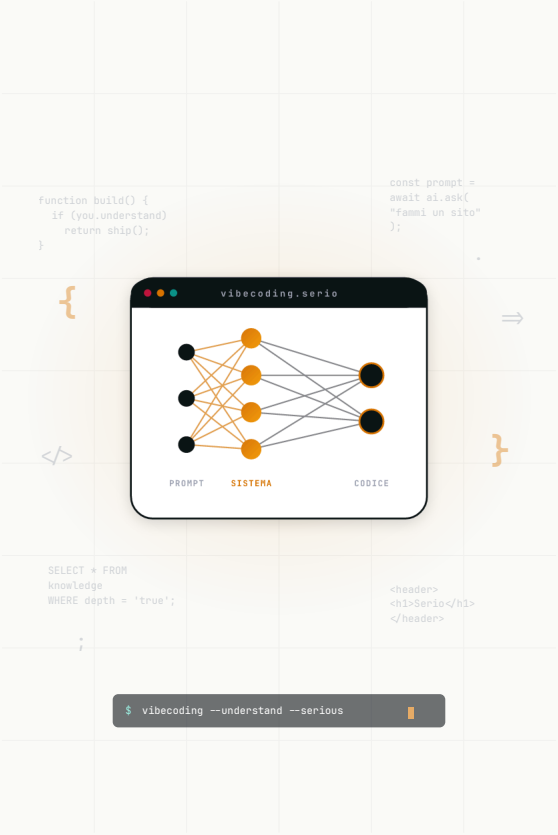
VIBECODING SERIO



Smetti di copiare prompt.

Inizia a capire cosa stai costruendo.

Il manuale per chi costruisce siti con l'AI
ma non capisce cosa sta costruendo.

Lupo Carro

Edizione 1.0 · 2026

{ vibecoding • serio }

Vibecoding Serio

Il manuale per chi costruisce siti con l'AI
ma non capisce cosa sta costruendo.

Lupo Carro

Edizione 1.0 — 2026

Vibecoding Serio
Il manuale per chi costruisce siti con l'AI ma non capisce cosa sta costruendo.

Edizione 1.0 — 2026

Composto in **Source Serif 4** per il testo, **Inter Tight** per i titoli, **JetBrains Mono** per il codice.
Impaginato con Typst.

vibecodingserio.vibecanyon.com
luca.cozza@gmail.com
github.com/cozzagit/vibecoding-serio

Indice

PREFAZIONE

Mettiti comodo

Hai aperto un libro che non avresti dovuto comprare. Sei arrivato qui perché qualcosa, in tutto questo costruire siti con l'AI, non ti torna. Il sito funziona, sì. Ma se ti chiedessero perché funziona, non sapresti rispondere. E quando si rompe — perché si rompe sempre — non sai nemmeno dove guardare.

Questo libro non ti renderà uno sviluppatore senior. Non è il suo scopo. Ti darà i fondamenti che separano un progetto fragile da uno che regge. I concetti, il lessico, le mappe mentali. Quelli che, una volta dentro, non ti escono più dalla testa.

Tre modi di leggerlo:

1. **Lineare** — dall'inizio alla fine, in 8-10 ore in 3-4 sessioni.
1. **Quick Wins** — i 5 capitoli «killer» che risolvono problemi che hai avuto la settimana scorsa.
1. **Riferimento** — glossari e appendici come dizionario quando ti servono.

Inizia come vuoi. Ma una raccomandazione: **non saltare il Modulo 0**. Sembra introduttivo, ma è il vocabolario su cui poggia tutto il resto.

L. C. — primavera 2026

Modulo 0 — Le parole che ti servono per partire

> **SUGGERIMENTO**
>
> 4 capitoli · 14 pagine · 50 minuti di lettura
> Pre-requisiti: nessuno. Hai aperto un browser almeno una volta.

Mettiti comodo

Prima di entrare nel dettaglio di come funziona un sito web, di come parlano un browser e un server, e di tutte quelle cose tecniche che ti spaventano — facciamo una cosa diversa.

Ti insegno **dieci parole**.

Dieci parole che probabilmente hai già sentito. Forse Claude le ha usate ieri in una sua risposta. Forse il tuo amico che si dice «developer» le ha buttate lì in chat. Tu hai annuito, perché annuire è gratis. Ma in realtà non sapevi cosa significassero davvero.

Le dieci parole sono queste:

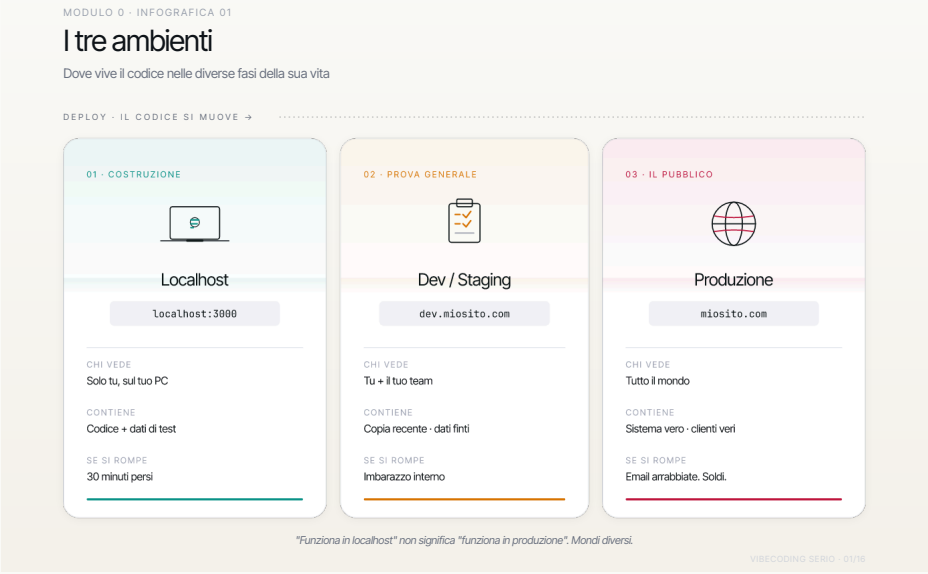
> **SUGGERIMENTO**
>
> deploy · ambiente · localhost · produzione · server · hosting · dominio · DNS · API · framework

Il resto del libro le useremo come fossero ovvie. Quindi sistemiamole adesso, in modo soft, senza fretta. Niente codice. Niente terminale. Solo le parole, le metafore giuste per ricordartele, e qualche esempio della vita di tutti i giorni.

Quando finisci questo modulo, non sarai diventato uno sviluppatore. Ma quando aprirai Claude e gli scriverai il tuo prossimo prompt, avrai cinque parole in più nel cassetto. E ogni parola in più è una iterazione in meno persa a chiarire.

Iniziamo dalla più importante: **dove vive il tuo sito**, e perché questa domanda ha più di una risposta.

0.1 — Cosa significa «deploy», «produzione», «ambiente»



Infografica 01 — I 3 ambienti — localhost, staging, produzione (a fianco di questo capitolo

Il momento «magia»

Hai costruito qualcosa con Bolt o Lovable o Cursor. Sul tuo PC, il sito si apre digitando un indirizzo strano nel browser, qualcosa tipo `http://localhost:3000`. Funziona. Lo vedi tu.

Poi clicchi un bottone — di solito si chiama «Deploy» o «Publish» — e dopo un minuto il sito è raggiungibile da un altro indirizzo. Adesso lo possono aprire i tuoi amici, tua madre, un cliente in Australia, chiunque conosca il link.

Quel «qualcosa è successo» che separa i due momenti ha un nome preciso: **deploy**.

Tre indirizzi, tre mondi

Il tuo sito può vivere in tre posti diversi nello stesso momento. Sembra strano, ma è normale. Si chiamano **ambienti**.

Ambiente 1 — Localhost (il tuo computer)

L'indirizzo è `localhost:3000` o `127.0.0.1:3000`. La parola «localhost» significa letteralmente «questo computer qui». Lo vedi solo tu, perché il sito sta girando sulla tua macchina. Se spegni il PC, il sito sparisce. Se chiudi il programma che lo fa girare, il sito sparisce.

È il tuo **laboratorio**. Qui puoi rompere quello che vuoi, sperimentare, cancellare il database per sbaglio, scrivere codice orribile. Nessuno lo vedrà mai. Quando hai finito di sperimentare, premi «deploy» e mandi solo la versione buona «fuori».

Ambiente 2 — Sviluppo (dev) o staging (server «di prova»)

Un indirizzo tipo `dev.miosito.com` o `staging-miosito.vercel.app`. È un server pubblico — quindi raggiungibile da chiunque conosca l'URL — ma di solito non è linkato da nessuna parte. È invisibile per Google, per il pubblico, per il tuo cliente. Lo usi per fare test «veri» prima di pubblicare la nuova versione.

Pensa al teatro: la prova generale. Tutto è in piedi (luci, scene, attori), ma il pubblico non c'è ancora.

Non tutti i progetti hanno questo ambiente. È utile quando lavori in team o quando il sito è critico (un e-commerce in saldi non vuole rischiare di esplodere durante il Black Friday).

Ambiente 3 — Produzione (prod) (il sito vero, live)

L'indirizzo è quello che dai al cliente. `miosito.it`. È quello che vedono **tutti**. Se si rompe, ricevi messaggi WhatsApp dei clienti alle 22:47.

Il termine «produzione» è preso in prestito dal mondo industriale. Una volta che il prodotto è «in produzione», non è più un prototipo: viene usato dai clienti veri.

«Deploy», spiegato in una riga

> **SUGGERIMENTO**
>
> **Deploy** = l'azione di spostare il codice da un ambiente al suo successivo.
>
> Tipicamente: dal tuo PC (localhost) → al server di sviluppo → al server di produzione.

Quando Vercel o Netlify ti dicono «deploy successful», stanno dicendo: «abbiamo preso il tuo codice, l'abbiamo messo sul server live, adesso il mondo lo vede».

Una tabella per non confonderti più

	Localhost	Dev / Staging	Produzione
URL tipico	`localhost:3000`	`dev.miosito.com`	`miosito.com`
Chi lo vede	Solo tu	Tu + team	Tutti
Cosa contiene	Codice + dati di test	Copia recente + dati finti	Sistema vero + dati clienti
Si rompe? Costa...	30 minuti di tempo	Imbarazzo interno	Email arrabbiate, soldi
Quanto puoi sperimentare	Tutto	Quasi tutto	Niente

Cosa ti porti via da questo capitolo

1. Il tuo sito può vivere in **tre ambienti**: localhost (PC tuo), dev/staging (server di prova), produzione (sito vero).
2. **Localhost mente**: funzionare lì non significa funzionare in produzione. Sono mondi diversi.
3. **Deploy** = spostare il codice da un ambiente all'altro. Il momento in cui il codice «esce di casa».
4. Quando qualcuno ti dice «in produzione non funziona», ti sta dicendo: «in localhost forse sì, ma sul sito vero c'è qualcosa che si comporta

diverso». È una **distinzione importantissima**, e il Modulo 7 ci tornerà a fondo.

0.2 — Server, hosting, dominio, DNS: quattro cose diverse spesso confuse

Infografica 02 — Anatomia di un sito online — quattro entità connesse

«Ho il server da Aruba»

Frase tipica del freelancer che vende siti. Suona competente. In realtà sta dicendo **quattro cose insieme** senza distinguerle. E quando una delle quattro va in crisi (e succede), non sa nemmeno dove guardare.

Smontiamole. Una alla volta.

Server — la macchina

Un **server** è un computer. Punto. Non è magia, non è «il cloud», non è una nuvola.

È un computer come il tuo, ma:

- Sta acceso 24/7
- Sta in una stanza enorme con altri mille computer (il «data center»)
- Non ha schermo, tastiera o mouse — ci si accede solo via rete
- Esegue **il tuo codice** invece che un browser o Word

Quando il tuo sito è «su Aruba», significa che il codice del tuo sito sta girando su uno di quei computer di Aruba.

> **SUGGERIMENTO**
>
> **Server fisico vs virtuale**: oggi quasi mai un server è una macchina fisica dedicata a te. Di solito è una macchina virtuale (VM) — un «computer simulato» che condivide la stessa macchina fisica con altri inquilini. Per te è uguale: lo controlli come fosse tutto tuo.

Hosting — il fornitore

L'**hosting** è chi ti **vende** quel server. Aruba, Vercel, Netlify, AWS, DigitalOcean, Hetzner. Sono tutti hosting provider.

Pensa alla differenza tra una **casa** e l'**agenzia immobiliare** che te l'affitta. La casa è il server. L'agenzia è l'hosting.

Diversi tipi di hosting per diversi bisogni:

- **Hosting condiviso** (Aruba «Hosting Linux», SiteGround): paghi 30€/anno, condividi la macchina con altri 200 siti, comodo per WordPress.
- **VPS — Virtual Private Server** (Aruba VPS, Hetzner): paghi 5-30€/mese, hai una macchina virtuale tutta per te. Più potente, più libertà, ma devi configurarla da solo.
- **PaaS — Platform as a Service** (Vercel, Netlify, Railway): paghi quanto consumi, non vedi mai il server, fai solo `git push` e tutto succede magicamente. Più caro per progetti grossi, comodissimo per cominciare.
- **IaaS — Infrastructure as a Service** (AWS, Google Cloud, Azure): roba enterprise, potentissima, complicata.

> **NOTA ITALIANA**
>
> **Per progetti italiani piccoli/medi**: Aruba VPS o Vercel coprono il 90% dei casi. Aruba se vuoi controllo totale e database/file pesanti. Vercel se vuoi zero pensieri e il sito è «front-end + qualche API».

Dominio — il nome

Il **dominio** è il nome che digiti nel browser. `miosito.it`, `vibecodingserio.vibecanyon.com`, `google.com`.

Si compra **separatamente** dall'hosting. Spesso da fornitori diversi:

- Lo compri da un **registrar** (Namecheap, Aruba domini, GoDaddy, Cloudflare Registrar)
- Lo paghi annualmente (5-50€/anno per i `.it` `.com` `.net` «normali»)
- Quando smetti di pagare, qualcun altro può prenderlo

Il dominio è solo un **nome**. Non contiene il sito. È come il nome sul citofono di casa: dice «qui abita Mario Rossi», ma Mario Rossi non è il citofono — è una persona che vive dentro l'appartamento.

> **SUGGERIMENTO**
>
> **Sottodominio**: quando vedi `dev.miosito.it` o `blog.miosito.it`, quei «dev» e «blog» sono **sottodomini**. Li configuri tu, gratis, una volta che hai il dominio principale. Non li paghi separatamente.

DNS — l'elenco telefonico

Qui arriviamo alla parte che a tutti i vibecoder sembra magia nera. In realtà è semplice.

Quando un utente digita `miosito.it` nel browser, il browser **non sa dove trovare il sito**. Il sito vive in un computer (server) da qualche parte nel mondo, identificato da un numero (l'**indirizzo IP**, tipo `188.213.170.214`). Ma l'utente ha digitato un nome, non un numero.

Come fa il browser a tradurre il nome in numero?

C'è un sistema mondiale che si chiama **DNS** (Domain Name System) che funziona come un **enorme elenco telefonico distribuito**. Tu chiedi «dov'è miosito.it?», il DNS risponde «all'indirizzo 188.213.170.214».

Configurare il DNS significa **scrivere nell'elenco telefonico**:

- «Quando qualcuno chiede `miosito.it`, mandalo a 188.213.170.214»
- «Quando qualcuno chiede `dev.miosito.it`, mandalo a 188.213.170.215»
- E così via.

Si fa nel pannello del registrar (Namecheap, Aruba, Cloudflare). Si chiamano «**record DNS**». I più comuni:

Record	Cosa fa	Esempio
A	Punta un dominio a un indirizzo IP	`miosito.it → 188.213.170.214`
CNAME	Punta un dominio a un altro dominio	`www.miosito.it → miosito.it`
MX	Dice «le email per questo dominio vanno qui»	`miosito.it → mail.aruba.it`
TXT	Note libere (per verifica proprietà, anti-spam)	`v=spf1 include:_spf.google.com`

> **SUGGERIMENTO**
>
> **Quando fai un cambio DNS**, ci vogliono da 5 minuti a 24 ore perché il cambiamento si propaghi in tutto il mondo. È il motivo per cui «ho cambiato il dominio ma il vecchio sito si vede ancora».

Mettiamo tutto insieme con un esempio reale

Vuoi mettere online un sito chiamato `miobellosito.it`. Ecco cosa succede in pratica:

1. **Compri il dominio** `miobellosito.it` su Namecheap → 12€/anno
2. **Compri un VPS** su Aruba → 5€/mese, ti danno l'IP `188.213.170.214`
3. **Carichi il codice** del sito sul VPS (vedi capitolo 0.4)
4. **Configuri il DNS** su Namecheap: aggiungi un record A che dice `miobellosito.it → 188.213.170.214`
5. **Aspetti 30 minuti** per la propagazione DNS
6. Apri `https://miobellosito.it` e il sito è online

Hai usato **quattro entità diverse**: un dominio (Namecheap), un hosting+server (Aruba), e il DNS (configurato su Namecheap, ma è un sistema mondiale). Ognuna fa una cosa diversa.

Cosa ti porti via

1. **Server** = la macchina che esegue il tuo codice.
2. **Hosting** = il fornitore che ti vende il server.
3. **Dominio** = il nome (`miosito.it`). Si compra separatamente.
4. **DNS** = l'elenco telefonico mondiale che traduce il nome nel numero IP del server.
5. Spesso compri queste cose da fornitori **diversi** e devi farle parlare tra loro.
6. Quando un sito «non si vede», quasi sempre il problema è in **uno di questi quattro pezzi** — non nel codice.

0.3 — Cos'è un'API, una libreria, un framework, un pacchetto, un SDK

Infografica 03 — Le scatole di Lego del codice — API, libreria, framework, pacchetto, SDK

Le parole che Claude usa cento volte al giorno

Apri Claude. Gli chiedi di farti un sito. Lui ti risponde con un piano:

> **SUGGERIMENTO**
>
> «Userò il framework Next.js, con la libreria Tailwind per lo stile, il pacchetto next-auth per l'autenticazione, e chiamerò l'API di Stripe tramite il loro SDK ufficiale.»

Tu copi, incolli, premi invio. Funziona. Non hai capito niente di quello che ha detto.

Sistemiamo le cinque parole. È più facile di quello che sembra.

Una metafora: la cucina

Pensa a quando cucini.

- **Una ricetta** è un foglio con istruzioni: «Per fare la pasta al pomodoro, fai bollire l'acqua, aggiungi sale, butta la pasta...». Tu la usi, segui i passi, ottieni il piatto.
- Una **cucina già attrezzata** è un ambiente intero: hai i fornelli, le pentole, gli utensili, e una struttura imposta (devi muoverti tra fornello-piano-lavandino-frigo). Non parti da zero, parti da uno spazio che ha già delle regole.
- Una **bottiglia di salsa già fatta** è un prodotto pronto: la apri, la usi, non ti chiedi come l'hanno fatta. Risparmi tempo.
- Il **menu di un ristorante** è una lista di cose che puoi chiedere e ottenere senza cucinarle tu. Ordini, paghi, ricevi il piatto.
- Un **kit per fare i sushi** è una scatola che contiene tutto il necessario (riso, alga, salsa di soia, bacchette, tappetino) + istruzioni mirate per farti partire subito con un task specifico.

Ognuna di queste cinque cose esiste anche nel mondo del codice. Hanno nomi precisi.

Libreria — la ricetta

Una **libreria** è un pezzo di codice riusabile che fa **una cosa specifica**. Tu la usi per non scrivere quel codice da zero.

Esempi:

- `axios` → libreria per fare chiamate HTTP (parla con altri server)
- `dayjs` → libreria per gestire date e orari
- `lodash` → libreria con tante funzioni utility (raggruppare, ordinare, filtrare)
- `react-icons` → libreria con migliaia di icone già pronte

In codice, una libreria si «importa» e si «usa»:

```
import axios from 'axios';
const response = await axios.get('https://api.example.com');
```

Tu importi `axios`. Lui ti dà la funzione `get` già scritta. Tu la usi. Risparmi 200 righe di codice di rete che avresti dovuto scrivere a mano.

Framework — la cucina già attrezzata

Un **framework** è più grosso di una libreria. Non ti dà una funzione, ti dà **un ambiente intero con regole**.

Esempi:

- **Next.js** → framework per costruire siti e app web in React (ti impone struttura cartelle, routing, server-side rendering)
- **Laravel** → framework PHP per app web
- **Django** → framework Python per app web
- **Rails** → framework Ruby per app web

La differenza con la libreria è importante: con una **libreria** sei tu che scrivi il programma e chiami la libreria quando ti serve. Con un **framework** è il framework che esegue il programma e chiama il tuo codice quando gli serve.

In gergo si dice «un framework ti dice come deve essere fatta la tua app». Una libreria invece te la metti in tasca e la tiri fuori al momento giusto.

> **SUGGERIMENTO**
>
> **Regola pratica**: se la cosa che usi ti dice «metti i tuoi file in `pages/` e scrivi le tue pagine come componenti React», è un framework. Se ti dice «chiama questa funzione quando vuoi fare X», è una libreria.

Pacchetto (package) — la bottiglia di salsa

Un **pacchetto** è il **file zippato** in cui viene distribuita una libreria o un framework.

Quando dici «ho installato il pacchetto axios», stai dicendo: «ho scaricato il file zippato che contiene la libreria axios e l'ho messo nel mio progetto».

I pacchetti si scaricano da **registri di pacchetti**:

- **npm** (Node Package Manager) per JavaScript / TypeScript → migliaia di pacchetti
- **pip** per Python
- **composer** per PHP
- **gem** per Ruby
- **cargo** per Rust

Il comando tipico è `npm install nome-pacchetto` (o `pnpm`, o `yarn` — sono varianti dello stesso strumento).

> **SUGGERIMENTO**
>
> **Distinzione fine ma utile**: `axios` è la **libreria** (il codice). `axios-1.6.0.tgz` su npm è il **pacchetto** (il file). Nella pratica si usano come sinonimi: «ho aggiunto il pacchetto axios» = «ho aggiunto la libreria axios».

API — il menu del ristorante

Una **API** (Application Programming Interface) è una **lista di cose che puoi chiedere a un servizio**, e che il servizio è disposto a farti.

Esempi:

- L'**API di OpenAI**: puoi chiedere «rispondi a questa domanda», «genera questa immagine», «trascrivi questo audio»
- L'**API di Stripe**: puoi chiedere «addebita 29€ a questo cliente», «crea un abbonamento», «manda una fattura»
- L'**API di Google Maps**: puoi chiedere «dammi le coordinate di Via Manzoni 14, Como»

Per usare un'API:

1. Ti registri al servizio
2. Ottieni una **API key** (una password segreta)
3. Mandi richieste HTTP a un certo indirizzo (`https://api.openai.com/v1/...`)
4. Il servizio risponde con il risultato

L'API non è codice che gira sul tuo server. È un **contratto** tra te e qualcun altro: tu mandi una richiesta in un certo formato, loro rispondono in un certo formato. Tu non sai cosa succede in mezzo (e non ti interessa).

> **SUGGERIMENTO**
>
> **Anche tu puoi creare API**. Quando il tuo backend espone delle «rotte» (`/api/users`, `/api/login`), stai creando un'API. Il tuo frontend la chiama. Lo vedremo nel Modulo 5.

SDK — il kit per fare i sushi

Un **SDK** (Software Development Kit) è un **pacchetto che ti dà tutti gli strumenti per usare facilmente un'API specifica**.

Esempio: l'API di Stripe è complessa. Per usarla «a mano» devi scrivere richieste HTTP, gestire autenticazione, parsare risposte JSON. Stripe ha pubblicato un **SDK ufficiale** che ti dà funzioni già pronte:

```
import Stripe from 'stripe';
const stripe = new Stripe('sk_test_...');
const charge = await stripe.charges.create({ amount: 2900,
currency: 'eur' });
```

Tre righe. Tu chiami `stripe.charges.create(...)` e dietro le quinte l'SDK fa tutto il lavoro: chiama l'API di Stripe, gestisce errori, ritorna l'oggetto.

Praticamente, un **SDK è una libreria specializzata** per usare un servizio specifico. Stripe SDK, Supabase SDK, OpenAI SDK, AWS SDK.

Tabella riassuntiva

Termine	Cos'è	Esempio	Lo «usi» così
Pacchetto	Il file zippato che si scarica	`axios-1.6.0.tgz`	`npm install axios`
Libreria	Codice riusabile per UNA cosa	`axios` , `dayjs`	`import axios from 'axios'`
Framework	Codice + struttura imposta	Next.js, Laravel	Costruisci tutto il progetto dentro
API	Lista di cose chiedibili a un servizio	API OpenAI, API Stripe	Mandi richieste HTTP
SDK	Libreria specializzata per usare un'API	Stripe SDK, Supabase SDK	Importi e chiami funzioni pronte

Cosa ti porti via

1. Quando Claude scrive «installa il pacchetto react-icons», sta dicendo «scarica questa libreria di icone».
2. Quando dice «useremo il framework Next.js», sta dicendo «tutto il progetto sarà costruito secondo le regole di Next.js».
3. Quando dice «chiamiamo l'API di OpenAI», sta dicendo «mandiamo una richiesta al loro server e aspettiamo una risposta».
4. Quando dice «usiamo l'SDK di Supabase», sta dicendo «usiamo le funzioni pronte di Supabase per non scrivere chiamate HTTP a mano».
5. Adesso queste parole non ti spaventano più.

0.4 — FTP, SSH, Git: i tre modi per spostare codice

Infografica 04 — Da localhost al server in 3 modi

Una domanda che nessuno ti ha mai spiegato

Hai scritto codice sul tuo PC. Funziona in localhost. Vuoi metterlo «sul server», così che il mondo lo veda. **Come ci arriva, fisicamente, da qui a là?**

Ci sono tre modi storici. Si sono susseguiti nel tempo, e oggi convivono. Ognuno è adatto a situazioni diverse.

FTP — il modo «vecchia scuola»

FTP sta per **File Transfer Protocol**. È nato negli anni "70 e si usa ancora oggi, soprattutto con hosting condivisi (Aruba «Hosting Linux», SiteGround).

Funziona così:

1. Apri un programma chiamato «client FTP» — il più popolare gratis è **FileZilla**
2. Inserisci tre cose: indirizzo del server (es. `ftp.miosito.it`), username, password
3. Vedi due pannelli affiancati: a sinistra i tuoi file locali, a destra i file sul server
4. **Trascini i file** da sinistra a destra. Vengono copiati sul server.

È il modo più «fisico» di mettere file online. Sembra di trascinare cartelle in Esplora Risorse.

Pro:

- Visivo, intuitivo per chi viene da Windows/Mac
- Funziona con qualsiasi hosting
- Gratis (FileZilla)

Contro:

- **Lento** se devi caricare un sito intero (centinaia di file = ore)
- **Insicuro** nella versione classica: la password viaggia in chiaro. Usa sempre **SFTP** (la versione cifrata) se disponibile
- Rischio errori umani: dimenticarti un file, sovrascrivere quello sbagliato
- Niente versioning: se rompi qualcosa, non puoi tornare indietro

Quando lo userai (raramente): WordPress su Aruba Hosting Linux. Caricamento di immagini massicce. Sito legacy in PHP che gira su hosting condiviso.

SSH — il modo «serio»

SSH sta per **Secure Shell**. È un modo per **aprire un terminale remoto** sul server.

Cioè: tu apri un terminale sul tuo PC, scrivi un comando, e quel comando viene eseguito **sul server**, non sul tuo PC. Come se fossi seduto fisicamente davanti al server.

Esempio reale di una sessione SSH:

```
$ ssh root@188.213.170.214        # mi connetto al server
$ cd /var/www/miosito              # entro nella cartella
del sito
$ git pull                         # scarico la versione
aggiornata
$ pnpm build                       # ricostruisco il sito
$ pm2 restart miosito              # riavvio l'applicazione
$ exit                             # esco dal server
```

Cinque comandi. Dieci secondi. Il sito è aggiornato.

Pro:
- **Velocissimo** una volta che ci hai preso la mano
- **Sicuro** (tutto cifrato, autenticazione con chiave invece di password)
- Puoi fare **qualsiasi cosa**: spostare file, modificare configurazioni, riavviare servizi, vedere i log
- Fondamentale per chi gestisce un VPS

Contro:
- **Solo riga di comando**: se hai paura del terminale, devi superarla
- Devi **configurare la chiave SSH** la prima volta (10 minuti di setup)
- Su un click sbagliato puoi rompere tutto (con grande potere viene grande responsabilità)

Quando lo userai (spesso, se hai un VPS): ogni volta che devi fare manutenzione, deploy manuale, troubleshooting di un sito su VPS Aruba, Hetzner, DigitalOcean.

Git push + deploy automatico — il modo «moderno»

Questo è il modo più **comodo** per chi non vuole pensare al server.

Funziona così:
1. Il tuo codice è in un **repository Git** (su GitHub, GitLab, Bitbucket)
2. Il tuo hosting (Vercel, Netlify, Railway, Render) è **collegato al repository**
3. Tu scrivi codice in locale, fai `git commit` e poi `git push`
4. Il servizio di hosting **si accorge automaticamente** del nuovo codice

5. Lo scarica, lo compila, lo mette online — tutto da solo
6. In 30 secondi il sito è aggiornato

Tu non tocchi mai il server. Non sai nemmeno dov'è. Lo gestisce il provider per te.

Pro:
- **Zero touch** dopo setup iniziale
- **Versioning gratis**: se rompi qualcosa, torni alla versione precedente con un click
- **Storia di tutto**: ogni modifica registrata, vedi chi ha cambiato cosa e quando
- **Anteprime gratis**: alcuni servizi (Vercel) ti danno un URL di prova per ogni branch del repo
- **Funziona da qualsiasi PC**: niente da configurare in locale a parte Git

Contro:
- Il **costo cresce** con l'uso (Vercel ti fa pagare per banda, build, ecc.)
- Sei **legato al provider**: se Vercel cambia prezzi o regole, ti adegui
- **Meno controllo**: se vuoi installare un programma custom sul server, non puoi
- Dietro le quinte ci sono ancora server, FTP, SSH — ma sono nascosti

Quando lo userai (probabilmente sempre): Next.js / Astro / SvelteKit su Vercel o Netlify. Progetti front-end. App piccole/medie. Tutto quello che fai con AI generative oggi.

Tabella decisionale: quale usare?

Situazione	Modo consigliato
Hai un sito Next.js / React / Astro statico	**Git push + Vercel/Netlify**
Gestisci un VPS Aruba con più siti, database, custom config	**SSH**
Hai WordPress su Aruba Hosting Linux	**SFTP**
Devi caricare un PDF in una cartella di un sito esistente	**SFTP** o **SSH**, dipende dall'host
Lavori in team su un progetto serio	**Git push + CI/CD**
Stai imparando, vuoi capire cosa succede	**SSH**, almeno una volta

Una nota sulla parola «Git»

Quando ho detto «Git push», ti ho usato un termine che merita un secondo.

Git è un sistema per **tracciare le modifiche al codice** nel tempo. Non è un modo per spostare file: è un modo per **versionare** il tuo progetto.

Ogni volta che fai un cambiamento e dici «questa è una modifica importante», Git la registra come un **commit**. Puoi tornare indietro a qualsiasi commit, vedere chi ha cambiato cosa, lavorare in parallelo con altre persone senza pestarsi i piedi.

GitHub (o GitLab, Bitbucket) è un **sito** che ospita repository Git online, in modo che tu possa condividerli con altri o usare i tuoi PC diversi.

Il «Git push» del modo moderno significa: «manda tutti i miei commit recenti al repository online». Da lì, il provider di hosting li raccoglie e fa il deploy.

> **SUGGERIMENTO**
>
> Approfondimento Git lo facciamo nel **Modulo 8**, quando parleremo di parlare a Claude. Per adesso ti basta sapere che `git push` = «manda il mio codice nel cloud, che poi ci pensa qualcun altro».

Cosa ti porti via

1. **FTP**: trascini file, vecchia scuola, lento. Si usa ancora con hosting condivisi.
2. **SSH**: terminale remoto sul server. Potente, veloce, richiede di non avere paura della riga di comando.
3. **Git push + deploy automatico**: il modo moderno. Tu fai `git push`, il provider fa il resto.
4. Probabilmente userai sempre il **terzo** modo, e ogni tanto il **secondo**.
5. Il primo modo è una conoscenza utile per non sentirti perso quando un cliente ti dice «ho un sito su Aruba con FTP».

Chiusura del Modulo 0

Hai letto il primo modulo. Adesso le **dieci parole** dell'inizio dovrebbero suonarti diverse:

> **SUGGERIMENTO**
>
> deploy · ambiente · localhost · produzione · server · hosting · dominio · DNS · API · framework

Non sei diventato uno sviluppatore. Sei diventato **uno che le parole le riconosce**. È un grosso primo passo: ogni libro tecnico, ogni risposta di Claude, ogni domanda che farai a uno sviluppatore vero adesso parte da una base condivisa.

Nei prossimi moduli costruiamo sopra:

- **Modulo 1** → cosa succede tecnicamente quando premi «deploy»
- **Modulo 2** → la grande divisione tra **client e server**, con tutti i linguaggi che vivono da una parte e dall'altra
- **Modulo 3** → l'**anatomia visuale** di una pagina web (quello che ti aiuta di più a parlare con Claude)

Prima di andare avanti, fai un piccolo test con te stesso.

🎯 Mini-quiz di autovalutazione

Rispondi a queste 5 domande. Se ne sbagli più di 2, rileggi il capitolo corrispondente prima di proseguire.

1. **Tuo amico ti dice: «il sito non funziona in produzione, in localhost sì». Cosa significa?**

> **SUGGERIMENTO**
>
> La risposta è alla fine.

2. **Hai comprato un dominio su Namecheap. Hai un VPS su Aruba. Ti manca un solo passaggio per far funzionare il dominio. Quale?**
3. **Differenza in una frase tra libreria e framework.**
4. **Quando Claude scrive «useremo il pacchetto stripe», cosa intende?**
5. **Hai un sito Next.js da deployare su Vercel. Quale dei tre modi userai (FTP, SSH, Git push)?**

Risposte

1. Significa che il sito è online ma c'è qualcosa che si comporta diversamente rispetto al suo PC. Quasi sempre è una variabile d'ambiente, un problema di CORS, o un database remoto diverso da quello locale. Lo vedremo a fondo nel Quick Win 1.
2. Configurare il **DNS**: aggiungere un record A su Namecheap che dice «manda chi cerca il mio dominio all'indirizzo IP del server Aruba».
3. Una **libreria** è codice che chiami quando ti serve (axios, dayjs). Un **framework** è codice che ti impone una struttura e che chiama il tuo codice (Next.js, Laravel).
4. «Scarica e installa la libreria di Stripe nel tuo progetto», facendo `npm install stripe`. Da lì potrai importarla nel tuo codice e chiamare le sue funzioni per parlare con l'API di Stripe.
5. **Git push**. Vercel è collegato al tuo repo GitHub: tu fai push, lui fa tutto il resto.

> **SUGGERIMENTO**
>
> Se sei arrivato fin qui senza saltare paragrafi, hai investito 50 minuti che ti faranno risparmiare ore di iterazioni con Claude nei prossimi mesi. Adesso prendi un caffè e poi vai al Modulo 1.

Modulo 0 redatto: 2026-04-25 · Versione 1.0 · 14 pagine · 3500 parole

Modulo 1 — Cosa sta succedendo davvero quando premi «Deploy»

SUGGERIMENTO

4 capitoli · 14 pagine · 50 minuti di lettura
Pre-requisiti: Modulo 0 letto. Conosci le parole deploy, ambiente, server, hosting, dominio, DNS, API, framework.

Mettiti comodo (di nuovo)

Adesso che hai le parole, possiamo guardare **come si mettono insieme**.

In questo modulo non ti insegno cose nuove — ti insegno a **vedere il sistema**. Quando un utente apre il tuo sito, succedono molte cose in pochi millisecondi. Tu le vedi solo come «ho premuto un link, è apparsa una pagina». Sotto, c'è una piccola coreografia tra browser, DNS, server, database e CDN.

Capirla cambia tutto. Quando il tuo sito si rompe, smetti di pensare «boh», e cominci a pensare «fammi vedere chi ha smesso di rispondere». È la

differenza tra il vibecoder che chiude il PC alle due di notte e il freelancer che fixa in dieci minuti.

Quattro tappe in questo modulo:

1. La metafora del meccanico (perché ti è utile capire e non solo usare)
2. La mappa del territorio (le 5 entità che compongono ogni sito web)
3. Cosa fa davvero Claude quando gli scrivi un prompt
4. Il viaggio del click in 8 passi (dal momento del click al risultato in pagina)

Andiamo.

1.1 — Il vibecoder e il meccanico

> **SUGGERIMENTO**
>
> Una metafora prima dei tecnicismi.

Due livelli di automobilisti

Ci sono due livelli di chi guida un'automobile.

Livello 1: chi sa guidare. Mette la chiave (o preme un pulsante), accelera, frena, parcheggia. Va dal punto A al punto B. È il 99% delle persone. È un livello onesto, valido. La macchina è uno strumento.

Livello 2: chi sa anche **cosa succede dentro**. Quando si accende la spia del motore, sa se è grave o se può aspettare il prossimo tagliando. Quando frena e sente un cigolio, sa che probabilmente sono le pastiglie consumate. Non è meccanico professionista. Ma quando qualcosa va storto, **non si sente nel panico**.

La differenza che conta

Il vibecoder oggi è al **livello 1** della programmazione web. Chiede a Claude di «fare un sito», riceve qualcosa che funziona, lo mette online. Va dal punto A al punto B.

Funziona finché tutto funziona. Quando si rompe (e si rompe sempre prima o poi), il vibecoder di livello 1 non sa neanche **dove guardare**. Apre Claude, scrive «non funziona», ottiene risposte generiche, prova fix a caso. Otto messaggi dopo, è peggio di prima.

Il **vibecoder serio** — quello che vogliamo diventare in questo libro — non è uno sviluppatore senior. Non scrive codice da zero. Continua a usare l'AI. Ma ha imparato il **livello 2**: quando il sito si rompe, sa **dove guardare** — e quindi sa **cosa chiedere a Claude**.

Cosa significa «capire il sistema»

Capire il sistema non significa saper costruire ogni pezzo da zero. Significa sapere:

- **Cosa** sta succedendo (browser → server → database)
- **Dove** può rompersi (DNS, certificato SSL, CORS, variabile d'ambiente mancante)
- **Quale entità** sta probabilmente avendo problemi quando vedi un certo errore
- **Chi chiamare**: a volte è il provider di hosting, a volte è il registrar del dominio, a volte è una variabile da settare nel pannello

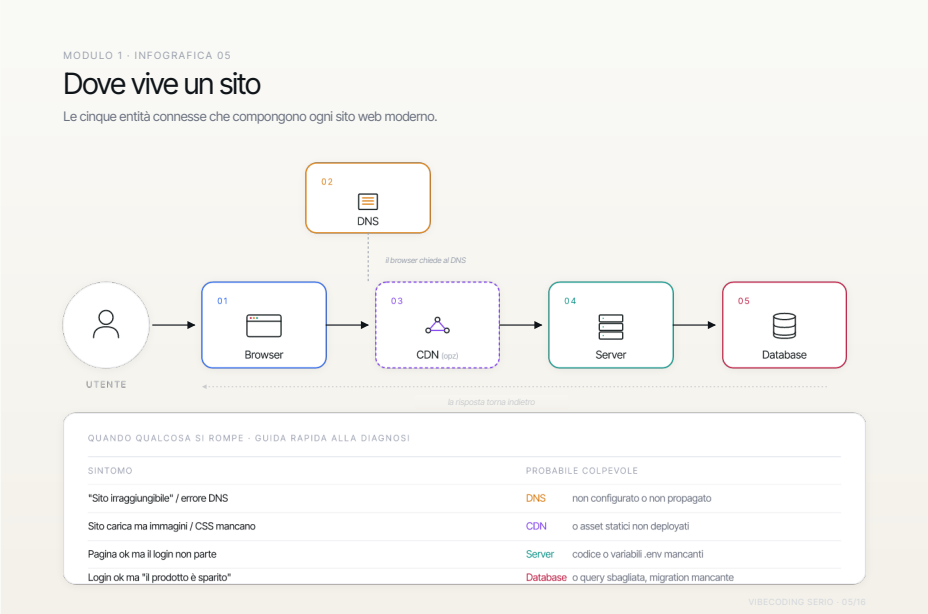
Una volta che hai questa mappa mentale, il «non funziona» smette di essere uno spavento. Diventa una **diagnosi**. E le diagnosi si fanno in dieci minuti, non in dieci ore.

I prossimi tre capitoli ti danno questa mappa.

Cosa ti porti via

1. Esistono due livelli di vibecoder: chi usa e basta, chi capisce abbastanza da diagnosticare.
2. Il livello 2 non è «diventa programmatore senior». È «smetti di andare nel panico quando qualcosa si rompe».
3. La mappa mentale del sistema è quello che separa i due livelli. È quello che costruiamo da qui in poi.

1.2 — La mappa del territorio: dove vive un sito



Infografica 05 — Dove vive un sito — utente, browser, DNS, CDN, server, database

Il sito non è «un posto»

Quando un utente visita `miosito.com`, il sito non vive in un solo posto. Vive in **almeno cinque entità diverse** che lavorano insieme.

Te le mostro tutte. Poi nei prossimi moduli le studieremo una per una.

Le cinque entità

1. **Il browser dell'utente**
(Chrome, Safari, Firefox, Edge)

È il programma che gira sul PC o sul telefono dell'utente. È la «porta d'ingresso» al tuo sito. Esegue il codice frontend (HTML, CSS, JavaScript) che il server gli ha mandato.

Caratteristica importante: il browser è **stupido per design**. Non si fida di niente, applica regole rigide di sicurezza (CORS, cookie httpOnly, HTTPS), e fa il minimo indispensabile da solo.

2. **Il sistema DNS**

L'elenco telefonico mondiale di cui abbiamo parlato nel Modulo 0. Quando l'utente digita `miosito.com`, il browser chiede al DNS dove vive il sito (qual è l'indirizzo IP del server). Il DNS risponde con un numero tipo `188.213.170.214`.

Ci vogliono 20-100 millisecondi. Se il DNS non risponde, il sito non parte nemmeno.

3. **La CDN**

(Content Delivery Network) — opzionale ma diffusissima

La CDN è una **rete di server distribuiti nel mondo**, ognuno con una copia dei file statici del tuo sito (immagini, CSS, JavaScript, font). Servizi come Cloudflare, Cloudfront (AWS), Fastly.

Quando un utente di Tokyo apre il sito, riceve i file dal server CDN più vicino a lui (Tokyo o Seoul) — invece che dal tuo server in Italia. Risparmia tempo: 200ms invece di 800ms.

Non tutti i siti usano una CDN. Vercel e Netlify ne hanno una integrata gratis. Su VPS Aruba la aggiungi tu (Cloudflare gratuito è perfetto).

4. **Il server (la tua applicazione)**

La macchina che esegue il **codice backend** del tuo sito. Quando il browser ha bisogno di dati dinamici (login, lista prodotti, salvataggio form), chiede al server.

Il server riceve la richiesta, **esegue il tuo codice**, magari interroga il database, e risponde.

Sui PaaS (Vercel, Netlify) il server è invisibile a te. Su VPS lo gestisci tu (con SSH, nginx, PM2).

5. **Il database**

Dove vivono i dati persistenti: utenti, ordini, prodotti, sessioni. Postgres, MySQL, MongoDB, Supabase, Firebase.

Il database **non parla mai direttamente col browser**. Solo il server può interrogare il database. Questa regola è fondamentale per la sicurezza: se il

browser potesse interrogare direttamente il DB, chiunque potrebbe rubare i dati di tutti.

Una mappa visuale

Queste cinque entità sono connesse da frecce. Il flusso è sempre lo stesso:

```
    Utente
      │ apre miosito.com
      ▼
   Browser
      │
      ▼
    DNS          (chi mi dà l'IP di miosito.com?)
      │
      ▼
    CDN          (file statici cachati vicino all'utente)
      │
      ▼
   Server        (codice backend, decide cosa fare)
      │
      ▼
  Database      (dati persistenti)
```

Le frecce vanno **a doppio senso** in realtà — il server risponde al browser, il database risponde al server. Ho semplificato per leggibilità.

Quando qualcosa si rompe, dove guardare

Questa mappa è oro. Quando il sito non funziona, percorrila in ordine:

Sintomo	Probabile colpevole
«Sito irraggiungibile» / errore DNS	DNS non configurato o non propagato
Sito si carica ma immagini/css mancano	CDN o asset non deployati
Pagina carica ma login non funziona	Server (codice o variabili d'ambiente)
Login funziona ma «il prodotto è sparito»	Database o query sbagliata

Sintomo	Probabile colpevole
Errore CORS in console	Server (regole CORS errate)
Lentezza solo in certe regioni del mondo	CDN mancante o configurata male

Vedi la differenza? Senza la mappa, dici «non funziona». Con la mappa, dici «il browser non riesce a parlare col server, probabilmente CORS o variabile mancante» — che è già un'ipotesi diagnostica.

Cosa ti porti via

1. Ogni sito web vive in **5 entità connesse**: browser, DNS, CDN (opzionale), server, database.
2. Il flusso va sempre **dall'utente al database e ritorno**.
3. Il **browser non parla mai col database**. Solo il server può interrogare il database. Questa regola ti salva la vita molte volte.
4. Quando qualcosa si rompe, **percorri la catena in ordine** invece di cercare a caso.

1.3 — Cosa fa davvero Claude quando scrivi un prompt

> **SUGGERIMENTO**
>
> Caso studio: il prompt più comune del vibecoder.

Il prompt che hai scritto cento volte

> **SUGGERIMENTO**
>
> «Crea un'app web con login Google e una dashboard utente.»

Lo hai scritto a Claude. Lui ti ha risposto con uno schema e con codice. Tu hai copiato, hai premuto qualche tasto, hai fatto deploy. Funziona.

Cosa ha fatto Claude in concreto? **Ha scritto sei pezzi di codice diversi, ognuno per una delle cinque entità che abbiamo appena visto** + una sesta entità (un servizio esterno). Te li mostro uno per uno.

I sei pezzi

Pezzo 1 — Pagina di login (frontend, gira nel browser)

Una pagina HTML con un bottone «Accedi con Google». Quando l'utente lo clicca, il browser viene reindirizzato a Google.

```
<button onClick={() ⇒ signIn('google')}>Accedi con Google</
button>
```

Banale. Ma è solo l'inizio.

Pezzo 2 — Configurazione OAuth (codice backend, gira sul server)

Il tuo backend deve dire a Google: «Ehi Google, sono il sito miosito.com, voglio che tu autentichi i miei utenti per me. Ecco le mie credenziali (clientid e clientsecret).»

Per fare questo, qualcuno deve essere andato sulla console di Google Cloud, aver creato un'applicazione OAuth, copiato due chiavi, e messe in un file `.env`. Spesso questo passaggio Claude lo dimentica di citarlo, e tu ti chiedi perché il login non parte.

Pezzo 3 — Endpoint di callback (codice backend, gira sul server)

Quando Google ha autenticato l'utente, lo manda di ritorno al tuo sito su un URL specifico, tipo `https://miosito.com/api/auth/callback/google`. Il tuo backend deve avere una rotta che riceve questa chiamata, verifica con Google che sia tutto ok, e crea una sessione per l'utente.

Pezzo 4 — Salvataggio sessione (cookie + database)

Una volta autenticato l'utente, devi **ricordare** che è autenticato. Altrimenti la prossima richiesta non lo riconosce.

Si fa con due strumenti:

- Un **cookie** nel browser (un piccolo testo che il browser re-invia ad ogni richiesta)
- Una **sessione** sul database (una nota che dice «questo cookie corrisponde a questo utente»)

Più tutto quello che Google ha mandato sull'utente: nome, email, foto profilo. Da salvare nel database.

Pezzo 5 — Pagina protetta (frontend + middleware backend)

La dashboard. Prima che il browser la renderizzi, il server deve verificare: «chi sta chiedendo questa pagina è autenticato?» Si guarda il cookie, si controlla la sessione, e se va bene si manda la dashboard. Se no, redirect al login.

Pezzo 6 — Servizio esterno (Google OAuth)

Tutto questo non funziona senza Google. Google è il **sesto pezzo**, ed è una delle cose più importanti da capire: **molti pezzi del tuo sito non sono «sul tuo server»**. Sono altrove, su servizi esterni che tu chiami via API.

Esempi di pezzi che spesso vivono altrove:
- Autenticazione → Auth0, Clerk, Supabase Auth, Google
- Pagamenti → Stripe, PayPal
- Email transazionali → Resend, Postmark, SendGrid
- File storage → Cloudflare R2, AWS S3, Supabase Storage
- AI → OpenAI, Anthropic, Replicate

Ognuno di questi è un pezzo «outsourcing» del tuo sistema. Tu gli mandi richieste API, loro fanno il lavoro complesso, ti rispondono.

Perché te li ho mostrati tutti

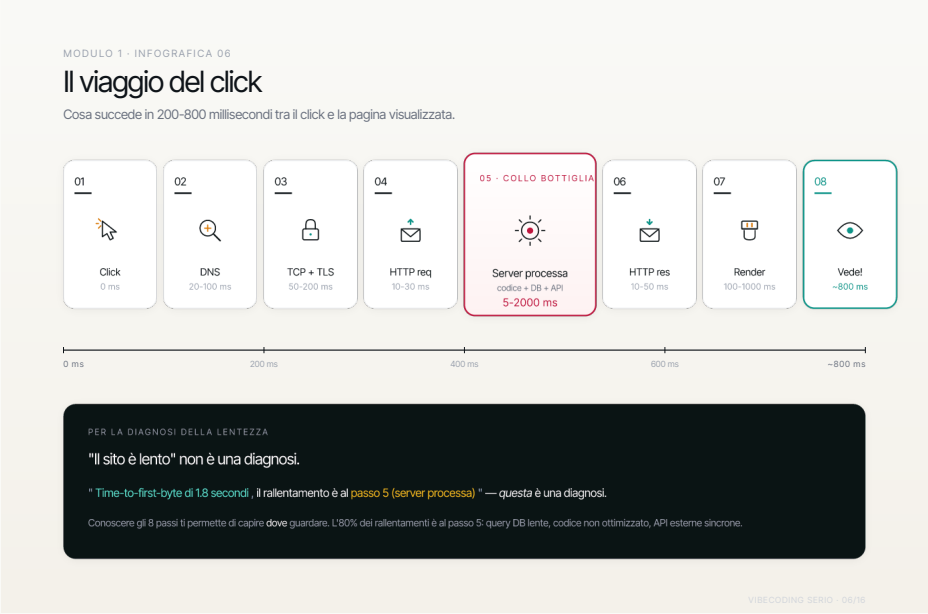
Quando il login Google «non funziona», **uno di questi sei pezzi è rotto**.
- Il bottone non parte → pezzo 1 (frontend)
- Google non riconosce il sito → pezzo 2 (configurazione OAuth)
- Dopo l'autenticazione redirect a una pagina di errore → pezzo 3 (callback URL non configurato in produzione)
- «L'utente è autenticato ma viene sloggato subito» → pezzo 4 (cookie non `Secure`, sessione non salvata)
- «La dashboard non protegge la pagina» → pezzo 5 (middleware mancante)
- «Errore 500 al click su Accedi» → variabili d'ambiente di Google mancanti in produzione

Senza questa mappa, scrivi a Claude «il login Google non funziona» e lui ti propone di provare 8 cose. Con questa mappa, scrivi «al click su Accedi vengo reindirizzato a Google ma poi sul callback ricevo un 401» e Claude ti porta dritto al pezzo 3.

Cosa ti porti via

1. Un prompt apparentemente semplice come «crea un login Google» nasconde **almeno 6 pezzi di codice diversi**.
2. I pezzi vivono in posti diversi: frontend, backend, database, e servizi esterni (Google, Stripe, ecc.).
3. **Buona parte del tuo sito non sta sul tuo server**: vive su servizi esterni che chiami via API.
4. Quando qualcosa «non funziona», il problema è quasi sempre in **uno specifico pezzo** — saperlo isolare riduce le iterazioni con Claude da 8 a 1.

1.4 — Il viaggio del click in 8 passi



Infografica 06 — Il viaggio del click — 8 passi numerati con icone e tempi

Una microstoria di 200 millisecondi

Un utente in un bar di Roma apre il telefono, digita `miosito.com`, preme invio. Dopo circa 200-800 millisecondi vede la pagina. In quei millisecondi succedono **otto cose distinte**.

Te le racconto come fossero scene di un film in slow motion. Le ho numerate.

Passo 1 — Il click

L'utente preme invio nel browser. Il browser vede `miosito.com` e si pone una domanda: «Dove diavolo è miosito.com?»

Lui non lo sa. È un nome, mica un indirizzo. Deve scoprirlo.

⏱️ Tempo: istantaneo

Passo 2 — DNS lookup

Il browser fa una richiesta al **resolver DNS** locale (di solito quello del provider di Internet, o il famoso `8.8.8.8` di Google).

«Hey, dov'è miosito.com?»

Il resolver, se non l'ha già in cache, fa una catena di chiamate ad altri server DNS (root, TLD, authoritative — tutto questo lo vediamo nell'infografica 16, Modulo 7) finché non trova la risposta: «miosito.com → 188.213.170.214».

Il browser adesso ha l'**indirizzo IP**. Sa dove andare.

⏱️ Tempo: 20-100 ms (10-30 ms se in cache)

Passo 3 — Connessione TCP + TLS

Il browser non può semplicemente «spedire» la richiesta. Deve **stabilire una connessione** col server. Si fa in due fasi:

TCP handshake (3 messaggi): «Ciao server, ci sei?» — «Sì, sono qui.» — «Bene, parliamo.» È il modo in cui Internet stabilisce ogni connessione.

TLS handshake (più messaggi): «Vorrei parlare in modo cifrato. Ecco la mia chiave.» — «Ecco il mio certificato firmato dall'autorità X.» — «OK, da adesso parliamo cifrato.» È quello che fa apparire il **lucchetto verde** nella barra del browser. Senza TLS, sarebbe HTTP «puro» e tutti potrebbero leggere i dati che viaggiano.

⏱️ Tempo: 50-200 ms

Passo 4 — HTTP request

Adesso che la connessione è aperta e cifrata, il browser manda finalmente la **richiesta vera**:

```
GET / HTTP/1.1
Host: miosito.com
User-Agent: Mozilla/5.0 ...
Accept: text/html, ...
Cookie: session_id=abc123...
```

Tradotto: «Dammi la pagina principale (/). Ah, e ti mando anche un cookie che dice chi sono.»

⏱️ Tempo: 10-30 ms (è solo l'invio dei dati)

Passo 5 — Server processa

Il server riceve la richiesta. Adesso esegue il tuo codice. A seconda di cosa fa la pagina, può:

- **Leggere file statici** (HTML, CSS, immagini) → veloce, pochi ms
- **Eseguire codice backend** (verificare il cookie, fare una query al database, chiamare un'API esterna) → da 10ms a parecchi secondi
- **Renderizzare HTML** dinamicamente con i dati appena letti dal database

Se il server deve fare 3 query al database e chiamare l'API di Stripe, qui ci vogliono 200-500 ms. Se è solo un file statico, 5 ms.

Questo è di solito il **collo di bottiglia** dei siti lenti. Quando un sito ci mette 4 secondi a caricare, il 95% di quelli sono qui dentro.

⏱️ Tempo: 5-2000 ms (estremamente variabile)

Passo 6 — HTTP response

Il server ha finito. Manda la **risposta** al browser:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Set-Cookie: session_id=xyz789; HttpOnly; Secure

<!DOCTYPE html>
<html>...
```

Tre cose:

- Lo **status code** (200, 404, 500...) — dice se è andata bene
- Gli **headers** (tipo di contenuto, cookie da settare, regole di cache...)
- Il **body** — il contenuto vero, in questo caso l'HTML della pagina

⏱️ Tempo: 10-50 ms

Passo 7 — Browser parsa e renderizza

Il browser riceve l'HTML. Inizia a «leggerlo» — in gergo si dice parsing. Mentre lo legge, scopre che la pagina ha bisogno di altri file:

- Un foglio di stile CSS (`<link rel="stylesheet" href="/style.css">`)
- Uno script JavaScript (`<script src="/app.js">`)
- Tre immagini (`<img src="...">`)

Per ognuno di questi file, il browser fa **un'altra richiesta** (passo 4 di nuovo, ma stavolta è più veloce perché la connessione è già aperta). Mentre li scarica, costruisce gradualmente la pagina:

1. **DOM**: la struttura della pagina (vedi Modulo 4)
2. **Stili applicati**: ogni elemento riceve i suoi colori, font, layout
3. **JavaScript eseguito**: aggiunge interattività (form, click, animazioni)
4. **Immagini caricate**: appaiono progressivamente

⏱️ Tempo: 100-1000 ms (dipende da quanto è «pesante» la pagina)

Passo 8 — L'utente vede

Finalmente, dopo tutto questo, **l'utente vede la pagina**. Probabilmente in 300-800 millisecondi totali, se tutto va bene.

Lui ha aspettato meno di un secondo. Non sa che dietro sono successe **otto cose distinte**. Tu, adesso, sì.

Riepilogo coi tempi

#	Cosa	Tempo tipico
1	Click	0 ms
2	DNS lookup	20-100 ms
3	TCP + TLS handshake	50-200 ms
4	HTTP request	10-30 ms
5	Server processa (codice + DB)	5-2000 ms
6	HTTP response	10-50 ms
7	Browser parsa + scarica risorse	100-1000 ms
8	Visualizzazione	—
	Totale tipico	**300-1500 ms**

Perché questa lista è oro

Quando uno strumento di analisi (Lighthouse, GTmetrix, Vercel Speed Insights) ti dice «il tuo sito è lento, score 47/100», ti sta dicendo: «uno di questi 8 passi sta richiedendo più tempo di quanto dovrebbe».

- Lentezza al passo 2 → DNS lento o non in cache
- Lentezza al passo 3 → TLS lento (raro), server geograficamente lontano
- Lentezza al passo 5 → query DB lente, codice mal scritto, API esterne lente — **questo è il 80% delle ottimizzazioni**
- Lentezza al passo 7 → file CSS/JS troppo grossi, troppi script bloccanti

Senza la lista, «il sito è lento» è un'affermazione vaga. Con la lista, è una **diagnosi**.

Cosa ti porti via

1. Quando un utente clicca un link, succedono **8 cose** in sequenza in poche centinaia di millisecondi.
2. Il **collo di bottiglia tipico** è il passo 5 (server + database).
3. Conoscere i passi ti aiuta a **diagnosticare la lentezza** invece di «boh il sito è lento».
4. È anche utile per la **sicurezza**: HTTPS al passo 3 protegge tutto quello che viaggia da lì in poi.

Chiusura del Modulo 1

Adesso hai due cose nuove in testa:

- **La mappa delle 5 entità** (browser, DNS, CDN, server, database) — il sistema in cui vive ogni sito web
- **Il viaggio del click in 8 passi** — la sequenza temporale di cosa succede quando l'utente apre una pagina

Più importante di tutto: hai capito che **il «sito» non è una cosa sola**. È un sistema di pezzi distribuiti che lavorano insieme. Ogni pezzo può rompersi indipendentemente. Sapere quali sono i pezzi è il primo passo per sapere dove guardare.

Nel **Modulo 2** entriamo nel cuore della distinzione più importante: **client e server**. Quali linguaggi vivono dove. Perché esistono separati. E perché JavaScript è speciale (gira in entrambi).

🎯 Mini-quiz di autovalutazione

1. **L'utente apre un sito. Il browser non riesce a contattare il server. Quale dei 5 pezzi del sistema è probabilmente il colpevole?**
2. **Il sito si carica ma il login non funziona («Accedi con Google» non parte). Quale pezzo del Modulo 1.3 cercheresti per primo?**
3. **Lighthouse ti dice che la pagina è lenta. Il «Time To First Byte» (TTFB) è di 1.8 secondi. In quale degli 8 passi del viaggio del click sta succedendo il rallentamento?**
4. **Perché il browser non può parlare direttamente col database?**
5. **Vero o falso: una CDN serve solo se il tuo sito ha utenti in tutto il mondo.**

Risposte

1. **Il DNS** (probabilmente). Se il browser non riesce a «trovare» il server, di solito il DNS non risponde, non è configurato, o non si è propagato dopo una modifica recente. Il server potrebbe essere giù, ma il sintomo «sito irraggiungibile» punta più spesso al DNS che al server.
2. **Il pezzo 1 (frontend del bottone)** o, più probabile in produzione, **il pezzo 2 (configurazione OAuth con Google)** — magari le credenziali Google sono nel `.env` locale ma non in quello di produzione.
3. **Passo 5 — il server processa**. TTFB elevato significa che il server impiega troppo tempo a generare la risposta. Spesso sono query lente al database, codice non ottimizzato, o chiamate ad API esterne sincrone.
4. **Per sicurezza**. Se il browser potesse parlare direttamente col database, qualunque utente con un po' di malizia potrebbe leggere/modificare i dati di tutti. Il server fa da «filtro autorizzato»: riceve la richiesta, verifica chi sei, decide cosa farti vedere.
5. **Falso**. Una CDN aiuta anche se i tuoi utenti sono tutti italiani — perché serve i file statici da nodi geograficamente vicini (Milano, Roma) molto più velocemente del tuo server centralizzato. Plus: scarica il tuo server dal traffico delle immagini/CSS, e ti protegge da picchi di traffico.

> **SUGGERIMENTO**
>
> Se hai sbagliato più di 2 risposte, rileggi i capitoli relativi prima del Modulo 2. Non c'è fretta. La mappa che stiamo costruendo qui è quella che userai per i prossimi 100 prompt a Claude.

Modulo 1 redatto: 2026-04-25 · Versione 1.0 · 14 pagine · 3700 parole

Modulo 2 — Client e server: chi fa cosa

SUGGERIMENTO

5 capitoli · 18 pagine · 1 ora 10 minuti
Pre-requisiti: Moduli 0 e 1 letti. Conosci le 5 entità (browser, DNS, CDN, server, database) e il viaggio del click.

Mettiti comodo

Adesso entriamo nel concetto **più importante** di tutto il libro.

Se assorbi solo questo modulo e lasci perdere il resto, hai già ottenuto il 60% del valore di «Vibecoding Serio». Tutto quello che farai con Claude da qui in poi sarà più chiaro, più diretto, con meno iterazioni perse.

Il concetto è: **client** e **server** sono due mondi separati, con regole diverse.
Sapere quale codice gira da una parte e quale dall'altra ti permette di:

- Capire perché certi errori succedono (e dove cercarli)
- Scrivere prompt più precisi a Claude
- Riconoscere subito cosa Claude sta scrivendo (o sbagliando)
- Non commettere errori di sicurezza ovvi

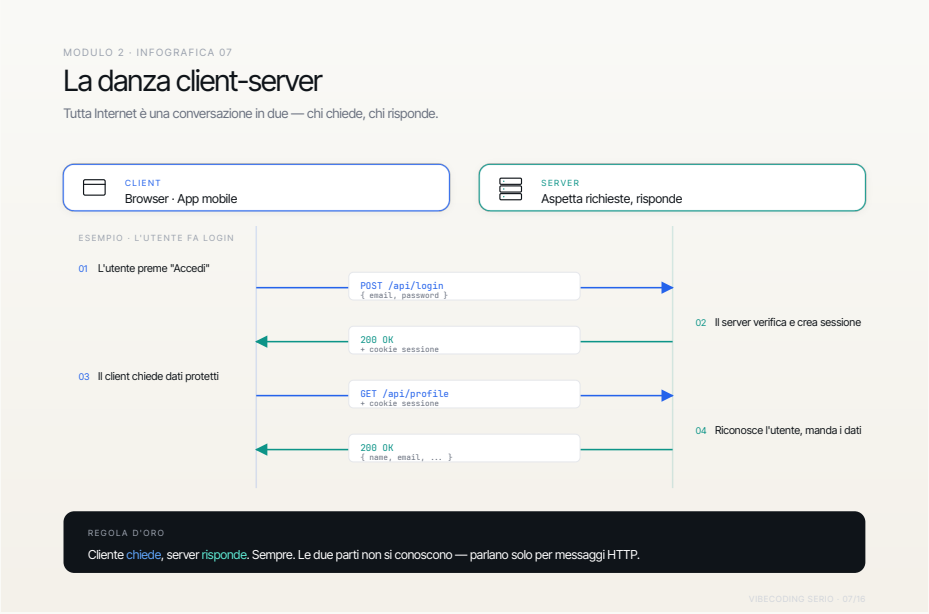
In questo modulo:

1. Il modello client-server in 5 minuti
2. I linguaggi del client (frontend) — cosa gira nel browser
3. I linguaggi del server (backend) — cosa gira sul server

4. Quando un linguaggio sta su entrambi (full-stack)
5. Il database: il terzo pilastro

Andiamo.

2.1 — Il modello client-server in cinque minuti



Infografica 07 — La danza client-server

Una conversazione in due

Tutta Internet, fondamentalmente, funziona così: **qualcuno chiede, qualcun altro risponde**.

- Tu apri un sito: il tuo browser **chiede**, il server del sito **risponde**.
- Apri una mail su Gmail: il browser **chiede**, il server di Google **risponde**.
- L'app di tua banca aggiorna il saldo: l'app **chiede**, il server della banca **risponde**.

Chi chiede si chiama **client**. Chi risponde si chiama **server**. Sempre.

Le due parti hanno ruoli diversi

Sono due metà della stessa moneta, ma fanno cose diverse.

Il client (di solito il browser dell'utente, o un'app mobile):
- Mostra cose all'utente (testi, immagini, bottoni)
- Riceve i suoi click, le sue digitazioni
- Manda **richieste** al server
- Riceve **risposte** e aggiorna lo schermo

Il server:
- Aspetta richieste
- Esegue codice (esempio: cerca nel database, calcola un prezzo, manda un'email)
- Risponde al client con il risultato
- Non vede mai l'utente. Sa solo che gli arrivano richieste.

Pensa al ristorante: il **cliente al tavolo** ordina dal menù. Il **cuoco in cucina** non vede il cliente. Riceve il foglietto dal cameriere (la richiesta), prepara il piatto (esegue codice), manda il piatto al cameriere (la risposta). Mai contatto diretto. Le due parti **non si conoscono**, parlano per messaggi.

Perché esistono separati

Una domanda legittima: «perché il browser non fa tutto da solo? Perché serve un server?»

Tre risposte, in ordine di importanza:

1. **Sicurezza**

Se il browser potesse fare tutto da solo, qualunque utente con un po' di malizia potrebbe leggere/modificare i dati di tutti. Un'app bancaria che gira «tutta nel browser» significherebbe che chiunque apre Chrome DevTools potrebbe vedere i saldi di tutti gli altri.

Il server agisce da **filtro autorizzato**: riceve la richiesta, verifica chi sei (autenticazione), decide cosa farti vedere (autorizzazione), e ti manda solo il pezzo che ti spetta.

2. **Dati condivisi**

Se hai un blog con 1000 utenti che leggono e commentano, quei dati (post, commenti) devono stare in un posto **centralizzato**. Non possono stare «nel browser di Mario», perché Lucia non vedrebbe mai i commenti di Mario.

Il server tiene i dati in un posto unico (il database), accessibile a tutti i client autorizzati.

3. **Cose che il browser non può fare**

Mandare un'email, processare un pagamento Stripe, integrare un servizio AI come OpenAI, salvare un file su un disco — queste cose il browser non le fa. O perché non ha gli strumenti, o perché sarebbe insicuro fargliele fare.

Il server le fa per conto del client.

Quando il confine si sposta (full-stack moderni)

Una nota importante. In progetti moderni come **Next.js**, frontend e backend possono **condividere lo stesso codice** e girare in entrambi i mondi.

Esempio: una funzione `validateEmail()` che controlla se una mail è ben formattata può girare:

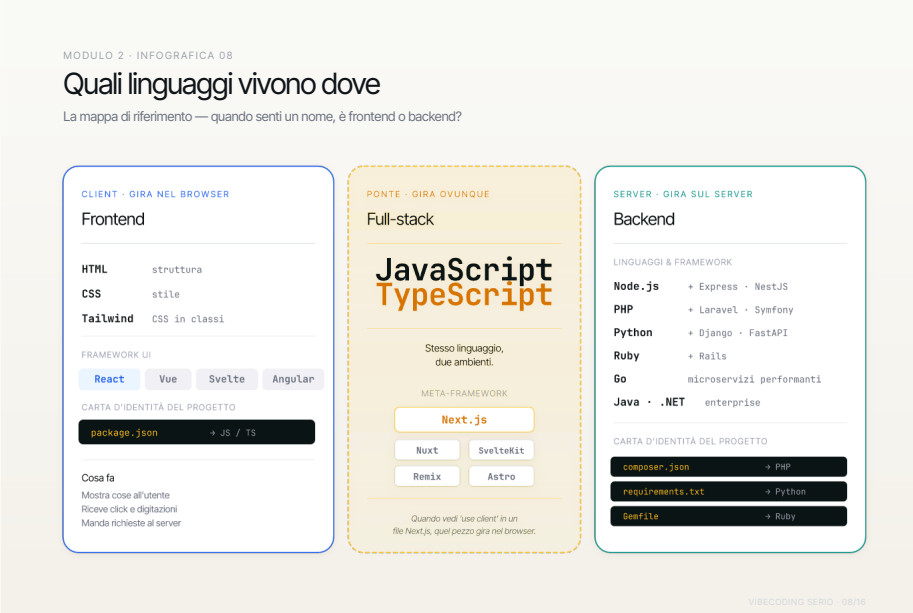
- **Lato client** per dare feedback istantaneo all'utente mentre digita
- **Lato server** per la sicurezza vera (perché lato client si può ingannare)

Stessa funzione, stesso file, due ambienti. Lo vedremo in 2.4.

Cosa ti porti via

1. Tutta Internet è una **conversazione client-server**: chi chiede, chi risponde.
2. **Il client** mostra e raccoglie input. **Il server** custodisce dati e fa cose pesanti.
3. Esistono separati per **sicurezza, dati condivisi, e cose che il browser non sa fare**.
4. La regola generale: **se richiede una password o un dato di altri utenti, deve passare dal server**. Senza eccezioni.

2.2 — I linguaggi del client (frontend)



Infografica 08 — Quali linguaggi vivono dove — la mappa di riferimento di tutto il libro

Cosa gira nel browser

Quando apri Chrome, Firefox o Safari, dentro al browser girano **solo tre tecnologie native**:

- **HTML** — la struttura della pagina
- **CSS** — l'aspetto visivo
- **JavaScript** — l'interattività

Punto. Tutto il resto che vedi su un sito (React, Vue, Tailwind, Svelte...) è **costruito sopra** queste tre tecnologie. Il browser non sa cosa è React. Sa solo HTML, CSS e JavaScript. I framework alla fine si «compilano» in questi tre.

Vediamoli uno per uno, senza mistificare.

HTML — lo scheletro

L'**HTML** (HyperText Markup Language) è una specie di Word per la struttura. Dici «questo è un titolo», «questo è un paragrafo», «questo è un bottone»,

«questa è un'immagine». Non dici **come si vede** (lo fa il CSS), né **come si comporta** (lo fa il JavaScript). Solo la struttura.

```
<header>...</header>
<main>
  <h1>Titolo della pagina</h1>
  <p>Un paragrafo di testo.</p>
  <button>Cliccami</button>
</main>
<footer>...</footer>
```

Quando Claude scrive una pagina, scrive HTML. È la prima cosa che genera.

CSS — la pelle

Il **CSS** (Cascading Style Sheets) descrive come si vede l'HTML. Colori, font, spaziature, layout, animazioni.

```
button {
  background: #2563EB;
  color: white;
  padding: 12px 24px;
  border-radius: 8px;
}
```

Tutto il «design» di un sito è CSS. È quello che separa un sito che sembra del 1998 da uno che sembra del 2026.

> **SUGGERIMENTO**
>
> **Tailwind CSS**: la libreria CSS più popolare oggi. Invece di scrivere `padding: 12px 24px`, scrivi `className="py-3 px-6"`. Stessa cosa, sintassi più compatta. Quando Claude scrive interfacce moderne, quasi sempre usa Tailwind.

JavaScript — i muscoli

Il **JavaScript** (JS) è il linguaggio che dà **vita** alla pagina. Quando clicchi un bottone, quando un menù si apre, quando un form si valida prima di inviarsi — tutto JavaScript.

```
button.addEventListener('click', () ⇒ {
  alert('Hai cliccato il bottone!');
});
```

Senza JavaScript, una pagina è una specie di brochure: la guardi, ma non interagisce.

TypeScript — JavaScript con i superpoteri

Il **TypeScript** (TS) è JavaScript con un controllo aggiuntivo dei «tipi». Tu dici «questa variabile è un numero», «questa funzione ritorna una stringa», e lo strumento ti avvisa se sbagli prima ancora di provare il codice.

```
function somma(a: number, b: number): number {
  return a + b;
}

somma('ciao', 5); // ❌ TypeScript ti dice "errore: 'ciao'
non è un numero"
```

In progetti seri si usa TypeScript invece di JavaScript «puro». Riduce i bug. Claude oggi quasi sempre scrive TypeScript per i progetti nuovi.

React, Vue, Svelte — i framework UI

Scrivere applicazioni complesse direttamente in JavaScript è faticoso. I **framework UI** ti danno strumenti per organizzare il codice in **componenti riusabili**.

- **React** (di Meta): il più diffuso. Creato nel 2013. Quello che Claude usa di default.
- **Vue**: alternativa più semplice da imparare. Popolare in Europa.
- **Svelte**: il più moderno (2019). Più snello, performance migliori.

- **Angular** (di Google): più enterprise, più rigido. Lo trovi in progetti aziendali grandi.

Sono tutti uguali in cosa fanno (UI dichiarativa con componenti) — diversi in come lo fanno. **React vince per popolarità e per quantità di librerie disponibili**, e per questo è quello su cui Claude ha la mano più allenata.

Next.js, Nuxt, SvelteKit — i meta-framework

Un livello sopra: i **meta-framework**. Prendono un framework UI (es. React) e ci aggiungono tutto quello che serve per costruire un sito serio:

- **Next.js** (basato su React) → il più popolare oggi
- **Nuxt** (basato su Vue)
- **SvelteKit** (basato su Svelte)
- **Remix** (basato su React)
- **Astro** (per siti statici/contenuto, agnostico)

Quando Claude scrive un'app web moderna, in 8 casi su 10 usa **Next.js**.

Cosa ti porti via

1. Il browser nativamente conosce solo **HTML + CSS + JavaScript**.
2. **Tailwind** è CSS scritto in modo più rapido.
3. **TypeScript** è JavaScript con controllo extra.
4. **React, Vue, Svelte** sono framework UI per organizzare codice JS in componenti. **React vince per popolarità**.
5. **Next.js** è il meta-framework più usato oggi (React + features). Quando vedi `pages/` o `app/`, è Next.js.

2.3 — I linguaggi del server (backend)

> **SUGGERIMENTO**
>
> Continua l'infografica 8.

Cosa gira sul server

Sul server può girare **qualunque linguaggio di programmazione**. Letteralmente. Il browser è limitato a JavaScript, il server no.

I più usati nel mondo web (in ordine di diffusione):

Node.js — JavaScript sul server

Node.js è una runtime che permette di eseguire JavaScript **fuori dal browser**. È la rivoluzione del 2009: per la prima volta, lo stesso linguaggio gira sia frontend che backend.

Framework backend Node.js comuni:
- **Express**: il più semplice, il «Flask» del mondo Node
- **Fastify**: alternativa moderna, performance migliori
- **NestJS**: più strutturato, stile Angular

Quando vedi su Vercel o Netlify un'app Next.js, sotto sta girando Node.js. Quando Claude scrive un endpoint `/api/...`, è codice Node.js.

PHP — il classico

Il **PHP** è il linguaggio dei siti web «tradizionali». WordPress, Joomla, Drupal — tutti PHP. Sembra antico, ma è ovunque: il **40% di Internet** gira ancora su PHP.

Framework PHP moderni:
- **Laravel**: il più popolare, elegante, sintassi pulita
- **Symfony**: più enterprise, modulare

Su hosting Aruba «Linux Hosting» il default è PHP. Se hai un sito legacy o un'agenzia te ne propone uno, è quasi sicuramente PHP+Laravel o WordPress.

Python — il linguaggio dell'AI

Il **Python** è il linguaggio scientifico per eccellenza. È diventato lo standard per:
- AI / Machine Learning (TensorFlow, PyTorch, OpenAI SDK)
- Data Science (pandas, numpy, jupyter)
- Web scraping
- Scripting / automation

Per app web esistono framework:
- **Django**: il «Laravel di Python», batterie incluse
- **Flask**: minimalista, fai tutto a mano
- **FastAPI**: il moderno, ottimo per le API

Se il tuo progetto coinvolge AI, processamento dati, modelli ML — quasi sicuramente Python. Cantiere ad esempio ha un servizio Python per l'OCR delle bolle, anche se il sito principale è Next.js.

Ruby — produttività estrema

Ruby + **Rails** è una coppia famosa. Usato in startup classiche (Airbnb, GitHub, Shopify, Basecamp). Ottimo per partire velocissimo. Meno popolare oggi rispetto a Node.js / Next.js, ma ancora forte.

Go — performance e semplicità

Go (o «Golang») è il linguaggio creato da Google. Più «low-level» rispetto a Node/Python, ma molto veloce. Perfetto per microservizi, sistemi che devono reggere tanto traffico, infrastrutture cloud.

Java, Kotlin, .NET (C#) — enterprise

Java (e **Kotlin**, suo cugino moderno) e **.NET** (Microsoft, linguaggio C#) sono i giganti del mondo enterprise. Banche, assicurazioni, grandi aziende. Robusti, super-tipizzati, lenti da scrivere ma stabili.

Se sviluppi un piccolo SaaS o un'app web standard, non li userai. Se finisci a lavorare con un'azienda da 1000 dipendenti, sì.

Tabella decisionale: cosa userà Claude in base al tuo progetto

Tipo di progetto	Stack tipico che Claude propone
App web moderna full-stack (login, dashboard, CRUD)	**Next.js** + Supabase / Postgres
Sito vetrina + form contatti	**Astro** o **Next.js** statico + form action
WordPress / blog tradizionale	**PHP + WordPress** (raramente Claude propone WordPress, ma se hai un legacy...)
Backend API per app mobile	**Node.js + Express** o **FastAPI**
Progetto AI / machine learning	**Python + FastAPI** + librerie ML
Microservizio ad alta performance	**Go**
App enterprise	**Java + Spring** o **.NET**