



Evolución y Gestión de la Configuración

Prueba de mínimos

Grado en Ingeniería Informática – Ingeniería del Software · Universidad de Sevilla ·
Curso 2026-2027

Como parte de un proceso selectivo, te han citado a una entrevista con el equipo que implanta la filosofía DevOps en una empresa tecnológica. Te harán preguntas de 5 bloques temáticos: en cada bloque, una pregunta conceptual y otra técnica. Varias preguntas técnicas se refieren a la aplicación web de gestión de películas (hecha con Flask) que desarrolla la empresa. Cada pregunta vale hasta 1 punto. Para obtener la calificación de **apto** tienes que cumplir dos criterios:

1. no obtener 0 puntos en ningún bloque temático; y
2. sumar al menos 5 puntos en total.

Importante: responde cada pregunta únicamente dentro de su recuadro en las hojas de respuesta, al final del cuadernillo. No se corregirá nada que se escriba fuera del recuadro ni en folios aparte.

¡Aquí comienza la entrevista! ¡Mucha suerte!

Criterios generales de corrección

- Cada pregunta se divide en dos componentes, A y B, de 0,5 puntos. Un componente puntúa 0,5 si cumple su criterio y 0 en caso contrario. La nota de la pregunta es la suma: 0, 0,5 o 1.
- Cada componente se evalúa con independencia del otro: puntúa aunque el otro esté en blanco o sea incorrecto. La rúbrica de cada pregunta indica a qué corresponde cada componente (un apartado del enunciado o un aspecto de la respuesta).
- Se aceptan formulaciones y nombres equivalentes a los de la respuesta esperada si el contenido es correcto.

Bloque 1 · CI/CD**1.1 Conceptual***(1 punto)*

Una persona recién incorporada a la empresa se ha unido al equipo que desarrolla un reproductor de vídeo para escritorio. Los usuarios descargan e instalan manualmente cada nueva versión en sus ordenadores, y el equipo mantiene a la vez varias versiones del reproductor para cada sistema operativo soportado. En su primera reunión, esta persona propone pasar a despliegue continuo “porque es la última tendencia en la industria”.

- a) Explica la diferencia entre entrega continua y despliegue continuo.
- b) ¿Encaja su propuesta con este producto? ¿Por qué?

Idea fundamental que se evalúa

La entrega continua mantiene el software siempre en un estado liberable, mientras que el despliegue continuo lleva automáticamente a producción cada release validada; elegir una u otra depende de si el equipo controla el entorno donde se ejecuta el software (2.2.5, 2.3.4, tabla 2.2).

Respuesta esperada

- a) **Entrega continua:** cada cambio que supera el pipeline deja una release identificada, trazable y lista para liberarse, pero el paso a producción (o la publicación) es una decisión expresa del equipo. **Despliegue continuo:** va un paso más allá: si la release supera todas las validaciones, se despliega *automáticamente* en producción, sin decisión manual (2.2.5, 2.3.4, tabla 2.2).

- b) **No encaja bien en este caso.** El despliegue continuo tiene sentido cuando el equipo controla el entorno de ejecución (servicios web, API, nube). Aquí cada usuario instala manualmente las nuevas versiones, y mantener varias versiones por sistema operativo exige decidir qué cambios llegan a cada una y cuándo se publica cada una. Lo razonable es practicar entrega continua (producto siempre listo para liberarse) y publicar siguiendo una política controlada (2.3.4, A.3).

Comp.	Puntúa si...
A (0,5)	a) Diferencia correctamente ambos conceptos: en la entrega continua el paso a producción es una decisión expresa y en el despliegue continuo es automático si se superan las validaciones.
B (0,5)	b) Razona sobre el caso a partir de sus características (instalación manual, varias versiones por sistema operativo, instalación y actualización en los equipos de los usuarios, que no controla el equipo...) y concluye que conviene entrega continua. También puntúa una respuesta condicional bien razonada (p. ej. “solo encajaría si hubiera actualizaciones automáticas controladas por el equipo”). No puntúa aceptar la propuesta porque “es tendencia” ni rechazarla sin argumentos.

1.2 Técnica

(1 punto)

La aplicación de películas se despliega automáticamente en preproducción con este workflow de GitHub Actions:

```

1 name: Deploy to preproduction
2 on:
3   push:
4     branches: [trunk]
5 jobs:
6   testing:
7     runs-on: ubuntu-24.04
8     steps:
9       - uses: actions/checkout@v5
10      - run: pip install -r requirements.txt
11      - run: pytest
12   deploy:
13     needs: testing
14     runs-on: ubuntu-24.04
15     steps:
16       - run: curl "${{ secrets.RENDER_PREPROD_DEPLOY_HOOK_URL }}"

```

- a) ¿Qué ocurre con el despliegue si falla alguna prueba? ¿Qué línea del fichero lo determina?
- b) En Render ya está configurado el servicio de producción, vinculado a la rama **main**, que solo se despliega a través de su *hook*. La URL de ese *hook* está guardada en el secreto **RENDER_PROD_DEPLOY_HOOK_URL**. Quieren un workflow equivalente que despliegue en **producción** cada vez que se actualice la rama **main**. Indica qué líneas cambiarías (por su número) y cómo quedarían.

Idea fundamental que se evalúa

Un pipeline de CI/CD automatiza el camino de un cambio hacia un entorno e impide que avance si no supera las pruebas; el mismo proceso puede llevar el software a distintos entornos cambiando solo su configuración (2.2.5, 2.3.2, 2.3.3).

Respuesta esperada

- a) El job **deploy** no se ejecuta, porque depende del job **testing** mediante **needs: testing** (línea 13): si las pruebas fallan, el cambio no avanza hacia el entorno (2.2.5, 2.3.3).
- b) Basta con cambiar dos líneas (y, opcionalmente, el nombre de la línea 1):

```

4     branches: [main]
16    - run: curl "${{ secrets.RENDER_PROD_DEPLOY_HOOK_URL }}"

```

Comp.	Puntúa si...
A (0,5)	a) Indica que el job de despliegue no se ejecuta si fallan las pruebas e identifica needs: testing (línea 13) como la causa.
B (0,5)	b) Cambia la rama a main (línea 4) y el secreto por RENDER_PROD_DEPLOY_HOOK_URL (línea 16). Si solo cambia uno de los dos, no puntúa.

Bloque 2 · Gestión del código fuente

2.1 Conceptual

(1 punto)

Un compañero del equipo del reproductor de vídeo ha terminado un módulo que llama a la función **calcular_tasa_reproduccion**. Antes de subir su trabajo, se trae los últimos cambios del repositorio compartido y Git completa el *merge* sin avisar de ningún conflicto. Sin embargo, al pasar las pruebas, una falla: otra persona había renombrado esa función a **calcular_velocidad_reproduccion**.

- ¿Por qué Git no ha detectado ningún conflicto? ¿Qué tipo de conflicto es este?
- Además de las pruebas automatizadas, ¿qué otras prácticas ayudan a detectar este tipo de problemas cuanto antes?

Idea fundamental que se evalúa

Que Git integre dos líneas de trabajo sin conflictos no garantiza que el programa funcione: los conflictos semánticos solo se descubren con comprobaciones adicionales (pruebas, compilación, análisis estático o revisión de código), e integrar con frecuencia ayuda a detectarlos antes (3.4.3, 2.1.3, 2.1.4, 2.1.7).

Respuesta esperada

- Git combina los cambios a nivel de texto y solo avisa de un conflicto cuando no puede fusionarlos automáticamente (por ejemplo, cuando dos líneas de trabajo modifican la misma zona de un fichero de forma incompatible). Aquí cada cambio toca zonas distintas (uno renombra la función, el otro añade llamadas con el nombre antiguo), así que la fusión se completa, pero Git no comprueba que las llamadas sigan siendo válidas ni que el programa funcione: puede integrar los cambios sin detectar incompatibilidades semánticas. Es un **conflicto semántico**: una incompatibilidad lógica entre cambios que se pueden combinar como texto (3.4.3).
- Además de las pruebas, pueden ayudar a detectar estas incompatibilidades la compilación, el análisis estático y la revisión de código. Por ejemplo, una herramienta de análisis estático podría advertir de una referencia a una función inexistente. Integrar frecuentemente y mantener ramas de vida corta también ayuda a detectarlas antes, cuando los cambios son pequeños (2.1.3, 2.1.4, 2.1.7, 3.5.1).

Comp.	Puntúa si...
A (0,5)	a) Explica que Git ha podido fusionar los cambios porque no chocan como texto, pero no comprueba que el programa resultante sea correcto, y lo identifica como conflicto semántico (o incompatibilidad lógica entre los cambios).
B (0,5)	b) Propone al menos una práctica adecuada distinta de las pruebas: compilación o construcción automática tras integrar, análisis estático (<i>linters</i>), revisión de código, o integración frecuente con ramas de vida corta. Proponer solo más pruebas no puntúa.

2.2 Técnica

(1 punto)

El equipo de la aplicación de películas integra con frecuencia las tareas terminadas en `trunk`; a `main` solo se llevan las entregas, que se etiquetan. Cada tarea se desarrolla en su propia rama. Una persona recién incorporada te enseña cómo ha subido la función `delete_movie` que acaba de programar:

```
$ git switch trunk
$ git status
On branch trunk
Changes not staged for commit:
  modified:   app/models.py
  modified:   app/tests/test_unit.py
Untracked files:
  .env
  app/__pycache__/
$ git add .
$ git commit -m "Añade delete_movie para eliminar películas por título"
$ git push
! [rejected]        trunk -> trunk (fetch first)
error: failed to push some refs to 'github.com:empresa/movies.git'
$ git push --force
```

- Identifica **dos** problemas en esta secuencia y explica qué consecuencia puede tener cada uno.
- Tras el rechazo del primer `git push`, ¿qué debería haber hecho en lugar de `git push --force`?

Idea fundamental que se evalúa

Colaborar en un repositorio compartido exige seguir el modelo de uso del equipo, versionar solo lo que corresponde y no reescribir la historia compartida, integrando los cambios remotos en lugar de forzar el envío (3.3.2, 3.4.2, 3.4.5, 3.5.3).

Respuesta esperada

- Basta con dos de los tres siguientes:
 - Trabaja directamente sobre trunk:** la tarea debería desarrollarse en su propia rama (p. ej. `delete-movie`) e integrarse en `trunk` al terminar (3.5.3).
 - git add . sin revisar:** incluye en el commit `.env` (probablemente con credenciales, que al subirse quedan en el historial y hay que considerar comprometidas) y `__pycache__` (archivos generados). Deberían estar en `.gitignore` (3.3.2).
 - git push --force:** si el remoto lo acepta, sustituye la rama compartida por la local y deja fuera de su historia los commits que otras personas habían subido (3.4.5).
- No forzar el envío: traerse los cambios del remoto e integrarlos con los suyos (`git fetch` seguido de `git merge origin/trunk`; también son válidos `git pull` o un *rebase* de sus commits locales, que aún no ha compartido), resolver los conflictos si los hay, pasar las pruebas y volver a hacer `git push` (3.4.2, 3.4.5). Antes de volver a subir el cambio, convendría además sacar del commit `.env` y `__pycache__` (añadirlos después a `.gitignore` no los elimina del commit ya hecho).

Comp.	Puntúa si...
A (0,5)	a) Identifica dos problemas y explica una consecuencia de cada uno. Si solo identifica uno, o no explica las consecuencias, no puntúa.
B (0,5)	b) Propone traer e integrar los cambios remotos (fetch + merge , pull o <i>rebase</i>) antes de volver a hacer push . No se exige mencionar que hay que sacar <code>.env</code> del commit (ya se evalúa en el apartado a).

Bloque 3 · Pruebas

3.1 Conceptual

(1 punto)

Un departamento de la empresa ha conseguido, con ayuda de una herramienta basada en un LLM, que sus pruebas ejecuten el 100 % de las líneas de código. Por ello, afirma que su producto ya no tiene defectos.

- Define con precisión qué son un error, un defecto y un fallo.
- ¿Es correcta la afirmación del departamento? ¿Por qué?

Idea fundamental que se evalúa

Las pruebas muestran la presencia de fallos, no su ausencia: ejecutar todo el código (cobertura) no equivale a comprobar que es correcto, y razonarlo exige distinguir entre error, defecto y fallo (4.2.2, 4.2.3, 4.4.6).

Respuesta esperada

- Error:** equivocación humana (o del agente IA). **Defecto** (*fault*, bug): la causa que queda introducida en el software (en el código, pero también en la configuración, el diseño...) como consecuencia del error. **Fallo** (*failure*): el comportamiento incorrecto que se observa al ejecutar el sistema (4.2.3).
- No.** La cobertura solo indica qué partes del código se han *ejecutado*, no que se haya comprobado bien su comportamiento. Una prueba puede ejecutar la línea defectuosa y pasar, porque además hace falta que el estado se vuelva incorrecto y que llegue a una salida observable (modelo RIP), y que el oráculo (las aserciones) lo detecte. Además, puede haber defectos fuera del código o comportamientos que no se han implementado. Las pruebas muestran la presencia de fallos, no su ausencia (4.2.2, 4.4.6).

Comp.	Puntúa si...
A (0,5)	a) Define correctamente los tres conceptos y los distingue entre sí. Si alguna definición es incorrecta o se confunden entre sí, no puntúa.
B (0,5)	b) Rechaza la afirmación argumentando que ejecutar el código no equivale a comprobar que es correcto (o una idea equivalente: hace falta un buen oráculo, la ejecución puede no revelar el fallo...).

3.2 Técnica

(1 punto)

La aplicación de películas recomienda películas según la nota media que obtiene de un servicio externo de valoraciones. El módulo `app/ratings.py` contiene estas funciones:

```
import requests
def get_rating(title):
    """Consulta la nota media de la película en un servicio externo."""
    response = requests.get("https://api.notas-peliculas.example/rating",
                            params={"title": title})
    return response.json()["rating"]

def is_recommended(title):
    """Una película se recomienda si su nota media es 7 o superior."""
    return get_rating(title) >= 7
```

Para probar `is_recommended`, el equipo ha escrito esta prueba unitaria:

```
from app import ratings
def test_gladiator_es_recomendada():
    assert ratings.is_recommended("Gladiator")
```

- ¿Te parece una buena prueba unitaria? ¿Por qué?
- ¿Cómo la mejorarías? Explícalo con palabras; no hace falta que escribas código.

Idea fundamental que se evalúa

Una prueba unitaria debe ser rápida, determinista y centrarse en una pieza pequeña del sistema, por lo que sus dependencias externas se sustituyen por dobles de prueba (4.3.2, 4.3.6).

Respuesta esperada

- a) **No.** La prueba llama al servicio externo real, así que su resultado no depende solo de `is_recommended`: depende de la red, de que el servicio esté disponible y de la nota que tenga “Gladiator” en ese momento. Por eso es lenta, frágil y no reproducible (4.3.2, 4.3.6).
- b) Sustituir, solo durante la prueba, la llamada al servicio externo por un **doble de prueba** (*mock*, *stub* o *fake*) que devuelva una nota fija, por ejemplo 8. Así la prueba es rápida, siempre da el mismo resultado y comprueba solo la lógica de `is_recommended` (4.3.6).

Comp.	Puntúa si...
A (0,5)	a) Responde que no es una buena prueba unitaria y lo justifica por su dependencia del servicio externo real (red, disponibilidad, datos que cambian...): el resultado no depende solo de nuestro código, o la prueba es lenta, frágil o no reproducible. Basta con un argumento correcto de este tipo.
B (0,5)	b) Propone sustituir la llamada al servicio externo por un doble de prueba (<i>mock</i> , <i>stub</i> , <i>fake</i> o un cliente falso) que devuelva un valor controlado. No puntúa proponer solo cambiar de película, repetir la prueba o añadir más aserciones.

Bloque 4 · Tareas e incidencias**4.1 Conceptual**

(1 punto)

Un departamento de la empresa resuelve sus incidencias en solo dos fases: **diagnosticar** (formular hipótesis y comprobarlas con experimentos) y **reparar** (corregir el defecto y añadir una prueba de regresión). ¿Crees que deberían añadir alguna fase más? En caso afirmativo, indica el nombre de cada fase que añadirías y describe brevemente en qué consiste.

Idea fundamental que se evalúa

Depurar una incidencia con método supone reproducir el fallo, diagnosticar su causa, repararla y analizar lo aprendido para evitar que el problema se repita (5.4).

Respuesta esperada

Sí. Según el método de depuración estudiado (5.4), faltan dos fases:

- **Reproducir**, antes de diagnosticar: encontrar una forma fiable de hacer que aparezca el fallo, controlando versión, entorno y entradas. Una reproducción fiable facilita mucho diagnosticar con garantías y comprobar después que se ha corregido. Si no se consigue, se documenta lo intentado y se investiga con otras evidencias, como registros (5.4.1).
- **Analizar lo aprendido**, después de reparar: preguntarse por qué no se detectó antes (faltaba una prueba, el diseño lo permitía, la plantilla de incidencias era pobre...) y convertirlo en prevención para que no se repitan problemas similares (5.4.4).

Como la pregunta es abierta, también puntúan otras fases pertinentes y bien descritas, como registrar y priorizar la incidencia o verificar la corrección con quien la reportó (véase la rúbrica).

Comp.	Puntúa si...
A (0,5)	Propone una fase pertinente y la describe correctamente.
B (0,5)	Propone una segunda fase pertinente, distinta de la anterior, y la describe correctamente.

Cada fase distinta, pertinente y correctamente descrita obtiene 0,5 puntos, hasta un máximo de dos fases. Una misma aportación no puede puntuarse dos veces. Son pertinentes las del método estudiado (reproducir; analizar lo aprendido, con nombres equivalentes como reflexionar, retrospectiva o *postmortem*) y otras que cubran una carencia real del proceso (p. ej. registrar y priorizar la incidencia, o verificar la corrección con quien la reportó).

No lo es renombrar o subdividir las fases de diagnosticar y reparar.

4.2 Técnica

(1 punto)

Un usuario ha enviado esta incidencia sobre la aplicación de películas:

```
# El sistema no funciona

No puedo crear películas.

## Entorno
Uso Ubuntu.

## Resultado esperado
Que funcione.

## Resultado obtenido
Sale un error.
```

Reescribe la incidencia para que el equipo pueda reproducir el problema. Incluye al menos un resumen, los pasos para reproducirlo, el resultado esperado, el resultado observado y el entorno. Puedes inventar los datos concretos que necesites.

Idea fundamental que se evalúa

Una incidencia solo puede resolverse con eficacia si su informe es específico y reproducible: resumen concreto, pasos para reproducirla, resultado esperado y observado, y entorno (5.3.2, 5.3.3).

Respuesta esperada

Una posible versión:

```
# Error 500 al crear una película con el título vacío

## Pasos para reproducir
1. Iniciar sesión en la aplicación (versión 1.4.2).
2. Ir a "Nueva película".
3. Dejar el campo "Título" vacío y pulsar "Guardar".

## Resultado esperado
Se muestra un mensaje de validación indicando que el título
es obligatorio, y no se crea la película.

## Resultado observado
La aplicación muestra "Internal Server Error" (error 500).

## Entorno
Firefox 128, Ubuntu 24.04, versión 1.4.2 de la aplicación.
Se adjunta captura de pantalla.
```

Lo importante es que sea específico (no “no funciona”), reproducible (pasos concretos), que distinga esperado de observado con precisión y que aporte datos de entorno relevantes (5.3.2, 5.3.3).

Comp.	Puntúa si...
A (0,5)	Reproducible: el resumen es específico (no “no funciona”) y los pasos son concretos y permiten observar el fallo.
B (0,5)	Preciso: los resultados esperado y observado son concretos y distinguibles (no “que funcione” / “sale un error”), y el entorno incluye al menos un dato relevante además del sistema operativo (navegador o versión de la aplicación).

Bloque 5 · Aislamiento

5.1 Conceptual

(1 punto)

Una persona del equipo de operaciones nos ha escrito: “*Hemos metido la aplicación en un contenedor Docker, así que está tan aislada como si usáramos una máquina virtual*”. ¿Estás de acuerdo? Explica la diferencia principal entre un contenedor y una máquina virtual, e indica una ventaja de cada uno.

Idea fundamental que se evalúa

Contenedores y máquinas virtuales ofrecen niveles distintos de aislamiento con costes distintos: el contenedor comparte el kernel del anfitrión y la máquina virtual ejecuta su propio sistema operativo (6.2, 6.3.1).

Respuesta esperada

- **No es cierto.** Un contenedor ejecuta procesos aislados que **comparten el kernel** del sistema anfitrión (el aislamiento se apoya en mecanismos como los *namespaces* y los *cgroups*). Una máquina virtual ejecuta un **sistema operativo invitado completo** sobre un hipervisor, por lo que, por su arquitectura, ofrece en general un aislamiento más fuerte (6.2, 6.3.1).
- **Ventajas:** el contenedor es más ligero, ocupa menos y arranca más rápido; la máquina virtual ofrece un aislamiento más fuerte y permite ejecutar sistemas operativos distintos. Elegir uno u otro depende de qué necesitamos aislar y qué coste podemos asumir.

Comp.	Puntúa si...
A (0,5)	Rechaza la afirmación explicando que el contenedor comparte el kernel del anfitrión, mientras que la máquina virtual ejecuta su propio sistema operativo. Decir solo que el contenedor es “más ligero” no puntúa.
B (0,5)	Indica una ventaja correcta de <i>cada uno</i> . Si solo aporta la ventaja de uno de los dos, no puntúa.

5.2 Técnica

(1 punto)

Para desplegar la aplicación de películas, el equipo de operaciones ha escrito este **Dockerfile**. La imagen se construye de nuevo en cada despliegue y, aunque el código no cambie, dos construcciones hechas en días distintos no siempre se comportan igual.

```
1 FROM python:latest
2 WORKDIR /app
3 COPY . .
4 RUN pip install flask gunicorn
5 EXPOSE 5000
6 CMD ["gunicorn", "--bind", "0.0.0.0:5000", "app:app"]
```

- ¿Qué líneas provocan este comportamiento? ¿Por qué?
- Reescribe esas líneas para reducir esa variabilidad fijando versiones. Basta con fijar lo que aparece en el fichero.

Idea fundamental que se evalúa

Para que un entorno o artefacto pueda reconstruirse de forma reproducible, debe depender solo de entradas declaradas con versiones fijadas, no de lo que esté disponible el día de la construcción (2.1.2, 6.1.2, 6.2.1).

Respuesta esperada

- Línea 1, **FROM python:latest**: **latest** es una etiqueta móvil que, con el tiempo, apunta a versiones distintas de Python. Línea 4, **RUN pip install flask gunicorn**: instala la última versión de cada paquete disponible el día de la construcción. La imagen depende de entradas cuyas versiones exactas no están fijadas: el proceso no es hermético (6.2.1).
- Fijar versiones:

```
1 FROM python:3.12-slim
4 RUN pip install flask==3.1.2 gunicorn==23.0.0
```

También es válido declarar las dependencias con versión fija en un `requirements.txt` y usar `RUN pip install -r requirements.txt`.

Comp.	Puntúa si...
A (0,5)	a) Identifica las dos líneas (1 y 4) y explica por qué varían con el tiempo: <code>latest</code> apunta a versiones distintas y los paquetes se instalan sin versión. Si solo identifica una, no puntúa.
B (0,5)	b) Reescribe las dos líneas fijando versiones: la de la imagen base y la de los paquetes (en la propia línea o con un <code>requirements.txt</code>). Si solo corrige una, no puntúa.

Hoja de calificación

Bloque	Conceptual	Técnica	Total bloque	> 0
1 · CI/CD				☐
2 · Gestión del código fuente				☐
3 · Pruebas				☐
4 · Tareas e incidencias				☐
5 · Aislamiento				☐
Total (mínimo 5)				

Apto si ningún bloque tiene 0 puntos *y* el total es ≥ 5 .