



Implementación de un pipeline de seguridad para la detección de vulnerabilidades en aplicaciones

Manuel José Bothert Martínez

Informe final para optar al título de Ingeniero de Telecomunicaciones

Asesor Interno

Luis Alejandro Fletscher Bocanegra, Doctor en Ingeniería.

Asesor Externo

Francisco Muñoz Cortés, Maestría en Ingeniería de Telecomunicaciones.

Universidad de Antioquia

Facultad de Ingeniería

Departamento de Ingeniería Electrónica y Telecomunicaciones

Medellín

2023

Cita	Bothert Martínez Manuel José [1]
Referencia	[1] M. Bothert Martínez, “Implementación de un pipeline de seguridad para la detección de vulnerabilidades en aplicaciones”, Semestre de Industria, Ingeniería de Telecomunicaciones, Universidad de Antioquia, Medellín, 2023.

Estilo IEEE (2020)



Repositorio Institucional: <http://bibliotecadigital.udea.edu.co>

Universidad de Antioquia - www.udea.edu.co

Rector: John Jairo Arboleda Céspedes.

Decano/Director: Jesús Francisco Vargas Bonilla.

Jefe departamento: Augusto Enrique Salazar Jiménez.

El contenido de esta obra corresponde al derecho de expresión de los autores y no compromete el pensamiento institucional de la Universidad de Antioquia ni desata su responsabilidad frente a terceros. Los autores asumen la responsabilidad por los derechos de autor y conexos.

Tabla de contenido

Resumen.	6
1. Introducción.	6
2. Objetivos.	8
2.1 Objetivo General.	8
2.2 Objetivos Específicos.	8
3. Marco Teórico.	9
3.1. Build.	10
3.2. Despliegue.	10
3.3. Monitoreo.	10
3.4. Escaneo estático.	11
3.5. Escaneo dinámico.	11
3.6. Contenedores.	12
3.7. Docker.	12
3.7.1. Docker Engine.	13
3.7.2. Docker Images.	14
3.7.3. Dockerfile.	14
3.7.4. Docker registry.	14
3.7.5. Docker Volume.	14
3.8. Kubernetes.	15
3.8.1. Control Plane.	16
3.8.2. Nodos.	17
3.8.3. Kubelet.	17
3.8.4. Kube-proxy.	17
3.8.5. Container runtime.	18
3.8.6. Service discovery y load balancing.	18
3.8.7. Orquestación de almacenamiento.	18
3.8.8. Rollouts y rollbacks automatizados.	18
3.8.9. Empaquetado automático.	19
3.8.10. Auto remediación.	19
3.8.11. Manejo de secretos y configuraciones.	19
3.9. Rancher.	19
3.10. Git.	20
3.11. Gitlab.	20
3.12. Jenkins.	20
3.13. SonarQube.	21

3.14. Owasp Zap.	22
4. Metodología.	23
4.1 Etapa 1: Seleccionar e incorporar los componentes en un pipeline que permita realizar escaneos de vulnerabilidades de forma ágil y continua.	23
4.1.1. Actividad 1.	23
4.1.2. Actividad 2.	23
4.1.3. Actividad 3.	23
4.2. Etapa 2: Integrar una aplicación que sirva para realizar pruebas y validaciones sobre las diversas etapas y componentes definidos para el pipeline.	24
4.2.1. Actividad 4.	24
4.2.2. Actividad 5.	25
4.2.3. Actividad 6.	25
4.3. Etapa 3: Configurar los scanners de vulnerabilidades seleccionados para establecer los criterios que evaluarán el código.	26
4.3.1. Actividad 7.	26
4.3.2. Actividad 8.	26
4.3.3. Actividad 9.	26
4.4. Etapa 4: Analizar los resultados obtenidos en el sistema implementado al someter el mismo a diversas pruebas que permitan garantizar su funcionamiento y mantenibilidad.	27
4.4.1. Actividad 10.	27
4.4.2. Actividad 11.	28
4.4.3. Actividad 12.	29
5. Resultados.	30
6. Conclusiones.	34

Índice de figuras

Figura 1. Ejemplo de fases del modelo DevOps.	8
Figura 2. Arquitectura de Docker.	12
Figura 3. Componentes de un clúster de Kubernetes.	14
Figura 4. Etapas del pipeline construido.	23
Figura 5. Repositorio del proyecto.	23
Figura 6. Frontend de la aplicación desplegada para pruebas.	24
Figura 7. Contenedor de nginx utilizado como ambiente de despliegue.	24
Figura 8. Configuración SonarQube.	25

Figura 9. Evolución del tiempo de ejecución del pipeline.	26
Figura 10. Elementos encontrados en el escaneo dinámico.	26
Figura 11. Elementos encontrados en el escaneo estático.	27
Figura 12. Estructura final del pipeline.	28
Figura 13. Lenguajes habilitados en SonarQube.	29
Figura 14. Habilitación de doble factor de autenticación.	29
Figura 15. Almacenamiento de tokens de acceso entre Jenkins y los componentes del pipeline.	29
Figura 16. Conexión entre servidor de Jenkins y nodos de Kubernetes.	30
Figura 17. Configuración de secretos para acceso al registro de contenedores.	30
Figura 18. Generación de un commit sobre la rama principal del repositorio de Gitlab.	31
Figura 19. Inicialización del Pipeline a partir del push realizado en Gitlab.	32
Figura 20. Logs de la ejecución del pipeline.	32

Índice de tablas

Tabla 1. Componentes principales del pipeline.	23
Tabla 2. Tiempo promedio para la ejecución manual de tareas.	27
Tabla 3. Oportunidades de mejora identificadas.	29

Resumen.

En las últimas décadas hemos sido testigos de la aceleración en los procesos asociados al desarrollo, consumo y mantenimiento de software, lo cual ha representado procesos más dinámicos en el ciclo de vida del software, pero a su vez ha introducido nuevos desafíos frente a la necesidad de garantizar el aseguramiento de aplicaciones que se encuentran en constante cambio. Ante esta problemática han surgido un conjunto de técnicas, métodos, herramientas y prácticas que se integran en el marco de una filosofía de cambio sobre el ciclo de vida del software como unidad, rompiendo barreras entre equipos de desarrollo, operación y seguridad, para trabajar con el objetivo de mantener un flujo continuo en la creación de software seguro y de calidad. A partir de la necesidad de mecanismos que permitan agregar seguridad a los procesos de desarrollo, se realizó un proyecto que consiguió automatizar y agilizar los procesos de identificación de vulnerabilidades sobre aplicaciones mediante la implementación de un conjunto de etapas de seguridad definidas a través de un pipeline que integra herramientas de control de versiones, repositorios, registros, escaneos estáticos y dinámicos.

Palabras clave: Pipeline, seguridad, DevOps, automatización, vulnerabilidad.

1. Introducción.

Aligo Defensores Informáticos es una compañía líder en la industria nacional de ciberseguridad que brinda experiencia y protección a múltiples clientes mediante soluciones que se personalizan para las necesidades específicas o sectores vulnerables del cliente en cuestión. A partir de la constante evolución presentada por las amenazas que ponen en riesgo la seguridad de las empresas, los métodos y herramientas tradicionales de defensa se vuelven insuficientes para proporcionar el nivel de solidez requerido para minimizar el riesgo de ser víctima de ataques realizados por delincuentes informáticos. Gran parte de las brechas de seguridad identificadas tienen como origen aplicaciones que se desarrollan en ambientes en los cuales no existen prácticas de seguridad básicas, lo cual conlleva a un producto final que es fácilmente explotable desde el código fuente. Ante el escenario mencionado, se identifica la necesidad de introducir mecanismos de defensa preventivos que permitan reconocer, categorizar y remediar potenciales amenazas o vulnerabilidades que se encuentren presentes desde fases tempranas del proceso de desarrollo de código, para conseguir minimizar los vectores de ataque presentes en las aplicaciones finales.

En vista de la problemática mencionada, Aligo ha identificado la relevancia de proveer una solución dirigida a compañías o entidades que realicen procesos de desarrollo de código, para brindar a estas la capacidad de identificar potenciales vulnerabilidades desde fases tempranas del

desarrollo hasta el despliegue, permitiendo así dar remediación a las mismas antes de que el producto se lance al público.

Para incursionar en el mercado con un producto o servicio que aborde los requerimientos mencionados, se requiere diseñar un producto mínimo viable que inicialmente se utilizará de forma interna en diversas fases de pruebas que permitirán identificar oportunidades de mejora, previo al lanzamiento de la solución finalizada al mercado. En esta fase inicial del proyecto, se plantea la implementación de un pipeline de CI/CD (integración continua y entrega continua) que posea algunas de las etapas propias del proceso de desarrollo de código y que a su vez integre herramientas que permitan automatizar el proceso de identificación de vulnerabilidades mediante diferentes tipos de escaneos, permitiendo así agregar una capa de seguridad que se adapte a la estructura de un ambiente de DevOps (desarrollo de software y operación de software). Es necesario mencionar que a partir de la filosofía de la compañía, se buscará realizar el proyecto haciendo uso de herramientas open source, dado que estas ofrecen un mayor grado de flexibilidad a la hora de incorporar funcionalidades extra, no requieren licencias o costos para su utilización y a su vez cuentan con comunidades que activamente trabajan de forma conjunta para mejorar las características de las herramientas en uso.

El presente proyecto se plantea como propuesta de práctica académica, buscando dar solución a una problemática actual de Aligo Defensores Informáticos, el cual se realizará en un periodo no mayor a seis meses, cumpliendo con los objetivos y actividades propuestos en este documento.

2. Objetivos.

2.1 Objetivo General.

Implementar un pipeline de seguridad, que integre múltiples herramientas para realizar la detección de vulnerabilidades durante el proceso de desarrollo de aplicaciones Web.

2.2 Objetivos Específicos.

- Seleccionar e incorporar los componentes en un pipeline que permita realizar escaneos de vulnerabilidades de forma ágil y continua.
- Integrar una aplicación que sirva para realizar pruebas y validaciones sobre las diversas etapas y componentes definidos para el pipeline.
- Configurar los scanners de vulnerabilidades seleccionados para establecer los criterios y umbrales que evaluarán el código.
- Analizar los resultados obtenidos en el sistema implementado al someter el mismo a diversas pruebas que permitan garantizar su funcionamiento y mantenibilidad.

3. Marco Teórico.

En años recientes, la gran demanda de servicios en forma de software ha conducido a la búsqueda e implementación de técnicas, metodologías y culturas que fomentan procesos ágiles y eficientes en el desarrollo de software. A partir de esta búsqueda han surgido metodologías ágiles como scrum, XP o kanba, las cuales buscan hacer más rápido el ciclo de vida de desarrollo de software manteniendo unos niveles de alta calidad y permitiendo que los requisitos o soluciones evolucionen de manera iterativa según la necesidad de los proyectos

Además de la implementación de metodologías ágiles, la creciente necesidad de software ha conllevado a la generación de prácticas específicas orientadas a la integración y entrega continua del producto desarrollado, dichas prácticas se conocen comúnmente mediante el nombre DevOps y actualmente han convertido en un estándar seguido por los líderes de la industria, los cuales lo han llevado de un conjunto de prácticas a un cambio completo sobre la cultura de desarrollo [1].

DevOps integra los mundos previamente separados de desarrollo y operación; para esto hace uso de procesos como los ilustrados en la **Figura 1**, los abarcan la automatización en el desarrollo, despliegue y monitoreo de infraestructura, los cuales en vez de ser realizados por agentes aislados de una organización, son distribuidos entre equipos con funciones conjuntas que trabajan en pro de realizar entregas de características operacionales de manera continua [2]. Mediante el esfuerzo conjunto de los equipos de trabajo mencionados, se consiguen mejoras sustanciales en las diversas etapas que conforman el ciclo de vida del software, como el tiempo detección y corrección de errores, la agilización de actualizaciones, la identificación de estadísticas de rendimiento, entre otros.

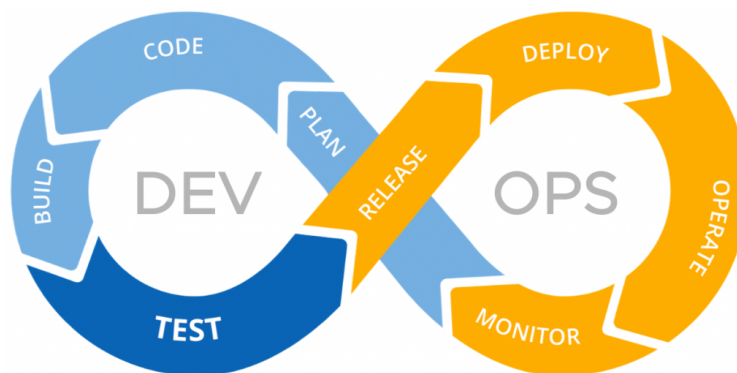


Figura 1. Ejemplo de fases del modelo DevOps.

El modelo de DevOps ofrece la ventaja de ser flexible y adaptable a diferentes tipos de estructuras organizacionales, ambientes de desarrollo, tipos de aplicaciones, entornos de despliegue, etc. Por esta razón, dependiendo de la estructura y necesidad de una organización este

puede consistir en diversas fases, como las presentadas en la figura 1. A pesar de la adaptabilidad del modelo descrito, normalmente este cuenta con un conjunto de fases básicas que conforman la estructura cíclica y continua del modelo. Dichas fases se describen a continuación:

3.1. Build.

En el contexto del desarrollo de software, “build” hace referencia a el proceso que convierte código fuente, archivos y otros activos generados o utilizados por los desarrolladores, en partes de una aplicación que se construye en una instancia de pruebas, con el objetivo de facilitar mecanismos de pruebas, detección de errores y bugs sobre el código. Generalmente en esta fase se utilizan herramientas que ofrecen capacidades de compilación, manejo de dependencias, generación de documentación, ejecución de pruebas, entre otros.

3.2. Despliegue.

Durante esta fase se utiliza el código construido en las etapas previas y se lanza el mismo sobre un sistema que simule de la mejor forma posible el ambiente de producción para el el cual se tiene destinada la aplicación. De esta forma se consigue emular el funcionamiento de la misma y se puede someter a pruebas de estrés que permitan identificar áreas de mejora de forma temprana y antes de que la aplicación se lance al público.

3.3. Monitoreo.

Esta etapa tiene como función principal obtener datos, estadísticas y métricas de la aplicación previamente desplegada, que permitan identificar posibles elementos de interés que puedan ser sujetos a cambios, esto con el objetivo de generar un ciclo de entrega constante que contribuya al mejoramiento del software desarrollado.

Las fases descritas son solo representan los componentes básicos de un ciclo de DevOps; Dada la naturaleza del modelo, diferentes organizaciones pueden hacer uso de estas fases, agregar nuevos componentes o componentes a estas para poder cumplir el objetivo de estructurar un ciclo de vida de desarrollo más ágil y eficiente.

A pesar de las ventajas descritas, el modelo de DevOps ha introducido nuevas problemáticas, una de las más desafiantes consiste en el aseguramiento del ciclo de vida del desarrollo, ya que al incrementar la velocidad de dicho ciclos y automatizar gran parte de las tareas que lo conforman, garantizar la protección ante amenazas de cada una de estas fases se convierte en una tarea ardua.

En vista de la problemática planteada, existen alternativas que pueden ofrecer una solución en pro de garantizar la seguridad en el desarrollo realizado. Una de las propuestas más interesantes consiste en la inclusión de verificaciones o escaneos de seguridad automatizados, que se ejecuten en paralelo con las diferentes fases del ciclo de DevOps, o incluso, como parte esencial de dicho ciclo. Al igual que con las fases del modelo de DevOps, las etapas o verificaciones de seguridad pueden variar de manera significativa dependiendo de la estructura organizacional, el tipo de aplicación, lenguaje de programación utilizado, ambiente de despliegue, entre otros. A pesar del amplio espectro de posibilidades que se tiene entre las verificaciones de seguridad que se pueden implementar, a continuación se presentan algunos mecanismos básicos que deberían estar presentes en los procesos de desarrollo para garantizar un mínimo de resiliencia contra posibles vulnerabilidades o amenazas.

3.4. Escaneo estático.

El análisis de código estático, también conocido como Prueba de seguridad de aplicaciones estáticas (SAST), es una metodología de escaneo de vulnerabilidades diseñada para trabajar en el código fuente en lugar de un ejecutable compilado. Las herramientas de análisis de código estático inspeccionan el código en busca de indicaciones de vulnerabilidades comunes, que luego se corrigen antes de que se lance la aplicación.

3.5. Escaneo dinámico.

Las pruebas dinámicas de seguridad de aplicaciones (DAST) consisten en el proceso de análisis para una aplicación para encontrar vulnerabilidades a través de ataques simulados. Este tipo de enfoque evalúa la aplicación desde "afuera hacia adentro" atacando una aplicación como lo haría un usuario malintencionado. Después de que un escáner DAST realiza los ataques determinados, este busca resultados que no representan comportamientos normales e identifica vulnerabilidades de seguridad.

Los escaneos descritos pueden integrarse de forma automática sobre un ambiente de desarrollo mediante la implementación de un pipeline, el cual básicamente consiste en un conjunto de procesos y herramientas automatizados que los equipos de desarrollo y operaciones utilizan para compilar, construir, probar, implementar y asegurar código de software de forma rápida y sencilla. Mediante los mecanismos presentados se tiene una base sólida de la cual partir para establecer un proceso de robustecimiento sobre la arquitectura de un ciclo de vida de desarrollo que permita identificar de manera temprana posibles vulnerabilidades presentes en la aplicación a desplegar.

Una vez descritas las etapas y escaneos que conforman el pipeline, es necesario establecer el ambiente base que soportará las diversas herramientas y servicios que se utilizarán en la implementación. El criterio para la selección de dicho ambiente se basó principalmente en 2 puntos: El rendimiento en manejo de recursos y que la tecnología escogida califique como open source.

A partir de los 2 puntos descritos, se decidió optar por la utilización de docker, kubernetes y rancher como plataformas de para la gestión de microservicios, ciclo de vida de contenedores y orquestación de clusters, los cuales se utilizarán durante la integración del sistema a realizar. Para comprender de mejor manera la interacción entre las tecnologías seleccionadas se hace necesario definir los siguientes conceptos:

3.6. Contenedores.

Los contenedores son tecnologías de virtualización a nivel de núcleo de sistema operativo, los cuales permiten que existan múltiples instancias aisladas de espacios de usuario. Estas instancias garantizan mecanismos de aislamiento entre las aplicaciones que se ejecuten sobre ellas y hacen uso del kernel del sistema para proporcionar métodos de administración de recursos que permitan limitar el impacto de las actividades de un contenedor sobre otros contenedores [3].

La ventaja de los contenedores sobre otras tecnologías de virtualización es que mediante estos, resulta muy fácil compartir recursos de CPU, memoria, almacenamiento y red a nivel de sistema operativo. Además, constituyen un mecanismo de empaquetado lógico en el que se pueden abstraer las aplicaciones del entorno en el que se ejecutan realmente, permitiendo así hacer un mejor uso de los recursos del sistema operativo base.

3.7. Docker.

Docker es una plataforma de gestión de contenedores, de código abierto y que es utilizada para desarrollar, implementar y administrar aplicaciones en entornos virtualizados livianos.

Docker se utiliza principalmente como una plataforma de administración y desarrollo de software, en la cual se despliegan aplicaciones que funcionan de manera eficiente independiente del entorno sobre el cual sean desplegadas. Las ventajas principales de Docker como tecnología son que al hacer que el sistema de software sea agnóstico, los desarrolladores no tienen que preocuparse por los problemas de compatibilidad entre librerías y dependencias, lo cual facilita el desarrollo, la implementación, el mantenimiento y el uso de aplicaciones [4].

Componentes principales de docker:

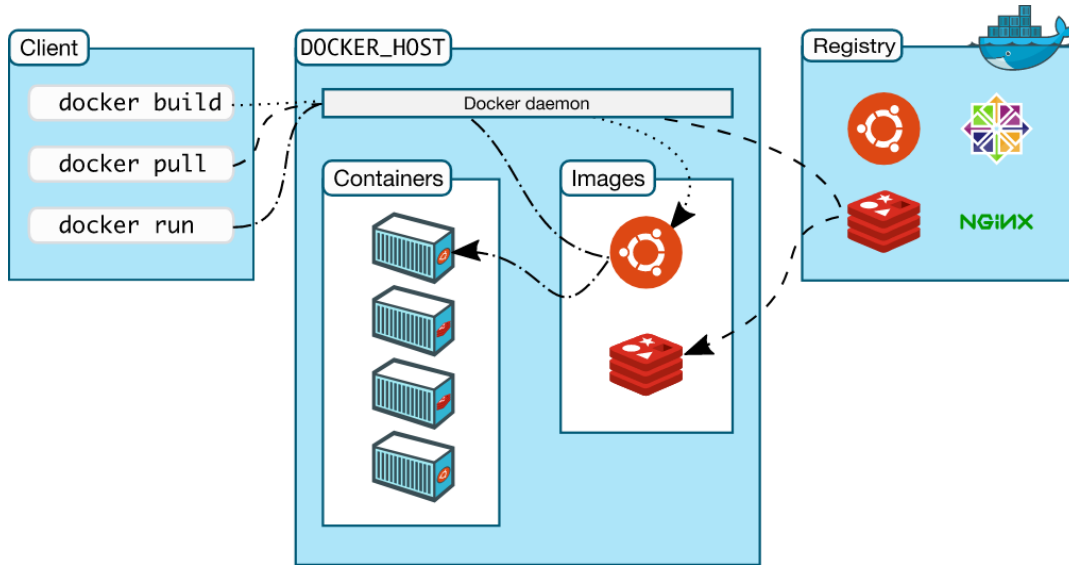


Figura 2. Arquitectura de Docker.

Como tecnología, docker está conformada por un conjunto de componentes que llevan a cabo un rol sobre el funcionamiento de la plataforma [5]. A continuación se definen los componentes mencionados:

3.7.1. Docker Engine.

El motor Docker (Docker Engine o DE) representa el núcleo de Docker como sistema, es instalado sobre el OS base y básicamente consiste en un sistema de tiempo de ejecución liviano que actúa bajo el modelo cliente-servidor para crear y administrar contenedores.

Docker engine se compone de 3 componentes:

- Un servidor que ejecuta un proceso de tipo daemon, denominado docker, el cual es responsable de crear y administrar los contenedores.
- Una API de tipo Rest, la cual especifica las interfaces que los programas o aplicaciones pueden usar para comunicarse con el docker daemon.
- Una interfaz de línea de comandos utilizada para ejecutar los comandos que permiten al usuario gestionar el funcionamiento de docker.

3.7.2. Docker Images.

Una imagen Docker es una plantilla de solo lectura con instrucciones para crear un contenedor Docker. Estas imágenes son archivos inmutables que consisten en el código fuente, librerías, dependencias, herramientas y cualquier otro tipo de configuración necesaria para

ejecutar una determinada aplicación. A menudo, una imagen se basa en otra imagen, con alguna personalización adicional. Por ejemplo, puede crear una imagen que se base en la imagen de ubuntu, pero instale el servidor web Apache así como los detalles de configuración requeridos. Las imágenes son de gran utilidad ya que estas aceleran los procesos de construcción de contenedores, son reutilizables y disminuyen el uso de disco.

3.7.3. Dockerfile.

También llamado archivo de docker, es un script que consiste en un conjunto de instrucciones que se utilizan para construir una imagen. Estas instrucciones incluyen especificaciones relacionadas al sistema operativo, lenguaje, variables de ambiente, ubicaciones de archivos, puertos de red, configuración de dependencias, librerías y cualquier otro tipo de componente necesitado para ejecutar una imagen. Los comandos en este archivo se escriben de manera secuencial, y son agrupados y ejecutados automáticamente.

3.7.4. Docker registry.

Un registro docker es un espacio designado para almacenar imágenes docker. Estos espacios pueden estar alojados en diferentes ambientes, ya sean en la nube como registros públicos a los que cualquier usuario puede acceder para subir sus imágenes o descargar imágenes de la comunidad, o registros privados creados de manera local cuando se tienen contenido confidencial o imágenes personalizadas que no pueden ser compartidas con el público.

3.7.5. Docker Volume.

Un volumen de docker es un mecanismo utilizado para resolver una de las limitaciones presentadas por los contenedores, el almacenamiento. Debido a su naturaleza ágil, portable y volátil, los contenedores no suelen ser apropiados para almacenar información, ya que esta desaparece una vez que el proceso de ejecución del contenedor se detiene o reinicia. Para afrontar esta limitación existen los volúmenes, los cuales permiten al usuario crear un enlace entre el almacenamiento interno del contenedor, con un espacio externo en el cual se repliquen los archivos de forma que estos perduren, puedan ser compartidos entre contenedores, aplicaciones externas o utilizados para otros fines.

Una vez definidos los conceptos básicos necesarios para comprender la naturaleza y utilidad que la tecnología proporcionada por docker prestará en el proyecto, es necesario abordar la problemática de la gestión de las cargas de trabajo que se desplegarán sobre los contenedores; ya que al aumentar el número de elementos desplegados, se hace necesario considerar las implicaciones en términos de consumo de recursos, ancho de banda, direccionamiento, acceso a puertos, manejo de volúmenes, entre otros. Para proporcionar una herramienta que aborde de forma directa muchas de estas implicaciones, se decide por hacer uso de Kubernetes.

3.8. Kubernetes.

Kubernetes consiste en un sistema de código abierto diseñado para implementar, escalar y administrar aplicaciones o cargas de trabajo alojadas en contenedores, ya sea mediante configuraciones declarativas o automatización [6].

Al desplegar Kubernetes, obtenemos un sistema distribuido denominado cluster. Un cluster de Kubernetes consiste en un conjunto de equipos denominados nodos, los cuales se encargan de ejecutar aplicaciones desplegadas en forma de contenedores. Los equipos que ejecutan los contenedores se denominan workers. Estos hacen uso de pods, los cuales consisten en la unidad de computación más pequeña que se puede desplegar, crear y administrar en kubernetes; estos se representan mediante un grupo de uno o más contenedores, con recursos compartidos de red, almacenamiento y archivos de configuración asociados a la ejecución de los contenedores [7].

Componentes de un clúster de Kubernetes:

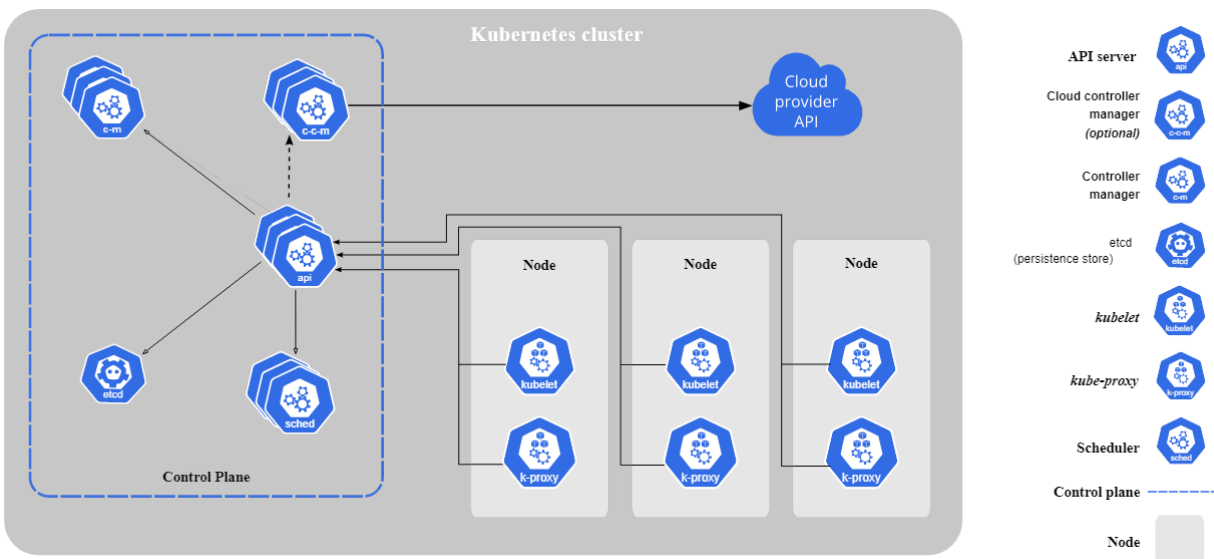


Figura 3. Componentes de un clúster de Kubernetes.

Los clusters de kubernetes pueden tener arquitecturas diferentes dependiendo de varios factores como el ambiente de despliegue o los sistemas base utilizados, sin embargo, existen 2 componentes principales que están presentes en todo cluster de Kubernetes, el control plane y los nodos [8].

3.8.1. Control Plane.

El control plane consiste en un conjunto de elementos que toman decisiones globales sobre el cluster, detectan y responden a eventos del cluster como inicialización de pods para generar réplicas de despliegues o cargas de trabajo.

Los componentes del control plane se pueden ejecutar sobre cualquiera de los equipos que sean parte del cluster. A pesar de esto, por simplicidad, los scripts de setup utilizados durante la instalación suelen instalar todos los componentes del control plane en un mismo equipo.

- **Kube-apiserver:** Consiste en un componente del control plane que tiene como función exponer la API de Kubernetes, la cual básicamente consiste API de tipo REST, que permite realizar consultas desde los nodos hacia el control plane, estas respuestas contienen información asociada a validaciones y configuraciones de objetos como pods, servicios, réplicas, entre otros.
- **etcd:** Consiste en un sistema distribuido de almacenamiento de valores de llaves, este suele ser utilizado para procesos de descubrimiento de servicios, coordinación de sistemas distribuidos, almacenamiento de estados, entre otros. Etcd es un componente clave de kubernetes ya que este permite distribuir datos asociados a configuraciones, permitiendo así que se provea redundancia y resiliencia de la información que requieren las cargas de trabajo desplegadas sobre los nodos.
- **Kube-scheduler:** Es un componente del control plane que constantemente se encuentra observando pods recientemente creados y que no posean un nodo asignado, al identificar un pod que cumpla con estas condiciones, el kube-scheduler selecciona y asigna un nodo que cumpla con los requisitos necesarios para garantizar la correcta ejecución del pod. Para realizar la decisión asociada a la selección y asignación de los nodos, se tienen en cuenta aspectos como requisitos colectivos e individuales de recursos, restricciones de hardware, software o políticas, especificaciones de afinidad, localización de datos, interferencia entre cargas de trabajo, entre otros.
- **Kube-controller-manager:** Es un componente del control plane que permite administrar unidades denominadas controllers. Los controladores son procesos separados requeridos para la ejecución de una tarea específica, pero para reducir complejidad, estos son

agrupados y compilados en un archivo binario que se ejecuta como una unidad que contiene un conjunto de procesos. A partir de su función, los controladores se pueden clasificar en diferentes tipos, los cuales son:

- **Controlador de nodo:** Es el responsable de ejecutar procesos encargados de notificar y responder cuando alguno de los nodos pertenecientes al clúster cambia de estado.
- **Controlador de trabajos:** Se encarga de observar los objetos de trabajo que representan tareas únicas y luego crea pods para ejecutar esas tareas hasta su finalización.
- **Controlador EndpointSlice:** Llena los objetos EndpointSlice (para proporcionar un vínculo entre los servicios y los pods).
- **Controlador ServiceAccount:** Crea cuentas de servicio predeterminadas para nuevos espacios de nombres.

3.8.2. Nodos.

Un nodo es la denominación que recibe una máquina de tipo Worker sobre un cluster de Kubernetes. Un nodo puede tener varios pods, los cuales son gestionados de forma automática por el control plane, el cual tiene en cuenta los recursos disponibles para realizar la distribución de los pods sobre los nodos.

Al igual que el control plane, los nodos también tienen componentes que llevan a cabo labores específicas como mantener los pods en ejecución o proveer el entorno de tiempo de ejecución de Kubernetes.

3.8.3. Kubelet.

Es un agente que se ejecuta en cada nodo que forma parte del clúster y cuya función es asegurarse de que los contenedores estén corriendo en un pod. Para esto, Kubelet toma el grupo de especificaciones que se proveen mediante varios mecanismos y se asegura de que los contenedores descritos en estas especificaciones se encuentren en ejecución y con un nivel de salubridad estable.

3.8.4. Kube-proxy.

Es un proxy de red que se ejecuta sobre cada nodo que forma parte de cluster, conformando así parte fundamental del concepto asociado al servicio de kubernetes. Kube-proxy también se encarga de mantener las reglas de red que se aplican sobre los nodos, estas reglas permiten la comunicación hacia los pods desde sesiones internas o externas al cluster, para esto kube-proxy

hace uso de la capa de filtro de paquetes del sistema operativo (si este tiene una disponible), o realiza el proceso de reenviar el tráfico por sí mismo.

3.8.5. Container runtime.

El container runtime o container engine, es un software responsable de administrar la ejecución de los contenedores que se despliegan sobre los nodos, esto incluye los procesos de carga de imágenes desde repositorios, monitoreo de sistemas de recursos locales, selección de recursos del sistema para el uso de contenedores y manejo de ciclo de vida de contenedores.

Los container runtimes comunes suelen funcionar junto con los orquestadores de contenedores. El orquestador es responsable de administrar los clústeres de contenedores, ocupándose de cuestiones como la escalabilidad, las redes y la seguridad de los contenedores, mientras que el container runtime asume la responsabilidad de administrar los contenedores individuales que se ejecutan en cada nodo del clúster.

Una vez definido Kubernetes, y sus componentes, se puede profundizar en algunas de las funcionalidades que provee Kubernetes, como lo son:

3.8.6. Service discovery y load balancing.

Kubernetes puede exponer un contenedor haciendo uso de un nombre DNS o una dirección IP específica; También permite balancear cargas de tráfico de red alto que sean dirigidas hacia contenedores, consiguiendo así mejorar la estabilidad y rendimiento de estos.

3.8.7. Orquestación de almacenamiento.

Kubernetes permite a los usuarios montar de manera ágil y simple, sistemas de almacenamiento de nuestra escogencia. Estos pueden ser volúmenes locales, proveedores de nubes públicas, privadas, entre otros.

3.8.8. Rollouts y rollbacks automatizados.

Los procesos de rollout y rollback consisten en describir de forma declarativa el estado al cual se desean llevar los contenedores desplegados, para que kubernetes pueda cambiar del estado actual al deseado de manera controlada. Un ejemplo de estos procesos puede ser automatizar Kubernetes para crear un nuevo contenedor para un despliegue, remover contenedores previamente existentes y adoptar los recursos de estos para el nuevo contenedor.

3.8.9. Empaquetado automático.

Kubernetes puede adaptarse a las necesidades del usuario en términos de consumo de recursos (CPU, RAM, etc) y distribuir las cargas de trabajo sobre los nodos del cluster, de manera que se mantenga un balance entre los límites establecidos y el rendimiento de dichas cargas.

3.8.10. Auto remediación.

Kubernetes tiene la capacidad de automáticamente reemplazar, eliminar o reiniciar contenedores que se encuentren en estados fallidos o que no respondan a los chequeos de salubridad programados por el usuario.

3.8.11. Manejo de secretos y configuraciones.

Kubernetes permite almacenar y administrar información sensible como contraseñas, tokens de autenticación, llaves de ssh, llaves de apis, certificados, entre otros. Se pueden desplegar y actualizar secretos y configuraciones de aplicación sin necesidad de reconstruir las imágenes utilizadas para crear los contenedores y sin exponer la información sensible en la configuración del stack.

Una vez descritas las tecnologías que gestionarán el ciclo de vida de contenedores y la administración de las cargas de trabajo, sólo resta establecer la tercera herramienta que formará parte del stack sobre el cual se podrá conformar el ambiente de trabajo en el cual se desplegaran los componentes del pipeline.

3.9. Rancher.

La herramienta seleccionada es Rancher, que consiste en una herramienta de administración de Kubernetes, permitiendo desplegar y ejecutar clústers en múltiples tipos de ambientes, de manera ágil y sencilla.

Rancher agrega un valor agregado sobre la infraestructura de Kubernetes, esto se consigue al agregar funcionalidades de centralización de autenticación y control de acceso basado en roles sobre los clusters, lo cual permite tener mayor capacidad de limitar los permisos y accesos a los recursos desplegados. Además de la funcionalidad descrita, Rancher también permite realizar procesos de monitoreo detallado sobre las alertas de los clusters y sus recursos, permite enviar registros a proveedores externos y se puede integrar directamente con herramientas externas para conseguir que el despliegue de una carga de trabajo preconfigurada se convierte en algo tan simple como agregar una librería e instalarla. Finalmente, otra de las funcionalidades de Rancher

que representa un gran valor para el proyecto a desarrollar es la simplificación que este ofrece a la hora de administrar un cluster de Kubernetes, ya que mediante Rancher podemos llevar a cabo tareas como eliminación de cargas de trabajo, reinicio de pods, agregación de secretos, generación de réplicas, etc, desde una interfaz gráfica intuitiva y amigable con el usuario final, que reemplaza acciones que previamente sólo podían ser realizadas desde interfaz de línea de comandos, lo anterior representa un gran valor y refuerza la visión de Rancher como plataforma completa de administración de contenedores para Kubernetes [9].

Una vez descritos los diferentes componentes que conformarán el stack de tecnología utilizado para el ambiente sobre el cual se desplegará el pipeline, se hace necesario describir las herramientas que serán utilizadas para implementar el pipeline. Dado que el pipeline realizará escaneos estáticos sobre el código fuente de una aplicación, el primer componente a definir son las herramientas de control de versiones y alojamiento de repositorios que se utilizarán.

3.10. Git.

Git es un sistema de control de versiones utilizado para dar seguimiento a cambios que se realicen sobre diversos tipos de archivos. El propósito principal de Git es administrar cualquier tipo de cambio efectuado sobre uno o más proyectos para un rango de tiempo específico, de esta forma consigue representar una gran ayuda a la hora de coordinar trabajo entre miembros de un proyecto y proporcionar mecanismos para visualizar el progreso de un proyecto sobre el tiempo.

3.11. Gitlab.

GitLab, un paquete de software DevOps, Open Source, que combina la capacidad de desarrollar, asegurar y operar software en una sola aplicación. Funciona como un repositorio de Git alojado en web, el cual proporciona espacios de trabajo abiertos y privados, capacidades de seguimiento de issues y wikis. Es una plataforma DevOps completa que permite a los profesionales realizar todas las tareas de un proyecto, desde la planificación del proyecto y la gestión del código fuente hasta la supervisión y la seguridad. Además, permite que los equipos colaboren , agilicen y mejoren los procesos asociados al desarrollo de software [10].

3.12. Jenkins.

Jenkins es una herramienta de CI/CD (continuous integration and continuous delivery) de código abierto que se puede utilizar para automatizar todo tipo de tareas relacionadas con la creación, prueba y entrega o implementación de software. Jenkins ofrece gran flexibilidad ya que puede

instalarse a través de paquetes de sistema nativos, Docker o incluso puede ejecutarse de forma independiente en cualquier máquina con Java Runtime Environment (JRE) instalado [11].

Los procesos de automatización se consiguen mediante la implementación de pipelines, los cuales básicamente consisten en scripts que se ejecutan desde el servidor de Jenkins con el objetivo de realizar tareas específicas que puedan ser repetitivas. Al combinar estos scripts con la gran cantidad de integraciones en forma de plugins que se encuentran disponible para Jenkins, se obtiene una herramienta capaz de automatizar procesos ejecutados por programas o sistemas externos, lo cual puede simplificar de forma significativa diversos procesos.

Dependiendo de la necesidad del usuario, Jenkins puede ser utilizado para tareas de diversos grados de complejidad. A pesar de esto, para los procesos asociados al desarrollo de software existen una serie de pasos que describen la interacción entre diferentes elementos en Jenkins, estos pasos son:

- Los desarrolladores realizan las modificaciones necesarias en el código fuente y confirman los cambios en el repositorio. Se creará una nueva versión de ese archivo en el sistema de control de versiones que se utiliza para mantener el repositorio de código fuente.
- El servidor Jenkins CI verifica continuamente el repositorio en busca de cambios (ya sea en forma de código o bibliotecas) y el servidor extrae los cambios.
- En el siguiente paso, el servidor realiza una compilación con el código y se genera un ejecutable si el proceso es exitoso. En caso de que se produzca un error, se genera una notificación automatizada con un enlace a los registros o elementos asociados al fallo detectado.
- En caso de una compilación exitosa, la aplicación generada (o ejecutable) se despliega en un servidor o ambiente de prueba. Este paso ayuda a realizar pruebas continuas en las que el la aplicación recién creada pasa por una serie de pruebas automatizadas.
- Si no hay problemas de compilación, integración y prueba con el código registrado, los cambios y la aplicación probada se implementan automáticamente en un servidor de Producción o Pre-Producción.

3.13. SonarQube.

SonarQube es una herramienta de código abierto desarrollada por SonarSource y cuyo objetivo es servir como plataforma integra para la inspección continua de la calidad del código mediante la realización de escaneos estáticos sobre el código, permitiendo detectar errores, vulnerabilidades,

code smells, malas prácticas de desarrollo y bugs en 29 lenguajes de programación [12]. Además de las características mencionadas, SonarQube permite generar de forma automática reportes sobre elementos duplicados en el código, pruebas unitarias, estado de cobertura de código, complejidad de código, comentarios, recomendaciones generales, entre otros.

La arquitectura de SonarQube como herramienta puede describirse como la conjunción de 4 partes principales:

- Servidor: Denominado SonarServer, consiste en un servidor de búsquedas que permite a los usuarios realizar consultas, un compute engine encargado de procesar y guardar informes de análisis de código y un servidor web que se utiliza como interfaz gráfica, gestor de instantáneas y gestor de configuraciones
- Base de datos: Almacena las configuración asociadas a seguridad, ajustes de plugins, estado de proyectos, vistas, perfiles, entre otros.
- Plugins: Ofrecen funcionalidades y complementos como idiomas, Source Code Management, integraciones, opciones adicionales de autenticación, entre otros.
- Scanner: Es el componente que se encarga de realizar el escaneo estático sobre el código fuente, aplicando las diversas reglas o parámetros configurados y generando una calificación del estado del código a partir de las directivas aplicadas.

3.14. Owasp Zap.

También conocido como Zap es una herramienta diseñada para el escaneo de aplicaciones web y pentesting, diseñada y mantenida por OWASP (Open Web Application Security Project). Es una de las herramientas open source más populares y utilizadas del mundo, cuenta con un equipo de voluntarios que se encargan de mantener, actualizar y agregar de manera continua funcionalidades nuevas a esta herramienta.

Para realizar el escaneo dinámico de las vulnerabilidades presentes en una aplicación web, Zap funciona como un proxy que se posiciona entre la aplicación web y el navegador del cliente, de esta forma Zap consigue capturar la información dirigida y proveniente de la aplicación web para determinar cómo esta responde a posibles peticiones maliciosas. Algunas de las funcionalidades que ofrece la herramienta son escaneos automáticos, escaneos pasivos, fuzzing, soporte para websockets, web crawlers y un marketplace con extensiones para personalizar o agregar funcionalidades adicionales [13].

4. Metodología.

4.1 Etapa 1: Seleccionar e incorporar los componentes en un pipeline que permita realizar escaneos de vulnerabilidades de forma ágil y continua.

4.1.1. Actividad 1.

Establecer el tipo de ambiente en el cual se desplegará el pipeline: Se definió un ambiente virtualizado basado en las tecnologías de docker, kubernetes y rancher. La implementación del entorno descrito no requirió inversión adicional de tiempo o recursos económicos, ya que la empresa cuenta con un laboratorio construido sobre los componentes mencionados, por lo que se decidió utilizar este espacio para el despliegue realizado.

4.1.2. Actividad 2.

Caracterizar los componentes y herramientas que se utilizarán en el pipeline: Se establecieron las características, funcionalidad y criterio de selección de los diversos componentes empleados para conformar el pipeline desarrollado.

Tabla 1. Componentes principales del pipeline.

Componente	Función
Jenkins	Herramienta de CI/CD, representa una plataforma centralizada desde la cual se conectan los diversos componentes del pipeline y se generan los scripts que automatizan el proceso de escaneo.
Gitlab	Repositorio de código, archivos y contenedores requeridos para el despliegue de la aplicación.
SonarQube	Escaner estático utilizado para realizar pruebas automáticas sobre el código fuente de la aplicación.
Contenedor Nginx	Simula un servidor HTTP sobre el cual se despliega la aplicación web construida.
OWASP Zap	Escaner dinámico utilizado para realizar pruebas automáticas sobre la aplicación previamente desplegada.

4.1.3. Actividad 3.

Definir las etapas o procesos que se ejecutarán sobre el pipeline: Se definió un pipeline compuesto por 5 etapas, evidenciadas en la *Figura 4*, las cuales abarcan los procesos básicos de

la metodología devops. Dichas etapas son: Copiado de repositorio, construcción de imagen, escaneo estático, despliegue de aplicación y escaneo dinámico.

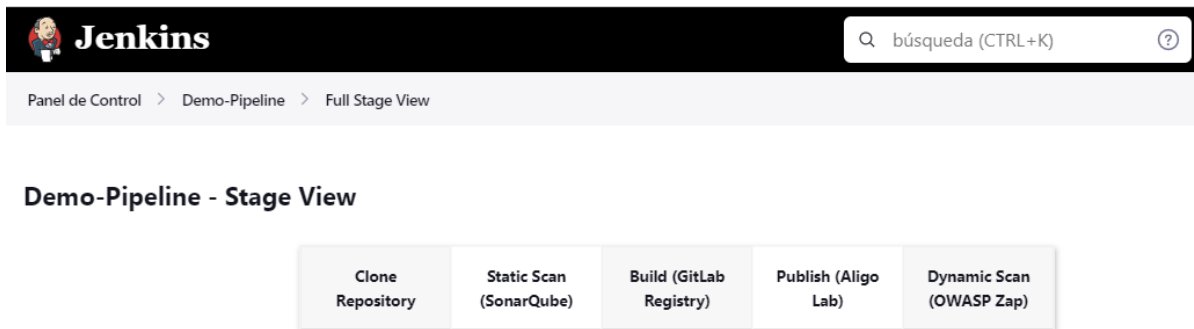


Figura 4. Etapas del pipeline construido.

4.2. Etapa 2: Integrar una aplicación que sirva para realizar pruebas y validaciones sobre las diversas etapas y componentes definidos para el pipeline.

4.2.1. Actividad 4.

Incorporar una herramienta de control de versiones para alojar el código fuente de la aplicación: Se realizaron las configuraciones necesarias para alojar el código fuente, imágenes docker y archivos manifiestos utilizados en el repositorio de Gitlab mostrado en la *Figura 5*, el cual fue previamente desplegado sobre la infraestructura empresarial.

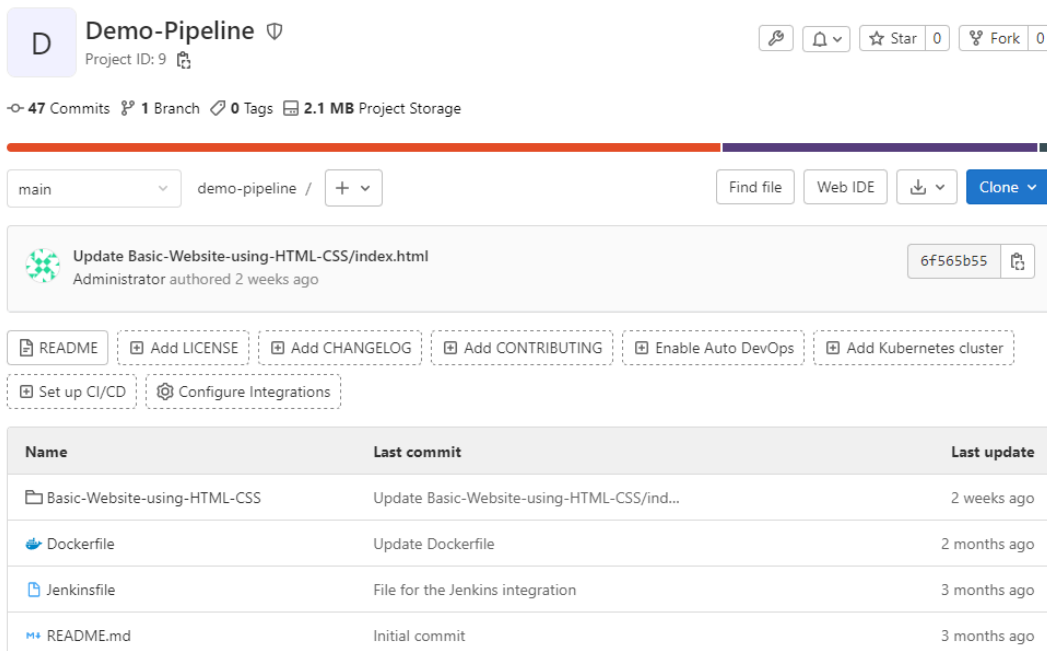


Figura 5. Repositorio del proyecto.

4.2.2. Actividad 5.

Determinar el tipo de aplicación y lenguaje de programación a utilizar para la aplicación de prueba: Se desplegó una aplicación simple y liviana consistente en un frontend mostrado en la **Figura 6**, el cual está basado en HTML, CSS y JavaScript. El criterio de selección de la aplicación se basó en establecer un elemento funcional y liviano que permitiese disminuir los tiempos en las iteraciones iniciales de las pruebas y configuraciones realizadas.

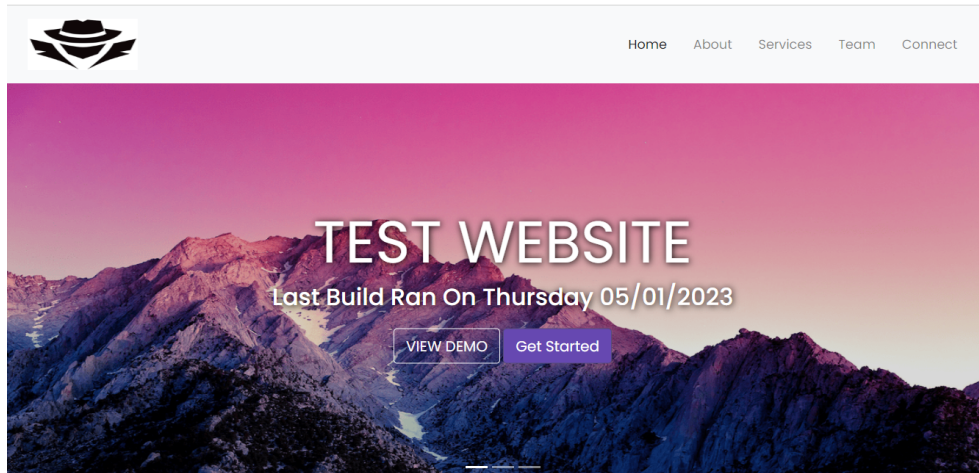


Figura 6. Frontend de la aplicación desplegada para pruebas.

4.2.3. Actividad 6.

Definir el ambiente sobre el cual se desplegará la aplicación construida: Se selecciona un ambiente virtualizado representado mediante un contenedor de nginx el cual se desplegó sobre la infraestructura del laboratorio de Aligo Defensores Informáticos tal y como se observa en la **Figura 7**; Dicho contenedor se configuró como un servidor web que aloja la aplicación construida.

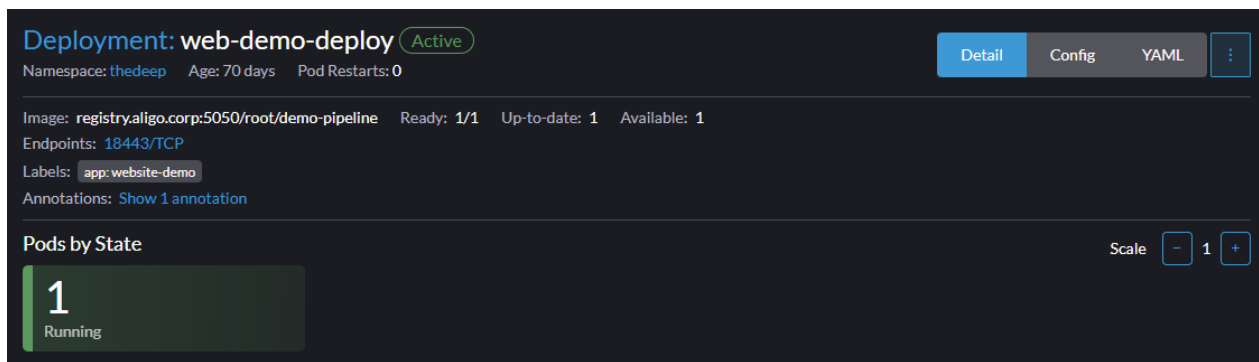


Figura 7. Contenedor de nginx utilizado como ambiente de despliegue.

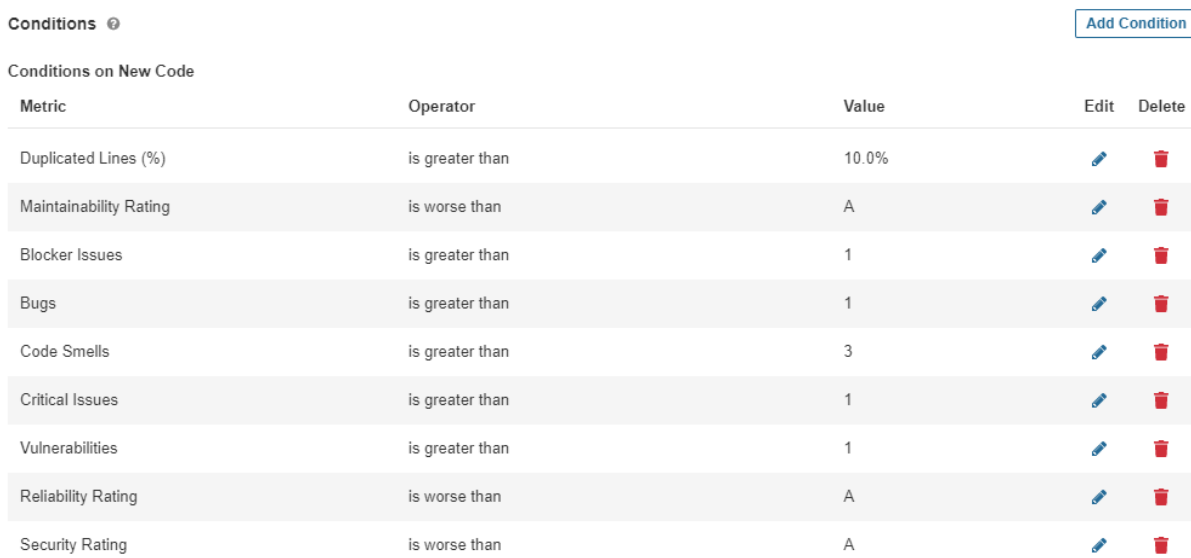
4.3. Etapa 3: Configurar los scanners de vulnerabilidades seleccionados para establecer los criterios que evaluarán el código.

4.3.1. Actividad 7.

Generar las API Keys necesarias para la comunicación entre las diversas herramientas seleccionadas: Se generaron las API Keys, llaves de ssh, certificados y demás controles necesarios para conectar los diversos componentes del pipeline con el servidor de Jenkins de manera segura.

4.3.2. Actividad 8.

Aplicar los umbrales que validarán las métricas de calidad del código: En la *Figura 8* se muestran las configuraciones realizadas en SonarQube para los perfiles y reglas a aplicar sobre la evaluación del código durante el escaneo estático.



The screenshot shows the 'Conditions' section in SonarQube. It features a table titled 'Conditions on New Code' with columns for Metric, Operator, Value, Edit, and Delete. There are 9 rows of conditions, each with a blue pencil icon for editing and a red trash icon for deleting. An 'Add Condition' button is located in the top right corner.

Metric	Operator	Value	Edit	Delete
Duplicated Lines (%)	is greater than	10.0%		
Maintainability Rating	is worse than	A		
Blocker Issues	is greater than	1		
Bugs	is greater than	1		
Code Smells	is greater than	3		
Critical Issues	is greater than	1		
Vulnerabilities	is greater than	1		
Reliability Rating	is worse than	A		
Security Rating	is worse than	A		

Figura 8. Configuración SonarQube.

4.3.3. Actividad 9.

Definir las acciones que tomará el pipeline cuando el código evaluado no cumpla con los requisitos establecidos: Se utilizaron los umbrales de validación de código para activar la acción de detener el pipeline al encontrar vulnerabilidades graves o no superar los valores establecidos sobre el código fuente analizado.

4.4. Etapa 4: Analizar los resultados obtenidos en el sistema implementado al someter el mismo a diversas pruebas que permitan garantizar su funcionamiento y mantenibilidad.

4.4.1. Actividad 10.

Obtener métricas asociadas al tiempo de ejecución de los stages del pipeline para comparar estas con la realización manual de los escaneos: Tal y como se observa en la *Figura 9* y la *Tabla 2*, se organizaron las métricas asociadas a los tiempos de ejecución, tanto para la realización manual de las tareas requeridas para efectuar los escaneos como para la automatización conseguida mediante el pipeline.

Tabla 2. Tiempo promedio para la ejecución manual de tareas.

Etapa	Tareas	Tiempo promedio
Clonación del repositorio	Acceso a la VPN, acceso SSH al nodo, login al repositorio, selección de versión, copiado de archivos, logout del repositorio y cierre de conexión ssh al nodo.	1 min 15 s
Escaneo estático	Acceso a interfaz gráfica de SonarQube, proceso de carga del código modificado, ejecución del escaneo, descarga de resultados y cierre de sesión.	1 min 30 s
Build	Acceso SSH al nodo, login al registry, construcción de la imagen, push de la imagen al repositorio, logout del registry y cierre de conexión SSH al nodo.	56 s
Publish	Acceso SSH al control plane, selección del namespace en Kubernetes, modificación de la carga de trabajo, reinicio de la carga de trabajo y cierre de conexión SSH al control plane.	1 min 3 s
Escaneo dinámico	Acceso SSH al nodo, selección del contenedor, ejecución de comandos para inicio del escaneo desde el contenedor, proceso de escaneo, descarga de resultados, cierre de conexión SSH al nodo.	2 min 45 s
		Total: 7 min 29 s

Demo-Pipeline - Stage View



Figura 9. Evolución del tiempo de ejecución del pipeline.

4.4.2. Actividad 11.

Revisar las vulnerabilidades encontradas para validar el nivel de riesgo que estas representan: En la **Figura 10** y **Figura 11** se observa el resultado de los reportes automatizados obtenidos a partir de la ejecución del pipeline, los cuales contienen referencias a las vulnerabilidades y bugs encontrados en el código fuente y la aplicación desplegada.

OWASP ZAP Report

Site: <http://192.168.15.19:18443>

Generated on Wed, 4 Jan 2023 20:10:41

Summary of Alerts

Risk Level	Number of Alerts
High	0
Medium	2
Low	2
Informational	0
False Positives:	0

Figura 10. Elementos encontrados en el escaneo dinámico.

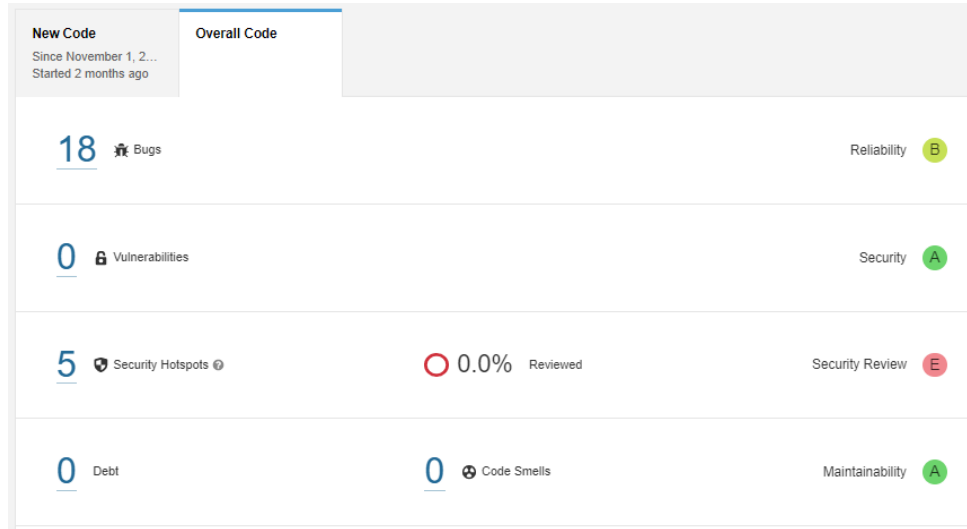


Figura 11. Elementos encontrados en el escaneo estático.

4.4.3. Actividad 12.

Identificar puntos de mejora y próximos pasos para la continuación del proyecto en futuras versiones: Se generaron espacios de discusión en los cuales se lograron identificar varios puntos de mejora a abordar en las próximas versiones del proyecto.

Tabla 3. Oportunidades de mejora identificadas.

Prioridad	Descripción
Alta	Realizar pruebas del pipeline actual con aplicaciones más robustas para evaluar el rendimiento en términos de tiempo de ejecución, vulnerabilidades encontradas, falsos positivos, etc.
Media	Integrar al pipeline una herramienta que pueda realizar escaneo de los contenedores utilizados en busca de posibles vulnerabilidades presentes en los mismos.
Media	Integrar al pipeline herramientas de monitoreo para obtener información asociada al consumo de recursos por parte de los componentes y las diferentes etapas del pipeline.
Media	Realizar pruebas de despliegue del pipeline sobre diferentes tipos de ambientes para validar si es necesario hacer configuraciones adicionales al cambiar el escenario de despliegue.
Baja	c

5. Resultados.

Una vez integrados los componentes descritos se consiguió construir un pipeline que permite llevar a cabo procesos de escaneos estáticos y dinámicos sobre una aplicación. La estructura final del pipeline se observa en la **Figura 12**. Este consiste básicamente en un repositorio de GitLab que aloja el código fuente y archivos necesarios para la aplicación web, un servidor de Jenkins que escucha los commits realizados sobre el repositorio para desplegar acciones, un contenedor de SonarQube que realiza una copia del código fuente para realizar un escaneo estático del mismo, instancias de docker y kubernetes que sirven para construir la imagen de la aplicación y desplegarla sobre el ambiente que simula producción y un contenedor de OWASP Zap que escanea de forma dinámica la aplicación en producción.

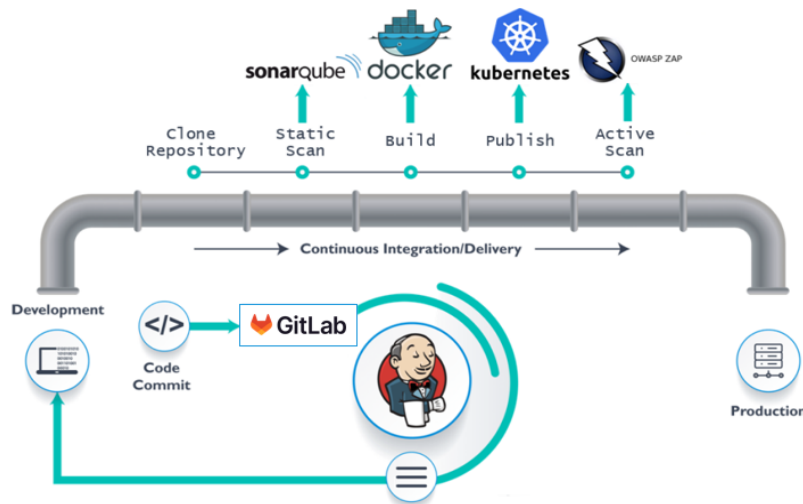


Figura 12. Estructura final del pipeline.

Con el objetivo de garantizar la funcionalidad del pipeline para diversos tipos de proyectos, se habilitaron en sonarQube grupos de reglas evidenciadas en la **Figura 13**, las cuales permiten al pipeline tener cobertura para diferentes lenguajes de programación.

Q Search for rules...		Kotlin 104	
v Language		Flex 82	
Q Search for languages...		HTML 65	
Java 627		Terraform 50	
C# 404		Ruby 48	
JavaScript 292		Scala 47	
TypeScript 267		Go 44	
PHP 257		XML 36	
Python 229		CloudFormation 27	
VB.NET 175		CSS 25	
		Kubernetes 13	

Figura 13. Lenguajes habilitados en SonarQube.

Una vez realizada la integración del pipeline, se realizó un proceso de robustecimiento mostrado en las figuras 14, 15, 16 y 17, el cual se aplicó sobre los diversos componentes utilizados en la integración para cumplir con un nivel de seguridad básico sobre la infraestructura desplegada.

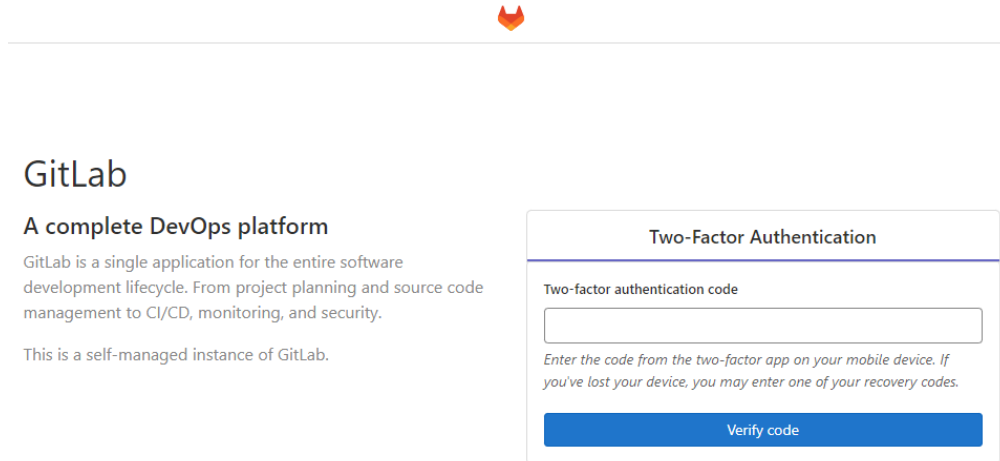


Figura 14. Habilitación de doble factor de autenticación.

Credentials

T	P	Store	Domain	ID	Name
🔑	👤	Jenkins	(global)	jenkins_kubectf_lab	Token de acceso kubectf lab01
👉	👤	Jenkins	(global)	jenkins_lab08	jenkins_lab08
👉	👤	Jenkins	(global)	jenkins_lab09	jenkins_lab09
🔑	👤	Jenkins	(global)	git-demo-pipeline-personal-access-token	personal acces token to the aligo git
🔑	👤	Jenkins	(global)	Jenkins-Sonar-Token	Jenkins-Sonar-Token

Figura 15. Almacenamiento de tokens de acceso entre Jenkins y los componentes del pipeline.

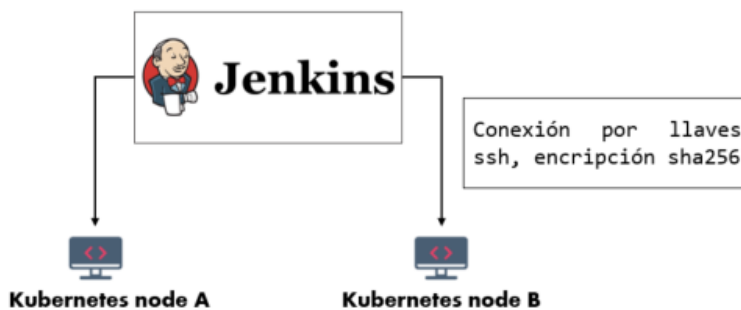


Figura 16. Conexión entre servidor de Jenkins y nodos de Kubernetes.

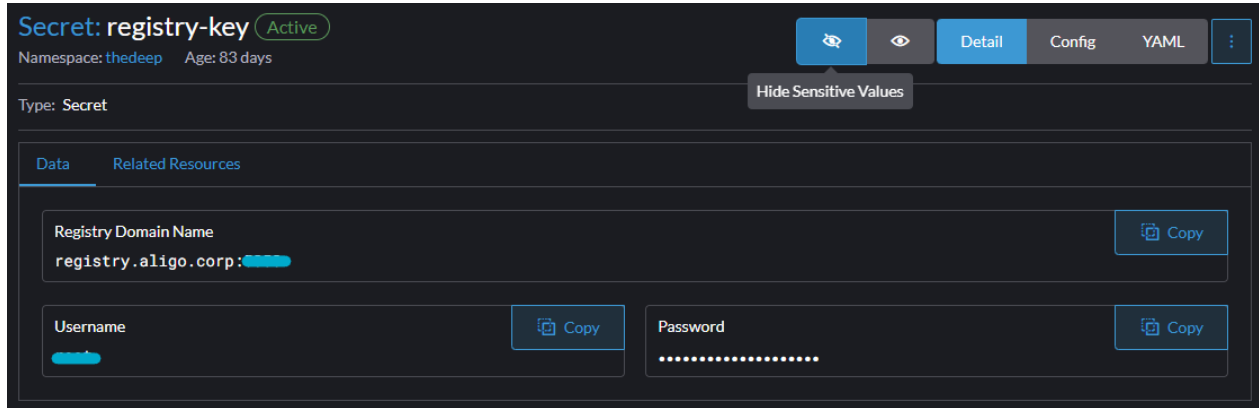


Figura 17. Configuración de secretos para acceso al registro de contenedores.

En las figuras 18 y 19, se presenta la evidencia asociada a la ejecución automática del pipeline construido ante la detección de un commit realizado por el usuario administrador sobre el repositorio de Gitlab monitoreado; lo anterior se realiza mediante el uso de un plugin de Jenkins que utiliza la API de Gitlab para establecer procesos de lectura constante sobre el repositorio de interés, permitiendo así detectar los cambios y generar la acción de lanzamiento del pipeline.

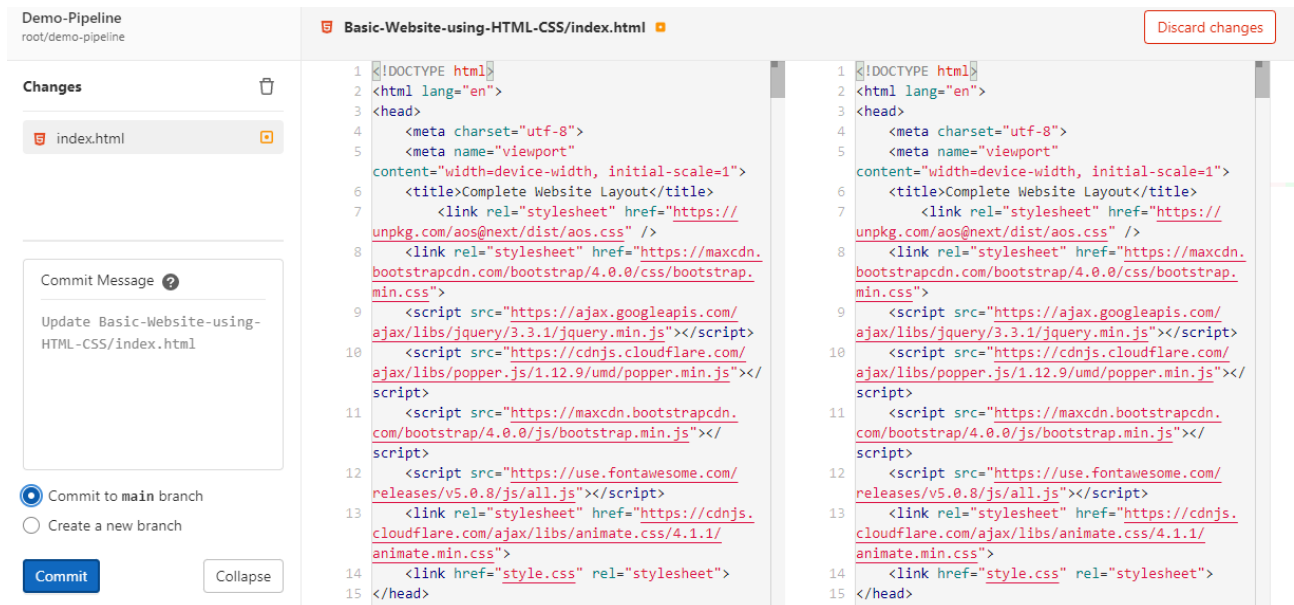


Figura 18. Generación de un commit sobre la rama principal del repositorio de Gitlab.

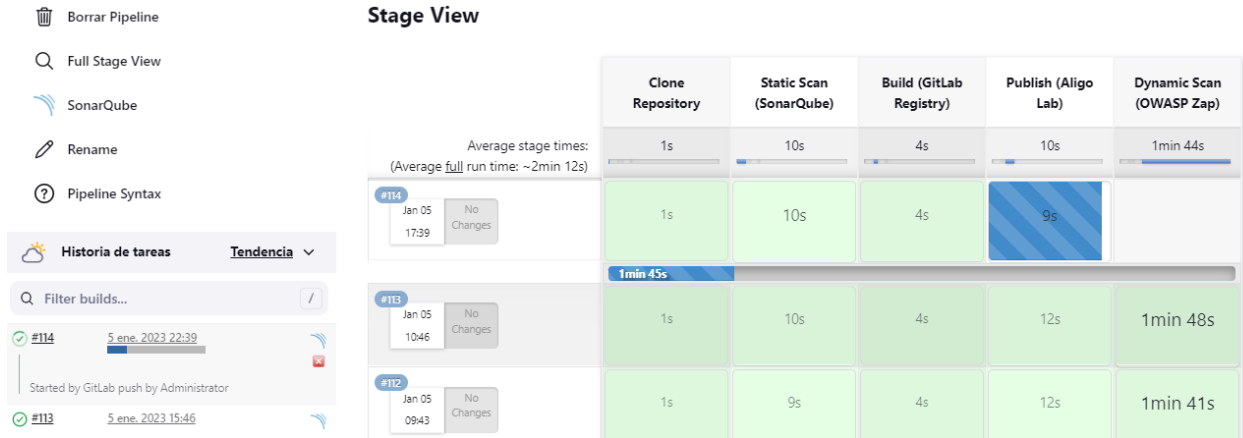


Figura 19. Inicialización del Pipeline a partir del push realizado en Gitlab.

Finalmente, en la **Figura 20** se presentan los logs correspondientes a la finalización exitosa del pipeline; mediante estos tenemos un mecanismo que nos permite realizar procesos de revisión y control sobre el estado de las acciones ejecutadas, así como también identificar posibles errores presentados durante la ejecución del pipeline, para así facilitar las fases de remediación que se requieran llevar a cabo.

```

JavaScript error: resource://gre/modules/AsyncShutdown.jsm, line 719: Error: Phase "quit-application-granted" is finished, it is too late to register completion condition "ContentParent: id=7fab51645500"
[===== ] 91% |
1672958476292 Marionette INFO Stopped listening on port 33283
Missing chrome or resource URL: resource://gre/modules/UpdateListener.sys.mjs
console.error: "Error during quit-application-granted: [Exception... \\"File error: Not found\\" nsresult: \\"0x80520012 (NS_ERROR_FILE_NOT_FOUND)\\" location: \\"JS frame :: resource:///modules/BrowserGlue.jsm :: _onQuitApplicationGranted/tasks< :: line 2009\\" data: no]"
[===== ] 91% /
[===== ] 96% -
[===== ] 97% \
[===== ] 99% |
[=====] 100%
Attack complete
Writing results to /home/zap/test-results-latest.html
[Pipeline] sh
+ docker cp 0c0d856d03ca:/home/zap/test-results-latest.html ../../
[Pipeline] }
[Pipeline] // node
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
    
```

Figura 20. Logs de la ejecución del pipeline.

6. Conclusiones.

A partir del proyecto realizado se consiguió automatizar el proceso de escaneo estático y dinámico de aplicaciones web mediante la implementación de un pipeline que ejecuta un conjunto de acciones al detectar cambios realizados sobre repositorio sobre el cual se aloja el proyecto configurado.

Se cumplió con el objetivo de realizar la implementación haciendo uso de tecnologías open source, lo cual se acopla con la filosofía empresarial y permite tener un mayor grado de flexibilidad a la hora realizar configuraciones o cambios sobre las herramientas.

Se logró realizar el despliegue de los componentes que conforman el pipeline sobre la infraestructura de laboratorio preexistente en Aligo defensores informáticos, lo cual implicó que no se generaron costos adicionales asociados a necesidades de recursos para complementar la infraestructura.

Se evidenció una mejora significativa en términos del tiempo de ejecución de los escaneos realizados manualmente, ejecutando cada una de las herramientas de forma individual y la ejecución automatizada del pipeline mediante las acciones de git detectadas sobre el repositorio.

Se generaron un conjunto de archivos manifiestos que contienen el detalle de las configuraciones propias de los componentes utilizados para el pipeline, lo cual permite desplegar o migrar dichos componentes a otros ambientes de manera ágil y sencilla.

Se logró construir un ambiente lo suficientemente flexible como para permitir realizar futuras modificaciones orientadas a mejoramiento o agregación de características al proyecto, o para desplegar nuevos pipelines que permitan automatizar procesos identificados como clave dentro del ambiente empresarial.

Referencias Bibliográficas

- [1] Mansfield-Devine, S. (2018). DevOps: finding room for security. *Network Security*, 2018(7), 15–20. doi:10.1016/s1353-4858(18)30070-9
- [2] Ebert, C., Gallardo, G., Hernantes, J., & Serrano, N. (2016). DevOps. *IEEE Software*, 33(3), 94–100. doi:10.1109/ms.2016.68
- [3] “¿Qué son los contenedores?”, Google Cloud. [Online]. Available at: <https://cloud.google.com/learn/what-are-containers> (Accessed Dec. 16, 2022).
- [4] S. Simic. “What is Docker?”, PhoenixNAP, (September 16, 2021). [Online]. Available at: <https://phoenixnap.com/kb/what-is-docker> (Accessed Dec. 17, 2022).
- [5] “Docker architecture”, Docker docs. [Online]. Available at: <https://docs.docker.com/get-started/overview/#docker-architecture> (Accessed Dec. 17, 2022).
- [6] “¿Qué es Kubernetes?”, Google Cloud. [Online]. Available at: <https://cloud.google.com/learn/what-is-kubernetes> (Accessed Dec. 20, 2022).
- [7] “Kubernetes Overview”, Kubernetes docs. [Online]. Available at: <https://kubernetes.io/docs/concepts/overview/> (Accessed Dec. 20, 2022).
- [8] “Kubernetes Components”, Kubernetes docs. [Online]. Available at: <https://kubernetes.io/docs/concepts/overview/components/> (Accessed Dec. 20, 2022).
- [9] “What is Rancher?”, Rancher Manager docs. [Online]. Available at: <https://ranchermanager.docs.rancher.com/> (Accessed Dec. 22, 2022).
- [10] K. Kelley. “What is GitLab and How to Use It?”, SimpliLearn, (December 21, 2022). [Online]. Available at: https://www.simplilearn.com/tutorials/git-tutorial/what-is-gitlab#what_is_gitlab (Accessed Dec. 27, 2022).
- [11] “Jenkins User Documentation”, Jenkins. [Online]. Available at: <https://www.jenkins.io/doc/> (Accessed Jan. 05, 2023).

[12] “SonarQube Documentation”, SonarQube docs 9.8. [Online]. Available at: <https://docs.sonarqube.org/latest/> (Accessed Jan. 05, 2023).

[13] “Introducing Zap”, Zap documentation. [Online]. Available at: <https://www.zaproxy.org/getting-started/#introducing-zap> (Accessed Jan. 07, 2023).

[14] M. Virmani, “Understanding DevOps & Bridging the Gap from Continuous Integration to Continuous Delivery,” Proc. 5th Int’l Conf. Innovative Computing Technology (INTECH 15), 2015, pp. 78–82.

[15] Mohan, V., & Othmane, L. B. (2016). SecDevOps: Is It a Marketing Buzzword? - Mapping Research on Security in DevOps. 2016 11th International Conference on Availability, Reliability and Security (ARES). doi:10.1109/ares.2016.92

[16] Virmani, M. (2015). Understanding DevOps & bridging the gap from continuous integration to continuous delivery. Fifth International Conference on the Innovative Computing Technology (INTECH 2015). doi:10.1109/intech.2015.7173368

[17] P. Bahrs. “Adopting the IBM DevOps approach for continuous software delivery”, IBM Developer, (2013). [Online]. Available at: <https://developer.ibm.com/articles/d-adoption-paths/> (Accessed Oct. 25, 2022).

[18] Berardi D, Giallorenzo S, Mauro J, Melis A, Montesi F, Prandini M. 2022. Microservice security: a systematic literature review. PeerJ Computer Science 8:e779 <https://doi.org/10.7717/peerj-cs.779>

[19] Constante, F.M.; Soares, R.; Pinto-Albuquerque, M.; Mendes, D.; Beckers, K. Integration of Security Standards in DevOps Pipelines: An Industry Case Study. In Product-Focused Software Process Improvement (PROFES 2020); Springer: Turin, Italy, 2020; pp. 434–451.

[20] Sharma, M. (2021). Review of the Benefits of DAST (Dynamic Application Security Testing) Versus SAST. Auricle Global Society of Education and Research, Bikaner, Rajasthan, India.

[21] A. Ozanich. “Understanding the DevOps Pipeline & How to Build One”, HubSpot blog, (2022). [Online]. Available at: <https://blog.hubspot.com/website/devops-pipeline> (Accessed Oct. 29, 2022).

[22] “What is Static Code Analysis?” Checkpoint. [Online]. Available at: <https://www.checkpoint.com/cyber-hub/cloud-security/what-is-static-code-analysis/> (Accessed Oct. 27, 2022).

[23] “What is Dynamic Application Security Testing (DAST)?” Microfocus. [Online]. Available at: <https://www.microfocus.com/en-us/what-is/dast> (Accessed Oct. 25, 2022).

Figuras

Figura 1. S. McGeown. “DevOps for Infrastructure: From VI Admin To DevOps Champion”, VMware blog, (2020). [Online]. Available at: <https://blogs.vmware.com/management/2020/03/vi-admin-to-devops.html> (accessed Oct. 27, 2022).

Figura 2. “Docker architecture”, Docker docs, (2022). [Online]. Available at: <https://docs.docker.com/engine/images/architecture.svg> (accessed Nov. 15, 2022).

Figura 3. “Kubernetes Components”, Kubernetes docs, (2022). [Online]. Available at: <https://d33wubrfki0168.cloudfront.net/2475489eaf20163ec0f54ddc1d92aa8d4c87c96b/e7c81/images/docs/components-of-kubernetes.svg> (accessed Nov. 18, 2022).