IACADEMY

DE FUNDAMENTOS A ARQUITECTURA DE IA

MÓDULO 16

Git y repositorios con IA

Avanzado

iacedemy.com — 2026

MÓDULO 16

Git y repositorios con IA

Nivel: Avanzado

Autor: Ricardo Gutierrez

Publicación: Mayo 2026

Plataforma: iacedemy.com

Git fundamentals

Git es un sistema de control de versiones. Cada cambio que haces queda registrado como un snapshot (commit). Puedes volver a cualquier punto, comparar versiones, trabajar en paralelo con ramas, y colaborar con otros sin pisar el trabajo de nadie.

Si no usas Git, cada cambio es un riesgo. Un archivo borrado, un bug introducido, un deploy fallido. Con Git, todo es reversible.

Comandos basicos

```
 # Inicializar un repositorio nuevo
git init

# Clonar un repositorio existente
git clone https://github.com/tu-usuario/tu-proyecto.git

# Ver el estado (que ha cambiado desde el ultimo commit)
git status

# Ver los cambios exactos (diff)
git diff                    # Cambios no staged
git diff --staged           # Cambios staged (listos para commit)

# Agregar archivos al staging area
git add src/api/routes.py          # Un archivo especifico
git add src/api/                   # Todo el directorio
git add .                          # Todo (cuidado: revisa antes con git
status)

# Crear un commit (snapshot permanente)
git commit -m "feat: add task creation endpoint with Pydantic validation"

# Ver historial de commits
git log --oneline              # Vista compacta
git log --oneline --graph      # Con grafico de ramas
git log -5                     # Ultimos 5 commits
```

Ramas (branches)

Las ramas permiten trabajar en funcionalidades nuevas sin afectar al codigo principal. Cuando terminas, fusionas (merge) la rama.

```
 # Crear y cambiar a una nueva rama
git checkout -b feature/user-auth

# Listar ramas
git branch          # Locales
git branch -a       # Locales + remotas

# Cambiar de rama
git checkout main

# Fusionar una rama en la actual
git checkout main
git merge feature/user-auth

# Eliminar rama (despues de merge)
git branch -d feature/user-auth
```

Merge vs Rebase

Dos formas de integrar cambios de una rama a otra:

```
 # MERGE: crea un commit de fusion. Preserva el historial completo.
git checkout main
git merge feature/user-auth
# Resultado: commit de merge que une las dos lineas

# REBASE: reescribe el historial. Linea limpia, sin bifurcaciones.
git checkout feature/user-auth
git rebase main
git checkout main
git merge feature/user-auth  # Fast-forward, sin commit de merge
```

Cuando usar merge: en equipo, cuando quieres preservar quien hizo que y cuando. Es la opcion segura.

Cuando usar rebase: trabajando solo, cuando quieres un historial limpio y lineal. Mas facil de leer.

Regla critica: **NUNCA** hagas rebase de ramas que ya estan en remoto y que otros pueden estar usando. Rebase reescribe el historial, y eso rompe el trabajo de los demas.

Resolver conflictos

```
 # Si un merge tiene conflictos, Git marca los archivos
# Abre el archivo y veras algo asi:
<<<<<<< HEAD
def get_user(user_id: str):
    return db.users.find_one({"id": user_id})
=======
def get_user(user_id: str) -> User:
    result = db.users.find_one({"id": user_id})
    return User(**result) if result else None
>>>>>>> feature/user-auth

# Decide que version quieres (o combina ambas), elimina los marcadores,
# y luego:
git add src/services/user_service.py
git commit -m "fix: resolve merge conflict in get_user"
```

GitHub essentials

GitHub no es solo donde guardas codigo. Es tu plataforma completa de desarrollo: repositorios, pull requests, issues, projects, discussions, actions, y pages.

Pull Requests (PRs)

Un PR es una propuesta de cambios. Antes de que el codigo llegue a main, alguien (o un bot de CI) lo revisa.

```
 # Flujo completo con gh CLI
# 1. Crear rama y hacer cambios
git checkout -b feature/task-api
# ... hacer cambios y commits ...

# 2. Push a remoto
```

```
git push -u origin feature/task-api

# 3. Crear PR
gh pr create \
  --title "feat: add task CRUD endpoints" \
  --body "## Summary
- Add POST /api/v1/tasks with Pydantic validation
- Add GET /api/v1/tasks with pagination
- Add PATCH /api/v1/tasks/{id}
- Add DELETE /api/v1/tasks/{id} (soft delete)

## Test plan
- [ ] Unit tests pass
- [ ] Integration tests pass
- [ ] Manual test with Swagger UI"

# 4. Ver PRs abiertos
gh pr list

# 5. Ver el diff de un PR
gh pr diff 42

# 6. Revisar y aprobar
gh pr review 42 --approve --body "LGTM, tests pasan"

# 7. Mergear
gh pr merge 42 --squash --delete-branch
```

Issues y Projects

```
 # Crear un issue
gh issue create \
  --title "Bug: tasks endpoint returns 500 on empty title" \
  --body "Steps to reproduce: POST /api/v1/tasks with empty title"

# Listar issues
gh issue list --state open

# Cerrar un issue (automatico con commit message)
git commit -m "fix: validate task title is not empty

Closes #42"
```

Branching strategy

Trunk-based development (developer solo o equipo pequeno)

Una rama principal: `main`. Feature branches cortas (horas, no semanas). Merge directo a main con PR. Deploy continuo desde main.

```
 # Flujo trunk-based
git checkout -b fix/empty-title-validation   # Branch corta
# ... cambios ...
git commit -m "fix: validate task title is not empty"
git push -u origin fix/empty-title-validation
gh pr create --title "fix: validate task title"
# CI pasa > review > merge > deploy automatico
```

Git Flow (equipo grande o releases programados)

```
 # Ramas permanentes
main          # Produccion (siempre deployable)
develop       # Integracion (donde se merge todo)

# Ramas temporales
feature/*     # Nueva funcionalidad (sale de develop, vuelve a develop)
release/*     # Preparar version (sale de develop, va a main y develop)
hotfix/*      # Parche urgente en produccion (sale de main, va a main y
develop)

# Ejemplo: nueva funcionalidad
git checkout develop
git checkout -b feature/user-roles
# ... desarrollo ...
git checkout develop
git merge feature/user-roles

# Ejemplo: release
git checkout develop
git checkout -b release/2.1.0
# ... ajustes finales, version bump ...
git checkout main
git merge release/2.1.0
git tag v2.1.0
```

```
git checkout develop
git merge release/2.1.0
```

> **Recomendacion**
>
> Si eres developer solo o equipo de 2-3, usa trunk-based. Si el equipo crece a 5+ personas o necesitas releases planificados, migra a Git Flow. No empieces con Git Flow para un proyecto personal.

CI/CD con GitHub Actions

CI/CD automatiza lo que harias a mano: comprobar que el codigo esta limpio, pasar tests, construir el artefacto, y deployar. Se dispara automaticamente con cada push o PR.

Workflow completo y real

```
 # .github/workflows/ci.yml
name: CI/CD Pipeline

on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

env:
  PYTHON_VERSION: "3.11"

jobs:
  lint:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: ${{ env.PYTHON_VERSION }}
      - name: Install linter
        run: pip install ruff
```

```
      - name: Check format
        run: ruff format --check src/
      - name: Check lint
        run: ruff check src/

  test:
    needs: lint  # Solo corre si lint pasa
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: ${{ env.PYTHON_VERSION }}
      - name: Install dependencies
        run: |
          pip install -r requirements.txt
          pip install pytest pytest-cov pytest-asyncio
      - name: Run tests
        run: pytest tests/ -v --cov=src --cov-report=term --cov-fail-under=70
      - name: Upload coverage
        if: always()
        uses: actions/upload-artifact@v4
        with:
          name: coverage-report
          path: htmlcov/

  security:
    needs: lint
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
        with:
          fetch-depth: 0  # Historial completo para gitleaks
      - name: Run gitleaks
        uses: gitleaks/gitleaks-action@v2
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}

  deploy:
    needs: [test, security]  # Solo si tests Y security pasan
    if: github.ref == 'refs/heads/main' && github.event_name == 'push'
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Deploy to Cloudflare Pages
```

```
        uses: cloudflare/wrangler-action@v3
        with:
          apiToken: ${{ secrets.CF_API_TOKEN }}
          command: pages deploy dist --project-name=mi-proyecto
```

Anatomia del workflow

- **on**: cuando se dispara (push a main, PR a main)
- **jobs**: tareas independientes que corren en paralelo (a menos que uses `needs`)
- **needs**: dependencias entre jobs. `test` espera a que `lint` pase.
- **if**: condiciones. Deploy solo en push a main, no en PRs.
- **secrets**: variables sensibles configuradas en Settings > Secrets de GitHub. NUNCA en el YAML.

Workflow para Next.js (frontend)

```
 # .github/workflows/frontend.yml
name: Frontend CI

on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: 20
          cache: 'npm'
      - run: npm ci
      - run: npm run lint
      - run: npm run build
      - run: npm test -- --coverage
```

GitHub Pages: hosting gratis

GitHub Pages te da hosting gratuito para sitios estaticos. Perfecto para documentacion, portfolios, landing pages, y blogs estaticos.

```
 # Opcion 1: Deploy automatico con Actions
# .github/workflows/deploy-pages.yml
name: Deploy to GitHub Pages

on:
  push:
    branches: [main]

permissions:
  contents: read
  pages: write
  id-token: write

jobs:
  deploy:
    runs-on: ubuntu-latest
    environment:
      name: github-pages
      url: ${{ steps.deployment.outputs.page_url }}
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: 20
      - run: npm ci && npm run build
      - uses: actions/configure-pages@v4
      - uses: actions/upload-pages-artifact@v3
        with:
          path: './dist'
      - id: deployment
        uses: actions/deploy-pages@v4

# Opcion 2: rama gh-pages (clasica)
# Settings > Pages > Source: Deploy from a branch > gh-pages

# Custom domain
# 1. Crear archivo CNAME en la raiz del site con tu dominio
```

```
echo "mi-proyecto.com" > dist/CNAME
# 2. Configurar DNS: CNAME apuntando a tu-usuario.github.io
```

GitHub Copilot vs Claude Code

Son herramientas complementarias, no competidoras. Cada una tiene un punto fuerte diferente.

GitHub Copilot

- **Mejor para**: autocompletar codigo en tiempo real mientras escribes en el editor
- **Contexto**: el archivo actual y archivos abiertos en el editor
- **Integracion**: VS Code, JetBrains, Neovim
- **Precio**: 10 USD/mes (individual), 19 USD/mes (business)
- **Ideal para**: escribir codigo nuevo, completar funciones, generar tests inline

Claude Code

- **Mejor para**: tareas complejas que afectan multiples archivos, refactoring, arquitectura
- **Contexto**: todo el repositorio, git history, archivos de configuracion
- **Integracion**: terminal, cualquier editor indirectamente
- **Precio**: incluido en Claude Pro (20 USD/mes) o por tokens via API
- **Ideal para**: refactors grandes, crear componentes completos, CI/CD, debugging

> **Combinacion ganadora**
>
> Usa **Copilot** mientras escribes codigo nuevo (autocompletar rapido). Usa **Claude Code** para tareas de arquitectura, refactoring multi-archivo, crear PRs, y debugging complejo. Son complementarios.

Claude Code + Git

Claude Code entiende Git de forma nativa. Lee el historial, entiende los diffs, hace commits con mensajes descriptivos, y puede crear PRs.

```
 # Claude Code lee el historial para entender el proyecto
claude "Mira los ultimos 20 commits y resume que se ha hecho esta semana"

# Claude Code analiza cambios
claude "Que archivos han cambiado desde el ultimo tag? Resume los cambios."

# Claude Code hace commits semanticos
claude "Haz commit de los cambios actuales con un mensaje
descriptivo siguiendo Conventional Commits"

# Claude Code crea PRs completos
claude "Crea un PR con los cambios de esta rama.
Titulo corto (max 70 chars).
Body con: Summary (3 bullets), Test plan (checklist)."

# Claude Code hace code review
claude "Revisa el diff del PR #42. Busca:
1. Bugs potenciales
2. Vulnerabilidades de seguridad
3. Violaciones de estilo
4. Tests faltantes"

# Claude Code resuelve conflictos
claude "Hay conflictos de merge en src/api/routes.py.
Resuelvelos manteniendo ambos cambios donde sea posible."
```

CLAUDE.md como contexto para Git

Claude Code lee tu archivo `CLAUDE.md` automaticamente. Incluye en el las convenciones de Git del proyecto:

```
 # En CLAUDE.md
## Convenciones Git
- Commits: Conventional Commits (feat:, fix:, chore:, docs:, refactor:)
- Ramas: feature/, fix/, chore/, docs/ + descripcion corta en kebab-case
- PRs: titulo max 70 chars, body con Summary y Test plan
```

```
- Merge strategy: squash merge para features, regular merge para releases
- NUNCA force push a main
```

.gitignore patterns

El .gitignore define que archivos NO deben entrar en el repositorio. Es critico para seguridad (secretos) y limpieza (archivos generados).

```
 # .gitignore completo para proyecto Python + Next.js

# === Secretos (CRITICO) ===
.env
.env.*
*.pem
*.key
*.p12
credentials.json
service-account.json

# === Python ===
__pycache__/
*.py[cod]
*$py.class
*.so
venv/
.venv/
*.egg-info/
dist/
build/
.pytest_cache/
.coverage
htmlcov/
.mypy_cache/
.ruff_cache/

# === Node.js / Next.js ===
node_modules/
.next/
out/
dist/
.turbo/
.vercel/
```

```
# === IDE ===
.vscode/settings.json    # settings personales (no del proyecto)
.idea/
*.swp
*.swo
*~

# === OS ===
.DS_Store
Thumbs.db

# === Bases de datos locales ===
*.sqlite
*.sqlite3
*.db

# === Logs ===
*.log
npm-debug.log*

# === Claude Code (memoria local, no compartir) ===
# .claude/memory/  # Decide si compartir o no segun el proyecto
```

Regla: si dudas, excluyelo

Es mas facil agregar un archivo al repo despues que quitarlo del historial. Si no estas seguro de si un archivo deberia estar en el repo, anadelo al .gitignore.

Monorepos

Un monorepo contiene multiples proyectos en un solo repositorio. Tiene sentido cuando los proyectos comparten codigo, dependencias, o necesitan coordinarse.

```
 # Estructura monorepo simple
monorepo/
  apps/
    web/           # Next.js frontend
    api/           # FastAPI backend
```

```
    docs/          # Documentacion
  packages/
    shared/        # Codigo compartido (tipos, utilidades)
    ui/            # Componentes UI reutilizables
  .github/
    workflows/     # CI/CD compartido
```

Herramientas para monorepos

- **Turborepo**: de Vercel, optimizado para JavaScript/TypeScript. Cache inteligente, ejecucion en paralelo.
- **Nx**: mas completo, soporta multiples lenguajes. Grafico de dependencias, afectados por cambios.
- **Scripts simples**: para monorepos pequenos, un Makefile o scripts bash son suficientes.

```
 # Makefile para monorepo simple
.PHONY: install test lint deploy

install:
        cd apps/api && pip install -r requirements.txt
        cd apps/web && npm ci

test:
        cd apps/api && pytest tests/ -v
        cd apps/web && npm test

lint:
        cd apps/api && ruff check src/
        cd apps/web && npm run lint

deploy-api:
        cd apps/api && ./deploy.sh

deploy-web:
        cd apps/web && npm run build && npx wrangler pages deploy dist
```

Cuando usar monorepo: proyectos que comparten codigo, equipos que necesitan cambios coordinados, APIs que deben mantenerse sincronizadas con el frontend.

Cuando usar repos separados: proyectos completamente independientes, equipos que deployean en cadencias diferentes, proyectos con stack completamente distinto.

Alternativas a GitHub

GitHub es el estandar, pero hay alternativas validas:

- **GitLab**: CI/CD integrado mas potente que Actions. Opcion self-hosted para soberania de datos. Gratis para equipos pequenos.
- **Codeberg**: alternativa open source, sin animo de lucro, basada en Gitea. Para proyectos open source que quieren independencia de Microsoft/GitHub.
- **Gitea**: self-hosted, ligero, facil de instalar. Perfecto si necesitas control total.
- **Bitbucket**: integracion nativa con Jira/Confluence (ecosistema Atlassian). Comun en empresas que usan Jira.

Para la mayoria de los proyectos, GitHub sigue siendo la mejor opcion por el ecosistema (Actions, Pages, Copilot, security features, comunidad).

Secretos en repos: como se filtran y como prevenirlo

Los secretos filtrados en repositorios son la causa numero uno de brechas en proyectos open source y una de las principales en proyectos privados. Merece repetirlo desde el M13 con mas detalle.

Como se filtran secretos

1. **Commit accidental**: anadir .env al commit sin darse cuenta
2. **Copiar y pegar**: pegar una API key en el codigo "para probar rapido"
3. **Logs**: imprimir variables de entorno en logs de debug
4. **Hacer publico un repo privado**: el historial contiene todo
5. **Forks**: alguien forkea tu repo antes de que borres el secreto

Doble barrera de prevencion

```
 # 1. gitleaks como pre-commit hook (primera barrera: local)
cat > .git/hooks/pre-commit << 'HOOK'
```

```
#!/bin/sh
gitleaks protect --staged --verbose
if [ $? -ne 0 ]; then
    echo ""
    echo "SECRETO DETECTADO. Commit BLOQUEADO."
    echo "Elimina el secreto del archivo y vuelve a intentar."
    exit 1
fi
HOOK
chmod +x .git/hooks/pre-commit

# 2. gitleaks en CI (segunda barrera: servidor)
# Ya incluido en el workflow de CI de arriba

# 3. pre-commit framework (alternativa mas completa)
pip install pre-commit

# .pre-commit-config.yaml
repos:
  - repo: https://github.com/gitleaks/gitleaks
    rev: v8.18.0
    hooks:
      - id: gitleaks
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v4.5.0
    hooks:
      - id: check-added-large-files
        args: ['--maxkb=1000']
      - id: detect-private-key

# Instalar hooks
pre-commit install
```

Que hacer si un secreto se filtro

```
 # 1. INMEDIATAMENTE: rotar el secreto en el servicio
# Genera una nueva API key en el dashboard del servicio

# 2. Eliminar del historial (si el repo es privado)
# Usa git filter-repo (recomendado sobre filter-branch)
pip install git-filter-repo
git filter-repo --path .env --invert-paths

# 3. Force push (DESTRUCTIVO, solo si eres el unico en el repo)
```

```
git push --force

# 4. Si el repo es publico: considera que el secreto esta comprometido
# GitHub escanea repos publicos y notifica a proveedores (GitHub Secret
Scanning)
# Pero bots maliciosos tambien escanean. Rota SIEMPRE.
```

Regla de oro de secretos en Git

Un secreto commiteado una vez esta en el historial **para siempre** (a menos que reescribas el historial, que es destructivo y complicado). La prevencion con pre-commit hooks es 100x mas facil que la remediacion. Invierte 5 minutos en configurar gitleaks ahora.

Ejercicio practico

Ejercicio M16: Setup completo de un repositorio profesional

1. **Crear repositorio**: Usa `gh repo create` para crear un repo privado con README y .gitignore.
2. **Estructura**: Crea la estructura de directorios (src/, tests/, .github/workflows/).
3. **CI/CD**: Configura un workflow de GitHub Actions con 3 jobs: lint, test, deploy. Usa el ejemplo de este modulo como base.
4. **gitleaks**: Instala gitleaks como pre-commit hook. Intenta commitear un archivo con un secreto falso para verificar que funciona.
5. **Feature branch**: Crea una rama feature/, haz cambios, y crea un PR con `gh pr create`.
6. **Claude Code**: Usa Claude Code para hacer code review del PR, hacer commit, y generar el changelog.
7. **GitHub Pages**: Si tienes un sitio estatico, deployalo con gh-pages y configura un dominio custom.

Bonus: Configura branch protection rules en Settings: requerir PR review y CI pass antes de merge a main.

Conclusiones clave

Key takeaways del M16

1. Git: init, add, commit, branch, merge, rebase. Estos 6 comandos cubren el 80% de tu uso diario.
2. GitHub: repos, PRs, issues, Actions. Tu plataforma completa. El PR es el punto de control antes de main.
3. Branching: trunk-based para developer solo o equipo pequeno. Git Flow para equipos grandes con releases.
4. CI/CD con GitHub Actions: lint > test > security > deploy. Automatizado, con secretos en Settings > Secrets.
5. Copilot para autocompletar en tiempo real. Claude Code para tareas multi-archivo, arquitectura y debugging. Complementarios.
6. Claude Code lee git history, hace commits semanticos, crea PRs y hace code review. Usa CLAUDE.md para convenciones.
7. .gitignore: secretos, node_modules, __pycache__, build, .env. Si dudas, excluyelo.
8. Secretos: gitleaks pre-commit + CI. Un secreto en el historial es permanente. Prevencion es 100x mas facil que remediacion.

Has completado el bloque avanzado

Con seguridad, arquitectura, bases de datos y Git dominados, tienes todas las herramientas para construir proyectos profesionales con IA. Revisa los modulos anteriores y aplica lo aprendido a tu proyecto real.

Volver al indice del curso

IACADEMY

iacedemy.com

De fundamentos a arquitectura de IA.
12 módulos prácticos. 24 recursos descargables.
Quizzes con certificado. Vídeos profesionales.

Empieza gratis en iacedemy.com/free