IACADEMY

DE FUNDAMENTOS A ARQUITECTURA DE IA

MÓDULO 04

Automatización con hooks y agents

Aplicación

iacedemy.com — 2026

MÓDULO 04

Automatización con hooks y agents

Nivel: Aplicación

Autor: Ricardo Gutierrez

Publicación: Mayo 2026

Plataforma: iacedemy.com

Hooks avanzados

En el M03 viste hooks basicos (PreCommit, PostFileWrite). Ahora vamos a usar hooks que controlan lo que Claude Code puede y no puede hacer, y que encadenan multiples validaciones.

PreToolUse: el guardia de seguridad

PreToolUse se ejecuta ANTES de que Claude Code use cualquier herramienta. Es el hook mas potente porque te permite bloquear operaciones peligrosas antes de que ocurran.

```
 // .claude/settings.json
{
  "hooks": {
    "PreToolUse": [{
      "command": "python .claude/hooks/validate-tool.py $TOOL_NAME
$TOOL_INPUT",
      "description": "Validar herramientas antes de ejecutar"
    }]
  }
}
```

Script de validacion:

```
 # .claude/hooks/validate-tool.py
#!/usr/bin/env python3
import sys
import json

tool_name = sys.argv[1]
tool_input = sys.argv[2] if len(sys.argv) > 2 else ""

# Operaciones bloqueadas
BLOCKED_PATTERNS = [
    "rm -rf",
    "DROP TABLE",
    "DELETE FROM",
    "TRUNCATE",
    "force push",
    "reset --hard"
```

```
]

for pattern in BLOCKED_PATTERNS:
    if pattern.lower() in tool_input.lower():
        print(f"BLOQUEADO: '{pattern}' detectado en la operacion")
        print(f"Tool: {tool_name}")
        print(f"Input: {tool_input[:200]}")
        sys.exit(1)

# Si llega aqui, la operacion esta permitida
sys.exit(0)
```

Con este hook activo, Claude Code no puede ejecutar operaciones destructivas aunque se lo pidas. Es un guardrail automatico.

Encadenamiento de hooks

Puedes tener multiples hooks en el mismo evento. Se ejecutan en orden y si uno falla, se cancela la operacion.

```
{
  "hooks": {
    "PreCommit": [
      {
        "command": "python -m pytest tests/ -x --tb=short",
        "description": "Tests unitarios"
      },
      {
        "command": "ruff check src/ --select E,W,F",
        "description": "Lint: errores y warnings"
      },
      {
        "command": "python -m mypy src/ --strict",
        "description": "Verificacion de tipos"
      }
    ],
    "PostCommit": [
      {
        "command": "python .claude/hooks/notify-slack.py",
        "description": "Notificar al canal de Slack"
      }
    ]
```

```
  }
}
```

Secuencia: tests > lint > tipos > commit > notificacion Slack. Si cualquiera de los 3 primeros falla, el commit no se hace.

[IMG: Diagrama de flujo mostrando la cadena de hooks PreCommit: tests, lint, mypy, con flechas de "pass" y "fail"]

> **Hook vs CI/CD**
>
> Los hooks se ejecutan **en tu maquina local, en tiempo real**. CI/CD se ejecuta en el servidor despues del push. Los hooks atrapan errores antes, CI/CD los atrapa despues. Usa ambos.

Agentes con 6 partes

Un agente profesional no es un prompt largo. Tiene 6 componentes diferenciados. Si te falta alguno, el agente funciona pero es fragil.

1. Prompt

Las instrucciones que definen que hace, como lo hace y que restricciones tiene. Es el "que" y el "como".

```
 Eres un agente de code review especializado en seguridad.
Analiza el diff del ultimo commit buscando:
- Inyeccion SQL
- XSS
- Secrets hardcodeados
- Endpoints sin autenticacion
Severidad: critica, alta, media, baja.
Formato: tabla markdown.
```

2. Tools (herramientas)

Las herramientas que puede usar. En Claude Code: leer archivos, ejecutar comandos, buscar texto, crear ficheros, acceder a MCP.

```
 Tools disponibles:
- git diff HEAD~1 (ver cambios recientes)
- grep -r "password\|secret\|api_key" src/ (buscar secrets)
- ruff check (verificar estilo)
- pytest (ejecutar tests)
- Bash (ejecutar cualquier comando)
```

3. Memory (memoria)

Lo que recuerda entre ejecuciones. Sin memoria, cada ejecucion empieza de cero.

```
 # .claude/memory/review-patterns.md
## Errores frecuentes del equipo
- Maria tiende a no parametrizar queries SQL
- Carlos olvida anadir try/except en endpoints
- Los archivos de utils/ raramente tienen tests
- El modulo de pagos tiene deuda tecnica conocida (no tocar sin supervision)
```

4. Planning (planificacion)

Como descompone una tarea compleja en pasos. El chain-of-thought interno del agente.

```
 Plan de ejecucion:
1. Leer el diff completo del ultimo commit
2. Clasificar archivos modificados por modulo
3. Para cada archivo: analizar con las 4 categorias de vulnerabilidad
4. Cruzar con memory/review-patterns.md para patrones conocidos
5. Generar tabla de resultados ordenada por severidad
6. Si hay criticas: flag de BLOQUEANTE
```

5. Execution (ejecucion)

El loop de accion: lee, decide, actua, verifica, repite. El agente no hace un solo paso; itera hasta que la tarea esta completa o se alcanza el limite de reintentos.

6. Reflection (reflexion)

El agente evalua su propio resultado antes de darlo por terminado.

```
 Auto-evaluacion:
- He cubierto TODOS los archivos del diff? (no solo algunos)
- Alguna sugerencia contradice las reglas del CLAUDE.md?
- He dado falsos positivos? (marcar algo como bug cuando no lo es)
- El formato de salida es exactamente el que pide el prompt?
```

Ejemplo completo: agente de security review

```
 # Prompt con las 6 partes integradas
claude "Eres un agente de security review.

[PROMPT] Analiza el diff del ultimo commit buscando vulnerabilidades
OWASP Top 10. Formato: tabla con [Archivo, Linea, Tipo, Severidad, Fix].

[TOOLS] Usa: git diff HEAD~1, grep, ruff check, pytest.

[MEMORY] Lee .claude/memory/review-patterns.md para patrones conocidos.

[PLANNING] Primero lee el diff completo. Luego analiza archivo por archivo.
Luego cruza con patrones conocidos.

[EXECUTION] Para cada vulnerabilidad encontrada: verifica que es real
ejecutando el codigo o buscando el contexto completo de la funcion.

[REFLECTION] Antes de presentar resultados, preguntate: he cubierto
todos los archivos? hay falsos positivos? el formato es correcto?"
```

[IMG: Diagrama hexagonal con las 6 partes del agente: Prompt, Tools, Memory, Planning, Execution, Reflection]

Pipelines multiagente

Un solo agente tiene un limite: el tamano del contexto. Cuando la tarea es grande, dividela en agentes especializados que se pasan el trabajo entre si.

Regla fundamental: comunicacion por archivos, no por contexto

Los agentes NO se pasan informacion a traves del contexto de la conversacion. Se comunican escribiendo y leyendo archivos en disco. Esto evita que el contexto se sature y pierdas precision.

```
 # Pipeline de 3 agentes para refactoring de seguridad

# Agente 1: Scanner
claude "Escanea todos los archivos en src/ buscando vulnerabilidades
OWASP Top 10. Para cada vulnerabilidad encontrada, escribe una linea
en .claude/memory/scan-results.md con formato:
[archivo]:[linea] | [tipo] | [severidad] | [descripcion]
Solo vulnerabilidades reales, no posibles."

# Agente 2: Fixer
claude "Lee .claude/memory/scan-results.md. Para cada vulnerabilidad
de severidad CRITICA o ALTA:
1. Lee el archivo original completo para entender el contexto
2. Aplica el fix minimo necesario (no refactorizar de mas)
3. Anade un comment explicando el fix
Escribe los archivos corregidos en .claude/memory/fixes-applied.md"

# Agente 3: Verifier
claude "Lee .claude/memory/fixes-applied.md. Para cada fix aplicado:
1. Ejecuta los tests existentes para verificar que no rompiste nada
2. Si hay tests de seguridad en tests/security/, ejecutalos tambien
3. Genera un informe final en .claude/memory/security-report.md
Formato: tabla [Vulnerabilidad | Fix aplicado | Tests OK/FAIL]"
```

Cada agente tiene un scope claro, un input definido y un output en archivo. El contexto de cada agente empieza limpio.

Cuando usar pipelines vs un solo agente

- **1 agente:** la tarea cabe en un solo contexto (menos de 5-10 archivos, 1 operacion).
- **Pipeline:** la tarea tiene fases distintas (analizar > implementar > verificar) o toca muchos archivos.

Coordinator + Workers

El patron mas potente para sistemas complejos. Un Coordinator planifica y N Workers ejecutan tareas concretas.

El Coordinator

El Coordinator es el "jefe de proyecto". Su trabajo:

1. Lee la tarea completa y entiende el alcance.
2. Descompone en subtareas concretas y especificas.
3. Asigna cada subtarea a un Worker.
4. **Lee los resultados reales de cada Worker** (NUNCA delega sin leer).
5. Sintetiza el resultado final.

```
 # Coordinator
claude "Eres el coordinador de un proyecto de migracion de base de datos.

Paso 1: Lee la estructura actual en src/models/ y identifica todas las
tablas y relaciones.

Paso 2: Para cada tabla, escribe una tarea especifica en
.claude/memory/tasks/ con el nombre del archivo como identificador.
Formato de tarea:
- Tabla: [nombre]
- Columnas a migrar: [lista]
- Relaciones: [FK a otras tablas]
- Transformaciones necesarias: [cambios de tipo, renames, etc.]

Paso 3: Para cada tarea, lanza un Worker (sub-agente) que ejecute
la migracion de esa tabla especifica.

Paso 4: Lee TODOS los resultados de los Workers y verifica que
la migracion es consistente (no hay FK rotas, no hay datos perdidos)."
```

Los Workers

Cada Worker tiene:

- **Scope acotado:** una sola tabla, un solo archivo, una sola operacion.

- **Scratchpad independiente:** su propio espacio de notas en `.claude/memory/`.
- **Sin comunicacion lateral:** habla solo con el Coordinator, nunca con otros Workers.
- **Circuit breaker:** si falla 3 veces, para y reporta al Coordinator.

```
 # Worker: migrar tabla "users"
claude "Lee la tarea en .claude/memory/tasks/users.md.
Ejecuta la migracion de la tabla users:
1. Crea el script de migracion en migrations/
2. Ejecuta el script en la base de datos de staging
3. Verifica que los datos se migraron correctamente (count, checksums)
4. Escribe el resultado en .claude/memory/results/users.md

Si algo falla despues de 3 intentos, escribe el error en
.claude/memory/results/users-ERROR.md y para."
```

[IMG: Diagrama de Coordinator arriba conectado a 4 Workers abajo, cada uno con su scratchpad. Flechas de ida (tareas) y vuelta (resultados)]

Anti-patterns a evitar

NUNCA hagas esto

- **Agente monolitico:** un agente que hace todo. Dividir siempre.
- **Contexto como canal:** pasar datos entre agentes por el contexto. Usar archivos.
- **Coordinator lazy:** "Basandote en lo que encontraste, haz lo que creas". El Coordinator debe leer los findings reales y dar instrucciones especificas.
- **Sin limites:** agentes que reintentan infinitamente. Siempre max_retries + fallback.
- **Workers que hablan entre si:** crea dependencias invisibles. Solo comunicacion con el Coordinator.

Circuit breakers

Un circuit breaker es un mecanismo que detiene a un agente cuando algo sale mal repetidamente. Sin circuit breakers, un agente puede entrar en un loop infinito quemando tokens y tiempo.

Implementacion basica

```
 # En las instrucciones del agente
claude "...
CIRCUIT BREAKERS:
- Si un test falla 3 veces seguidas con el mismo error: para y reporta
- Si llevas mas de 20 iteraciones sin progreso: para y reporta
- Si un archivo tiene mas de 500 lineas de cambios: para y pide aprobacion
- Si necesitas tocar mas de 10 archivos: para y pide aprobacion
- Si detectas que estas deshaciendo un cambio previo: para (loop detectado)

Cuando se active un circuit breaker, escribe en
.claude/memory/circuit-breaker-log.md:
- Fecha/hora
- Agente
- Motivo de la parada
- Estado actual (que completaste y que falta)"
```

Circuit breakers recomendados

- **Max retries:** 3 intentos para la misma operacion. Si falla 3 veces, para.
- **Max iterations:** 20-30 iteraciones totales por agente. Si no termina, el scope es demasiado grande.
- **Scope guard:** si el agente intenta modificar mas archivos de los esperados, para y confirma.
- **Loop detection:** si esta deshaciendo cambios que acaba de hacer, hay un bug en la logica.
- **HITL (Human In The Loop):** para operaciones destructivas, pedir confirmacion humana.

Ejercicio practico

Ejercicio M04: Pipeline Coordinator + Workers

1. Elige un proyecto tuyo (o usa el del M03).
2. Crea un directorio `.claude/memory/tasks/` y `.claude/memory/results/`.
3. Configura un hook PreToolUse que bloquee operaciones peligrosas (usa el script de ejemplo).
4. Escribe un Coordinator que analice tu proyecto y genere 3 tareas de mejora en `.claude/memory/tasks/`.
5. Ejecuta 2 Workers: uno para la tarea de mayor prioridad y otro para la segunda.
6. Verifica los resultados en `.claude/memory/results/`.
7. Anade circuit breakers a tus agentes (max 3 retries, max 15 iteraciones).

Bonus: Crea un agente de verificacion (Agente 3) que lea los resultados de los Workers y confirme que los cambios son correctos ejecutando tests.

Conclusiones clave

Key takeaways del M04

1. PreToolUse es el hook mas potente: bloquea operaciones peligrosas antes de que ocurran.
2. Un agente profesional tiene 6 partes: Prompt, Tools, Memory, Planning, Execution, Reflection.
3. Los agentes se comunican por archivos, nunca por contexto. Cada agente empieza con contexto limpio.
4. Coordinator + Workers: uno planifica, N ejecutan. El Coordinator siempre lee los resultados reales.
5. Circuit breakers son obligatorios. Sin ellos, un agente puede loopear infinitamente.

6. Anti-patterns: agente monolitico, contexto como canal, coordinator lazy, sin limites.

Has completado los 4 primeros modulos

Ya sabes como funciona la IA, como hacer prompts profesionales, como usar Claude Code y como construir sistemas multiagente. En el M05 conectamos todo con MCP para integrar herramientas externas.

Ir al Modulo 05

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