☑ 50 Preguntas Técnicas de CI (Continuous Integration) — Respuestas Completas

---

1. ¿Qué es Continuous Integration (CI) y cuál es su objetivo?

Práctica de integrar código frecuentemente en un repositorio compartido y ejecutar validaciones automáticas.
Objetivo: **detectar errores temprano y mantener el código siempre desplegable**.

---

2. Beneficios de CI en proyectos de datos

- Previene fallas en producción
- Automatiza tests de SQL, ETL, modelos y pipelines
- Valida esquemas, linaje y dependencias
- Acelera desarrollo y revisiones
- Mejora calidad del código y del dato

---

3. Herramientas de CI

GitHub Actions, GitLab CI, Jenkins, CircleCI, Azure Pipelines, Bitbucket Pipelines, Argo Workflows.

---

4. ¿Qué es un pipeline CI y qué etapas tiene?

Secuencia automatizada:

- Checkout
- Linting
- Tests
- Build
- Validación

- Generación de artifacts
- Notificaciones

---

5. ¿Qué son los runners?

Máquinas que ejecutan jobs del pipeline. Pueden ser:

- Hosted (cloud)
- Self-hosted (on-premise o personalizados)

---

6. Trigger de CI y tipos

Evento que inicia el pipeline:

- push
- pull/merge request
- schedule
- manual
- cambios en rutas específicas

---

7. Linting

Análisis del estilo de código para asegurar consistencia y buenas prácticas.

---

8. Static code analysis

Análisis automático de vulnerabilidades, bugs y malas prácticas sin ejecutar el código.

---

9. Build en CI

Proceso para empaquetar código o compilar artefactos (contenedores, wheels, ejecutables).

---

10. Artifacts

Archivos generados por el pipeline: builds, reportes, logs, documentación.

---

11. Cache dependencies

Caché de librerías para acelerar pipelines (pip, npm, maven, dbt deps).

---

12. Pipeline reproducible

- Versionar dependencias
- Usar containers
- Pinning de versiones
- Variables inmutables

---

13. Pipeline as code

Definición del pipeline en archivos dentro del repositorio (YAML).

---

14. Archivo de configuración (.yaml)

Documento que define jobs, stages, triggers, variables y condiciones del pipeline.

---

15. Stages en CI

Agrupaciones lógicas: build, test, validate, release, deploy.

---

16. Paso pre-deploy

Validaciones antes de desplegar: tests, validación de esquema, linting, dry-run.

---

17. Ejecutar pruebas automáticas

Usar frameworks: pytest, unittest, pytest-dbt, sqlfluff, datafold.

---

18. Medir duración del pipeline

Herramientas integradas en GitHub/GitLab o dashboards externos (Datadog, Grafana).

---

19. Parallel jobs

Ejecución simultánea de tareas para acelerar el pipeline.

---

20. Optimizar pipelines lentos

- Cache
- Paralelismo
- Saltar etapas innecesarias
- Reutilizar artifacts
- Jobs condicionales

---

21. Branch protection

Reglas que impiden merges directos sin pasar tests, revisiones, o approvals.

---

22. Merge requests / Pull requests

Solicitudes para integrar cambios con revisiones, CI y comentarios.

---

23. Integrar CI con repos privados

Mediante tokens, deploy keys, o identity federation.

---

24. Manejar secretos en CI

Variables protegidas, vaults, secret managers (AWS, GCP, Azure).

---

25. Variables protegidas vs no protegidas

- **Protegidas:** solo para ramas protegidas (main, releases).
- **No protegidas:** disponibles en ramas normales.

---

26. CI badge

Icono en el README que muestra el estado del pipeline (passing/failing).

---

27. Test coverage

Porcentaje de código cubierto por tests.

---

28. Calcular cobertura

Usar herramientas: pytest-cov, coverage.py, nyc, sonarcloud.

---

29. Pipeline monolítico vs modular

- **Monolítico:** un solo archivo grande.
- **Modular:** múltiples archivos heredados, plantillas y jobs reusables.

---

30. Fan-in / Fan-out

- **Fan-out:** un job genera múltiples jobs en paralelo.
- **Fan-in:** varios jobs deben completarse para continuar.

---

31. Entornos dev/test/prod

Usar variables, job rules y matrices para ejecutar pipelines específicos por entorno.

---

32. Matrix builds

Ejecutar combinaciones:

- versiones de Python
- diferentes motores (Spark, Snowflake, BigQuery)
- sistemas operativos

---

33. Commitlint

Reglas para formatear mensajes de commit correctamente.

---

34. Semantic versioning

Formato **MAJOR.MINOR.PATCH** para versionar software.

---

35. ¿Cómo CI previene errores en producción?

- Pruebas automatizadas
- Validación de esquemas
- Linting
- Revisiones automáticas
- Detectar regresiones temprano

---

36. Dry-run

Simulación de una ejecución sin aplicar cambios.

---

37. Rollback plan

Procedimiento documentado para revertir despliegues en caso de fallo.

---

38. Versionar modelos de datos con CI

Checks automáticos de esquemas, migraciones, pruebas y validación de breaking changes.

---

39. Integrar CI con dbt

- Ejecutar `dbt deps`, `dbt test`, `dbt build`
- Validar documentación
- Ejecutar tests custom

---

40. Automated documentation generation

Generar documentación automáticamente: dbt docs, Sphinx, MkDocs, Javadoc.

---

41. Dependencias entre pipelines

Usar triggers, artifacts, o pipelines padres/hijos.

---

42. CI idempotente

Múltiples ejecuciones generan el **mismo resultado**.

---

43. Detectar regresiones automáticamente

Comparar resultados de tests, linters, métricas o snapshots previos.

---

44. Artifact retention policy

Tiempo que se conservarán artifacts antes de ser eliminados.

---

45. Ejecutar pipelines solo si cambia una carpeta

Usar reglas:

- `paths` (GitHub Actions)
- `changes` (GitLab CI)

---

46. Monorepo CI

Pipeline diseñado para múltiples proyectos dentro de un mismo repositorio.

---

47. Container-based CI

Ejecutar jobs dentro de contenedores para asegurar reproducibilidad.

---

48. On-premise CI vs cloud CI

- **On-premise:** control total, más mantenimiento.
- **Cloud:** escalable, menos administración.

---

49. Métricas para evaluar CI

- Duración

- Estabilidad
- Frecuencia de ejecuciones
- Porcentaje de fallos
- Cobertura
- Tiempo de merge

---

50. Buenas prácticas de CI para equipos de datos

- Tests de calidad de datos
- Linting SQL y Python
- Validación de esquemas
- Monitoreo de breaking changes
- Pipelines rápidos y modulares
- Uso estricto de branch protection