Guía de Ayuda — Avance 3 (Prototipo Funcional)

Subtítulo: Implementación, pruebas y entrega final

Asignatura:	Taller de Proyecto de la Especialidad
Carrera:	Técnico en Programación y Análisis de Sistemas
Docente:	Víctor M. Valderrama M.
Semestre/Año:	2025-2
Versión:	1.0 (AIEP)

Contents

1 Unidad 1: Preparación del entorno y estructura del repositorio

Antes de programar, es esencial preparar correctamente el entorno de trabajo y organizar el repositorio del proyecto. Una estructura ordenada refleja profesionalismo, facilita la evaluación y permite mantener trazabilidad entre los avances.

1.1 Importancia de la organización

En esta etapa se consolida el paso desde el análisis (Avance 2) hacia la implementación del prototipo (Avance 3). Un repositorio bien estructurado garantiza:

- Claridad para el evaluador o colaborador que revise el código.
- Separación entre documentación, diseño y código fuente.
- Facilidad para mantener y evolucionar el sistema.

Regla general: cada tipo de contenido tiene su lugar. No mezcles código con informes ni diagramas con pruebas.

1.2 Estructura base del repositorio

/src	Código fuente del prototipo (módulos, controladores, vistas o plantillas).
/database	Scripts SQL (estructura, inserciones, respaldo).
/design	Diagramas MER, casos de uso, flujos y wireframes (de Avances 1–2).
/docs	Informes PDF, plan de pruebas ejecutado y evidencias.
/tests	Pruebas unitarias o funcionales y sus resultados.
README.md	Descripción del proyecto y guía de ejecución.
.gitignore	Archivos a excluir del control de versiones (por ejemplo, archivos temporales o de configuración privada).

Comandos de inicio (Windows CMD / PowerShell):

```
cd C:\AIEP\TallerProyectoEspecialidad
mkdir src database design docs tests
(echo # Proyecto Avance 3) > README.md
```

Errores a evitar:

- Poner informes o capturas dentro de `/src`.
- No usar `.gitignore`.
- Dejar `README.md` vacío o sin instrucciones.

1.3 Flujo básico con Git y GitHub

Git es el sistema de control de versiones que te permite registrar cada cambio del proyecto, volver atrás si algo falla y trabajar de forma ordenada.

Conceptos esenciales:

- **Commit**[1] — registra un cambio en la historia del proyecto.
- **Branch**[2] — permite crear versiones paralelas del proyecto.
- **Merge**[3] — combina los cambios de diferentes ramas.
- **Tag / Release**[4] — marca y publica una versión estable del proyecto.

Buenas prácticas con Git:

- Realiza *commits* pequeños y frecuentes con mensajes claros.
- No borres el historial ni uses `--force` salvo que sepas lo que haces.
- Usa etiquetas (`tags`) para marcar hitos o entregas oficiales.

Ejemplo de flujo de trabajo básico:

```
git add .
git commit -m "Implementa estructura inicial del proyecto"
git push origin main
git tag -a v3.0 -m "Versin Avance 3"
git push origin v3.0
```

Uso del tablero Kanban de GitHub: Permite organizar tareas en columnas ("Por hacer", "En progreso", "Hecho"). Cada tarjeta debe corresponder a una actividad concreta del desarrollo.

1.4 Actividad práctica

Actividad de aula:

1. Crear la estructura del proyecto en tu carpeta local siguiendo el modelo anterior.
2. Inicializar Git (`git init`) y subir el proyecto a GitHub.
3. Verificar que el repositorio contiene las carpetas vacías y el archivo `README.md`.

[1] *Commit:* acción de guardar un conjunto de cambios en el repositorio. En español, "confirmar" o "registrar" los cambios.
[2] *Branch:* una "rama" de desarrollo donde se puede trabajar sin afectar el código principal.
[3] *Merge:* acción de unir una rama secundaria con la principal. En español, "fusionar".
[4] *Tag:* etiqueta que marca una versión específica. *Release:* publicación formal de una versión, normalmente con notas o adjuntos.

4. Crear un primer `commit` con mensaje claro y subirlo a la rama principal.

Errores comunes:

- Subir un único `commit` al final del proyecto.
- No agregar mensajes descriptivos (“update”, “final” no sirven).
- Mezclar archivos de pruebas con documentos en el mismo directorio.

Organizar tu entorno y repositorio no es un requisito burocrático: es el primer paso para demostrar tu nivel profesional como desarrollador.

“El orden en el código refleja el orden en tu pensamiento técnico.”

2 Unidad 2: Implementación del prototipo funcional

Esta unidad enseña cómo transformar los requerimientos definidos en el Avance 2 en código funcional y probado. Se busca que el estudiante aplique metodologías modernas, buenas prácticas de desarrollo y control de versiones para entregar un módulo sólido, coherente con el diseño del sistema.

2.1 De los requerimientos al código

El paso del diseño al desarrollo implica leer los requerimientos funcionales (RF) y no funcionales (RNF) y decidir qué parte del sistema implementar primero. El criterio recomendado es iniciar por el módulo más representativo o crítico, generalmente un CRUD principal (**Create, Read, Update, Delete**[5]).

Ejemplo de trazabilidad:

Requerimiento	Módulo del sistema	Funcionalidad esperada
RF-01	Gestión de clientes	CRUD de registro, edición y eliminación de clientes.
RF-03	Reportes de ventas	Consulta filtrada por rango de fechas.
RNF-02	Rendimiento	Respuesta menor a 2 segundos por consulta.

2.2 Buenas prácticas de programación

Desarrollar un sistema funcional no es solo "que funcione": también debe ser limpio, mantenible y comprensible. A continuación se resumen principios universales del desarrollo moderno.

- **Clean Code**[6] — escribir código claro, sin repeticiones y fácil de entender.
- **KISS**[7] — usa la solución más simple posible.
- **DRY**[8] — reutiliza funciones o componentes en lugar de copiar y pegar.
- **SRP**[9] — cada función o clase debe tener una única responsabilidad.

[5] *Create, Read, Update, Delete:* operaciones básicas de gestión de datos. En español, crear, leer, actualizar y eliminar.
[6] *Clean Code:* código limpio y legible, con funciones cortas y nombres claros. En español, "código limpio".
[7] *KISS: Keep It Simple, Stupid.* En español, "Mantenlo simple estúpido". Evita complejidad innecesaria.
[8] *DRY: Don't Repeat Yourself.* En español, "No te repitas". No dupliques código o lógica.
[9] *SRP: Single Responsibility Principle.* En español, "Principio de responsabilidad única". Cada módulo debe tener un solo propósito.

Ejemplo de código limpio (pseudocódigo):

```
# Funcin que crea un nuevo cliente
def crear_cliente(nombre, telefono, email):
    if not validar_email(email):
        return "Error: correo invlido"
    insertar_en_db(nombre, telefono, email)
    return "Cliente creado correctamente"
```

Ejemplo incorrecto:

- Función con múltiples tareas (validar, insertar, reportar) sin separación.
- Variables con nombres genéricos como `x1`, `dato1`, `res`.
- Repetición de código similar en varias partes del proyecto.

2.3 Gestión del código con Git

El uso disciplinado de Git y GitHub permite mantener el control de los cambios y facilita trabajar de forma profesional incluso siendo un único programador.

Flujo recomendado:

1. Crear una rama de desarrollo (`feature-crud-clientes`).
2. Realizar commits pequeños y descriptivos.
3. Fusionar la rama al finalizar el módulo (`merge` hacia `main`).
4. Etiquetar una versión estable cuando el módulo funcione correctamente.

```
# Crear rama, agregar cambios y etiquetar versin
git checkout -b feature-crud-clientes
git add .
git commit -m "Agrega CRUD bsico de clientes"
git push origin feature-crud-clientes
git checkout main
git merge feature-crud-clientes
git tag -a v3.1 -m "Versin CRUD Clientes completo"
git push origin v3.1
```

Aclaraciones de términos:

- **Commit** — registrar un cambio en la historia del repositorio.
- **Branch** — una "rama" donde se trabaja sin afectar la versión principal.
- **Merge** — fusionar una rama con otra.
- **Tag** — marcar una versión específica.
- **Release** — publicación formal de una versión con notas y adjuntos.

2.4 Validaciones y retroalimentación

Cada formulario o proceso debe incluir validaciones adecuadas:

- Campos obligatorios y formatos correctos.
- Mensajes claros de error o éxito.
- Retroalimentación visual (por ejemplo, alertas o resaltado de campos).

Ejemplo de mensaje correcto: *"Cliente agregado correctamente"* **Ejemplo de error claro:** *"El correo electrónico ingresado no es válido."*

Evita:

- Mensajes ambiguos como "Error general" o "No funciona".
- Formularios que no validan antes de enviar.
- Dejar que los errores del sistema se muestren al usuario final.

2.5 Actividad práctica

Ejercicio:

1. Implementa un CRUD funcional (puede ser de clientes, productos o alumnos).
2. Valida todos los campos y muestra mensajes de confirmación o error.
3. Registra tus cambios con commits frecuentes y descriptivos.
4. Sube la rama a GitHub y verifica que el tablero Kanban refleje las tareas completadas.

La implementación del prototipo no es solo escribir código: es aplicar orden, claridad y responsabilidad técnica. Cada línea del código debe reflejar lo planificado en el diseño.

"Programa como si la persona que mantendrá tu código fuera un asesino que sabe dónde vives."

(Frase popular en desarrollo de software sobre la importancia de escribir código claro y mantenible.)

3 Unidad 3: Pruebas y control de calidad

El desarrollo de software no termina cuando el código "funciona", sino cuando ha sido verificado. Las pruebas son el medio para garantizar la calidad técnica y el cumplimiento de los requerimientos. Esta unidad enseña a planificar, ejecutar y documentar pruebas de manera profesional.

3.1 Importancia de las pruebas

Probar un sistema no es buscar errores al azar, sino demostrar que el software cumple con lo planificado. Una prueba bien diseñada confirma que un requerimiento está implementado correctamente.

- Verificar que el sistema cumple los **RF** (requerimientos funcionales).
- Asegurar que los **RNF** (rendimiento, usabilidad, seguridad) también se cumplen.
- Reducir errores en versiones futuras y facilitar el mantenimiento.

Tipos comunes de pruebas:

- **Unitarias**[a].
- **Integración** — verifica que varios módulos trabajen juntos.
- **Funcionales** — prueban procesos completos (por ejemplo, todo un CRUD).
- **Exploratorias** — pruebas libres para detectar fallos no previstos.

[a] *Unit test:* prueba que verifica una función o módulo individual. En español, "prueba unitaria".

3.2 Plan de pruebas

Un **plan de pruebas** describe qué se probará, cómo y con qué datos. Cada caso de prueba debe derivarse de un requerimiento y tener un resultado esperado claro.

Estructura de un plan de pruebas:

Caso	Entrada	Resultado esperado / obtenido	Estado
CRUD-01	Nombre, teléfono y email válidos.	Registro exitoso y mensaje "Cliente creado correctamente".	OK
CRUD-02	Email vacío.	Error mostrado: "El correo electrónico es obligatorio."	OK
CRUD-03	ID inexistente en eliminar cliente.	Sistema debe advertir: "Cliente no encontrado."	OK

Definition of Done[a]: Un requerimiento está completado solo si ha sido probado y documentado con evidencia.

[a] *Definition of Done:* criterios que definen cuándo una tarea está realmente terminada. En español, "definición de hecho".

3.3 Evidencias y registro

Cada prueba debe dejar una evidencia:

- Capturas de pantalla, videos o logs del sistema.
- Resultados exportados a PDF o guardados en la carpeta `/docs/tests`.
- Listado de errores encontrados y su corrección.

Ejemplo de evidencia (texto breve dentro del informe):

- Caso CRUD-02 ejecutado el 05/09/2025.
- Resultado esperado: validación de email vacío.
- Resultado obtenido: mensaje "El correo electrónico es obligatorio".
- Estado: OK. Captura en Anexo A.

Errores comunes:

- No guardar las capturas o registros de las pruebas.
- Probar solo "lo que funciona".
- No comparar resultados esperados vs. obtenidos.

3.4 Gestión de errores en GitHub

Usa la sección **Issues** de tu repositorio para registrar errores detectados durante las pruebas. Cada *issue* debe tener título claro, descripción, pasos para reproducirlo y estado.

```
# Ejemplo de creacin manual de un Issue
Titulo: "Error en validacin de correo"
Descripcion:
1. Ingresar formulario con email vaco
2. Sistema no muestra mensaje de error
Resultado esperado: advertencia visible
Estado: Abierto
```

Cuando el error se corrige, comenta en el issue con la solución y márcalo como "Closed". Esto deja evidencia de la trazabilidad entre prueba → error → corrección.

3.5 Automatización básica y trazabilidad

Si el lenguaje lo permite, puedes crear pruebas automáticas para verificar funciones críticas. Aunque opcional, esto demuestra profesionalismo y dominio técnico.

```
# Ejemplo bsico de prueba automatizada (Python)
def test_email_valido():
    assert validar_email("usuario@correo.com") == True
```

Trazabilidad:

- Cada RF debe vincularse con al menos un caso de prueba.
- Cada caso de prueba debe tener evidencia en el informe.
- Cada corrección debe registrarse en GitHub como *issue* cerrado.

3.6 Actividad práctica

Tarea práctica:

1. Crea un plan de pruebas con al menos 3 casos derivados de tus RF.
2. Ejecuta las pruebas y guarda las evidencias (capturas o logs).
3. Registra los resultados en un PDF dentro de `/docs/tests`.
4. Sube los cambios y adjunta el PDF al repositorio.

Probar no es perder tiempo: es asegurar calidad. Un sistema sin pruebas es un sistema sin confianza.

"No basta con que funcione, debe funcionar bien y poder demostrarse."

(Principio esencial del aseguramiento de calidad en software.)

4 Unidad 4: Documentación técnica y evidencias de ejecución

Objetivo: consolidar una **documentación clara, breve y verificable** del prototipo; dejar instrucciones de instalación/uso, evidencias de funcionamiento y versionado final. Un software sin documentación *no es transferible ni evaluable.*

4.1 Documentación técnica del código (`/src`)

La documentación técnica debe explicar *cómo está organizado el código* y *cómo se ejecuta*, sin repetir el código fuente.

- Estructura interna de `/src`: módulos, controladores, vistas/plantillas, utilidades.
- Comentarios breves en cabeceras de archivo: propósito, entradas/salidas, autor.
- Convenciones de nombres: consistentes, en español neutro o inglés técnico (pero coherentes).

Ejemplo de cabecera técnica (pseudocódigo):

```
/**
 * Mdulo: clientes_controller.py
 * Propsito: CRUD de Clientes (crear, leer, actualizar, eliminar)
 * Entradas: request (JSON/form), conexin a BD
 * Salidas: JSON con resultado (xito/error)
 * Autor: Nombre Apellido - 2025-2
 */
```

Evitar: archivos sin cabecera, nombres genéricos (`utils2.py`, `final_ok.py`), comentarios extensos que repiten el código o documentación desactualizada.

4.2 Guía de instalación y uso (`README.md`)

El **README**[10] es la puerta de entrada para ejecutar el prototipo desde cero.

- Descripción breve del sistema y su alcance.
- Requisitos: versión de lenguaje (p. ej., Python/Node/.NET), gestor de paquetes, motor de BD.
- Pasos de instalación: clonado, dependencias, variables de entorno, scripts SQL.
- Pasos de ejecución y pruebas básicas (comandos).
- Credenciales de prueba (si aplica).

[10] *README:* archivo "léeme" con instrucciones para instalar y ejecutar el sistema. En español: *léame.*

Plantilla mínima de `README.md`:

```
# Taller Proyecto - Avance 3 (Prototipo)
## Descripcin
CRUD de Clientes para Taller Automotriz. Coherente con RF/RNF aprobados.

## Requisitos
- Python 3.11 / SQL Server 2019
- pip, virtualenv

## Instalacin
git clone https://github.com/usuario/proyecto.git
cd proyecto
python -m venv .venv && .venv\Scripts\activate
pip install -r requirements.txt
# Base de datos
sqlcmd -S . -E -i database\create_schema.sql
sqlcmd -S . -E -i database\seed_data.sql

## Ejecucin
python src\app.py
# Abrir http://localhost:5000

## Pruebas rpidas
# Crear cliente de ejemplo y verificar listado (capturas en /docs/tests)
```

4.3 Evidencias técnicas (`/docs`)

Las evidencias *demuestran* que el sistema funciona según lo planificado (Unidad 3).

- Capturas de cada operación clave (alta, baja, modificación, consulta).
- PDF con plan de pruebas ejecutado y resultados (OK/FAIL), vinculado a RF/RNF.
- Organización sugerida: `/docs/tests` (pruebas), `/docs/screens` (capturas).

Caso	Entrada	Esperado / Obtenido (evidencia)	Estado
CRUD-01	Cliente válido	Alta exitosa / Captura `screen_alta.png`	OK
CRUD-02	Email vacío	Mensaje “Correo obligatorio” / Captura `screen_val.png`	OK
CRUD-03	Eliminar ID inexistente	Advertencia “No encontrado” / Log `test_del.txt`	OK

Errores comunes: no adjuntar capturas, no indicar fecha/versión de la evidencia, imágenes borrosas o ilegibles, mezclar evidencias dentro de `/src`.

4.4 Registro de cambios (CHANGELOG.md)

El **CHANGELOG**[11] facilita entender la evolución del sistema.

- Formato sugerido: Encabezado por versión y fecha; secciones `Added/Changed/Fixed`.
- Mantener entradas breves, accionables y en orden cronológico.

Ejemplo de `CHANGELOG.md`:

```
## [v3.0] - 2025-10-29
### Added
- CRUD completo de Clientes (alta, baja, mod, consulta)
- Validaciones de email y campos obligatorios
### Changed
- README con pasos de instalacin/ejecucin
### Fixed
- Mensajes de error ms claros en formularios
```

4.5 Versionado final: etiqueta y publicación

Marcar la entrega con una **etiqueta** (`tag`) y crear la **release**[12] en GitHub:

- `v3.0` = Avance 3 (prototipo funcional).
- Adjuntar PDF de evidencias y, si corresponde, un ZIP del prototipo.
- Notas de versión: breve, con vínculo a `/docs`.

```
# Etiquetar y publicar la entrega del Avance 3
git tag -a v3.0 -m "Entrega Avance 3 (Prototipo funcional)"
git push origin v3.0

# Crear Release en GitHub (interfaz web):
# - Ttulo: v3.0 - Avance 3
# - Notas: resumen de mdulos implementados y pruebas
# - Adjuntar: PDF de evidencias /docs/avance3_evidencias.pdf
```

Checklist de cierre (mínimo):

- `README.md` actualizado (instalación, ejecución, credenciales de prueba).
- `CHANGELOG.md` con resumen de v3.0.
- PDF de evidencias en `/docs/tests` y referenciado en la release.
- Etiqueta `v3.0` y release publicada en GitHub.

[11] *Changelog:* "registro de cambios" que documenta qué se modificó en cada versión. En español: *registro de cambios*.

[12] *Release:* publicación formal de una versión etiquetada que incluye notas y adjuntos. En español: *publicación/versión liberada*.

4.6 Actividad práctica

Tarea:

1. Completa tu `README.md` con requisitos, instalación y ejecución (ver plantilla).
2. Genera `CHANGELOG.md` con entradas `Added/Changed/Fixed`.
3. Exporta a PDF tu plan de pruebas ejecutado con capturas (`/docs/tests`).
4. Crea la etiqueta `v3.0` y publica la release con notas y adjuntos.

La documentación y las evidencias convierten tu prototipo en un entregable **auditable y profesional**. El código muestra *cómo*; la documentación explica *por qué* y *cómo reproducirlo*.

"Si no está documentado, no existe."

5 Unidad 5: Presentación y defensa del prototipo

Objetivo: preparar al estudiante para exponer su trabajo de manera profesional, demostrando el funcionamiento del prototipo, las decisiones técnicas tomadas y la coherencia con los requerimientos definidos.

5.1 Estructura de la presentación

Toda defensa técnica debe tener una estructura lógica que permita comunicar con claridad los resultados:

1. **Introducción:** resumen del problema, los objetivos SMART y el alcance del sistema.
2. **Descripción técnica:** principales módulos, arquitectura y herramientas empleadas.
3. **Demostración:** ejecución real del sistema (CRUD, validaciones, reportes, etc.).
4. **Cierre:** aprendizajes, limitaciones y proyecciones futuras.

Recomendación: prepara una presentación de 10 a 12 minutos con apoyo visual (diapositivas o video corto). Evita sobrecargar con texto: usa capturas y diagramas.

5.2 Recomendaciones técnicas

- Ejecuta el sistema localmente, sin depender de conexión externa.
- Carga previamente datos de prueba coherentes con los requerimientos.
- Ten preparado un respaldo de video o capturas por si ocurre un error durante la demostración.
- Verifica que las rutas, puertos y permisos estén configurados correctamente antes de la exposición.

Ejemplo de preparación correcta:

- Ensayar el CRUD completo antes de la presentación.
- Mostrar el tablero Kanban de GitHub y las etiquetas de commits.
- Explicar brevemente la estructura de carpetas del repositorio.

Errores comunes:

- Confiar en la conexión a internet para ejecutar dependencias externas.
- No tener datos de prueba o fallar al ingresar credenciales.
- Mostrar pantallas o código sin explicar su propósito.

5.3 Comunicación efectiva

Una presentación técnica es también una demostración de competencias profesionales. Hablar con claridad y seguridad refuerza la confianza en tu trabajo.

- Evita leer texto: explica lo que hiciste y por qué lo hiciste así.
- Utiliza términos técnicos con precisión (por ejemplo, "módulo", "función", "proceso").
- Refuerza la coherencia con los RF/RNF y las fases previas del taller.

Consejo: muestra cómo tu prototipo responde al problema inicial. El jurado o docente valora más la lógica y el proceso que la apariencia visual.

5.4 Criterios de evaluación de la defensa

Criterio	Descripción	Ponderación
Coherencia técnica	El prototipo corresponde al diseño aprobado y funciona correctamente.	30%
Claridad comunicacional	El estudiante explica con seguridad, usa lenguaje técnico adecuado y mantiene un hilo lógico.	25%
Demostración funcional	El sistema se ejecuta correctamente, mostrando operaciones clave (CRUD, validaciones, reportes).	25%
Recursos de apoyo	Presentación clara, repositorio ordenado y uso correcto de GitHub.	10%
Reflexión y cierre	Reconoce limitaciones, aprendizajes y posibles mejoras.	10%

5.5 Actividad práctica

Tarea:

1. Ensaya tu presentación con cronómetro (10 minutos máximo).
2. Verifica el orden de los temas: problema → solución → demostración → cierre.
3. Prepara una copia local del prototipo y los datos de prueba.
4. Crea una carpeta `/docs/presentacion` con tu archivo de diapositivas o video corto.

Tu presentación no solo muestra un sistema, sino también tu identidad profesional. La claridad, el orden y la seguridad reflejan tus competencias técnicas y tu capacidad para trabajar en entornos reales.

"No solo se evalúa el código, sino cómo comunicas el valor de lo que hiciste."

6 Unidad 6: Reflexión profesional y portafolio

Objetivo: cerrar el Avance 3 con una reflexión crítica del proceso y convertir el repositorio en una evidencia **presentable** para prácticas o empleo. El foco está en: (1) autoevaluación honesta; (2) curaduría del repositorio como *portafolio*[a].

[a] *Portfolio / Portafolio:* colección seleccionada de trabajos que evidencian capacidades. En español: *portafolio*.

6.1 Autoevaluación guiada

- **Logros**: ¿qué funcionalidades clave implementaste con calidad?
- **Dificultades**: ¿qué bloqueó el avance? ¿cómo lo resolviste?
- **Aprendizajes**: 3 cosas técnicas y 1 de gestión personal (tiempo/hábitos).
- **Próximo paso**: si tuvieras 2 semanas más, ¿qué mejorarías primero y por qué?

Redacta una reflexión de **media página** y súbela como `/docs/reflexion_avance3.pdf`. Usa un tono técnico y concreto (evita frases genéricas).

6.2 El repositorio como evidencia profesional

Convierte tu repo en una pieza de presentación clara para revisores:

Elemento	Qué debe mostrar
`README.md`	Descripción breve, requisitos, instalación, ejecución, credenciales de prueba, capturas clave.
`/docs`	Informes, plan de pruebas ejecutado con evidencias, guía de uso si corresponde.
`/design`	Diagramas (MER, flujos) coherentes con lo implementado.
`/src`	Código limpio, modular, con comentarios breves y consistentes.
`Tags / Releases`	`v3.0` publicada con notas y PDF de evidencias adjunto.
`Issues / Projects`	Registro de bugs y tablero Kanban con tareas cerradas.

Buena práctica de curaduría:

- Agrega una sección "*Resumen para evaluadores*" al README con 5 bullets (alcance, módulos, pruebas, cómo correr, versión).
- Incluye 2–3 capturas representativas (listar, crear, validación).

6.3 Presentación breve del repositorio (portafolio)

Estructura sugerida para adjuntar en postulaciones:

- **Título del proyecto** + 1–2 líneas de propósito.

- **Tecnologías**: lenguaje, framework, BD.
- **Logros**: 3 bullets medibles (p. ej., "CRUD completo con validación y plan de pruebas con 6 casos OK").
- **Enlace al repo** y a la **release** `v3.0`.

6.4 Extra opcional: GitHub Pages / demo estática

Si tu prototipo tiene parte web, puedes publicar una demo estática o documentación:

```
# Rama para GitHub Pages (opcional, si es contenido esttico)
git checkout -b gh-pages
# colocar HTML esttico o documentacin generada
git add . && git commit -m "Publica demo/documentacin en gh-pages"
git push origin gh-pages
```

No es obligatorio. Úsalo solo si tu proyecto lo permite (páginas estáticas o documentación).

6.5 Checklist final de portafolio

- `README.md` claro y actualizado (con capturas).
- `CHANGELOG.md` con `v3.0` y notas de versión.
- Evidencias de pruebas en `/docs/tests` y PDF resumen.
- Diagrams en `/design` alineados con lo implementado.
- Estructura limpia: nada sensible subido, ni archivos temporales.

6.6 Actividad

Tarea:

1. Completa la **reflexión** y súbela a `/docs/reflexion_avance3.pdf`.
2. Mejora el **README** con la sección "Resumen para evaluadores" y 3 capturas clave.
3. Verifica **release** `v3.0` con notas + PDF de evidencias adjunto.

Tu repo es tu carta de presentación: debe permitir a un tercero **instalar, ejecutar y verificar** tu trabajo en minutos. Lo que no se puede reproducir, no se puede evaluar.

"Desarrollar es aprender; documentar y curar tu trabajo es profesionalizarte."