MÓDULO 04: GIT PROFESIONAL

Más Allá del Commit: Conventional Commits, Hooks y Branching

Guía MLOps v5.0: Senior Edition | DuqueOM | Noviembre 2025

MÓDULO 04: Git Profesional

Control de Versiones que Impresiona en Code Review

"Un historial de Git limpio es la documentación que nunca miente."

Duración	Teoría	Práctica
4-5 horas	25%	75%

Lo Que Lograrás en Este Módulo

1. **Escribir** commits que cuentan una historia clara
2. **Configurar** pre-commit hooks que previenen errores
3. **Aplicar** estrategias de branching profesionales
4. **Dominar** comandos avanzados (rebase, cherry-pick, bisect)

4.1 Conventional Commits: El Estándar de Industria

¿Por Qué Importa el Formato del Commit?

```
╔═══════════════════════════════════════════════════════════════╗
║                  × HISTORIAL TÍPICO (CAÓTICO)                 ║
╠═══════════════════════════════════════════════════════════════╣
║                                                               ║
║   * fix                                                       ║
║   * wip                                                       ║
║   * más cambios                                               ║
║   * asdfgh                                                    ║
║   * funcionaaaa                                               ║
║   * ahora sí                                                  ║
║   * merge conflict resuelto                                   ║
║   * updates                                                   ║
║                                                               ║
║   PROBLEMAS:                                                  ║
║   • Imposible saber qué cambió sin leer el código             ║
║   • No puedes generar changelog automático                    ║
║   • git bisect es inútil                                      ║
║   • Code review es un infierno                                ║
╚═══════════════════════════════════════════════════════════════╝

╔═══════════════════════════════════════════════════════════════╗
║                HISTORIAL PROFESIONAL (CONVENTIONAL)            ║
╠═══════════════════════════════════════════════════════════════╣
║                                                               ║
║   * feat(api): add /predict endpoint with batch support       ║
║   * fix(training): handle NaN values in CreditScore column    ║
║   * test(pipeline): add integration tests for full pipeline   ║
║   * docs(readme): update installation instructions            ║
║   * refactor(config): migrate from dict to Pydantic models    ║
║   * ci(actions): add caching for pip dependencies             ║
║   * perf(inference): reduce latency from 150ms to 45ms        ║
║                                                               ║
║   BENEFICIOS:                                                 ║
║   • Changelog generado automáticamente                        ║
║   • Semantic versioning automático                            ║
║   • git bisect encuentra bugs rápidamente                     ║
║   • Code review enfocado                                      ║
╚═══════════════════════════════════════════════════════════════╝
```

Anatomía de un Conventional Commit

```
<type>(<scope>): <description>

[optional body]

[optional footer(s)]
```

Tipos Permitidos

Tipo	Cuándo Usar	Ejemplo
`feat`	Nueva funcionalidad	`feat(api): add batch prediction endpoint`
`fix`	Corrección de bug	`fix(training): handle missing values in Age`
`docs`	Solo documentación	`docs(readme): add API usage examples`
`style`	Formato (no afecta lógica)	`style: apply ruff formatting`
`refactor`	Refactor sin cambio funcional	`refactor(config): use Pydantic BaseSettings`
`test`	Añadir o corregir tests	`test(inference): add unit tests for predictor`
`perf`	Mejora de performance	`perf(pipeline): cache preprocessor transformations`
`ci`	Cambios en CI/CD	`ci(actions): add Python 3.12 to test matrix`
`build`	Cambios en build/deps	`build(deps): upgrade scikit-learn to 1.4.0`
`chore`	Mantenimiento general	`chore: update .gitignore`

Scopes Comunes en MLOps

```
# Por componente
feat(training): ...
feat(inference): ...
feat(api): ...
feat(config): ...
feat(data): ...

# Por capa
feat(model): ...
feat(features): ...
feat(pipeline): ...

# Por herramienta
ci(actions): ...
ci(docker): ...
ci(dvc): ...
```

Ejemplos Completos

```
# Simple
git commit -m "feat(api): add health check endpoint"

# Con body explicativo
git commit -m "fix(training): handle class imbalance in target variable

The training was failing silently when class ratio exceeded 1:10.
Added class_weight='balanced' to RandomForestClassifier.

Fixes #123"

# Breaking change (incrementa MAJOR version)
git commit -m "feat(api)!: change response format to include confidence scores

BREAKING CHANGE: The /predict response now returns an object instead of
a single float. Clients must update to handle the new format:
{\"probability\": 0.85, \"confidence\": 0.92, \"prediction\": \"churn\"}"
```

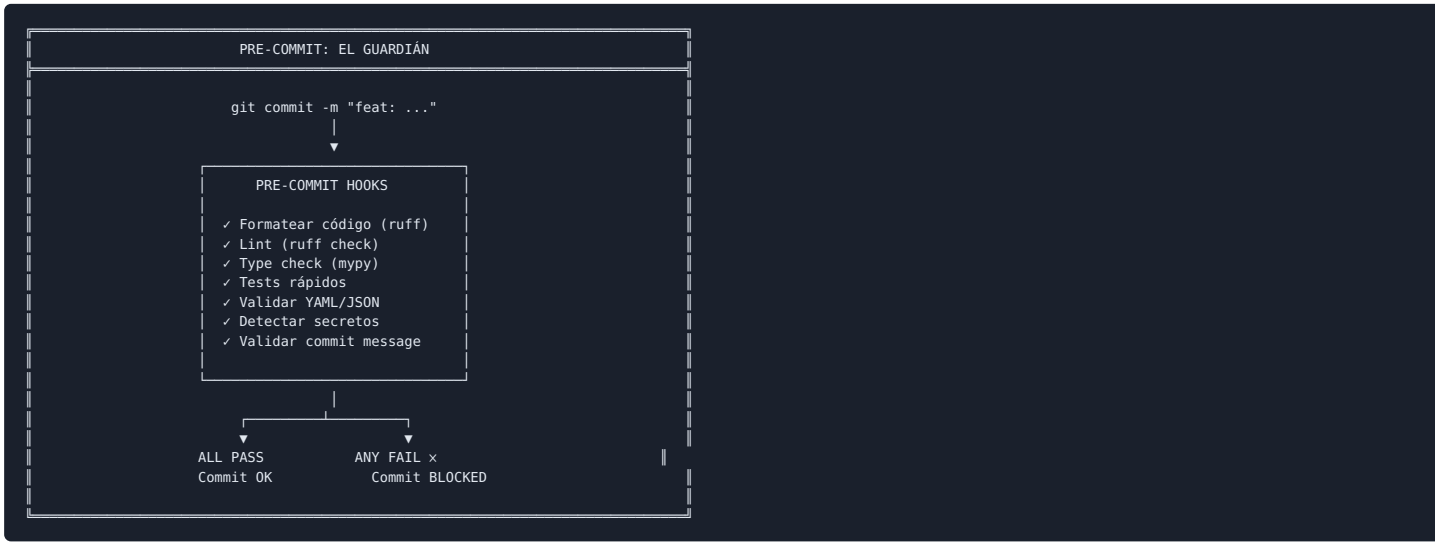
Configurar Commitlint (Validación Automática)

```
# Instalar commitlint
npm install -g @commitlint/cli @commitlint/config-conventional

# Crear config
cat > commitlint.config.js << 'EOF'
module.exports = {
  extends: ['@commitlint/config-conventional'],
  rules: {
    'scope-enum': [2, 'always', [
      'api', 'training', 'inference', 'config', 'data',
      'pipeline', 'model', 'features', 'tests', 'docs',
      'ci', 'docker', 'dvc', 'deps'
    ]],
    'subject-case': [2, 'always', 'lower-case'],
  }
};
EOF
```

4.2 Pre-commit Hooks: Prevenir Errores Antes del Commit

¿Qué Son los Pre-commit Hooks?



Instalación y Setup

```
# Instalar pre-commit
pip install pre-commit

# Instalar hooks en el repo
pre-commit install
pre-commit install --hook-type commit-msg  # Para commitlint

# Ejecutar en todos los archivos (primera vez)
pre-commit run --all-files
```

.pre-commit-config.yaml Completo

```
# .pre-commit-config.yaml
repos:
  # ═══════════════════════════════════════════════════════════
  # FORMATEO Y LINTING
  # ═══════════════════════════════════════════════════════════
  - repo: https://github.com/astral-sh/ruff-pre-commit
    rev: v0.1.6
    hooks:
      - id: ruff
        args: [--fix, --exit-non-zero-on-fix]
      - id: ruff-format

  # ═══════════════════════════════════════════════════════════
  # TYPE CHECKING
  # ═══════════════════════════════════════════════════════════
  - repo: https://github.com/pre-commit/mirrors-mypy
    rev: v1.7.0
    hooks:
      - id: mypy
        args: [--ignore-missing-imports]
        additional_dependencies:
          - pydantic>=2.0.0
          - types-PyYAML

  # ═══════════════════════════════════════════════════════════
  # GENERAL
  # ═══════════════════════════════════════════════════════════
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v4.5.0
    hooks:
      - id: trailing-whitespace
      - id: end-of-file-fixer
      - id: check-yaml
        args: [--unsafe]  # Para YAML con tags como !ref
      - id: check-json
      - id: check-toml
      - id: check-added-large-files
        args: [--maxkb=1000]
      - id: check-merge-conflict
      - id: detect-private-key
      - id: no-commit-to-branch
        args: [--branch, main, --branch, master]

  # ═══════════════════════════════════════════════════════════
  # SEGURIDAD
  # ═══════════════════════════════════════════════════════════
  - repo: https://github.com/Yelp/detect-secrets
    rev: v1.4.0
    hooks:
      - id: detect-secrets
        args: [--baseline, .secrets.baseline]

  - repo: https://github.com/PyCQA/bandit
    rev: 1.7.5
    hooks:
      - id: bandit
        args: [-c, pyproject.toml]
        additional_dependencies: ["bandit[toml]"]

  # ═══════════════════════════════════════════════════════════
  # CONVENTIONAL COMMITS
  # ═══════════════════════════════════════════════════════════
  - repo: https://github.com/compilerla/conventional-pre-commit
    rev: v3.0.0
    hooks:
      - id: conventional-pre-commit
        stages: [commit-msg]
        args: [feat, fix, docs, style, refactor, test, perf, ci, build, chore]

  # ═══════════════════════════════════════════════════════════
  # JUPYTER NOTEBOOKS
  # ═══════════════════════════════════════════════════════════
  - repo: https://github.com/kynan/nbstripout
    rev: 0.6.1
    hooks:
      - id: nbstripout  # Limpia outputs de notebooks

  # ═══════════════════════════════════════════════════════════
  # DOCKER
  # ═══════════════════════════════════════════════════════════
  - repo: https://github.com/hadolint/hadolint
    rev: v2.12.0
    hooks:
      - id: hadolint-docker
        args: [--ignore, DL3008, --ignore, DL3013]

# Configuración global
default_language_version:
  python: python3.11

ci:
  autofix_commit_msg: "style: auto-fix by pre-commit hooks"
  autoupdate_commit_msg: "chore: update pre-commit hooks"
```

pyproject.toml Sección Bandit

```
# pyproject.toml
[tool.bandit]
exclude_dirs = ["tests", "scripts"]
skips = ["B101"]  # Skip assert warnings in tests
```

Comandos Pre-commit Útiles

```
# Ejecutar en archivos staged
pre-commit run

# Ejecutar en todos los archivos
pre-commit run --all-files

# Ejecutar hook específico
pre-commit run ruff --all-files
pre-commit run mypy --all-files

# Actualizar hooks a últimas versiones
pre-commit autoupdate

# Skip hooks temporalmente (emergencia)
git commit --no-verify -m "hotfix: emergency fix"
# ⚠ USAR SOLO EN EMERGENCIAS
```

4.3 Estrategias de Branching

Git Flow vs GitHub Flow vs Trunk-Based

GitHub Flow para MLOps (Recomendado)

```
gitGraph
    commit id: "initial"
    branch feature/add-mlflow
    commit id: "feat(tracking): add MLflow integration"
    commit id: "test(tracking): add tests for experiment tracking"
    checkout main
    merge feature/add-mlflow id: "PR #12"
    branch feature/api-batch
    commit id: "feat(api): add batch prediction endpoint"
    checkout main
    merge feature/api-batch id: "PR #13"
    branch fix/nan-handling
    commit id: "fix(training): handle NaN in features"
    checkout main
    merge fix/nan-handling id: "PR #14"
```

Convenciones de Naming para Branches

```
# Features
feature/add-mlflow-tracking
feature/api-batch-prediction
feature/JIRA-123-user-auth

# Fixes
fix/nan-handling
fix/memory-leak-inference
fix/JIRA-456-login-error

# Refactors
refactor/config-pydantic
refactor/training-pipeline

# Experiments (para ML)
experiment/xgboost-vs-rf
experiment/feature-selection

# Releases (si usas Git Flow)
release/1.2.0
hotfix/1.2.1
```

4.4 Comandos Avanzados que Todo Senior Debe Conocer

Rebase Interactivo: Limpiar Historial

```
# Últimos 3 commits
git rebase -i HEAD~3

# Opciones en el editor:
# pick   = usar commit as-is
# reword = cambiar mensaje
# edit   = pausar para editar
# squash = combinar con anterior
# fixup  = combinar sin mensaje
# drop   = eliminar commit

# Ejemplo: Combinar 3 commits WIP en uno
# pick abc123 feat(api): add endpoint
# squash def456 wip
# squash ghi789 fix typo
# → Se convierten en un solo commit limpio
```

Cherry-pick: Traer Commits Específicos

```
# Traer un commit de otra rama
git cherry-pick abc123

# Traer varios commits
git cherry-pick abc123 def456

# Traer sin commitear (para combinar)
git cherry-pick --no-commit abc123
```

Bisect: Encontrar el Commit que Rompió Algo

```
# Iniciar bisect
git bisect start

# Marcar estado actual como malo
git bisect bad

# Marcar un commit conocido como bueno
git bisect good v1.0.0

# Git te lleva a un commit intermedio
# Testear y marcar:
git bisect good  # Si funciona
git bisect bad   # Si está roto

# Repetir hasta encontrar el commit culpable
# Al final:
git bisect reset
```

Stash: Guardar Cambios Temporalmente

```
# Guardar cambios actuales
git stash

# Con mensaje descriptivo
git stash push -m "WIP: refactoring config"

# Listar stashes
git stash list

# Aplicar último stash
git stash pop

# Aplicar stash específico
git stash apply stash@{2}

# Crear branch desde stash
git stash branch feature/from-stash
```

Reflog: Recuperar lo “Perdido”

```
# Ver historial de operaciones
git reflog

# Recuperar commit "perdido" después de reset
git reflog
# abc123 HEAD@{3}: commit: feat: important change
git checkout abc123
# o
git reset --hard abc123
```

4.5 .gitignore Profesional para MLOps

```
# .gitignore para proyectos MLOps

# ═══════════════════════════════════════════════════════════
# PYTHON
# ═══════════════════════════════════════════════════════════
__pycache__/
*.py[cod]
*$py.class
*.so
.Python
build/
develop-eggs/
dist/
downloads/
eggs/
.eggs/
lib/
lib64/
parts/
sdist/
var/
wheels/
*.egg-info/
.installed.cfg
*.egg

# ═══════════════════════════════════════════════════════════
# ENTORNOS VIRTUALES
# ═══════════════════════════════════════════════════════════
.venv/
venv/
ENV/
env/
.conda/

# ═══════════════════════════════════════════════════════════
# IDEs
# ═══════════════════════════════════════════════════════════
.idea/
.vscode/
*.swp
*.swo
*~
.spyderproject
.spyproject

# ═══════════════════════════════════════════════════════════
# JUPYTER NOTEBOOKS
# ═══════════════════════════════════════════════════════════
.ipynb_checkpoints/
*.ipynb_checkpoints/

# ═══════════════════════════════════════════════════════════
# DATOS Y MODELOS (gestionados por DVC)
# ═══════════════════════════════════════════════════════════
data/raw/*
data/processed/*
models/*.pkl
models/*.joblib
!data/raw/.gitkeep
!data/processed/.gitkeep
!models/.gitkeep

# DVC
/data/*.csv
/data/*.parquet

# ═══════════════════════════════════════════════════════════
# MLFLOW
# ═══════════════════════════════════════════════════════════
mlruns/
mlartifacts/

# ═══════════════════════════════════════════════════════════
# SECRETOS Y CONFIGURACIÓN LOCAL
# ═══════════════════════════════════════════════════════════
.env
.env.*
!.env.example
*.pem
*.key
secrets/
```

```
credentials/

# ═══════════════════════════════════════════════════════════
# TESTING Y COVERAGE
# ═══════════════════════════════════════════════════════════
.coverage
.pytest_cache/
htmlcov/
.tox/
.nox/
coverage.xml
*.cover
.hypothesis/

# ═══════════════════════════════════════════════════════════
# BUILDS Y DOCS
# ═══════════════════════════════════════════════════════════
site/
docs/_build/
*.log

# ═══════════════════════════════════════════════════════════
# OS
# ═══════════════════════════════════════════════════════════
.DS_Store
Thumbs.db
```

4.6 Ejercicio Integrador: Setup Completo de Git

Paso 1: Configurar Git Global

```
# Identidad
git config --global user.name "Tu Nombre"
git config --global user.email "tu@email.com"

# Editor (VS Code)
git config --global core.editor "code --wait"

# Alias útiles
git config --global alias.st "status -sb"
git config --global alias.co "checkout"
git config --global alias.br "branch"
git config --global alias.cm "commit -m"
git config --global alias.lg "log --oneline --graph --all"
git config --global alias.last "log -1 HEAD --stat"
git config --global alias.unstage "reset HEAD --"

# Auto-setup remote tracking
git config --global push.autoSetupRemote true

# Default branch
git config --global init.defaultBranch main
```

Paso 2: Inicializar Proyecto

```
# Crear repo
mkdir bankchurn-predictor && cd bankchurn-predictor
git init

# Crear estructura
mkdir -p src/bankchurn/{data,models,utils} tests/{unit,integration} configs docs

# Archivos base
touch src/bankchurn/__init__.py
touch .gitignore .pre-commit-config.yaml pyproject.toml README.md

# Primer commit
git add .
git commit -m "chore: initial project structure"
```

Paso 3: Configurar Pre-commit

```
# Instalar
pip install pre-commit

# Copiar el .pre-commit-config.yaml de la sección 4.2

# Instalar hooks
pre-commit install
pre-commit install --hook-type commit-msg

# Ejecutar en todos los archivos
pre-commit run --all-files
```

Paso 4: Crear Feature Branch y PR

```
# Crear branch
git checkout -b feature/add-config

# Hacer cambios...
# Commit con conventional commits
git commit -m "feat(config): add Pydantic configuration models"

# Push
git push -u origin feature/add-config

# Crear PR en GitHub
# (usar template de PR si existe)
```

Checklist de Verificación

```
CONFIGURACIÓN:
[ ] Git configurado con nombre y email
[ ] Alias útiles configurados
[ ] Default branch es main

PRE-COMMIT:
[ ] pre-commit instalado
[ ] Hooks activos (commit + commit-msg)
[ ] Todos los hooks pasan en --all-files

FLUJO:
[ ] Puedo crear feature branches correctamente
[ ] Commits siguen Conventional Commits
[ ] .gitignore excluye archivos correctos
```

4.7 Autoevaluación

Preguntas de Reflexión

1. ¿Por qué Conventional Commits permite generar changelogs automáticamente?
2. ¿Cuál es la diferencia entre `git rebase` y `git merge`?
3. ¿Cuándo usarías `git stash` vs crear un branch?
4. ¿Por qué `no-commit-to-branch` es un hook útil?

Comandos que Debes Dominar

```
# Básicos
git status, add, commit, push, pull

# Branching
git branch, checkout -b, merge

# Historial
git log --oneline --graph, diff, show

# Avanzados
git rebase -i, cherry-pick, bisect, stash, reflog

# Pre-commit
pre-commit run, --all-files, autoupdate
```

Siguiente Paso

Con Git dominado, es hora de versionar **datos** profesionalmente.

Ir a Módulo 05: Ingeniería de Datos y DVC →

Módulo 04 completado. Tu historial de Git ahora cuenta una historia clara.