ALESSIO SFERRO / GUIDES

Vim / Neovim

Da zero a produttivo

Il terminale come officina.
Editing, codice e strumenti
per chi ama capire come funzionano.

```
:help your-next-idea
```

AUTORE: OPENAI GPT-5 (CODEX)

Idea e test personali: Alessio Sferro

23 CAPITOLI / LABORATORI / SOLUZIONI

EDIZIONE AMPLIATA 2026 - GRATUITA

La mappa del viaggio

Un percorso da leggere in ordine, un manuale a cui tornare. Le voci sono cliccabili; i numeri corrispondono alle pagine del PDF.

IL PATTO DI LETTURA

Prima di premere un tasto

Autore: OpenAI GPT-5, tramite Codex. Il testo, gli esempi e l'impaginazione di questo manuale sono stati realizzati dall'assistente AI. Alessio Sferro ha fornito l'idea e ha testato personalmente il materiale. La voce professionale e appassionata adottata nel libro è una scelta narrativa: non rappresenta esperienze lavorative vissute dal modello.

Il terminale ha una qualità rara: ti lascia vedere le giunture. Un file è un file, un processo ha un nome, un errore ha una riga. Vim vive benissimo in questo ambiente. Non cerca di nascondere tutto dietro un pulsante; ti offre un linguaggio per intervenire sul testo. All'inizio quel linguaggio sembra una serratura. Poi capisci la combinazione, e la serratura diventa una maniglia.

Questo manuale parte da zero e arriva al lavoro quotidiano su un progetto: navigare, modificare, cercare, lanciare test, leggere un diff, diagnosticare una configurazione. Non promette di esaurire ogni opzione dell'editor. Promette un percorso completo per lavorarci con criterio e per sapere dove cercare quando il problema cambia.

Il tono è quello di una bottega: si apre il cofano, si fanno esperimenti, si guarda il risultato. Essere hacker, qui, significa capire abbastanza uno strumento da poterlo piegare a un problema e riparare quando qualcosa si rompe. Vale per il servizio che mantieni al lavoro, per uno script domestico e per un progetto iniziato una domenica sera. La bravura non sta nel numero di finestre nere aperte.

Come leggere e come provare

Tieni l'editor aperto accanto al libro. I capitoli 1-8 costruiscono la grammatica; 9-15 la portano nel progetto; 16-20 affrontano configurazione, strumenti semantici e manutenzione. Seguono laboratori con soluzioni, un percorso di quattro settimane e un prontuario. Puoi saltare alle ricette, ma torna alle basi quando una sequenza ti sembra magia.

Le sequenze come `ciw` si digitano in Normal mode. Nei blocchi contrassegnati **TASTI**, `<Esc>` significa premere Escape, non scriverne il nome. `<C-r>` significa tenere Control e premere r; `<CR>` significa Invio. Gli spazi fra comandi in una spiegazione separano passi, non sono necessariamente tasti. I comandi che iniziano con `:` si digitano nell'editor e si confermano con Invio. Nei blocchi **SHELL**, invece, sei nel terminale.

La base comune usa Vim 9.x e Neovim. La configurazione Lua richiede Neovim; il capitolo LSP usa le API disponibili da Neovim 0.11. Le caratteristiche dipendono anche dalla build, dal sistema operativo e dalla configurazione: controlla `:version` e la documentazione installata. Non serve installare l'ultima versione solo per imparare `ciw`.

Il patto di lavoro

Prova gli esercizi in una cartella dedicata. Prima di un intervento ampio, conosci lo stato Git, salva ciò che vuoi conservare e prepara una verifica. Non eseguire un filtro esterno su un file importante solo perché il comando è elegante. L'eleganza migliore è sapere che cosa succederà.

Regola di bottega: prima rendi una modifica comprensibile, poi ripetibile, infine veloce.

CAPITOLO 01

Entrare, scrivere, uscire

La prima sessione deve finire con un file salvato e la sensazione di sapere dove sei. Non con una distribuzione di plugin e tre ore di configurazione. Apri una cartella per gli esperimenti e avvia l'editor senza la tua configurazione: avrai un punto di riferimento che puoi ritrovare in qualunque momento.

SHELL

```
mkdir -p vim-lab
cd vim-lab
vim --clean appunti.txt
# In alternativa:
nvim --clean appunti.txt
```

`--clean` riduce le interferenze della configurazione personale. Non significa che Vim e Neovim avranno ogni default identico: significa che gli esercizi non dipendono dai tuoi plugin. Per il tutorial integrato usa `vimtutor` dalla shell o `:Tutor` in Neovim, se disponibile nella tua installazione.

Il primo giro completo

Premi `i`, scrivi una frase, premi `<Esc>`. Sei passato da Normal a Insert e sei tornato. Digita `:w` e premi Invio: il contenuto viene scritto su disco. `:q` chiude la finestra corrente; se era l'ultima, termina l'editor. `:wq` scrive e chiude. Se Vim rifiuta `:q` perché il buffer è modificato, sta proteggendo un lavoro non salvato.

`:q!` abbandona le modifiche non salvate del buffer quando questo viene scaricato; non è il modo ordinario di uscire. Con più finestre sullo stesso buffer la situazione è più articolata. Per il momento lavora con un solo file e scegli consapevolmente fra salvare e scartare. `:qa` tenta di chiudere tutte le finestre; `:qa!` può perdere lavoro in più buffer. Il punto esclamativo non è una decorazione.

La modalità è parte del comando

In Normal, `x` elimina un carattere. In Insert, scrive una x. In una riga di comando dopo `:`, fa parte di un comando Ex. Quando qualcosa sembra assurdo, la prima domanda è «in che modo sono?». Escape riporta normalmente a Normal; dal terminale integrato useremo una sequenza diversa, spiegata più avanti.

Per inserire testo, `i` entra prima del cursore e `a` dopo. `I` va al primo carattere non vuoto della riga; `A` alla fine. `o` apre una riga sotto e `O` sopra. Non devi ricordarli tutti oggi: usa `i`, `A` e `o` in tre piccole modifiche e guarda quanto lavoro risparmiano rispetto a raggiungere il punto carattere per carattere.

Quando perdi l'orientamento

Se compare una richiesta di completare un operatore, premi Escape. Se hai aperto per errore la finestra della cronologia con `q:`, chiudila con `:q`. Se il terminale si blocca dopo Control-s, alcuni ambienti hanno ancora il controllo di flusso attivo: Control-q può sbloccarlo. Non attribuire automaticamente all'editor un comportamento del terminale.

Prova. Scrivi tre righe. Aggiungine una in mezzo con `o`. Annulla con `u`, ripristina con `<C-r>`, salva e riapri il file. Il traguardo è riconoscere questi passaggi senza panico, non completarli in un tempo record.

CAPITOLO 02

I modi sono una conversazione

In molti editor stai quasi sempre scrivendo e tieni premuti modificatori per fare altro. Vim inverte la prospettiva: stai spesso descrivendo un'azione, poi entri brevemente a scrivere. Normal non è un parcheggio fra due sessioni di Insert. È il posto in cui lavori sul testo già esistente, che in un progetto reale è spesso la parte maggiore del lavoro.

Immagina di correggere `timeout = 30` in `timeout = 60`. Puoi entrare in Insert e usare Backspace. Puoi anche mettere il cursore sul numero e dire `ciw60<Esc>`. Il vantaggio non sono i millisecondi: hai definito una trasformazione completa, che potrai riutilizzare con il punto.

Selezionare quando serve

`v` seleziona caratteri, `V` righe intere, `<C-v>` un rettangolo. Muovi il cursore per estendere la selezione e applica un operatore: `d` elimina, `y` copia, `>` indenta. Con `o` in Visual cambi l'estremità attiva della selezione; `gv` riseleziona l'ultima area visuale. Sono dettagli preziosi quando hai selezionato bene quasi tutto e non vuoi ricominciare.

Visual è ottimo quando vuoi vedere un intervallo prima di cambiarlo. I text object diventeranno più convenienti quando l'intervallo ha un nome: parola, stringa, paragrafo. Non trasformare questa distinzione in una gara. In una review condivisa, una selezione visibile può essere più comunicativa di un comando breve.

Le modifiche piccole si capiscono meglio

In Insert puoi completare una frase intera. Non occorre premere Escape dopo ogni parola. Cerca invece un confine naturale: una riga di codice, un argomento, il corpo di un messaggio. Tornare in Normal a quel confine crea spesso un'unità di undo e di ripetizione utile. Spostamenti e mapping possono alterare il raggruppamento dell'undo: osserva il risultato anziché assumere che una sessione di Insert sia sempre un unico blocco.

`r` sostituisce un solo carattere senza lasciarti in Insert: `r;` corregge una punteggiatura. `R` entra in Replace e sovrascrive durante la digitazione; è un utensile diverso, utile in campi a larghezza fissa ma sorprendente nel codice. `s` cambia il carattere corrente e passa in Insert; `S` cambia la riga, preservando l'indentazione secondo le opzioni attive.

Prova. In una riga con `status = "draft"`, cambia solo il contenuto della stringa con `ci"ready<Esc>`. Poi annulla e seleziona lo stesso contenuto con `vi"`. Devi vedere il medesimo oggetto attraverso due interfacce: una per agire, una per ispezionare.

Il modo giusto è quello che rende esplicita la tua intenzione. Se non sai descrivere la selezione, guardala prima di automatizzarla.

CAPITOLO 03

Muoversi per destinazioni

La mano impara presto `h`, `j`, `k`, `l`. Il rischio è usarli come quattro frecce costose: venti pressioni per raggiungere una riga che ha già un nome, un numero o un frammento riconoscibile. Tienili per gli aggiustamenti. Per viaggiare usa una destinazione.

`w` raggiunge l'inizio della parola successiva, `e` la fine, `b` torna indietro. Le varianti `W`, `E`, `B` trattano come WORD una sequenza separata da spazi: utili per percorsi e token con punteggiatura. La nozione di word dipende da `iskeyword`, perciò `foo-bar` può comportarsi diversamente da ciò che immagini. Quando hai dubbi, prova su una riga piccola e leggi `:help word`.

Dentro una riga

`0` significa colonna iniziale; `^` primo carattere non vuoto; `$` fine. Con `f,` raggiungi la prossima virgola e con `t,` ti fermi prima. `F` e `T` cercano indietro. `;` ripete la ricerca di carattere, `,` la ripete nel verso opposto. Questi movimenti rimangono sulla riga: per cercare altrove useremo `/`.

INPUT / OUTPUT

```
send_message(user, subject, body)
```

Dal primo carattere della riga, `f(` raggiunge la parentesi. `f,` raggiunge la prima virgola; `;` la seconda. È più robusto che contare diciassette colonne. Ma `f,` non conosce il linguaggio: se un argomento contiene una stringa con una virgola, la troverà comunque.

Nel file e sullo schermo

`gg` va alla prima riga, `G` all'ultima, `42G` alla 42. `}` e `{` attraversano paragrafi delimitati da righe vuote, non necessariamente funzioni. `%` salta fra parentesi corrispondenti quando trova un elemento riconosciuto; plugin e configurazione possono estenderlo. Per la struttura semantica del programma arriveranno i language server.

`<C-d>` e `<C-u>` scorrono circa mezza finestra. `zz` centra la riga corrente, `zt` la porta in alto, `zb` in basso. Se una lunga riga è spezzata solo a video, `gj` e `gk` seguono le righe visualizzate, mentre `j` e `k` seguono quelle del file. Distinguere testo e rappresentazione evita molti movimenti apparentemente strani.

I conteggi sono una comodità, non una prova di aritmetica: `5j`, `3w`, `2f,`. La numerazione relativa, `:set relativenumber number`, aiuta a scegliere un salto locale. Quando il bersaglio è un nome, cercalo; quando è a cinque righe, conta.

Prova. Apri un file di cinquanta righe. Raggiungi una funzione con `/nome`, centra con `zz`, salta alla chiusura con `%`, torna all'inizio con `gg`. Ripeti usando solo ciò che rende riconoscibile ogni destinazione. Stai costruendo una mappa, non allenando un dito a tenere premuto j.

CAPITOLO 04

Parlare la grammatica dell'editing

Un operatore dice che cosa fare; un movimento o un text object dice su quale testo. `d` elimina, `c` cambia e apre Insert, `y` copia. Una volta imparato un nuovo oggetto, tutti questi verbi diventano più potenti. È il motivo per cui Vim continua a restituire valore anche dopo anni.

`dw` elimina fino all'inizio della parola successiva. `de` arriva alla fine della parola. `d$` elimina fino a fine riga; `D` è una scorciatoia per questa operazione. `c$`, o `C`, cambia quel tratto e passa in Insert. Non sono sinonimi perfetti di «cancella la parola»: inclusività, spazi e posizione del cursore fanno parte del risultato.

Inside e around

Con il cursore in `customer_name`, `ciw` cambia l'intera parola, anche se sei in mezzo. `diw` elimina la parola; `daw` comprende anche spaziatura adiacente secondo le regole dell'oggetto. Su `name = "Ada Lovelace"`, `ci"Grace Hopper<Esc>` cambia il contenuto delle virgolette senza eliminarle. `ca"` include anche le virgolette. Lo stesso principio vale per `i)`, `a)`, `i]`, `a]`, `i{`, `a{`.

`ip` seleziona il paragrafo interno, `ap` comprende la separazione; `it` e `at` lavorano sui tag supportati. Questi oggetti non sono un parser universale: stringhe con escape e strutture non bilanciate meritano un controllo visivo. Il comando corto non ti esonera dal leggere il testo.

Conta il lavoro, non i tasti

`dd` elimina una riga, `yy` la copia, `cc` la cambia. Raddoppiare molti operatori li applica alla riga. `3dd` agisce su tre righe. In `d2w`, il conteggio riguarda il movimento; in `2d3w` i conteggi si moltiplicano. Conoscere questa regola è utile; usarla per rendere indecifrabile una modifica non lo è.

`>` e `<` spostano l'indentazione; `=` la ricalcola secondo le regole configurate per il file. `=ip` può essere comodo, ma non sostituisce un formatter del linguaggio. `gUiw` rende maiuscola una parola, `guiw` minuscola; `g~iw` ne inverte il caso. La struttura resta la stessa: verbo e oggetto.

Un refactoring minuscolo

PYTHON

```
def greet(person):
    return "Hello, " + person
```

Metti il cursore dentro `Hello,` e usa `ci"Ciao, <Esc>`. La stringa cambia, le virgolette restano. Poi scegli `person` con una ricerca, non con una raffica di frecce. Se vuoi rinominare un simbolo in tutto il progetto, questo capitolo non basta: una sostituzione testuale non distingue omonimi, commenti e scope. Tratteremo la differenza nel capitolo LSP.

Prova. Descrivi ad alta voce tre modifiche prima di farle: «cambia dentro le virgolette», «copia il paragrafo», «elimina fino alla virgola». Traducile in `ci"`, `yip`, `dt,`. Se la frase è chiara, la sequenza diventa molto meno misteriosa.

CAPITOLO 05

Il punto: progettare una modifica

Il punto ripete l'ultima modifica ripetibile. Sembra un dettaglio, finché non inizi a costruire le modifiche perché siano ripetibili. A quel punto un piccolo refactoring diventa una successione di decisioni: trovo un bersaglio, guardo, applico.

Considera queste righe:

PYTHON

```
status = "draft"
fallback = "draft"
message = "draft pending"
```

Cerca `/draft`, raggiungi il primo risultato e usa `ci"ready<Esc>`. Con `n` vai al risultato successivo e con `.` ripeti. Alla terza riga fermati: lì la trasformazione cambierebbe l'intera frase, non solo la parola. La ripetizione è fedele alla tua modifica, non al significato che speravi avesse.

Una modifica contiene un contratto

`ciwready<Esc>` promette di sostituire una parola. `ci"ready<Esc>` promette di sostituire il contenuto di una stringa. `A;<Esc>` promette di aggiungere un punto e virgola a fine riga. Possono produrre lo stesso risultato su un caso, ma sono programmi diversi. Scegli quello che descrive l'invariante dei bersagli successivi.

Il percorso con cui raggiungi un bersaglio generalmente non fa parte della modifica ripetuta con il punto. È una qualità: puoi cercare, saltare, leggere, poi applicare. Se devi ripetere anche navigazione e più modifiche, serve una macro. Se devi trasformare un insieme di

corrispondenze, forse serve una sostituzione. Non forzare un utensile a fare il lavoro dell'altro.

Ripetere con supervisione

Il ciclo `n.` è un piccolo strumento di review. Non eseguirlo meccanicamente su cento risultati: leggi il contesto. In un file misto con codice, messaggi e documentazione, una revisione manuale locale può essere più sicura di un pattern sofisticato. In altri casi una sostituzione con conferma esprime meglio il problema.

`u` è il compagno del punto. Se la seconda applicazione è sbagliata, annulla subito, controlla la prima e ridefinisci l'operazione. Evita di accumulare altre modifiche prima di capire dove hai sbagliato. Stai affinando un piccolo programma, non eseguendo una coreografia da ricordare.

Prova. Prepara tre righe `alpha`, `beta`, `gamma`. Sulla prima fai `A;<Esc>`, poi `j.` due volte. Riparti dal file iniziale e fai `Iexport <Esc>`, poi `j.` due volte. Confronta la proprietà comune: in un caso la fine della riga, nell'altro il primo carattere non vuoto. Aggiungi indentazione a una riga e verifica che il comando continui a comportarsi come previsto.

Una buona modifica è un piccolo programma con un ingresso chiaro. Il punto è il tuo primo interprete.

CAPITOLO 06

Registri: il banco degli attrezzi

Hai copiato una funzione, cancellato una riga inutile e incollato. Compare la riga cancellata. Non è un bug: in Vim anche molte eliminazioni scrivono nei registri. Conoscere tre registri risolve quasi tutto il disagio iniziale.

Il registro senza nome, `"`, è quello usato normalmente da `p` e `P`. Il registro `0` conserva l'ultimo yank ordinario; molte cancellazioni aggiornano il senza nome ma non `0`. Quindi `"0p` recupera ciò che hai copiato anche dopo una cancellazione. I registri `a-z` sono cassetti scelti da te: `"ayip` copia un paragrafo in a, `"ap` lo incolla.

Copiare non significa sempre clipboard

`y` copia nell'editor. `"+y` usa il registro della clipboard di sistema, se la build o il provider la supportano. In Vim guarda `:version` e `:help clipboard`; in Neovim consulta `:checkhealth` e `:help provider-clipboard`. In SSH la clipboard locale non diventa automaticamente quella remota. In un container potrebbe non esserci alcun servizio grafico.

Per questo il setup del libro non forza `clipboard=unnamedplus`: prima impari che cosa stai copiando e dove. Se lavori su un solo desktop e vuoi quel comportamento, puoi abilitarlo dopo una prova. OSC 52, provider e terminali possono colmare certi scenari remoti, ma compatibilità e autorizzazioni dipendono dall'ambiente. Non affidare un flusso di lavoro a un incolla mai verificato.

Eliminare senza sporcare il banco

Il registro _, detto black hole, scarta il testo: `"_dd` elimina una riga senza sostituire il registro senza nome. È utile prima di un incolla già preparato. `:registers` mostra il contenuto dei registri; `:reg a 0 "` restringe l'ispezione. Non incollare alla cieca quando puoi guardare il cassetto.

`p` mette dopo il cursore o sotto la riga, `P` prima o sopra, a seconda del tipo di registro. Uno yank di riga come `yy` non si comporta come `yiw`, che è per caratteri. Un registro può anche contenere un blocco rettangolare. Queste informazioni accompagnano il testo e spiegano molti incolla apparentemente capricciosi.

In Insert, `<C-r>a` inserisce il contenuto del registro a. La maiuscola quando scrivi un registro nominato aggiunge al contenuto: `"Ayy` accoda una riga ad a. È comodo per raccogliere estratti da più punti senza lasciare il file.

Prova. Copia una riga con `yy`, elimina un'altra con `dd`, poi confronta `p` e `"0p`, annullando fra le prove. Ripeti con `"ayy` e `"ap`. Il tuo obiettivo è sapere in anticipo cosa verrà incollato. La fiducia nel registro è più utile di un incolla spettacolare.

CAPITOLO 07

Undo, recupero e sangue freddo

Un editor potente deve lasciarti sperimentare. `u` annulla, `<C-r>` ripristina. Ma la cronologia di Vim è un albero: se torni indietro e fai una nuova modifica, esiste un ramo alternativo. `g-` e `g+` percorrono stati precedenti e successivi in ordine temporale; `:undolist` offre una vista dei rami. Non confondere questa navigazione con ripetere semplicemente undo e redo.

Puoi chiedere `:earlier 2m` o `:later 2m` per navigare nella cronologia disponibile. Non è un sistema di backup con garanzia temporale: i cambi devono esistere nell'undo e gli intervalli dipendono dal lavoro fatto. Prima di esplorare stati lontani, salva una copia del buffer corrente con `:write copia-di-sicurezza.txt`, scegliendo un nome nuovo.

Tre reti, tre mestieri

Undo gestisce le modifiche nell'editor. Lo swap aiuta a recuperare una sessione interrotta. Git conserva una storia scelta del progetto. Nessuno dei tre sostituisce completamente gli altri. Un file mai aggiunto a Git può ancora avere undo; un processo interrotto può avere uno swap; un commit rimane utile anche dopo che la sessione è finita.

Con `undofile` puoi conservare la cronologia fra sessioni, se percorso e permessi sono corretti. Lo configureremo in una directory dedicata. Ricorda che undo e swap possono contenere porzioni di testo eliminato: per file con segreti applica la stessa cura che riservi al file originale. Non includerli nel repository.

La schermata dello swap

Se Vim segnala uno swap esistente, leggi il nome del file, il proprietario, il processo indicato e la data. Potrebbe esserci un'altra sessione viva. Eliminare lo swap per togliere il messaggio è un modo rapido per cancellare l'indizio che ti serviva.

Su una copia di lavoro puoi usare `vim -r percorso/file` oppure `:recover`. Dopo il recupero, scrivi su un nome nuovo e confronta con il file su disco. Rimuovi uno swap vecchio soltanto dopo avere verificato che nessuna sessione lo usa e che il contenuto utile è salvo. La procedura completa è in `:help recovery`.

Una via d'uscita leggibile

Se vuoi abbandonare tutte le modifiche del buffer e rileggere il file, `:edit!` lo fa, ma è distruttivo rispetto al lavoro non salvato. Prima chiediti se volevi soltanto annullare l'ultima operazione. Non usare `!` come calmante.

Prova. In un file di laboratorio scrivi A, salva, aggiungi B, annulla B e aggiungi C. Esplora con `g-` e `g+`. Poi torna allo stato voluto e salva. Capire un ramo in un esempio da tre righe è molto meglio che scoprirlo durante un refactoring di trecento.

CAPITOLO 08

Cercare come un investigatore

La ricerca è navigazione. In un file grande il nome di una funzione è un indirizzo più stabile della sua riga. `/pattern` cerca in avanti, `?pattern` indietro. `n` ripete nella direzione della ricerca, `N` in quella opposta. Dopo una ricerca iniziata con `?`, quindi, n continua all'indietro.

`*` cerca la parola sotto il cursore come parola intera, `#` nel verso opposto. `g*` non impone gli stessi confini. Un simbolo può avere occorrenze in stringhe o commenti: la ricerca testuale non sa a quale variabile appartengano. Usala per esplorare, non per dedurre che tutti i match siano equivalenti.

Controllare il pattern

Con `:set incsearch hlsearch` vedi il risultato mentre digiti e l'evidenziazione dopo la ricerca. `:nohlsearch` spegne l'evidenziazione corrente senza cancellare il pattern. `ignorecase` e `smartcase` rendono le ricerche in minuscolo insensibili al caso, lasciando significativa una maiuscola digitata nel pattern; `\c` e `\C` forzano esplicitamente il comportamento.

La sintassi delle espressioni regolari di Vim non coincide in tutto con quella di JavaScript o ripgrep. `\v` attiva very magic e rende molti operatori più simili alle regex che potresti conoscere. `\V` usa very nomagic: è utile per un testo quasi letterale, ma backslash e delimitatore della ricerca richiedono ancora attenzione. Non chiamarlo «escape universale».

COMANDI EX

```
/\<timeout\>
/\Vapi.example.test\/v1
/\v(error|warning)
```

Nel secondo esempio lo slash interno è scritto come `\/` perché uno slash non protetto terminerebbe il pattern nella riga di ricerca. Very nomagic semplifica il pattern, ma non elimina il ruolo del delimitatore. È un dettaglio da ricordare ogni volta che cerchi un percorso.

La cronologia è memoria esterna

Con `/` e le frecce recuperi ricerche precedenti. `q/` apre una finestra in cui puoi editarle come testo; Invio esegue quella scelta e `:q` chiude la finestra. Lo stesso principio vale per `q:` e i comandi Ex. Una ricerca complessa merita una riga modificabile, non una gara di Backspace.

Prima di una sostituzione, cerca. Se il pattern trova troppo, riducilo. Se non trova niente, controlla maiuscole, delimitatori e caratteri speciali. Poi usa la stessa idea in una trasformazione. La ricerca separa due responsabilità: scegliere il bersaglio e modificarlo.

Prova. Crea righe con `timeout`, `timeout_ms`, `Timeout` e `"timeout"`. Confronta `/timeout`, `/\<timeout\>` e `/\CTimeout`. Annota quali occorrenze vorresti davvero rinominare nel codice. Quella differenza è il confine fra testo e semantica.

CAPITOLO 09

Sostituzioni e comandi Ex

Hai trovato dodici riferimenti a un vecchio nome. Ora devi decidere se cambiarli uno alla volta, per riga o in tutto il file. Ex è il linguaggio dei comandi dopo `:`. La sua forza sta negli intervalli: prima scegli dove agire, poi dici che cosa fare.

`:s/old/new/` sostituisce il primo match della riga corrente; con `g` agisce su tutti i match della riga. `:%s/old/new/gc` estende l'operazione a tutto il buffer e chiede conferma. Durante la conferma, y accetta, n salta, q termina, a accetta anche i successivi. Il flag `g` non significa «tutti i file»: riguarda le occorrenze su ogni riga dell'intervallo.

Scegliere il perimetro

COMANDI EX

```
:10,20s/old/new/gc
:.,$s/old/new/gc
:%s/\<old_name\>/new_name/gc
```

Il primo comando lavora sulle righe 10-20; il secondo dalla riga corrente alla fine. `%` equivale all'intero buffer. Puoi selezionare righe con V e premere `:`: apparirà l'intervallo `'<,'>`. Una selezione visuale di caratteri, usata così, produce comunque un intervallo di righe; per limitarsi ai caratteri selezionati esiste `\%V`. È una distinzione da conoscere prima di lanciare un comando su una selezione apparentemente precisa.

Nel testo sostitutivo `&` indica il match completo. Per inserire una e commerciale letterale usa `\&`. Per riferirti ai gruppi catturati usa `\1`, `\2` e così via. Un separatore diverso da `/` rende più leggibili i percorsi:

COMANDI EX

```
:%s#api/v1#api/v2#gc
:%s/\v^(\w+): (\d+)$/\2 - \1/
```

L'ultimo esempio trasforma `jobs: 12` in `12 - jobs`, soltanto se l'intera riga ha quella forma. Il pattern è un contratto: se il file contiene trattini o nomi con spazi, non promette di gestirli.

Global: prima guarda, poi esegui

`:g/pattern/comando` applica un comando alle righe che corrispondono; `:v/pattern/comando` a quelle che non corrispondono. Prova prima `:g/^DEBUG/p`: stampa le righe candidate. Solo dopo valuta `:g/^DEBUG/d`, che le elimina. La lettera g di global è un comando Ex, diversa dal flag g di substitute.

`:%s/\s\+$//e` rimuove spazi a fine riga; e evita l'errore quando non c'è un match. Non farlo indiscriminatamente: due spazi finali in Markdown possono rappresentare un'interruzione di riga. Un formatter non conosce automaticamente il contratto del tuo documento.

Prova. Trasforma tre righe `name: numero` con la sostituzione a gruppi. Aggiungi una riga fuori formato e verifica che resti invariata. Poi controlla il diff. Il miglior pattern è spesso quello che rifiuta un caso ambiguo invece di «sistemarlo» a sorpresa.

CAPITOLO 10

Macro che sanno dove andare

Una macro registra tasti in un registro. Non registra la tua intenzione. Se premi tre volte l per raggiungere una virgola, la macro ricorderà tre spostamenti, non «vai alla virgola». Il lavoro interessante consiste nel costruire movimenti che restano veri su tutti i dati.

Partiamo da tre righe:

INPUT / OUTPUT

```
alpha
beta
gamma
```

Vogliamo ottenere tre stringhe Python con una virgola finale. Parti dalla prima riga e digita la sequenza seguente, in cui `<Esc>` indica il tasto Escape:

TASTI

```
qaI"<Esc>A",<Esc>jq
2@a
```

`qa` registra in a. `I` raggiunge l'inizio utile della riga e inserisce la virgoletta; `A` aggiunge la chiusura e la virgola; j posiziona sulla prossima riga; q chiude la registrazione. La prima riga è già trasformata durante la registrazione, quindi rimangono due applicazioni. `@@` ripete l'ultima macro eseguita.

Il punto di arrivo conta quanto quello di partenza

Una macro buona ha una precondizione e una postcondizione. Qui parte su una riga dati e termina sulla seguente. Se ci sono righe vuote in mezzo, la macro le trasformerà in stringhe vuote. Se sei già all'ultima riga, il movimento può fallire e interrompere la sequenza. Non usare `999@a` come sostituto del capire i confini.

Per lavorare solo su righe note puoi selezionarle e applicare una macro che **non** contenga j:

COMANDI EX

```
:'<,'>normal! @a
```

In quel caso Ex porta già il cursore su ogni riga dell'intervallo. Mescolare quell'iterazione con un avanzamento interno rende il ragionamento più difficile. Registra una seconda macro apposta e provala su due righe prima di usarla su duecento. `normal!` evita i mapping per i comandi Normal direttamente interpretati; verifica comunque macro e ambiente, soprattutto se richiamano altri comandi.

Ispezionare una macro

`:reg a` mostra il registro. `"ap` ne incolla il contenuto come testo; alcuni tasti speciali non saranno belli da leggere. Per aggiungere tasti alla registrazione usa `qA`, con la maiuscola. Per una macro breve e sbagliata, registrarla da capo è spesso più chiaro che modificare una rappresentazione piena di caratteri di controllo.

Prova. Ripeti il laboratorio con nomi di lunghezze diverse e con spazi iniziali. Poi inserisci una riga vuota. Decidi se saltarla o trattarla come dato: entrambe le scelte sono valide se sono esplicite. Quando il file è un vero CSV con quoting ed escape, passa a un parser. Vim è un ottimo banco da lavoro, non un motivo per reinventare ogni utensile.

CAPITOLO 11

Buffer, finestre e tab page

Un buffer contiene testo in memoria e può essere associato a un file. Una finestra è una vista su un buffer. Una tab page raccoglie un layout di finestre. La confusione nasce quando cerchi di far coincidere questi tre oggetti con le tab di un browser.

Puoi mostrare lo stesso buffer in due finestre, una sul codice e una sulle sue chiamate più in basso. Puoi avere dieci buffer e una sola finestra. Puoi chiudere una finestra senza perdere il buffer. Quando il modello diventa chiaro, smetti di aprire file solo per «tenerli visibili».

COMANDI EX

```
:edit src/report.py
:ls
:buffer report
:bnext
:bprevious
:vsplit tests/test_report.py
```

<C-^> passa al buffer alternativo in molti ambienti; `:buffer #` è una forma facile da leggere e da digitare quando la tastiera rende scomodo quel tasto. `:set hidden` permette di lasciare un buffer modificato senza doverlo salvare subito quando non ha più finestre. Il lavoro resta in memoria: prima di uscire controlla i buffer modificati con `:ls`.

Due viste che collaborano

`:split` divide orizzontalmente, `:vsplit` verticalmente. <C-w>h, j, k, l cambiano finestra. <C-w>= distribuisce lo spazio; `:resize 12` regola l'altezza della finestra corrente. `:close` chiude la vista, `:only` tenta di lasciare soltanto quella corrente. Evita varianti con ! finché non hai verificato i buffer modificati.

`:bdelete` rimuove un buffer dall'elenco ordinario e può cambiare il layout; non è il sinonimo di chiudere una tab del browser. Se vuoi soltanto più spazio, chiudi una finestra. Se vuoi scaricare un file dalla sessione, allora considera il buffer.

Tab come scrivanie

Una tab page può ospitare implementazione e test; un'altra un confronto fra due file. Apri con `:tabnew`, passa con `gt` e `gT`, osserva con `:tabs`. Non crearne una per ogni file per abitudine. Prova un solo layout finché non sai quale secondo contesto vorresti conservare.

La directory corrente influisce su `:edit`, ricerca e comandi esterni. `:pwd` la mostra; `:cd` la cambia globalmente, `:lcd` per una finestra. Quando una ricerca «non trova niente», verifica prima la radice del progetto. Il problema potrebbe non essere il pattern.

Prova. Apri un file in due split. Modifica una riga e osserva l'altra vista: il buffer è uno. Poi apri un secondo file in una delle viste e usa `:ls`. Devi poter disegnare la relazione fra buffer e finestre su un foglio; tutto il resto diventa più semplice.

CAPITOLO 12

Navigare in un progetto

Dentro un progetto, l'editor deve aiutarti a recuperare contesto. Hai letto una funzione, seguito una chiamata, aperto un test e dimenticato dove eri partito. Una buona navigazione conserva il percorso di ritorno.

I jump registrano molti salti significativi. `<C-o>` torna a un punto precedente, `<C-i>` avanza; `:jumps` mostra la lista. Tab e Control-i possono essere indistinguibili in certi terminali: prima di rimappare Tab, controlla cosa stai sacrificando. La changelist è diversa: `g;` torna a una posizione modificata, `g,` va avanti, `:changes` le elenca.

Segnalibri a costo quasi zero

`ma` crea il mark a. `'a` raggiunge la riga, a `la posizione precisa. I mark minuscoli sono locali al buffer; le maiuscole possono aiutare a ritrovare posizioni in altri file. :marks` permette di ispezionarli. Non servono cinquanta nomi: a può essere l'implementazione che stai leggendo, b il test che la descrive.

Per aprire un percorso sotto il cursore c'è `gf`, purché il testo e le opzioni di ricerca siano adatti. `:find` cerca usando l'opzione `path`. In un repository piccolo puoi provare `:set path+=**` e `:find report.py`; in un monorepo con dipendenze e artefatti generati, una ricerca ricorsiva indiscriminata può diventare costosa. Parti da directory mirate, non dall'intero disco.

Leggere prima di cambiare

Quando entri in una base di codice nuova, cerca il punto d'ingresso, un test e una funzione che trasforma i dati. Tieni visibili soltanto i due contesti che stai confrontando. Un file explorer è una vista dell'albero; non è l'unico modo di navigare. Cercare una frase di errore spesso porta più vicino al bug del percorrere sei cartelle.

I tag offrono un indice di simboli creato da strumenti come Universal Ctags. Con un file tags appropriato, `<C-]>` salta a un tag e `<C-t>` torna indietro. L'indice va generato e aggiornato; il comando `ctags` di un sistema potrebbe essere un'implementazione diversa. Controlla `ctags --version` e non assumere che ogni installazione supporti le stesse opzioni.

Un language server aggiunge conoscenza del linguaggio; i tag e la ricerca restano utili in ambienti semplici, su server remoti o quando il progetto non si indicizza. Avere più vie d'accesso è libertà pratica, non nostalgia.

Prova. Metti un mark nel punto che stai studiando, apri un secondo file, esegui una ricerca e usa la jumplist per tornare. Poi confronta con il ritorno al mark. La prima segue il viaggio; il secondo salta a una destinazione scelta. Sono due memorie diverse.

CAPITOLO 13

Quickfix: una lista di lavoro

Quickfix è una delle idee più utili dell'editor: un elenco di posizioni con file, riga, colonna e messaggio. La ricerca nel progetto, gli errori di compilazione e i risultati di un tool possono diventare la stessa interfaccia. Una volta caricata la lista, il modo di attraversarla non cambia.

COMANDI EX

```
:vimgrep /TODO/j src/**/*.py
:copen
:cnext
:cprevious
:cfirst
:clast
```

`vimgrep` usa i pattern di Vim e legge i file indicati. Il flag j evita il salto immediato al primo match. Senza g trovi al massimo una corrispondenza per riga; con g puoi raccoglierne più di una. Verifica che il glob corrisponda ai file attesi e parti da una directory piccola.

Portare ripgrep nell'editor

Se `rg` è installato e raggiungibile dal PATH, puoi impostare la ricerca esterna così:

COMANDI EX

```
:set grepprg=rg\ --vimgrep\ --smart-case
:set grepformat=%f:%l:%c:%m
:grep TODO src
:copen
```

`--vimgrep` produce righe con le coordinate necessarie. La ricerca usa adesso la sintassi regex di ripgrep, non quella di Vim. Le opzioni

smart-case dei due strumenti sono configurazioni distinte. Nomi di file o pattern con caratteri di shell richiedono quoting appropriato; prova prima ricerche semplici come questa. Un pattern complesso non dovrebbe essere anche il tuo primo esperimento con l'escaping della shell.

`:colder` e `:cnewer` attraversano liste precedenti e successive nella cronologia quickfix. Le location list sono simili ma associate alla finestra: `:lopen`, `:lnext`, `:lprevious`. Possono ospitare risultati locali senza rimpiazzare la lista globale con cui stai seguendo un problema.

Lavorare su più file, con criterio

`:cdo` esegue un comando per ogni voce della lista; `:cfdo` per ogni file presente. Se un file ha cinque match e vuoi una sostituzione sull'intero buffer, cfdo evita cinque passate sullo stesso file. Prima ispeziona l'elenco e restringilo. Una lista di ricerca è un insieme di candidati, non un'approvazione automatica.

COMANDI EX

```
:cfdo %s/\<old_name\>/new_name/gce | update
```

Il comando conferma le sostituzioni e scrive i buffer modificati con `update`. La q alla domanda di conferma interrompe la sostituzione nel buffer corrente; un ciclo esterno può proseguire su altri file. Per interrompere un'operazione lunga usa Control-c e controlla quali file sono già stati scritti. Non esiste una transazione globale con rollback automatico.

Prova. Crea due file di laboratorio, carica i match in quickfix e visita ogni risultato prima di modificare. Fai prima un intervento manuale, poi uno con cfdo. Confronta il diff finale. Il valore di quickfix è rendere esplicita una coda di lavoro, non soltanto accelerare una sostituzione.

CAPITOLO 14

Il terminale è parte dell'editor

Un programma Unix ben fatto trasforma un input in un output. Vim può diventare il punto in cui prepari l'input e leggi l'output. Questa composizione è una delle ragioni per cui un editor nato decenni fa rimane interessante.

`:!comando` esegue un comando esterno. `:read !comando` inserisce il suo output nel buffer dopo la riga corrente. `:[intervallo]!comando` sostituisce l'intervallo con il risultato del filtro. Sono tre contratti diversi: eseguire, importare, trasformare.

COMANDI EX

```
:!git status --short
:read !date
:'<,'>!sort
```

L'ultimo comando ordina le righe selezionate attraverso il programma esterno sort. Vim ha anche `:sort`, che non avvia la shell, e `:sort u` per ordinare eliminando duplicati. Prima di scegliere un filtro chiediti se l'editor offre già l'operazione. Prima di ordinare del codice chiediti se l'ordine è semanticamente importante.

Un formatter è un programma con condizioni

Su una copia di un JSON valido puoi provare `:%!python3 -m json.tool`. Se l'input è invalido, un filtro può fallire e lasciare un risultato inatteso. Salva, controlla il messaggio e usa u se necessario. Non agganciare un filtro al salvataggio finché non sai come gestisce gli errori.

Il codice aperto nell'editor non viene automaticamente eseguito dal filtro come programma, ma il comando che digiti nella shell sì. Evita di costruire stringhe di shell concatenando nomi di file non controllati. Per

automazioni Lua preferisci le API che ricevono una lista di argomenti, quando disponibili, e verifica la versione richiesta.

Un terminale dentro una finestra

In Neovim `:terminal` apre un terminal buffer; premi i per inviare tasti al processo. `<C-\><C-n>` torna a Terminal-Normal e ti permette di navigare o cambiare finestra. In Vim, il terminale integrato richiede il supporto della build e il passaggio a Terminal-Normal usa `<C-w>N`. Non sono la stessa scorciatoia.

Un processo nel terminale integrato può continuare a girare mentre guardi un altro buffer. Chiudere forzatamente la finestra o il buffer può interromperlo o farti perdere il suo contesto. Per un server di sviluppo lungo, scegli consapevolmente fra una finestra dell'editor e una sessione tmux esterna.

tmux è un multiplexer, non un prerequisito per Vim. Può tenere editor, test e log in pannelli separati e permettere di scollegarti da una sessione remota. Introducilo quando senti quel bisogno; due sistemi di split imparati nello stesso pomeriggio sono spesso più confusione che potenza.

Prova. Ordina una lista di parole con `:sort`, annulla, ripeti con il filtro esterno. Poi esegui `:read !date`. Spiega che cosa è cambiato nel buffer e che cosa è stato soltanto eseguito fuori. Questa distinzione ti proteggerà quando i comandi saranno più seri.

CAPITOLO 15

Git, diff e lavoro di squadra

Un refactoring non è finito quando il codice sembra più bello nell'editor. È finito quando puoi mostrare un diff comprensibile, spiegare perché il comportamento è corretto e ripetere le verifiche. Vim accelera le mani; Git e i test mettono le modifiche in una storia leggibile.

Prima di iniziare guarda `git status --short`. Distingui il lavoro tuo da modifiche già presenti. Dopo una trasformazione usa `git diff -- percorso/file` per leggere esattamente ciò che hai cambiato. `git diff --check` aiuta a trovare problemi di whitespace, non verifica la correttezza del programma. Separa un cambio funzionale da una riformattazione estesa quando rende la review più chiara.

Il diff dell'editor

Per confrontare due copie locali usa `vimdiff prima.py dopo.py` oppure `nvim -d prima.py dopo.py`. Sono due file reali: assicurati di sapere quale è la copia di riferimento. In diff mode `]c` e `[c` saltano fra i cambiamenti. `:diffupdate` ricalcola il confronto quando necessario.

`:diffget` porta nel buffer corrente una differenza da un altro buffer; `:diffput` invia la differenza corrente all'altro. Non memorizzarli come «accetta» e «rifiuta»: la direzione dipende dalla finestra in cui sei e, con più sorgenti, da quale scegli. Per un merge a tre vie leggi i nomi dei buffer e la documentazione del mergetool prima di muovere contenuto.

La review è una forma di lettura

Un cambiamento di venti righe può nascondere tre decisioni diverse. Cerca prima il comportamento, poi i casi limite e infine la forma. Usa la ricerca per visitare le chiamate, una seconda finestra per i test, i mark per tenere i punti da discutere. Non hai bisogno di un plugin Git per essere metodico, anche se un'integrazione ben scelta può ridurre attrito.

Per creare un commit selettivo, `git add -p` consente di scegliere gli hunk nella shell. Leggi comunque lo staged diff con `git diff --cached`. Il contenuto che vedi nel buffer può essere più recente di quello sul disco, e il contenuto staged può essere una terza versione. Salva prima di confrontare, ma non dare per scontato che salvare significhi aggiungere all'indice.

Quando il progetto è di tutti

Se la tua configurazione converte tab, finali di riga o encoding senza che tu lo sappia, stai producendo rumore per i colleghi. Rispetta le convenzioni del repository e gli strumenti che le applicano. Un formatter di progetto è una decisione condivisa; il tuo gusto per una colonna più stretta non lo è.

Prova. Su un repository di laboratorio modifica una stringa e una riga di documentazione. Salva, leggi il diff, prepara uno stage parziale e controllalo. Riesci a descrivere il commit in una frase concreta? Se no, forse stai mettendo insieme lavori diversi.

CAPITOLO 16

Configurare Vim senza perdere la mappa

Una configurazione personale è codice che mantieni. Ogni riga dovrebbe rispondere a una domanda osservata: «perché questa ricerca distingue le maiuscole?», «dove finisce il mio undo?», «quale plugin ha mappato questo tasto?». Una configurazione piccola è un vantaggio finché rimane adeguata al lavoro, non una religione.

Su sistemi Unix il file classico è `~/.vimrc`. I percorsi possono cambiare su Windows e in installazioni personalizzate: consulta `:help vimrc`. Fai una copia della configurazione esistente prima di sperimentare. Questo esempio usa Vimscript ed è pensato per Vim 9.x in stile compatibile con configurazioni tradizionali.

VIMRC

```
set nocompatible
filetype plugin indent on
syntax enable
set number
set relativenumber
set hidden
set ignorecase
set smartcase
set incsearch
set hlsearch
set splitright
set splitbelow
set expandtab
set shiftwidth=4
set softtabstop=4
set tabstop=4
let mapleader = ' '
nnoremap <leader>w :write<CR>
nnoremap <leader>h :nohlsearch<CR>

if has('persistent_undo')
  let s:undo_dir = expand('~/.vim/undo')
  call mkdir(s:undo_dir, 'p', 0700)
  let &undodir = s:undo_dir
  set undofile
endif
```

Le opzioni sugli spazi sono un punto di partenza per gli esempi Python, non un'imposizione al repository. Un ftplugin o una convenzione locale può richiedere tab reali, per esempio nei Makefile. `:verbose setlocal expandtab? shiftwidth?` ti mostra il valore attivo e dove è stato impostato. È più utile di aggiungere un'altra riga in fondo sperando che vinca.

Mapping con un nome e una ragione

La leader dell'esempio è lo spazio. `<leader>w` diventa quindi spazio, poi w in Normal mode. Il comando deve contenere davvero `<leader>` e `:write<CR>`: mappare la sola w distruggerebbe il movimento fondamentale fra parole. `nnoremap` evita la rimappatura ricorsiva del lato destro; questa scelta rende più facile prevedere il comportamento.

`:verbose nmap <leader>w` mostra la mappa e la sua origine. `:scriptnames` elenca gli script caricati. Se il problema scompare con `vim --clean`, aggiungi metà configurazione alla volta finché trovi la sezione responsabile. È una ricerca binaria, ed è molto più efficiente del cambiare cinque plugin insieme.

Prova. Avvia Vim con una copia temporanea del vimrc tramite `vim -u percorso/lab.vim file.txt`. Verifica salvataggio e ricerca. Torna alla tua configurazione solo dopo avere capito ogni riga del laboratorio. Il file di configurazione deve essere un manuale che hai scritto per il te stesso di fra sei mesi.

CAPITOLO 17

Neovim: Lua senza una distribuzione intera

Neovim condivide il linguaggio di editing che hai appena imparato, ma offre una configurazione Lua e API pensate per estendere l'editor. Non devi scegliere fra due tribù: puoi usare Vim su una macchina remota e Neovim sul portatile. Le tue competenze di editing viaggiano con te; i dettagli della configurazione vanno riconosciuti.

Per trovare la directory corretta usa `:lua print(vim.fn.stdpath('config'))`. Su molti sistemi Unix è `~/.config/nvim`; non assumerlo per ogni piattaforma. Neovim carica `init.lua` oppure `init.vim`: evita di mantenere entrambi come configurazioni concorrenti. Conserva una copia del setup esistente e prova quello del libro con `nvim -u percorso/init.lua file.py`.

Una base leggibile

INIT.LUA

```
vim.g.mapleader = ' '
vim.g.maplocalleader = ','

vim.opt.number = true
vim.opt.relativenumber = true
vim.opt.hidden = true
vim.opt.ignorecase = true
vim.opt.smartcase = true
vim.opt.incsearch = true
vim.opt.hlsearch = true
vim.opt.splitright = true
vim.opt.splitbelow = true
vim.opt.expandtab = true
vim.opt.shiftwidth = 4
vim.opt.softtabstop = 4
vim.opt.tabstop = 4
vim.opt.scrolloff = 5
vim.opt.signcolumn = 'yes'

local undo = vim.fn.stdpath('state') .. '/undo'
vim.fn.mkdir(undo, 'p', 448)
vim.opt.undodir = undo
vim.opt.undofile = true

vim.keymap.set('n', '<leader>w', '<cmd>write<CR>', {
  desc = 'Salva il buffer',
})
vim.keymap.set('n', '<leader>h', '<cmd>nohlsearch<CR>', {
  desc = 'Spegni evidenziazione ricerca',
})
```

448 è la rappresentazione decimale dei permessi Unix 0700 usata qui dalla funzione mkdir. La directory di stato ospita dati locali, non la configurazione da condividere. Il comportamento dei permessi dipende dalla piattaforma. `vim.opt` gestisce opzioni, `vim.g` variabili globali, `vim.keymap.set` mapping che per default non sono ricorsivi. Le descrizioni non sono decorazione: aiutano ispezione e manutenzione.

Crescere per attrito reale

Quando il file diventa lungo, sposta opzioni, mapping e LSP in moduli dentro `lua/`. Per esempio `lua/config/options.lua` si carica con `require('config.options')`. Non serve una gerarchia di dieci cartelle per trenta righe. La separazione deve rendere più facile trovare il responsabile di un comportamento.

Gli autocomandi richiedono ancora più disciplina perché agiscono senza una pressione esplicita. Un gruppo con `clear = true` rende una configurazione ricaricabile senza accumulare copie dello stesso autocomando:

LUA

```lua
local group = vim.api.nvim_create_augroup('BookYank', {
  clear = true,
})
vim.api.nvim_create_autocmd('TextYankPost', {
  group = group,
  callback = function()
    vim.highlight.on_yank({ timeout = 120 })
  end,
})
```

Questo effetto visivo è facoltativo. Non aggiungere un'azione automatica di formattazione, salvataggio o esecuzione senza poter spiegare quando scatta e come si disattiva. Le sorprese piacevoli al primo giorno diventano incidenti il giorno della consegna.

Prova. Ricarica il file di configurazione due volte e controlla che una pressione su spazio-w salvi una volta sola. Poi esegui `:verbose nmap <leader>w`. Sapere ispezionare ciò che hai configurato fa parte della configurazione stessa.

CAPITOLO 18

LSP: quando il testo diventa un programma

La ricerca trova caratteri. Un language server può comprendere simboli, tipi, riferimenti e diagnostica nel contesto di un progetto. È qui che rinominare `user` può diventare un'operazione semantica invece di una sostituzione su tutte le stringhe che lo contengono. La qualità dipende dal server, dal linguaggio e da quanto correttamente il progetto viene riconosciuto.

Neovim include il client LSP; non per questo include tutti i server. Questo esempio richiede Neovim 0.11 o successivo e il programma `pyright-langserver` nel PATH dell'editor. Pyright è una scelta dimostrativa per Python, non una dipendenza obbligatoria del tuo workflow. Installa il server con il metodo previsto dal tuo ambiente e verifica dalla shell `command -v pyright-langserver` prima di cercare il problema in Lua.

Una configurazione esplicita

Il blocco seguente può vivere in un modulo caricato dall'init.lua. La radice viene individuata usando un marcatore di progetto. Nel laboratorio aggiungeremo un `pyproject.toml`, così non dipenderemo da una cartella Git casuale più in alto.

LSP.LUA

```
if vim.fn.has('nvim-0.11') == 1 then
  vim.lsp.config('book_pyright', {
    cmd = { 'pyright-langserver', '--stdio' },
    filetypes = { 'python' },
    root_markers = { 'pyproject.toml', '.git' },
  })
  vim.lsp.enable('book_pyright')
end

local group = vim.api.nvim_create_augroup('BookLsp', {
  clear = true,
})
vim.api.nvim_create_autocmd('LspAttach', {
  group = group,
  callback = function(event)
    local function map(lhs, rhs, description)
      vim.keymap.set('n', lhs, rhs, {
        buffer = event.buf,
        desc = description,
      })
    end
    map('gd', vim.lsp.buf.definition, 'Definizione')
    map('gr', vim.lsp.buf.references, 'Riferimenti')
    map('K', vim.lsp.buf.hover, 'Documentazione')
    map('<leader>rn', vim.lsp.buf.rename, 'Rinomina')
    map('<leader>ca', vim.lsp.buf.code_action, 'Azioni')
    map('<leader>e', vim.diagnostic.open_float, 'Diagnostica')
  end,
})
```

Le mappe sono locali al buffer collegato a un server. Sono scelte del libro, non un elenco di default universali. Non tutti i server implementano tutte le richieste: un mapping può esistere e una capacità non essere disponibile. Non registrare anche un secondo plugin che avvia lo stesso server senza capire chi possiede la configurazione.

Diagnosticare dal basso

Apri un file .py dentro il progetto e controlla `:set filetype?`, `:pwd` e `:checkhealth vim.lsp`. Con `:lua print(vim.inspect(vim.lsp.get_clients({bufnr=0})))` puoi vedere i client collegati al buffer. Se non c'è un client, verifica eseguibile, filetype e root. Se c'è ma mancano import, controlla interprete e ambiente virtuale secondo la documentazione del server.

La diagnostica non è il test suite. Un type checker può trovare un nome inesistente e non accorgersi che hai usato il prezzo sbagliato. Una rename semantica può comunque coinvolgere molti file: leggi il diff e fai girare i test. «Lo ha fatto il server» non è una spiegazione da portare in review.

Completamento e formattazione

In Insert, `<C-n>` e `<C-p>` completano parole disponibili all'editor anche senza LSP. Il completamento LSP richiede un percorso configurato, per esempio omnifunc o il completamento nativo delle versioni recenti: consulta `:help lsp-completion` per la tua versione. Evita di installare tre motori di completamento per ottenere un menu che non sai diagnosticare.

Per formattare con un server capace di farlo puoi usare `:lua vim.lsp.buf.format({async=false})`. Pyright non è un formatter: per Python scegli uno strumento dedicato se ne hai bisogno. Se più client possono formattare, selezionane uno tramite filtro o nome; la formattazione casuale al salvataggio non è un workflow.

Prova. Introduci intenzionalmente un riferimento a un nome non definito nel progetto di laboratorio. Osserva se il server segnala il problema, correggilo e avvia i test. Poi rinomina una funzione con LSP e controlla tutti i file coinvolti. Se il server non è installato, completa il laboratorio con ricerca e test: è un percorso supportato, non un fallimento.

CAPITOLO 19

Plugin, prestazioni e manutenzione

Un plugin vale il problema che ti risolve ogni giorno. Un fuzzy finder può rendere immediata l'apertura dei file; un'integrazione Git può facilitare lo stage di un hunk; Treesitter può migliorare l'analisi sintattica e alcune operazioni. Nessuno di questi strumenti rende superfluo capire buffer, ricerca e registri.

Treesitter e LSP sono livelli diversi. Un parser sintattico conosce la struttura del testo secondo una grammatica; un language server può risolvere simboli e tipi nel progetto. Evidenziare bene una funzione non significa sapere tutti i suoi riferimenti. Quando una funzionalità smette di funzionare, distinguere il livello riduce molto il numero di ipotesi.

Una scheda per ogni dipendenza

Prima di installare annota tre cose: il problema, il comando equivalente senza plugin e il modo per rimuoverlo. Controlla versione minima dell'editor, manutenzione, licenza e dipendenze esterne. Blocca versioni o commit quando il gestore lo permette, soprattutto su una macchina di lavoro. Aggiorna in un momento in cui puoi leggere i cambiamenti e tornare a una configurazione nota.

Un gestore di plugin è uno strumento per distribuire codice, non un confine di sicurezza. Plugin e configurazioni Lua possono eseguire programmi con i permessi dell'utente. Leggi gli script di bootstrap prima di eseguirli e non attivare configurazioni locali di repository sconosciuti per abitudine. Il terminale ti dà controllo; ti chiede anche di sapere a chi lo stai cedendo.

Misurare il problema giusto

Puoi registrare l'avvio di Neovim con `nvim --startuptime startup.log file.py`. Il log aiuta a capire dove passa il tempo durante l'avvio; non misura automaticamente latenza della digitazione, completamento o costi differiti dopo l'apertura. Per questi problemi osserva il momento preciso: salvataggio, ingresso in Insert, caricamento di un file enorme, primo collegamento del server.

Prova con `nvim --clean` per separare editor e configurazione. Disattiva metà dei componenti sospetti, ripeti lo stesso caso e restringi. Non cambiare contemporaneamente terminale, font, parser e server LSP: avrai quattro spiegazioni possibili e nessuna certezza.

I file grandi sono un'altra disciplina

Un log da centinaia di megabyte può richiedere strumenti a flusso come less, rg, head o tail prima dell'editor. Apri un estratto rilevante, preservando l'originale. Evidenziazione sintattica, parser e scansioni automatiche non sono gratis. Una buona configurazione riconosce anche quando deve fare meno.

Prova. Scrivi l'elenco dei plugin che usi davvero in una giornata. Per ciascuno indica un gesto concreto. Se non trovi quel gesto, disattivalo per una settimana e osserva se ti manca. Non è una competizione di minimalismo: è manutenzione del tuo ambiente di lavoro.

CAPITOLO 20

Scrivere, allineare e leggere altri formati

Vim non è soltanto un posto in cui scrivere codice. Un messaggio di commit, un README e una lista di dati richiedono operazioni diverse, e spesso sono questi file piccoli a far emergere la flessibilità dell'editor.

`gq` formatta testo secondo opzioni come `textwidth` e le regole attive. `gqip` riformatta un paragrafo; prova `:setlocal textwidth=72` in una copia di un messaggio di commit. `J` unisce righe con la spaziatura prevista dall'editor, `gJ` le unisce senza aggiungere quello spazio. Nessuno dei due sa se stai unendo due frasi o due istruzioni che devono restare separate.

Rettangoli e colonne

La selezione a blocco con `<C-v>` è utile per prefissi allineati. Seleziona la prima colonna di tre righe, premi I, scrivi `#` e premi Escape: il prefisso viene applicato alle righe del blocco. Tab e caratteri a larghezza variabile possono rendere meno intuitiva la geometria; verifica il risultato su dati reali. Non trattare un CSV con campi quotati come un rettangolo soltanto perché appare allineato a video.

Per ordinare una selezione usa `:'<,'>sort`. Per rimuovere duplicati con ordinamento, aggiungi u. Se l'ordine originale ha significato, quell'operazione non è equivalente a «tieni la prima occorrenza»: servono altri strumenti o una trasformazione esplicita.

Encoding e finali di riga

`:setlocal fileencoding? fileformat?` mostra come il buffer verrà scritto. Un file DOS usa tipicamente CRLF; un file Unix LF. `:setlocal fileformat=unix` cambia la scelta per la scrittura successiva. Non «normalizzare» un repository senza sapere quale formato richiede. Lo stesso vale per encoding e BOM.

Se vedi caratteri strani, distinguine la causa: byte interpretati male, font senza glifi o sequenze di controllo nel file. Cambiare font non corregge un encoding sbagliato. Riaprire con un encoding esplicito tramite `:edit ++enc=...` richiede di gestire prima le modifiche non salvate; conserva l'originale e consulta `:help ++enc`.

Leggere con meno rumore

I fold nascondono temporaneamente porzioni di testo. Con un metodo manuale puoi selezionare righe e usare zf per creare un fold; zo apre e zc chiude. I metodi basati su indentazione, sintassi o espressioni cambiano chi decide i confini. Prima di accusare il file di essere «sparito», controlla `:setlocal foldmethod? foldenable?`.

Prova. Scrivi un piccolo README con un paragrafo lungo, un elenco e un blocco di codice. Riformatta soltanto il paragrafo con gqip. Il codice deve restare identico. È un esercizio semplice di selezione del perimetro: la stessa disciplina che userai per una sostituzione in produzione.

CAPITOLO 21

Laboratorio: un piccolo strumento vero

Ora mettiamo insieme le parti. Costruiamo un riepilogatore di log in Python: legge righe come `INFO avvio`, conta i livelli e stampa il risultato. Non è un parser per ogni formato di log del mondo. È un programma piccolo con un contratto preciso, abbastanza realistico da obbligarti a navigare fra implementazione, test e input.

Usa una cartella di laboratorio nuova. Crea `logbook.py`, `test_logbook.py` e `pyproject.toml`. Il pyproject può contenere solo il commento `# Progetto di laboratorio Vim`: ci serve come marcatore LSP, non come pacchetto da pubblicare. Gli esempi richiedono Python 3.9 o successivo e non hanno dipendenze esterne.

Implementazione di partenza

LOGBOOK.PY

```
from collections import Counter
import sys


def count_levels(lines):
    counts = Counter()
    for line in lines:
        parts = line.split(maxsplit=1)
        if not parts:
            continue
        level = parts[0].upper()
        counts[level] += 1
    return dict(counts)


def main():
    for level, count in sorted(count_levels(sys.stdin).items()):
        print(f"{level}: {count}")


if __name__ == "__main__":
    main()
```

Il programma normalizza il primo token in maiuscolo, ignora righe vuote e accetta qualsiasi livello. Quest'ultima scelta è intenzionale: il contratto non dice ancora che esistano solo INFO, WARN ed ERROR. Se vuoi limitarli, prima cambia i test e decidi che cosa fare con le righe sconosciute.

TEST_LOGBOOK.PY

```
import unittest
from logbook import count_levels


class CountLevelsTests(unittest.TestCase):
    def test_empty_input(self):
        self.assertEqual(count_levels([]), {})

    def test_counts_and_normalises(self):
        rows = ["INFO avvio", "error rete", "INFO fine"]
        self.assertEqual(
            count_levels(rows), {"INFO": 2, "ERROR": 1}
        )

    def test_skips_blank_lines(self):
        self.assertEqual(count_levels(["", "   "]), {})


if __name__ == "__main__":
    unittest.main()
```

Il ciclo di lavoro

Apri implementazione e test in due split. Cerca `count_levels`, leggi gli usi e metti un mark sul test della normalizzazione. Dalla shell nella cartella del laboratorio esegui:

SHELL

```
python3 -m unittest -v
printf 'INFO avvio\nerror rete\nINFO fine\n' |
  python3 logbook.py
```

L'output del secondo comando deve essere `ERROR: 1` e `INFO: 2`, in quest'ordine, su due righe. L'ordinamento alfabetico rende l'output deterministico. Da Vim puoi lanciare i test con `:!python3 -m unittest -v` dopo aver salvato entrambi i file. Se falliscono, leggi il traceback: non serve configurare un'intera integrazione quickfix per capire un errore da dieci righe.

Una modifica con una ragione

Vuoi chiamare la funzione `summarize_levels`. Cerca prima tutte le occorrenze in questi due file, rinomina con LSP se disponibile oppure con sostituzioni confermate nei file ispezionati. Salva, esegui i test, leggi il diff. Aggiungi poi un test che dimostri che `WARN` e `warn` finiscono nello stesso contatore. Non modificare l'implementazione se soddisfa già la proprietà.

Criterio di completamento. I test passano, l'output CLI coincide con quello atteso, il vecchio nome non compare nei due file e il diff contiene soltanto rename e nuovo test. Se hai modificato l'indentazione di tutto il progetto per sbaglio, il lavoro non è ancora pronto per la review.

CAPITOLO 22

Laboratori brevi, soluzioni spiegate

Ogni esercizio si fa su una copia separata. Le soluzioni sono proposte, non l'unica sequenza valida. Il punto è poter spiegare perché il risultato è corretto. Se trovi una via più chiara per te, confrontala sul medesimo input.

A. Cambiare il valore, non la frase

Input: `label = "temporary value"`. Risultato: `label = "ready"`. Con il cursore dentro le virgolette, usa `ci"ready<Esc>`. L'oggetto i" preserva i delimitatori e sostituisce tutta la stringa. `ciw` cambierebbe solo una parola, lasciando il resto. Annulla e usa `vi"` per vedere esattamente l'oggetto.

B. Salvare uno yank prezioso

Copia la riga `KEEP THIS`, elimina la riga `remove me`, poi incolla il contenuto copiato sotto un'altra riga. Soluzione: `yy`, spostamento, `dd`, spostamento, `"0p`. Il registro 0 conserva lo yank ordinario. Una variante più esplicita è `"ayy` all'inizio e `"ap` alla fine. Controlla `:reg 0 a` per capire entrambe.

C. Rinominare solo una parola completa

Input su tre righe: `old_name`, `old_name_extra`, `old_name`. Vuoi cambiare la prima e la terza in `new_name`. Usa `:%s/\<old_name\>/new_name/gc` e conferma le due occorrenze. L'underscore fa normalmente parte di iskeyword: controlla l'opzione se la seconda riga viene selezionata in un ambiente personalizzato. Il risultato atteso lascia `old_name_extra` invariato.

D. Riordinare una coppia

Input: `jobs: 12`. Risultato: `12 - jobs`. Usa `:s/\v^(\w+): (\d+)$/\2 - \1/`. Le parentesi catturano nome e numero; i riferimenti invertono l'ordine. Gli ancoraggi impediscono di cambiare una porzione di una riga più complessa. Una riga `queued jobs: 12` deve restare invariata: il pattern non ammette spazi nel nome.

E. Trasformare una lista

Input: tre righe `alpha`, `beta`, `gamma`. Risultato: ogni riga diventa una stringa fra doppie virgolette con una virgola finale. Dalla prima riga usa `qaI"<Esc>A",<Esc>jq`, poi `2@a`. La prima trasformazione avviene già mentre registri. Se ottieni quattro righe o modifichi una riga due volte, controlla la posizione iniziale e l'avanzamento della macro.

F. Pulizia mirata di un log

Input: `DEBUG dettaglio`, `INFO avvio`, `DEBUG cache`, `ERROR rete`, una riga per voce. Prima `:g/^DEBUG/p` per ispezionare, poi `:g/^DEBUG/d`. Devono restare soltanto INFO ed ERROR. Il carattere ^ vincola la corrispondenza all'inizio: una riga `INFO testo DEBUG` non deve sparire.

G. Un caso limite per Logbook

Aggiungi un test con `['warn disco', 'WARN rete']`, aspettandoti `{'WARN': 2}`. L'implementazione già normalizza con upper, quindi il test deve passare senza modifiche al programma. Se fallisce, controlla di aver salvato il file e di lanciare il comando nella cartella corretta. Cambiare codice per accontentare un test eseguito contro un'altra copia è una perdita di tempo molto comune.

H. Quando fermarsi

Hai trenta occorrenze di `user`, ma alcune sono campi JSON pubblici e altre variabili locali. Non usare una sostituzione globale come risposta automatica. Ispeziona i riferimenti, definisci il nuovo contratto e separa un rename locale da una migrazione di formato. La soluzione corretta dell'esercizio è riconoscere che la richiesta non contiene abbastanza informazione per un unico comando.

Fermarsi davanti a un'ambiguità è competenza tecnica. Il comando più breve non è sempre quello che completa il lavoro.

CAPITOLO 23

Quattro settimane per farlo diventare tuo

L'entusiasmo iniziale porta facilmente a due errori opposti: cambiare tutto il proprio ambiente in un weekend, oppure rimandare l'uso vero finché non si conoscono tutti i comandi. Funziona meglio un'adozione graduata, su attività abbastanza piccole da poter tornare indietro senza interrompere una consegna.

Settimana uno: orientamento

Dedica venti minuti al giorno a file non critici. Il primo giorno fai un giro completo di apertura, modifica, salvataggio e chiusura. Il secondo usa w, b, e e i confini di riga. Il terzo aggiungi f e t. Il quarto combina d, c e y con oggetti semplici. Il quinto usa ricerca e n. Negli ultimi due giorni ripeti una piccola attività reale: correggere un README o un test.

Annota un solo attrito per sessione: «ho perso il testo copiato», «non so tornare alla funzione», «la ricerca prende troppo». Non annotare «devo diventare più veloce»: è troppo vago per suggerire un esperimento. Se un comando non serve nei tuoi file, puoi impararlo più avanti.

Settimana due: ripetizione controllata

Usa il punto per trasformazioni brevi, i registri per conservare testo e una macro per una lista omogenea. Esegui i laboratori A-F. Per ogni automatismo controlla almeno il primo risultato, un risultato in mezzo e l'ultimo. Se l'input è irregolare, aggiungi un caso irregolare alla prova prima di lanciare la trasformazione completa.

Settimana tre: il progetto

Lavora su Logbook o su un progetto personale equivalente. Apri implementazione e test, usa quickfix per cercare candidati, controlla il diff dopo ogni trasformazione significativa. Configura LSP solo quando riesci già a navigare con ricerca e buffer. Così puoi distinguere il problema nuovo - il server - da quello già risolto - il modo in cui ti muovi.

Settimana quattro: manutenzione

Semplifica la configurazione e documenta i mapping che conservi. Prova ad avviare senza plugin e completa una piccola modifica. Aggiorna una sola dipendenza alla volta. Scegli un'attività professionale a rischio contenuto e svolgila con il nuovo ambiente, mantenendo disponibile quello precedente.

La misura del progresso non è quanti tasti hai eliminato. È quante operazioni completi con fiducia, quante sai annullare, quante sai spiegare a un collega. Se la tua configurazione ti impedisce di lavorare su una macchina pulita, hai allenato soprattutto la configurazione.

Un diario di cinque righe

Alla fine di una sessione scrivi: obiettivo, attrito, comando provato, risultato, prossima verifica. Un esempio: «rinominare una costante; troppi match nelle stringhe; ricerca per parola; migliori candidati ma ancora ambigui; provare rename LSP sul simbolo». È una nota piccola, ma trasforma la pratica da accumulo di scorciatoie a ricerca personale.

Non devi diventare una dimostrazione vivente di Vim. Devi aprire un progetto, capire cosa serve e costruirlo con piacere. Quando le mani seguono il pensiero e puoi riparare il tuo ambiente, hai già ottenuto qualcosa di molto concreto.

APPENDICE A

Prontuario da tenere accanto al terminale

Questo prontuario è un indice operativo. Se una voce è nuova, torna al capitolo corrispondente prima di usarla su file importanti. Le sequenze partono da Normal salvo indicazione diversa; i comandi Ex richiedono Invio.

Iniziare e tornare in controllo

- `i`, `a`, `I`, `A`: inserisci prima, dopo, all'inizio utile, alla fine.
- `o`, `O`: apri una riga sotto o sopra.
- `<Esc>`: torna normalmente a Normal.
- `:w`, `:q`, `:wq`: scrivi, chiudi, scrivi e chiudi.
- `u`, `<C-r>`: annulla e ripristina.
- `g-`, `g+`: esplora gli stati della cronologia per tempo.
- `:ls`: ispeziona i buffer, inclusi quelli modificati.

Muoversi e trasformare

- `w`, `b`, `e`: inizio successivo, inizio precedente, fine parola.
- `0`, `^`, `$`: colonna iniziale, primo non vuoto, fine riga.
- `fX`, `tX`, `;`, `,`: trova X sulla riga, fermati prima, ripeti.
- `gg`, `G`, `42G`: inizio, fine, riga scelta.
- `ciw`, `ci"`, `ci)`: cambia parola, stringa, interno parentesi.
- `daw`, `dip`, `dd`: elimina parola con spazio, paragrafo, riga.
- `yiw`, `yy`, `p`, `P`: copia e incolla secondo il tipo di testo.
- `"0p`, `"ap`, `"_dd`: yank precedente, registro a, scarta riga.

- `.`, `qa...q`, `@a`, `@@`: ripeti modifica, registra ed esegui macro.

Navigare nel lavoro

- `/testo`, `?testo`, `n`, `N`: cerca e attraversa risultati.
- `*`, `#`: cerca la parola sotto il cursore.
- `:nohlsearch`: spegni l'evidenziazione corrente.
- `:vsplit file`, `<C-w>h/j/k/l`: apri e cambia vista.
- `:buffer nome`, `:buffer #`: scegli un buffer o l'alternativo.
- `<C-o>`, `<C-i>`, `:jumps`: torna, avanza, ispeziona i salti.
- `ma`, `'a`: segna e raggiungi la riga a.
- `:copen`, `:cnext`, `:cprevious`: percorri quickfix.
- `:pwd`: verifica da quale directory lavori.

Diagnostica rapida

- `:version`: versione e caratteristiche della build.
- `:verbose setlocal opzione?`: valore e provenienza di un'opzione.
- `:verbose nmap tasti`: provenienza di una mappa Normal.
- `:messages`: messaggi che potresti non avere letto.
- `:scriptnames`: script caricati.
- `:checkhealth`: controlli di ambiente in Neovim.
- `vim --clean` / `nvim --clean`: confronta con una base pulita.

APPENDICE B

Problemi frequenti, ragionamento breve

«Scrivo lettere ma il testo scompare»

Probabilmente sei in Normal e stai eseguendo comandi. Premi Escape, annulla le modifiche indesiderate con u, poi entra in Insert. Se il danno è più grande di quanto pensi, conserva una copia e guarda la cronologia prima di salvare sopra l'originale.

«Incollo il testo sbagliato»

Ispeziona `:registers`. Una cancellazione potrebbe avere aggiornato il registro senza nome. Prova `"0p` per l'ultimo yank ordinario o usa un registro nominato la prossima volta. Se il problema riguarda il sistema operativo, verifica il provider clipboard invece di cambiare il mapping di p.

«L'indentazione cambia da sola»

Controlla `:verbose setlocal expandtab? shiftwidth? indentexpr?`. Filetype plugin, autocomandi e formatter possono essere coinvolti. Individua chi scrive l'opzione e correggi lì. Aggiungere una regola globale può risolvere Python e rompere un Makefile.

«LSP non parte»

Verifica versione di Neovim, eseguibile del server, filetype e root di progetto. Guarda i client del buffer e i messaggi. Un server che parte dalla shell potrebbe non essere nel PATH dell'app che ha avviato Neovim. Un server avviato senza il giusto ambiente può segnalare import mancanti pur essendo tecnicamente collegato.

«La sostituzione non ha cambiato niente»

Cerca prima il pattern con `/`. Controlla l'intervallo, le maiuscole, l'escaping e i confini di parola. Verifica di essere nel buffer giusto. Il flag e può nascondere un mancato match intenzionalmente: durante il debug toglilo per vedere l'errore.

«Una macro funziona solo sulla prima riga»

Controlla la precondizione: stessa struttura dei dati, cursore nello stesso punto logico, registri disponibili, modalità corretta. Sostituisci i movimenti per colonna con confini come `^`, `$`, f e ricerche. Se l'input non ha una struttura regolare, scegli un parser o una procedura supervisionata.

«L'editor è lento»

Descrivi il momento: avvio, apertura, digitazione, ricerca, salvataggio. Ripeti con configurazione pulita. Verifica dimensione del file e costo di parser, autocomandi, formatter e server. Un log di avvio non dimostra che un formatter sia veloce e una CPU tranquilla non esclude attese di rete.

«I test ignorano la mia modifica»

Salva il buffer, controlla `:pwd`, verifica quale interprete o ambiente stai usando e quale file esegue il comando. Nei progetti con build, considera anche artefatti generati e cache. Non correggere un algoritmo finché non hai dimostrato che stai eseguendo quella copia dell'algoritmo.

APPENDICE C

Glossario e documentazione da esplorare

Il vocabolario essenziale

Buffer: testo in memoria, con o senza file associato. **Finestra:** vista di un buffer. **Tab page:** composizione di finestre. **Motion:** movimento utilizzabile anche dopo un operatore. **Text object:** intervallo nominato come una parola o l'interno di una stringa. **Registro:** contenitore di testo o tasti, spesso con un tipo di selezione associato.

Ex: famiglia dei comandi introdotti da due punti. **Quickfix:** lista globale di posizioni e messaggi. **Location list:** lista analoga legata a una finestra. **Mapping:** associazione fra una sequenza di tasti e un'azione. **Autocomando:** azione eseguita quando accade un evento. **Runtime:** script, documentazione e risorse caricati dall'editor.

LSP: protocollo con cui editor e server di linguaggio scambiano richieste e risultati. **Language server:** processo che offre funzioni semantiche. **Parser sintattico:** strumento che riconosce strutture del testo; non coincide con un type checker. **Provider:** integrazione con una capacità esterna, come la clipboard. **Root del progetto:** directory usata come contesto di lavoro da uno strumento.

La documentazione locale è la prima fermata

`:help` apre il manuale della versione che stai usando. Digita `:help` seguito da un argomento e usa il completamento per esplorare. Nella finestra di aiuto, `<C-]>` segue un tag e `<C-t>` torna indietro. `:helpgrep parola` cerca nella documentazione e popola quickfix: puoi applicare al manuale gli strumenti che stai imparando.

Percorso di approfondimento: `:help usr_toc.txt` per il manuale utente, `:help motion.txt` per movimenti e oggetti, `:help registers` per i registri, `:help undo.txt` per la cronologia, `:help recovery` per il recupero, `:help pattern.txt` per i pattern, `:help quickfix.txt` per liste e strumenti esterni. In Neovim aggiungi `:help lua-guide`, `:help lsp`, `:help terminal` e `:help diagnostic`.

Riferimenti primari

Gli esempi e il percorso narrativo di questo manuale sono originali. Le specifiche complete dei comandi appartengono alla documentazione ufficiale: usala per dettagli, eccezioni e differenze fra versioni. Le pagine web possono descrivere una versione più recente della tua.

- Vim, documentazione e manuale: https://www.vim.org/docs.php
- Vim, manuale utente: https://vimhelp.org/usr_toc.txt.html
- Neovim, manuale utente: https://neovim.io/doc/user/
- Neovim, guida Lua: https://neovim.io/doc/user/lua-guide/
- Neovim, client LSP: https://neovim.io/doc/user/lsp/
- Neovim, quickfix: https://neovim.io/doc/user/quickfix/
- Pyright, configurazione e server: https://github.com/microsoft/pyright
- ripgrep, guida e opzioni: https://github.com/BurntSushi/ripgrep
- Git, documentazione: https://git-scm.com/docs
- Python, unittest: https://docs.python.org/3/library/unittest.html

Non serve leggere tutto prima di lavorare. Segui una domanda concreta: perché daw prende quello spazio? Chi imposta quel mapping? Il server riconosce la root? La documentazione diventa interessante quando hai una domanda abbastanza precisa da aprire la pagina giusta.

APPENDICE D

Chiudi il libro, lascia aperto il terminale

All'inizio sembrava necessario ricordare una tastiera intera. Adesso hai una struttura: bersagli, operatori, registri, ripetizioni, liste di lavoro. Hai anche un modo per fermarti, annullare, leggere un diff e ricostruire ciò che è successo. Questo è il patrimonio che rimane quando cambi tema, portatile o versione dell'editor.

Il piacere di Vim non sta soltanto nel fare più in fretta. Sta nella sensazione che un gesto corrisponda a un'idea: cambiare dentro una stringa, visitare gli errori, applicare la stessa trasformazione dove ha senso. Il terminale diventa un'officina quando sai quali utensili collegare, e quando sai lasciarli sul banco.

Il prossimo passo può essere molto piccolo. Apri un file che conosci, trova una modifica utile e completala con attenzione. Poi salva, verifica, chiudi. Non hai bisogno di una cerimonia per usare bene uno strumento.

Autore: OpenAI GPT-5 (Codex). Idea e test personali: Alessio Sferro.

Vim / Neovim - Da zero a produttivo. Edizione ampliata, settembre 2026. Il PDF è distribuito gratuitamente nella sezione Libri di alessiosferro.dev. Il download non richiede registrazione o donazioni.

Prova gli esempi in una cartella dedicata. Poi annota questo libro e tienilo accanto al terminale: è lì che serve.