

Estendere Claude Code

Skills, Subagents, Hooks e Plugins spiegati bene: cosa sono, quando usare cosa, e come si combinano tra loro.

Aggiornato: luglio 2026 · Fonte: documentazione ufficiale Anthropic (code.claude.com/docs) · Companion del cheatsheet dei comandi slash.

0. La mappa delle estensioni

quale strumento per quale problema

Claude Code combina un modello che ragiona sul codice con tool integrati (file, ricerca, esecuzione, web). Le **estensioni** sono il livello sopra: personalizzano quello che Claude sa, lo collegano a servizi esterni e automatizzano i flussi. Ognuna aggrancia una parte diversa del loop agentico:

Feature	Cosa fa	Quando usarla	Esempio
CLAUDE.md	Contesto persistente caricato in ogni sessione , automaticamente.	Convenzioni di progetto, regole "fai sempre X".	"Usa pnpm, non npm. Lancia i test prima di committare."
Skill	Istruzioni, conoscenza e workflow riusabili, caricati on demand .	Contenuti riusabili, documentazione di riferimento, task ripetibili.	<code>/deploy</code> esegue la tua checklist di deploy; skill con i pattern delle API.
Subagent	Worker isolato con il proprio contesto: fa il lavoro e restituisce solo il riassunto.	Isolamento del contesto, task paralleli, worker specializzati.	Ricerca che legge decine di file ma riporta solo i risultati chiave.
Hook	Script/HTTP/prompt eseguito deterministicamente a eventi del ciclo di vita.	Automazioni che devono girare sempre, senza chiedere.	ESLint dopo ogni modifica a un file.
MCP	Protocollo per collegare Claude a servizi esterni (tool e dati).	Dati o azioni esterne al filesystem.	Query al database, post su Slack, controllo del browser.
Agent team	Più sessioni Claude Code indipendenti che comunicano tra loro (sperimentale).	Lavoro complesso che richiede discussione e coordinamento.	Reviewer paralleli su sicurezza, performance e test.
Plugin	Impacchetta skill + agenti + hook + MCP in un'unità installabile.	Riusare lo stesso setup su più repo, o distribuirlo.	Plugin di code-intelligence per il tuo linguaggio, da marketplace.

► Costruisci il setup nel tempo: il trigger giusto per ogni aggiunta

Quando succede questo...	...aggiungi questo
Claude sbaglia una convenzione o un comando due volte	Riga in <code>CLAUDE.md</code>
Continui a digitare lo stesso prompt per avviare un task	Skill invocabile <code>/nome</code>
Incolli in chat lo stesso playbook per la terza volta	Skill
Copi dati da una tab del browser che Claude non vede	Server <code>MCP</code>
Un task secondario inonda la conversazione di output che non rileggerai	Subagent
Vuoi che una cosa succeda ogni volta , senza chiederlo	Hook
Un secondo repository ha bisogno dello stesso setup	Plugin

1. Skills

l'estensione più flessibile

Una skill è una cartella con un file `SKILL.md`: frontmatter YAML che dice a Claude **quando** usarla, e istruzioni markdown che dicono **cosa** fare. La invochi tu con `/nome-skill`, oppure Claude la carica da solo quando la description corrisponde al task. A differenza di `CLAUDE.md`, il corpo di una skill **si carica solo quando serve**: materiale di riferimento anche lungo costa quasi zero finché non viene usato.

Nota storica: i comandi custom sono stati fusi nelle skill. Un file `.claude/commands/deploy.md` e una skill `.claude/skills/deploy/SKILL.md` creano entrambi `/deploy` e funzionano allo stesso modo; le skill aggiungono file di supporto, controllo dell'invocazione e caricamento automatico. Seguono lo standard aperto **Agent Skills**, condiviso tra più tool AI.

◆ Anatomia e prima skill

```
# Struttura di una skill
my-skill/
├─ SKILL.md      # istruzioni (obbligatorio)
├─ template.md   # template da compilare
├─ examples/     # output di esempio
└─ scripts/      # script eseguibili

# ~/.claude/skills/summarize-changes/SKILL.md
---
description: Riassume le modifiche non committate
             e segnala i rischi. Usala quando l'utente chiede
             cosa è cambiato o vuole un commit message.
---

## Modifiche correnti
!`git diff HEAD`

## Istruzioni
Riassumi le modifiche in 2-3 punti, poi elenca i
rischi (error handling mancante, valori hardcoded,
test da aggiornare). Se il diff è vuoto, dillo.
```

I tre superpoteri nel file a fianco

- **description ben scritta** → Claude la invoca da solo quando chiedi "cosa ho cambiato?", senza che tu digiti / summarize-changes.
- **Iniezione dinamica** `!`comando`` → Claude Code esegue il comando *prima* che Claude legga la skill e ne inietta l'output: le istruzioni arrivano già con il diff reale dentro, non con quello che Claude immagina.
- **File di supporto** → template, esempi, script e doc di riferimento citati dal SKILL.md, caricati solo quando servono.

Regola d'oro: corpo conciso

Una volta caricata, la skill **resta in contesto** per i turni successivi: ogni riga è un costo ricorrente in token. Scrivi *cosa fare*, non narrazioni sul perché. Stesso test di concisione di CLAUDE.md (che a sua volta va tenuto sotto ~200 righe: se cresce, sposta il materiale in skill o in `.claude/rules/`).

◆ Dove vivono e chi vince

Livello	Percorso	Vale per
Enterprise	managed settings dell'organizzazione	Tutti gli utenti dell'org (priorità massima)
Personale	~/.claude/skills/<nome>/SKILL.md	Tutti i tuoi progetti
Progetto	.claude/skills/<nome>/SKILL.md	Solo quel progetto (committala nel repo!)
Plugin	<plugin>/skills/<nome>/SKILL.md	Dove il plugin è abilitato, namespace <code>plugin:skill</code>

A parità di nome: enterprise > personale > progetto, e ogni livello **sovrascrive le skill bundled** omonime (una `code-review` di progetto rimpiazza quella di serie). Le skill si caricano anche da directory annidate: in un monorepo, `packages/frontend/.claude/skills/` si attiva quando Claude lavora su quei file (nome qualificato: `/packages/frontend:deploy`). Le modifiche su disco vengono rilevate **live**, senza riavviare.

◆ Frontmatter: i campi che contano

Campo	A cosa serve
description	Cosa fa la skill e quando usarla. È il criterio con cui Claude decide di caricarla da solo: metti il caso d'uso principale all'inizio (troncata a 1.536 caratteri nel listing insieme a <code>when_to_use</code>).
disable-model-invocation	<code>true</code> = solo tu puoi invocarla con /nome; Claude non la carica mai da solo. Per workflow con effetti (deploy, commit) che non vuoi partano in autonomia.
user-invocable	<code>false</code> = nascosta dal menu /: conoscenza di background che non ha senso invocare a mano.
allowed-tools / disallowed-tools	Tool utilizzabili senza chiedere permesso mentre la skill è attiva / tool rimossi dal pool (es. niente <code>AskUserQuestion</code> in un loop autonomo).
model / effort	Modello ed effort da usare mentre la skill è attiva (override per il turno corrente, poi si torna a quelli di sessione).
context: fork + agent	Esegue la skill in un subagente forked invece che inline; <code>agent</code> sceglie quale tipo di subagente. Il ponte tra skill e subagente.
argument-hint / arguments	Suggerimento mostrato nell'autocomplete (es. <code>[issue-number]</code>) e nomi posizionali per le sostituzioni <code>\$nome</code> .
paths	Glob che limitano l'attivazione automatica ai soli file corrispondenti (es. solo quando tocchi <code>*.sql</code>).
hooks	Hook con scope limitato al ciclo di vita della skill.

◆ Sostituzioni disponibili nel corpo

Variabile	Espansione
\$ARGUMENTS	Tutti gli argomenti passati all'invocazione (se assente, vengono accodati come <code>ARGUMENTS: ...</code>).
\$0, \$1... / \$ARGUMENTS[N]	Singolo argomento per posizione (0-based).
\$nome	Argomento nominato dichiarato in <code>arguments:</code> (es. <code>arguments: [issue, branch] → \$issue, \$branch</code>).
\${CLAUDE_SKILL_DIR}	Directory della skill: per referenziare script e file inclusi indipendentemente dalla working directory.
\${CLAUDE_PROJECT_DIR} / \${CLAUDE_SESSION_ID} / \${CLAUDE Effort}	Root del progetto, ID sessione, livello di effort corrente.

2. Subagents

worker isolati, contesto pulito

Un subagente è un assistente specializzato con **la propria finestra di contesto**, un system prompt custom, tool e permessi indipendenti. Il caso d'uso chiave: un task secondario che inonderebbe la conversazione principale di ricerche, log e contenuti di file che non rileggerai mai. Il subagente fa quel lavoro nel suo contesto e **restituisce solo il riassunto**. Definisci un subagente custom quando ti accorgi di lanciare sempre lo stesso tipo di worker con le stesse istruzioni.

Perché usarli (in 5 punti)

- **Preservano il contesto:** esplorazione e ricerca restano fuori dalla conversazione principale.
- **Vincolano le capacità:** limiti quali tool può usare (es. read-only).
- **Si riusano:** quelli in `~/claude/agents/` valgono per tutti i progetti.
- **Specializzano il comportamento:** system prompt focalizzato su un dominio.
- **Controllano i costi:** instrada i task semplici su modelli veloci ed economici come Haiku.

◆ I subagenti built-in

Agente	Tool	Quando Claude lo usa
Explore	Solo lettura (Write/Edit negati)	Cercare e capire il codebase senza modificarlo. Claude specifica la profondità: quick , medium o very thorough . Eredita il modello della conversazione (cap a Opus su API Anthropic). Salta CLAUDE.md per restare veloce ed economico.
Plan	Solo lettura	Ricerca sul codebase durante il plan mode : l'esplorazione resta in una finestra separata mentre la conversazione principale rimane read-only.
general-purpose	Tutti i tool	Task complessi multi-step che richiedono sia esplorazione che modifica.

◆ Crearne uno: file markdown + frontmatter

Un subagente è un file markdown in `./claude/agents/` (progetto) o `~/claude/agents/` (personale). Il modo più rapido: **chiedi a Claude di scriverlo** ("Crea un subagente code-improver in `~/claude/agents/` che... read-only, su Sonnet"). Le modifiche ai file vengono rilevate in pochi secondi senza riavvio.

```
# ~/claude/agents/code-reviewer.md
---
name: code-reviewer
description: Reviewer esperto. Usalo
  proattivamente dopo modifiche al codice.
tools: Read, Glob, Grep
model: sonnet
---

Sei un senior code reviewer. Analizza il
codice e dai feedback specifici e azionabili
su qualità, sicurezza e best practice.
```

Il corpo diventa il **system prompt** del subagente: riceve solo quello più le info base d'ambiente, non l'intero system prompt di Claude Code. La `description` è ciò che Claude usa per decidere quando delegare: scrivila chiara.

Priorità a parità di nome

Managed settings > flag CLI `--agents` (JSON, solo per la sessione) > `./claude/agents/` del progetto > `~/claude/agents/` > plugin. Le directory sono scansate ricorsivamente: puoi organizzare in sottocartelle (`agents/review/`, `agents/research/`), l'identità viene solo dal campo `name`.

Skill vs Subagent, in una frase

Skill = contenuto riusabile che carichi in un contesto (si somma alla tua finestra). **Subagent** = worker isolato che lavora in una finestra separata e ti restituisce solo il sunto. E si combinano: un subagente può precaricare skill (campo `skills:`), una skill può girare in un subagente (`context: fork`).

◆ Frontmatter: i campi che contano

Campo	A cosa serve
name / description	Gli unici obbligatori: identificatore (minuscole e trattini) e criterio di delega.
tools / disallowedTools	Allowlist / denylist di tool. Se omessi eredita tutto. Supportano pattern MCP: <code>mcp__github</code> toglie tutti i tool di quel server, <code>mcp__*</code> tutti gli MCP.
model	<code>sonnet</code> , <code>opus</code> , <code>haiku</code> , <code>fable</code> , un model ID completo, o <code>inherit</code> (default). Il trucco per i costi: ricerca su Haiku, implementazione sul modello grosso.
permissionMode	<code>default</code> , <code>acceptEdits</code> , <code>auto</code> , <code>dontAsk</code> , <code>bypassPermissions</code> , <code>plan</code> : permessi indipendenti dalla sessione madre.
skills	Skill precaricate nel contesto del subagente all'avvio (contenuto completo, non solo la description).
isolation: worktree	Il subagente lavora in un git worktree temporaneo : copia isolata del repo, ripulita automaticamente se non fa modifiche. Zero conflitti con il tuo working tree.
background	<code>true</code> = gira sempre in background (dalle versioni recenti è comunque il default quando possibile: tu continui a lavorare, il risultato arriva).
memory	Scope di memoria persistente (<code>user</code> / <code>project</code> / <code>local</code>): il subagente impara tra le sessioni .
maxTurns / effort / color / hooks	Limite di turni, livello di ragionamento, colore nel transcript, hook scoped al suo ciclo di vita.

Subagent vs Agent team (sperimentale)

I **subagenti** girano dentro la tua sessione e riportano solo al chiamante: costi contenuti, ideali per task focalizzati. Gli **agent team** sono sessioni Claude Code indipendenti che si messaggiano tra loro, con task list condivisa e auto-coordinamento: servono quando i teammates devono confrontarsi (ipotesi in competizione, review parallela, feature con ownership separate), ma ogni teammate è un'istanza Claude completa, quindi costa. Punto di transizione: se i tuoi subagenti paralleli sbattono contro i limiti di contesto o avrebbero bisogno di parlarsi, il passo naturale è il team.

3. Hooks

determinismo: succede sempre, non "se Claude se lo ricorda"

Gli hook sono comandi shell (o richieste HTTP, o prompt) che scattano in punti precisi del ciclo di vita di Claude Code. La differenza filosofica rispetto a tutto il resto: **non dipendono dal giudizio del modello**. Un hook garantisce che una cosa accada sempre — formattare dopo ogni edit, bloccare il tocco di certi file, notificare quando serve input — invece di sperare che l'LLM scelga di farlo. Per decisioni che richiedono giudizio esistono anche hook **prompt-based** e **agent-based** che usano un modello per valutare la condizione.

Si configurano nel blocco `"hooks"` di un file settings (`~/ .claude/settings.json` personale, `.claude/settings.json` di progetto). Il menu `/hooks` è in sola lettura: per modificarli editi il JSON o lo chiedi a Claude.

◆ Gli eventi principali

Evento	Quando scatta
SessionStart	All'avvio o ripresa della sessione (matcher <code>compact</code> = dopo ogni compattazione). Lo stdout viene aggiunto al contesto di Claude.
UserPromptSubmit	Quando invii un prompt, prima che Claude lo processi.
PreToolUse	Prima dell'esecuzione di un tool. Può bloccarla (exit code 2).
PermissionRequest	Quando sta per apparire un dialogo di permesso: può rispondere al posto tuo con JSON <code>{"behavior": "allow"}</code> .
PostToolUse / PostToolUseFailure	Dopo che un tool è riuscito / fallito.
Notification	Quando Claude Code manda una notifica (matcher: <code>permission_prompt</code> , <code>idle_prompt</code> ...).
Stop / SubagentStart / SubagentStop	Quando Claude finisce di rispondere / un subagente parte o termina.
ConfigChange / FileChanged / CwdChanged	Quando cambiano file di configurazione, file osservati su disco, o la working directory (es. ricaricare <code>direnv</code>).

Tutti gli hook corrispondenti a un evento girano **in parallelo**; i comandi identici vengono deduplicati. L'input arriva come JSON su stdin (comodo da leggere con `jq`).

◆ Le tre ricette che usano tutti

```
// 1. Auto-format dopo ogni edit
// .claude/settings.json (nel progetto)
{
  "hooks": {
    "PostToolUse": [{
      "matcher": "Edit|Write",
      "hooks": [{
        "type": "command",
        "command": "jq -r '.tool_input.file_path'
                    | xargs npx prettier --write"
      }]
    }]
  }
}
```

```
// 2. Notifica desktop quando serve input
// ~/.claude/settings.json (Linux)
{
  "hooks": {
    "Notification": [{
      "matcher": "",
      "hooks": [{
        "type": "command",
        "command": "notify-send 'Claude Code'
                    'Serve la tua attenzione'"
      }]
    }]
  }
}
```

```
# 3. Proteggere file sensibili
# .claude/hooks/protect-files.sh
# (registrato su PreToolUse, matcher Edit|Write)
#!/bin/bash
INPUT=$(cat)
FILE_PATH=$(echo "$INPUT" | jq -r \
  '.tool_input.file_path // empty')

PROTECTED=(".env" "package-lock.json" ".git/")

for pattern in "${PROTECTED[@]%%}"; do
  if [[ "$FILE_PATH" == *"$pattern"* ]]; then
    echo "Bloccato: $FILE_PATH" >&2
    exit 2 # exit 2 = blocca l'azione;
          # Claude riceve il motivo e
          # adatta l'approccio
  fi
done
exit 0
```

Il contratto in breve

- `exit 0` = tutto ok, si prosegue.
- `exit 2` = **blocca** l'azione; lo stderr torna a Claude come spiegazione.
- Su **SessionStart** lo stdout viene iniettato nel contesto: perfetto per re-iniettare convenzioni dopo un `/compact` (matcher `compact`) o l'output di `git log --oneline -5`.
- Su **PermissionRequest**, matcher **stretti**: un matcher vuoto o `.*` auto-approverebbe tutto, incluse scritture e shell.

4. Plugins

il livello di
packaging

Un plugin impacchetta **skill + subagenti + hook + server MCP** in una singola unità installabile, distribuita come repo Git (o zip/URL). Le skill dei plugin sono namespaced (`/mio-plugin:review`), quindi più plugin convivono senza conflitti. È la risposta al trigger "un secondo repository ha bisogno dello stesso setup": invece di copiare file `.claude/` in giro, installi.

Come si usano

- `/plugin` apre il menu; sottocomandi diretti: `list`, `install`, `enable`, `disable`.
- I **marketplace** distribuiscono raccolte di plugin: Anthropic ne ospita uno ufficiale, e team/community ne pubblicano altri (repo Git pubblici).
- `/reload-plugins` applica le modifiche senza riavviare.
- Per motivi di sicurezza, i subagenti dei plugin ignorano `hooks`, `mcpServers` e `permissionMode`: se ti servono, copia il file in `.claude/agents/`.
- Chicca: aggiungi un `.claude-plugin/plugin.json` a una cartella skill e quella skill diventa un plugin capace di includere agenti, hook e MCP.

Vet prima di fidarti

Un plugin è **codice di terzi che gira nel tuo ambiente**: hook che eseguono shell, MCP che parlano con l'esterno. Prima di installare da marketplace di community, leggi cosa contiene (skill, hook e configurazioni sono file di testo nel repo del plugin: si ispezionano). Per i team, il plugin "di casa" committato e revisionato come codice è il pattern più solido.

5. Combinazioni che valgono oro

quando i pezzi si parlano

Ricetta	Come e perché
Skill che gira in un subagente	<code>context: fork</code> (+ <code>agent:</code> per scegliere il tipo) nel frontmatter della skill: il workflow parte con /nome ma lavora in contesto isolato e ti torna solo il risultato. Ideale per procedure lunghe e rumorose.
Subagente con skill precaricate	Campo <code>skills:</code> nel frontmatter dell'agente: il worker nasce già con le tue convenzioni API o la checklist di review in pancia, contenuto completo, senza doverle scoprire.
Subagente economico per l'esplorazione	Un agente <code>Explore</code> custom con <code>model: haiku</code> sovrascrive il built-in e tiene la ricerca sul modello più economico, mentre la conversazione principale resta su quello potente.
Hook + skill di sicurezza	Hook <code>PreToolUse</code> che blocca i file sensibili (deterministico) + skill di review invocabile (giudizio): cintura e bretelle.
Deploy "a prova di entusiasmo"	Skill <code>/deploy</code> con <code>disable-model-invocation: true</code> (parte solo se la lanci tu) + <code>allowed-tools</code> ristretti + iniezione <code>!`git status`</code> per decidere sul reale.
MCP + skill che lo usa bene	L'MCP dà a Claude i tool (es. il database); la skill gli insegna schema e pattern di query del tuo team. Tool + conoscenza d'uso.
Tutto insieme → plugin	Quando la combinazione funziona, impacchettala: skill + agenti + hook + MCP in un plugin installabile su ogni repo del team.

Fonti: code.claude.com/docs — features-overview, skills, sub-agents, hooks-guide, plugins · Le funzionalità evolvono in fretta: /release-notes è la verità della tua versione.