🛠️ PROTOCOLO MAESTRO: ESTRATEGIA GITHUB Y GESTIÓN DE ACTIVOS

Organización: Value Strategy Consulting **Versión:** 2026.1 - Integridad, Colaboración y Control de IA

Este documento establece la normativa técnica obligatoria para evitar la pérdida de datos y conflictos de sincronización. El objetivo es garantizar la trazabilidad del código y establecer un gobierno estricto sobre las intervenciones de Inteligencia Artificial.

---

ontenido.

I. Categorías de Acción (Temática)

- **Azul (🏗️/📁)**: Arquitectura de sistemas, rutas locales puras y estructura de directorios.
- **Verde (🚀/🤝)**: Onboarding, comandos de inicio, flujo multiusuario y éxito de sincronización.
- **Amarillo/Naranja (🤖)**: Protocolos de IA, advertencias de alcance (Scope) y validaciones preventivas.

- **Púrpura (🌿)**: Gestión avanzada de ramas (Branches) y versionado independiente.
- **Naranja (📦)**: Almacenamiento externo y respaldos "fríos" manuales en Drive.
- **Rojo (🚨)**: Comandos de emergencia, recuperación de datos y mitigación de errores críticos.

II. Estados de Git (Visualización en Editor)

Para la auditoría de cambios antes de cada commit, se utiliza la nomenclatura oficial de Git visible en el explorador de archivos:

- **Verde [U] (Untracked)**: **Archivo Nuevo.** El archivo ha sido creado pero aún no está bajo el control de versiones de Git.
- **Naranja/Amarillo [M] (Modified)**: **Archivo Modificado.** El archivo ya existía pero ha sufrido cambios en su contenido.
- **Rojo [D] (Deleted)**: **Archivo Eliminado.** El archivo ha sido removido del proyecto pero el cambio aún no se ha confirmado.

🏗️ 1. ARQUITECTURA DE ENTORNOS Y SEGURIDAD

Para evitar la pérdida de información y conflictos de sincronización, se define una separación física estricta de los entornos. Se prohíbe el uso de carpetas sincronizadas automáticamente por Drive para el desarrollo activo.

Nivel	Entorno	Ubicación / Método	Propósito Funcional
01	**Local (PC)**	Ruta local pura (ej.`C:/Proyectos/`).	Trabajo activo y ejecución fuera de Drive.
02	**GitHub**	Value Strategy Consulting.	Control de versiones y respaldo histórico.
03	**Drive**	Solo archivos .zip manuales de hitos.	Respaldos finales de versiones terminadas.

🚀 2. ONBOARDING: INICIO DE PROYECTO

Todo nuevo desarrollo o incorporación debe iniciar clonando el activo desde la organización oficial para asegurar la vinculación correcta.

- **Comando de Clonación**:

```
git clone [https://github.com/ValueStrategyConsulting/nombre-del-proyecto.git]
(https://github.com/ValueStrategyConsulting/nombre-del-proyecto.git)
```

- **Configuración Inicial**: Es obligatorio configurar un archivo `.gitignore` para excluir archivos temporales y datos sensibles.

🤖 3. PROTOCOLO DE ALCANCE PARA IA (SCOPE CONTROL)

Para evitar que las herramientas de IA modifiquen archivos externos no esenciales, se aplica un perímetro de contención basado en ramas independientes.

1. **Definición de Alcance**: Antes de procesar una tarea, especifique a la IA la ruta exacta de los archivos permitidos; prohíba la edición de configuraciones globales.
2. **Aislamiento en Ramas**: Cualquier código generado por IA debe implementarse en una **Rama Independiente** para no comprometer la estabilidad.
3. **Auditoría de IA**: Es obligatorio ejecutar `git status` tras la intervención de una IA para validar que el alcance de la modificación fue respetado.

🤝 4. FLUJO MULTIUSUARIO Y COLABORACIÓN

El trabajo en equipo sobre un mismo archivo requiere una coordinación basada en la prevención de conflictos y commits frecuentes.

- **Sincronización Preventiva (Pull)**: Antes de iniciar cualquier edición, descargue siempre los cambios más recientes del repositorio central.

```
git pull origin [nombre-de-rama]
```

- **Gestión de Conflictos**: Si Git detecta cambios en la misma línea, el usuario debe elegir la versión final manualmente antes de confirmar el cambio.
- **Commits Atómicos**: Guarde cambios pequeños y frecuentes con mensajes descriptivos por cada funcionalidad.

🌿 5. GESTIÓN DE RAMAS (BRANCHES)

Las ramas permiten probar modificaciones arriesgadas o experimentos de IA sin afectar el código funcional.

- **Crear y Cambiar a Rama**: `git checkout -b nombre-rama`.
- **Regresar a Rama Principal**: `git checkout main`.
- **Fusión de Trabajo**: Una vez que la rama independiente es estable, se integra a la principal mediante un proceso de unión (merge).

🔄 6. CICLO DE TRABAJO SEGURO (COMANDOS)

Este ciclo garantiza que el código esté respaldado y documentado al final de cada sesión.

1. **Verificar Cambios**: Ejecute `git status` para revisar archivos modificados.
2. **Preparar**: Añada los archivos modificados al área de preparación (`git add .`).
3. **Confirmar**: Cree un commit pequeño y descriptivo.

```
git commit -m "feat: descripción clara del cambio"
```

4. **Sincronizar**: Realice `git push` al finalizar el día para asegurar el código en la nube.

🚨 7. RECUPERACIÓN ANTE ERRORES (PANIC BUTTON)

Mecanismos para restaurar el proyecto ante ediciones accidentales o fallos críticos.

- **Restaurar Archivo Específico**: Recupera un archivo a su estado funcional previo.

```
git checkout <nombre_archivo>
```

- **Reinicio de Estado**: Vuelve el repositorio completo a un estado funcional anterior.

```
git reset --hard
```

✅ CHECKLIST DE SUPERVIVENCIA TÉCNICA

- ☐ **Ubicación**: El proyecto reside en una ruta local pura fuera de Drive
- ☐ **Filtro**: Archivo `.gitignore` configurado para excluir basura y sensibles.
- ☐ **Orden**: Commits pequeños y descriptivos realizados por cada cambio.
- ☐ **Ramas**: Uso de ramas independientes para pruebas arriesgadas o de IA.
- ☐ **Sincronización**: `git push` realizado al finalizar cada jornada laboral.
- ☐ **Hitos**: Respaldos manuales en `.zip` en Drive solo para versiones finales.
- ☐ **Validación**: Verificación del estado con `git status` antes de cada commit[cite: 22].