**DIS-017 Modelo de Operación de Verificación Post-Producción**

**Anexo Reporte de Evaluación de Solución Tecnológica – Transparencia y Acceso a la Información**

**14/dic/2018**

**Contenido**

[1. Objetivo 3](#_Toc528580693)

[2. Introducción 3](#_Toc528580694)

[3. Alcance 4](#_Toc528580695)

[4. Detalle del Entregable 5](#_Toc528580696)

[4.1. Contexto de la Solución Tecnológica 5](#_Toc528580697)

[4.2. Tipo de Solicitud 5](#_Toc528580698)

[4.3. Evaluación de Código Fuente 5](#_Toc528580699)

[4.4. Alineación Entre Diseño y Construcción de Soluciones Tecnológicas 11](#_Toc528580700)

[4.5. Consistencia entre Ambientes del Ciclo de Vida de Desarrollo 11](#_Toc528580701)

[4.6. Cumplimiento Documental para Liberación en Ambiente Productivo 12](#_Toc528580702)

[4.7. Resultado Integral 13](#_Toc528580703)

[4.8. Propuestas de Acciones de Mejora 18](#_Toc528580704)

[5. Conclusiones 19](#_Toc528580705)

1. Objetivo

El entregable tiene por objetivo especificar las brechas y riesgos identificados derivado de la evaluación realizada sobre la solución tecnológica en los siguientes ámbitos:

**Calidad de código. -** Registrar el resultado del análisis de calidad del código fuente de una solución tecnológica específica con base en diversos estándares aceptados en la industria. La evaluación permite identificar el uso de malas prácticas, brechas de seguridad, *Bugs* que pueden provocar incidencias en la operación, deuda técnica excesiva y oportunidades de mejora en la codificación.

Una vez obtenida la información es posible generar propuestas de mejora que permitan elevar la calidad del código, reduciendo la posibilidad de afectación a la operación del Instituto y facilitando su mantenimiento.

**Verificar alineación entre diseño y construcción de la solución. -** Verificar que exista alineación entre los elementos considerados en la etapa de diseño detallado de la solución tecnológica (Capas de desarrollo, Paquetes, Clases, Métodos, Distribución de Componentes, Tecnologías) y su implementación durante la etapa construcción.

**Validar consistencia entre ambientes del ciclo de vida de desarrollo. -** Validar la consistencia entre los ambientes de desarrollo, pruebas y producción en términos de tecnologías (versiones, *releases* y parches) utilizadas.

**Documentación para liberación en ambiente productivo. -** Verificar que el proyecto cumple con la documentación necesaria para realizar la liberación de la solución tecnológica en ambiente productivo.

1. Introducción

Actualmente existen diversas problemáticas que dificultan la mejora continua de las soluciones tecnológicas desarrolladas dentro del Instituto, las cuales se mencionan a continuación:

* No se cuenta con un repositorio central de código fuente.
* Dificultad en la gestión de cambios realizados al código fuente durante el ciclo de vida de desarrollo.
* Falta de definición de estándares y mejores prácticas que regulen la construcción de soluciones.
* No se realiza la validación de la documentación necesaria para liberar en producción.
* Existen inconsistencias entre el diseño y la construcción de soluciones tecnológicas.
* Se presentan inconsistencias entre los ambientes utilizados durante el ciclo de vida de desarrollo.
* No se evalúa la calidad del código fuente generada.
* El código puesto en producción no cumple con estándares.

El equipo de verificación post-producción es responsable de cubrir las áreas de oportunidad antes mencionadas y proveer al Instituto la siguiente propuesta de valor:



Ilustración Propuesta de Valor

1. Alcance

La evaluación de soluciones tecnológicas considera los siguientes aspectos:

**Calidad de Código**: La evaluación de calidad de código se realiza con apoyo de la herramienta *SonarQube* ya que permite ejecutar una inspección estática sobre la codificación de la solución con base en los siguientes estándares*: Common Weakness Enumeration (CWE), Escal Institute of Advanced Technologies (SANS), Open Web Application Security Project (OWASP), The Motor Industry Software Reliability Association (MISRA) y Code Signing Certifcates CERTM*. La inspección permite identificar diversos errores alojados en el código que aumentan el riesgo de presentar un error en la operación.

**Alineación Entre Diseño y Construcción de Soluciones Tecnológicas:** Verificar el nivel de alineación que existe entre el diseño de soluciones tecnológicas y su construcción. De esta forma se puede identificar discrepancias entre la propuesta inicial de diseño y las decisiones consideradas en la etapa de construcción.

**Consistencia entre Ambientes del Ciclo de Vida de Desarrollo:** Validar que existe consistencia en la configuración de los ambientes (Versionamiento, *releases* y parches) utilizados a lo largo del ciclo de vida de desarrollo (Desarrollo, pruebas y producción). La validación se enfoca en los productos de *Software*, versionamiento y parches instalados en cada ambiente.

**Cumplimiento Documental para Liberación en Ambiente Productivo:** Verificar que el proyecto cumple con la documentación necesaria para realizar la liberación de la solución tecnológica en ambiente productivo.

1. Detalle del Entregable
   1. Contexto de la Solución Tecnológica

El **Sistema de Transparencia y Acceso a la Información** tiene por objetivo hacer accesible, la información sobre asuntos que le competen al Instituto y está obligado a proporcionar.

* 1. Tipo de Solicitud

| **Tipo de Solicitud** | Descripción | **Tipo** |
| --- | --- | --- |
| Nuevo | Construcción de una solución tecnológica nueva. |  |
| Mantenimiento Correctivo | Modificación reactiva de una solución tecnológica derivada del descubrimiento de incidencias durante la operación.  Ej. Incidencias reportadas por usuarios. |  |
| Mantenimiento Adaptivo | Modificación realizada para adaptar la solución tecnológica a su entorno.  Ej. Derivado de la actualización de versión de base de datos. |  |
| Mantenimiento Perfectivo | Modificación realizada con el objetivo de agregar funcionalidad nueva, mejorar su desempeño y/o facilitar su mantenimiento.  Ej. Requerimientos que agreguen funcionalidad. |  |
| Mantenimiento Preventivo | Modificación realizada para evitar fallas en la operación.  Ej. Optimización de Código. |  |
| Identificación de Código Fuente | Identificación de código fuente dentro del Instituto. | X |

* **Descripción de la Solicitud**

Se identificó el código fuente del proyecto como parte del esfuerzo realizado por la gerencia de Ingeniería de TI enfocado en centralizar los códigos fuente del INFONAVIT por medio de su almacenamiento en el repositorio institucional.

* 1. Evaluación de Código Fuente

La inspección permite identificar diversas incidencias alojados en el código los cuales aumentan el riesgo de presentar una falla en la operación. La herramienta de evaluación de calidad de código *SonarQube* agrupa las incidencias identificadas por medio de categorías que a su vez se clasifican por su severidad.

* **Categoría**
  + ***Bugs***

Los *Bugs* son errores en el código fuente de un sistema o aplicación que deriva en un comportamiento incorrecto/inesperado generando un resultado indeseado. Es importante identificar y solucionar de manera temprana los *Bugs* de programación para asegurar la calidad en los productos desarrollados, así como su funcionamiento de acuerdo a lo esperado.

* + **Vulnerabilidades de Seguridad**

Las Vulnerabilidades de Seguridad son imperfecciones alojadas en el código fuente las cuales son susceptibles a ser aprovechadas por una atacante con el fin ejecutar código malicioso y/o acceder a información sensible manipulada por una solución tecnológica.

Por lo anterior, es importante identificar de manera temprana las vulnerabilidades de seguridad para así fortalecer el código fuente y, en consecuencia, generar productos de *software* seguros que disminuyan el riesgo de ataques por parte de terceros.

* + **Optimizadores**

Los optimizadores o “*Code* *Smells*” representan un indicador de un problema más profundo. Este tipo de observaciones **NO** representan un *Bug* o falla técnica dentro del código, señalan diversos factores que pueden causar deficiencias a futuro: Aumento de presencia de fallas, bajo desempeño e incluso aumento de deuda técnica.

Por lo anterior, es importante identificar de manera temprana los optimizadores necesarios para aplicar medidas correctivas, de esta forma aumentaría la calidad del producto de *software* y reducir el riesgo de presencia de fallas durante la operación de la solución tecnológica.

* **Severidades**

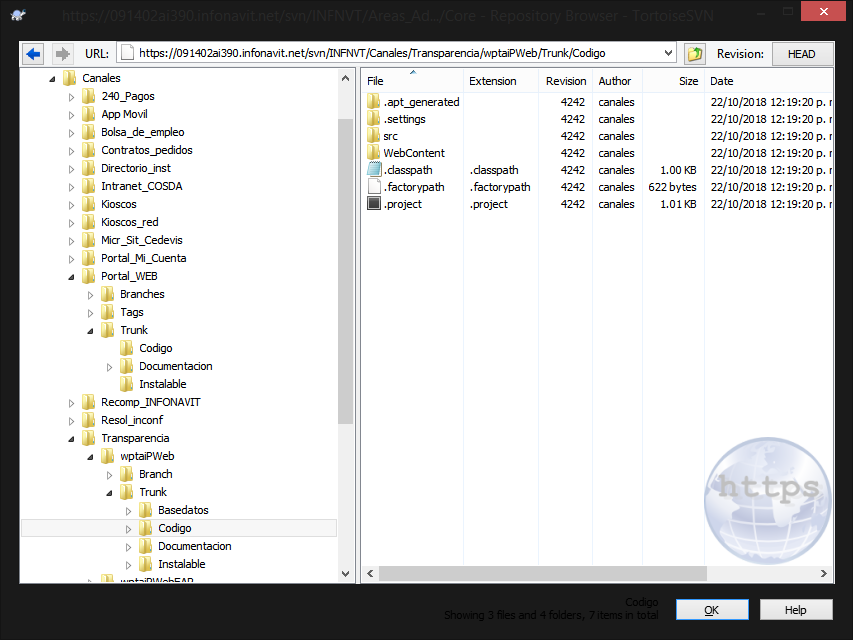
Los errores categorizados anteriormente son clasificados por severidades las cuales son definidas con base en 2 factores: **Impacto** y **Probabilidad**.

|  | **Impacto** | **Probabilidad** | **Descripción** |
| --- | --- | --- | --- |
| ***Blocker*** |  |  | * Alta probabilidad de ocurrencia * En caso de ocurrir puede provocar un error de gran escala que imposibilite la ejecución correcta de la operación del Instituto * Su corrección debe ser inmediata |
| ***Critical*** |  |  | * Baja probabilidad de ocurrencia * En caso de ocurrir puede provocar un error de gran escala que imposibilite la ejecución correcta de la operación del Instituto * Su corrección debe ser inmediata |
| ***Major*** |  |  | * Alta probabilidad de ocurrencia * En caso de ocurrir puede provocar un error de afectación menor |
| ***Minor*** |  |  | * Baja probabilidad de ocurrencia * En caso de ocurrir puede provocar un error de afectación menor |

* + 1. Ubicación en el Repositorio Institucional

Los artefactos evaluados (Código fuente y/o ejecutables) se encuentran en la siguiente ubicación dentro del repositorio:

<https://091402ai390.infonavit.net/svn/INFNVT/Canales/Transparencia>



* + 1. Estructura de Archivo *sonar.properties*

La inspección se realiza por medio de la definición del *archivo sonar.properties* el cual indica a la herramienta *Sonar Scanner* los parámetros necesarios para ejecutar el análisis de calidad de código.

* + 1. Ejecución de Análisis de Código Fuente

El análisis de la calidad del código fuente se realiza por medio de la ejecución del siguiente comando: ***sonar-scaner.bat***

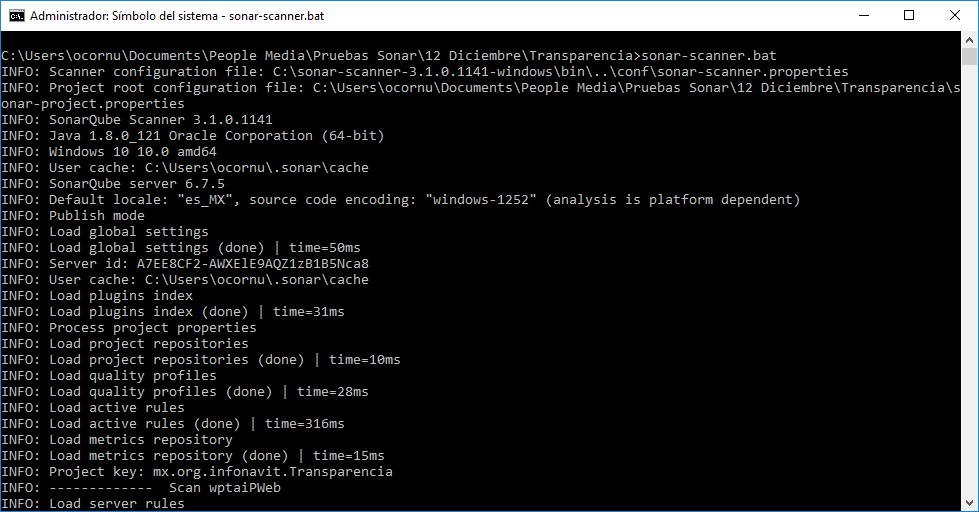


Ilustración Pantalla 1 de ejecución *sonar-scaner.bat*

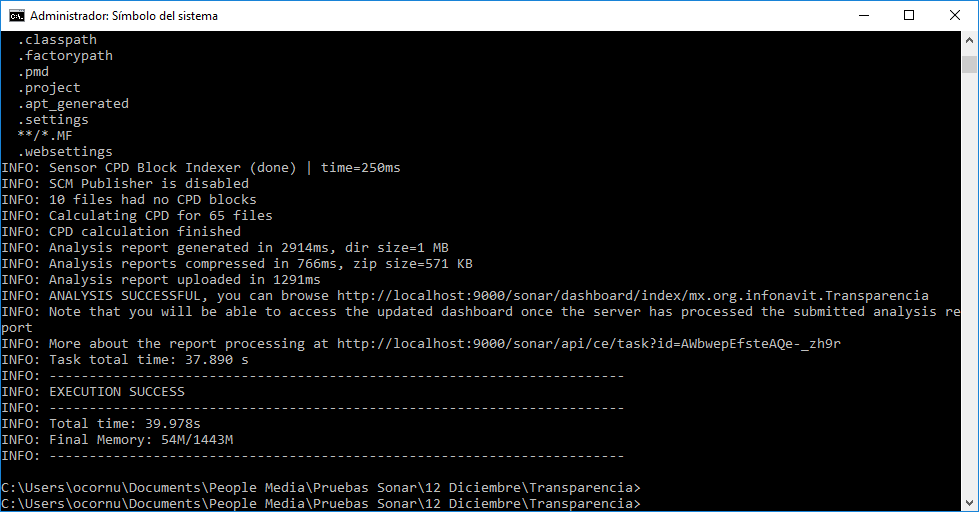


Ilustración Pantalla 2 de ejecución *sonar-scaner.bat*

* + 1. Resultado del Análisis de Código Fuente

A continuación, se muestran los resultados del análisis de calidad de código categorizado por el tipo de incidencias encontradas:

Proyecto: wptaiPWeb

Ilustración Gráfico de *Bug*s

Ilustración Gráfico de Vulnerabilidades

Ilustración Gráfico de Optimizadores

A continuación se enlistan las incidencias del tipo Vulnerabilidad, Bugs y Optimizadores las cuales fueron identificadas por medio del análisis de calidad de código y que deben ser atendidas a la brevedad posible con el objetivo de reducir el riesgo de fallo durante la operación.

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| *Bug* | *Blocker* | **Los recursos deben estar cerrados**  Las conexiones, secuencias, archivos y otras clases que implementan la interfaz *Closeable* o *AutoCloseable*, deben cerrarse después de su uso. Además, esa llamada cerrada debe hacerse en un bloque final, de lo contrario una excepción podría impedir que se haga la llamada. Preferiblemente, cuando la clase implementa “*AutoCloseable*”, el recurso debe ser creado usando el patrón "*try*-*with*-*resources*" y se cerrará automáticamente. Si no se cierran correctamente los recursos, se producirá una fuga de recursos. | 6 |
| *Bug* | *Major* | **Los *Null Pointers* deben estar referenciados**  Una referencia a un *null* nunca debe ser desreferenciada/accedida. En caso de realizarse, se lanzará una *NullPo*int*erException*. En el mejor de los casos, tal excepción causará la terminación abrupta del programa. En el peor de los casos, podría exponer información de depuración que sería útil para un atacante, o podría permitir a un atacante eludir las medidas de seguridad.  Se debe considerar que cuando están presentes, esta regla aprovecha las anotaciones @*CheckForNull* y @*Nonnull* definidas en *JSR*-*305* para entender qué valores son y cuáles no son nulos excepto cuando se utiliza @*Nonnull* en el parámetro a igual, que por contrato siempre debe trabajar con *null*. | 4 |
| *Bug* | *Major* | **Las sentencias de salto no deben ocurrir en bloques "*Finally*”**  Utilizar return, *break*, *throw*, etc. desde un bloque final se suprime la propagación de cualquier *throwable* no manejado que haya sido lanzado en el bloque de *try* o *catch*. | 1 |
| *Bug* | *Major* | **Todos los *branches* de una estructura condicional no deberían tener exactamente la misma implementación**  Tener todos los *branches* en un *switch* o en una cadena *if* con la misma implementación es un error. Esta regla no se aplica a las cadenas *if* sin hacer uso de *else* o en caso de utilizar *switch* sin cláusulas por defecto. | 1 |
| *Bug* | *Minor* | **No deben utilizarse sentencias “with”**  El uso de la palabra clave “with” produce un error en el código JavaScript.  Usar “with” permite un acceso rápido a las propiedades de un objeto, asumiendo que ya están configuradas. Pero úsalo para acceder a alguna propiedad que no esté ya establecida en el objeto. Dado que los efectos de “with” dependen completamente del objeto que se le ha pasado, “with” puede ser peligrosamente impredecible, y nunca debe ser utilizado. | 1 |
| Vulnerabilidad | *Critical* | **El código no debe ser inyectado y ejecutado dinámicamente**  La función de *eval* es una manera de ejecutar código arbitrario en tiempo de ejecución. La evaluación dinámica del código es lenta y un problema potencial de seguridad cuando los argumentos no han sido validados correctamente. | 4 |
| Vulnerabilidad | *Minor* | ***Throwable.printStackTrace(...)* no debe ser utilizado**  *Throwable.printStackTrace(...)* imprime un *Throwable* y su *stack trace* a algún *stream*. Por defecto, ese *streamSystem.Err* podría exponer inadvertidamente información sensible.  Se recomienda el uso de *Loggers* en lugar de imprimir *Throwables*, ya que tienen algunas ventajas:   * Los usuarios pueden recuperar fácilmente los *logs*. * El formato de los mensajes de log es uniforme y permite a los usuarios navegar fácilmente por ellos.   Esta regla genera un problema cuando *printStackTrace* se usa sin argumentos, es decir, cuando el *stack trace* se imprime en el *stream* predeterminado. | 14 |
| Vulnerabilidad | *Minor* | **No debe utilizarse "*alert(...)"***  *alert(....)* así como *confirm(....)* y *prompt(...)* puede de utilidad para depurar durante la etapa de desarrollo, pero en ambiente de producción este tipo de *pop-up* podría exponer información sensible a los atacantes, por lo que no debe mostrarse. | 11 |
| Vulnerabilidad | *Minor* | Las variables de una clase no deben tener accesibilidad pública  Las variables públicas en una clase no respetan el principio de encapsulamiento y tienen 3 desventajas principales:   * No se puede agregar Comportamiento adicional como la validación. * La representación interna está expuesta y no puede modificarse posteriormente. * Los valores de las variables están sujetos a cambios desde cualquier parte del código y pueden no cumplir con las suposiciones del programador.   Mediante el uso de atributos privados y métodos de acceso (*set* y *get*), se evitan las modificaciones no autorizadas. | 6 |
| Vulnerabilidad | *Minor* | **Los campos "*public static*" deben ser constantes**  No se recomienda declarar un campo "*public*" y "*static*" sin declararlo "*final*". Esto es un *kludge* para compartir un estado entre varios objetos, pero con este enfoque, cualquier objeto modificará el estado compartido, como cambiarlo a *null*. | 6 |
| Vulnerabilidad | *Minor* | **Las direcciones *IP* no deben estar en código duro**  Poner en duro dirección *IP* dentro del código fuente es una mala idea por varias razones:   * Se necesita compilar de nuevo si la dirección *IP* cambia. * Obliga a utilizar la misma dirección en todos los ambientes (Desarrollo, Pruebas y Producción). * Deja la responsabilidad de colocar el valor a utilizar en producción en manos del desarrollador. * Permite a los atacantes descompilar el código y descubrir direcciones potencialmente sensibles. | 4 |
| Vulnerabilidad | *Minor* | **Los campos mutables no deben ser "*public static*"**  No existen razones para tener un objeto mutable como “público (por defecto) *static*” en una interfaz. Estas variables deberían trasladarse a clases y su visibilidad debería reducirse.  Del mismo modo, los miembros estáticos mutables de clases y enumeraciones a los que se accede directamente, en lugar de a través de *getters* y *setters*, deben protegerse en la medida de lo posible. Esto puede hacerse reduciendo la visibilidad o haciendo que el campo sea *final*, si procede.  Tenga en cuenta que al crear un campo mutable, como un arreglo, al final evitará que la variable sea reasignada, pero al hacerlo no tiene ningún efecto sobre la mutabilidad del estado interno del arreglo (es decir, no logra la meta).  Esta regla genera problemas para los miembros de arreglos, *Collection, Date, y awt.Point* que sean *public static*. | 2 |
| Vulnerabilidad | *Minor* | **Los valores de retorno no deben ignorarse cuando contienen el código de estado de operación**  Cuando el valor de retorno de una función contiene el código de estado de operación, este valor debe comprobarse para asegurarse que la operación se completó correctamente.  Esta regla genera un problema cuando los valores de retorno de los siguientes métodos sin ignorados:   * *java.io.File, operaciones que retornan un código de estado (excepto mkdirs)* * *Iterator.hasNext()* * *Enumeration.hasMoreElements()* * *Lock.tryLock()* * *Métodos no-void de Condition.await\** * *CountDownLatch.await(long, TimeUnit)* * *Semaphore.tryAcquire* * *BlockingQueue:offer, remove, drainTo* | 1 |
| *Optimizador* | *Blocker* | **Las variables deberán declararse explícitamente**  Debe evitarse la creación de variables globales. Este escenario ocurre cuando se declara una variable dentro de una función o la cláusula *for* de un bucle *for-loop* sin keywords como *let, const* o *var*. | 23 |
| Optimizador | *Blocker* | **La lógica de cortocircuito debe usarse en contextos booleanos**  Debe utilizar la lógica de Cortocircuito en un contexto booleano ya que de lo contrarioes probable que se provoquen errores de programa a medida que las condiciones se evalúan en las circunstancias equivocadas. | 2 |
| Optimizador | *Critical* | **Las literales de los *strings* no deben duplicarse**  Las cadenas de texto duplicadas literalmente hacen que el proceso de refactorización sea propenso a errores, ya que debe asegurarse de actualizar todas las ocurrencias. Por otra parte, las constantes pueden ser referenciadas desde muchos lugares, pero sólo necesitan ser actualizadas en un solo lugar. El umbral por defecto es de 3. | 80 |
| *Optimizador* | *Critical* | **Los nombres de las constantes deben cumplir con una convención de nombres**  Las convenciones de codificación compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres constantes coincidan con una expresión regular proporcionada.  Utilizar la siguiente expresión regular por defecto: ^[A-Z][A-Z0-9]\*(\_[A-Z0-9]+)\*$: | 20 |
| *Optimizador* | *Critical* | **La complejidad cognitiva de los métodos no debe ser demasiado alta**  La Complejidad Cognitiva es una medida que indica la dificultad de interpretar el flujo de control de un método. Los métodos con alta Complejidad Cognitiva son difíciles de mantener. | 7 |
| *Optimizador* | *Critical* | **Los métodos no deben estar vacíos**  Hay varias razones para que un método no tenga cuerpo de método:   * Es una omisión involuntaria, y debe ser corregida para evitar un comportamiento inesperado en la producción. * Todavía no se apoya, o nunca se apoyará. En este caso, se debe lanzar una excepción de operación no soportada. * El método es una anulación intencionalmente en blanco. En este caso, un comentario anidado debe explicar el motivo de la anulación en blanco. | 4 |
| *Optimizador* | *Critical* | **Las declaraciones “*switch*" deben terminar con cláusulas “*default*”**  El requisito para una cláusula final “*defect*” es un ejemplo de programación defensiva. La cláusula debería adoptar las medidas apropiadas o contener un comentario adecuado sobre la razón por la que no se ha adoptado alguna medida. | 2 |
| *Optimizador* | *Critical* | **Se debe utilizar *try-with-resources***  *Java 7* introdujo la declaración *try-with-resources*, que garantiza el cierre del recurso en cuestión. Esta regla verifica que los recursos que puedan cerrarse se abren en una declaración de prueba con recursos. | 1 |
| *Optimizador* | *Major* | **Las reservas muertas deben ser removidas**  Un almacén muerto ocurre cuando a una variable local se le asigna un valor que no es leído por ninguna instrucción subsiguiente. Calcular o recuperar un valor sólo para luego sobrescribirlo o tirarlo a la basura, podría indicar un grave error en el código. Aunque no sea un error, en el mejor de los casos es un desperdicio de recursos. Por lo tanto, se deben utilizar todos los valores calculados. | 51 |
| Optimizador | *Major* | **Las salidas estándar no deben utilizarse directamente para generar *logs***  Al grabar en *log* un mensaje hay varias necesidades importantes que deben cumplirse:   * El usuario debe ser capaz de recuperar fácilmente los registros. * El formato de todos los mensajes registrados debe ser uniforme para permitir al usuario leer fácilmente el registro. * Los datos deben ser registrados. * Los datos confidenciales sólo deben registrarse de forma segura.   Si un programa escribe directamente en las salidas estándar, no hay forma de cumplir con estos requisitos. Es por eso que es altamente recomendable definir y usar un registrador dedicado. | 36 |
| Optimizador | *Major* | **Las asignaciones no deben hacerse desde dentro de subexpresiones**  Las asignaciones dentro de las subexpresiones son difíciles de detectar y por lo tanto hacen que el código sea menos legible. Idealmente, las subexpresiones no deberían tener efectos secundarios. | 32 |
| Optimizador | *Major* | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 24 |
| Optimizador | *Major* | **Las clases sincronizadas *Vector*, *Hashtable*, *Stack* y *StringBuffer* no se deben utilizar**  Las primeras clases de la *API* de *Java,* como *Vector*, *Hashtable* y *StringBuffer*, se sincronizaron para hacerlas seguras para los hilos. La sincronización tiene un gran impacto negativo en el rendimiento, incluso cuando se utilizan estas colecciones desde un solo hilo. Es mejor usar sus nuevos reemplazos no sincronizados:   * *ArrayList* o *LinkedList* en lugar de *Vector.* * *Deque* en lugar de *Stack.* * *HashMap* en lugar de *Hashtable.* * *StringBuilder* en lugar de *StringBuffer.* | 24 |
| Optimizador | *Major* | **Bloques de código no deben ser "comentados"**  Los programadores no deben comentar el código, ya que bloquea los programas y reduce la legibilidad. El código no utilizado debe ser borrado y puede ser recuperado del historial de control de fuente si es necesario. | 23 |
| *Optimizador* | *Major* | **Los constructores no deben ser utilizados para instanciar las clases "*String*" y *primitive*-*wrapper***  Los constructores de *Strings* y los objetos utilizados para envolver primitivos nunca deben utilizarse. Hacerlo es menos claro y usa más memoria que simplemente usar el valor deseado en el caso de los *strings*, y usar *valueOf* para todo lo demás. | 9 |
| Optimizador | *Major* | **Los bloques anidados de código no deben dejarse vacíos**  Los bloques de código vacíos tanto deben llenarse o retirarse. | 6 |
| Optimizador | *Major* | **Los métodos no deben tener demasiados parámetros**  Una larga lista de parámetros puede indicar que se debe crear una nueva estructura para completar los numerosos parámetros o que la función está haciendo demasiadas cosas. Se deben utilizar como máximo de 4 parámetros. | 4 |
| Optimizador | *Major* | **Las excepciones genéricas nunca deben ser lanzadas**  El uso de excepciones genéricas como *Error, RuntimeException, Throwable* y *Exception* evita que los métodos de llamada manejen las excepciones reales generadas por el sistema de manera diferente a los errores generados por la aplicación. | 4 |
| Optimizador | *Major* | **Las clases *Utility* no deben tener constructores públicos**  Las clases *Utility*, son colecciones de miembros estáticos que no se consideran para ser instanciadas. Incluso las clases de *Utility* abstractas que pueden ampliarse, no deberían tener constructores públicos.  Java añade un constructor público implícito. Por lo tanto, debe definirse al menos un constructor no público. | 3 |
| Optimizador | *Major* | **Los *arrays* vacíos y las colecciones deben ser devueltos en lugar de nulos**  Devolver *null* en lugar de un *array* real o coleccionar fuerza a los que llaman del método a probar explícitamente la nulidad, haciéndolos más complejos y menos legibles. Adicional, en muchos casos, *null* se utiliza como sinónimo de vacío. | 3 |
| *Optimizador* | *Major* | **Las reservas muertas deben ser removidas**  Un almacén muerto ocurre cuando a una variable local se le asigna un valor que no es leído por ninguna instrucción subsiguiente. Calcular o recuperar un valor sólo para luego sobrescribirlo o tirarlo a la basura, podría indicar un grave error en el código. Aunque no sea un error, en el mejor de los casos es un desperdicio de recursos. Por lo tanto, se deben utilizar todos los valores calculados. | 2 |
| Optimizador | *Major* | Las variables y funciones no deben ser declaradas de nuevo  Esta regla verifica que una declaración no utilice un nombre que ya está en uso. De hecho, es posible utilizar el mismo símbolo varias veces como variable o como función, pero hacerlo puede confundir a los mantenedores. Además, es posible que tales reasignaciones se hagan por error, sin que el desarrollador se dé cuenta de que el valor de la variable es sobrescrito por la nueva asignación.  Esta regla también se aplica a los parámetros de función. | 2 |
| Optimizador | *Major* | **Campos "*private*" que no son utilizados deben removerse**  Si un campo privado es declarado, pero no utilizado en el programa, puede ser considerado código muerto y por lo tanto debe ser eliminado.  Obsérvese que esta regla no tiene en cuenta la reflexión, lo que significa que se plantearán cuestiones sobre los campos privados a los que sólo se accede mediante la *API* de reflexión. | 1 |
| *Optimizador* | *Minor* | **Los nombres de parámetros de variables locales y métodos deben cumplir con una convención de nombres**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla plantea un problema cuando una variable local o un nombre de parámetro de función no coincide con la expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$:* | 100 |
| Optimizador | *Minor* | **Los nombres de los métodos deben cumplir con una convención de nomenclatura**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres de método coincidan con una expresión regular proporcionada.  Con la expresión regular proporcionada por defecto *^[a-z][a-zA-Z0-9]\*$:* | 27 |
| Optimizador | *Minor* | **Las variables locales no utilizadas deben eliminarse**  Si una variable local es declarada pero no utilizada, es código muerto y debe ser eliminada. Esto mejorará la mantenibilidad porque los desarrolladores no se preguntarán para qué sirve la variable. | 24 |
| Optimizador | *Minor* | **Los nombres de los campos deben cumplir con una convención de nombres**  Compartir algunas convenciones de nomenclatura es un punto clave para que un equipo pueda colaborar eficazmente. Esta regla permite comprobar que los nombres de campo coinciden con una expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$* | 15 |
| Optimizador | *Minor* | **Los nombres de campo estáticos no finales deben cumplir con una convención de nombres**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que los nombres de campo estáticos no finales coincidan con una expresión regular proporcionada. | 13 |
| Optimizador | *Minor* | **Los indicadores de arreglos "[ ]" deben estar en el tipo, no en la variable**  Los indicadores de arreglos deben estar siempre localizados en el tipo para una mejor legibilidad del código. De lo contrario, los desarrolladores deben mirar tanto el tipo como el nombre de la variable para saber si una variable es o no un *array*. | 7 |
| *Optimizador* | *Minor* | **Los objetos del tipo "Wrapper" no deben ser usados para tipos primitivos**  El uso de objetos de envoltura para tipos primitivos es gratuito, confuso y peligroso. Si se utiliza un constructor de objetos "Wrapper" para la conversión de tipos, simplemente elimine la nueva palabra clave y obtendrá un valor primitivo automáticamente. Si se usa un objeto "Wrapper" como una forma de añadir propiedades a un primitivo, se debe redefinir el diseño. Estos usos se consideran una mala práctica y deben ser reconsiderados. | 6 |
| Optimizador | *Minor* | **El Parseo debe usarse para convertir *Strings* en primitivas**  En lugar de crear un primitivo encajonado de una cadena para extraer el valor primitivo, utilice en su lugar el método de análisis relevante. | 6 |
| Optimizador | *Minor* | **Las variables locales no utilizadas deben eliminarse**  Si una variable local es declarada pero no utilizada, es código muerto y debe ser eliminada. Esto mejorará la mantenibilidad porque los desarrolladores no se preguntarán para qué sirve la variable. | 3 |
| Optimizador | *Minor* | Las declaraciones deben utilizar interfaces de *Java* *Collection* como "*List*" en lugar de clases de implementación específicas como "*LinkedList*"  El propósito de *Java* *Collections* *API* es proporcionar una jerarquía bien definida de interfaces para ocultar los detalles de implementación.  Las clases de implementación deben ser utilizadas para instanciar nuevas colecciones, pero el resultado de una instanciación debe ser almacenado idealmente en una variable cuyo tipo sea una interfaz de colección *Java*.  Esta regla plantea un problema cuando una clase de implementación:   * Es devuelto de un método público. * Se acepta como argumento para un método público. * Se expone como miembro público. | 3 |
| *Optimizador* | *Minor* | **Los nombres de clase deben cumplir con una convención de nombres**  Las convenciones de codificación compartida permiten a los equipos colaborar eficazmente. Esta regla permite comprobar que todos los nombres de clase coinciden con una expresión regular.  Con la expresión regular proporcionada por defecto *^[A-Z][a-zA-Z0-9]\*$:* | 2 |
| Optimizador | *Minor* | **Las comparaciones de cadenas insensibles a mayúsculas y minúsculas deben realizarse sin carcasa intermedia superior o inferior**  El uso de *toLowerCase*() o *toUpperCase* () para hacer comparaciones insensibles al caso es ineficiente porque requiere la creación de objetos *String* temporales e intermedios. | 2 |
| *Optimizador* | *Minor* | **Las declaraciones "*throws*" no deberían ser superfluas**  Una excepción en una declaración de lanzamientos en *Java* es superflua si:   * Aparece con frecuencia. * Una subclase de otra excepción de la lista. * Una *RuntimeException*, o uno de sus descendientes. * Completamente innecesario porque el tipo de excepción declarado no se puede lanzar realmente. | 1 |
| Optimizador | *Minor* | **Los nombres de los paquetes deben cumplir con una convención de nombres**  Las convenciones de codificación compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres de los paquetes coincidan con una expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z\_]+(\.[a-z\_][a-z0-9\_]\*)\*$* | 1 |
| Optimizador | *Minor* | Los literales booleanos no deberían ser redundantes  Los literales booleanos redundantes deben eliminarse de las expresiones para mejorar la legibilidad. | 1 |

El detalle completo del Análisis de Código Fuente puede ser consultado en la siguiente liga: <http://10.80.1.146:9000/sonarqube7/projects>

* 1. Alineación Entre Diseño y Construcción de Soluciones Tecnológicas

El nivel de alineación entre la definición de la solución tecnológica generada en la etapa de Diseño de Soluciones y los artefactos generados en la etapa de Construcción es determinado con base en la siguiente tabla de niveles de conformidad:

| **Representación Gráfica de Alineación** | **Descripción de los Niveles de Alineación** | **Valor** |
| --- | --- | --- |
|  | La implementación no tiene características en común con el diseño de la solución | 1 |
|  | La implementación tiene algunas características en común con el diseño de la solución y tales características están implementadas en total conformidad con el diseño. Sin embargo, existen características en la especificación que no fueron implementadas y existen características implementadas que no son cubiertas en la especificación | 2 |
|  | Existe total concordancia entre el diseño de la solución y su implementación. Todas las características están implementadas conforme a la especificación y no existen características implementadas que no sean cubiertas por la especificación | 3 |

**Nota:** No se evaluó la alineación entre Diseño y Construcción de Soluciones Tecnológicas ya que no se generó documentación para la atención de la solicitud.

| **Aspecto Evaluado** | **Diseño** | **Construcción** | **Nivel de Alineación** |
| --- | --- | --- | --- |
| NA | NA | NA | NA |
| **Nivel de Alineación Final** | | | NA |

* 1. Consistencia entre Ambientes del Ciclo de Vida de Desarrollo

El nivel de consistencia entre Ambientes del Ciclo de Vida del Desarrollo es determinado con base en la siguiente tabla de niveles de conformidad:

| **Representación Gráfica de Alineación** | **Descripción de los Niveles de Consistencia** | **Valor** |
| --- | --- | --- |
|  | La tecnología evaluada no es consistente en términos de producto de *Software* utilizado, versión y *release* entre los 3 ambientes considerados en el ciclo de vida del desarrollo | 1 |
|  | La tecnología evaluada es consistente en términos de producto de *Software* utilizado, versión y *release* en 2 de los 3 ambientes considerados en el ciclo de vida del desarrollo | 2 |
|  | La tecnología evaluada es consistente en términos de producto de *Software* utilizado, versión y *release* en los 3 ambientes considerados en el ciclo de vida del desarrollo | 3 |

**Nota:** No se evaluó la alineación entre Diseño y Construcción de Soluciones Tecnológicas ya que no fue posible obtener información detallada sobre los ambientes utilizados.

| **Tecnología** | **Ambiente de Desarrollo** | **Ambiente de Pruebas** | **Ambiente de Producción** | **Nivel de alineación de ambientes** |
| --- | --- | --- | --- | --- |
| NA | NA | NA | NA | NA |
| **Nivel de Alineación Final** | | | | NA |

* 1. Cumplimiento Documental para Liberación en Ambiente Productivo

**Nota:** No se evaluó el cumplimiento Documental para Liberación en Ambiente Productivo ya que no se generó documentación para la atención de la solicitud.

| **Nombre del Documento** | **Entregado** | **No Entregado** |
| --- | --- | --- |
| Plan de trabajo detallado de la solución | NA | NA |
| Diseño detallado de la solución | NA | NA |
| Plan de Implementación (Instalación, pruebas, retorno) | NA | NA |
| Componentes tecnológicos de la solución | NA | NA |
| Acuerdo de Niveles de Servicio | NA | NA |
| Solicitud de formalización del acuerdo de control de atributos de acceso | NA | NA |
| Acuerdo de control de atributos de acceso | NA | NA |
| Solicitud de custodia de cuenta privilegiada | NA | NA |
| Formato de liberación de aplicación | NA | NA |
| Formato de excepción a políticas de seguridad | NA | NA |
| Carta de Aceptación de Pruebas | NA | NA |
| Pruebas de Ancho de Banda | NA | NA |
| Formato de Alta o Modificación de Procesos | NA | NA |
| Atención a Usuario Final, Procesos de Gestión y CASIA | NA | NA |
| Documentos Varios (Memoria Técnica) | NA | NA |
| 01 (Servicio) - Información General | NA | NA |
| 02 (Servicio) - Procedimiento de Operación | NA | NA |
| 03 (Servicio) - Procedimiento de Recuperación | NA | NA |
| 04 (Servicio) - Monitoreo Disponibilidad | NA | NA |
| 05 (Servicio) - Cédula de Respaldos | NA | NA |
| Formatos para Control en CMDB | NA | NA |
| Formato RG1000 | NA | NA |
| Catálogo de Servicios | NA | NA |
| Capacitación a las áreas de Soporte Técnico a la Producción, al usuario final, y a Operaciones | NA | NA |

* 1. Resultado Integral

El resultado global es una vista de alto nivel sobre las observaciones que surgieron a partir de la evaluación de la solución tecnológica. Esta vista es generada por medio de los indicadores generados por cada uno de los rubros evaluados y un resultado integral considerado el porcentaje de valor de cada rubro.

* **Indicadores de Evaluación de Calidad de Código**
  + **Resultado General:** Este indicador muestra la cantidad total de líneas de código del proyecto y la cantidad de incidencias (*Bugs*, Vulnerabilidades y Optimizadores) encontradas por medio del análisis de calidad de código.

Ilustración Resultado General - Código

Ilustración Resultado General - Incidencias

* + **Comparación entre Versiones:** Este indicador muestra la comparativa entre en análisis de calidad de código aplicado a la **última versión desplegada del proyecto** (Versión operando en ambiente productivo y la **versión actualizada más reciente** (Versión con modificaciones recientes que aún no ha sido desplegada en ambiente productivo).

La evaluación de la solución tecnológica consideró únicamente la última versión desplegada en ambiente productivo ya que no se identifica una versión más reciente del proyecto.

Ilustración Gráfico Comparativo - Código

Ilustración Comparativo – Incidencias General

Ilustración Gráfico Comparativo de *Bug*s

Ilustración Gráfico Comparativo de Vulnerabilidades

Ilustración Gráfico Comparativo de Optimizadores

* **Indicadores de Alineación Entre Diseño y Construcción de Soluciones**
  + **Nivel de alineación por aspecto evaluado**: El nivel de alineación determina el grado de apego entre el diseño de soluciones tecnológicas y su construcción. Se utiliza la tabla anterior para definir el nivel de alineación de los aspectos evaluados.
  + **Nivel de alineación final**: Se genera al obtener el promedio de los niveles de alineación obtenidos.

**Nota:** No se especifican Indicadores de Alineación entre Diseño y Construcción de Soluciones ya que no se generó documentación para la atención de la solicitud.

| Aspecto Evaluado | Nivel de Alineación |
| --- | --- |
| Aspecto 1 | NA |
| Aspecto 2 | NA |
| Aspecto 3 | NA |
| Nivel de alineación final | NA |

* **Indicadores de Consistencia entre Ambientes**
  + **Nivel de alineación por aspecto evaluado**: Se obtiene con base en los criterios definidos en la tabla anterior.
  + **Nivel de alineación final**: Se genera al obtener el promedio de los niveles de alineación obtenidos.

**Nota:** No se especifican Indicadores de Consistencia entre Ambientes ya que no fue posible obtener información detallada sobre los ambientes utilizados.

| Tecnología Evaluada | Nivel de Alineación |
| --- | --- |
| Tecnología 1 | NA |
| Tecnología 2 | NA |
| Tecnología 3 | NA |
| Nivel de alineación final | NA |

* **Cumplimiento Documental**
  + **Cumplimiento Documental para Liberación en Ambiente Productivo**: Identificar el nivel de cumplimiento de entrega documental para realizar la liberación de la solución tecnológica en ambiente productivo

**Nota:** No se especifican Indicadores de Cumplimiento Documental para Liberación en Ambiente Productivo ya que no se generó documentación para la atención de la solicitud.

| **Indicador** | **Valor** |
| --- | --- |
| Cumplimiento documental para liberación en ambiente productivo | NA |

Proyecto: wptaiPWebEAR

Ilustración Gráfico de *Bug*s

Ilustración Gráfico de Vulnerabilidades

Ilustración Gráfico de Optimizadores

A continuación se enlistan las incidencias del tipo Vulnerabilidad, Bugs y Optimizadores las cuales fueron identificadas por medio del análisis de calidad de código y que deben ser atendidas a la brevedad posible con el objetivo de reducir el riesgo de fallo durante la operación.

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| *Bug* | *Blocker* | **Los recursos deben estar cerrados**  Las conexiones, secuencias, archivos y otras clases que implementan la interfaz *Closeable* o *AutoCloseable*, deben cerrarse después de su uso. Además, esa llamada cerrada debe hacerse en un bloque final, de lo contrario una excepción podría impedir que se haga la llamada. Preferiblemente, cuando la clase implementa “*AutoCloseable*”, el recurso debe ser creado usando el patrón "*try*-*with*-*resources*" y se cerrará automáticamente. Si no se cierran correctamente los recursos, se producirá una fuga de recursos. | 8 |
| *Bug* | *Major* | **Los *Null Pointers* deben estar referenciados**  Una referencia a un *null* nunca debe ser desreferenciada/accedida. En caso de realizarse, se lanzará una *NullPo*int*erException*. En el mejor de los casos, tal excepción causará la terminación abrupta del programa. En el peor de los casos, podría exponer información de depuración que sería útil para un atacante, o podría permitir a un atacante eludir las medidas de seguridad.  Se debe considerar que cuando están presentes, esta regla aprovecha las anotaciones @*CheckForNull* y @*Nonnull* definidas en *JSR*-*305* para entender qué valores son y cuáles no son nulos excepto cuando se utiliza @*Nonnull* en el parámetro a igual, que por contrato siempre debe trabajar con *null*. | 4 |
| *Bug* | *Major* | **Las sentencias de salto no deben ocurrir en bloques "*Finally*”**  Utilizar return, *break*, *throw*, etc. desde un bloque final se suprime la propagación de cualquier *throwable* no manejado que haya sido lanzado en el bloque de *try* o *catch*. | 1 |
| *Bug* | *Major* | **Todos los *branches* de una estructura condicional no deberían tener exactamente la misma implementación**  Tener todos los *branches* en un *switch* o en una cadena *if* con la misma implementación es un error. Esta regla no se aplica a las cadenas *if* sin hacer uso de *else* o en caso de utilizar *switch* sin cláusulas por defecto. | 1 |
| *Bug* | *Minor* | **El *boxing* y *unboxing* no debe revertirse inmediatamente**  *Boxing* es el proceso de poner un valor primitivo en un objeto análogo, como por ejemplo crear un entero para mantener un valor. El *Unboxing* es el proceso de recuperar el valor primitivo de tal objeto.    Dado que el valor original no se modifica durante el *boxing* y el *unboxing*, no tiene sentido hacerlo cuando no es necesario. Esto también se aplica al *autoboxing* y *auto*-*unboxing.* | 2 |
| *Bug* | *Minor* | **No deben utilizarse sentencias “with”**  El uso de la palabra clave “with” produce un error en el código JavaScript.  Usar “with” permite un acceso rápido a las propiedades de un objeto, asumiendo que ya están configuradas. Pero úsalo para acceder a alguna propiedad que no esté ya establecida en el objeto. Dado que los efectos de “with” dependen completamente del objeto que se le ha pasado, “with” puede ser peligrosamente impredecible, y nunca debe ser utilizado. | 1 |
| Vulnerabilidad | *Critical* | **El código no debe ser inyectado y ejecutado dinámicamente**  La función de *eval* es una manera de ejecutar código arbitrario en tiempo de ejecución. La evaluación dinámica del código es lenta y un problema potencial de seguridad cuando los argumentos no han sido validados correctamente. | 4 |
| Vulnerabilidad | *Minor* | Las variables de una clase no deben tener accesibilidad pública  Las variables públicas en una clase no respetan el principio de encapsulamiento y tienen 3 desventajas principales:   * No se puede agregar Comportamiento adicional como la validación. * La representación interna está expuesta y no puede modificarse posteriormente. * Los valores de las variables están sujetos a cambios desde cualquier parte del código y pueden no cumplir con las suposiciones del programador.   Mediante el uso de atributos privados y métodos de acceso (*set* y *get*), se evitan las modificaciones no autorizadas. | 15 |
| Vulnerabilidad | *Minor* | **Los campos "*public static*" deben ser constantes**  No se recomienda declarar un campo "*public*" y "*static*" sin declararlo "*final*". Esto es un *kludge* para compartir un estado entre varios objetos, pero con este enfoque, cualquier objeto modificará el estado compartido, como cambiarlo a *null*. | 15 |
| Vulnerabilidad | *Minor* | ***Throwable.printStackTrace(...)* no debe ser utilizado**  *Throwable.printStackTrace(...)* imprime un *Throwable* y su *stack trace* a algún *stream*. Por defecto, ese *streamSystem.Err* podría exponer inadvertidamente información sensible.  Se recomienda el uso de *Loggers* en lugar de imprimir *Throwables*, ya que tienen algunas ventajas:   * Los usuarios pueden recuperar fácilmente los *logs*. * El formato de los mensajes de log es uniforme y permite a los usuarios navegar fácilmente por ellos.   Esta regla genera un problema cuando *printStackTrace* se usa sin argumentos, es decir, cuando el *stack trace* se imprime en el *stream* predeterminado. | 14 |
| Vulnerabilidad | *Minor* | **No debe utilizarse "*alert(...)"***  *alert(....)* así como *confirm(....)* y *prompt(...)* puede de utilidad para depurar durante la etapa de desarrollo, pero en ambiente de producción este tipo de *pop-up* podría exponer información sensible a los atacantes, por lo que no debe mostrarse. | 11 |
| Vulnerabilidad | *Minor* | **Las direcciones *IP* no deben estar en código duro**  Poner en duro dirección *IP* dentro del código fuente es una mala idea por varias razones:   * Se necesita compilar de nuevo si la dirección *IP* cambia. * Obliga a utilizar la misma dirección en todos los ambientes (Desarrollo, Pruebas y Producción). * Deja la responsabilidad de colocar el valor a utilizar en producción en manos del desarrollador. * Permite a los atacantes descompilar el código y descubrir direcciones potencialmente sensibles. | 5 |
| Vulnerabilidad | *Minor* | **Los campos mutables no deben ser "*public static*"**  No existen razones para tener un objeto mutable como “público (por defecto) *static*” en una interfaz. Estas variables deberían trasladarse a clases y su visibilidad debería reducirse.  Del mismo modo, los miembros estáticos mutables de clases y enumeraciones a los que se accede directamente, en lugar de a través de *getters* y *setters*, deben protegerse en la medida de lo posible. Esto puede hacerse reduciendo la visibilidad o haciendo que el campo sea *final*, si procede.  Tenga en cuenta que al crear un campo mutable, como un arreglo, al final evitará que la variable sea reasignada, pero al hacerlo no tiene ningún efecto sobre la mutabilidad del estado interno del arreglo (es decir, no logra la meta).  Esta regla genera problemas para los miembros de arreglos, *Collection, Date, y awt.Point* que sean *public static*. | 3 |
| Vulnerabilidad | *Minor* | **Los valores de retorno no deben ignorarse cuando contienen el código de estado de operación**  Cuando el valor de retorno de una función contiene el código de estado de operación, este valor debe comprobarse para asegurarse que la operación se completó correctamente.  Esta regla genera un problema cuando los valores de retorno de los siguientes métodos sin ignorados:   * *java.io.File, operaciones que retornan un código de estado (excepto mkdirs)* * *Iterator.hasNext()* * *Enumeration.hasMoreElements()* * *Lock.tryLock()* * *Métodos no-void de Condition.await\** * *CountDownLatch.await(long, TimeUnit)* * *Semaphore.tryAcquire* * *BlockingQueue:offer, remove, drainTo* | 1 |
| *Optimizador* | *Blocker* | **Las Variables deben ser declaradas de forma explicita**  La creación de variables globales se da cuando se declara una variable dentro de una función o la cláusula “*for*” en de un bucle *for-loop* utilizando palabras clave como *let*, *const* o *var.* | 23 |
| Optimizador | *Blocker* | **La lógica de cortocircuito debe usarse en contextos booleanos**  Debe utilizar la lógica de Cortocircuito en un contexto booleano ya que de lo contrarioes probable que se provoquen errores de programa a medida que las condiciones se evalúan en las circunstancias equivocadas. | 2 |
| Optimizador | *Critical* | **Las literales de los *strings* no deben duplicarse**  Las cadenas de texto duplicadas literalmente hacen que el proceso de refactorización sea propenso a errores, ya que debe asegurarse de actualizar todas las ocurrencias. Por otra parte, las constantes pueden ser referenciadas desde muchos lugares, pero sólo necesitan ser actualizadas en un solo lugar. El umbral por defecto es de 3. | 80 |
| *Optimizador* | *Critical* | **Los nombres de las constantes deben cumplir con una convención de nombres**  Las convenciones de codificación compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres constantes coincidan con una expresión regular proporcionada.  Utilizar la siguiente expresión regular por defecto: ^[A-Z][A-Z0-9]\*(\_[A-Z0-9]+)\*$: | 20 |
| *Optimizador* | *Critical* | **La complejidad cognitiva de los métodos no debe ser demasiado alta**  La Complejidad Cognitiva es una medida que indica la dificultad de interpretar el flujo de control de un método. Los métodos con alta Complejidad Cognitiva son difíciles de mantener. | 7 |
| *Optimizador* | *Critical* | **Los métodos no deben estar vacíos**  Hay varias razones para que un método no tenga cuerpo de método:   * Es una omisión involuntaria, y debe ser corregida para evitar un comportamiento inesperado en la producción. * Todavía no se apoya, o nunca se apoyará. En este caso, se debe lanzar una excepción de operación no soportada. * El método es una anulación intencionalmente en blanco. En este caso, un comentario anidado debe explicar el motivo de la anulación en blanco. | 4 |
| *Optimizador* | *Critical* | **Las declaraciones “*switch*" deben terminar con cláusulas “*default*”**  El requisito para una cláusula final “*defect*” es un ejemplo de programación defensiva. La cláusula debería adoptar las medidas apropiadas o contener un comentario adecuado sobre la razón por la que no se ha adoptado alguna medida. | 3 |
| *Optimizador* | *Critical* | **Se debe utilizar *try-with-resources***  *Java 7* introdujo la declaración *try-with-resources*, que garantiza el cierre del recurso en cuestión. Esta regla verifica que los recursos que puedan cerrarse se abren en una declaración de prueba con recursos. | 1 |
| *Optimizador* | *Major* | **Las reservas muertas deben ser removidas**  Un almacén muerto ocurre cuando a una variable local se le asigna un valor que no es leído por ninguna instrucción subsiguiente. Calcular o recuperar un valor sólo para luego sobrescribirlo o tirarlo a la basura, podría indicar un grave error en el código. Aunque no sea un error, en el mejor de los casos es un desperdicio de recursos. Por lo tanto, se deben utilizar todos los valores calculados. | 51 |
| Optimizador | *Major* | **Las salidas estándar no deben utilizarse directamente para generar *logs***  Al grabar en *log* un mensaje hay varias necesidades importantes que deben cumplirse:   * El usuario debe ser capaz de recuperar fácilmente los registros. * El formato de todos los mensajes registrados debe ser uniforme para permitir al usuario leer fácilmente el registro. * Los datos deben ser registrados. * Los datos confidenciales sólo deben registrarse de forma segura.   Si un programa escribe directamente en las salidas estándar, no hay forma de cumplir con estos requisitos. Es por eso que es altamente recomendable definir y usar un registrador dedicado. | 38 |
| Optimizador | *Major* | **Las asignaciones no deben hacerse desde dentro de subexpresiones**  Las asignaciones dentro de las subexpresiones son difíciles de detectar y por lo tanto hacen que el código sea menos legible. Idealmente, las subexpresiones no deberían tener efectos secundarios. | 32 |
| Optimizador | *Major* | **Bloques de código no deben ser "comentados"**  Los programadores no deben comentar el código, ya que bloquea los programas y reduce la legibilidad. El código no utilizado debe ser borrado y puede ser recuperado del historial de control de fuente si es necesario. | 28 |
| Optimizador | *Major* | **Las clases sincronizadas *Vector*, *Hashtable*, *Stack* y *StringBuffer* no se deben utilizar**  Las primeras clases de la *API* de *Java,* como *Vector*, *Hashtable* y *StringBuffer*, se sincronizaron para hacerlas seguras para los hilos. La sincronización tiene un gran impacto negativo en el rendimiento, incluso cuando se utilizan estas colecciones desde un solo hilo. Es mejor usar sus nuevos reemplazos no sincronizados:   * *ArrayList* o *LinkedList* en lugar de *Vector.* * *Deque* en lugar de *Stack.* * *HashMap* en lugar de *Hashtable.* * *StringBuilder* en lugar de *StringBuffer.* | 25 |
| Optimizador | *Major* | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 24 |
| *Optimizador* | *Major* | **Los constructores no deben ser utilizados para instanciar las clases "*String*" y *primitive*-*wrapper***  Los constructores de *Strings* y los objetos utilizados para envolver primitivos nunca deben utilizarse. Hacerlo es menos claro y usa más memoria que simplemente usar el valor deseado en el caso de los *strings*, y usar *valueOf* para todo lo demás. | 9 |
| Optimizador | *Major* | **Los bloques anidados de código no deben dejarse vacíos**  Los bloques de código vacíos tanto deben llenarse o retirarse. | 6 |
| Optimizador | *Major* | **Las excepciones genéricas nunca deben ser lanzadas**  El uso de excepciones genéricas como *Error, RuntimeException, Throwable* y *Exception* evita que los métodos de llamada manejen las excepciones reales generadas por el sistema de manera diferente a los errores generados por la aplicación. | 6 |
| Optimizador | *Major* | **Las clases *Utility* no deben tener constructores públicos**  Las clases *Utility*, son colecciones de miembros estáticos que no se consideran para ser instanciadas. Incluso las clases de *Utility* abstractas que pueden ampliarse, no deberían tener constructores públicos.  Java añade un constructor público implícito. Por lo tanto, debe definirse al menos un constructor no público. | 5 |
| Optimizador | *Major* | **Los métodos no deben tener demasiados parámetros**  Una larga lista de parámetros puede indicar que se debe crear una nueva estructura para completar los numerosos parámetros o que la función está haciendo demasiadas cosas. Se deben utilizar como máximo de 4 parámetros. | 4 |
| Optimizador | *Major* | **Los *arrays* vacíos y las colecciones deben ser devueltos en lugar de nulos**  Devolver *null* en lugar de un *array* real o coleccionar fuerza a los que llaman del método a probar explícitamente la nulidad, haciéndolos más complejos y menos legibles. Adicional, en muchos casos, *null* se utiliza como sinónimo de vacío. | 3 |
| *Optimizador* | *Major* | **Las reservas muertas deben ser removidas**  Un almacén muerto ocurre cuando a una variable local se le asigna un valor que no es leído por ninguna instrucción subsiguiente. Calcular o recuperar un valor sólo para luego sobrescribirlo o tirarlo a la basura, podría indicar un grave error en el código. Aunque no sea un error, en el mejor de los casos es un desperdicio de recursos. Por lo tanto, se deben utilizar todos los valores calculados. | 2 |
| Optimizador | *Major* | **Las variables y funciones no deben ser declaradas de nuevo**  Esta regla verifica que una declaración no utilice un nombre que ya está en uso. De hecho, es posible utilizar el mismo símbolo varias veces como variable o como función, pero hacerlo puede confundir a los mantenedores. Además, es posible que tales reasignaciones se hagan por error, sin que el desarrollador se dé cuenta de que el valor de la variable es sobrescrito por la nueva asignación.  Esta regla también se aplica a los parámetros de función. | 2 |
| Optimizador | *Major* | **Campos "*private*" que no son utilizados deben removerse**  Si un campo privado es declarado, pero no utilizado en el programa, puede ser considerado código muerto y por lo tanto debe ser eliminado.  Obsérvese que esta regla no tiene en cuenta la reflexión, lo que significa que se plantearán cuestiones sobre los campos privados a los que sólo se accede mediante la *API* de reflexión. | 2 |
| *Optimizador* | *Minor* | **Los nombres de parámetros de variables locales y métodos deben cumplir con una convención de nombres**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla plantea un problema cuando una variable local o un nombre de parámetro de función no coincide con la expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$:* | 100 |
| Optimizador | *Minor* | **Los nombres de los métodos deben cumplir con una convención de nomenclatura**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres de método coincidan con una expresión regular proporcionada.  Con la expresión regular proporcionada por defecto *^[a-z][a-zA-Z0-9]\*$:* | 28 |
| Optimizador | *Minor* | **Las variables locales no utilizadas deben eliminarse**  Si una variable local es declarada pero no utilizada, es código muerto y debe ser eliminada. Esto mejorará la mantenibilidad porque los desarrolladores no se preguntarán para qué sirve la variable. | 24 |
| Optimizador | *Minor* | **Los nombres de campo estáticos no finales deben cumplir con una convención de nombres**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que los nombres de campo estáticos no finales coincidan con una expresión regular proporcionada. | 22 |
| Optimizador | *Minor* | **Los nombres de los campos deben cumplir con una convención de nombres**  Compartir algunas convenciones de nomenclatura es un punto clave para que un equipo pueda colaborar eficazmente. Esta regla permite comprobar que los nombres de campo coinciden con una expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$* | 15 |
| Optimizador | *Minor* | **El Parseo debe usarse para convertir *Strings* en primitivas**  En lugar de crear un primitivo encajonado de una cadena para extraer el valor primitivo, utilice en su lugar el método de análisis relevante. | 8 |
| Optimizador | *Minor* | **Los indicadores de arreglos "[ ]" deben estar en el tipo, no en la variable**  Los indicadores de arreglos deben estar siempre localizados en el tipo para una mejor legibilidad del código. De lo contrario, los desarrolladores deben mirar tanto el tipo como el nombre de la variable para saber si una variable es o no un *array*. | 7 |
| *Optimizador* | *Minor* | **Los objetos del tipo "Wrapper" no deben ser usados para tipos primitivos**  El uso de objetos de envoltura para tipos primitivos es gratuito, confuso y peligroso. Si se utiliza un constructor de objetos "Wrapper" para la conversión de tipos, simplemente elimine la nueva palabra clave y obtendrá un valor primitivo automáticamente. Si se usa un objeto "Wrapper" como una forma de añadir propiedades a un primitivo, se debe redefinir el diseño. Estos usos se consideran una mala práctica y deben ser reconsiderados. | 6 |
| Optimizador | *Minor* | Las declaraciones deben utilizar interfaces de *Java* *Collection* como "*List*" en lugar de clases de implementación específicas como "*LinkedList*"  El propósito de *Java* *Collections* *API* es proporcionar una jerarquía bien definida de interfaces para ocultar los detalles de implementación.  Las clases de implementación deben ser utilizadas para instanciar nuevas colecciones, pero el resultado de una instanciación debe ser almacenado idealmente en una variable cuyo tipo sea una interfaz de colección *Java*.  Esta regla plantea un problema cuando una clase de implementación:   * Es devuelto de un método público. * Se acepta como argumento para un método público. * Se expone como miembro público. | 4 |
| Optimizador | *Minor* | **Las variables locales y funciones locales no utilizadas deben ser eliminadas**  Si se declara una variable local o una función local pero no se utiliza, es considerado código muerto y debe eliminarse. Hacerlo mejorará la mantenibilidad del código. | 3 |
| *Optimizador* | *Minor* | **Los nombres de clase deben cumplir con una convención de nombres**  Las convenciones de codificación compartida permiten a los equipos colaborar eficazmente. Esta regla permite comprobar que todos los nombres de clase coinciden con una expresión regular.  Con la expresión regular proporcionada por defecto *^[A-Z][a-zA-Z0-9]\*$:* | 2 |
| Optimizador | *Minor* | **Las comparaciones de cadenas insensibles a mayúsculas y minúsculas deben realizarse sin carcasa intermedia superior o inferior**  El uso de *toLowerCase*() o *toUpperCase* () para hacer comparaciones insensibles al caso es ineficiente porque requiere la creación de objetos *String* temporales e intermedios. | 2 |
| *Optimizador* | *Minor* | **Los campos privados utilizados únicamente como variables locales en los métodos deben convertirse en variables locales**  Cuando el valor de un campo privado se asigna siempre en métodos de clase antes de ser leído, no se utiliza para almacenar información de clase. Por lo tanto, debería convertirse en una variable local en los métodos pertinentes para evitar confusiones. | 2 |
| Optimizador | *Minor* | **Los modificadores deben ser declarados en el orden correcto**  La especificación de lenguaje *Java* recomienda listar los modificadores en el siguiente orden:   1. *Annotations* 2. *Public* 3. *Protected* 4. *Private* 5. *Abstract* 6. *Static* 7. *Final* 8. *Transient* 9. *Volatile* 10. *Synchronized* 11. *Native* 12. *Strictfp*   No seguir esta convención no tiene ningún impacto técnico, pero reducirá la legibilidad del código porque la mayoría de los equipos de desarrollo están acostumbrados al orden estándar. | 1 |
| *Optimizador* | *Minor* | **Las declaraciones "*throws*" no deberían ser superfluas**  Una excepción en una declaración de lanzamientos en *Java* es superflua si:   * Aparece con frecuencia. * Una subclase de otra excepción de la lista. * Una *RuntimeException*, o uno de sus descendientes. * Completamente innecesario porque el tipo de excepción declarado no se puede lanzar realmente. | 1 |

El detalle completo del Análisis de Código Fuente puede ser consultado en la siguiente liga: <http://10.80.1.146:9000/sonarqube7/projects>

* 1. Alineación Entre Diseño y Construcción de Soluciones Tecnológicas

El nivel de alineación entre la definición de la solución tecnológica generada en la etapa de Diseño de Soluciones y los artefactos generados en la etapa de Construcción es determinado con base en la siguiente tabla de niveles de conformidad:

| **Representación Gráfica de Alineación** | **Descripción de los Niveles de Alineación** | **Valor** |
| --- | --- | --- |
|  | La implementación no tiene características en común con el diseño de la solución | 1 |
|  | La implementación tiene algunas características en común con el diseño de la solución y tales características están implementadas en total conformidad con el diseño. Sin embargo, existen características en la especificación que no fueron implementadas y existen características implementadas que no son cubiertas en la especificación | 2 |
|  | Existe total concordancia entre el diseño de la solución y su implementación. Todas las características están implementadas conforme a la especificación y no existen características implementadas que no sean cubiertas por la especificación | 3 |

**Nota:** No se evaluó la alineación entre Diseño y Construcción de Soluciones Tecnológicas ya que no se generó documentación para la atención de la solicitud.

| **Aspecto Evaluado** | **Diseño** | **Construcción** | **Nivel de Alineación** |
| --- | --- | --- | --- |
| NA | NA | NA | NA |
| **Nivel de Alineación Final** | | | NA |

* 1. Consistencia entre Ambientes del Ciclo de Vida de Desarrollo

El nivel de consistencia entre Ambientes del Ciclo de Vida del Desarrollo es determinado con base en la siguiente tabla de niveles de conformidad:

| **Representación Gráfica de Alineación** | **Descripción de los Niveles de Consistencia** | **Valor** |
| --- | --- | --- |
|  | La tecnología evaluada no es consistente en términos de producto de *Software* utilizado, versión y *release* entre los 3 ambientes considerados en el ciclo de vida del desarrollo | 1 |
|  | La tecnología evaluada es consistente en términos de producto de *Software* utilizado, versión y *release* en 2 de los 3 ambientes considerados en el ciclo de vida del desarrollo | 2 |
|  | La tecnología evaluada es consistente en términos de producto de *Software* utilizado, versión y *release* en los 3 ambientes considerados en el ciclo de vida del desarrollo | 3 |

**Nota:** No se evaluó la alineación entre Diseño y Construcción de Soluciones Tecnológicas ya que no fue posible obtener información detallada sobre los ambientes utilizados.

| **Tecnología** | **Ambiente de Desarrollo** | **Ambiente de Pruebas** | **Ambiente de Producción** | **Nivel de alineación de ambientes** |
| --- | --- | --- | --- | --- |
| NA | NA | NA | NA | NA |
| **Nivel de Alineación Final** | | | | NA |

* 1. Cumplimiento Documental para Liberación en Ambiente Productivo

**Nota:** No se evaluó el cumplimiento Documental para Liberación en Ambiente Productivo ya que no se generó documentación para la atención de la solicitud.

| **Nombre del Documento** | **Entregado** | **No Entregado** |
| --- | --- | --- |
| Plan de trabajo detallado de la solución | NA | NA |
| Diseño detallado de la solución | NA | NA |
| Plan de Implementación (Instalación, pruebas, retorno) | NA | NA |
| Componentes tecnológicos de la solución | NA | NA |
| Acuerdo de Niveles de Servicio | NA | NA |
| Solicitud de formalización del acuerdo de control de atributos de acceso | NA | NA |
| Acuerdo de control de atributos de acceso | NA | NA |
| Solicitud de custodia de cuenta privilegiada | NA | NA |
| Formato de liberación de aplicación | NA | NA |
| Formato de excepción a políticas de seguridad | NA | NA |
| Carta de Aceptación de Pruebas | NA | NA |
| Pruebas de Ancho de Banda | NA | NA |
| Formato de Alta o Modificación de Procesos | NA | NA |
| Atención a Usuario Final, Procesos de Gestión y CASIA | NA | NA |
| Documentos Varios (Memoria Técnica) | NA | NA |
| 01 (Servicio) - Información General | NA | NA |
| 02 (Servicio) - Procedimiento de Operación | NA | NA |
| 03 (Servicio) - Procedimiento de Recuperación | NA | NA |
| 04 (Servicio) - Monitoreo Disponibilidad | NA | NA |
| 05 (Servicio) - Cédula de Respaldos | NA | NA |
| Formatos para Control en CMDB | NA | NA |
| Formato RG1000 | NA | NA |
| Catálogo de Servicios | NA | NA |
| Capacitación a las áreas de Soporte Técnico a la Producción, al usuario final, y a Operaciones | NA | NA |

* 1. Resultado Integral

El resultado global es una vista de alto nivel sobre las observaciones que surgieron a partir de la evaluación de la solución tecnológica. Esta vista es generada por medio de los indicadores generados por cada uno de los rubros evaluados y un resultado integral considerado el porcentaje de valor de cada rubro.

* **Indicadores de Evaluación de Calidad de Código**
  + **Resultado General:** Este indicador muestra la cantidad total de líneas de código del proyecto y la cantidad de incidencias (*Bugs*, Vulnerabilidades y Optimizadores) encontradas por medio del análisis de calidad de código.

Ilustración Resultado General - Código

Ilustración Resultado General - Incidencias

* + **Comparación entre Versiones:** Este indicador muestra la comparativa entre en análisis de calidad de código aplicado a la **última versión desplegada del proyecto** (Versión operando en ambiente productivo y la **versión actualizada más reciente** (Versión con modificaciones recientes que aún no ha sido desplegada en ambiente productivo).

La evaluación de la solución tecnológica consideró únicamente la última versión desplegada en ambiente productivo ya que no se identifica una versión más reciente del proyecto.

Ilustración Gráfico Comparativo - Código

Ilustración Comparativo – Incidencias General

Ilustración Gráfico Comparativo de *Bug*s

Ilustración Gráfico Comparativo de Vulnerabilidades

Ilustración Gráfico Comparativo de Optimizadores

* **Indicadores de Alineación Entre Diseño y Construcción de Soluciones**
  + **Nivel de alineación por aspecto evaluado**: El nivel de alineación determina el grado de apego entre el diseño de soluciones tecnológicas y su construcción. Se utiliza la tabla anterior para definir el nivel de alineación de los aspectos evaluados.
  + **Nivel de alineación final**: Se genera al obtener el promedio de los niveles de alineación obtenidos.

**Nota:** No se especifican Indicadores de Alineación entre Diseño y Construcción de Soluciones ya que no se generó documentación para la atención de la solicitud.

| Aspecto Evaluado | Nivel de Alineación |
| --- | --- |
| Aspecto 1 | NA |
| Aspecto 2 | NA |
| Aspecto 3 | NA |
| Nivel de alineación final | NA |

* **Indicadores de Consistencia entre Ambientes**
  + **Nivel de alineación por aspecto evaluado**: Se obtiene con base en los criterios definidos en la tabla anterior.
  + **Nivel de alineación final**: Se genera al obtener el promedio de los niveles de alineación obtenidos.

**Nota:** No se especifican Indicadores de Consistencia entre Ambientes ya que no fue posible obtener información detallada sobre los ambientes utilizados.

| Tecnología Evaluada | Nivel de Alineación |
| --- | --- |
| Tecnología 1 | NA |
| Tecnología 2 | NA |
| Tecnología 3 | NA |
| Nivel de alineación final | NA |

* **Cumplimiento Documental**
  + **Cumplimiento Documental para Liberación en Ambiente Productivo**: Identificar el nivel de cumplimiento de entrega documental para realizar la liberación de la solución tecnológica en ambiente productivo

**Nota:** No se especifican Indicadores de Cumplimiento Documental para Liberación en Ambiente Productivo ya que no se generó documentación para la atención de la solicitud.

| **Indicador** | **Valor** |
| --- | --- |
| Cumplimiento documental para liberación en ambiente productivo | NA |

* 1. Propuestas de Acciones de Mejora
* **Calidad de Código**

Se sugiere atender de forma inmediata, las incidencias de tipo **Vulnerabilidad** y las de tipo ***Bug*** y **Optimizadores** con severidad ***Blocker***, ***Critical*** con el objetivo de reducir el riesgo de fallo durante la operación. Una vez atendidas las incidencias con severidad más alta, se sugiere establecer un plan de mantenimiento con el objetivo de corregir las incidencias con menor nivel de severidad.

Se recomienda ampliamente el uso de la extensión del IDE, “**Sonar Lint**”. El *plugin* antes mencionado permite identificar incidencias desde la construcción de la solución.

* **Alineación entre Diseño y Construcción de Soluciones Tecnológicas**

No se generan propuestas de acción de mejor debido a que no fue posible evaluar la Alineación entre Diseño y Construcción de Soluciones Tecnológicas.

* **Consistencia entre Ambientes del Ciclo de Vida de Desarrollo**

No se generan propuestas de acción de mejor debido a que no fue posible evaluar la Consistencia entre Ambientes del Ciclo de Vida de Desarrollo.

* **Cumplimiento Documental para Liberación en Ambiente Productivo**

No se generan propuestas de acción de mejor debido a que no fue posible evaluar el Cumplimiento Documental para Liberación en Ambiente Productivo.

1. Conclusiones

Preservar la calidad en el ciclo de vida de las soluciones tecnológicas es un factor de gran relevancia para el Instituto ya que reduce el riesgo de una afectación durante la operación del negocio.

Con base en el resultado de la evaluación de soluciones tecnológicas plasmada en este documento se logran identificar brechas de calidad que deben ser atendidas con el objetivo de incrementar la calidad de las soluciones tecnológicas generadas dentro del Instituto. Las brechas especificadas en este documento sirven de referencia para evitar errores similares en futuros proyectos.