

Práctica calificada 4 CC3S2

Fecha de entrega

- La entrega deberá realizarse antes de las 4 pm del lunes 25. Archivos entregados después de esta hora no serán considerados. Este horario será validado automáticamente por el servidor del correo y el repositorio GitHub.

Formato de entrega

- El archivo comprimido debe incluir un informe con las siguientes secciones, redactadas en tu propio estilo y con capturas de tu código ejecutándose:
 - Introducción: Explica tu razonamiento para el diseño.
 - Detalle del código: Incluye explicaciones de cada módulo en palabras propias.
 - Métricas y análisis: Presenta los resultados de herramientas de cobertura (pytest-cov) y complejidad (radon), con capturas y una conclusión personal.
- El repositorio debe contener al menos 4 commits significativos para cada servicio (gestión de usuarios y catálogo de productos), mostrando el progreso del desarrollo. Cada commit debe incluir un mensaje claro que describa los cambios realizados.

Instrucciones de desarrollo

Pregunta 1

Proyecto 1 y Proyecto 2:

1. **Organización del repositorio:** Los directorios user_management/ y product_catalog/ deben contener un archivo NOTES.md, donde expliques cómo aplicaste principios SOLID en cada módulo. Ejemplo: cómo separaste responsabilidades entre el controlador, el servicio y el repositorio.
2. **Pruebas y métricas:** Incluye pruebas unitarias y de integración completas. Cada archivo de pruebas debe tener al menos 3 casos de pruebas adicionales con escenarios inventados por ti. Si no se detectan pruebas únicas, no se otorgará puntaje completo en esta sección.

Docker y conexión de servicios

Configuración del entorno

Debes incluir un archivo docker-compose.override.yml, diseñado para simular fallos en la red entre los contenedores. En este archivo, implementa una configuración personalizada para validar el manejo de errores en la conexión.

Conexión entre servicios : Incluye capturas del funcionamiento de los servicios, generadas desde tus logs (obligatoriamente usando la librería estándar logging de Python con configuraciones personalizadas). Estas capturas deben incluir mensajes DEBUG e INFO indicando los pasos del flujo.

Pregunta 2

Configuración inicial

- Archivo de configuración obligatorio :** El archivo JSON o YAML de configuración debe incluir al menos 5 claves adicionales, personalizadas por el estudiante, que definen configuraciones específicas para los nodos (por ejemplo: disk_quota, network_bandwidth). Estas claves adicionales serán revisadas para detectar similitudes entre entregas.
- Propagación personalizada:** Debes implementar un retraso simulado (latencia) de forma aleatoria entre nodos, usando un rango definido por ti. Este retraso debe ser documentado en los logs generados por tu sistema.

Gestión de configuraciones

- Restricción de validaciones:** Debes implementar al menos dos validaciones adicionales para las configuraciones, como verificar formatos (por ejemplo, memory_limit debe incluir una unidad válida: MB/GB) o reglas específicas (por ejemplo, cpu_limit debe ser un múltiplo de 0.5). Estas validaciones deben estar documentadas en tu archivo de reporte.
- Errores controlados:** Si las configuraciones contienen errores, el sistema debe registrar logs detallados en formato JSON, describiendo el nodo afectado, el error detectado y la acción tomada (continuar o detener). Este log debe ser parte del reporte final.

Simulación de nodos

- Estados avanzados:** Cada nodo debe tener al menos 3 estados posibles (activo, inactivo, degradado). Estos estados deben influir en el comportamiento del nodo al recibir configuraciones. Por ejemplo, un nodo degradado podría aplicar configuraciones parcialmente o con advertencias."
- Personalización obligatoria:** El estudiante debe personalizar la cantidad de nodos y definir al menos dos nodos con comportamiento especial (por ejemplo, uno con fallos recurrentes y otro que rechace configuraciones específicas). Documenta estas personalizaciones en un archivo README.

Logging

- Niveles detallados de log:** Debes implementar mensajes diferenciados para cada nivel de log (DEBUG, INFO, ERROR), incluyendo ejemplos claros en los logs generados por tu sistema. Por ejemplo, los logs DEBUG deben incluir detalles sobre el tiempo de propagación entre nodos.

2. **Formato único:** El formato del log debe incluir un identificador único para cada operación (como un timestamp o ID generado). Los estudiantes deben documentar cómo diseñaron este formato en su informe.

Interfaz de línea de comandos (CLI)

1. **Opciones avanzadas:** Además de las funcionalidades requeridas, implementa un comando adicional para simular el reinicio de nodos específicos. Este comando debe generar un log explicando los cambios realizados.
2. **Restricción en el diseño:** La CLI debe incluir un menú interactivo (usando argparse o click), y cada opción debe estar documentada con ejemplos prácticos en el README.

Reporte final

1. **Contenido del reporte:** El reporte debe incluir:
 - Estado final de cada nodo (activo/inactivo/degradado).
 - Errores encontrados durante la propagación.
 - Tiempo total del proceso.
 - Análisis personal de los resultados y propuestas de mejora."
2. **Validación contra logs:** El reporte debe incluir capturas de los logs generados, explicando cómo se relacionan con el estado final de cada nodo.

Pruebas

Pruebas unitarias

1. **Cobertura única:** Debes incluir al menos 3 escenarios no triviales en tus pruebas unitarias. Por ejemplo, nodos que rechacen configuraciones específicas o propagación que falle a la mitad. Estas pruebas serán revisadas para asegurar su originalidad."
2. **Reporte automatizado:** Genera un reporte de cobertura de código usando pytest-cov, y documenta los resultados en tu informe. La cobertura debe ser mayor al 90% para obtener el puntaje completo.
- 3.

Pruebas de integración

1. **Simulación compleja:** Implementa un clúster con al menos 15 nodos, de los cuales al menos 3 deben estar inactivos y 2 degradados. Documenta cómo diseñaste esta simulación en tu README.
2. **Validación exhaustiva :** Incluye pruebas para:
 - La propagación completa en presencia de fallos.
 - El manejo correcto de logs para nodos fallidos.
 - La generación del reporte final, verificando su contenido."

Estilo de código

1. **Restricción adicional:** El código debe seguir los principios SOLID estrictamente, y cada clase debe incluir una docstring explicando su responsabilidad. Ejemplo: class Node debe tener una docstring que detalle cómo maneja estados y configuraciones.
2. **Modularidad obligatoria:** Cada funcionalidad (validación, propagación, logging) debe estar en un módulo separado. Código monolítico será penalizado.

Métricas

1. **Cobertura obligatoria:** La cobertura de código debe superar el 90%, y los métodos con complejidad ciclomática mayor a 10 deben ser refactorizados antes de entregar."
2. **Análisis personal:** Incluye un análisis explicando por qué ciertas partes de tu código tienen mayor complejidad, y qué mejoras implementarías para reducirla."

Preguntas adicionales

Restricciones integradas en las preguntas

1. **Mocks y stubs:**
 - a. "Además de responder la pregunta, incluye un ejemplo práctico en código Python mostrando cómo implementarías un mock y un stub para la interacción entre el kubelet y el API Server. Respuestas sin ejemplos no serán evaluadas."
2. **Inyección de dependencias:**
 - a. "La respuesta debe incluir un fragmento de código que explique cómo inyectarías dependencias en el Scheduler, simulando respuestas controladas del etcd y el API Server. Justifica cómo tu implementación sigue principios SOLID."
3. **Principios SOLID en el Controller Manager:**
 - a. "Explica cómo separarías responsabilidades del Controller Manager, incluyendo clases específicas y sus funciones. Respuestas generales o sin ejemplos no puntuarán."
4. **Uso de fakes para el kube-proxy:**
 - a. "Incluye un ejemplo de código para simular tráfico de red con un fake. Describe cómo este fake se diferencia de un mock o stub en tu diseño."
5. **Spy con pytest:**
 - a. "Proporciona un ejemplo práctico usando pytest-mock para capturar interacciones entre el Controller Manager y el Cloud Provider API. Justifica tu implementación."

Preguntas

En el uso de mocks y fakes: Cuando respondas sobre mocks y stubs, incluye ejemplos prácticos en pseudocódigo basados en el escenario de la figura. Si la respuesta solo incluye texto sin ejemplos o es general, no se calificará esta sección.

Uso de dependencias inyectadas: La respuesta debe incluir cómo aplicarías el patrón de inyección de dependencias, con código de ejemplo que explique la interacción entre el Scheduler y los nodos."

Justificación del diseño SOLID: Incluye ejemplos de cómo dividirías responsabilidades del Controller Manager. Si no justificas el uso de al menos dos principios SOLID, no se otorgará el puntaje.

Reporte final

Formato y validación del reporte : El reporte debe incluir un análisis personal sobre las métricas obtenidas: ¿por qué crees que algunas partes de tu código tienen mayor complejidad ciclomática? Propón dos mejoras basadas en tus resultados. Respuestas genéricas no serán aceptadas.

Criterios de evaluación

Entrega completa y autenticidad: Si se detectan respuestas similares entre estudiantes o patrones comunes con herramientas generativas, se descalificará automáticamente la práctica.

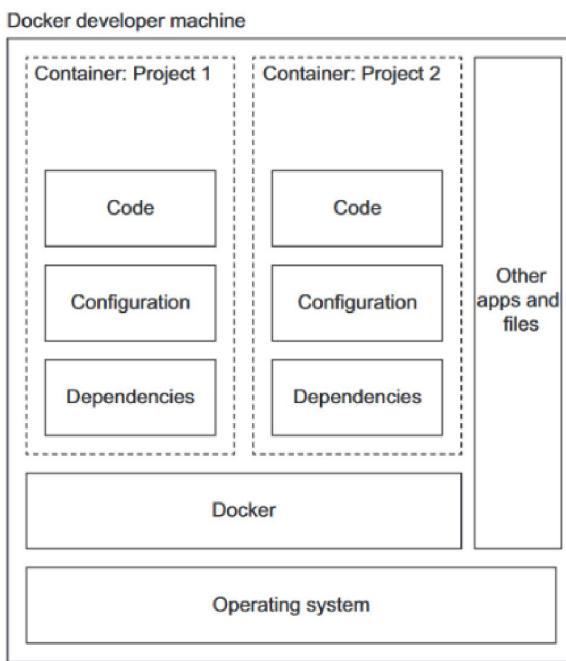
Diseño único: El diseño de los servicios y la implementación de Docker Compose deben ser originales. Cualquier similitud no justificada entre entregas será evaluada con puntuación mínima en esta sección.

Notas adicionales para el estudiante: Se espera que esta práctica sea un reflejo de tu propio razonamiento y aprendizaje. Respuestas genéricas, copias de otros estudiantes o salidas generadas automáticamente serán consideradas plagio y no serán evaluadas.

Ejercicio: Desarrollo de pequeños servicios con conexión y Docker

Objetivo

Desarrollar dos pequeños servicios independientes pero conectados, implementando principios SOLID, inyección de dependencias, dobles de prueba y evaluando métricas de calidad como cobertura de código y complejidad ciclomática. El ejercicio se basa en la arquitectura representada en la figura proporcionada, donde cada proyecto se ejecuta en su propio contenedor Docker. Ambos proyectos se gestionarán desde un único repositorio de Git.



La figura ilustra un entorno de desarrollo en una máquina con Docker. Cada contenedor contiene:

1. **Code:** El código del proyecto (proyecto 1 o proyecto 2).
2. **Configuration:** Archivos de configuración específicos del proyecto.
3. **Dependencies:** Las dependencias requeridas por el proyecto.

En este escenario:

- **Proyecto 1:** Representa un servicio de gestión de usuarios.
- **Proyecto 2:** Representa un servicio de catálogo de productos.

Ambos servicios están conectados a través de una red Docker para permitir la interacción entre ellos. Esta conexión se implementará utilizando Docker Compose.

Estructura del ejercicio

1. Organización del repositorio

El repositorio contendrá ambos proyectos con la siguiente estructura (referencia):

```
/ServiceProject
├── .devcontainer/ (Configuración común para ambos proyectos)
├── docker-compose.yml (Archivo para conectar los servicios en una red Docker)
└── user_management/ (Proyecto 1: gestión de usuarios)
    ├── src/
    ├── tests/
    └── requirements.txt
```

```
|   └── configuraciones específicas del contenedor  
|   └── product_catalog/  (Proyecto 2: catálogo de productos)  
|       ├── src/  
|       ├── tests/  
|       └── requirements.txt  
|   └── configuraciones específicas del contenedor  
└── README.md
```

2. Configuración de los contenedores

Cada proyecto se ejecutará en su propio contenedor, como muestra la figura. Esto asegura:

- **Aislamiento:** Cada contenedor tiene su propio código, configuración y dependencias.
- **Flexibilidad:** Los servicios pueden ser modificados o actualizados independientemente.

3. Conexión entre proyectos

- Los contenedores estarán conectados a través de una red Docker común definida en docker-compose.yml.
- El servicio **catálogo de productos** realizará solicitudes HTTP al servicio **gestión de usuarios** para validar usuarios al asociar productos.

Tareas del ejercicio

Proyecto 1: Gestión de usuarios (2 puntos)

1. **Funcionalidad:**
 - a. Crear, leer, actualizar y eliminar usuarios.
 - b. Exponer un API REST para estas operaciones.
2. **Diseño:**
 - a. Organizar el código en módulos: controladores, servicios y repositorios.
 - b. Aplicar principios SOLID, asegurando una separación clara de responsabilidades.
3. **Pruebas:**
 - a. Implementar pruebas unitarias y de integración para cada componente.
 - b. Simular dependencias con dobles de prueba.
4. **Métricas:**
 - a. Medir la cobertura de código y la complejidad ciclomática.

Observación: Debes completar todos los ítems para el puntaje total.

Proyecto 2: Catálogo de productos (2 puntos)

- 1. Funcionalidad:**
 - a. Crear, leer, actualizar y eliminar productos.
 - b. Asociar cada producto a un usuario, validando su existencia con el servicio de Gestión de Usuarios.
- 2. Diseño:**
 - a. Seguir principios SOLID y estructurar el código en módulos.
 - b. Implementar mocks para probar la integración con el servicio de usuarios.
- 3. Pruebas:**
 - a. Diseñar pruebas unitarias e integración.
 - b. Asegurarse de que los casos de prueba cubren la conexión entre servicios.
- 4. Métricas:**
 - a. Evaluar la cobertura de código y la complejidad del sistema.

Observación: Para poder realizar la siguiente sección debes completar las dos anteriores-

Actividades principales (4 puntos)

- 1. Configuración del entorno:**
 - a. Configurar un entorno de desarrollo estándar utilizando DevContainers.
 - b. Crear un archivo docker-compose.yml para orquestar los servicios.
- 2. Diseño de los servicios:**
 - a. Cada servicio debe ser modular y seguir los principios SOLID.
 - b. Usar inyección de dependencias para desacoplar la lógica de negocio del acceso a datos.
- 3. Conexión entre servicios:**
 - a. Implementar la comunicación entre los microservicios mediante solicitudes HTTP.
 - b. Manejar posibles errores en la conexión y garantizar la robustez.
- 4. Pruebas y métricas:**
 - a. Diseñar pruebas unitarias para validar cada componente.
 - b. Generar reportes de cobertura de código y analizar la complejidad ciclomática.

Observación: Debes completar todos los ítems para que puestue la pregunta.

Entrega

- 1. Código del repositorio:**
 - a. Incluir ambos proyectos con configuraciones específicas.
 - b. Incluir un archivo README.md con instrucciones para ejecutar y probar los servicios.
- 2. Informe de métricas:**
 - a. Reportar la cobertura de código y la complejidad ciclomática de ambos proyectos.
 - b. Comparar las métricas obtenidas y proponer mejoras.
- 3. Demostración:**

- a. Ejecutar ambos servicios simultáneamente usando Docker Compose.
- b. Probar la interacción entre los servicios.

Criterios de evaluación

1. **Diseño:**
 - a. Aplicación de principios SOLID.
 - b. Separación clara de responsabilidades y modularidad del código.
2. **Pruebas:**
 - a. Uso efectivo de pruebas unitarias e integración.
 - b. Cobertura de código mayor al 80%.
3. **Conexión:**
 - a. Funcionamiento correcto de la comunicación entre los servicios.
 - b. Manejo robusto de errores en la conexión.
4. **Calidad del código:**
 - a. Complejidad ciclomática aceptable en los módulos principales.
 - b. Documentación clara y concisa.
5. **Entrega final:**
 - a. Configuración funcional del entorno de desarrollo con DevContainers y Docker Compose.
 - b. Código bien estructurado y documentado.

Ejercicio: Sistema de gestión de configuraciones distribuidas

Desarrolla un sistema de gestión de configuraciones distribuidas que simule un entorno Kubernetes, enfocado en el manejo secuencial de configuraciones entre nodos. Este sistema debe:

- **Validar** configuraciones antes de su propagación.
- **Actualizar** los nodos secuencialmente.
- **Manejar errores** y generar reportes detallados.
- **Implementar pruebas exhaustivas** para asegurar la robustez del sistema.

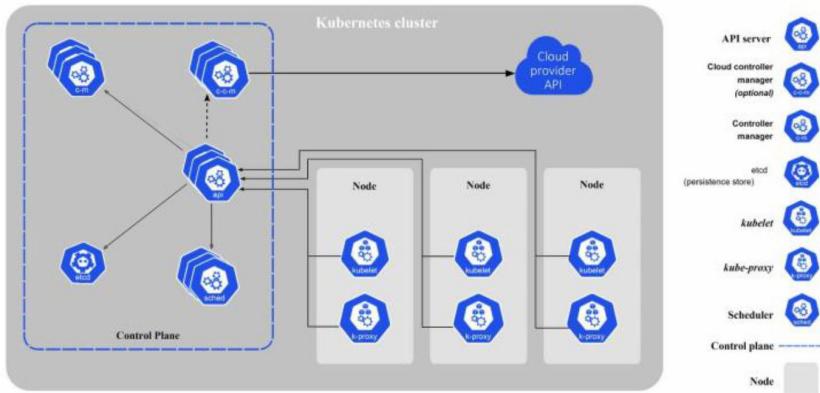
Objetivo

Diseñar e implementar un sistema modular que refleje la propagación de configuraciones en un clúster de Kubernetes, representando componentes como:

- **ConfigManager**
- **Node**
- **Logger**

El sistema debe seguir un flujo secuencial en la propagación de configuraciones. Debes utilizar en tus respuestas la siguiente figura dada en clases.

Este ejercicio **no requiere** una instalación real de Kubernetes. El objetivo es **simular** un sistema de gestión de configuraciones distribuidas que refleje el comportamiento de un clúster de Kubernetes, pero implementado completamente en código (por ejemplo, en Python).



Requisitos

1. Funciones principales (6 puntos)

a. Gestión de configuraciones (1 punto)

- El **ConfigManager** debe:
 - **Cargar configuraciones** desde un archivo (JSON o YAML).
 - **Validar** que las configuraciones incluyen las claves obligatorias:
 - `memory_limit`
 - `cpu_limit`
 - **Verificar** que los valores están dentro de rangos válidos.
 - **Propagar** las configuraciones a los nodos simulados en orden secuencial.

b. Simulación de nodos (1 punto)

- Los **nodos** deben:
 - **Recibir y aplicar** configuraciones.
 - **Generar errores** si están inactivos, sin interrumpir la propagación a los siguientes nodos.

c. Logging (2 puntos)

- **Registrar** cada operación, incluyendo:
 - **Errores**
 - **Configuraciones exitosas**
- Implementar niveles de logging:
 - DEBUG

- INFO
- ERROR

d. Interfaz de línea de comandos (CLI) (1 punto)

- Permitir al usuario:
 - Cargar configuraciones
 - Visualizar el estado de los nodos
 - Actualizar configuraciones específicas desde la línea de comandos

e. Reporte final (1 punto)

- Generar un reporte al finalizar que incluya:
 - El estado de cada nodo (éxito o fallo)
 - El tiempo total del proceso

Observación: No puedes pasar esta sección si haber completado los pasos anteriores. No hay puntos intermedios. Debes completar todas las secciones.

2. Pruebas (2 puntos)

a. Pruebas unitarias

- Probar la validación de configuraciones:
 - Válidas e inválidas
- Simular actualización de:
 - Nodos activos
 - Nodos inactivos
- Asegurar que el logger registra eventos correctamente

b. Pruebas de integración

- Simular un clúster con:
 - 10 nodos activos
 - 2 nodos inactivos
- Validar:
 - La propagación secuencial
 - Los reportes generados

Observación: No puedes pasar esta sección si haber completado los pasos anteriores. La puntuación es total no existen puntos medios.

3. Estilo de código

- El código debe ser:
 - Modular
 - Seguir los **principios SOLID**

4. Métricas (2 puntos)

- Cobertura de código:
 - Mayor al **90%** (medida con pytest-cov)
- Complejidad ciclomática:
 - Menor a **10** por método (medida con radon)

Observación: No puedes pasar esta sección si haber completado los pasos anteriores. La puntuación es total no existen puntos medios.

Entradas

1. Archivo de Configuración:
 - a. **Formato:** JSON o YAML.
 - b. **Contenido:** Configuraciones que deben incluir las claves obligatorias:
 - i. `memory_limit`: Define el límite de memoria para un nodo.
 - ii. `cpu_limit`: Define el límite de CPU para un nodo.
 - c. **Valores:** Deben estar dentro de rangos válidos predefinidos, por ejemplo:
 - i. `memory_limit`: Entre 256MB y 64GB.
 - ii. `cpu_limit`: Entre 0.5 y 32 cores.
2. Comandos de la CLI:
 - a. **Cargar configuraciones:** Permite al usuario especificar el archivo de configuración a cargar.
 - b. **Visualizar estado de los nodos:** Muestra el estado actual (activo/inactivo) de cada nodo.
 - c. **Actualizar configuraciones específicas:** Permite modificar valores de configuración directamente desde la línea de comandos.

Salidas

1. Logs de operaciones:
 - a. **Niveles de logging:**
 - i. DEBUG: Información detallada para diagnóstico.

- ii. INFO: Confirmación de que las cosas están funcionando como se esperaba.
- iii. ERROR: Errores debido a eventos más serios.

b. **Contenido:**

- i. Operaciones realizadas (carga, validación, propagación).
- ii. Errores encontrados durante la validación o propagación.
- iii. Confirmación de configuraciones aplicadas exitosamente en los nodos.

2. **Reporte final:**

a. **Estado de cada nodo:**

- i. Indica si la configuración se aplicó con éxito o si falló.
- ii. Incluye detalles del error si corresponde.

b. **Tiempo total del proceso:**

- i. Mide el tiempo desde el inicio hasta la finalización de la propagación de configuraciones.

3. **Salida de la CLI:**

- a. Mensajes al usuario indicando el resultado de las operaciones solicitadas.
- b. Información del estado de los nodos cuando se solicita.

Técnicas

1. **Diseño e implementación:**

a. **Programación orientada a objetos:**

- i. **Clases principales:**
 - 1. ConfigManager: Maneja la carga y validación de configuraciones.
 - 2. Node: Representa un nodo en el clúster. Puede estar activo o inactivo.
 - 3. Logger: Gestiona el registro de eventos en diferentes niveles.

b. **Principios SOLID:**

- i. Single Responsibility Principle: Cada clase tiene una única responsabilidad.
- ii. Open/Closed Principle: Las clases están abiertas para extensión pero cerradas para modificación.
- iii. Liskov Substitution Principle: Las clases derivadas deben ser sustituibles por sus clases base.
- iv. Interface Segregation Principle: Las interfaces deben ser específicas para cada cliente.
- v. Dependency Inversion Principle: Depender de abstracciones en lugar de concreciones.

2. **Validación de configuraciones:**

- a. Verificar que todas las claves obligatorias están presentes.
- b. Asegurar que los valores están dentro de los rangos permitidos.
- c. Manejar excepciones si se encuentran errores durante la validación.

3. **Propagación secuencial:**

- a. Los nodos reciben las configuraciones uno tras otro.
- b. Si un nodo está inactivo, se registra un error pero el proceso continúa con el siguiente nodo.
- c. Se puede simular latencia o tiempos de respuesta para cada nodo.

4. **Simulación de nodos:**
 - a. Crear una lista o colección de objetos Node.
 - b. Asignar estados de actividad (activo/inactivo) a los nodos.
 - c. Los nodos inactivos lanzan excepciones o retornan errores cuando se intenta aplicar una configuración.
5. **Logging:**
 - a. Utilizar la librería estándar logging de Python.
 - b. Configurar manejadores para escribir en consola y/o archivos.
 - c. Formatear los mensajes de log para incluir timestamps y niveles de severidad.
6. **Interfaz de línea de comandos (CLI):**
 - a. Utilizar librerías como argparse o click para manejar argumentos y comandos.
 - b. Proporcionar ayuda y documentación accesible desde la línea de comandos.
7. **Pruebas unitarias y de integración:**
 - a. **Herramientas:**
 - i. pytest: Para escribir y ejecutar pruebas.
 - ii. pytest-cov: Para medir la cobertura de código.
 - b. **Cobertura de código:**
 - i. Apuntar a más del 90% de cobertura.
 - c. **Complejidad ciclomática:**
 - i. Utilizar radon para medir y asegurar que la complejidad por método sea menor a 10.

Flujo general del sistema

1. **Inicio del programa:**
 - a. El usuario ejecuta el programa desde la línea de comandos con las opciones deseadas.
2. **Carga de configuración:**
 - a. El ConfigManager carga el archivo especificado.
 - b. Se realiza la validación de los datos.
3. **Propagación de configuración:**
 - a. Se inicia la propagación secuencial a cada nodo.
 - b. Para cada nodo:
 - i. Si el nodo está activo:
 1. Se aplica la configuración.
 2. Se registra un log de éxito.
 - ii. Si el nodo está inactivo:
 1. Se captura el error.
 2. Se registra un log de error.
 - iii. El proceso continúa con el siguiente nodo independientemente del estado.
4. **Generación de reporte:**
 - a. Al finalizar la propagación, se compila el reporte final.
 - b. Se muestra o guarda el reporte según se haya configurado.

Preguntas adicionales

Relacionadas a la figura dada, responde de manera consecuente las siguientes preguntas:

- **Pregunta:** En un escenario donde necesitas probar la interacción del kubelet con el API Server, ¿cómo implementarías mocks para simular la respuesta del API Server? ¿Cómo diferenciarías el uso de un mock y un stub para esta interacción?
- **Pregunta:** El Scheduler requiere información del etcd y del API Server. ¿Cómo aplicarías el patrón de inyección de dependencias para testear el comportamiento del Scheduler con respuestas controladas?
- **Pregunta:** Considera que el Controller Manager tiene varias responsabilidades en esta arquitectura. ¿Qué principios SOLID aplicarías para separar esas responsabilidades en componentes más pequeños y testeables?
- **Pregunta:** Si necesitas probar el kube-proxy pero sin usar un entorno de red real, ¿cómo implementarías un fake para simular el tráfico de red esperado?
- **Pregunta:** Supón que quieres verificar cuántas veces el Controller Manager interactúa con el Cloud Provider API. ¿Cómo usarías un spy con pytest para capturar estas interacciones?

Observación: Contesta todas las preguntas para que puntue. No se admite preguntar solo responder algunas de las preguntas del cuestionario.