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

**Anexo Reporte de Evaluación de Solución Tecnológica – Sistema de Agendas de Autoevaluación**

**15/nov/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 Sistema de Agendas de Autoevaluación** tiene por objetivo permitir a los gerentes responsables de operar los procesos en las delegaciones autoevaluar las actividades críticas descritas en los procedimientos instituciones.

* 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**

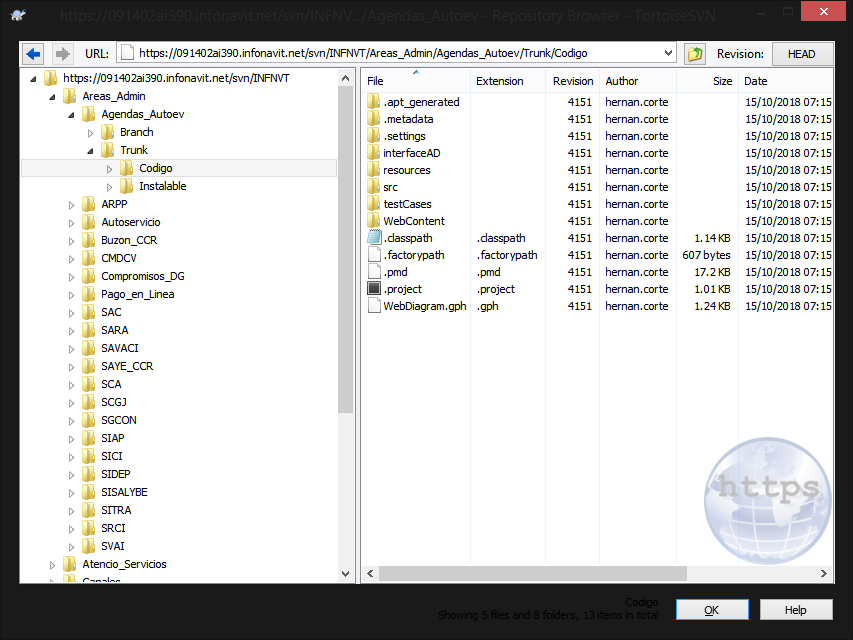
Los errores categorizados anteriormente son clasificados por severidades las cuales son definidas con base en 2 factores: **Impacto** y **Probabilidad**.

|  | **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/Areas\_Admin/Agendas\_Autoev*](https://091402ai390.infonavit.net/svn/INFNVT/Areas_Admin/Agendas_Autoev)



* + 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:

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

#

# á - \u00e1 Á - \u00c1

# é - \u00e9 É - \u00c9

# í - \u00ed Í - \u00cd

# ó - \u00f3 Ó - \u00d3

# ú - \u00fa Ú - \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. agendasautoeval

sonar.projectName=Sistema de Agendas de Autoevaluaci\u00f3n

sonar.projectVersion=1.0

sonar.scm.disabled=True

#Sources

sonar.sources=.

#Bin

sonar.java.binaries=Agendas/src

#Librerías

sonar.java.libraries=Agendas/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-scaner.bat***

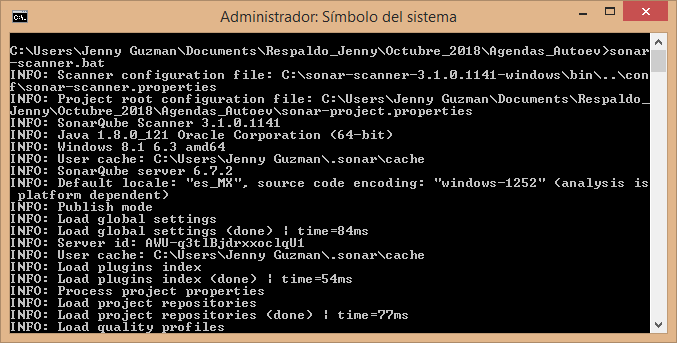


Ilustración Pantalla 1 de ejecución *sonar-scaner.bat*

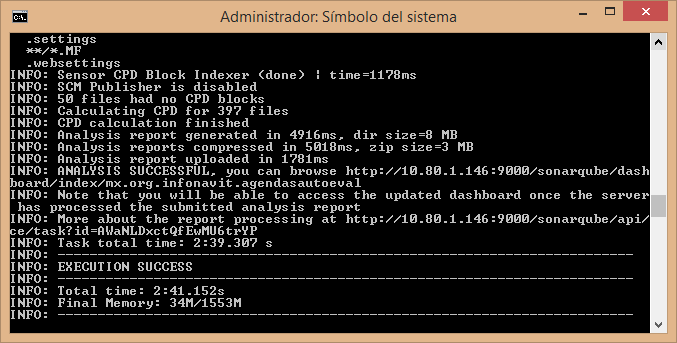


Ilustración Pantalla 2 de ejecución *sonar-scaner.bat*

* + 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:

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* | *Critical* | **Cero no debería ser un posible denominador**  Si el denominador a una operación de división o módulo es cero, resultaría en un error fatal. | 1 |
| Bug | *Major* | **No se deben hacer verificaciones de igualdad**  Las comparaciones de diferentes tipos siempre dan como resultado *false*. Por lo que las comparaciones y el código dependiente debe eliminarse. Esto incluye:   * Comparando un objeto con *null.* * Comparando un objeto con un primitivo no emparentado (E. G. *string* con un int). * Comparación de clases no relacionadas. * Comparación de una clase e interfaz no relacionada. * Comparación de tipos de interfaz no relacionados. * Comparando un *array* con un *no-array*. * Comparando dos *arrays.*   Específicamente en el caso de los *arrays*, dado que los *arrays* no anulan *Objects.equals(),* llamar iguales en dos *arrays* es lo mismo que comparar sus direcciones. Esto significa que *array1*.*equals*(*array2)* es equivalente a *array1*===*array2*.  Sin embargo, algunos equipos de desarrollo pueden esperar que *Array.equals(Object obj*) haga más que una simple comparación de direcciones de memoria, comparando por ejemplo el tamaño y contenido de los dos *arrays*. En su lugar, == *operator* o *Arrays.equals(array1, array2*) siempre debe ser usado con *arrays*. | 15 |
| 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. | 15 |
| 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 = | 7 |
| 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. | 5 |
| 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.  Definir una clase con un constructor duplicado generará un error. | 4 |
| 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. | 4 |
| Bug | *Major* | **No se deben hacer llamadas a valores que no se pueden llamar**  Sintácticamente es aceptable invocar cualquier expresión como si el valor fuera una función, pero se corre el riesgo de recibir un error *TypeError*. | 2 |
| Bug | *Major* | **Las variables no deben asignarse a sí mismas**  No es necesario reasignar una variable a sí misma ya que esta sentencia es redundante y debe eliminarse, o bien la reasignación es un error y en su lugar se ha previsto otro valor o variable para la asignación. | 2 |
| Bug | *Major* | **Las asignaciones no deben ser redundantes**  La propiedad transitiva indica que si A == B y B == C, entonces A == C. En tales casos, no es necesario asignar A a C o viceversa porque ya son equivalentes. Esta regla plantea un problema cuando una asignación es inútil porque la variable asignada ya contiene el valor en todos los trayectos de ejecución. | 2 |
| Bug | *Major* | **Los nombres de la interfaz deben cumplir una convención para definición de nombres**  Compartir algunas convenciones de nomenclatura es un punto clave para que un equipo pueda colaborar eficazmente. Esta regla permite comprobar que todos los nombres de interfaz coinciden con una expresión regular proporcionada. | 1 |
| Bug | *Major* | **Las sentencias de salto no deben ir seguidas de otras sentencias**  Las sentencias *Jump* (*return*, *break* y *continue*) y expresiones *throw* mueven el flujo de control fuera del bloque de código actual. | 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* | **Los resultados de "*compareTo*" no deben verificarse para valores específicos**  Mientras que la mayoría de los métodos de *comparaciónTo* devuelven -1,0 o 1, algunos no lo hacen y probar el resultado de *compareTo* contra un valor específico distinto de 0 podría dar lugar a falsos negativos. | 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 | 1 |
| *Bug* | *Minor* | **"*equals(Object obj)*" y "*hashCode()*" deben ser anulados por pares**   * Si dos objetos son iguales según el método de *equals*(*Object*), entonces es necesario llamar al método *hashCode* en cada uno de los dos objetos debe producir el mismo resultado entero. * No es necesario realizar la anulación si dos objetos son desiguales de acuerdo con el método *equals* (*Java.lang*.*Object*).   Para cumplir con este acuerdo, estos métodos deben ser heredados o sustituidos. | 1 |
| *Bug* | *Minor* | **Los parámetros del método, las excepciones capturadas y los valores iniciales de las variables *foreach* no deben ignorarse**  Técnicamente correcto asignar a los parámetros desde dentro de los cuerpos del método, hacerlo antes de que se lea el valor del parámetro es considerado un error. En cambio, los valores iniciales de los parámetros, las excepciones capturadas y los parámetros de *foreach* deben ser tratados como finales o leídos antes de la reasignación. | 1 |
| *Bug* | *Minor* | **"equals(Object obj)" debería probar el tipo de argumento**  Debido a que el método *equals* toma un Objeto genérico como parámetro, cualquier tipo de objeto puede ser pasado a él. El método no debe asumir que sólo se utilizará para probar objetos de su tipo de clase. En su lugar, debe comprobar el tipo de parámetro. | 1 |
| Vulnerabilidad | *Blocker* | **Deben utilizarse mecanismos de vinculación *Structured Query Language* (*SQL*)**  Las aplicaciones que ejecutan comandos *SQL* deben neutralizar cualquier valor proporcionado externamente utilizado en esos comandos. De lo contrario, un atacante podría incluir cadenas para modificar la consulta de manera que se ejecuten comandos involuntarios o se expongan datos confidenciales.  Esta regla verifica una variedad de métodos de diferentes *frameworks* que son susceptibles a la inyección *SQL* si no se usan correctamente. Los *frameworks* que se cubren son Java *Java Data Base Connection*, *Java Persistence API (JPA)*, *Java Data Objects (JDO),* *Hibernate* y *Spring*. Se prueban los siguientes métodos específicos:   * *org.hibernate.Session.createQuery* * *org.hibernate.Session.createSQLQuery* * *java.sql.Statement.executeQuery* * *java.sql.Statement.execute* * *java.sql.Statement.executeUpdate* * *java.sql.Statement.executeLargeUpdate* * *java.sql.Statement.addBatch* * *java.sql.Connection.prepareStatement* * *java.sql.Connection.prepareCall* * *java.sql.Connection.nativeSQL* * *javax.persistence.EntityManager.createNativeQuery* * *javax.persistence.EntityManager.createQuery* * *org.springframework.jdbc.core.JdbcOperations.batchUpdate* * *org.springframework.jdbc.core.JdbcOperations.execute* * *org.springframework.jdbc.core.JdbcOperations.query* * *org.springframework.jdbc.core.JdbcOperations.queryForList* * *org.springframework.jdbc.core.JdbcOperations.queryForMap* * *org.springframework.jdbc.core.JdbcOperations.queryForObject* * *org.springframework.jdbc.core.JdbcOperations.queryForRowSet* * *org.springframework.jdbc.core.JdbcOperations.queryForInt* * *org.springframework.jdbc.core.JdbcOperations.queryForLong* * *org.springframework.jdbc.core.JdbcOperations.update* * *org.springframework.jdbc.core.PreparedStatementCreatorFactory.<init>* * *org.springframework.jdbc.core.PreparedStatementCreatorFactory.newPreparedStatementCreator* * *javax.jdo.PersistenceManager.newQuery* * *javax.jdo.Query.setFilter* * *javax.jdo.Query.setGrouping*   Si se define un método en una interfaz, también se prueban las implementaciones. Por ejemplo, este es el caso de *org.springframework.jdbc.core.JdbcOperations*, que usualmente es usado como org*.springframework.jdbc.core.JdbcTemplate.* | 5 |
| 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. | 6 |
| 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. | 29 |
| 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. | 22 |
| 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. | 11 |
| 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*. | 3 |
| 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. | 2 |
| 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*. | 1 |
| 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.* | 49 |
| Optimizador | *Blocker* | **Los *Switch* *cases* deben terminar con una declaración incondicional "*break*"**  Cuando la ejecución no se termina explícitamente al final de un *switch* *case*, continúa ejecutando las sentencias del siguiente caso. Mientras que esto es a veces intencional, a menudo es un error que lleva a un comportamiento inesperado del aplicativo. | 1 |
| 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. | 152 |
| 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. | 121 |
| 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. | 32 |
| 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. | 15 |
| 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]+)\*$: | 13 |
| Optimizador | *Critical* | **La ejecución del *Garbage* *Collector* sólo debe ser activada por la *JVM***  No se recomienda llamar a *System.gc()* o *Runtime.getRuntime(). gc()* debido a que no hay manera de saber exactamente lo que se hará bajo la campana por el *JVM* debido a que el comportamiento dependerá de su proveedor, versión y opciones.  Una aplicación que se basa en estos métodos impredecibles también es impredecible y, por lo tanto aumenta el riesgo de fallo. La tarea de ejecutar el recolector de basura debe dejarse exclusivamente al *JVM*. | 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* | **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. | 938 |
| 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. | 187 |
| Optimizador | *Major* | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 174 |
| 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. | 99 |
| Optimizador | *Major* | **Las variables y funciones no deben volver a declararse**  Esta regla verifica que una declaración no utilice un nombre que ya está en uso. Es posible utilizar el mismo símbolo varias veces como variable o como función, pero hacerlo puede confundir al equipo responsable de dar mantenimiento al código. Adicional, es posible que tales reasignaciones se hagan por error, sin que el equipo de desarrollo 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. | 79 |
| 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. | 54 |
| 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. | 45 |
| 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. | 36 |
| 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. | 36 |
| 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.* | 28 |
| 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. | 21 |
| Optimizador | *Major* | **Declaraciones plegables de "*if*" deben fusionarse**  Se deben fusionar declaraciones “*if*” plegable si las sentencias aumentan la legibilidad del código. | 20 |
| 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. | 18 |
| Optimizador | *Major* | **"entrySet()" debe iterarse cuando se necesitan tanto la clave como el valor**  Cuando sólo se necesitan las teclas de un mapa en un bucle, iterar el keySet tiene sentido. Pero cuando se necesita tanto la clave como el valor, es más eficiente iterar el entrySet, ya que dará acceso tanto a la clave como al valor. | 17 |
| 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. | 16 |
| Optimizador | *Minor* | **El operador diamante ("<>") debe ser utilizado**  La versión 7 de *Java* introdujo el operador de diamantes (<>) para simplificar el código genérico. | 593 |
| 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. | 283 |
| 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. | 96 |
| 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. | 84 |
| Optimizador | *Minor* | **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. | 67 |
| 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]\*$:* | 66 |
| 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]\*$* | 62 |
| 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. | 39 |
| 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. | 24 |
| Optimizador | *Minor* | **El código deprecado debe ser eliminado**  Esta regla está pensada para ser usada como una forma de rastrear código que está marcado como obsoleto. El código deprecado debe ser eventualmente eliminado. | 23 |
| Optimizador | *Minor* | ***Collection.isEmpty()* debe usarse para probar vacíos**  *Collection.size ()* se usa para probar los trabajos vacíos, pero el uso de *Collection.isEmpty* () hace que el código sea más legible y más eficaz. La complejidad temporal de cualquier implementación del método *isEmpty* () debe ser *O(1)* considerando que algunas implementaciones de tamaño () pueden ser *O(n*). | 20 |
| 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. | 20 |
| 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]\*$:* | 19 |
| 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. | 19 |
| Optimizador | *Minor* | **Las capturas deberían combinarse**  Desde la versión 7 de *Java* ha sido posible capturar múltiples excepciones a la vez. Por lo tanto, cuando varios bloques de captura tienen el mismo código, deben combinarse para una mejor legibilidad. | 19 |

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.