo de vida del software, tales como validación de entradas, manejo de errores y codificación segura. Asimismo, se implementan pruebas automatizadas, herramientas de análisis estático y documentación técnica para garantizar la trazabilidad de los requisitos funcionales y no funcionales. En este trabajo, sin embargo, referente a los requisitos no funcionales, se centrará únicamente en los no funcionales relacionados con la seguridad. El detalle de esta identificación se presenta en la Tabla 10.

Tabla 10. Identificación del proyecto

Elemento definido	**Descripción en este proyecto de aplicación**
Tipo de aplicación	Aplicación web tipo CRUD para gestión de publicaciones y comentarios
Funcionalidades principales	Obtener todas las publicaciones, Obtener todas las publicaciones filtradas por título, Crear una publicación, Editar una publicación y Crear un comentario
Tecnologías seleccionadas	Node.js, Express, TypeScript, AWS EC2, RDS
Entorno operativo	Entorno *cloud* en AWS
Actores involucrados	Usuario (No autenticado)
Amenazas preliminares identificadas	Posibles inyecciones SQL o ataques XSS

Fuente: elaboración propia

8.2 Definición de requisitos

En esta sección se presentan tanto los requisitos funcionales como los no funcionales de seguridad. Los requisitos funcionales especifican las acciones principales que deben estar disponibles para el usuario. Por su parte, los requisitos no funcionales de seguridad aseguran que el sistema prevenga vulnerabilidades mediante la sanitización de los datos ingresados por el usuario.

Estos requisitos deberán basarse en las funcionalidades establecidas en la etapa anterior, de igual manera los requisitos no funcionales de seguridad deben estar basados en las amenazas preliminares identificadas en la etapa anterior, de esta forma se puede asegurar la trazabilidad de los requisitos y las funcionalidades entre las etapas.

8.2.1 Definición de requisitos funcionales

A continuación, se presentan los requisitos funcionales, escritos siguiendo la estructura recomendada en la Sección 7.2.1.

RF-1 (Obtener todas las publicaciones): El sistema debe permitir al usuario obtener todas las publicaciones, ordenando las publicaciones por fecha de creación en orden descendente.

RF-2 (Obtener todas las publicaciones filtradas por título): El sistema debe permitir al usuario filtrar las publicaciones por título, ordenando las publicaciones por fecha de creación en orden descendente.

RF-3 (Crear una publicación): El sistema debe permitir al usuario crear una nueva publicación, verificando que contenga los siguientes campos obligatorios: Título (title), Contenido (post) y Nombre del autor (author_name).

RF-4 (Editar una publicación): El sistema debe permitir al usuario editar una publicación existente, verificando que contenga los siguientes campos obligatorios: Título (title), Contenido (post) y Nombre del autor (author_name).

RF-5 (Crear un comentario): El sistema debe permitir al usuario crear un comentario en una publicación existente, verificando que el comentario contenga los siguientes campos obligatorios: Contenido del comentario (content) y Nombre del autor del comentario (author_name).

8.2.2 Definición de requisitos no funcionales de seguridad

A continuación, se presentan los requisitos no funcionales de seguridad siguiendo la estructura recomendada en la Sección 7.2.2.

RNS-1 (Sanitización del filtrado de las publicaciones): El sistema debe asegurar que los parámetros de entrada del filtrado de las publicaciones sean sanitizados, para prevenir inyecciones SQL o XSS.

RNS-2 (Sanitización en la creación de una publicación): El sistema debe asegurar que los datos ingresados en las publicaciones sean sanitizados, para prevenir inyecciones de código o ataques XSS.

RNS-3 (Sanitización en la creación de un comentario): El sistema debe asegurar que los datos ingresados en los comentarios sean sanitizados, para prevenir inyecciones de código o ataques XSS.

8.3 Modelado del sistema y del software

Esta etapa consiste en representar gráficamente la arquitectura del sistema y su comportamiento. Se usa el diagrama de clases, el diagrama de despliegue y el diagrama de secuencia, para describir tanto la estructura lógica como la distribución física de los componentes, lo que permite anticipar riesgos y planificar mecanismos de protección.

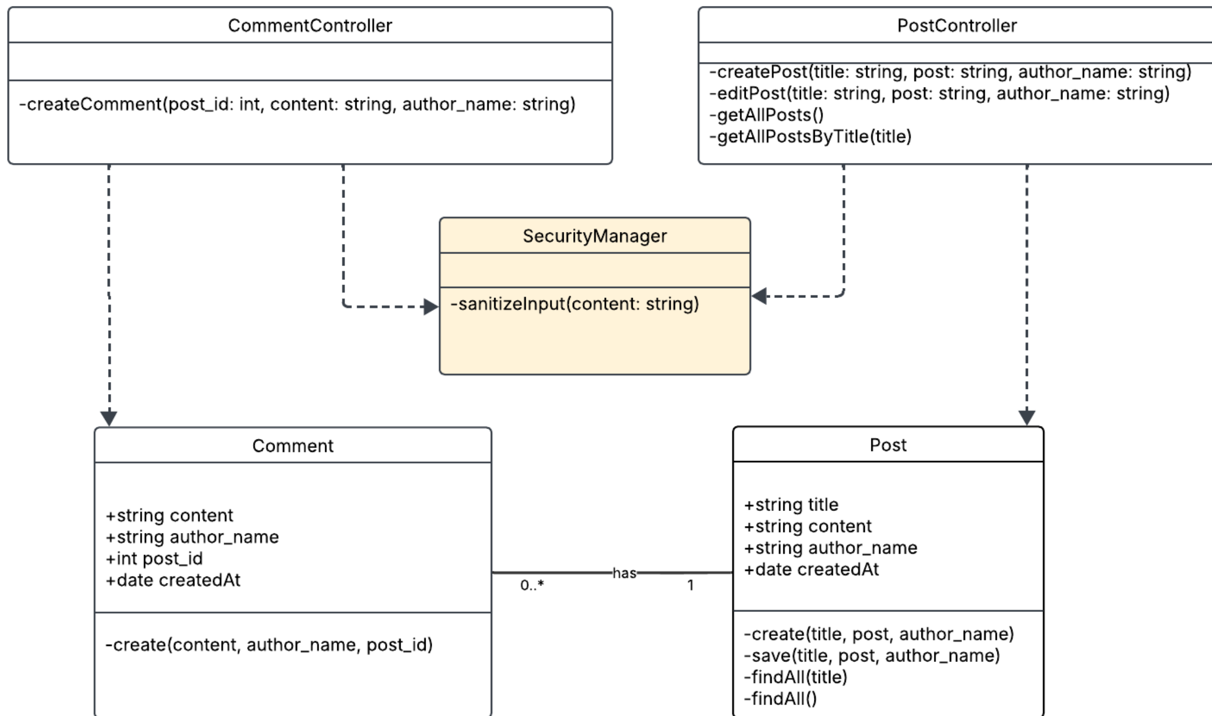
8.3.1 Diagrama de clases

En el contexto de este trabajo, el diagrama de clases cumple un doble propósito: por un lado, garantizar la trazabilidad de los requisitos funcionales a través de las clases PostController y CommentController, cada una con métodos específicos que implementan operaciones como obtener todas las publicaciones, obtener todas las publicaciones filtradas por título, crear una publicación, editar una publicación y crear un comentario; y por otro lado, asegurar que los requisitos no funcionales de seguridad estén representados explícitamente en componentes del sistema, mediante la clase SecurityManager que contiene un método que realiza la sanitización de datos.

La Figura 9 ilustra el diagrama de clases, correspondiente a los principales requisitos funcionales y no funcionales de seguridad definidos en el sistema. Este diagrama permite visualizar cómo se organizan las clases y los métodos que intervienen en la gestión de publicaciones y comentarios, así como los mecanismos de protección aplicados.

Nota: Los nombres de clases, métodos, atributos y relaciones se han dejado en inglés para mantener la coherencia con las demás etapas, dado que las tecnologías empleadas tradicionalmente utilizan este idioma.

Figura 9. Diagrama de clases representando las interacciones entre Controladores y Modelos



Fuente: elaboración propia

En el diagrama se distinguen cinco clases, divididas en tres controladores y dos modelos:

- PostController, responsable de manejar las operaciones relacionadas con publicaciones, incluyendo el Crear una publicación (createPost), el Editar una publicación (editPost), Obtener todas las publicaciones (getAllPosts) y Obtener todas las publicaciones filtradas por título (getAllPostsByTitle).

- CommentController, responsable de manejar las operaciones relacionadas con comentarios, incluyendo la funcionalidad de Crear un comentario (createComment).

- SecurityManager, que es el controlador de seguridad y expone el método sanitizeInput, utilizado como medida de protección para validar y limpiar datos de entrada.

- Post, muestra la estructura de datos del modelo Post, donde se encuentran almacenadas las publicaciones.

- Comment, muestra la estructura del modelo Comment, donde se encuentran almacenados los comentarios.

La relación entre Post y Comment refleja una asociación uno-a-muchos, ya que una publicación puede contener múltiples comentarios.

La Tabla 11 presenta una vista detallada de esta trazabilidad, vinculando cada requisito identificado con su respectiva clase y método. Esta correspondencia facilita la validación técnica del sistema, permite un análisis más claro de la implementación y fortalece la trazabilidad que se busca mantener a lo largo del ciclo de vida del desarrollo. Al integrar tanto requisitos funcionales como de seguridad en un mismo diagrama, se refuerza la coherencia del diseño y se establecen las bases para una implementación alineada con los requisitos definidos desde las etapas iniciales del proyecto.

Tabla 11. Trazabilidad de requisitos al diagrama de clases

Id requisito	**Tipo de requisito**	**Elemento del diagrama de clases**
RF-1 (Obtener todas las publicaciones)	Funcional	Clase: PostController Método: getAllPosts Clase: Post Método: findAll
RF-2 (Obtener todas las publicaciones filtradas por título)	Funcional	Clase: PostController Método: getAllPostsByTitle Clase: Post Método: findAll
RF-3 (Crear una publicación)	Funcional	Clase: PostController Método: createPost Clase: Post Método: create
RF-4 (Editar una publicación)	Funcional	Clase: PostController Método: editPost Clase: Post Método: save
RF-5 (Crear un comentario)	Funcional	Clase: CommentController Método: createComment Clase: Comment Método: findAll
RNS-1 (Sanitización del filtrado de las publicaciones)	No Funcional de Seguridad	Clase: PostController Método: getAllPostsByTitle Clase: SecurityManager Método: sanitizeInput

RNS-2 (Sanitización en la creación de un post)	No Funcional de Seguridad	Clase: PostController Método: createPost Clase: SecurityManager Método: sanitizeInput
RNS-3 (Sanitización en la creación de un comentario)	No Funcional de Seguridad	Clase: PostController Método: createComment Clase: SecurityManager Método: sanitizeInput

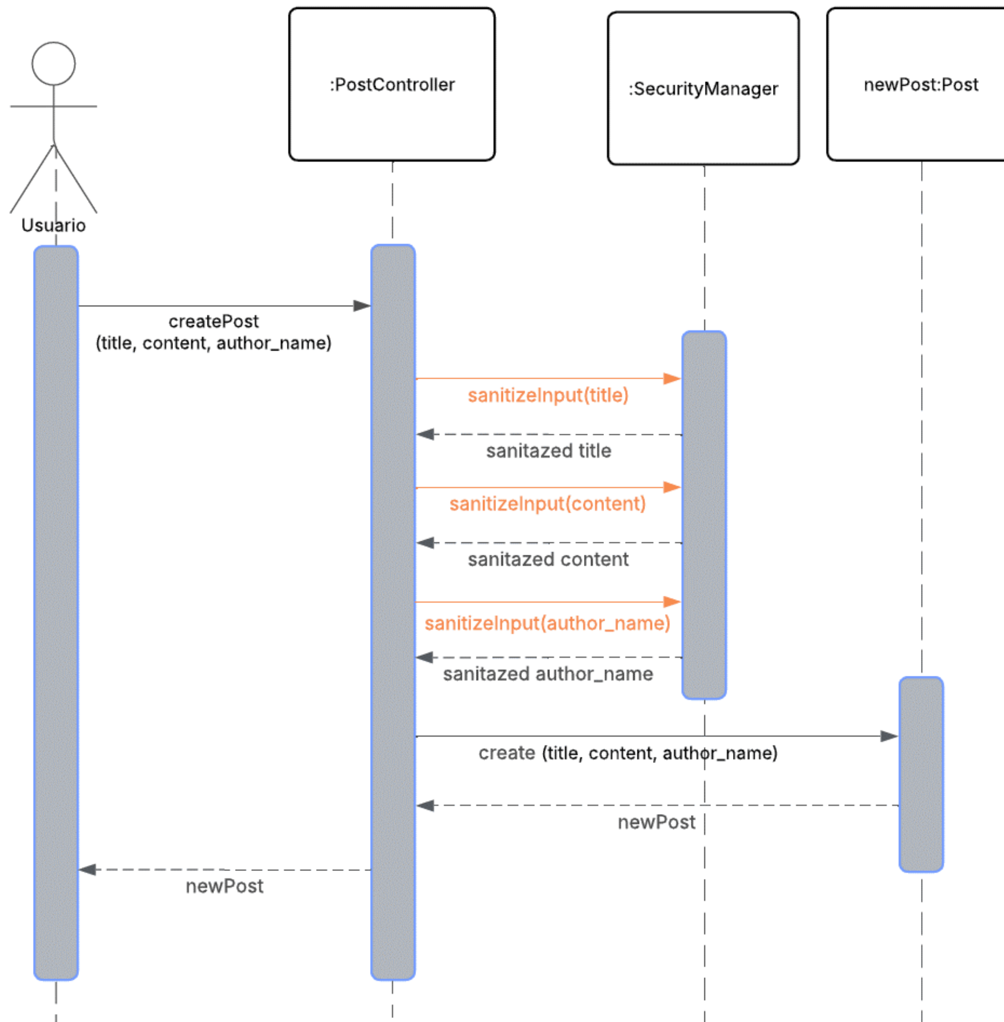
Fuente: elaboración propia.

8.3.2 Diagrama de secuencia

El diagrama de secuencia es útil para describir el flujo usado para ejecutar una funcionalidad dentro del sistema, reflejando tanto la lógica de negocio como la incorporación de mecanismos de seguridad y asegurar su trazabilidad en las diferentes funcionalidades del sistema.

Con fines prácticos e ilustrativos, solo se elaboró el diagrama que describe el flujo de Crear una publicación. La Figura 10 muestra la secuencia asociada a este requisito funcional (RF-3) e incorpora los controles definidos en el requisito no funcional de seguridad (RNS-2), lo que garantiza la protección de los datos procesados. El diagrama evidencia cómo los mecanismos de seguridad se integran en la lógica funcional del sistema, resaltados de un color distintivo para hacer la trazabilidad de seguridad.

Figura 10. Diagrama de secuencia de la creación de un post



Fuente: elaboración propia.

El diagrama de la Figura 10 representa como el proceso se inicia cuando el usuario envía una solicitud al sistema a través del método createPost, proporcionando los parámetros title, content y author_name. Esta solicitud es gestionada por la clase PostController, que actúa como intermediario entre la petición inicial de usuario y la lógica de negocio. Antes de proceder con la creación del objeto Post, cada uno de los parámetros es enviado a la clase SecurityManager, donde se aplica el método sanitizeInput. Esta operación tiene como finalidad neutralizar amenazas comunes como inyecciones SQL y ataques de tipo *Cross-Site Scripting* (XSS), asegurando que los datos recibidos no representen un riesgo para el sistema. Una vez

sanitizados los campos, PostController delega la creación del objeto a la clase Post, utilizando los valores validados. El objeto newPost es entonces generado, y una confirmación es retornada a PostController, el cual responde al usuario con los detalles de la publicación creada.

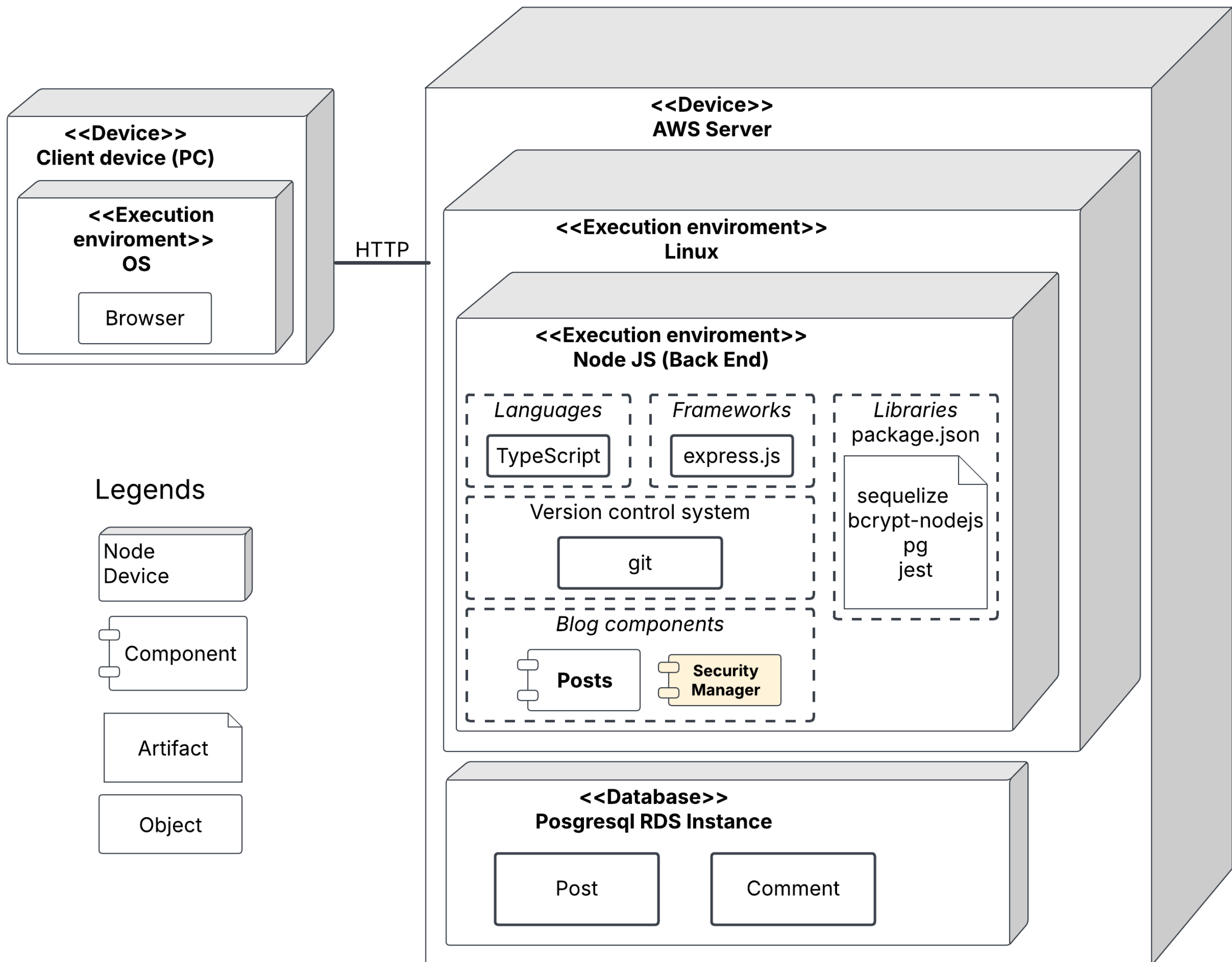
8.3.3 Diagrama de despliegue

Este diagrama detalla los componentes que conforman la arquitectura del sistema y la manera en que interactúan dentro del entorno de ejecución. La visualización de esta estructura no solo permite identificar la distribución lógica y física del sistema, sino que también establece una correspondencia directa entre los elementos técnicos implementados y los requisitos funcionales (RF) y no funcionales de seguridad (RNF) definidos en las etapas anteriores.

Este modelado de despliegue permite observar de forma clara la trazabilidad entre los artefactos de diseño lógico y su correspondencia con la infraestructura real. Esta representación respalda la estrategia de trazabilidad propuesta, al mostrar cómo los elementos de seguridad se encuentran resaltados y trazados a través de toda la arquitectura operativa, como se ilustra en la Figura 11.

Nota: El texto de este diagrama está en inglés, ya que tradicionalmente los diagramas de despliegue usan terminología técnica que originalmente se escribe en este idioma.

Figura 11. *Diagrama de despliegue de la aplicación*



Fuente: elaboración propia.

Analizando la Figura 11, el acceso al sistema se realiza desde distintos dispositivos cliente, haciendo las solicitudes que se transmiten por protocolo HTTP hacia un servidor en la nube, desplegado en un entorno proporcionado por *Amazon Web Services* (AWS).

En este servidor se configura un entorno de ejecución basado en Linux, donde opera el backend desarrollado con Node.js, haciendo uso del lenguaje *TypeScript* y el *framework Express.js*. Este entorno incluye además un sistema de control de versiones (Git), así como librerías y dependencias gestionadas a través de package.json, entre las cuales se encuentran Sequelize, bcrypt-nodejs, pg y Jest.

Dentro del backend se alojan los componentes centrales de la aplicación: el módulo Posts, responsable de gestionar la lógica asociada a las publicaciones, y el módulo SecurityManager, que implementa las medidas de validación y sanitización de entradas. En cumplimiento con los requisitos no funcionales de seguridad previamente definidos. Este último componente resulta clave para evidenciar cómo las decisiones de seguridad son integradas desde el diseño e implementadas de manera verificable.

La persistencia de datos se realiza mediante una instancia del sistema gestor de bases de datos, desplegada en RDS (*Relational Database Service*) de AWS. Esta base de datos contiene las entidades Post y Comment, las cuales se corresponden con el diagrama de clases definido en etapas anteriores, lo que permite trazar cada requisito funcional hacia su almacenamiento y verificación en tiempo de ejecución.

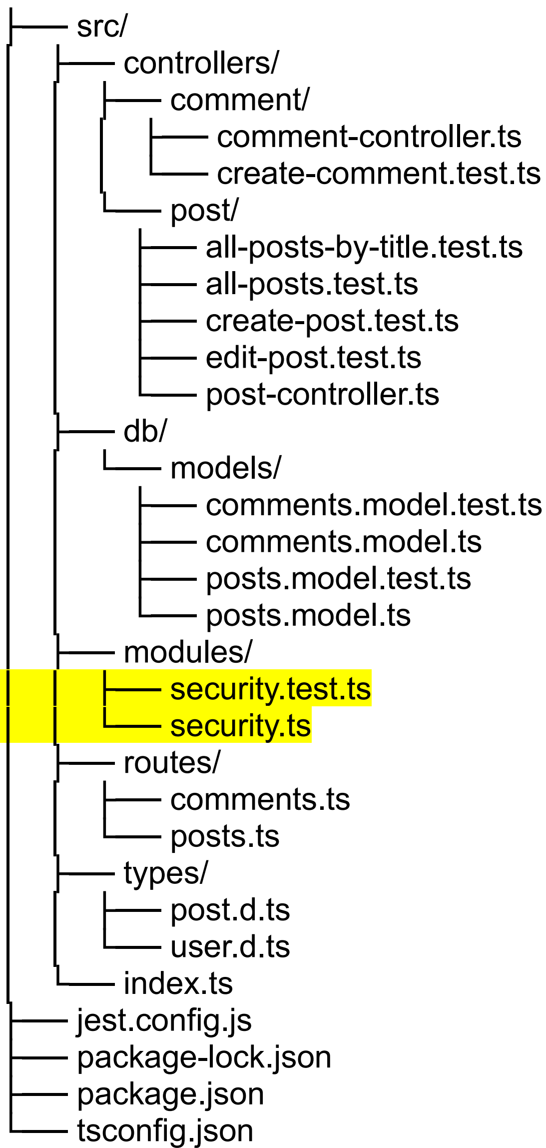
8.4 Implementación y pruebas unitarias

8.4.1 Implementación

Durante la etapa de implementación, uno de los objetivos principales, además de desarrollar lo definido en la etapa de requisitos, es mantener la trazabilidad entre los requisitos y los elementos concretos del código fuente que los satisfacen. Esta trazabilidad permite verificar que cada funcionalidad esperada del sistema ha sido desarrollada, localizada y documentada correctamente en su respectivo archivo y método, facilitando la validación, el mantenimiento y la auditoría técnica del sistema.

En la Figura 12 se muestra el árbol de directorios que contiene el proyecto desarrollado y en la Tabla 12 se presenta la información sobre la trazabilidad de requisitos a los archivos, los cuales permiten hacer el seguimiento de cada funcionalidad desarrollada, con su respectivo archivo y también cada uno de los requisitos usados para este fin. Esta estructura está elaborada siguiendo las recomendaciones que se definieron en la Sección 7.4.1.

Figura 12. Árbol de directorios que contienen el proyecto



Fuente: elaboración propia.

Tabla 12. Trazabilidad de Requisitos a los archivos del sistema

Id requisito	**Elemento del diagrama de clases**	**Directorio o archivo que lo implementa**
RF-1 (Obtener todas las publicaciones)	Clase: PostController Método: getAllPosts Clase: Post Método: findAll	Archivo: src/controllers/post/post-controller.ts Archivo: src/db/models/posts.model.ts
RF-2 (Obtener todas las publicaciones filtradas por título)	Clase: PostController Método: getAllPostsByTitle Clase: Post Método: findAll	Archivo: src/controllers/post/post-controller.ts Archivo: src/db/models/posts.model.ts
RF-3 (Crear una publicación)	Clase: PostController Método: createPost Clase: Post Método: create	Archivo: src/controllers/post/post-controller.ts Archivo: src/db/models/posts.model.ts
RF-4 (Editar una publicación)	Clase: PostController Método: editPost Clase: Post Método: save	Archivo: src/controllers/post/post-controller.ts Archivo: src/db/models/posts.model.ts
RF-5 (Crear un comentario)	Clase: CommentController Método: createComment Clase: Comment Método: findAll	Archivo: src/controllers/post/comment-controller.ts Archivo: src/db/models/comments.model.ts
RNS-1 (Sanitización del filtrado de las publicaciones)	Clase: PostController Método: getAllPostsByTitle Clase: SecurityManager Método: sanitizeInput	Archivo: src/controllers/post/post-controller.ts Archivo: src/modules/security.ts
RNS-2 (Sanitización en la creación de un post)	Clase: PostController Método: createPost Clase: SecurityManager Método: sanitizeInput	Archivo: src/controllers/post/post-controller.ts Archivo: src/modules/security.ts
RNS-3 (Sanitización en la creación de un comentario)	Clase: PostController Método: createComment Clase: SecurityManager Método: sanitizeInput	Archivo: src/controllers/post/comment-controller.ts Archivo: src/modules/security.ts

Fuente: elaboración propia.

La Tabla 12 muestra la correspondencia directa entre:

- Los requisitos del sistema, tanto funcionales (RF) como no funcionales de seguridad (RNS).
- La clase responsable de ejecutar la lógica correspondiente (por ejemplo, PostController, CommentController, SecurityManager) y el método exacto que materializa el comportamiento requerido, y a su vez el modelo de base de datos
- La ruta del archivo en el proyecto donde se encuentra implementada dicha lógica (por ejemplo, src/controllers/post/post-controller.ts o src/modules/security.ts).

Esta relación estructurada garantiza que, ante cualquier necesidad de modificación, prueba o revisión, sea posible localizar rápidamente el fragmento de código que implementa un determinado requerimiento. Por ejemplo, el requisito funcional RF-3 (Crear una publicación) se encuentra implementado en el método createPost de la clase PostController, ubicada en el archivo post-controller.ts. De manera similar, el requisito de seguridad RNS-2 (Sanitización en la creación de un post) se relaciona con el método sanitizeInput de la clase SecurityManager, ubicada en el módulo security.ts.

Este enfoque promueve la trazabilidad bidireccional, es decir, permite tanto identificar qué archivo implementa un requisito como determinar qué requisitos están siendo cubiertos por un determinado módulo de seguridad del sistema. Esta práctica se recomienda para garantizar la trazabilidad del ciclo de vida del software, facilitar la revisión del sistema, y asegurar el cumplimiento de los estándares de seguridad definidos desde la etapa de definición de requisitos al inicio del ciclo de vida del desarrollo de software.

Este enlace dirige al repositorio del código del proyecto, en caso de querer realizar la consulta. https://github.com/davidrzuluaga/MEng-security-owasp-BE

8.4.2 Pruebas unitarias

Las pruebas unitarias son esenciales para comprobar, de forma aislada, que cada método del sistema se comporta según lo previsto. En esta sección, se detallan las pruebas aplicadas al servidor del blog, que funciona como API, con el fin de verificar tanto los requisitos funcionales como, de manera transversal, los requisitos no funcionales de seguridad.

En esta sección se describen las pruebas ejecutadas sobre el servidor del blog (API) para verificar los requisitos funcionales y, de modo transversal, los de seguridad. Las pruebas cubren las operaciones clave, obtener todas las publicaciones, obtener todas las publicaciones filtradas por título, crear una publicación, editar una publicación y crear un comentario, contemplando tanto escenarios válidos como errores habituales (peticiones incompletas o errores del servidor). En paralelo se incluyen verificaciones de seguridad que confirman la sanitización de entradas mediante el método sanitizeInput, con el fin de detectar posibles inyecciones SQL o ataques XSS antes de que los datos se almacenen o se muestren.

La Tabla 13 resume las pruebas diseñadas para cada requisito, especificando el método bajo evaluación y los distintos escenarios considerados, tanto funcionales como de seguridad. Este enfoque refuerza la trazabilidad entre los requisitos definidos y su verificación efectiva durante la implementación.

Tabla 13. Matriz de verificación de requisitos con las pruebas unitarias

Id requisito	**Pruebas para realizar**
RF-1 (Obtener todas las publicaciones)	Verificar que el método (findAll) que obtiene todas las publicaciones las devuelva correctamente y estén ordenadas por fecha de creación en orden descendente.
RF-2 (Obtener todas las publicaciones filtradas por título)	Verificar que el método (findAll) retorne un error si el título a filtrar no es correcto. Verificar que el método (findAll) retorne las publicaciones filtradas por titulo
RF-3 (Crear una publicación)	Verificar que el método (create) crea una nueva publicación correctamente Verificar que el método (create) retorna error si faltan campos obligatorios Verificar que el método (create) maneja errores internos del servidor correctamente
RF-4 (Editar una publicación)	Verificar que el método (update) retorna error si la publicación no existe Verificar que el método (update) actualiza correctamente una publicación existente Verificar que el método (update) maneja errores internos del servidor correctamente
RF-5 (Crear un comentario)	Verificar que el método (create) crea un nuevo comentario en base de datos

	Verificar que el método (create) retorna un error si la creación del comentario falla Verificar que el método (create) maneja errores internos del servidor correctamente
RNS-1 (Sanitización del filtrado de las publicaciones)	Verificar que el método al filtrar, sanitiza los parámetros de entrada para prevenir inyecciones de código o ataques XSS
RNS-2 (Sanitización en la creación de un post)	Verificar que el método al crear un post, sanitiza los datos ingresados para prevenir inyecciones de código o ataques XSS
RNS-3 (Sanitización en la creación de un comentario)	Verificar que el método al crear un comentario, sanitiza los datos ingresados para prevenir inyecciones de código o ataques XSS

Fuente: elaboración propia.

A continuación se presenta la Tabla 14, esta tabla tiene el objetivo de mantener una trazabilidad entre los requisitos definidos, su implementación en el código fuente y su correspondiente validación mediante pruebas.

Tabla 14. Trazabilidad de los requisitos con los archivos del sistema y la ubicación de las pruebas unitarias en los archivos del sistema

Id requisito	**Elemento del diagrama de clases**	**Directorio o archivo que lo implementa**	**Directorio o archivo donde se implementa la prueba**
RF-1 (Obtener todas las publicaciones)	Clase: PostController Método: getAllPosts Clase: Post Método: findAll	Archivo: src/controllers/post/post-controller.ts Archivo: src/db/models/posts.model.ts	Archivo: all-posts.test.ts
RF-2 (Obtener todas las publicaciones filtradas por título)	Clase: PostController Método: getAllPostsByTitle Clase: Post Método: findAll	Archivo: src/controllers/post/post-controller.ts Archivo: src/db/models/posts.model.ts	Archivo: all-posts-by-title.test.ts
RF-3 (Crear una publicación)	Clase: PostController Método: createPost Clase: Post Método: create	Archivo: src/controllers/post/post-controller.ts Archivo: src/db/models/posts.model.ts	Archivo: create-post.test.ts
RF-4 (Editar una publicación)	Clase: PostController Método: editPost Clase: Post Método: save	Archivo: src/controllers/post/post-controller.ts Archivo: src/db/models/posts.model.ts	Archivo: edit-post.test.ts

RF-5 (Crear un comentario)	Clase: CommentController Método: createComment Clase: Comment Método: findAll	Archivo: src/controllers/post/comment-controller.ts Archivo: src/db/models/comments.model.ts	Archivo: create-comment.test.ts
RNS-1 (Sanitización del filtrado de las publicaciones)	Clase: PostController Método: getAllPostsByTitle Clase: SecurityManager Método: sanitizeInput	Archivo: src/controllers/post/post-controller.ts Archivo: src/modules/security.ts	Archivo: security.test.ts
RNS-2 (Sanitización en la creación de un post)	Clase: PostController Método: createPost Clase: SecurityManager Método: sanitizeInput	Archivo: src/controllers/post/post-controller.ts Archivo: src/modules/security.ts	Archivo: security.test.ts
RNS-3 (Sanitizaciónen la creación de un comentario)	Clase: PostController Método: createComment Clase: SecurityManager Método: sanitizeInput	Archivo: src/controllers/post/comment-controller.ts Archivo: src/modules/security.ts	Archivo: security.test.ts

Fuente: elaboración propia.

8.5 Integración y pruebas del sistema

Una vez concluida la etapa de implementación y culminadas las pruebas unitarias, se procede a la etapa de integración y pruebas del sistema. Esta etapa es fundamental para asegurar que los distintos módulos interactúan correctamente entre sí y que la aplicación cumple con los requisitos funcionales y no funcionales definidos durante las etapas previas del ciclo de vida.

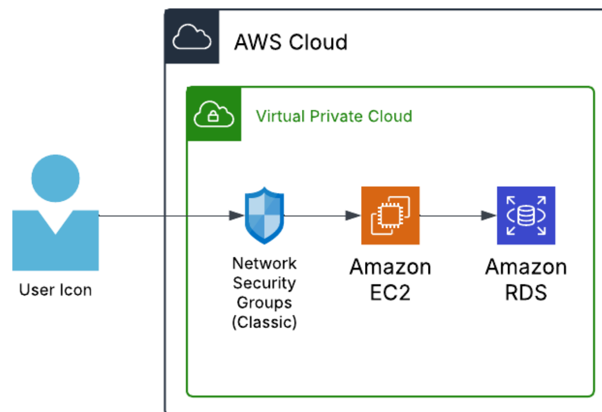
8.5.1 Integración

En la etapa de integración se ensamblan y conectan todos los componentes del sistema, red, servidor de aplicación y base de datos, en un entorno controlado (por ejemplo, una VPC en la nube), de modo que cada elemento interactúe según las especificaciones definidas y bajo políticas de seguridad centralizadas. Esta etapa resulta clave en este trabajo porque es el

momento en que se verifica que los controles de acceso, la comunicación segura y el almacenamiento protegido estuvieron alineados con lo definido en la etapa de identificación y en los requisitos de seguridad establecidos desde el inicio; así, se documenta de forma ordenada cómo cada componente mantiene la trazabilidad de dichos requisitos hasta el despliegue final.

Para este proyecto se decide usar la infraestructura de *Amazon Web Services* (AWS) La Figura 13 representa la arquitectura de integración de la aplicación desplegada en la nube, siguiendo la referencia dada por AWS [19].

Figura 13. Representación de la arquitectura de integración de componentes



Fuente: elaboración propia

La figura ilustra la aplicación desplegada dentro de una *Virtual Private Cloud* (VPC) en AWS, donde las instancias de Amazon EC2 ejecutan la lógica de negocio protegida por *Network Security Groups* que filtran el tráfico entrante y saliente, garantizando que solo peticiones autorizadas alcancen el servidor; desde allí, las operaciones de lectura y escritura de datos se canalizan hacia una base de datos relacional gestionada por Amazon RDS ubicada en subredes privadas, asegurando la persistencia, disponibilidad y control de acceso a la información. Esta configuración, alineada con el ciclo de vida en cascada, permite mantener la trazabilidad de los requisitos de seguridad al documentar cada componente (VPC, *Security Groups*, EC2 y RDS) que interviene en la protección y gestión de datos, facilitando así la validación y el seguimiento de las medidas de seguridad definidas desde la etapa de requisitos hasta el despliegue final.

8.5.2 Pruebas del sistema

En la etapa de pruebas del sistema se verifica el comportamiento completo de la aplicación desplegada, asegurando que todas las funcionalidades interactúan de manera correcta y que los mecanismos de seguridad definidos se apliquen eficazmente. Esta etapa es crítica para garantizar la calidad final del software antes de su liberación hacia entornos de producción.

Para este ejemplo, como complemento a las validaciones integradas en el *pipeline* de despliegue, se utilizan pruebas funcionales de sistema mediante la herramienta *Postman*, la cual permite simular solicitudes HTTP dirigidas a la API del sistema y validar su comportamiento en escenarios de uso real. Estas pruebas son diseñadas para verificar que los distintos *endpoints* expuestos por el *backend* respondan correctamente a solicitudes válidas, rechacen entradas maliciosas y mantengan la lógica funcional y de seguridad esperada.

En el contexto de esta estrategia, se definen colecciones de pruebas en *Postman* que cubren los casos más relevantes asociados a los requisitos funcionales y no funcionales. Cada solicitud es parametrizada para incluir variaciones en los datos de entrada, encabezados, métodos de autenticación y condiciones límite. A través de *scripts* de prueba escritos en *JavaScript*, se evalúan automáticamente las respuestas del sistema, verificando el código de estado, el contenido de los mensajes y la integridad de los datos retornados.

Para facilitar su ejecución continua, estas pruebas se integran en el flujo de despliegue automatizado utilizando *Newman*, el motor de línea de comandos de *Postman*. Esto permite ejecutar las pruebas como parte del *pipeline* gestionado con Jenkins, generando reportes automatizados con los resultados de cada ejecución. En caso de fallos, el proceso puede detenerse o generar alertas, lo que contribuye a garantizar que el sistema cumpla con sus especificaciones antes de ser promovido a producción.

Esta integración de pruebas funcionales con *Postman* complementa la validación estática realizada con *SonarQube*, ofreciendo una capa adicional de verificación orientada a la experiencia real del usuario, fortaleciendo así la calidad y seguridad del sistema desde una perspectiva de ejecución.

En la Tabla 15 se presenta la trazabilidad entre los requisitos definidos en la etapa de requisitos y de diseño y las pruebas ejecutadas durante la etapa de pruebas del sistema. Esta trazabilidad permite verificar que cada funcionalidad implementada y cada medida de seguridad incorporada ha sido correctamente evaluada cuando el sistema está completamente integrado.

Tabla 15. Trazabilidad de requisitos a pruebas de integración

ID del Requisito	Descripción del Requisito	Caso de Prueba del Sistema	Resultado Esperado	Resultado Obtenido	Evidencia
RF-1 (Obtener todas las publicaciones)	El sistema debe permitir a los usuarios obtener todas las publicaciones, ordenadas por fecha descendente.	Solicitud de obtención de publicaciones sin filtros.	El sistema retorna todas las publicaciones correctamente ordenadas.	Las publicaciones se listan correctamente en orden descendente.	test_rf1_get_all_posts.log
RF-2 (Filtrar publicaciones por título)	El sistema debe permitir a los usuarios filtrar las publicaciones por título, ordenadas por fecha descendente.	Solicitud de publicaciones filtradas por un valor de título.	El sistema retorna solo las publicaciones que coinciden con el filtro aplicado.	El filtrado se ejecuta correctamente según el título indicado.	test_rf2_filter_title.log
RF-3 (Crear una publicación)	El sistema debe permitir crear una publicación con los campos obligatorios: título, contenido, autor.	Solicitud con datos válidos en todos los campos requeridos.	La publicación es creada exitosamente y se retorna confirmación.	La publicación fue almacenada correctamente.	test_rf3_create_post.log
RF-4 (Editar una publicación)	El sistema debe permitir a los usuarios editar una publicación existente.	Solicitud de modificación sobre una publicación válida.	El sistema actualiza el contenido y retorna la versión editada.	Los cambios fueron registrados correctamente.	test_rf4_edit_post.log
RF-5 (Crear un comentario)	El sistema debe permitir crear comentarios con contenido y nombre del autor.	Solicitud de creación de comentario con campos completos.	El comentario es almacenado correctamente.	El comentario aparece vinculado a la publicación.	test_rf5_create_comment.log

RNS-1 (Sanitización del filtrado de publicaciones)	El sistema deberá sanitizar los parámetros del filtro por título para prevenir inyecciones SQL o XSS.	Envío de parámetros maliciosos en el filtro por título.	El sistema bloquea o limpia la entrada sin comprometer la consulta.	El sistema respondió sin ejecutar contenido malicioso.	test_rns1_filter_sanitization.log
RNS-2 (Sanitización en creación de post)	El sistema deberá sanitizar los datos ingresados en los posts para prevenir ataques.	Envío de contenido con scripts maliciosos al crear un post.	El sistema rechaza o limpia los scripts sin almacenarlos.	El contenido fue neutralizado correctamente.	test_rns2_post_sanitization.log
RNS-3 (Sanitización en creación de comentario)	El sistema deberá sanitizar los datos en comentarios para evitar almacenamiento de código malicioso.	Envío de comentario con intento de XSS.	El sistema filtra o bloquea el contenido antes de guardarlo.	El comentario fue rechazado o limpiado correctamente.	test_rns3_comment_sanitization.log

Fuente: elaboración propia.

Cada fila de la tabla establece una relación directa entre un requisito funcional o no funcional de seguridad y el caso de prueba que permite validarlo. Se documentan el comportamiento esperado, el resultado obtenido y la evidencia generada durante la ejecución de la prueba. Este mecanismo asegura que tanto las funcionalidades básicas como los controles de seguridad mantienen su vigencia y efectividad en el entorno final, después de la integración de los componentes.

En conjunto, esta tabla contribuye a cerrar el ciclo de trazabilidad definido en la estrategia propuesta, proporcionando soporte documental para verificar el cumplimiento de cada requisito a lo largo del ciclo de vida del software en el modelo en cascada.

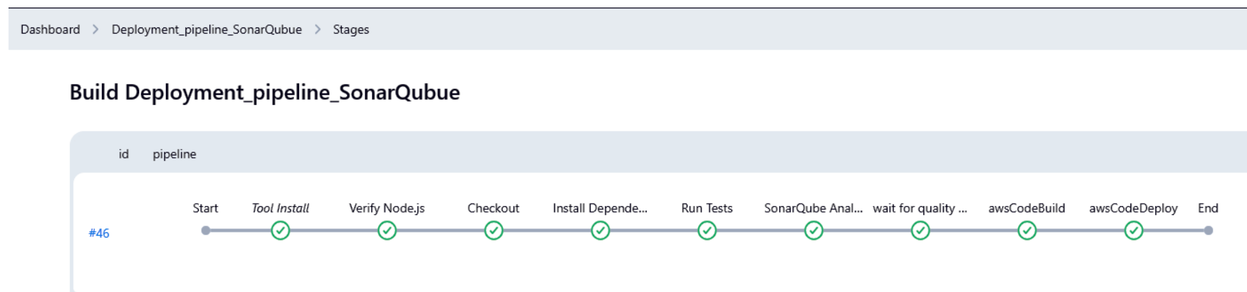
8.5.3 *Pipeline* de despliegue

Un *pipeline* de despliegue contiene los requisitos que se deben cumplir para entregar un conjunto de componentes al entorno donde se desea desplegar. Es importante detallar su utilidad y configuración a la luz de esta trazabilidad ya que complementa todos los controles

definidos en las etapas anteriores y asegura que en última instancia se despliegue el código con la integridad esperada.

En la imagen de la Figura 14 se puede observar el proceso que hace el *pipeline* para completar el despliegue en AWS

Figura 14. Proceso despliegue: Pipeline configurado en AWS



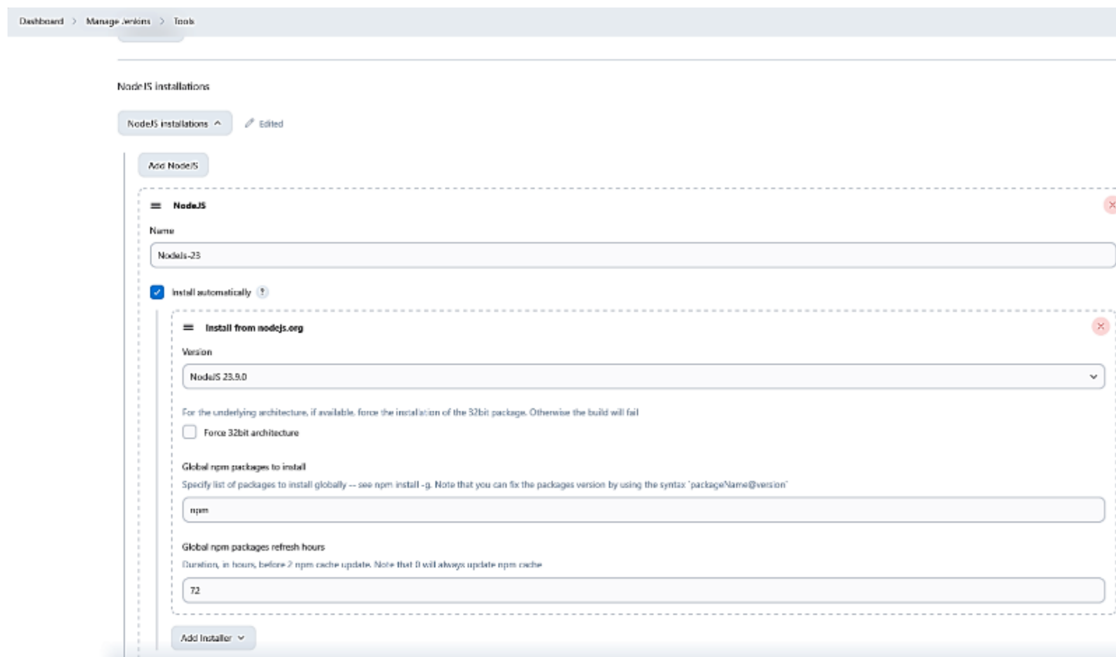
Fuente: elaboración propia.

Estos son los pasos que ejecuta el *pipeline* de manera explicada:

- Tool install: instala las dependencias propias de la herramienta de despliegue.
- Verify NodeJS: ejecuta un comando para verificar la versión de Node instalada.
- Checkout: se clona el repositorio desde *GitHub*.
- Install Dependencies: instala las dependencias del proyecto con npm install.
- Run tests: ejecuta las pruebas del proyecto con el comando npm run test
- SonarQube análisis: se conecta con el servidor de *SonarQube* y se ejecuta el análisis para identificar errores en el código y/o vulnerabilidades.
- Wait for quality gate: se espera la respuesta del análisis de SonarQube, si no prueba el análisis se detiene la ejecución del *pipeline*, si pasa el análisis se continua con los siguientes pasos.
- awsCodeBuild: se conecta con aws y se ejecuta el build del proyecto.
- awsCodeDeploy: se conecta con aws y se realiza el despliegue del proyecto en la instancia EC2.

La Figura 15 muestra la configuración de *Jenkins* con las reglas de despliegue y de pruebas.

Figura 15: *Proceso despliegue: Configuración de Jenkins*

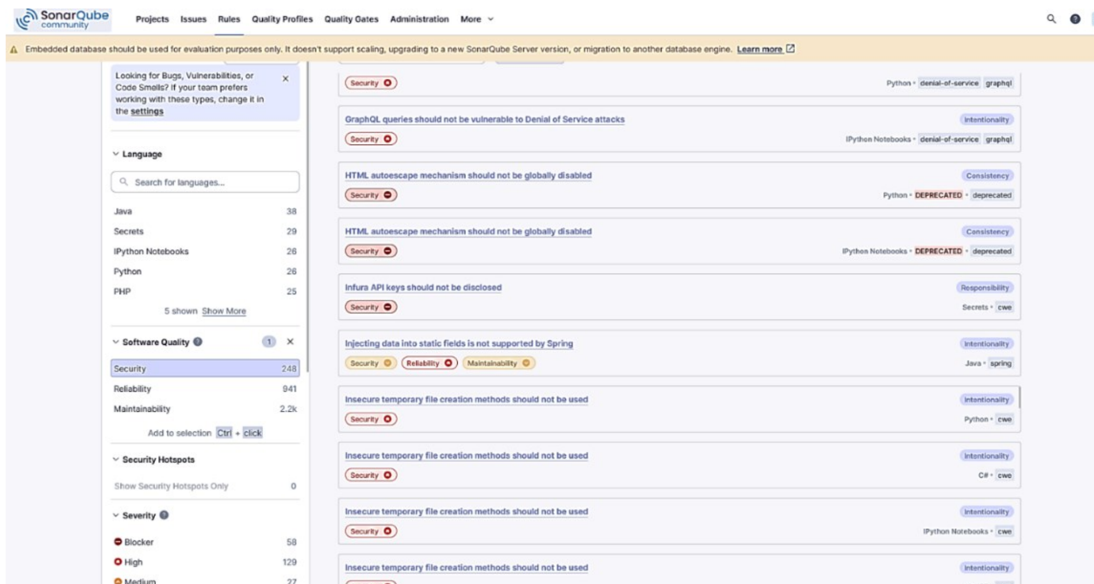


Fuente: elaboración propia.

Reglas de seguridad preconfiguradas para *SonarQube*

Con el objetivo de garantizar que los requisitos de seguridad definidos se cumplan de manera satisfactoria, *SonarQube* se configura con un conjunto de reglas de seguridad que ayuda a detectar vulnerabilidades. En la Figura 16 se puede ver algunas de las reglas preconfiguradas para el análisis de seguridad del código en *SonarQube*.

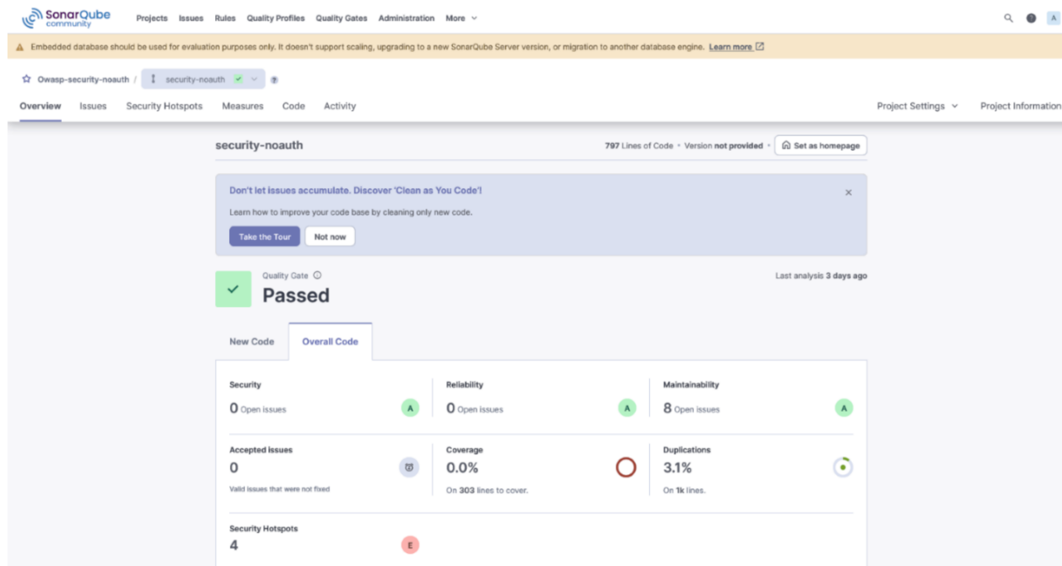
Figura 16. Proceso despliegue: Reglas preconfiguradas para el análisis de seguridad del código en SonarQube



Fuente: elaboración propia.

El resultado luego de la ejecución del *pipeline* en Jenkins, en el servidor de *SonarQube* se ilustra en la Figura 17, se pueden ver los resultados del análisis del código.

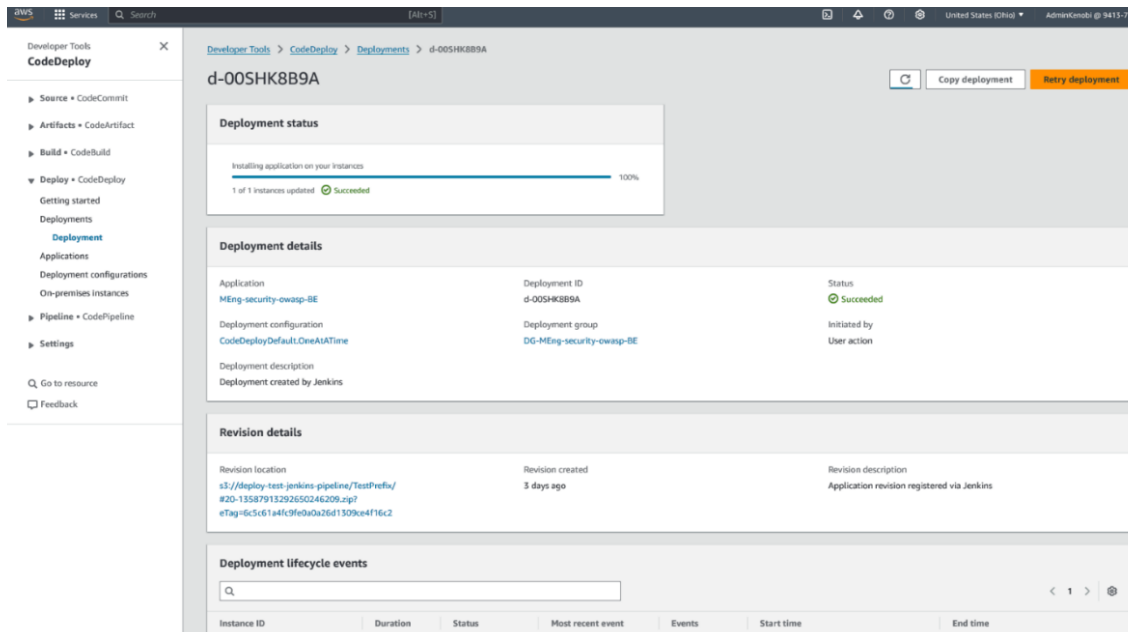
Figura 17. Proceso despliegue: Resultado luego de la ejecución del pipeline en Jenkins



Fuente: elaboración propia.

La Figura 18 muestra el proceso final luego de la ejecución del pipeline en Jenkins, es posible verificar que en la herramienta *CodeDeploy* de AWS, donde muestra que el despliegue se realizó de manera exitosa.

Figura 18. Proceso despliegue: Proceso final luego de la ejecución del pipeline en Jenkins



Fuente: elaboración propia.

Este pipeline garantiza la calidad del código mediante pruebas unitarias y análisis estático, integrando de manera fluida las herramientas de *SonarQube* junto con Jenkins en un entorno de despliegue. Con esta solución, el proceso de despliegue será más confiable ya que ayuda a que los problemas se puedan detectar y solucionar antes del despliegue final en producción.

8.6 Operación y mantenimiento

En esta sección se detalla el periodo posterior al despliegue: monitoreo, gestión de parches y auditorías continuas.

8.6.1 Operación

Para mantener una operación segura y robusta, es importante mantener control sobre los procedimientos que se realizan para mantener la disponibilidad, debe planificarse cuidadosamente la forma de iniciar y detener un sistema para evitar condiciones de error y garantizar que el sistema permanezca seguro ante situaciones imprevistas. También es importante mencionar la importancia de tener entornos de *staging* y planes de contingencia para que, si ocurre una caída o se detecta un incidente de seguridad, sea posible reanudar operaciones rápidamente y sin afectar a los datos y la estabilidad del sistema. Respecto a planes de contingencia, en esta aplicación web de ejemplo de posts/comentarios, una implementación imprescindible debe incluir respaldos automáticos de la base de datos, previniendo con la utilización de procedimientos de restauración, lo cual garantiza la continuidad del servicio.

Otro punto importante para garantizar la operación del sistema es la implementación de un sistema de *logs*, alertas y auditorias, para verificar en tiempo real el potencial uso malicioso del sistema, rastreando *logs* y monitoreando que los usuarios estén usando de manera correcta las funcionalidades y así procurar seguridad y calidad.

8.6.2 Mantenimiento

En esta etapa se definen las estrategias necesarias para mantener el sistema seguro y funcional a lo largo del tiempo:

- Se identifican los ajustes requeridos ante cambios tecnológicos, la corrección de fallos posteriores al lanzamiento y las acciones que permitan evolucionar el sistema sin necesidad de rehacerlo completamente.

- Se establecen estrategias de mantenimiento a largo plazo que incluyen la revisión periódica de librerías de acceso a datos, el análisis estático de código para detectar funciones defectuosas, y la aplicación de parches y actualizaciones de seguridad.
- Se planifica para anticipar los cambios que la industria tecnológica experimenta, permitiendo una adaptación eficaz.

Ahora bien, cuando se detecten signos de obsolescencia, se aplica una estrategia de reingeniería; esta consiste en actualizar componentes críticos, como la migración a *frameworks* o librerías que gestionen consultas de forma segura, preservando el núcleo funcional del sistema y mitigando vulnerabilidades estructurales. También se incorpora el *refactoring* como mantenimiento preventivo, mejorando la estructura del código, reduciendo su complejidad y facilitando su comprensión.

9 EVALUACIÓN DE LA ESTRATEGIA DE TRAZABILIDAD

En esta sección se encuentran las evaluaciones cualitativa y cuantitativa correspondientes a los resultados de la aplicación de la estrategia de trazabilidad presentada en la sección 8, derivadas de la estrategia desarrollada en la sección 7.

9.1 Comparación cualitativa

La aplicación de la estrategia de trazabilidad permitió evidenciar, cómo los requisitos de seguridad pueden ser definidos, documentados y verificados a lo largo de todas las etapas del ciclo de vida del software.

Con el fin de analizar el valor agregado de esta estrategia se presenta la Tabla 16, donde se establece una comparación entre el modelo en cascada con trazabilidad propuesta en este trabajo y el modelo en cascada.

Tabla 16. Comparación cualitativa entre el modelo en cascada con trazabilidad y el modelo en cascada tradicional.

Etapa	**Modelo en Cascada con la estrategia de Trazabilidad de Seguridad propuesta en este trabajo**	**Modelo en Cascada**
Identificación del proyecto	Se delimitan aspectos claves del proyecto para ser usados más adelante en las diferentes etapas del desarrollo de software.	Esta etapa no es contemplada de manera explícita en el modelo en cascada tradicional.
Definición de requisitos	Cada requisito de seguridad recibe un código único y se registra en una tabla de trazabilidad que permanece durante toda la vida del proyecto, enlazándolo desde el inicio con riesgos y controles.	Los requisitos se establecen en el inicio del proyecto y permanecen fijos y consignados en los documentos para auditorías; no existen mecanismos explícitos que garanticen su vigencia a lo largo del ciclo.
Modelado del sistema y del software	La trazabilidad propuesta exige que cada decisión tomada desde el diseño, refleje el requisito que la originó, en especial los de seguridad, manteniendo el seguimiento de estos. Se propone desarrollar diagramas de clases, de secuencia y de despliegue para facilitar la vinculación de cada requisito con los elementos que lo implementan.	Se entrega un único paquete de diseño con diagramas formales, la seguridad suele quedar descrita de forma genérica y sin trazabilidad. De manera tradicional, no se recomiendan diagramas explícitamente, dejándolo a la elección de quien lo implementa.

Implementación y pruebas unitarias	Se documentan prácticas de codificación segura. Se realiza la trazabilidad con referencias precisas (clase, método, ruta de archivo) y con resultados de herramientas como *SonarQube* / *SonarLint*, asegurando que el código precede al requisito que lo originó, en especial, los requisitos de seguridad. Las pruebas unitarias incluyen validaciones de seguridad específicas, con una trazabilidad precisa y documentada de la relación que existe entre cada prueba y su requisito asociado.	El desarrollo se centra en la funcionalidad. La trazabilidad entre requisitos y clases/métodos es difusa. Las pruebas se concentran en la funcionalidad. Por ejemplo, las pruebas unitarias a menudo se aplican al final de la etapa de implementación, pero no se crea un vínculo explicito con los requisitos iniciales.
Integración y pruebas del sistema	Incorpora explícitamente la verificación de requisitos funcionales y no funcionales de seguridad, aplicados en el mismo momento en que se validan los flujos funcionales integrados. Se propone la configuración de un *pipeline* de integración continua con pasos específicos para análisis estático, pruebas de seguridad y despliegue progresivo, incluyendo mecanismos de monitoreo y alertas.	No realiza la verificación con los requisitos al momento de la integración y el despliegue, suele posponerse hasta etapas posteriores o incluso hasta el mantenimiento. El despliegue es único. No incluye verificaciones estructuradas de seguridad antes de producción.
Operación y Mantenimiento	Se planean actualizaciones, refactorizaciones y migraciones como parte del mantenimiento. Se registran decisiones técnicas de seguridad a lo largo del tiempo, realizando la respectiva trazabilidad de los elementos de seguridad hasta esta etapa.	El mantenimiento de manera tradicional reacciona a incidentes; no se documenta la relación entre requisitos y modificaciones. Corregir hallazgos implica reabrir etapas anteriores, con alto costo de reproceso.

Fuente: elaboración propia

La tabla anterior evidencia que la estrategia de trazabilidad de elementos de seguridad, integrada al modelo en cascada, proporciona un seguimiento continuo que enlaza requisitos, diseño, código, pruebas, integración y mantenimiento sin alterar la lógica secuencial del modelo en cascada. Desde la primera etapa, se define el contexto del riesgo, facilitando la formulación de criterios de aceptación y respaldando futuras auditorías. De esta forma, permite una planificación a largo plazo, puesto que localiza y verifica cualquier requisito, especialmente los de seguridad, en cualquier etapa del ciclo de vida. Durante el diseño, conserva la trazabilidad al mostrar cómo cada requisito de seguridad se plasma en los diagramas

correspondientes. Además, al asignar un código único a cada requisito, deja evidencia de qué parte de la implementación lo satisface y qué pruebas lo cubren, reduciendo el costo de reproceso. Finalmente, hace posible rastrear cada prueba de integración, en particular las de seguridad, antes de la puesta en producción.

9.2 Comparación cuantitativa

A continuación, se presenta la Tabla 17 que contiene una comparación cuantitativa, con algunas métricas, entre la aplicación desarrollada usando el modelo en cascada con trazabilidad propuesto en este trabajo y el modelo en cascada.

Tabla 17. Comparación cuantitativa entre el modelo en cascada con trazabilidad y el modelo en cascada tradicional.

Métrica	Modelo en cascada con la estrategia de trazabilidad de seguridad propuesta en este trabajo	Modelo en cascada
N.º de requisitos funcionales	5	5
N.º de requisitos no funcionales de seguridad	3	3
Requisitos que se trazaron hasta el final del ciclo de vida	3	0
Duración del proyecto de principio a fin	3 meses	2 meses
Vulnerabilidades mitigadas debido a lo descubierto en la etapa inicial de identificación del proyecto	2	0

Fuente: elaboración propia

La Tabla 17 revela que partiendo del mismo alcance que se desarrolló en la aplicación de la estrategia en la sección 8 (cinco requisitos funcionales y tres de seguridad), la incorporación de la trazabilidad permite visualizar diferencias. Con la estrategia propuesta, los tres requisitos de seguridad se trazaron hasta el final del ciclo de vida, lo que garantiza la cobertura y visibilidad, en tanto que en el modelo en cascada tradicional ese vínculo se perdió. Por otro lado, la trazabilidad permitió detectar y mitigar dos vulnerabilidades detectadas desde la etapa inicial, evitando que se propagaran a etapas posteriores donde el costo de corrección podría haber sido mayor. El precio de estos beneficios fue un aumento del tiempo total del proyecto: tres meses frente a dos en el enfoque clásico, es decir, un 50 % más de duración atribuible al esfuerzo adicional de documentar y hacer la trazabilidad en cada etapa. En síntesis, la estrategia mejora la gobernanza y reduce los riesgos de seguridad, pero lo hace a costa de un sobrecosto temporal que se justifica cuando se consideran las exigencias regulatorias del proyecto y estas exigen un control riguroso de los requisitos de seguridad.

10 CONCLUSIONES

El desarrollo de este trabajo demostró que es posible implementar una estrategia de trazabilidad de los elementos de seguridad para cada etapa del ciclo de vida del desarrollo de software, usando el modelo en cascada. La propuesta integró guías, tablas, figuras y ejemplos que enlazaron requisitos con el modelado, el código fuente, las pruebas y el despliegue, cumpliendo así el objetivo general planteado.

Durante el desarrollo del trabajo se identificaron los momentos clave en cada etapa del ciclo de vida del desarrollo de software, lo que permitió incorporar de manera sistemática los requisitos de seguridad desde el inicio del proceso.

Se diseñó una estrategia que permitió trazar estos requisitos de seguridad a lo largo de todo el ciclo de vida del desarrollo de software, garantizando su seguimiento continuo. La estrategia de trazabilidad contiene tabla matriz que asigna un identificador único a cada requisito y lo enlaza con diagramas UML, código, pruebas, despliegue y documentación.

La aplicación de esta estrategia de trazabilidad en un desarrollo de software demostró la viabilidad del enfoque en un caso práctico, evidenciando cómo se vinculan los requisitos de seguridad, con cada etapa del ciclo de vida del software usando el modelo cascada con trazabilidad.

Esta estrategia de trazabilidad proporciona visibilidad y control permanente de los requisitos de seguridad, enlazando cada requisito con su diseño, implementación, pruebas y despliegue, facilitando auditorías, ajustes tempranos y un menor costo de corrección. El modelo en cascada, al carecer de estos vínculos explícitos, corre el riesgo de que la seguridad se diluya o se aborde tarde, con mayor esfuerzo y mayor exposición a fallos.

La comparación de la estrategia planteada con el modelo en cascada, evidencia que, aunque este último tiene sus propios ritmos y métodos, la incorporación de una trazabilidad explícita actúa como un mecanismo transversal que añade valor al modelo original. Así, se logra gestionar la seguridad de manera formal y trazable en proyectos estructurados, aportando visibilidad y control de los requisitos de seguridad sin la necesidad de modificar drásticamente la metodología de trabajo. Durante esta comparación se descubrió que esta estrategia conlleva un tiempo más alto de ejecución, respecto al modelo en cascada

tradicional, pero este sobrecosto, puede ser beneficioso al traducirse en los beneficios nombrados anteriormente.

En definitiva, este trabajo evidencia que el modelo en cascada puede enriquecerse con prácticas modernas de seguridad por medio de la trazabilidad, logrando un proceso más robusto sin renunciar a la planificación y la documentación exhaustiva que lo caracterizan. De esta manera, se aporta una ruta práctica para que las organizaciones que dependen de métodos secuenciales puedan avanzar hacia un desarrollo de software seguro, trazable y verificable.

Limitaciones de esta propuesta

Durante el desarrollo de este trabajo se identificaron tres limitaciones principales.

En primer lugar, la estrategia se aplicó a un proyecto de dimensión reducida, debido al tiempo y alcance de este trabajo de grado. Esto podría dificultar su extrapolación a proyectos más grandes, donde el modelo en cascada se emplea con mayor frecuencia. Aun así, el estudio de caso resultó suficiente para demostrar el alcance y el valor de la propuesta.

En segundo lugar, las etapas de integración y pruebas se ejecutaron en un entorno tecnológico específico; por consiguiente, su eficacia podría variar en contextos con infraestructuras muy distintas o sin procesos de automatización.

En tercer lugar, aunque la estrategia aporta beneficios, también introduce un posible sobrecosto al incrementar el esfuerzo adicional requerido para usar trazabilidad. Sin embargo, en proyectos de alta criticidad o regulación estricta, este seguimiento se puede traducir en reducción de riesgos a la seguridad y a la reputación.

Trabajos futuros

A largo plazo, el modelo en cascada seguirá siendo relevante en sectores que demandan alta trazabilidad, validación formal y cumplimiento normativo. Como trabajo futuro, se propone diseñar estrategias de trazabilidad para proyectos de larga duración que integren tecnologías consolidadas como *blockchain* con el fin de asegurar la inmutabilidad y verificabilidad de cada decisión técnica. Dichas estrategias podrían sustentarse en repositorios de requisitos de

seguridad trazables, capaces de evolucionar con el tiempo y de facilitar auditorías retrospectivas incluso décadas después de la puesta en producción. Así, un sistema en operación por años podría mantener un registro íntegro de cambios, parches y actividades de reingeniería sin comprometer el diseño original.

REFERENCIAS

[1] J. M. B. T. H. Borky, «Protecting Information with Cybersecurity.,» *Effective Model-Based Systems Engineering,* p. 345–404, 2019.

[2] Google Inc, «7 ways we're incorporating security by design into our products and services,» Google, Oct 2024. [En línea]. Available: https://blog.google/technology/safety-security/google-secure-by-design-pledge/. [Último acceso: May 2025].

[3] microsoft, «Security Development Lifecycle (SDL) Practices,» microsoft, [En línea]. Available: https://www.microsoft.com/en-us/securityengineering/sdl/practices. [Último acceso: May 2025].

[4] Netflix Technology Blog, «Announcing Security Monkey — AWS Security Configuration Monitoring and Analysis,» Jun 2014. [En línea]. Available: https://netflixtechblog.com/announcing-security-monkey-aws-security-configuration-monitoring-and-analysis-1f2bfb001708. [Último acceso: May 2025].

[5] G. McGraw, «Software Security,» *IEEE Security and Privacy,* p. 80–83, 2004.

[6] P. Rotella, «Software security vulnerabilities: Baselining and benchmarking.,» *Proceedings - International Conference on Software Engineering,* p. 3–10, 2018.

[7] A. Kiezun, P. Guo, K. Jayaraman y M. Ernst, «Automatic creation of SQL Injection and cross-site scripting attacks,» *International Conference on Software Engineering,* pp. 199-209, 2009.

[8] S. Sanyal, «ntroduction to SonarQube: Elevate Your Code Quality and Security,» medium, 24 Nov 2024. [En línea]. Available: https://medium.com/@sanyal.s271/introduction-to-sonarqube-elevate-your-code-quality-and-security-1c42fd092bdb.

[9] F. Tian, T. Wang, P. Liang y C. Wang, «The impact of traceability on software maintenance and evolution: A mapping study,» *Journal of Software: Evolution and Process,* p. 33(10), 2021.

[10] M. E. Tabaréz, F. Arango y R. Anaya, «Una revisión de modelos y semánticas para la trazabilidad de requisitos,» *Revista EIA,* vol. 6, pp. 33-44, 2006.

[11] J. A. Osorio, R. A. Villalobos y M. Zapata, «Elicitación y trazabilidad de requerimientos utilizando patrones de seguridad,» *Revista Avances en Sistemas e Informática,* vol. 4, nº 2, p. 65–72, 2007.

[12] I. Sommerville, Software engineering, Edinburgh: Pearson, 2016.

[13] N. Medvidovic, D. S. Rosenblum, D. F. Redmiles y J. E. Robbins, «Modeling software architectures in the Unified Modeling Language.,» *ACM Transactions on Software Engineering and Methodology (TOSEM),* vol. 11, p. 2–57, 2002.

[14] R. Mohammadi y A. Barforoush, «Enforcing component dependency in UML deployment diagram for cloud applications,» *7'th International Symposium on Telecommunications,* pp. 412-417, 2014.

[15] K. Peffers, T. Tuunanen, M. Rothenberger y S. Chatterjee, «A design science research methodology for information systems research.,» *Journal of Management Information Systems,* vol. 24, nº 3, pp. 45-77, 2007.

[16] A. I. y S. R, Writing Better Requierements, Addison-Wesley, 2002.

[17] H. Hnaini, R. Mazo, P. Vallejo, A. Lopez, J. Champeau y J. Galindo, «SECRET: A New SECurity REquirements SpecificaTion Template.,» *Lecture Notes in Networks and Systems,* vol. 933 LNNS, p. 235–246, 2024.

[18] S. Kunjumohamed, H. Sattari, A. Bretet y G. Warin, Spring MVC: Designing Real-World Web Applications, Packt Publishing, 2016.

[19] Amazon Web Services, «AWS Reference Architecture Diagrams,» AWS, [En línea]. Available: https://aws.amazon.com/es/architecture/reference-architecture-diagrams/. [Último acceso: 05 2025].

[20] The Linux Foundation, «Pipeline,» 2025. [En línea]. Available: https://www.jenkins.io/doc/book/pipeline/. [Último acceso: 05 2025].