



A BEGINNER'S MANUAL

CLAUDE EDITION

Autonomous AI Trading

Build, test, and run a hands-off trading bot, step by step

Autonomous AI Trading

Build, test, and run a hands-off trading bot, step by step

CLAUDE EDITION Expanded edition written by Claude (Anthropic's AI) in Claude Code and published in GuidesForEverything PR #4.

This manual walks you, one small step at a time, from "I have never written code" to a **trading bot that runs by itself**: it wakes up on a schedule, reads market data, decides, checks its own safety rules, places orders through a broker's API, writes a log, and sends you a daily report. You'll build it in Python with an AI assistant (Claude Code) as your pair programmer, test it on years of history, and run it on **paper money** for weeks before a single real dollar is involved.

It is also an honest book. Most people who trade actively lose money, and no bot, AI model, or course can guarantee a profit. You'll see the real studies, learn the legal and tax basics, and build the guardrails (position limits, stop-losses, daily loss limits and a kill switch) that keep a mistake small. The goal is a calm, well-tested system you understand, sized so that even its worst day is one you can shrug off.

HOW TO READ THIS MANUAL

Tip — a shortcut or good habit.

Warning — easy to get wrong.

`$ command` — text you type, exactly as shown.

Note — useful background.

Danger — can lose data, money, or access.

`Settings > Privacy` — a menu path to click.

New words appear in **bold** the first time and are collected in the Glossary at the back.

This manual is education, not financial, legal, or tax advice. Trading involves risk, including the loss of all money you put in; past and simulated results do not predict future returns. Only trade money you can afford to lose, and paper-trade first. Broker features, fees, rules, and tax law were checked against official sources in October 2026 and change often; confirm current details with your broker, the IRS, FINRA, the SEC, or a qualified professional before acting.

Contents

01 What Autonomous AI Trading Is (and Isn't)	8
1.1 A bot is a loop with rules	8
1.2 What "AI" really adds	9
1.3 What a bot is good at	9
1.4 The system you'll build	10
1.5 A realistic timeline	10
1.6 How to use this manual	11
02 The Honest Odds and Risks	12
2.1 What the studies found	12
2.2 Why the odds are tilted	13
2.3 The extra risks a bot brings	13
2.4 Nothing guarantees profit	14
2.5 A definition of success you can win	14
2.6 Before you continue	14
03 Legal, Account and Tax Basics	15
3.1 Brokers and exchanges	15
3.2 Cash accounts versus margin accounts	16
3.3 The pattern day trader rule ended in 2026	16
3.4 Crypto rules in brief	17
3.5 Taxes: the essentials	17
3.6 Other countries	18
04 Choosing a Market	19
4.1 The two beginner-friendly markets	19
4.2 Leave these for later	20
4.3 Pick a starter watchlist	21
4.4 Choose a timeframe	21
05 Choosing a Broker or Exchange	22
5.1 What matters for a bot	22
5.2 Stock brokers	23
5.3 Crypto exchanges	24

5.4	Side by side	25
06	Setting Up Your Computer	26
6.1	Install Python {{Free}}	26
6.2	Install VS Code {{Free}}	27
6.3	Create your project	27
6.4	Use Git from day one	28
6.5	Install Claude Code	29
6.6	Teach Claude about your project	29
6.7	Working with AI safely	30
07	API Key Safety	31
7.1	Create your paper keys	31
7.2	The principle of least privilege	32
7.3	IP allowlists	32
7.4	Store keys in a .env file	33
7.5	Rules for keys	34
7.6	If a key leaks	34
08	Market Data	35
8.1	Bars and OHLCV	35
8.2	Feeds: IEX and SIP	36
8.3	Your first data script	36
8.4	Adjusted prices	37
8.5	How data goes wrong	37
8.6	Other data you might want	38
09	Strategies in Plain English	39
9.1	Dollar-cost averaging (DCA)	39
9.2	Rebalancing	40
9.3	Moving-average trend following	41
9.4	Mean reversion	41
9.5	Momentum	42
9.6	Grid bots (crypto)	43
9.7	Choosing your first strategy	43
10	Where AI Really Fits	45
10.1	Four roles, ranked by safety	45
10.2	Role 2: scoring news with a language model	45

10.3	Role 3: machine-learning predictions	47
10.4	Role 4: AI agents that trade	47
10.5	Prompts for using AI well	49
11	Backtesting Step by Step	50
11.1	Step 1: Save history to a file	50
11.2	Step 2: A backtest in pandas	51
11.3	Step 3: Measure it	52
11.4	Step 4: The same test in vectorbt	53
11.5	The traps	54
11.6	Walk-forward testing	54
12	Paper Trading for Weeks	56
12.1	What paper trading catches	56
12.2	How long?	57
12.3	Write your success criteria first	57
12.4	Compare paper with the backtest	58
12.5	The weekly review	58
13	Risk Management in Code	59
13.1	Position sizing: the 1–2% rule	59
13.2	Stop-losses	60
13.3	Daily loss limit	61
13.4	Position and order limits	61
13.5	The kill switch	62
13.6	The full rule set	62
14	Build the Bot, Part 1: Foundations	63
14.1	The project layout	64
14.2	Step 1: The config file	65
14.3	Step 2: The settings loader	66
14.4	Step 3: The data module	67
14.5	Step 4: The signal module	68
14.6	Building it with Claude Code	68
15	Build the Bot, Part 2: Risk, Orders and the Main Loop	70
15.1	Step 5: The risk module	70
15.2	Step 6: The state file	70
15.3	Step 7: The orders module	72

15.4	Step 8: Logging and the trade log	74
15.5	Step 9: Alerts	75
15.6	Step 10: The main loop	75
15.7	Step 11: Tests	79
16	Running It 24/7	83
16.1	Market hours and holidays	83
16.2	Option 1: A small cloud server	84
16.3	Option 2: Schedule with cron	85
16.4	Option 3: Schedule with systemd	86
16.5	Option 4: GitHub Actions	87
16.6	Option 5: n8n	88
16.7	Which should you pick?	88
17	Monitoring, Alerts and a Dashboard	89
17.1	Telegram alerts {{Free}}	89
17.2	Slack alerts	90
17.3	Email alerts	90
17.4	What to alert on	91
17.5	The daily report	91
17.6	A heartbeat for silent failures	92
17.7	A simple dashboard	92
18	Going Live with Tiny Money	94
18.1	Are you ready?	94
18.2	How much?	95
18.3	Switching from paper to live	95
18.4	The first month	96
18.5	Scaling up, slowly	96
18.6	When to stop	96
19	Maintaining and Improving It	97
19.1	Routines	97
19.2	Updating libraries safely	98
19.3	Changing the strategy	98
19.4	Ideas for improvement	98
19.5	Claude Code as your maintenance partner	99
20	Scams to Avoid	100

20.1	What regulators say	100
20.2	The five common scams	101
20.3	Red flags	101
20.4	How to check a firm	101
20.5	If you've been targeted	102
21	Checklists and Troubleshooting	103
21.1	Master checklists	103
21.2	Troubleshooting	104
21.3	Quick reference	105
	Glossary	106

What Autonomous AI Trading Is (and Isn't)

An **autonomous trading bot** is a small computer program that buys and sells investments for you, on a schedule, following rules you wrote down in advance. Once it's switched on, it doesn't wait for you to click anything. That's the "autonomous" part. The "AI" part is optional, and smaller than the ads suggest. This chapter clears up what you're actually building.

IN THIS CHAPTER

- What a trading bot does, step by step
- Where AI genuinely helps, and where it doesn't
- The system you'll build in this manual
- A realistic timeline from zero to live

1.1 A bot is a loop with rules

Strip away the marketing and every trading bot does the same five things, over and over:

1. **Wake up.** A scheduler (a clock on a computer) starts the program, for example every weekday at 10:00 a.m.
2. **Look.** It downloads recent prices from your broker's data service.
3. **Decide.** It runs your **strategy**, the written rules that say when to buy and when to sell. For example: "Own this fund when its 50-day average price is above its 200-day average."
4. **Check.** It runs **risk rules**: Is this trade too big? Have we already lost too much today? Is the kill switch on?
5. **Act and record.** If every check passes, it sends an **order** (an instruction to buy or sell) to your broker through an **API**, writes what it did to a log file, and messages you.

An **API** (application programming interface) is a doorway that lets one program talk to another. Your broker's website is a doorway for humans; its API is a doorway for programs. Your bot will use it to read your balance, fetch prices, and place orders.

i NOTE

"Autonomous" doesn't mean "unsupervised forever." It means the bot doesn't need you *in the moment*. You still review its reports, update it, and decide whether it keeps running. Think of a dishwasher: it runs by itself, but you still check it's not leaking.

1.2 What "AI" really adds

People use "AI trading" to mean four very different things:

KIND	WHAT IT IS	HONEST VERDICT
Rule-based bot	Fixed if-then rules, no AI	Most real bots; the safest start
AI as coder	An assistant like Claude writes and reviews your bot	Huge help for beginners
AI as analyst	A language model reads news and scores it	Useful as one input, never alone
AI as trader	An AI agent decides and places trades	Experimental; needs tight guardrails

This manual uses AI in all four ways, but in that order of trust. Your bot's *decisions* will start as plain, testable rules. AI will help you write the code, explain errors, review your logic, and (later, optionally) add a news-based filter. You'll also see how to connect an AI agent to a paper account, with fences around it.

☆ IMPORTANT

No AI can predict prices reliably. Markets react to news nobody has seen yet. The U.S. Commodity Futures Trading Commission (CFTC) puts it bluntly in a customer advisory titled "*AI Won't Turn Trading Bots into Money Machines*": AI technology can't predict the future or sudden market changes.

1.3 What a bot is good at

A bot is not smarter than you. Its advantages are about *behaviour*:

- **It follows the plan.** It never panics in a crash or gets greedy in a rally.
- **It never forgets.** It checks every rule on every run, even at 3 a.m.
- **It's testable.** You can replay years of history and see how the rules *would* have behaved.
- **It's consistent and logged.** Every decision is written down, so you can learn from it.

And what it's bad at: it can't tell when the world has changed, it will repeat a bug thousands of times without blinking, and it will happily trade itself into a loss if your rules allow it. That's why half of this manual is about testing and safety.

1.4 The system you'll build

By the end of this manual you'll have a working bot made of small, understandable parts:

- Python project on your computer
- Paper trading account at Alpaca
- Config file with your rules
- Data module (daily prices)
- Strategy module (signals)
- Risk module (the “no” rules)
- Orders module (the only part that trades)
- Log file and state file
- Telegram or Slack alerts
- Daily report and small dashboard
- Tests that check the logic
- Scheduler on a cloud server

We use **Alpaca** for the main build because it offers a free paper trading account to anyone with an email address, a well-documented Python library, commission-free U.S. stock trading, and an official AI connector. You can adapt the same design to other brokers; Chapter 5 compares them.

1.5 A realistic timeline

FROM ZERO TO A LIVE BOT

- 1 Weeks 1–2: Learn and set up.**
Read Chapters 1–8, install Python and VS Code, open a paper account, and get your first script running.
- 2 Weeks 3–4: Build and backtest.**
Write your strategy, test it on history, and learn why most good-looking backtests are lies.
- 3 Weeks 5–6: Build the full bot.**
Add risk rules, orders, logging, state, alerts, and tests.

4

Weeks 7–14 (at least): Paper trade.

Run it on a schedule with fake money and compare results with the backtest.

5

After that: Go live tiny.

If, and only if, paper trading met your success criteria, switch to a very small real-money account.

 TIP

Treat this as learning a craft, like woodworking. Your first table will wobble. That's fine, as long as you build it with scrap wood (paper money) rather than the good oak (your savings).

1.6 How to use this manual

Read the chapters in order the first time. Every chapter ends with a checklist; don't move on until you can tick it. Code appears in short blocks with an explanation between each one. You don't have to type it: you can paste it, or ask Claude Code to write it with the prompts provided. Either way, **read every line** before you run it. A bot is software that spends money.

- ☐ I can explain what a trading bot does in five steps
- ☐ I know that “AI” in this manual mostly means an AI coding helper
- ☐ I understand no tool can guarantee profits
- ☐ I've blocked out several weeks for paper trading before going live

The Honest Odds and Risks

Before you write a line of code, you deserve the truth about the game you're joining. This isn't meant to scare you off. It's meant to set your expectations so that a bad week doesn't feel like a betrayal and a good week doesn't make you reckless.

IN THIS CHAPTER

- What the research says about active traders
- Why the odds are tilted against short-term trading
- The specific risks a bot adds
- How to define “success” in a way you can actually win

2.1 What the studies found

Researchers have studied millions of real trading accounts. The results are remarkably consistent:

STUDY	WHAT THEY LOOKED AT	FINDING
Barber & Odean, <i>Journal of Finance</i> (2000)	66,465 U.S. households, 1991–1996	The most active traders earned 11.4% a year while the market returned 17.9%
Barber, Lee, Liu & Odean, <i>Review of Financial Studies</i> (2009)	Every individual investor in Taiwan	Individuals' trading cost them 3.8 percentage points a year in aggregate
Barber, Lee, Liu & Odean, <i>Journal of Financial Markets</i> (2014)	Taiwanese day traders, 1992–2006	Less than 1% could predictably and reliably earn positive returns after fees
Chague, De-Losso & Giovannetti (2019)	Everyone who started day trading Brazilian futures, 2013–2015	97% of those who kept going over 300 days lost money; only 1.1% earned more than minimum wage

Day trading means opening and closing positions within the same day. The Brazilian study is especially sobering because it followed *persistent* traders, the people who stuck with it. Practice didn't save them; the authors found no evidence that day traders learn their way into profit.

☆ IMPORTANT

These studies are about humans clicking buttons, but a bot doesn't escape the same forces. It pays the same costs, competes against the same professionals, and faces the same randomness. Automation removes emotional mistakes. It doesn't create an edge.

2.2 Why the odds are tilted

- **Costs add up.** Even “commission-free” trading has a **spread** (the gap between the buying price and the selling price), small regulatory fees, and **slippage** (getting a slightly worse price than you expected). Crypto exchanges charge explicit fees on every trade. A strategy that trades daily pays these hundreds of times a year.
- **You're trading against professionals.** Big firms have faster computers, better data, and teams of PhDs. In short-term trading, you're often the slowest player at the table.
- **Taxes.** In the U.S., profits on things held a year or less are taxed at your ordinary income rate, usually higher than the rate on long-term gains (Chapter 3).
- **Randomness looks like skill.** Over a few months, luck dominates. A coin-flipping strategy can look brilliant for a while.
- **Even professionals struggle.** S&P Dow Jones Indices' long-running SPIVA reports find that most actively managed funds trail their benchmark index over long periods.

2.3 The extra risks a bot brings

RISK	WHAT IT LOOKS LIKE	YOUR DEFENSE (CHAPTER)
Bugs	A typo buys 100 shares instead of 10	Tests, size caps (13, 15)
Runaway loops	The bot re-sends the same order every minute	Order limits per run (13)
Bad data	A price glitch triggers a sell	Sanity checks (8)
Outages	Your server or the broker is down	Alerts, heartbeat (17)
Overfitting	Rules tuned to the past fail in the future	Walk-forward testing (11)
Stolen keys	Someone uses your API key	Trade-only keys, IP limits (7)
AI mistakes	A model misreads news or invents facts	AI as a filter only, guardrails (10)

2.4 Nothing guarantees profit

Say it out loud once: *there is no strategy, bot, AI model, course, or signal group that guarantees profits*. Anyone who promises otherwise is either mistaken or running a scam (Chapter 20 shows the warning signs). Regulators at the SEC, FINRA, and the North American Securities Administrators Association issued a joint alert warning about platforms claiming their AI “can’t lose” or picks “guaranteed” winners.

2.5 A definition of success you can win

You can’t control returns. You *can* control process. For your first year, define success like this:

- ☐ I built a bot I understand line by line
- ☐ It ran for months without a single unintended trade
- ☐ Its live results stayed close to its paper results
- ☐ My worst day lost less than the limit I set in advance
- ☐ I never added money I couldn’t afford to lose
- ☐ I learned enough to judge new strategies sceptically

That’s a genuinely valuable outcome even if the bot only matches, or slightly trails, simply buying and holding an index fund. Many people who go through this process end up running a calm, automated **DCA** (regular-investment) or **rebalancing** bot, which is a perfectly good result. Those bots are covered first in Chapter 9.

⚠ DANGER

Only trade money you can afford to lose completely. Not your emergency fund, not rent, not borrowed money. If losing the whole amount would change your life, the amount is too big.

2.6 Before you continue

- ☐ I’ve read the research and accept that most active traders lose
- ☐ I’ve chosen an amount I could lose without real harm
- ☐ I’m committed to paper trading first
- ☐ My goal for year one is a safe, well-tested process

CHAPTER 3

Legal, Account and Tax Basics

You don't need a law degree to run a small bot, but you do need to know the handful of rules that trip beginners up. This chapter focuses on the United States, with a short note for other countries at the end. It's general education, not tax or legal advice.

IN THIS CHAPTER

- Brokers versus crypto exchanges
- Cash accounts versus margin accounts
- The end of the “pattern day trader” rule in 2026
- Crypto protections (and the lack of them)
- Short-term gains, wash sales, Form 1099-DA, and record keeping

3.1 Brokers and exchanges

A **broker** (or **brokerage**) is a company that holds your account and sends your orders to the market. To trade U.S. stocks and **ETFs** (exchange-traded funds: baskets of many investments that trade like a single stock), you open an account with a broker such as Alpaca or Interactive Brokers.

A **crypto exchange** is a company that runs a marketplace for **cryptocurrencies** like bitcoin, usually holding your coins for you. Coinbase and Kraken are examples. Some brokers, including Alpaca, offer crypto too.

	STOCK BROKER	CRYPTO EXCHANGE
Regulated by	SEC and FINRA	Mix of federal and state rules
Account protection	SIPC up to \$500,000 (incl. \$250,000 cash)	No SIPC for most crypto
Trading hours	Weekdays, set hours	24 hours, 7 days
Typical fees	\$0 commission on stocks	A percentage per trade

SIPC (the Securities Investor Protection Corporation) protects customers if a member brokerage *fails* and assets go missing. It doesn't protect you from market losses. SIPC states that it does not protect crypto assets that aren't registered securities, even when held at a SIPC member firm.

⚠ WARNING

If a crypto exchange collapses, you may be an unsecured creditor waiting years to recover a fraction of your money. Keep only what the bot needs on any exchange.

3.2 Cash accounts versus margin accounts

- A **cash account** lets you buy only with money you've deposited. You can't lose more than you put in. **Start here.**
- A **margin account** lets you borrow from the broker to buy more, and is required for **short selling** (betting a price falls). Borrowing multiplies gains *and* losses, and losses can exceed your deposit.

Cash accounts have one rule to understand: **settlement**. When you sell a stock, the cash isn't officially yours until the trade *settles*. Since May 28, 2024, most U.S. stock trades settle **T+1**, one business day after the trade date. FINRA warns that if you buy with unsettled cash and then sell that new position before the original sale settles, you can get a **good faith violation**. Buying and selling before paying at all is **free-riding**, which under the Federal Reserve's Regulation T can get your cash account restricted for 90 days.

💡 TIP

The bot in this manual trades at most once a day per symbol and only uses settled cash, which keeps you well clear of these rules. If you build something faster, track settled cash in code.

3.3 The pattern day trader rule ended in 2026

For decades, FINRA Rule 4210 labelled anyone who made four or more day trades within five business days in a margin account a **pattern day trader (PDT)** and required them to keep at least **\$25,000** in the account. Many beginner guides still describe this rule.

It has changed. In **FINRA Regulatory Notice 26-10** (April 20, 2026), FINRA announced new **intraday margin standards** that replace the day-trading margin requirements entirely, including the PDT designation and the \$25,000 minimum. The SEC approved the change on April 14, 2026.

DATE	WHAT HAPPENS
June 4, 2026	New rule takes effect
Until Oct 20, 2027	Brokers may phase in the change

Instead of counting trades, firms must make sure your margin account has enough equity for the exposure you take on *during* the day. The standard minimums for margin accounts still apply.

☆ IMPORTANT

Because of the phase-in, **your broker may still apply the old PDT rules** until it switches over. Check your broker's own notice before relying on the change. Alpaca, for example, has a help page titled *Understanding FINRA's New Intraday Margin Rule and the End of PDT*. And cash accounts were never subject to PDT; settlement rules still apply.

3.4 Crypto rules in brief

- U.S. crypto exchanges must register with the Treasury's FinCEN and often hold state licenses. Check that any platform is licensed where you live.
- The SEC and CFTC have jointly listed bitcoin, ether, and a number of other large tokens as examples of **digital commodities** rather than securities.
- Crypto markets never close, prices can move 10% in an hour, and many tokens have little real trading activity. Your bot must handle weekends and sudden moves.
- Stick to large, liquid coins on a licensed U.S. exchange while learning.

3.5 Taxes: the essentials

Short-term versus long-term gains

According to the IRS, if you hold an asset **one year or less**, your gain or loss is **short-term**; more than a year is **long-term**. Short-term gains are taxed as ordinary income. Long-term gains get lower rates (0%, 15%, or 20% for most people). Almost every trade a bot makes is short-term.

If losses exceed gains in a year, you can deduct up to **\$3,000** of net loss against other income (\$1,500 if married filing separately) and carry the rest forward.

The wash-sale rule

IRS Publication 550 explains that a **wash sale** happens when you sell stock or securities at a loss and, **within 30 days before or after**, buy substantially identical stock or securities. The loss is disallowed for now and added to the cost of the new shares.

Bots trigger wash sales constantly, because they often sell and rebuy the same fund within a month. That doesn't usually raise your total tax over time, but it makes your records messy. Your broker tracks wash sales for stocks within one account.

NOTE

As of this edition, the wash-sale rule's text covers "stock or securities," and the IRS treats crypto as property. A bill introduced in June 2026, **H.R. 9172**, would extend wash-sale rules to most digital assets, and as written would apply to sales after its introduction date if enacted. Check its status on [congress.gov](https://www.congress.gov) and ask a tax professional.

Form 1099-DA for crypto

Brokers send you a **1099** form each year listing your sales. For crypto, the IRS created **Form 1099-DA**. Under the IRS's final regulations, custodial crypto brokers must report **gross proceeds** for sales on or after **January 1, 2025**, and **cost basis** for certain sales on or after **January 1, 2026**. Every U.S. tax return also asks a yes/no **digital assets question**; selling or exchanging crypto means "Yes."

Keep records anyway

Your bot makes many small trades. Keep your own record of every fill: date and time, symbol, side, quantity, price, fees, and order ID. You'll build a trade log that does this automatically (Chapter 15). Export your broker's yearly tax documents and keep them with the bot's logs.

3.6 Other countries

The ideas carry over, but the rules don't. In the UK, look up HMRC's capital gains guidance and the Financial Conduct Authority's register. In the EU, check your national regulator and ESMA. In Canada, check CIRO and your province. In Australia, check ASIC and the ATO. Alpaca's paper-only account is available globally, so you can still learn everything here before choosing a local broker.

- ☐ I'll start with a cash account, not margin
- ☐ I know the PDT rule was replaced, but my broker may still apply it until it switches
- ☐ I know crypto usually isn't SIPC-protected
- ☐ Bot profits are mostly short-term; I'll keep a trade log and my 1099s

Choosing a Market

Where your bot trades matters as much as how. Some markets forgive beginner mistakes; others punish them within minutes. This chapter helps you pick a calm playground.

IN THIS CHAPTER

- Stocks and ETFs versus spot crypto
- Why to avoid leverage, options, and futures at first
- Picking a starter watchlist

4.1 The two beginner-friendly markets

Stocks and ETFs

A **stock** (or **share**) is a small piece of ownership in a company. An **ETF** holds many stocks or bonds at once. For a first bot, ETFs are ideal: one ticker gives you hundreds of companies, so a single company's bad news can't sink you.

STRENGTH	WHY IT HELPS A BEGINNER
Deep markets	Popular ETFs trade millions of shares a day, so your order barely moves the price
Set hours	The bot only needs to run on weekdays (9:30 a.m. to 4:00 p.m. Eastern for the core session)
Low costs	\$0 commissions at many brokers; tiny spreads on big ETFs
Protections	SIPC coverage, regulated brokers

Spot crypto

Spot means you buy the actual coin, not a contract about it. Crypto is popular for bots because it trades around the clock and has big price swings.

STRENGTH	CAUTION
Open 24/7	Your bot must handle nights and weekends
Easy to start	Higher fees per trade, often 0.1%–0.6%
Big moves	Big moves go both ways; 20% drops happen
Many coins	Small coins can be illiquid or manipulated

TIP

Start with stocks/ETFs for your first bot. The fixed hours give you nights and weekends to read logs and fix things. Add crypto later as a second bot with its own budget.

4.2 Leave these for later

PRODUCT	WHAT IT IS	WHY NOT YET
Margin	Borrowed money	Losses can exceed your deposit
Options	Contracts giving the right to buy/sell at a set price by a date	Complex pricing; many expire worthless; small bugs get expensive
Futures	Contracts to buy/sell later, with built-in leverage	One bad day can wipe out an account
Perpetuals	Crypto futures with no expiry	High leverage, forced liquidations
Forex/CFDs	Currency and contract-for-difference trading	Heavy leverage; many offshore scams

Leverage means controlling a large position with a small deposit. At 10× leverage, a 10% move against you wipes out 100% of your money. A bot with a bug and leverage can lose everything before you wake up.

DANGER

Several broker APIs and AI connectors can trade options and futures. Turn those permissions off. Your risk module will also reject any symbol that's not on your approved list.

4.3 Pick a starter watchlist

A **watchlist** is the short list of symbols your bot is allowed to trade. Keep it small, liquid, and diversified. A sensible starter list mixes broad U.S. stocks, international stocks, bonds and gold, for example:

SYMBOL	WHAT IT TRACKS
SPY	500 large U.S. companies
QQQ	100 large Nasdaq companies
IWM	2,000 smaller U.S. companies
EFA	Developed markets outside North America
TLT	Long-term U.S. Treasury bonds
GLD	Gold

These are examples to learn with, not recommendations. Whatever you choose, look up each fund's issuer page to see what it holds and what it costs to own (its expense ratio).

For crypto, a starter list is just **BTC/USD** and **ETH/USD**: the two largest and most heavily traded pairs.

4.4 Choose a timeframe

A bot's **timeframe** is how often it looks at prices and can trade.

TIMEFRAME	TRADES PER YEAR	BEGINNER FIT
Monthly	~12	Excellent (DCA, rebalancing)
Daily	Dozens	Excellent (this manual's main bot)
Hourly	Hundreds	Possible later
Minutes or seconds	Thousands	Avoid: costs and competition win

Slower is better while learning. A daily bot gives every trade time to make sense, keeps costs low, and survives a server hiccup without drama.

- ☐ My first bot will trade liquid ETFs (or BTC/ETH spot)
- ☐ No margin, options, futures, perpetuals or leverage
- ☐ I've written a starter watchlist of 3–6 symbols
- ☐ My bot will run daily, not every minute

Choosing a Broker or Exchange

Your bot needs a broker with two things: an **API** it can use, and a **paper trading** or **sandbox** environment where it can practise with fake money. **Paper trading** simulates real orders against live prices without risking money. A **sandbox** is a test copy of the API; some sandboxes simulate trading, others just return canned example answers.

IN THIS CHAPTER

- What to look for in a broker for bots
- Alpaca, Interactive Brokers, Tradier, tastytrade, Schwab
- Coinbase Advanced Trade and Kraken for crypto
- A side-by-side comparison

5.1 What matters for a bot

FEATURE	WHY IT MATTERS
Paper trading that simulates fills	You can run the real bot for weeks safely
A maintained Python library	Less code for you to write
Clear fees	You can include them in backtests
Key permissions	Trade-only keys with no withdrawals
Good docs and status page	You can debug problems yourself

5.2 Stock brokers

Alpaca FREE

Alpaca is built for programmers and is this manual's main example.

- **Paper trading:** Free for everyone. Anyone in the world can open a **Paper Only Account** with just an email address. New paper accounts start with \$100,000 of pretend money, and you can open and delete extra paper accounts from the dashboard.
- **Fees:** Commission-free U.S. stock and ETF trading. Crypto has volume-tiered fees; at the lowest tier they're 0.15% for **maker** orders (orders that wait on the order book) and 0.25% for **taker** orders (orders that fill immediately).
- **Python library:** `alpaca-py` (version 0.44 in October 2026; needs Python 3.10 or newer).
- **Market data:** The free **Basic** plan includes real-time data from the IEX exchange and historical data, with 200 data requests per minute. **Algo Trader Plus** (\$99/month) adds all U.S. exchanges.
- **AI connector:** An official MCP server lets AI assistants read your account and place orders (Chapter 10).

NOTE

Alpaca's docs list what paper trading *doesn't* simulate: market impact, slippage from delays, queue position, price improvement, regulatory fees, and dividends. Paper results will usually look a bit better than live ones.

Interactive Brokers (IBKR)

A large global broker with access to many countries' markets.

- **APIs:** The **TWS API** (works with its desktop Trader Workstation or IB Gateway program, with Python support) and the REST-based **Web API**.
- **Paper trading:** Offers a paper trading account with simulated funds.
- **Fees:** IBKR Lite advertises \$0 commissions on U.S. stocks and ETFs; IBKR Pro uses per-share pricing.
- **Good for:** People who want international markets later. The API is more complex for beginners.

Tradier

A broker focused on API trading. A free account gets you both live trading and a **paper trading sandbox**, each with its own access token. Check Tradier's pricing page for current plans.

tastytrade

Offers an open API with a **sandbox** that resets every 24 hours and uses 15-minute delayed quotes. Sandbox and production credentials are separate. Better known for options, which you're avoiding for now.

Charles Schwab

Schwab's **Trader API – Individual** lets you build apps for your own accounts. Access requires a developer account and an approval process. Look for current paper-trading options on developer.schwab.com before planning around it.

5.3 Crypto exchanges

Coinbase Advanced Trade

Coinbase's **Advanced Trade API** offers REST and WebSocket access to spot trading, with an official Python SDK, `coinbase-advanced-py`.

- **Keys:** Created in the Coinbase Developer Platform (CDP) portal under **Secret API Keys**. You can set an **IP allowlist** and permission restrictions, and you must choose the **ECDSA** signature algorithm.
- **Sandbox:** A *static* sandbox: responses are pre-defined examples, and only account and order endpoints are available. It's good for checking your code's format, but it doesn't simulate fills, so it isn't true paper trading.
- **Fees:** Tiered maker/taker fees; check Coinbase's Advanced fee page.

Kraken

Kraken's spot API offers REST and WebSocket access.

- **Key permissions:** Separate switches for Query Funds, Deposit Funds, **Withdraw Funds**, querying orders, Modify Orders, and Cancel/Close Orders. Kraken's own guidance: a trading key would almost certainly not need Withdraw Funds. You can also set IP whitelisting and a key expiration date.
- **Paper trading:** Kraken's command-line tool, **kraken-cli**, includes paper trading against live prices with no API keys, plus a built-in MCP server for AI agents.

5.4 Side by side

BROKER	MARKETS	PAPER/SANDBOX	PYTHON
Alpaca	U.S. stocks, ETFs, options, crypto	Full paper trading, free	<code>alpaca-py</code>
Interactive Brokers	Global stocks, options, futures	Paper account	TWS API
Tradier	U.S. stocks, options	Paper sandbox	REST
tastytrade	U.S. stocks, options, futures	Sandbox (resets daily)	REST
Schwab	U.S. stocks, options	Check current docs	REST
Coinbase	Crypto	Static sandbox only	<code>coinbase-advanced-py</code>
Kraken	Crypto	kraken-cli paper mode	REST

TIP

Not sure? Start with an Alpaca paper account. It costs nothing, takes minutes, and everything in Chapters 14–15 works with it directly. You can pick a live broker later, once you know what you need.

OPEN AN ALPACA PAPER ACCOUNT

- 1 **Sign up.**
Go to alpaca.markets and create an account with your email. Choose the paper-only option if you aren't ready to open a live account.
- 2 **Turn on two-factor authentication.**
Find it in your account's security settings.
- 3 **Open the paper dashboard.**
Make sure the account switcher at the top left shows a paper account.
- 4 **Don't create keys yet.**
You'll do that safely in Chapter 7.

- ☐ I've chosen a broker with an API and a real paper-trading mode
- ☐ I know which fees apply to my market
- ☐ I have an Alpaca paper account (or my chosen broker's equivalent)
- ☐ Two-factor authentication is on

Setting Up Your Computer

Time to build your workshop. You'll install three free tools: **Python** (the programming language your bot is written in), **VS Code** (an editor for writing code), and **Claude Code** (an AI assistant that writes, explains, and reviews code with you).

IN THIS CHAPTER

- Installing Python 3 and VS Code
- Creating a project folder and a virtual environment
- Installing Claude Code and teaching it about your project
- Prompts for using AI as a careful coding partner

6.1 Install Python FREE

INSTALL PYTHON

- 1 Download it.**
Go to python.org/downloads and download the latest Python 3 release for your system. Use Python **3.12 or newer**: the current versions of pandas and vectorbt need at least 3.11.
- 2 Windows: tick "Add python.exe to PATH".**
This box is on the first installer screen. It lets your terminal find Python.
- 3 Mac: run the installer package.**
Then open **Terminal** (press `Cmd` + `Space` and type Terminal).
- 4 Check it worked.**
Open a new terminal window and type `python3 --version` (on Windows, `python --version`). You should see a version number.

A **terminal** is a window where you type commands instead of clicking. You'll only need a handful of commands, and each is explained.

6.2 Install VS Code FREE

Download Visual Studio Code from code.visualstudio.com and install it. Then open it, click the **Extensions** icon in the left bar, search for **Python**, and install the extension published by Microsoft. It adds colour, error hints, and a way to run scripts.

6.3 Create your project

CREATE THE PROJECT FOLDER

1 Make a folder.

In your terminal:

TERMINAL

```
mkdir trading-bot  
cd trading-bot
```

2 Create a virtual environment.

A **virtual environment** is a private box of Python libraries for this one project, so it can't clash with anything else on your computer.

TERMINAL

```
python3 -m venv .venv
```

3 Turn it on (activate it).

On Mac or Linux:

TERMINAL

```
source .venv/bin/activate
```

On Windows PowerShell:

POWERSHELL

```
.venv\Scripts\Activate.ps1
```

Your prompt now starts with `(.venv)`. Do this every time you open a new terminal for the project.

4 Install the libraries.**TERMINAL**

```
pip install alpaca-py pandas python-dotenv requests pytest
```

5 Open the folder in VS Code.

Use **File** > **Open Folder** and pick `trading-bot`. When VS Code asks which Python to use, choose the one inside `.venv`.

Here's what each library does:

LIBRARY	JOB
<code>alpaca-py</code>	Talks to Alpaca: prices, account, orders
<code>pandas</code>	Tables of prices, averages, and statistics
<code>python-dotenv</code>	Loads secret keys from a <code>.env</code> file
<code>requests</code>	Sends alert messages over the internet
<code>pytest</code>	Runs automatic tests of your code

Save the list so you (or a server) can reinstall it exactly:

TERMINAL

```
pip freeze > requirements.txt
```

6.4 Use Git from day one

Git keeps a history of every change to your code, so you can undo mistakes. Install it from git-scm.com, then in your project folder:

TERMINAL

```
git init
```

Create a file named `.gitignore` so secrets and junk never get saved into history:

.GITIGNORE

```
.venv/
.env
state.json
logs/
__pycache__/
KILL_SWITCH
```

⚠ DANGER

The line `.env` is the most important line in that file. Your `.env` file will hold API keys. If it's ever pushed to GitHub, assume the keys are stolen and delete them at the broker immediately.

6.5 Install Claude Code

Claude Code is an AI coding agent that works inside your project folder. It needs a paid Claude plan (Pro or higher) or an Anthropic Console account. On macOS, Linux, or WSL:

TERMINAL

```
curl -fsSL https://claude.ai/install.sh | bash
```

On Windows PowerShell:

POWERSHELL

```
irm https://claude.ai/install.ps1 | iex
```

Open a new terminal, go to your project folder, and type `claude`. Follow the login steps in your browser. (The companion manual *Claude Code* covers this in depth.)

6.6 Teach Claude about your project

Claude Code reads a file named `CLAUDE.md` in your project at the start of every session. Use it to set ground rules:

CLAUDE.MD

```
# Trading bot project rules
- This bot trades REAL money if misconfigured. Be careful.
- Always default to Alpaca PAPER trading (paper=True).
- Never print, log, or commit API keys. Secrets live in .env only.
- Only bot/orders.py may submit orders.
- Every order must pass bot/risk.py checks first.
- Write a pytest test for every new rule or signal.
- Explain changes in plain English for a beginner.
- Never add margin, shorting, options, or leverage.
```

Then block Claude from reading your secrets file. Create `.claude/settings.json`:

`.CLAUDE/SETTINGS.JSON`

```
{
  "permissions": {
    "deny": [
      "Read(./env)"
    ]
  }
}
```

6.7 Working with AI safely

AI is a superb tutor and a fast typist, but it can be confidently wrong, especially about library details that changed recently. Use it like this:

- **Ask it to explain before it writes.** “Explain how you’ll do this, then wait.”
- **Make it test.** Ask for a test with every function.
- **Ask for a review.** A second pass catches many bugs.
- **Check names against the docs.** If an import fails, paste the error back and ask it to check the library’s current documentation.

PROMPT

I'm a complete beginner building a Python trading bot with alpaca-py on a PAPER account. Read CLAUDE.md first. Explain, in plain English, what files a simple daily bot needs and what each one does. Don't write any code yet.

PROMPT

Review bot/risk.py like a cautious senior engineer. List every way this code could let through a trade that is too big, repeated, or on the wrong symbol. Then write pytest tests for each case.

- ☐ `python3 --version` shows 3.12 or newer
- ☐ VS Code is installed with the Python extension
- ☐ My project has a `.env` and the libraries installed
- ☐ `.gitignore` lists `.env`
- ☐ Claude Code starts in my project and has a `CLAUDE.md`
- ☐ Claude is denied from reading `.env`

API Key Safety

An **API key** is a pair of secret codes that lets your bot act on your account without your password. Usually there's a **key ID** (like a username) and a **secret key** (like a password). Anyone holding both can do whatever the key is allowed to do. Treat them like the keys to your house.

IN THIS CHAPTER

- Creating paper keys at Alpaca
- Least privilege: trade-only, no withdrawals
- IP allowlists and expiry dates
- Storing keys in a `.env` file
- What to do if a key leaks

7.1 Create your paper keys

GENERATE ALPACA PAPER KEYS

- 1 Log in and switch to your paper account.**
Check the account switcher at the top left.
- 2 Find the API Keys panel.**
On the dashboard home page it's in the right sidebar.
- 3 Click Generate New Keys.**
Alpaca shows your key ID and secret.
- 4 Copy both into your `.env` file immediately** (next section). The secret is shown only once; if you lose it, regenerate the keys.

NOTE

Paper and live accounts have **different keys**. A paper key can't touch real money. That's a natural safety barrier, so keep paper keys and live keys in separate files and never mix them up.

7.2 The principle of least privilege

Least privilege means giving each key only the powers it truly needs. A trading bot needs to read balances and place and cancel orders. It never needs to **withdraw** or **transfer** money.

EXCHANGE	TURN ON	TURN OFF
Kraken	Query Funds, Query Open Orders & Trades, Modify Orders, Cancel/Close Orders	Withdraw Funds, Deposit Funds
Coinbase (CDP)	View and Trade permissions you need	Transfer
Any broker	Trading for the one account the bot uses	Anything about money movement

Coinbase's documentation spells out why: if someone obtains an API secret with the **transfer** permission, they can send all the crypto out of your account. Without it, a thief can trade badly but can't empty the account.

7.3 IP allowlists

An **IP address** is your server's address on the internet. An **IP allowlist** (also called whitelisting) tells the exchange: "only accept this key from these addresses." Coinbase and Kraken both offer it, and Coinbase recommends it.

LOCK A KEY TO YOUR SERVER

- 1 Find your server's public IP.**
On the server, run `curl -4 ifconfig.me`. (On a home connection the IP can change; a cloud server's is usually fixed.)
- 2 Edit the key.**
In the exchange's API key settings, add that address to the allowlist.
- 3 Test.**
Run the bot once from the server. Then try from your laptop: it should be refused.

Kraken also lets you give a key an **expiration date**. A key that expires after three months forces a regular, healthy rotation.

7.4 Store keys in a .env file

Never type keys into your code. Put them in a file called `.env` in your project folder:

.ENV

```
ALPACA_API_KEY=PKXXXXXXXXXXXXXXXXXXXXX
ALPACA_SECRET_KEY=XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
TELEGRAM_BOT_TOKEN=
TELEGRAM_CHAT_ID=
SLACK_WEBHOOK_URL=
```

Your code loads it with `python-dotenv` :

PYTHON

```
import os
from dotenv import load_dotenv

load_dotenv() # reads .env into the environment
key = os.environ["ALPACA_API_KEY"] # crashes loudly if missing: good
```

Also create a `.env.example` with the same names and no values. That file is safe to commit and reminds you which settings exist.

TIP

On Mac or Linux, make the file readable only by you: `chmod 600 .env`.

7.5 Rules for keys

Do

- Use separate keys for paper and live
- Give live keys trade-only permissions
- Add an IP allowlist where offered
- Rotate keys every few months
- Turn on two-factor authentication on the account

Never

- Paste keys into code, chat apps, screenshots or AI prompts
- Commit `.env` to Git
- Give withdrawal permission to a bot
- Share a key with a “signal service” or third-party bot

7.6 If a key leaks

EMERGENCY RESPONSE

- 1 Delete or regenerate the key at the broker.**
Do this first, before investigating.
- 2 Check recent orders and activity** in the dashboard for anything you didn't do.
- 3 Cancel unexpected open orders** and close unexpected positions.
- 4 Find the leak.**
A public GitHub repo? A pasted log? Fix it.
- 5 Create a new key** with minimal permissions and update `.env`.

- ☐ I've generated Alpaca **paper** keys and saved them in `.env`
- ☐ `.env` is in `.gitignore`, and `.env.example` has no values
- ☐ I know which permissions a live key should never have
- ☐ I know how to add an IP allowlist and what to do after a leak

Market Data

Your bot can only be as good as the prices it sees. **Market data** is the stream of prices and trading volume coming out of the exchanges. In this chapter you'll download your first prices, understand what they mean, and learn how data goes wrong.

IN THIS CHAPTER

- Bars, OHLCV, and timeframes
- Feeds: IEX versus SIP
- Downloading daily bars with `alpaca-py`
- Adjusted prices, missing data, and sanity checks

8.1 Bars and OHLCV

Prices are usually grouped into **bars** (also called candles). A daily bar summarises one trading day with five numbers, known as **OHLCV**:

FIELD	MEANING
Open	First trade price of the period
High	Highest price
Low	Lowest price
Close	Last trade price
Volume	How many shares traded

The bot in this manual uses **daily closes**: one number per day per symbol.

8.2 Feeds: IEX and SIP

U.S. stocks trade on many exchanges at once. The **SIP** (securities information processor) combines all of them. **IEX** is a single exchange. Alpaca's free Basic plan gives you real-time IEX data, while for historical requests it can serve full SIP data as long as you don't ask for the most recent 15 minutes. For a daily bot that reads *yesterday's* closes, the free plan is plenty.

8.3 Your first data script

Create a file named `hello_data.py` :

HELLO_DATA.PY

```
import os
from datetime import datetime, timedelta, timezone

from dotenv import load_dotenv
from alpaca.data.historical import StockHistoricalDataClient
from alpaca.data.requests import StockBarsRequest
from alpaca.data.timeframe import TimeFrame
from alpaca.data.enums import Adjustment

load_dotenv()
client = StockHistoricalDataClient(
    os.environ["ALPACA_API_KEY"], os.environ["ALPACA_SECRET_KEY"]
)

request = StockBarsRequest(
    symbol_or_symbols=["SPY", "TLT"],
    timeframe=TimeFrame.Day,
    start=datetime.now(timezone.utc) - timedelta(days=30),
    adjustment=Adjustment.ALL,
)
bars = client.get_stock_bars(request).df
print(bars.tail(10))
```

Run it with `python hello_data.py`. You'll see a table with a two-level index (symbol, then timestamp) and columns including `open`, `high`, `low`, `close`, and `volume`.

What each part does:

- `StockHistoricalDataClient` is the object that talks to Alpaca's data service.
- `StockBarsRequest` describes what you want: which symbols, what bar size, and from when.
- `Adjustment.ALL` asks for prices **adjusted** for stock splits and dividends (explained below).
- `.df` turns the answer into a pandas **DataFrame**, a table you can calculate with.

Reshape into one column per symbol

Strategies are easier to write when each symbol is its own column:

PYTHON

```
closes = bars["close"].unstack(level=0)  # rows: dates, cols: symbols
closes.index = closes.index.date         # drop the time of day
print(closes.tail())
```

Crypto data

Alpaca's crypto data doesn't even need keys:

PYTHON

```
from alpaca.data.historical import CryptoHistoricalDataClient
from alpaca.data.requests import CryptoBarsRequest

crypto = CryptoHistoricalDataClient()
req = CryptoBarsRequest(
    symbol_or_symbols=["BTC/USD"],
    timeframe=TimeFrame.Day,
    start=datetime.now(timezone.utc) - timedelta(days=30),
)
print(crypto.get_crypto_bars(req).df.tail())
```

Note the slash: Alpaca writes crypto pairs as `BTC/USD`.

8.4 Adjusted prices

When a company does a 4-for-1 **stock split**, the price drops to a quarter overnight, but nobody lost money. Unadjusted data shows a fake 75% crash, and your bot might sell in panic. **Adjusted** data rewrites history to remove that jump, and also accounts for **dividends** (cash payments to shareholders). Always backtest and signal on adjusted prices.

8.5 How data goes wrong

PROBLEM	SYMPTOM	DEFENSE
Missing days	Fewer rows than expected	Require a minimum row count
Today's partial bar	Last "close" is a mid-day price	Drop today's row before computing signals
Bad tick	One absurd price	Reject moves over, say, 25% in a day for an ETF
Stale data	Last date is days old	Check the last date is the previous trading day
Rate limits	HTTP 429 errors	Fewer requests; ask for many symbols at once

A quick sanity check you'll reuse in the bot:

PYTHON

```
def sane(series, max_jump=0.25, min_rows=200):  
    """True if a price series looks usable."""  
    s = series.dropna()  
    if len(s) < min_rows:  
        return False  
    biggest_move = s.pct_change().abs().max()  
    return biggest_move < max_jump
```

⚠ WARNING

Free, unofficial data sources such as the `yfinance` library are handy for quick experiments, but they scrape public websites, can break without warning, and aren't licensed for every use. For anything your bot trades on, use your broker's official data.

8.6 Other data you might want

- **News:** Alpaca's data API includes news articles for stocks and crypto (used in Chapter 10).
- **Market calendar and clock:** Alpaca's trading API tells you if the market is open and lists trading days and early closes (Chapter 16).
- **Fundamentals and alternative data:** Company earnings, sentiment, and so on. Skip these for your first bot.

- ☐ I ran `hello_data.py` and saw real daily bars
- ☐ I can reshape bars into one column per symbol
- ☐ I use adjusted prices for anything my bot decides with
- ☐ I know the five ways data goes wrong and how to check for them

Strategies in Plain English

A **strategy** is a set of rules precise enough that a computer can follow them without asking you. This chapter covers six classic beginner strategies, from the safest to the most active, each with plain-English rules and a short Python function. None of them is a secret money machine; each works in some markets and fails in others.

IN THIS CHAPTER

- DCA and rebalancing bots: the safest autonomous start
- Moving-average trend following
- Mean reversion
- Momentum
- Grid bots for crypto
- How to choose

9.1 Dollar-cost averaging (DCA)

Rule: Every week (or month), buy a fixed dollar amount of the same fund, no matter the price.

Dollar-cost averaging doesn't try to predict anything. When prices are low, your fixed amount buys more shares; when high, fewer. It's the simplest useful bot, and a great first project because there's almost nothing to go wrong.

DCA.PY

```

from alpaca.trading.requests import MarketOrderRequest
from alpaca.trading.enums import OrderSide, TimeInForce

def dca_buy(trading, symbol: str, dollars: float):
    """Buy a fixed dollar amount (fractional shares allowed)."""
    order = MarketOrderRequest(
        symbol=symbol,
        notional=dollars,          # dollars, not shares
        side=OrderSide.BUY,
        time_in_force=TimeInForce.DAY,
    )
    return trading.submit_order(order)

```

`notional` means “spend this many dollars.” Alpaca then buys a **fractional share** (part of a share) if needed. Fractional dollar orders use a `DAY` time in force: the order is valid only for that trading day.

9.2 Rebalancing

Rule: Keep a fixed mix, say 60% stocks and 40% bonds. When the mix drifts more than 5 percentage points from target, trade back to it.

Rebalancing quietly sells what has risen and buys what has fallen. It’s a classic, low-turnover approach and an excellent autonomous bot.

REBALANCE.PY

```

def rebalance_plan(values: dict, targets: dict, band=0.05):
    """Return {symbol: dollars to buy (+) or sell (-)}."""

    values: current dollar value held of each symbol
    targets: desired weights that add up to 1.0
    """
    total = sum(values.values())
    plan = {}
    for sym, weight in targets.items():
        current = values.get(sym, 0.0)
        drift = current / total - weight if total else -weight
        if abs(drift) > band:
            plan[sym] = round(weight * total - current, 2)
    return plan

```

Example: you hold \$7,000 of VTI and \$3,000 of BND, targeting 60/40. VTI is at 70%, ten points over, so the plan says sell \$1,000 of VTI and buy \$1,000 of BND.

PYTHON

```
>>> rebalance_plan({"VTI": 7000, "BND": 3000}, {"VTI": .6, "BND": .4})
{'VTI': -1000.0, 'BND': 1000.0}
```

 TIP

DCA and rebalancing are the safest autonomous start. Many readers should build one of these first, run it for a few months, and only then move on to signal-based strategies.

9.3 Moving-average trend following

Rule: Own the fund when its 50-day average price is above its 200-day average. Otherwise hold cash.

A **moving average** (MA) is the average closing price over the last N days, recalculated each day. **Trend following** assumes that prices that have been rising tend to keep rising for a while. The 50/200 crossover is slow and trades only a few times a year. Its goal is to sidestep long, deep declines, at the cost of being late on both the way in and the way out, and of getting “whipsawed” in choppy markets.

SIGNALS.PY

```
import pandas as pd

def trend_signal(closes: pd.Series, fast=50, slow=200) -> str:
    """'BUY' if the fast average is above the slow one, else 'SELL'.
    Returns 'HOLD' if there isn't enough history yet."""
    closes = closes.dropna()
    if len(closes) < slow:
        return "HOLD"
    fast_ma = closes.rolling(fast).mean().iloc[-1]
    slow_ma = closes.rolling(slow).mean().iloc[-1]
    return "BUY" if fast_ma > slow_ma else "SELL"
```

`rolling(50).mean()` computes the 50-day moving average for every day; `.iloc[-1]` takes the latest one. This is the strategy your full bot uses in Chapters 14 and 15.

9.4 Mean reversion

Rule: When a fund falls unusually far below its recent average, buy; sell when it returns to the average.

Mean reversion bets that stretched prices snap back. A common measure is the **z-score**: how many standard deviations today's price is from its 20-day average. (**Standard deviation** measures how spread out prices usually are.)

PYTHON

```
def zscore_signal(closes: pd.Series, window=20, entry=-2.0) -> str:
    """'BUY' when price is unusually low, 'SELL' when back to normal."""
    closes = closes.dropna()
    if len(closes) < window:
        return "HOLD"
    recent = closes.iloc[-window:]
    z = (closes.iloc[-1] - recent.mean()) / recent.std()
    if z <= entry:
        return "BUY"
    if z >= 0:
        return "SELL"
    return "HOLD"
```

⚠ WARNING

Mean reversion fails badly when a drop is the start of a real collapse. Always pair it with a stop-loss (Chapter 13), and use it on diversified ETFs, not single companies.

9.5 Momentum

Rule: Once a month, rank your watchlist by its return over the past 12 months, skipping the most recent month. Hold the top two.

Momentum compares assets with each other: the recent winners tend to keep outperforming for a while. Skipping the last month avoids short-term reversals.

PYTHON

```
def momentum_picks(closes: pd.DataFrame, top_n=2,
                  lookback=252, skip=21) -> list[str]:
    """Symbols with the best 12-1 month return."""
    if len(closes) < lookback:
        return []
    past = closes.iloc[-lookback]
    recent = closes.iloc[-skip]
    returns = (recent / past - 1).dropna()
    return list(returns.sort_values(ascending=False).index[:top_n])
```

There are about 252 trading days in a year and 21 in a month, which is where those numbers come from.

9.6 Grid bots (crypto)

Rule: Place buy orders at evenly spaced prices below the current price and sell orders above it. Each time a buy fills, place a sell one level higher.

A **grid bot** profits from prices bouncing up and down within a range. It's popular on crypto exchanges.

PYTHON

```
def grid_levels(low: float, high: float, n: int) -> list[float]:
    """n+1 evenly spaced prices from low to high."""
    step = (high - low) / n
    return [round(low + i * step, 2) for i in range(n + 1)]

# grid_levels(60000, 70000, 5)
# -> [60000.0, 62000.0, 64000.0, 66000.0, 68000.0, 70000.0]
```

⚠ DANGER

Grid bots have a nasty failure mode: if the price falls out of the bottom of your range and keeps going, you end up holding a pile of coins bought all the way down. Decide in advance what the bot does below the range (usually: stop and alert you), and make sure fees are smaller than the gap between levels.

9.7 Choosing your first strategy

STRATEGY	TRADES/YEAR	HARDEST PART	BEGINNER FIT
DCA	12–52	Nothing	Best
Rebalancing	1–4	Cash flows	Best
MA trend	2–6 per symbol	Whipsaws	Very good
Mean reversion	10–30	Falling knives	Okay with stops
Momentum	12 rebalances	Turnover, taxes	Good
Grid	Many	Breakouts, fees	Later

i NOTE

A strategy's rules are only a hypothesis. Chapter 11 shows you how to test whether a strategy *would* have worked, and why “would have” is a weaker promise than it sounds.

PROMPT

Explain the 50/200-day moving-average strategy to a beginner. Then list three market conditions where it tends to lose money, with a historical example of each. Be honest about its weaknesses.

- ☐ I can describe each strategy's rule in one sentence
- ☐ I understand why DCA and rebalancing are the safest start
- ☐ I've chosen a first strategy to test (the MA trend is the default here)
- ☐ I know each strategy's main way of failing

Where AI Really Fits

This is the chapter the ads are about. Here's what AI can and can't do for a trading bot today, with working examples and the guardrails each one needs.

IN THIS CHAPTER

- Four honest roles for AI
- Scoring news with a language model
- Machine-learning price predictions and their limits
- AI agents that trade through MCP servers
- Guardrails for every AI component

10.1 Four roles, ranked by safety

1. **Coding partner** (safest): AI writes, explains, tests and reviews code. You decide what runs.
2. **Analyst**: AI reads text, such as news, and produces a score your rules can use as one input.
3. **Predictor**: a machine-learning model guesses tomorrow's direction from past data.
4. **Agent trader** (riskiest): an AI agent decides and places trades itself.

10.2 Role 2: scoring news with a language model

A **large language model (LLM)** such as Claude reads and writes text. It's good at summarising a headline's tone. A sensible use is as a **filter**: "skip new buys in this fund if today's news about it is strongly negative." The model never places orders.

First, fetch headlines from Alpaca's news API:

NEWS.PY

```
import os
from alpaca.data.historical.news import NewsClient
from alpaca.data.requests import NewsRequest

news = NewsClient(os.environ["ALPACA_API_KEY"],
                  os.environ["ALPACA_SECRET_KEY"])

def headlines(symbol: str, limit=10) -> List[str]:
    result = news.get_news(NewsRequest(symbols=symbol, limit=limit))
    return [a.headline for a in result.data["news"]]
```

Then ask Claude to classify them. Install the official library with `pip install anthropic` and put an `ANTHROPIC_API_KEY` from the Claude Console in your `.env`. API use is billed per token, separately from Claude subscriptions; the smallest current model, Claude Haiku 5.5, is the cheapest choice for simple classification.

SENTIMENT.PY

```
import anthropic

client = anthropic.Anthropic() # reads ANTHROPIC_API_KEY
ALLOWED = {"positive", "neutral", "negative"}

def news_mood(symbol: str, items: List[str]) -> str:
    if not items:
        return "neutral"
    text = "\n".join(f"- {h}" for h in items)
    msg = client.messages.create(
        model="claude-haiku-5-5",
        max_tokens=5,
        messages=[{"role": "user", "content":
            f"Headlines about {symbol}:\n{text}\n\n"
            "Overall tone for the fund's price over the next "
            "few days? Answer one word: positive, neutral, "
            "or negative."}],
    )
    answer = msg.content[0].text.strip().lower()
    return answer if answer in ALLOWED else "neutral"
```

Notice the last line: if the model answers anything unexpected, the code treats it as “neutral.” That’s a **guardrail**, a rule that keeps a component’s mistakes harmless.

⚠ WARNING

Headlines are often old news by the time you read them; professionals react in milliseconds. Treat a news score as a weak, unproven input. Backtest it if you can (historical news is available from Alpaca's news API), and paper-trade it before trusting it.

10.3 Role 3: machine-learning predictions

Machine learning (ML) means letting a program find patterns in data instead of writing rules by hand. It's tempting: feed in years of prices, get a model that predicts tomorrow.

The honest limits:

PROBLEM	WHY IT BITES
Low signal	Daily price changes are mostly noise; even 52% accuracy is hard to get
Overfitting	Models memorise the past beautifully and fail on new data
Changing markets	Patterns stop working once others discover them
Leakage	It's easy to accidentally feed the model future information
Costs	A small edge disappears after fees and slippage

If you experiment with ML, treat it as research: split data by time (train on older years, test on newer ones you never touched), compare against a simple baseline such as “always buy and hold,” and include costs. If the model can't clearly beat the baseline on unseen years, don't trade it.

10.4 Role 4: AI agents that trade

An **AI agent** is an AI that can take actions, not just talk. **MCP** (Model Context Protocol) is a standard way to connect AI apps to outside tools. Alpaca publishes an official **Trading API MCP server**, so you can ask Claude, in plain English, to check your account or place a paper order.

According to Alpaca's docs, the server exposes 65 tools, including reading account info and positions, market data, news, and placing and cancelling orders. Setup for Claude Code (it needs Python 3.10+ and `uv`):

TERMINAL

```
claude mcp add alpaca --scope user --transport stdio uvx alpaca-mcp-server \  
  --env ALPACA_API_KEY=your_paper_key \  
  --env ALPACA_SECRET_KEY=your_paper_secret
```

Install `uv` first with `curl -LsSf https://astral.sh/uv/install.sh | sh` (Mac/Linux) or `irm https://astral.sh/uv/install.ps1 | iex` (Windows PowerShell).

Two settings matter:

VARIABLE	DEFAULT	ADVICE
<code>ALPACA_PAPER_TRADE</code>	<code>true</code>	Leave it. Setting <code>false</code> means live trading
<code>ALPACA_TOOLSETS</code>	<code>all</code>	Limit to what's needed, e.g. <code>account,stock-data,news</code> for read-only

Kraken's `kraken-cli` also includes an MCP server with a paper-trading mode, and its default service set includes paper trading but asks for confirmation before “dangerous” calls.

Guardrails for agents

Claude Code lets you block or require approval for specific MCP tools in `.claude/settings.json`. MCP tools are named `mcp__<server>__<tool>`:

`.CLAUDE/SETTINGS.JSON`

```
{
  "permissions": {
    "allow": ["mcp__alpaca__get_*"],
    "ask": ["mcp__alpaca__place_*", "mcp__alpaca__cancel_*"],
    "deny": ["mcp__alpaca__close_all_positions"]
  }
}
```

With this file, Claude can read freely, must ask you before placing or cancelling any order, and can never liquidate everything.

Agent guardrail rules

- Paper keys only, until months of clean results
- Read-only toolsets unless trading is the point
- Human approval for every order (ask rules)
- Hard limits live in code the agent can't edit
- A daily dollar cap and a small account size
- Log every action the agent takes

⚠ DANGER

Don't give an AI agent live keys and walk away. Language models can misread a number, act on a misleading headline, or be manipulated by text hidden inside a web page or news article (a **prompt injection**). If an agent ever trades live, it should do so inside an account so small that its worst possible mistake is a lesson, not a loss.

10.5 Prompts for using AI well

PROMPT

Act as a sceptical quant reviewer. Here is my strategy and backtest summary: [paste]. List the five most likely reasons these results won't hold up in live trading, and a test for each.

PROMPT

Using the Alpaca MCP tools on my PAPER account only, show my positions and today's P/L. Do not place, cancel, or change any orders.

- ☐ I know the four roles AI can play, and which is safest
- ☐ Any AI score in my bot is a filter with a safe default
- ☐ I understand why ML price prediction rarely beats a simple baseline
- ☐ If I connect an AI agent, it uses paper keys and asks before trading

Backtesting Step by Step

A **backtest** replays your strategy over historical prices to see how it *would* have done. It's the cheapest way to throw away bad ideas. It's also the easiest place to fool yourself, so this chapter spends as long on the traps as on the code.

IN THIS CHAPTER

- Downloading history and saving it to a file
- A from-scratch backtest in pandas, with costs
- The numbers that matter: CAGR, drawdown, Sharpe
- The same test in vectorbt; a note on backtrader
- Overfitting, look-ahead bias, and walk-forward testing

11.1 Step 1: Save history to a file

Download once, then test as often as you like without hitting the API:

DOWNLOAD.PY

```

import os
from datetime import datetime
from dotenv import load_dotenv
from alpaca.data.historical import StockHistoricalDataClient
from alpaca.data.requests import StockBarsRequest
from alpaca.data.timeframe import TimeFrame
from alpaca.data.enums import Adjustment

load_dotenv()
client = StockHistoricalDataClient(os.environ["ALPACA_API_KEY"],
                                   os.environ["ALPACA_SECRET_KEY"])

req = StockBarsRequest(
    symbol_or_symbols=["SPY", "QQQ", "IWM", "EFA", "TLT", "GLD"],
    timeframe=TimeFrame.Day,
    start=datetime(2016, 1, 1),
    adjustment=Adjustment.ALL,
)
closes = client.get_stock_bars(req).df["close"].unstack(level=0)
closes.index = closes.index.date
closes.to_csv("closes.csv")
print(closes.shape, "saved")

```

11.2 Step 2: A backtest in pandas

The idea: compute the signal each day, hold the position **from the next day**, and charge costs whenever the position changes.

BACKTEST.PY

```

import pandas as pd

def backtest_ma(close: pd.Series, fast=50, slow=200,
                cost=0.001) -> pd.Series:
    """Daily equity curve of the MA strategy, starting at 1.0.

    cost: fraction lost per trade (fees + slippage), 0.1% here.
    """
    fast_ma = close.rolling(fast).mean()
    slow_ma = close.rolling(slow).mean()
    signal = (fast_ma > slow_ma).astype(int)
    position = signal.shift(1).fillna(0) # act the NEXT day
    daily_ret = close.pct_change().fillna(0)
    trades = position.diff().abs().fillna(0)
    strat_ret = position * daily_ret - trades * cost
    return (1 + strat_ret).cumprod()

```

Line by line:

- `signal` is 1 on days the fast average is above the slow one, else 0.
- `shift(1)` moves the signal forward one day. You only know today's close *after* the market shuts, so the earliest you can act is tomorrow. Forgetting this is called **look-ahead bias**, the most common backtest bug.
- `daily_ret` is each day's percentage change.
- `trades` is 1 on days the position changes; each change costs `cost`.
- `cumprod()` compounds the daily returns into an **equity curve**: what \$1 grew into.

11.3 Step 3: Measure it

STATS.PY

```
import numpy as np

def stats(equity: pd.Series) -> dict:
    days = len(equity)
    years = days / 252
    cagr = equity.iloc[-1] ** (1 / years) - 1
    peak = equity.cummax()
    max_dd = (equity / peak - 1).min()
    ret = equity.pct_change().dropna()
    sharpe = np.sqrt(252) * ret.mean() / ret.std()
    return {"CAGR": round(cagr, 4), "MaxDD": round(max_dd, 4),
            "Sharpe": round(sharpe, 2)}
```

NUMBER	MEANING	ROUGH GUIDE
CAGR	Compound annual growth rate: the average yearly return	Compare with buy-and-hold
Max drawdown	Worst fall from a peak, e.g. -0.33 = -33%	Could you sit through it?
Sharpe ratio	Return per unit of volatility (ignoring the risk-free rate here)	Above 1 is good; above 2 is suspicious

Now compare with simply holding:

PYTHON

```
closes = pd.read_csv("closes.csv", index_col=0, parse_dates=True)
spy = closes["SPY"].dropna()
strategy = backtest_ma(spy)
hold = spy / spy.iloc[0]
print("Strategy:", stats(strategy))
print("Buy&hold:", stats(hold))
```

☆ IMPORTANT

Always compare against a **benchmark**, usually buy-and-hold of the same fund. A strategy that made 9% a year sounds good until you learn holding made 13%. Trend strategies often earn *less* than holding but with smaller drawdowns. Whether that trade-off suits you is a personal decision.

11.4 Step 4: The same test in vectorbt

vectorbt is a fast open-source backtesting library (`pip install vectorbt`; it needs Python 3.11 or newer). Once you understand the pandas version, it saves time:

PYTHON

```
import vectorbt as vbt

fast = vbt.MA.run(spy, 50)
slow = vbt.MA.run(spy, 200)
entries = fast.ma_crossed_above(slow).vbt.fshift(1)
exits = fast.ma_crossed_below(slow).vbt.fshift(1)

pf = vbt.Portfolio.from_signals(
    spy, entries, exits,
    init_cash=10_000, fees=0.0005, slippage=0.0005, freq="1D",
)
print(pf.stats())
```

`fshift(1)` shifts the signals forward a day, for the same look-ahead reason as before. `fees` and `slippage` are fractions per trade.

📘 NOTE

You'll see **backtrader** in many tutorials. It works, but its last release was in April 2023, so expect rough edges with new Python versions. **backtesting.py** (`pip install backtesting`) is another simple option; note its AGPL licence if you ever plan to distribute your code.

11.5 The traps

Fees and slippage

Real fills are worse than the closing price. Include at least 0.05%–0.1% per trade for liquid ETFs, more for crypto (where the explicit fee alone can be 0.25% at Alpaca's lowest tier). Rerun your test at double the costs: if the strategy dies, it was never robust.

Overfitting

Overfitting means tuning rules until they fit the past perfectly, including its random noise. If you try 500 combinations of averages and keep the best, the best one is probably lucky.

Warning signs:

- Results change a lot when you nudge a parameter (49 days great, 51 days terrible)
- Very few trades
- A Sharpe ratio above 2 on daily data
- Works on one fund only

Look-ahead and survivorship bias

Look-ahead bias is using information you wouldn't have had: same-day closes, revised data, or future index membership. **Survivorship bias** is testing only on companies that exist today, ignoring the ones that went bust. ETFs on broad indexes reduce, but don't remove, both problems.

11.6 Walk-forward testing

Walk-forward testing mimics real life: pick parameters using only the past, then test them on the next, unseen period, and roll forward.

WALKFORWARD.PY

```
def walk_forward(close, fasts=(20, 50, 100), slows=(150, 200),
                train_years=3, test_years=1):
    results = []
    start = close.index[0].year
    end = close.index[-1].year
    for y in range(start + train_years, end + 1, test_years):
        train = close[str(y - train_years):str(y - 1)]
        best = max(((f, s) for f in fasts for s in slows if f < s),
                  key=lambda p: backtest_ma(train, *p).iloc[-1])
        window = close[str(y - 1):str(y)]      # warm-up + test year
        eq = backtest_ma(window, *best)
        test_eq = eq[str(y)] / eq[str(y)].iloc[0]
        results.append((y, best, round(test_eq.iloc[-1] - 1, 4)))
    return results
```

Each row shows a year, the parameters chosen *before* it, and how they did *in* it. Only the out-of-sample years count. If those years look much worse than your original backtest, you were overfitting.

 TIP

Simple rules with few parameters, tested on many symbols and many years, are the ones most likely to survive contact with the future.

PROMPT

Here's my backtest code: [paste]. Check it for look-ahead bias, missing costs, and any way it uses information from the future. Explain each problem in plain English and show the fix.

- ☐ I have a saved price file and a working pandas backtest
- ☐ My backtest acts the day after the signal and charges costs
- ☐ I compared it with buy-and-hold
- ☐ I reran it with double costs and nearby parameters
- ☐ I ran a walk-forward test and judged the out-of-sample years

Paper Trading for Weeks

A backtest tells you how rules behaved on old data. **Paper trading** tells you how your *whole system* behaves today: the scheduler, the data, the orders, the logs, the alerts, and you. It's where you find the bugs that history can't show.

IN THIS CHAPTER

- Why paper trading comes before any real money
- How long to run it
- Success criteria you write down in advance
- A weekly review routine

12.1 What paper trading catches

PROBLEM	BACKTEST SEES IT?	PAPER TRADING SEES IT?
Strategy logic	Yes	Yes
Scheduler didn't run	No	Yes
Orders rejected (wrong size, wrong symbol)	No	Yes
Holidays and early closes	Rarely	Yes
Alerts not arriving	No	Yes
Fills worse than closes	Only if modelled	Partly
Your own urge to "fix" things	No	Yes

12.2 How long?

Run the bot on paper for **at least eight weeks**, and longer for slow strategies. A daily 50/200 trend bot may trade only a few times in that period, so the main goal is proving the *machinery* works: dozens of clean runs, at least one holiday, at least one order of each type, and at least one alert.

TIP

Speed up the machinery test without changing your real strategy: run a second paper account with a deliberately twitchy setting (such as 5/20-day averages) so orders happen often. Delete it afterwards.

12.3 Write your success criteria first

Decide *before* you start what “pass” means, and put it in a file so you can’t move the goalposts later:

PAPER-CRITERIA.MD

```
# Paper trading pass criteria (written 2026-10-17)
- [ ] 40+ scheduled runs, zero missed without an alert
- [ ] Zero unintended orders (wrong symbol, size, or duplicate)
- [ ] Every order logged with ID, size, price, and reason
- [ ] Risk limits tested: kill switch, daily loss, max positions
- [ ] Paper fills within 0.2% of the price the bot expected
- [ ] Paper results roughly match a backtest of the same period
- [ ] I read every daily report and understood every trade
```

Notice there’s nothing about making money. In eight weeks, profit or loss is mostly luck. You’re testing reliability and honesty.

12.4 Compare paper with the backtest

At the end, rerun your backtest on exactly the paper period. If the backtest says three trades and the bot made seven, something differs: data, timing, or a bug. Track it down before going further.

12.5 The weekly review

EVERY WEEKEND

- 1 Read the log.**
Skim `logs/bot.log` for warnings and errors.
- 2 Reconcile.**
Compare the bot's trade log with the broker's order history. They should match one for one.
- 3 Check limits.**
Did any risk rule fire? Was it correct?
- 4 Note surprises.**
Write one line per surprise in a journal.
- 5 Change at most one thing.**
And only with a test proving the fix.

PROMPT

Here is a week of my bot's log and the broker's order history:
[paste both]. Do they match exactly? List any order in one but not the other, any errors, and anything that looks unusual.

- ☐ I wrote pass criteria before starting
- ☐ My bot ran on paper for at least eight weeks
- ☐ Broker history and my trade log match exactly
- ☐ I compared paper results with a backtest of the same period
- ☐ Every criterion is ticked, or I've fixed the cause and restarted the clock

Risk Management in Code

Strategies decide what you'd *like* to do. Risk rules decide what you're *allowed* to do. In a well-built bot, the risk module can veto anything, and nothing reaches the broker without passing it. This chapter explains each rule; Chapter 15 puts them in the bot.

IN THIS CHAPTER

- Position sizing and the 1–2% rule
- Stop-losses
- Daily loss limits and maximum drawdown
- Maximum positions and order caps
- Kill switches and circuit breakers

13.1 Position sizing: the 1–2% rule

Position sizing answers “how much do I buy?” The **1–2% rule** says: size each position so that if your stop-loss is hit, you lose no more than 1% (or at most 2%) of your account.

The math:

- **Risk dollars** = account equity × risk per trade
- **Loss per share** = entry price × stop distance
- **Shares** = risk dollars ÷ loss per share

Worked example. Account: \$10,000. Risk per trade: 1% = \$100. Fund price: \$500. Stop 8% below entry, so a stop-out loses \$40 per share. $100 \div 40 = 2.5$, rounded **down** to 2 shares. Add a cap too: no single position over 20% of the account (\$2,000, or 4 shares). The smaller answer wins: **2 shares**.

BOT/RISK.PY (PART 1)

```
import math

def position_size(equity, price, risk_pct, stop_pct, max_pos_pct):
    """Whole shares so that a stop-out loses about risk_pct."""
    if price <= 0 or stop_pct <= 0:
        return 0
    by_risk = (equity * risk_pct) / (price * stop_pct)
    by_cap = (equity * max_pos_pct) / price
    return max(0, math.floor(min(by_risk, by_cap)))
```

 NOTE

Using 1% doesn't mean you can only lose 1% per trade. Prices can **gap** past your stop overnight. It means a *typical* bad trade costs about 1%, so ten losses in a row cost about 10%, not your account.

13.2 Stop-losses

A **stop-loss** (or **stop order**) is an order that sells automatically if the price falls to a level you choose. Your bot places it with the broker at the same time as the buy, so it protects you even if your server is down.

At Alpaca, an **OTO** order (one-triggers-other) submits a buy plus a stop that activates once the buy fills. A **bracket** order adds a take-profit too. Bracket legs must use `day` or `gtc` time in force. **GTC** (good-till-cancelled) orders stay open until filled or cancelled, but Alpaca cancels GTC orders automatically 90 days after creation, so a long-running bot must check its stops still exist.

 WARNING

A stop becomes a market order when triggered, so in a fast fall you can be filled well below your stop price. Stops limit losses; they don't guarantee a price.

13.3 Daily loss limit

A **max daily loss** stops trading for the rest of the day if the account has dropped too much since yesterday's close. Alpaca's account object includes `equity` (now) and `last_equity` (at the previous close), which makes this easy:

BOT/RISK.PY (PART 2)

```
def daily_loss_breached(equity, last_equity, max_loss_pct):
    if last_equity <= 0:
        return False
    return (last_equity - equity) / last_equity >= max_loss_pct

def drawdown_breached(equity, peak_equity, max_dd_pct):
    if peak_equity <= 0:
        return False
    return (peak_equity - equity) / peak_equity >= max_dd_pct
```

Maximum drawdown is the fall from the account's highest value. If it passes your limit, say 15%, the bot halts until you review it. That's a **circuit breaker**: a rule that stops the system when results suggest something is wrong.

13.4 Position and order limits

BOT/RISK.PY (PART 3)

```
def approve_buy(symbol, qty, price, cash, n_open, rules, allowed):
    """Return (ok, reason). Every new buy must pass this."""
    if symbol not in allowed:
        return False, "symbol not on the approved list"
    if qty < 1:
        return False, "size rounds to zero shares"
    if n_open >= rules["max_open_positions"]:
        return False, "too many open positions"
    if qty * price > cash:
        return False, "not enough settled cash"
    if qty * price > rules["max_order_dollars"]:
        return False, "order bigger than the per-order cap"
    return True, "ok"
```

Each line blocks a specific disaster: trading an unexpected symbol, a zero-size order, over-diversifying into too many positions, spending money you don't have, and a giant order caused by a bug.

The bot also caps **orders per run**. If a bug makes it think every symbol needs a trade, it stops after a handful instead of sending hundreds.

13.5 The kill switch

A **kill switch** is a way to stop the bot and flatten everything instantly. The simplest version is a file: if a file named `KILL_SWITCH` exists in the project folder, the bot cancels all orders, closes all positions, alerts you, and exits.

BOT/RISK.PY (PART 4)

```
from pathlib import Path

KILL_FILE = Path(__file__).resolve().parent.parent / "KILL_SWITCH"

def kill_switch_on() -> bool:
    return KILL_FILE.exists()
```

To pull it from a server: `touch KILL_SWITCH`. To pull it from anywhere, you can also log in to the broker's website and close positions by hand; practise both.

☆ IMPORTANT

Rehearse the kill switch on paper. Create the file, run the bot, confirm everything closes, delete the file. A safety feature you've never tested is a hope, not a feature.

13.6 The full rule set

CONFIG.TOML ([RISK] SECTION)

```
[risk]
risk_per_trade = 0.01      # 1% of equity at risk per position
stop_loss_pct = 0.08       # protective stop 8% below entry
max_position_pct = 0.20    # no position over 20% of equity
max_open_positions = 5
max_order_dollars = 2500   # hard cap per order
max_daily_loss_pct = 0.03  # halt for the day if down 3%
max_drawdown_pct = 0.15   # halt until reviewed if 15% below peak
max_orders_per_run = 5
```

- ☐ I can calculate a position size by hand with the 1% rule
- ☐ Every buy will carry a broker-side stop-loss
- ☐ Order size, position count, orders per run are capped; I can pull the kill switch

Build the Bot, Part 1: Foundations

Now you'll assemble everything into one real, autonomous bot that trades on Alpaca's paper API. It runs once per trading day, follows the 50/200-day trend rule on a small ETF watchlist, and refuses anything its risk rules don't approve. This chapter sets up the project, configuration, data, and signals. Chapter 15 adds risk checks, orders, state, logging, the main loop, and tests.

IN THIS CHAPTER

- The project layout and why it's split into modules
- `config.toml` and the settings loader
- The data module
- The signal module
- Building it with Claude Code

14.1 The project layout

TEXT

```
trading-bot/
├── .env                # secrets (never committed)
├── .env.example        # names of secrets, no values
├── .gitignore
├── config.toml         # your rules: symbols, risk limits
├── requirements.txt
├── bot/
│   ├── __init__.py    # empty file: marks a Python package
│   ├── config.py      # loads config.toml and .env
│   ├── data.py        # fetches daily prices
│   ├── signals.py     # strategy: BUY / SELL / HOLD
│   ├── risk.py        # every rule that can say no
│   ├── orders.py      # the ONLY code that sends orders
│   ├── state.py       # remembers things between runs
│   ├── notify.py      # Telegram / Slack alerts
│   ├── logs.py        # log file setup and trade log
│   └── main.py        # one run, start to finish
├── tests/
│   ├── test_signals.py
│   └── test_risk.py
└── logs/              # created automatically
```

Why so many files? Each one has one job, so you can test it alone and read it in a minute. Most importantly, **only** `orders.py` **talks to the broker's order system**. If you ever wonder “what can place a trade?”, there's exactly one place to look.

Create the folders and the empty package file:

TERMINAL

```
mkdir -p bot tests logs
touch bot/__init__.py
```

(On Windows PowerShell, create `bot\__init__.py` as an empty file from VS Code instead of `touch .`)

14.2 Step 1: The config file

Your rules live in a **TOML** file, a simple settings format of names and values that Python can read without extra libraries.

CONFIG.TOML

```
[bot]
paper = true
symbols = ["SPY", "QQQ", "IWM", "EFA", "TLT", "GLD"]
fast_days = 50
slow_days = 200

[risk]
risk_per_trade = 0.01
stop_loss_pct = 0.08
max_position_pct = 0.20
max_open_positions = 5
max_order_dollars = 2500
max_daily_loss_pct = 0.03
max_drawdown_pct = 0.15
max_orders_per_run = 5
```

Changing your rules means editing this file, not the code. Commit it to Git so every change to your rules is recorded.

14.3 Step 2: The settings loader

BOT/CONFIG.PY

```
"""Load rules from config.toml and secrets from .env."""
import os
import toml
from dataclasses import dataclass
from pathlib import Path

from dotenv import load_dotenv

ROOT = Path(__file__).resolve().parent.parent

@dataclass(frozen=True)
class Settings:
    api_key: str
    secret_key: str
    paper: bool
    symbols: list
    fast_days: int
    slow_days: int
    risk: dict
```

A **dataclass** is a tidy container for related values. `frozen=True` means nothing can change the settings by accident while the bot runs.

BOT/CONFIG.PY (CONTINUED)

```
def load_settings() -> Settings:
    load_dotenv(ROOT / ".env")
    with open(ROOT / "config.toml", "rb") as f:
        cfg = toml.load(f)
    paper = cfg["bot"].get("paper", True)
    if not paper and os.getenv("ALLOW_LIVE") != "yes":
        raise SystemExit("Live trading needs ALLOW_LIVE=yes in .env")
    return Settings(
        api_key=os.getenv("ALPACA_API_KEY"),
        secret_key=os.getenv("ALPACA_SECRET_KEY"),
        paper=paper,
        symbols=cfg["bot"]["symbols"],
        fast_days=cfg["bot"]["fast_days"],
        slow_days=cfg["bot"]["slow_days"],
        risk=cfg["risk"],
    )
```

The `ALLOW_LIVE` check is a **two-key lock**: switching to real money requires changing `config.toml` and adding a line to `.env`. One accidental edit isn't enough.

14.4 Step 3: The data module

BOT/DATA.PY

```
"""Fetch completed daily closes from Alpaca."""
from datetime import datetime, timedelta, timezone
from zoneinfo import ZoneInfo

from alpaca.data.enums import Adjustment
from alpaca.data.requests import StockBarsRequest
from alpaca.data.timeframe import TimeFrame

NY = ZoneInfo("America/New_York")

def get_daily_closes(client, symbols, days):
    """DataFrame of closes: one column per symbol, one row per day.
    Today's unfinished bar is dropped."""
    start = datetime.now(timezone.utc) - timedelta(days=int(days * 1.6))
    req = StockBarsRequest(
        symbol_or_symbols=symbols,
        timeframe=TimeFrame.Day,
        start=start,
        adjustment=Adjustment.ALL,
    )
    bars = client.get_stock_bars(req).df
    closes = bars["close"].unstack(level=0)
    closes.index = closes.index.tz_convert(NY).date
    today = datetime.now(NY).date()
    return closes[closes.index < today]
```

Two details worth understanding:

- `days * 1.6` asks for extra calendar days, because 200 *trading* days need roughly 290 calendar days once you skip weekends and holidays.
- The last line drops today's row. The bot runs during the trading day, when today's "close" isn't final yet. Using only completed days keeps the bot identical to the backtest.

14.5 Step 4: The signal module

This is the same function from Chapter 9, plus the sanity check from Chapter 8:

BOT/SIGNALS.PY

```
"""Strategy logic. Pure functions: no network, easy to test."""
import pandas as pd

def sane(series: pd.Series, min_rows: int, max_jump=0.25) -> bool:
    s = series.dropna()
    if len(s) < min_rows:
        return False
    return s.pct_change().abs().max() < max_jump

def trend_signal(closes: pd.Series, fast=50, slow=200) -> str:
    closes = closes.dropna()
    if len(closes) < slow:
        return "HOLD"
    fast_ma = closes.rolling(fast).mean().iloc[-1]
    slow_ma = closes.rolling(slow).mean().iloc[-1]
    return "BUY" if fast_ma > slow_ma else "SELL"
```

“Pure function” means it only uses its inputs and returns an answer, without touching the internet, files, or the clock. That’s why it’s trivial to test.

14.6 Building it with Claude Code

You can type these files yourself, or ask Claude Code to create them, one module at a time:

PROMPT

Read CLAUDE.md. Create bot/config.py and config.toml exactly as described: a frozen Settings dataclass, tomlib for config, dotenv for secrets, and a two-key lock (paper=false only works if ALLOW_LIVE=yes). Explain each line for a beginner. Don't create any other files.

PROMPT

Create bot/data.py with get_daily_closes(client, symbols, days) using alpaca-py's StockBarsRequest with TimeFrame.Day and Adjustment.ALL. Drop today's unfinished bar using the America/New_York date. Check the current alpaca-py docs for class names before writing.

 TIP

After each file, run a quick check before moving on: `python -c "from bot.config import load_settings; print(load_settings().symbols)"`. Small steps make errors easy to find.

- ☐ My project has the folder layout above
- ☐ `config.toml` holds my symbols and risk limits
- ☐ `load_settings()` loads without errors and refuses live mode without `ALLOW_LIVE`
- ☐ `get_daily_closes()` returns one column per symbol, ending yesterday
- ☐ `trend_signal()` returns BUY, SELL or HOLD

Build the Bot, Part 2: Risk, Orders and the Main Loop

With data and signals in place, you'll now add the parts that keep the bot safe and accountable, then wire everything together into one run.

IN THIS CHAPTER

- The risk module, assembled
- The state file
- The orders module: buys with stops, clean exits
- Logging and a trade log
- Alerts
- The main loop, a dry-run mode, and tests

15.1 Step 5: The risk module

Put the four parts from Chapter 13 into one file, `bot/risk.py`: `position_size`, `daily_loss_breached`, `drawdown_breached`, `approve_buy`, and `kill_switch_on`. Put the imports (`import math` and `from pathlib import Path`) at the top. Nothing in this file talks to the broker; it only answers yes or no.

15.2 Step 6: The state file

The broker remembers your positions and orders. A few things it doesn't know, such as the account's highest-ever value or whether you halted the bot, go in a small **JSON** file (a common text format for structured data).

BOT/STATE.PY

```
"""A tiny JSON file that remembers things between runs."""
import json
from pathlib import Path

ROOT = Path(__file__).resolve().parent.parent
STATE_FILE = ROOT / "state.json"
DEFAULT = {"peak_equity": 0.0, "halted": False,
           "halt_reason": "", "last_run": ""}

def load_state(path: Path = STATE_FILE) -> dict:
    if path.exists():
        return {**DEFAULT, **json.loads(path.read_text())}
    return dict(DEFAULT)

def save_state(state: dict, path: Path = STATE_FILE) -> None:
    tmp = path.with_suffix(".tmp")
    tmp.write_text(json.dumps(state, indent=2))
    tmp.replace(path)  # swap in one step: never half-written
```

Writing to a temporary file and then renaming it is called an **atomic write**. If the server loses power mid-save, you get either the old file or the new one, never a corrupted mix.

Once `halted` is `true`, the bot does nothing until *you* edit the file back to `false` after investigating. That's deliberate.

15.3 Step 7: The orders module

BOT/ORDERS.PY

```

"""The ONLY module allowed to send orders to the broker."""
import time

from alpaca.trading.enums import (OrderClass, OrderSide,
                                   QueryOrderStatus, TimeInForce)
from alpaca.trading.requests import (GetOrdersRequest,
                                     MarketOrderRequest,
                                     StopLossRequest)

def buy_with_stop(trading, symbol, qty, ref_price, stop_pct, tag):
    """Market buy of whole shares plus a broker-side stop-loss."""
    stop_price = round(ref_price * (1 - stop_pct), 2)
    order = MarketOrderRequest(
        symbol=symbol,
        qty=qty,
        side=OrderSide.BUY,
        time_in_force=TimeInForce.GTC,
        order_class=OrderClass.OTO,
        stop_loss=StopLossRequest(stop_price=stop_price),
        client_order_id=f"{tag}-{symbol}-buy",
    )
    return trading.submit_order(order)

```

What's happening:

- An **OTO** order sends the buy and, once it fills, activates the stop-loss at the broker.
- `stop_price` is 8% (your `stop_loss_pct`) below yesterday's close.
- `client_order_id` is *your* label for the order, built from the date, so every order in the broker's history traces back to a specific run. Alpaca treats it as an identifier you supply for each order; using the date means a second run on the same day can't quietly create a look-alike order.

Selling has one wrinkle: the shares are already reserved by the open stop order, so you must cancel it first.

BOT/ORDERS.PY (CONTINUED)

```
def cancel_open_orders(trading, symbol, wait_s=10):
    req = GetOrdersRequest(status=QueryOrderStatus.OPEN,
                           symbols=[symbol])
    for o in trading.get_orders(filter=req):
        trading.cancel_order_by_id(o.id)
    for _ in range(wait_s):      # wait for cancels to finish
        if not trading.get_orders(filter=req):
            return
        time.sleep(1)
    raise RuntimeError(f"Open orders for {symbol} did not cancel")

def sell_all(trading, symbol):
    cancel_open_orders(trading, symbol)
    return trading.close_position(symbol)
```

If cancelling takes too long, the function raises an error instead of guessing. The main loop catches it, logs it, and alerts you.

15.4 Step 8: Logging and the trade log

BOT/LOGS.PY

```
"""Human-readable log plus a CSV trade log for taxes and review."""
import csv
import logging
from datetime import datetime, timezone
from logging.handlers import RotatingFileHandler
from pathlib import Path

LOG_DIR = Path(__file__).resolve().parent.parent / "logs"

def setup_logging():
    LOG_DIR.mkdir(exist_ok=True)
    fmt = "%(asctime)s %(levelname)s %(name)s: %(message)s"
    file = RotatingFileHandler(LOG_DIR / "bot.log",
                               maxBytes=1_000_000, backupCount=5)
    logging.basicConfig(level=logging.INFO, format=fmt,
                        handlers=[file, logging.StreamHandler()])

def record_trade(symbol, side, qty, price, reason, order_id):
    path = LOG_DIR / "trades.csv"
    new = not path.exists()
    with open(path, "a", newline="") as f:
        w = csv.writer(f)
        if new:
            w.writerow(["time_utc", "symbol", "side", "qty",
                       "ref_price", "reason", "order_id"])
        now = datetime.now(timezone.utc).isoformat(timespec="seconds")
        w.writerow([now, symbol, side, qty, price, reason, order_id])
```

A **rotating** log file starts a new file once it reaches about 1 MB and keeps the last five, so logs can never fill your disk. The trade log is a **CSV** (comma-separated values) file you can open in any spreadsheet. Fill prices come from the broker's order history; the trade log records what the bot *intended* and why.

15.5 Step 9: Alerts

BOT/NOTIFY.PY

```
"""Send a message to Telegram and/or Slack. Never crash the bot."""
import logging
import os

import requests

log = logging.getLogger(__name__)

def notify(text: str) -> None:
    token = os.getenv("TELEGRAM_BOT_TOKEN")
    chat_id = os.getenv("TELEGRAM_CHAT_ID")
    slack_url = os.getenv("SLACK_WEBHOOK_URL")
    try:
        if token and chat_id:
            requests.post(
                f"https://api.telegram.org/bot{token}/sendMessage",
                data={"chat_id": chat_id, "text": text}, timeout=10)
        if slack_url:
            requests.post(slack_url, json={"text": text}, timeout=10)
    except requests.RequestException as e:
        log.warning("Alert failed: %s", e)
```

If neither is configured, `notify` silently does nothing; Chapter 17 shows how to get the tokens. A failed alert is logged but never stops trading logic.

15.6 Step 10: The main loop

`bot/main.py` runs once, top to bottom, then exits. The scheduler (Chapter 16) starts it every trading day. First the imports and safety gates:

BOT/MAIN.PY

```
"""One complete run of the bot."""
import argparse
import logging
from datetime import date, datetime, timezone

from alpaca.data.historical import StockHistoricalDataClient
from alpaca.trading.client import TradingClient

from bot import orders, risk
from bot.config import load_settings
from bot.data import get_daily_closes
from bot.logs import record_trade, setup_logging
from bot.notify import notify
from bot.signals import sane, trend_signal
from bot.state import load_state, save_state

log = logging.getLogger("bot")

def halt(state, reason):
    state["halted"], state["halt_reason"] = True, reason
    save_state(state)
    log.error("HALTED: %s", reason)
    notify(f"Bot HALTED: {reason}")
```

Next, the start of `run()`: connect, check the kill switch, the halt flag, the market clock, and the circuit breakers.

BOT/MAIN.PY (CONTINUED)

```

def run(dry_run=False):
    s = load_settings()
    rules = s.risk
    state = load_state()
    trading = TradingClient(s.api_key, s.secret_key, paper=s.paper)
    data = StockHistoricalDataClient(s.api_key, s.secret_key)

    if risk.kill_switch_on():
        log.warning("Kill switch ON: closing everything.")
        if not dry_run:
            trading.close_all_positions(cancel_orders=True)
            notify("KILL SWITCH: all orders cancelled, positions closed.")
            return
    if state["halted"]:
        log.warning("Halted earlier: %s", state["halt_reason"])
        return
    clock = trading.get_clock()
    if not clock.is_open:
        log.info("Market closed. Next open: %s", clock.next_open)
        return

    acct = trading.get_account()
    equity, last_eq = float(acct.equity), float(acct.last_equity)
    cash = float(acct.cash)
    state["peak_equity"] = max(state["peak_equity"], equity)
    if risk.daily_loss_breached(equity, last_eq,
                               rules["max_daily_loss_pct"]):
        notify(f"Daily loss limit hit (equity {equity:,.0f}). "
              "No trades today.")
        save_state(state)
        return
    if risk.drawdown_breached(equity, state["peak_equity"],
                              rules["max_drawdown_pct"]):
        return halt(state, f"drawdown limit, equity {equity:,.0f}")

```

Notice the order: the cheapest, most important checks come first. Note that the daily loss limit pauses only for today, while the drawdown breaker halts until you investigate.

Now the decision loop:

BOT/MAIN.PY (CONTINUED)

```

positions = {p.symbol: p for p in trading.get_all_positions()}
closes = get_daily_closes(data, s.symbols, s.slow_days)
tag, sent, summary = date.today().isoformat(), 0, []

for sym in s.symbols:
    if sent >= rules["max_orders_per_run"]:
        log.warning("Order cap reached; stopping for today.")
        break
    if sym not in closes or not sane(closes[sym], s.slow_days):
        log.warning("%s: data failed sanity check, skipped", sym)
        continue
    signal = trend_signal(closes[sym], s.fast_days, s.slow_days)
    price = float(closes[sym].dropna().iloc[-1])
    held = sym in positions
    if signal == "SELL" and held:
        qty = float(positions[sym].qty)
        log.info("%s: SELL %s (trend down)", sym, qty)
        if not dry_run:
            o = orders.sell_all(trading, sym)
            record_trade(sym, "sell", qty, price, "trend", o.id)
        sent += 1
        summary.append(f"SELL {sym}")

```

And the buy branch, which must pass every risk check:

BOT/MAIN.PY (CONTINUED)

```

elif signal == "BUY" and not held:
    qty = risk.position_size(
        equity, price, rules["risk_per_trade"],
        rules["stop_loss_pct"], rules["max_position_pct"])
    ok, why = risk.approve_buy(sym, qty, price, cash,
                               len(positions), rules,
                               s.symbols)

    if not ok:
        log.info("%s: buy skipped (%s)", sym, why)
        continue
    log.info("%s: BUY %d @ ~%.2f", sym, qty, price)
    if not dry_run:
        o = orders.buy_with_stop(trading, sym, qty, price,
                                 rules["stop_loss_pct"], tag)
        record_trade(sym, "buy", qty, price, "trend", o.id)
    cash -= qty * price
    positions[sym] = None      # counts toward the limit
    sent += 1
    summary.append(f"BUY {qty} {sym}")

```

Finally, save state, send the summary, and the entry point that catches crashes:

BOT/MAIN.PY (CONTINUED)

```

state["last_run"] = datetime.now(timezone.utc).isoformat()
save_state(state)
mode = "DRY RUN" if dry_run else ("PAPER" if s.paper else "LIVE")
notify(f"[{mode}] equity {equity:,.0f}. "
      + (" ".join(summary) or "No trades."))

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--dry-run", action="store_true",
                        help="decide and log, but send no orders")
    args = parser.parse_args()
    setup_logging()
    try:
        run(dry_run=args.dry_run)
    except Exception as e:
        logging.getLogger("bot").exception("Run crashed")
        notify(f"Bot CRASHED: {type(e).__name__}: {e}")
        raise

```

Run it from the project folder:

TERMINAL

```
python -m bot.main --dry-run
```

`--dry-run` makes every decision and logs it but sends no orders. Use it after every change. When the dry run looks right, run it for real on paper with `python -m bot.main`.

 NOTE

The bot exits early when the market is closed, so running it on a weekend just logs “Market closed.” To test the decision logic anyway, temporarily comment out the clock check, run with `--dry-run`, and put it back.

15.7 Step 11: Tests

Tests are small programs that check your code gives the right answers. `pytest` finds every file named `test_*.py` and runs every function starting with `test_`.

TESTS/TEST_SIGNALS.PY

```
import pandas as pd

from bot.signals import sane, trend_signal


def test_rising_prices_mean_buy():
    s = pd.Series(range(1, 301), dtype=float)
    assert trend_signal(s, 50, 200) == "BUY"


def test_falling_prices_mean_sell():
    s = pd.Series(range(300, 0, -1), dtype=float)
    assert trend_signal(s, 50, 200) == "SELL"


def test_too_little_history_holds():
    s = pd.Series(range(1, 100), dtype=float)
    assert trend_signal(s, 50, 200) == "HOLD"


def test_sanity_rejects_a_crazy_jump():
    s = pd.Series([100.0] * 250 + [300.0])
    assert not sane(s, min_rows=200)
```

TESTS/TEST_RISK.PY

```
from bot import risk

RULES = {"max_open_positions": 5, "max_order_dollars": 2500}
OK = ["SPY", "TLT"]

def test_worked_example_gives_two_shares():
    assert risk.position_size(10_000, 500, 0.01, 0.08, 0.20) == 2

def test_cap_limits_size():
    # 1% risk, tiny 1% stop would mean 100 shares; cap allows 4
    assert risk.position_size(10_000, 500, 0.01, 0.01, 0.20) == 4

def test_unknown_symbol_is_refused():
    ok, _ = risk.approve_buy("XYZ", 1, 100, 10_000, 0, RULES, OK)
    assert not ok

def test_order_cap_is_enforced():
    ok, why = risk.approve_buy("SPY", 10, 300, 10_000, 0, RULES, OK)
    assert not ok and "cap" in why

def test_daily_loss_trips_at_three_percent():
    assert risk.daily_loss_breached(9_700, 10_000, 0.03)
    assert not risk.daily_loss_breached(9_800, 10_000, 0.03)
```

Run them all:

TERMINAL

```
python -m pytest -q
```

You should see every test pass. Run the tests before every change you deploy. If you change a rule, change its test first.

PROMPT

Write more pytest tests for bot/risk.py and bot/state.py: zero or negative prices, missing state file, a corrupted state file, and the drawdown breaker. Use tmp_path for files. Don't touch other modules.

- ☐ `bot/risk.py`, `state.py`, `orders.py`, `logs.py`, `notify.py` and `main.py` exist
- ☐ `python -m pytest -q` passes
- ☐ `python -m bot.main --dry-run` logs a decision for every symbol
- ☐ One real paper run placed orders I could see in the Alpaca dashboard
- ☐ Each paper buy has a matching stop order
- ☐ The kill switch closed everything on paper, and I deleted the file afterwards

Running It 24/7

Your laptop sleeps, travels, and updates itself at awkward moments. An autonomous bot needs a computer that's always on. This chapter covers four ways to schedule it, from a cloud server to GitHub Actions, and how to handle market hours and holidays.

IN THIS CHAPTER

- Market hours, holidays and early closes
- A small cloud server (VM)
- Scheduling with cron and with systemd timers
- GitHub Actions schedules
- n8n as a scheduler

16.1 Market hours and holidays

The NYSE core trading session runs **9:30 a.m. to 4:00 p.m. Eastern Time**, Monday to Friday, except market holidays. Some days close early at **1:00 p.m.**; in late 2026 these include **Friday, November 27** (the day after Thanksgiving) and **Thursday, December 24**.

You don't need to hard-code any of this. Your bot already asks Alpaca's **clock** whether the market is open (`trading.get_clock()`), and exits if not. Alpaca's **calendar** lists every trading day with its open and close times:

PYTHON

```

from datetime import date
from alpaca.trading.requests import GetCalendarRequest

days = trading.get_calendar(GetCalendarRequest(
    start=date(2026, 11, 20), end=date(2026, 12, 31)))
for d in days:
    print(d.date, d.open.time(), d.close.time())

```

So the scheduler's job is simple: start the bot every weekday at a time the market is normally open, such as **10:00 a.m. Eastern**, and let the clock check handle holidays.

 TIP

Why 10:00 and not 9:30? The first minutes after the open are the most volatile, with the widest spreads. Waiting half an hour usually gets calmer fills. On early-close days, 10:00 is still well inside the session.

Crypto never closes, so a crypto bot can run on any schedule, but remember that weekend liquidity is thinner.

16.2 Option 1: A small cloud server

A **VM** (virtual machine) is a rented computer in a data centre. A bot like this one needs very little: the smallest Linux VM from providers such as DigitalOcean, AWS Lightsail, Google Cloud, Hetzner, or Oracle Cloud is enough, and several providers offer free tiers or credits. Check current pricing on each provider's site.

SET UP AN UBUNTU SERVER

1

Create the VM.

Choose Ubuntu LTS and the smallest size. Add your SSH key when asked (SSH is a secure way to log in to a server from your terminal).

2

Log in.

`ssh root@YOUR_SERVER_IP`, then create a normal user: `adduser bot` and `usermod -aG sudo bot`. Log in again as `bot`.

3

Update and install Python.

TERMINAL

```

sudo apt update && sudo apt upgrade -y
sudo apt install -y python3-venv git

```

4 Set the server's time zone to New York.

This makes schedules easy to read.

TERMINAL

```
sudo timedatectl set-timezone America/New_York
```

5 Copy your project.

Clone it from a *private* GitHub repo, or copy it with `scp`. Then create the `.venv` and install `requirements.txt` as in Chapter 6.

6 Create `.env` on the server by hand.

Never copy secrets through Git. Run `chmod 600 .env`.

7 Test.

```
python -m bot.main --dry-run.
```

⚠ WARNING

Turn on the provider's firewall and allow only SSH. Your bot makes outgoing connections only; nothing needs to reach it from the internet.

16.3 Option 2: Schedule with cron

cron is the classic Linux scheduler. Each line of a **crontab** says *when* and *what*. Open yours with `crontab -e` and add:

TEXT

```
# m h dom mon dow command
0 10 * * 1-5 cd /home/bot/trading-bot && .venv/bin/python -m bot.main >> logs/cron.log 2>&1
```

Read it as: minute 0, hour 10, any day of month, any month, Monday to Friday (1–5). Because you set the server's time zone to New York, 10 means 10:00 a.m. Eastern, and daylight saving is handled for you. The `>> logs/cron.log 2>&1` part saves all output, including errors.

16.4 Option 3: Schedule with systemd

systemd is the service manager built into Ubuntu. Its **timers** add a feature cron lacks:

`Persistent=true` runs a missed job when the server comes back after downtime. Create two files:

/ETC/SYSTEMD/SYSTEM/TRADINGBOT.SERVICE

```
[Unit]
Description=Trading bot, one run

[Service]
Type=oneshot
User=bot
WorkingDirectory=/home/bot/trading-bot
ExecStart=/home/bot/trading-bot/.venv/bin/python -m bot.main
```

/ETC/SYSTEMD/SYSTEM/TRADINGBOT.TIMER

```
[Unit]
Description=Run the trading bot on weekday mornings

[Timer]
OnCalendar=Mon..Fri 10:00
Persistent=true

[Install]
WantedBy=timers.target
```

Then enable it and check:

TERMINAL

```
sudo systemctl daemon-reload
sudo systemctl enable --now tradingbot.timer
systemctl list-timers tradingbot.timer
journalctl -u tradingbot.service -n 50
```

`OnCalendar` uses the server's local time (New York, from step 4). `journalctl` shows the output of recent runs.

16.5 Option 4: GitHub Actions

GitHub Actions can run your bot on GitHub's computers on a schedule, with no server to maintain. GitHub's docs note some important limits:

- The shortest interval is once every **5 minutes**.
- Scheduled runs can be **delayed** during busy periods, especially at the start of every hour, and some queued jobs may be dropped.
- Schedules run in **UTC** by default, but you can add a `timezone`.
- In a **public** repository, scheduled workflows are disabled after **60 days** with no activity.

Use a **private** repository, and store keys in `Settings` > `Secrets and variables` > `Actions`. Then add:

```
.GITHUB/WORKFLOWS/BOT.YML

name: Trading bot
on:
  schedule:
    - cron: "7 10 * * 1-5"
      timezone: "America/New_York"
  workflow_dispatch:

jobs:
  run:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v7
      - uses: actions/setup-python@v7
        with:
          python-version: "3.13"
      - run: pip install -r requirements.txt
      - run: python -m bot.main
    env:
      ALPACA_API_KEY: ${ secrets.ALPACA_API_KEY }
      ALPACA_SECRET_KEY: ${ secrets.ALPACA_SECRET_KEY }
      TELEGRAM_BOT_TOKEN: ${ secrets.TELEGRAM_BOT_TOKEN }
      TELEGRAM_CHAT_ID: ${ secrets.TELEGRAM_CHAT_ID }
```

The minute is set to 7, not 0, to avoid the busy top of the hour. `workflow_dispatch` adds a **Run workflow** button for manual runs.

☆ IMPORTANT

Each Actions run starts on a fresh computer, so `state.json` and `logs/` vanish afterwards. For a daily bot that's survivable (the broker remembers positions and orders), but your peak-equity record and halt flag are lost. Either keep using a server, or have the workflow save those files somewhere persistent, such as committing them back to the private repo. Ask Claude Code to add that step if you go this route.

16.6 Option 5: n8n

n8n is a visual automation tool (see the *n8n* manual in this series). Its **Schedule Trigger** node can start a workflow on weekdays at 10:00. On a self-hosted n8n, you can run the bot as a command on the same machine; on n8n Cloud, use an HTTP Request node to call a small web endpoint you host instead. n8n is most useful *around* the bot: forwarding alerts, building weekly summaries, or logging trades to a spreadsheet.

16.7 Which should you pick?

OPTION	COST	RELIABILITY	EFFORT
Cloud VM + systemd	Small monthly fee	High	Medium
Cloud VM + cron	Small monthly fee	High	Low
GitHub Actions	Free tier minutes	Medium (delays possible)	Low
n8n	Depends on hosting	Medium	Medium
Your laptop	Free	Low	Low

- ☐ My bot runs automatically every trading day
- ☐ I know where its output goes (log file or journal)
- ☐ Holidays and early closes are handled by the clock check
- ☐ Secrets are on the server or in Actions secrets, never in Git
- ☐ I tested a missed run and a manual run

Monitoring, Alerts and a Dashboard

A bot you can't see is a bot you can't trust. Monitoring means three things: **alerts** when something happens, a **daily report** so silence is never ambiguous, and a **dashboard** for the big picture.

IN THIS CHAPTER

- Setting up Telegram, Slack and email alerts
- What to alert on
- A daily report
- A heartbeat that notices when the bot *didn't* run
- A small dashboard with Streamlit

17.1 Telegram alerts FREE

CREATE A TELEGRAM BOT FOR ALERTS

1 Talk to BotFather.

In Telegram, search for **@BotFather** (the official bot for creating bots), send `/newbot`, and follow the prompts. It gives you a **token**.

2 Message your new bot.

Open your bot's chat and send it any message, such as "hi".

3 Find your chat ID.

In a browser, open `https://api.telegram.org/bot<TOKEN>/getUpdates` (replace `<TOKEN>`). Look for `"chat":{"id":` and copy the number.

4

Add both to `.env`.Set `TELEGRAM_BOT_TOKEN` and `TELEGRAM_CHAT_ID`.

5

Test.

```
python -c "from dotenv import load_dotenv; load_dotenv(); from bot.notify import
notify; notify('hello from the bot')"
```

The token is a secret: anyone with it can send messages as your bot, so keep it in `.env` only.

17.2 Slack alerts

In Slack, create an app with **Incoming Webhooks** enabled and add a webhook to a channel. Slack gives you a URL starting with `https://hooks.slack.com/services/`. Put it in `.env` as `SLACK_WEBHOOK_URL`. The bot posts JSON like `{"text": "..."} to it. Treat the URL as a secret.`

17.3 Email alerts

Python's built-in `smtplib` can send email through your provider's SMTP server. Most providers (Gmail included) require an **app password** rather than your normal password. Email is fine for the daily report but too slow and easy to miss for urgent alerts.

BOT/EMAILER.PY

```
import os
import smtplib
from email.message import EmailMessage

def send_email(subject: str, body: str) -> None:
    msg = EmailMessage()
    msg["Subject"], msg["To"] = subject, os.environ["ALERT_EMAIL_TO"]
    msg["From"] = os.environ["SMTP_USER"]
    msg.set_content(body)
    with smtplib.SMTP(os.environ["SMTP_HOST"], 587) as s:
        s.starttls()
        s.login(os.environ["SMTP_USER"], os.environ["SMTP_PASSWORD"])
        s.send_message(msg)
```

17.4 What to alert on

EVENT	URGENCY	ALREADY IN THE BOT?
Any order sent	Info	Yes (run summary)
Crash / exception	High	Yes
Daily loss limit hit	High	Yes
Drawdown halt	High	Yes
Kill switch used	High	Yes
Bot didn't run	High	Add a heartbeat (below)
Data failed sanity check	Medium	Logged; add alert if you like

17.5 The daily report

Add a short report after the market closes. It lists equity, today's change, and every position:

BOT/REPORT.PY

```
from bot.config import load_settings
from bot.notify import notify
from alpaca.trading.client import TradingClient

def daily_report():
    s = load_settings()
    t = TradingClient(s.api_key, s.secret_key, paper=s.paper)
    a = t.get_account()
    eq, last = float(a.equity), float(a.last_equity)
    change = (eq - last) / last * 100 if last else 0.0
    lines = [f"Equity ${eq:,.2f} ({change:+.2f}% today)",
            f"Cash ${float(a.cash):,.2f}"]
    for p in t.get_all_positions():
        lines.append(f"{p.symbol}: {p.qty} sh, "
                    f"P/L ${float(p.unrealized_pl):,.2f}")
    notify("Daily report\n" + "\n".join(lines))

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

Schedule it for about 4:30 p.m. Eastern on weekdays with another cron line or systemd timer:

```
python -m bot.report .
```

17.6 A heartbeat for silent failures

The scariest failure is the one that sends no alert: the server is off, or the scheduler broke. A **heartbeat** checks that the bot *did* run. The bot writes `last_run` to `state.json`; a separate tiny script, run by a second schedule (or a free external uptime monitor that watches a URL), alerts you if `last_run` is more than a day old on a trading day.

BOT/HEARTBEAT.PY

```
from datetime import datetime, timedelta, timezone

from bot.notify import notify
from bot.state import load_state

last = load_state()["last_run"]
age = (datetime.now(timezone.utc) - datetime.fromisoformat(last)
       if last else timedelta.max)
if age > timedelta(hours=26):
    notify(f"Heartbeat: bot hasn't completed a run since {last!r}")
```

NOTE

`last_run` is only saved on days the market was open and the bot reached the end. Run the heartbeat on weekday afternoons, and expect one extra alert after a holiday. A daily report that *doesn't* arrive is your other heartbeat: if you don't hear from the bot, assume something's wrong.

17.7 A simple dashboard

Streamlit turns a Python script into a web page (`pip install streamlit`). This one shows your equity history from Alpaca and your trade log:

DASHBOARD.PY

```
import pandas as pd
import streamlit as st
from alpaca.trading.client import TradingClient
from alpaca.trading.requests import GetPortfolioHistoryRequest

from bot.config import load_settings

s = load_settings()
t = TradingClient(s.api_key, s.secret_key, paper=s.paper)
st.title("Trading bot dashboard")

a = t.get_account()
st.metric("Equity", f"${float(a.equity):.2f}")

h = t.get_portfolio_history(
    GetPortfolioHistoryRequest(period="3M", timeframe="1D"))
curve = pd.Series(h.equity, index=pd.to_datetime(h.timestamp, unit="s"))
st.line_chart(curve)

st.subheader("Trade log")
st.dataframe(pd.read_csv("logs/trades.csv"))
```

Run it with `streamlit run dashboard.py` and open the address it prints. On a server, don't expose it to the internet; view it through an SSH tunnel (`ssh -L 8501:localhost:8501 bot@SERVER`), then open `http://localhost:8501` on your own computer.

- ☐ Telegram or Slack alerts arrive on my phone
- ☐ I get a daily report after the close
- ☐ A heartbeat or a missing report would tell me the bot stopped
- ☐ I can see equity history and the trade log in a dashboard

Going Live with Tiny Money

If your bot passed every paper-trading criterion, you've earned the right to try it with real money: a *small* amount. The goal of going live isn't profit yet. It's discovering what paper trading couldn't simulate.

IN THIS CHAPTER

- The go-live checklist
- Choosing a tiny starting amount
- Switching from paper to live safely
- The first month: what to watch
- When to scale up, and when to stop

18.1 Are you ready?

- ☐ Every paper criterion from Chapter 12 is ticked
- ☐ The kill switch, daily loss limit and drawdown halt were all tested on paper
- ☐ I've reread Chapter 2 and still choose to continue
- ☐ The money I'll use could vanish without hurting me or anyone I support
- ☐ I have an emergency fund and no high-interest debt
- ☐ I know how to close all positions by hand on the broker's website

18.2 How much?

Start with an amount small enough to be boring: many people choose a few hundred dollars. With tiny accounts, two things change:

- **Whole shares get chunky.** A \$500 account can't buy a \$500 fund with 1% risk. Pick cheaper ETFs, raise `max_position_pct`, or switch the bot to fractional dollar orders (without OTO stops, which need whole shares at Alpaca; your bot's daily check then acts as the exit).
- **Fees matter more on crypto.** A 0.25% fee per trade is 0.5% for a round trip.

⚠ DANGER

Never deposit more because the bot is losing ("it's due to come back"), and never raise limits to "win it back." That's how small, survivable losses turn into big ones.

18.3 Switching from paper to live

GO LIVE SAFELY

- 1 **Open and fund a live account.**
Use a **cash** account. Turn on two-factor authentication.
- 2 **Create live keys.**
Use the most restrictive permissions offered and an IP allowlist if available.
- 3 **Keep live keys separate.**
Put them in the server's `.env` only, replacing the paper keys, and add `ALLOW_LIVE=yes`.
- 4 **Flip the config.**
Set `paper = false` in `config.toml`. The two-key lock now lets the bot run live.
- 5 **Lower the limits.**
Cut `max_order_dollars` to a small number for the first month.
- 6 **Dry-run on live.**
`python -m bot.main --dry-run`. Check the account number and equity in the log are your live account's.
- 7 **Watch the first real run.**
Be at your screen. Confirm each order and stop in the broker's dashboard.

18.4 The first month

Compare every live fill with the price the bot expected (the `ref_price` in your trade log). The difference is your real **slippage**. Also compare live results with your paper account running the same rules in parallel. Keep the paper bot running: it's your control group.

WATCH FOR	WHAT IT MEANS
Fills consistently worse than expected	Raise cost assumptions in your backtest
Orders rejected	Account permissions, buying power, or settlement
Live and paper diverge	Timing, data, or a bug that only shows with real fills
Your stress level	If you check your phone hourly, the amount is too big

18.5 Scaling up, slowly

If, after three months live, the bot ran cleanly and live results tracked paper, you *may* add money in small steps: perhaps double, then wait another three months. Never add money because of a hot streak. And there's no obligation to scale at all.

18.6 When to stop

Write your stop rules now, while you're calm:

STOP-RULES.MD

```
I will turn the bot off and review if ANY of these happen:
- The drawdown halt fires
- An unintended order is ever placed
- Live results trail paper by more than 3% over 3 months
- I can't explain a trade from the log
- I feel tempted to override it or raise limits
```

Turning a bot off isn't failure. It's the risk system working, with you as its final layer.

- ☐ I'm live with an amount I consider boring
- ☐ Live keys are trade-only and stored only on the server
- ☐ I watched the first live run and confirmed orders and stops
- ☐ A paper bot runs in parallel as a control
- ☐ I've written my stop rules

Maintaining and Improving It

A bot isn't finished when it first runs. Libraries update, brokers change their APIs, markets change character, and your own ideas evolve. A little routine maintenance keeps it trustworthy.

IN THIS CHAPTER

- Daily, weekly, monthly and yearly routines
- Updating libraries safely
- Changing the strategy without fooling yourself
- Using Claude Code for reviews and upgrades

19.1 Routines

WHEN	TASK
Daily	Read the daily report; act on any alert
Weekly	Skim the log; reconcile trade log with broker history
Monthly	Check stops still exist (GTC orders expire after 90 days at Alpaca); review costs and slippage
Quarterly	Rotate API keys; update libraries; rerun tests and backtests
Yearly	Download tax forms; archive logs; reread your stop rules

19.2 Updating libraries safely

`alpaca-py` and other libraries change over time, and an update can rename something your bot uses. Update on your own computer first, never directly on the server:

UPGRADE SAFELY

1 Create a branch.

```
git checkout -b upgrade-libs.
```

2 Upgrade.

```
pip install --upgrade alpaca-py pandas, then pip freeze > requirements.txt.
```

3 Read the changelog.

Look for “breaking changes” in the library’s release notes.

4 Test.

```
python -m pytest -q and python -m bot.main --dry-run.
```

5 Paper first.

Run the upgraded bot on paper for a week.

6 Deploy.

Merge, pull on the server, reinstall from `requirements.txt`.

19.3 Changing the strategy

Every change is a new hypothesis. Treat it the same way as the first one: backtest with costs, walk-forward test, then paper trade before it goes live. Keep a **change log**: date, what changed, why, and the evidence.

⚠ WARNING

The most dangerous moment is right after a losing streak, when you’re tempted to tweak parameters until the recent past looks good. That’s overfitting in real time. Decide changes on weekends, with data, never during market hours.

19.4 Ideas for improvement

- **More symbols, same rules.** Diversification is the closest thing to a free lunch.
- **A volatility filter.** Size positions smaller when markets are jumpy.
- **A second, uncorrelated bot.** For example a rebalancing bot alongside the trend bot, each with its own budget.

- **Better fills.** Use limit orders near the current price instead of market orders.
- **The AI news filter** from Chapter 10, tested on paper first.

19.5 Claude Code as your maintenance partner

PROMPT

Review this whole repository as a cautious senior engineer. Find bugs, unhandled errors, and any path where an order could be sent without passing bot/risk.py. Rank issues by danger. Don't change any files yet.

PROMPT

alpaca-py has a new version. Read its changelog and the current docs, check every alpaca import and call in this repo, and list anything that changed. Then run the tests.

PROMPT

Add a new risk rule: skip new buys when the symbol's 20-day volatility is above 3% per day. Put it in bot/risk.py, add it to config.toml, write pytest tests first, then the code. Show me the diff.

- ☐ I have a written routine and I follow it
- ☐ Library upgrades go through tests and paper first
- ☐ Strategy changes go through backtest, walk-forward, then paper
- ☐ I keep a change log

Scams to Avoid

Wherever people hope to make easy money, scammers follow, and “AI trading” is one of their favourite costumes. The U.S. Federal Trade Commission (FTC) reports that consumers lost more than **\$7.9 billion** to investment scams in 2025, with a median loss of more than **\$10,000** per person. This chapter shows the common patterns so you can walk away early.

IN THIS CHAPTER

- What regulators are warning about
- The five most common AI-trading scams
- Red flags and how to check a firm
- What to do if you’ve been targeted

20.1 What regulators say

- **CFTC:** Its customer advisory “*AI Won’t Turn Trading Bots into Money Machines*” warns that fraudsters tout automated trading algorithms, signal strategies and crypto schemes promising unreasonably high or guaranteed returns. Its case study, Mirror Trading International, promised a bot “guaranteed” at least 10% a month; it was a Ponzi scheme that took more than \$1.7 billion in bitcoin from at least 23,000 people and showed customers fake balances from demo accounts.
- **SEC, NASAA and FINRA:** A joint investor alert, “*Artificial Intelligence (AI) and Investment Fraud*,” warns about unregistered platforms claiming to use AI, pitches like “Our proprietary AI trading system can’t lose!”, and deepfake videos impersonating real people.
- **FTC:** Warns about “coaches” who offer to teach you to make a fortune in stocks, forex or crypto, and reports that more than half of the money lost to social-media scams in 2025 went to investment scams that started there.

20.2 The five common scams

SCAM	HOW IT WORKS
"Guaranteed" AI bot	You deposit money with a platform whose bot supposedly earns 1–3% a day. Early withdrawals work, building trust; later ones are blocked by "fees" or "taxes"
Signal groups	A Telegram, WhatsApp or Discord group sells "VIP signals." Often the organisers buy first and dump on members (a pump-and-dump)
Fake platforms	A slick app or website shows growing profits, but it's just numbers on a screen. There's no real trading
Romance or friendship scams	A new online friend shows you their trading success and guides you to a "platform." Sometimes called pig butchering
"Bot software" for sale	A one-time fee for a bot with impossible backtests, usually overfitted, sometimes malware that steals keys

20.3 Red flags

Walk away if you see any of these

- Guaranteed, "risk-free," or fixed daily/monthly returns
- Pressure to act now or recruit friends
- A request for your API key with withdrawal permission
- Deposits only in crypto, gift cards, or wire transfers
- Withdrawals blocked until you pay a "fee" or "tax"
- A platform you can't find on a regulator's register
- Screenshots of profits instead of audited, verifiable records
- Someone you've never met in person guiding your investing

20.4 How to check a firm

CHECK BEFORE YOU TRUST

1

Brokers:

Search FINRA's **BrokerCheck** (brokercheck.finra.org).

2

Investment advisers and professionals:

Use the SEC's search at investor.gov.

- 3 **Futures and forex firms:**
Check the National Futures Association's BASIC database, and the CFTC's RED List of unregistered foreign entities.
- 4 **Crypto platforms:**
Confirm state money-transmitter licences and FinCEN registration.
- 5 **Search the name plus "scam" or "complaint".**
Absence of results proves nothing, but presence is a strong warning.

☆ **IMPORTANT**

The best protection is the setup in this manual: your money stays at a regulated broker in *your* name, your bot's keys can't withdraw, and the code is yours and readable. Nobody else needs your login, your keys, or your deposit.

20.5 If you've been targeted

- **Stop sending money.** Including "fees" to unlock withdrawals; that's part of the scam.
- **Revoke any API keys** you shared and change passwords.
- **Report it:** to the FTC at [ReportFraud.ftc.gov](https://www.reportfraud.ftc.gov), the FBI's Internet Crime Complaint Center at [ic3.gov](https://www.ic3.gov), the SEC at [sec.gov/tcr](https://www.sec.gov/tcr), or the CFTC at [cftc.gov/complaint](https://www.cftc.gov/complaint).
- **Beware recovery scams.** People who promise to recover lost crypto for an upfront fee are very often the same fraudsters, back for a second round.

- ☐ I can name the five common scam patterns
- ☐ I know how to check a firm on BrokerCheck and investor.gov
- ☐ I'll never give anyone a key with withdrawal permission
- ☐ I know where to report fraud

Checklists and Troubleshooting

Everything in one place: master checklists to print, a big troubleshooting table, and a quick reference.

IN THIS CHAPTER

- Master checklists from setup to live
- Troubleshooting common errors
- Quick command reference

21.1 Master checklists

Before writing code

- ☐ I've read the honest odds and accept that profit isn't guaranteed
- ☐ I've chosen a loss-tolerable amount and a cash account
- ☐ I know the PDT rule was replaced but my broker may still apply it
- ☐ I've picked ETFs (or BTC/ETH spot) and a daily timeframe
- ☐ I have a paper account with two-factor authentication

Before paper trading

- ☐ Backtest acts the next day and includes costs
- ☐ Walk-forward results are acceptable
- ☐ All tests pass, and the dry run makes sense
- ☐ Kill switch, daily loss and drawdown limits are configured
- ☐ Alerts and the daily report arrive
- ☐ Pass criteria are written down

Before going live

- ☐ Eight-plus weeks of clean paper runs; trade log matches the broker
- ☐ Live keys are trade-only, IP-limited where possible, stored only on the server
- ☐ `paper = false` and `ALLOW_LIVE=yes` set deliberately; limits lowered
- ☐ I know how to close everything by hand
- ☐ Stop rules are written down

21.2 Troubleshooting

SYMPTOM	LIKELY CAUSE	FIX
<code>KeyError: 'ALPACA_API_KEY'</code>	<code>.env</code> missing or not loaded	Check the file is in the project root and names match
HTTP 401 / 403 "unauthorized" or "forbidden"	Wrong keys, or paper keys used live	Regenerate keys; match <code>paper=</code> to the key type
<code>ModuleNotFoundError: alpaca</code>	Virtual environment not active	<code>source .venv/bin/activate</code> , then reinstall
<code>ModuleNotFoundError: bot</code>	Run from wrong folder	<code>cd</code> to the project root; use <code>python -m bot.main</code>
<code>tomllib</code> not found	Python older than 3.11	Install Python 3.12+
Error mentioning your subscription and SIP data	Asking for the last 15 minutes on the free plan	Request completed days only, or use the IEX feed
HTTP 429	Too many requests	Fetch many symbols in one request; slow down
Empty DataFrame	Wrong symbol, weekend start date, or too short a range	Print the request; widen <code>start</code>
Every signal is HOLD	Not enough history	Fetch more days than <code>slow_days</code>
"insufficient qty available" on sell	Shares held by an open stop order	Cancel open orders first (the bot does)
Order rejected: buying power	Unsettled cash or too large	Lower size; respect settlement
Bracket/OTO rejected	Fractional qty or wrong time in force	Whole shares; <code>day</code> or <code>gtc</code>
Stop orders disappeared	GTC orders auto-cancel after 90 days	Monthly stop check; re-place stops

Bot never runs	Scheduler, time zone, or path wrong	<code>systemctl list-timers</code> or check cron log
Runs at the wrong hour	Server in UTC	<code>timedatectl set-timezone America/New_York</code>
Actions schedule late or skipped	GitHub load at the top of the hour	Use an odd minute; accept delays or use a VM
Scheduled workflow stopped	Public repo inactive 60 days	Use a private repo; re-enable the workflow
No Telegram messages	Never messaged the bot, or wrong chat ID	Message it first; recheck <code>getUpdates</code>
Bot says halted	Drawdown breaker fired	Investigate, then set <code>halted</code> to false in <code>state.json</code>
Kill switch does nothing	File in the wrong folder	Create <code>KILL_SWITCH</code> in the project root
Live results far below paper	Slippage, timing, fees	Measure slippage; raise backtest costs
Two buys of the same fund	Logic bug or a run repeated	Check <code>client_order_id</code> tags and the position check
MCP tools don't appear	<code>uv</code> missing or client not restarted	Install <code>uv</code> ; restart; check keys in config

21.3 Quick reference

Everyday commands

- Activate the environment: `source .venv/bin/activate`
- Run tests: `python -m pytest -q`
- Dry run: `python -m bot.main --dry-run`
- Real (paper) run: `python -m bot.main`
- Daily report: `python -m bot.report`
- Dashboard: `streamlit run dashboard.py`
- Kill switch on: `touch KILL_SWITCH`; off: `rm KILL_SWITCH`
- Timer status: `systemctl list-timers tradingbot.timer`
- Recent output: `journalctl -u tradingbot.service -n 50`
- Follow the log: `tail -f logs/bot.log`

TIP

When you're stuck, paste the full error and the code around it into Claude Code and ask: "Explain this error for a beginner, find the cause, and propose the smallest fix." Then read the fix before accepting it.

Glossary

Adjusted prices	Historical prices rewritten to remove the effect of stock splits and dividends.
AI agent	An AI that can take actions, such as placing orders, not just answer questions.
API	Application programming interface: a doorway that lets one program talk to another, such as your bot talking to your broker.
API key	A pair of secret codes (key ID and secret) that lets a program act on your account.
Atomic write	Saving a file by writing a temporary copy and swapping it in, so the file is never half-written.
Backtest	Replaying a strategy on historical prices to see how it would have performed.
Bar	A summary of trading over a period (open, high, low, close, volume). Also called a candle.
Benchmark	What you compare a strategy with, usually buying and holding the same fund.
Bracket order	An entry order plus a take-profit and a stop-loss that activate when it fills.
Broker	A company that holds your investment account and sends your orders to the market.
CAGR	Compound annual growth rate: the average yearly return, with compounding.
Cash account	A brokerage account where you can only buy with deposited money; no borrowing.
Circuit breaker	A rule that halts the bot when results suggest something is wrong, such as a large drawdown.

Client order ID	Your own label attached to an order, so you can trace it back to a specific run.
cron	The classic Linux scheduler that runs commands at set times.
Crypto exchange	A company that runs a marketplace for cryptocurrencies, usually holding your coins.
CSV	Comma-separated values: a plain-text table that opens in any spreadsheet.
Daily loss limit	A rule that stops trading for the day after the account falls a set percentage.
Dataclass	A Python container for a group of related settings or values.
DataFrame	A pandas table of rows and columns, used for prices and calculations.
Day trading	Opening and closing positions within the same trading day.
Digital assets question	The yes/no question about crypto on U.S. federal tax returns.
Digital commodity	A crypto asset, such as bitcoin, that the SEC and CFTC treat as a commodity rather than a security.
Dividend	A cash payment a company or fund makes to its shareholders.
Dollar-cost averaging (DCA)	Buying a fixed dollar amount at regular intervals regardless of price.
Drawdown	The fall from an account's or strategy's highest value to a later low.
Dry run	Running the bot so it makes and logs decisions without sending any orders.
Equity curve	A chart or series showing how an account's value changed over time.
ETF	Exchange-traded fund: a basket of investments that trades like a single stock.
Form 1099-DA	The IRS form crypto brokers use to report your digital asset sales.
Fractional share	Part of one share, bought by specifying a dollar amount.
Free-riding	Buying and selling a security in a cash account before paying for it; can lead to a 90-day restriction.
Good faith violation	Selling a security bought with unsettled funds before those funds settle, in a cash account.
Grid bot	A bot that places buy and sell orders at evenly spaced prices to profit from price swings in a range.

GTC	Good-till-cancelled: an order that stays open until filled or cancelled (Alpaca cancels them after 90 days).
Guardrail	A rule that keeps a component's mistakes harmless, such as defaulting an unexpected AI answer to "neutral."
Heartbeat	A check that the bot actually ran, so silent failures are noticed.
IEX	Investors Exchange, a single U.S. stock exchange whose real-time data Alpaca provides free.
Intraday margin standards	FINRA's 2026 replacement for the pattern day trader rules, based on exposure during the day.
IP allowlist	A setting that only accepts an API key from listed internet addresses. Also called whitelisting.
Kill switch	A way to stop the bot and close all positions immediately.
Large language model (LLM)	An AI model that reads and writes text, such as Claude.
Least privilege	Giving each key or tool only the permissions it truly needs.
Leverage	Controlling a larger position than your deposit, magnifying gains and losses.
Liquidity	How easily something can be bought or sold without moving its price.
Look-ahead bias	A backtest error that uses information you wouldn't have had at the time.
Machine learning (ML)	Letting a program find patterns in data instead of writing rules by hand.
Maker / taker	A maker order waits on the order book and adds liquidity; a taker order fills immediately. Fees often differ.
Margin account	A brokerage account that lets you borrow to buy or sell short.
Market data	Prices and volumes coming from exchanges.
Market order	An order to buy or sell immediately at the best available price.
MCP	Model Context Protocol: a standard way to connect AI apps to outside tools such as a broker.
Mean reversion	A strategy betting that unusually stretched prices return toward their average.
Momentum	A strategy that buys recent relative winners, expecting them to keep outperforming for a while.
Moving average	The average price over the last N days, recalculated daily.

Notional order	An order specified in dollars rather than shares.
OHLCV	Open, high, low, close, volume: the five numbers in a price bar.
Order	An instruction to your broker to buy or sell.
OTO order	One-triggers-other: an entry order that activates a second order (such as a stop-loss) when it fills.
Overfitting	Tuning rules so closely to past data that they capture noise and fail on new data.
Paper trading	Simulated trading with fake money against real prices.
Pattern day trader (PDT)	A former FINRA designation for frequent day traders in margin accounts, with a \$25,000 minimum; replaced in 2026.
Position	The amount of a particular investment you currently hold.
Position sizing	Deciding how much to buy, for example with the 1–2% risk rule.
Prompt injection	Hidden text that tries to trick an AI into doing something it shouldn't.
Pure function	A function that only uses its inputs and returns an answer, with no network or file access.
Rebalancing	Trading back to a target mix of investments when it drifts.
Risk rules	Code that can veto a trade, such as size limits and loss limits.
Rotating log	A log file that starts a new file at a size limit and keeps only a few old ones.
Sandbox	A test version of an API; some simulate trading, others return canned answers.
Settlement	When a trade officially completes and cash or shares change hands; T+1 for most U.S. stocks.
Sharpe ratio	Return divided by volatility; a measure of return per unit of risk.
Short selling	Selling borrowed shares to profit if the price falls; requires margin.
Short-term capital gain	Profit on an asset held one year or less; taxed as ordinary income in the U.S.
SIP	Securities information processor: the combined data feed of all U.S. exchanges.
SIPC	Securities Investor Protection Corporation: protects customers if a member brokerage fails, up to \$500,000 including \$250,000 cash.
Slippage	The difference between the price you expected and the price you got.

Spot	Buying the actual asset (such as a coin) rather than a contract about it.
Spread	The gap between the best buying and selling prices.
Standard deviation	A measure of how spread out values are around their average.
State file	A small file where the bot remembers things between runs.
Stop-loss	An order that sells automatically if the price falls to a chosen level.
Strategy	Written rules precise enough for a computer to follow.
Survivorship bias	Testing only on assets that still exist, ignoring those that failed.
systemd timer	A Linux scheduler built into Ubuntu that can catch up on missed runs.
T+1	Settlement one business day after the trade date.
Terminal	A window for typing commands instead of clicking.
Timeframe	How often the bot looks at prices and can trade (daily, hourly, and so on).
TOML	A simple settings file format of names and values.
Trend following	A strategy that holds assets while their prices are trending up.
Virtual environment	A private set of Python libraries for one project.
VM	Virtual machine: a rented computer in a data centre.
Walk-forward testing	Choosing parameters on past data, testing them on the next unseen period, and rolling forward.
Wash sale	Selling at a loss and buying substantially identical securities within 30 days before or after; the loss is deferred.
Watchlist	The short list of symbols the bot is allowed to trade.
Z-score	How many standard deviations a value is from its average.