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

**Anexo Reporte de Evaluación de Solución Tecnológica - Intranet**

**15/oct/2018**

**Contenido**

[1. Objetivo 3](#_Toc519840867)

[2. Introducción 3](#_Toc519840868)

[3. Alcance 4](#_Toc519840869)

[4. Detalle del Entregable 5](#_Toc519840870)

[4.1. Contexto de la Solución Tecnológica 5](#_Toc519840871)

[4.2. Tipo de Solicitud 5](#_Toc519840872)

[4.3. Evaluación de Código Fuente 5](#_Toc519840873)

[4.4. Alineación Entre Diseño y Construcción de Soluciones Tecnológicas 11](#_Toc519840874)

[4.5. Consistencia entre Ambientes del Ciclo de Vida de Desarrollo 11](#_Toc519840875)

[4.6. Cumplimiento Documental para Liberación en Ambiente Productivo 12](#_Toc519840876)

[4.7. Resultado Integral 13](#_Toc519840877)

[4.8. Propuestas de Acciones de Mejora 18](#_Toc519840878)

[5. Conclusiones 19](#_Toc519840879)

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 Intranet** tiene por objetivo ser una plataforma de comunicación interna del Instituto

* 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**

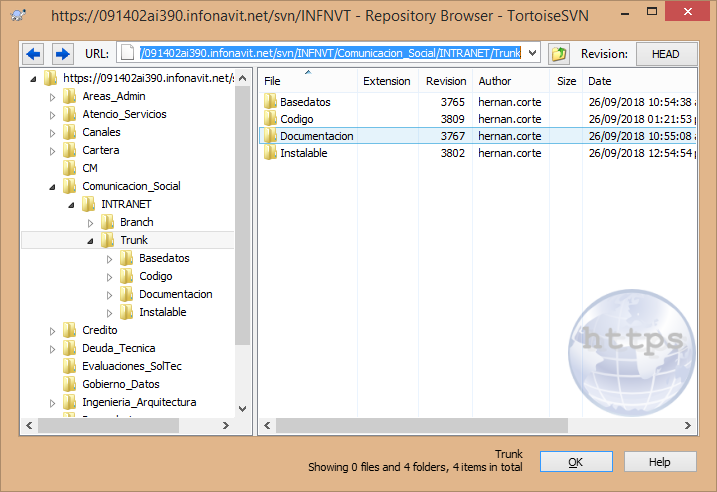
Los errores categorizados anteriormente son clasificados por severidades las cuales son definidas con base en 2 factores: **Impacto** y **Probabilidad**.

|  | **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/Comunicacion\_Social/INTRANET/Trunk*](https://091402ai390.infonavit.net/svn/INFNVT/Comunicacion_Social/INTRANET/Trunk)



* + 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.

#################################################

# Caracteres Unicode con acentos:

# ===============================

#

# a - \u00e1 A - \u00c1

# e - \u00e9 E - \u00c9

# i - \u00ed I - \u00cd

# o - \u00f3 O - \u00d3

# u - \u00fa U - \u00da

#

# Caracteres Unicode con diéresis

# ===============================

# u - \u00fc U - \u00dc

#

# Caracteres Unicode con tilde

# ===============================

#

# n - \u00f1 N - \u00d1

#

# Interrogación que abre - \u00bf

#

#

################################################

#Datos Generales del Proyecto

sonar.projectKey=mx.org.infonavit.intranet

sonar.projectName=Intranet

sonar.projectVersion=1.0

sonar.scm.disabled=True

#Módulos

sonar.modules=AdministradorIHS,AvisosSuspRetIntranet,ConstanciaInteresesIntranet,SaldoFondoViviendaIntranet,SaldosMovimientosIntranet

AdministradorIHS.sonar.projectName=AdministradorIHS

AvisosSuspRetIntranet.sonar.projectName=AvisosSuspRetIntranet

ConstanciaInteresesIntranet.sonar.projectName=ConstanciaInteresesIntranet

SaldoFondoViviendaIntranet.sonar.projectName=SaldoFondoViviendaIntranet

SaldosMovimientosIntranet.sonar.projectName=SaldosMovimientosIntranet

#Sources

AdministradorIHS.sonar.sources=.

AvisosSuspRetIntranet.sonar.sources=.

ConstanciaInteresesIntranet.sonar.sources=.

SaldoFondoViviendaIntranet.sonar.sources=.

SaldosMovimientosIntranet.sonar.sources=.

#Bin

AdministradorIHS.sonar.java.binaries=WebRoot/WEB-INF/classes

AvisosSuspRetIntranet.sonar.java.binaries=WebContent/WEB-INF

ConstanciaInteresesIntranet.sonar.java.binaries=WebContent/WEB-INF

SaldoFondoViviendaIntranet.sonar.java.binaries=WebContent/WEB-INF

SaldosMovimientosIntranet.sonar.java.binaries=WebContent/WEB-INF/classes

#Librerías

AdministradorIHS.sonar.java.libraries=WebRoot/WEB-INF/lib

AvisosSuspRetIntranet.sonar.java.libraries=WebContent/WEB-INF/lib

ConstanciaInteresesIntranet.sonar.java.libraries=WebContent/WEB-INF/lib

SaldoFondoViviendaIntranet.sonar.java.libraries=WebContent/WEB-INF/lib

SaldosMovimientosIntranet.sonar.java.libraries=WebContent/WEB-INF/lib

#Exclusions

sonar.exclusions=.classpath,.factorypath,.pmd,.project,.apt\_generated,.settings,\*\*/\*.MF,.websettings

sonar.cpd.exclusions=.classpath,.factorypath,.pmd,.project,.apt\_generated,.settings,\*\*/\*.MF,.websettings



* + 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-scanner.bat***

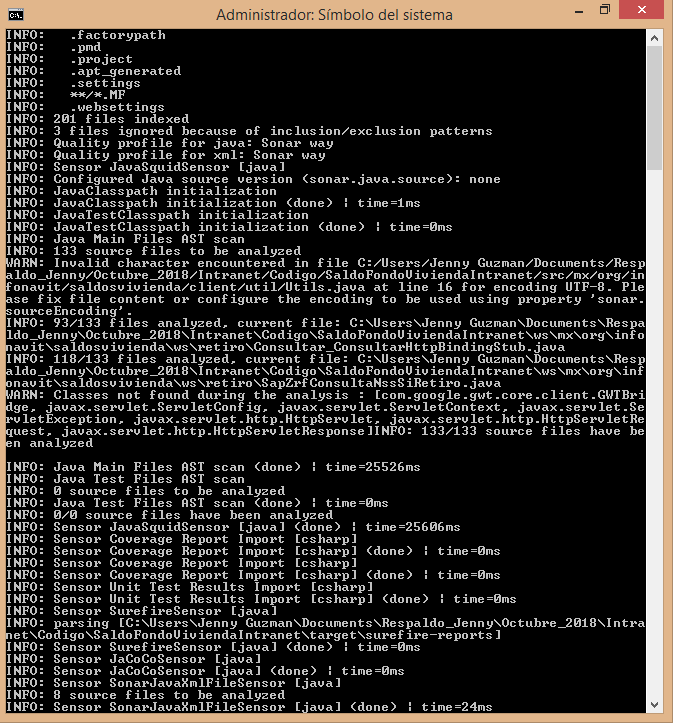


Ilustración Pantalla 1 de ejecución *sonar-scanner.bat*

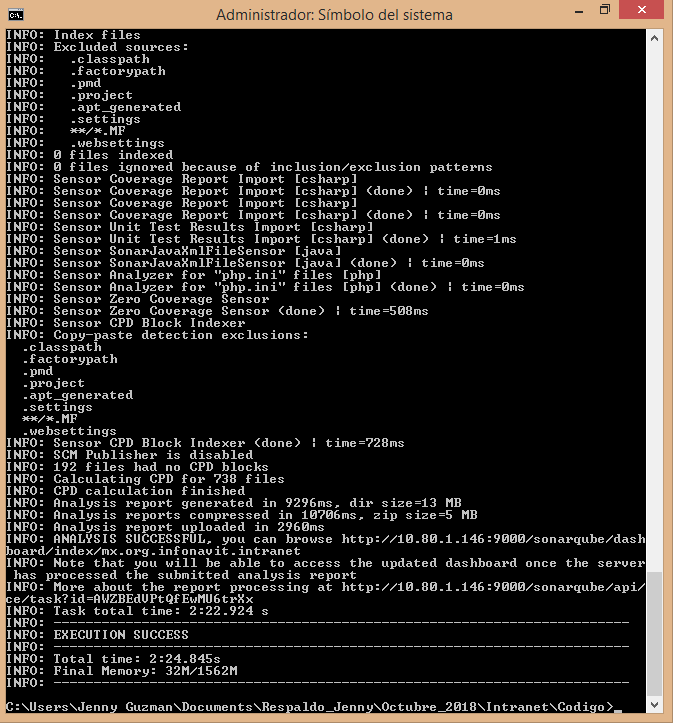


Ilustración Pantalla 2 de ejecución *sonar-scanner.bat*

* + 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:

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. | 4 |
| Bug | *Blocker* | **Los bucles no deben ser infinitos**  Un bucle infinito es aquel que nunca terminará mientras el programa se está ejecutando, es decir, es necesario forzar el cierre del programa para salir del bucle. Ya sea por cumplir con la condición final del bucle o por una ruptura, cada bucle debe tener una condición final. | 1 |
| Bug | *Critical* | **Las llamadas de función no deben pasar argumentos adicionales**  Es posible llamar a una función *JavaScript* con más argumentos de los necesita, pero los argumentos adicionales serán ignorados por la ejecución de la función. | 25 |
| *Bug* | *Major* | **Bloques ejecutados condicionalmente deben ser accesibles**  Las expresiones condicionales que son siempre verdaderas o falsas pueden conducir a código muerto. Dicho código siempre es considerado un *bug* y nunca debe usarse en la producción. | 26 |
| *Bug* | *Major* | **Los *Servlets* no deberían tener campos de instancia mutable**  Por contrato, un contenedor de *servlets* crea una instancia de cada *servlet* y luego se adjunta un hilo dedicado a cada nueva petición *HTTP* entrante para procesar esta petición. Así que todos los hilos están compartiendo las instancias de *servlet* y por extensión los campos de instancia. Para evitar alguna mala interpretación y comportamiento inesperado en tiempo de ejecución, todos los campos de *servlet* deben ser estáticos y/o finales, o simplemente eliminados. Con *Struts* 1. X, la misma restricción existe en *org.apache.struts.action.Action*. | 19 |
| *Bug* | *Major* | **Los punteros nulos (*Null Pointers*) no deben ser descodificados**  Una referencia a un *null* nunca debe ser descodificada/accedida. En caso de accesar, se lanzará una excepción del tipo *NullPointerException*. Tal excepción causará la terminación abrupta del programa o en el escenario más drástico, podría exponer información de depuración que sería útil para un atacante o podría permitir a un atacante eludir las medidas de seguridad. | 13 |
| *Bug* | *Major* | **No se debe acceder a las propiedades de las variables con valores "*null*" o "*undefined*"**  Cuando a una variable se le asigna un valor no *undefined* o *null*, no se generan propiedades. Intentar acceder a las propiedades de una variable de este tipo resulta en un *TypeError*, causando una terminación abrupta del *script* si el error no se detecta en un bloque de captura. | 7 |
| *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. | 5 |
| *Bug* | *Major* | **Los campos *Non-thread-safe* no deben ser estáticos**  No todas las clases de la librería *Java* estándar están escritas para ser seguras con los hilos. Es muy probable que su uso de múltiples subprocesos cause problemas con los datos o excepciones en tiempo de ejecución.  Esta regla plantea un problema cuando una instancia de validación de Calendar, *DateFormat, Javax. xml. xpath. XPath, orJavax. xml. SchemaFactory* está marcada como estática. | 4 |
| *Bug* | *Major* | **Los nombres de las propiedades no deben ser duplicados dentro de una clase u objeto**  *JavaScript* permite duplicar el nombre de las propiedades en las clases y objetos pero sólo la última instancia de un nombre duplicado determina el valor real que será utilizado. Por lo tanto, cambiar los valores de otras ocurrencias de un nombre duplicado no tendrá ningún efecto y puede causar errores. | 3 |
| *Bug* | *Major* | **Los bloques de multilínea deben estar encerrados en llaves**  Las llaves pueden omitirse de un bloque de una sola línea, por ejemplo con una sentencia *if* o para bucle, pero hacerlo puede ser engañoso e inducir a errores.  Esta regla plantea un problema cuando el espaciado en blanco de las líneas después de un bloque de una línea indica la intención de incluir esas líneas en el bloque, pero la omisión de las llaves significa que las líneas se ejecutarán incondicionalmente una vez. | 3 |
| *Bug* | *Major* | **"=+" no debe usarse en lugar de "+="**  El uso de operadores pares ( =+, =- o =! ) en los que el operador único invertido (+=, -= o! =) compilará y ejecutará, pero no producirá los resultados esperados.  Esta regla plantea un problema cuando =+, =-, =-, o =! se utiliza sin espacios entre los dos operadores y cuando hay al menos un carácter de espacio en blanco después. | 2 |
| *Bug* | *Major* | **No deben utilizarse moldes (casts) redundantes**  Expresiones innecesarias hacen que el código sea más difícil de leer y entender. | 2 |
| *Bug* | *Major* | **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. | 1 |
| *Bug* | *Major* | **No se deben ignorar los valores de retorno de las funciones sin efectos secundarios**  Cuando la llamada a una función no tiene efectos secundarios. En tal caso, o bien la llamada a la función no es viable y debe ser eliminada o el código fuente no se comporta como se esperaba.  Para evitar la generación de falsos positivos, esta regla activa una incidencia sólo en la siguiente lista predefinida de clases inmutables en la *API* de *Java*:   * java.lang.String * java.lang.booleano * Entero java.lang. * java.lang.Doble * java.lang.float * java.lang.byte * java.lang.Character * java.lang.Short * java.lang.stackTraceElement * java.time.DayOfWeek * java.time.Duration * java.time.Instant * java.time.LocalDate * java.time.LocalDateTime * java.time.LocalTime * java.time.Month * java.time.MonthDay * java.time.OffsetDateTime * java.time.OffsetTime * java.time.Period * java.time.Year * java.time.yearMonth * java.time.ZonedDateTime * java.math.BigInteger * java.math.BigDecimal * java.util.Optional | 1 |
| *Bug* | *Major* | **Bloques ejecutados condicionalmente deben ser accesibles**  Las expresiones condicionales que son siempre verdaderas o falsas pueden conducir a código muerto. Dicho código siempre es considerado un *bug* y nunca debe usarse en la producción. | 1 |
| *Bug* | *Major* | **No se deben utilizar operadores de igualdad estricta con tipos diferentes**  Realizar comparaciones de tipos diferentes utilizando operadores de igualdad estricta como === y !== siempre devolverá el mismo valor ya que no se realiza ninguna conversión de tipo antes de la comparación, por lo que se deduce se trata de un bug inmerso en el código. | 1 |
| *Bug* | *Major* | **No deben utilizarse expresiones idénticas en ambos lados de un operador binario**  No deben utilizarse expresiones idénticas en ambos lados de un operador binario ya que puede provocar un error.  Esta regla ignora \*, +, y = | 1 |
| Vulnerabilidad | *Blocker* | **Las credenciales (*Password*) no deben estar en código duro**  Debido a la facilidad de extraer cadenas de una aplicación compilada, las credenciales nunca deben estar en código duro. De ser así, está casi garantizado que terminarán en manos de un atacante. Esto es especialmente cierto en el caso de las aplicaciones distribuidas.  Las credenciales deben almacenarse fuera del código en un archivo de configuración o base de datos cifrado y fuertemente protegido. | 2 |
| 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. | 19 |
| 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. | 19 |
| Optimizador | *Blocker* | **Los campos de las clases hijo no deben ensombrecer los campos de la clase padre**  Tener una variable con el mismo nombre dentro de una jerarquía de clase genera confusión. | 4 |
| 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*. | 3 |
| Optimizador | *Blocker* | **Los métodos y los nombres de campo no deben ser iguales o diferir sólo por la capitalización**  Visualizar el conjunto de métodos en una clase, incluyendo los métodos de superclase, y encontrar dos métodos o campos que sólo difieren por capitalización es confuso para los usuarios de la clase. Es igualmente confuso tener un método y un campo que difieren sólo en mayúsculas o un método y un campo con exactamente el mismo nombre y visibilidad.  En el caso de los métodos, puede haber sido un error por parte del desarrollador original, quien intentó anular un método de superclase, pero en su lugar agregó un nuevo método con casi el mismo nombre. De lo contrario, esta situación indica una mala denominación. Los nombres de los métodos deben estar orientados a la acción y, por lo tanto, contener un verbo, lo cual es improbable en el caso de que tanto un método como un miembro tengan el mismo nombre (con o sin diferencias de capitalización). Sin embargo, cambiar el nombre de un método público podría ser perjudicial para las entidades que mandan llamar al método. Por lo tanto, cambiar el nombre del miembro es la acción recomendada. | 1 |
| *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. | 109 |
| *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 | 52 |
| *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. | 47 |
| *Optimizador* | *Critical* | **Los campos de una clase "*Serializable*" deben ser transitorios o serializables**  Datos no transitorios y no serializables podrían causar bloqueos del programa y abrir la puerta a los atacantes. En general, se espera que una clase *Serializable* cumpla con su contrato y no tenga un comportamiento inesperado cuando una instancia es serializada.    Esta regla plantea un problema en los campos no *Serializable* y en los campos de colección cuando no son privados (porque se les pueden asignar valores no serializables externamente) y cuando se les asignan tipos no serializables dentro de la clase. | 35 |
| *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. | 24 |
| *Optimizador* | *Critical* | **Las estructuras de control deben usar curly braces**  Si bien no es técnicamente incorrecto, la omisión de *curly braces* puede ser engañosa y puede llevar a la introducción de errores durante el mantenimiento. | 14 |
| *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. | 9 |
| *Optimizador* | *Critical* | Las constantes no deben definirse en interfaces  El patrón de interfaz constante es un mal uso de las interfaces. El hecho de que una clase utilice algunas constantes internamente es un detalle de implementación. La implementación de una interfaz constante hace que este detalle de implementación se filtre en la *API* exportada de la clase. Si en una futura versión la clase se modifica para que ya no necesite utilizar las constantes, debe implementar la interfaz para asegurar la compatibilidad binaria. Si una clase no final implementa una interfaz constante, todas sus subclases tendrán sus espacios de nombres contaminados por las constantes en la interfaz. | 7 |
| *Optimizador* | *Critical* | **Los tipos de *wildcard* genéricos no *deben* utilizarse en los parámetros de return**  Se recomienda no utilizar *wildcards* como tipos de devolución. Debido a que las reglas de inferencia de tipo son bastante complejas, es poco probable que el usuario de esa *API* sepa cómo utilizarla correctamente. El uso de tipos de *wildcards* debe limitarse a los parámetros del método. | 6 |
| *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. | 4 |
| *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 | 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. | 2 |
| *Optimizador* | *Critical* | **Los métodos de instancia no deben escribir en campos *"static”***  Actualizar correctamente un campo estático desde un método no estático es difícil de corregir y podría fácilmente llevar a errores si hay múltiples instancias de clase y/o múltiples hilos en juego. Idealmente, los campos estáticos sólo se actualizan a partir de métodos estáticos sincronizados. | 1 |
| *Optimizador* | *Major* | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 126 |
| *Optimizador* | *Major* | **Los parámetros de métodos no utilizados deben ser eliminados**  Los parámetros no utilizados son engañosos. Cualesquiera que sean los valores pasados a dichos parámetros, el comportamiento será el mismo. | 82 |
| *Optimizador* | *Major* | **Deberían eliminarse los *dead store***  Una *dead* *store* ocurre cuando a una variable local se le asigna un valor, incluyendo nulo, que no es leído por ninguna instrucción posterior. Calcular o recuperar un valor sólo para luego sobre escribirlo o desecharlo, podría indicar un error grave en el código. Aunque no sea un error, representa un desperdicio de recursos.  Incluso la asignación nula a una variable es un almacén muerto si la variable no se utiliza posteriormente. Asignar nulo como una pista al recolector de basura solía ser una práctica común, pero ya no es necesario y ese código debería ser eliminado. | 75 |
| *Optimizador* | *Major* | **Deberían eliminarse los *dead store***  Una *dead* *store* ocurre cuando a una variable local se le asigna un valor, incluyendo nulo, que no es leído por ninguna instrucción posterior. Calcular o recuperar un valor sólo para luego sobre escribirlo o desecharlo, podría indicar un error grave en el código. Aunque no sea un error, representa un desperdicio de recursos.  Incluso la asignación nula a una variable es un almacén muerto si la variable no se utiliza posteriormente. Asignar nulo como una pista al recolector de basura solía ser una práctica común, pero ya no es necesario y ese código debería ser eliminado. | 59 |
| *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. | 41 |
| *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 | 29 |
| *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. | 27 |
| *Optimizador* | *Major* | **Bloques ejecutados condicionalmente deben ser accesibles**  Las expresiones condicionales que son siempre verdaderas o falsas pueden conducir a código muerto. Dicho código siempre es considerado un *bug* y nunca debe usarse en la producción. | 26 |
| *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. | 25 |
| *Optimizador* | *Major* | **Las clases internas anónimas que contienen sólo un método deben convertirse en Lambdas**  Antes de la versión 8 de *Java*, la única manera de soportar parcialmente los cierres en *Java* era utilizando clases internas anónimas. Pero la sintaxis de las clases anónimas puede parecer difícil de manejar y poco clara. | 23 |
| *Optimizador* | *Major* | **Preconditions" y los argumentos de registro no deberían requerir evaluación**  Pasar los argumentos de mensaje que requieren una evaluación adicional en una *Guava.com.google.common.base.Preconditionscheck* puede resultar en una penalización de rendimiento. Esto se debe a que, independientemente de si es necesario o no, cada argumento debe resolverse antes de que se llame realmente al método.  Del mismo modo, pasar cadenas concatenadas a un método de registro también puede tener un impacto innecesario en el rendimiento debido a que las concatenaciones se realizarán cada vez que se llame al método, independientemente de que el nivel de registro sea lo suficientemente bajo como para mostrar el mensaje. En su lugar, se debe estructurar el código para pasar valores estáticos o pre-calculados en condiciones de verificación de condiciones previas y registro de llamadas.  Específicamente, el formato de cadena incorporado debe ser usado en lugar de concatenación de cadena, y si el mensaje es el resultado de una llamada de método, entonces las precondiciones deben ser omitidas en alto, y la excepción relevante debe ser lanzada condicionalmente en su lugar. | 22 |
| *Optimizador* | *Major* | **Las expresiones booleanas deben cambiar la evaluación de la condición**  Si una expresión booleana no cambia la evaluación de la condición, entonces es completamente innecesaria y puede ser eliminada. | 21 |
| *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. | 20 |
| *Optimizador* | *Major* | **Los bloques anidados de código no deben dejarse vacíos**  Los bloques de código vacíos tanto deben llenarse o retirarse. | 20 |
| *Optimizador* | *Major* | **Los *Servlets* no deberían tener campos de instancia mutable**  Por contrato, un contenedor de *servlets* crea una instancia de cada *servlet* y luego se adjunta un hilo dedicado a cada nueva petición *HTTP* entrante para procesar esta petición. Así que todos los hilos están compartiendo las instancias de *servlet* y por extensión los campos de instancia. Para evitar alguna mala interpretación y comportamiento inesperado en tiempo de ejecución, todos los campos de *servlet* deben ser estáticos y/o finales, o simplemente eliminados. Con *Struts* 1. X, la misma restricción existe en *org.apache.struts.action.Action*. | 19 |
| *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]\*$:* | 214 |
| *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. | 196 |
| *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. | 135 |
| *Optimizador* | *Minor* | **El operador diamante ("<>") debe ser utilizado**  La versión 7 de *Java* introdujo el operador de diamantes (<>) para simplificar el código genérico. | 112 |
| *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. | 70 |
| *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. | 62 |
| *Optimizador* | *Minor* | **Las variables locales no deben ser declaradas y luego devueltas o lanzadas inmediatamente**  Declarar una variable sólo para devolverla o desecharla inmediatamente es una mala práctica. Esta variable es un detalle interno de implementación que no está expuesto a los que llaman al método. El nombre del método debe ser suficiente para que las personas que llamen sepan exactamente lo que se devolverá. | 56 |
| *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. | 45 |
| *Optimizador* | *Minor* | **Los literales booleanos no deberían ser redundantes**  Los literales booleanos redundantes deben eliminarse de las expresiones para mejorar la legibilidad. | 30 |
| *Optimizador* | *Minor* | **Deben eliminarse las importaciones que no son utilizadas**  La parte de importaciones de un archivo debe ser manejada por el *Integrated Development Environment* (*IDE*), no manualmente por el desarrollador. Las importaciones no utilizadas no deberían producirse si ese es el caso. Dejarlos dentro reduce la legibilidad del código, ya que su presencia puede ser confusa. | 30 |
| *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. | 29 |
| *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. | 25 |
| *Optimizador* | *Minor* | **Las constantes públicas y los campos inicializados en la declaración deben ser "final estático (*static* *final*)" en lugar de meramente "*final*"**  Además, cuando un campo final no público no es también estático, implica que diferentes instancias pueden tener diferentes valores. Sin embargo, la inicialización de un campo final no estático en su declaración obliga a cada instancia a tener el mismo valor. Por lo tanto, estos campos deben ser estáticos o inicializados en el constructor. | 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]\*$* | 23 |
| *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\_]\*)\*$* | 20 |
| *Optimizador* | *Minor* | **Rastrear el uso de *tags* "TODO”**  Los *tags* “TODO” generalmente son utilizados para marcar ubicaciones en dónde se colocará más código en etapas posteriores del desarrollo.  Esta regla se genera con el objetivo de rastrear los *tags* y asegurar que no se mantengan en el código. | 57 |

El detalle completo del Análisis de Código Fuente puede ser consultado en la siguiente liga: [*http://10.80.1.146:9000/sonarqube7/projects*](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.