ABRIL · 2026

POR SANTIAGO FERNÁNDEZ

LINKEDIN @SANFERNANDEZF EMAIL

Stack Tecnológico para Trading

Servidor · Claude Code · Datos · Ejecución

[H]	{C}	<P>	|pg|	<V>
HETZNER	**CLAUDE**	**PYTHON**	**POSTGRESQL**	**VSCODE**
VPS	Agente	3.12+	TSDB	Remote

Menos stack. Más ejecución.

STACK TECNOLÓGICO PARA TRADING

Servidor · Claude · Datos · Ejecución

HETZNER VPS

CLAUDE Agente

PYTHON 3.12+

POSTGRESQL TSDB

VSCODE Remote

🚀 Menos stack. Más ejecución.

01 POR QUÉ EL TRADING ES TECNOLÓGICO — Concepto

El trading algorítmico es un problema tecnológico: transformar información en acciones de forma automática.

❌ Script (no es suficiente)
Depende de ti. Si cierras el portátil, muere.
frágil · manual

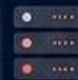
Sistema (lo que construimos)
Persiste, se reinicia si cae, está automatizado y conectado.
robusto · autónomo

02 TU VPS EN HETZNER — Hetzner

Un servidor es un ordenador siempre encendido y conectado a internet. Tu sistema vive aquí, no en tu portátil.

Tu ordenador local
- Lo enciendes y lo apagas
- Todo muere al cerrar la tapa
- Depende de que estés
- No accesible desde el exterior

frágil · dependiente



Servidor Hetzner VPS
- ✓ Activo 24/7
- ✓ Procesos persistentes (tmux)
- ✓ Accesible desde cualquier lugar vía SSH
- ✓ No depende de ti

siempre encendido · autónomo

¿POR QUÉ HETZNER?

Precio
Desde ~4€/mes. El más barato para la potencia que ofrece. AWS cuesta 8–10x más.

Rendimiento
CPUs AMD EPYC + SSD NVMe. Ideal para ML, backtesting y trading.

Sencillez
Interfaz clara. Un servidor en marcha en menos de 60s, sin líos.

Fiabilidad
Datacenter en Frankfurt y Helsinki. Baja latencia y GDPR compliance.

03 TERMINAL Y SSH — SSH

- SSH es la puerta segura a tu servidor.
- Accedes desde cualquier lugar.
- Clave SSH (no contraseña) = seguridad.

```
ssh usuario@tu-servidor-ip
```

04 TMUX: PERSISTENCIA — tmux

- Mantiene tus sesiones vivas.
- Si te desconectas, todo sigue ejecutándose.
- Ideal para procesos largos (bots, backtests).

```
tmux new -s bot → Ctrl+B D → tmux ls → tmux attach -t bot
```

05 VSCODE CONECTADO — VSCode

- Desarrollo cómodo desde tu editor favorito.
- Extensión Remote - SSH.
- Editas en local, ejecutas en el servidor.

```
Ctrl+Shift+P → Remote-SSH: Connect to Host
```

06 ORGANIZACIÓN Y GITHUB — Git

- Git para control de versiones.
- GitHub para copias de seguridad y colaboración.
- Historial, ramas y rollback.

```
git add . → git commit -m "update" → git push
```

07 CLAUDE CODE: TU COPILOTO — Claude

- Claude Code te ayuda a construir más rápido.
- Entiende tu código, sugiere, corrige y explica.
- Tu copiloto en la terminal.

```
claude → /init → /help
```

08 SCRIPTS, SYSTEMD Y TELEGRAM — Cron

- systemd para servicios que siempre arrancan.
- Cron para tareas programadas.
- Telegram para alertas en tiempo real.

```
systemctl status mi-servicio • crontab -e
```

09 DESCARGA Y ALMACENAMIENTO — PostgreSQL

- PostgreSQL + TimescaleDB = base de datos de series temporales (TSDB).
- Almacena OHLCV, indicadores y resultados.
- Rápido, fiable y escalable.

```
psql → CREATE TABLE ohlcv (...)
```

10 LOGS Y OBSERVABILIDAD — Logs

- Logs para saber qué pasa y cuándo.
- journalctl, tail, logs de aplicaciones.
- Métricas básicas para monitorear tu sistema.

```
journalctl -u mi-servicio -f • tail -f /var/log/app.log
```

11 AHORA ESTÁS PREPARADO — Siguiente

INFRAESTRUCTURA LISTA
Tu servidor está activo, seguro y preparado para producir.

EJECUCIÓN CONTINUA
Tus sistemas corren 24/7. Sin depender de ti.

ITERA Y MEJORA
Ahora puedes probar, medir y escalar sin límites.

🏆 RECUERDA
La ventaja no está en la estrategia. Está en la capacidad de construir sistemas que operan sin ti.
MENOS STACK. MÁS EJECUCIÓN.

ÍNDICE

Stack Tecnológico para Trading

Guía infraestructura real

La infraestructura que necesitas. Nada más.

> Este manual no enseña trading. Enseña a construir la **máquina**.
> La pieza central es **Claude Code, Codex o Kimi** — tu copiloto de desarrollo.

#	CAPÍTULO	TECNOLOGÍA
01	Por qué el trading es tecnológico	Concepto
02	Tu VPS en Hetzner	Hetzner
03	Terminal y SSH	SSH
04	tmux: persistencia	tmux
05	VSCode conectado	VSCode
06	Organización y GitHub	Git
07	**Claude Code: tu copiloto**	Claude
08	Scripts, Systemd y Telegram	Cron
09	Descarga y almacenamiento	PostgreSQL
10	Logs y observabilidad	Logs
11	Ahora estás preparado	Siguiente

CAP. 01

Por Qué el Trading Algorítmico Es un Problema Tecnológico

El cambio de mentalidad antes de escribir una línea de código

> Cuando la mayoría de las personas escuchan "trading algorítmico", lo primero que imaginan son gráficos, mercados financieros y decisiones de inversión. Es una asociación lógica, pero incompleta. **En la práctica, el trading algorítmico no es, en esencia, un problema financiero. Es un problema tecnológico.**

Un sistema de trading algorítmico no es más que una máquina que transforma información en acciones. Recibe datos, los procesa, genera una señal y ejecuta una orden. Todo esto ocurre de forma automática, sin intervención humana directa. Si lo piensas detenidamente, este patrón es idéntico al de muchos sistemas modernos: entrada de datos, procesamiento y ejecución. La diferencia no está en la estructura. **Está en lo que está en juego.**

EL ERROR MÁS COMÚN AL EMPEZAR

La mayoría de las personas empieza por el lugar equivocado. Se preguntan: "¿Qué estrategia funciona mejor?" o "¿Cómo puedo predecir el mercado?" Pero esas preguntas llegan demasiado pronto. Porque antes de poder ejecutar una estrategia, necesitas algo mucho más básico: **un sistema que funcione de forma continua.** Si no puedes ejecutar código de forma persistente, automatizada y conectada, cualquier estrategia —por buena que sea— es irrelevante.

EL PATRÓN FUNDAMENTAL

DE SCRIPTS A SISTEMAS: EL SALTO QUE IMPORTA

Script (no es suficiente)

Ejecutas código → obtienes resultado → todo termina. Depende de que estés delante. Si cierras el portátil, muere.

`frágil · manual`

Sistema (lo que construimos)

Sigue funcionando aunque no estés. Persistente, automatizado, conectado, con estructura. Se reinicia solo si cae.

`robusto · autónomo`

EL CAMBIO DE MENTALIDAD

� Dejas de pensar en...

- Indicadores que "predicen el futuro"
- Estrategias mágicas
- Señales perfectas

� Empiezas a pensar en...

- Sistemas que se ejecutan solos
- Automatización robusta con watchdogs
- Infraestructura que no se cae

LA VENTAJA REAL

La ventaja no está en la estrategia. Está en la capacidad de construir sistemas que operan sin ti. Cuando tienes esa capacidad: puedes probar, iterar, automatizar. Sin ella, todo se queda en teoría.

CAP. 02

El Servidor: Corazón del Sistema

Por qué necesitas un VPS y por qué elegimos Hetzner

> ¿Dónde vive el sistema? Todo lo que has probado en tu ordenador desaparece cuando cierras el portátil. Para construir algo real necesitas un entorno que no dependa de ti. **Un lugar donde el código siga ejecutándose aunque no estés delante. Ese lugar es un servidor.**

Un servidor es simplemente un ordenador que está siempre encendido y conectado a internet. No necesitas imaginar racks, cables o centros de datos. Desde tu punto de vista es una máquina a la que puedes acceder y que no se apaga cuando tú te desconectas. La diferencia real no está en el hardware, sino en el uso.

TU ORDENADOR VS UN SERVIDOR HETZNER

Tu ordenador local

- Lo enciendes y lo apagas
- Todo muere al cerrar la tapa
- Depende de que estés presente
- No accesible desde el exterior

`frágil · dependiente`

Servidor Hetzner VPS

- Activo 24 horas, 7 días a la semana
- Procesos persistentes con tmux
- Accesible desde cualquier lugar vía SSH
- No depende de que tú estés

`siempre encendido · autónomo`

POR QUÉ HETZNER Y NO AWS NI DIGITALOCEAN

Dentro de todas las opciones disponibles, trabajamos con **Hetzner Cloud**. No por teoría. Por pragmatismo.

Precio

Desde ~4€/mes. El más barato para la potencia que ofrece. AWS equivalente cuesta 8-10x más.

Rendimiento

CPUs AMD EPYC + SSD NVMe. Más que suficiente para ML, backtesting y trading en producción.

Sencillez

Interfaz clara y directa. Un servidor en marcha en menos de 60 segundos, sin configuración compleja.

Fiabilidad

Datacenter en Frankfurt y Helsinki. Baja latencia hacia exchanges europeos. GDPR compliance.

🔗 hetzner.com/cloud — pricing y planes

🔗 docs.hetzner.com/cloud

⚙ console.hetzner.cloud

NUESTRO SERVIDOR: QUÉ CORRE EN PRODUCCIÓN AHORA MISMO

```
# Sistema operativo base
Ubuntu 24.04 LTS · Linux 6.8.0-101-generic

# Servicios nativos activos
ssh          → acceso remoto seguro (puerto 22)
nginx        → reverse proxy para múltiples servicios
postgresql   → base de datos v16 (nativa + contenedores)
cron         → scheduler de procesos automáticos
php8.3-fpm   → PHP para WordPress sites
docker       → contenedores aislados por servicio

# Contenedores Docker activos (docker ps)
n8n              → workflows de automatización  :5678
listmonk         → newsletter sender           :9000
wordpress ×4     → sitios web                  :808x
postgres ×3      → bases de datos              :543x
nginx ×4         → reverse proxies por dominio
```

CONSEJO PRÁCTICO

Un servidor es un entorno perfecto para aprender. Puedes crearlo en minutos, borrarlo si algo sale mal y empezar de nuevo sin consecuencias. No hay riesgo real. Solo aprendizaje. Hetzner permite borrar y recrear servidores con un clic.

CAP. 03

Terminal y SSH: Hablar con el Servidor

La barrera invisible — y por qué no es tan difícil como parece

> A diferencia de tu ordenador, el servidor no tiene pantalla, ratón ni iconos sobre los que hacer clic. La forma de interactuar con él es a través de la terminal. Para muchas personas, la terminal es el primer gran bloqueo. **Pantalla negra. Texto. Comandos. Pero la realidad es mucho más sencilla de lo que parece.**

La terminal no es más que una forma diferente de comunicarse con un ordenador. Cuando usas tu ordenador de forma habitual, interactúas a través de interfaces visuales: haces clic, arrastras, abres ventanas. La terminal elimina todo eso y lo sustituye por algo más directo: **escribes instrucciones. El sistema responde.** Nada más.

SSH — SECURE SHELL: ABRIR LA PUERTA AL SERVIDOR

SSH (Secure Shell) es la forma de conectarte a tu servidor desde tu ordenador. Es como abrir una puerta segura hacia otra máquina. Cuando ejecutas ese comando y la conexión funciona, sigues viendo tu pantalla y escribiendo en tu teclado, pero ya no estás en tu ordenador. **Estás dentro del servidor.** Este cambio no es técnico. Es mental. Y cuando lo interiorizas, todo empieza a encajar.

```
# Conectarse al servidor por SSH
ssh usuario@65.21.xxx.xxx

# Con clave SSH (más seguro que contraseña)
ssh -i ~/.ssh/id_rsa usuario@65.21.xxx.xxx

# ~/.ssh/config — para no escribir la IP cada vez
Host mi-servidor
    HostName 65.21.xxx.xxx
    User usuario
    IdentityFile ~/.ssh/id_rsa

# Desde ese momento, solo necesitas:
ssh mi-servidor

# Y aparece el prompt del servidor
usuario@mi-servidor:~$
```

COMANDOS ESENCIALES PARA EMPEZAR

No necesitas aprender decenas de comandos desde el principio. Con estos puedes hacer casi todo al empezar. El resto vendrá con el uso.

COMANDO	FUNCIÓN	EJEMPLO REAL
`ls -la`	Lista archivos con permisos y tamaños	`ls -la ~/mi-proyecto/`
`cd`	Navegar entre carpetas	`cd mi-proyecto/app`
`mkdir -p`	Crear carpetas (con padres)	`mkdir -p logs data models`
`python3`	Ejecutar scripts Python	`python3 app/app.py`
`tail -f`	Ver logs en tiempo real	`tail -f logs/production.log`
`ps aux | grep`	Ver procesos en ejecución	`ps aux | grep python`
`kill -9`	Matar un proceso por PID	`kill -9 12345`
`htop`	Monitor de recursos interactivo	`htop`
`df -h`	Ver espacio en disco	`df -h /`
`free -h`	Ver uso de RAM	`free -h`

CAP. 04

tmux: Persistencia Sin Fisuras

El gestor de sesiones que mantiene el sistema vivo aunque te desconectes

> Hay un problema silencioso que nadie te explica al principio. Si te conectas al servidor, ejecutas un script y cierras la terminal, el script muere. La conexión SSH es un hilo: cuando lo cortas, los procesos que dependían de él desaparecen. **tmux resuelve esto de forma elegante.**

Cuando te conectas por SSH, estás creando una sesión activa entre tu ordenador y el servidor. Esa sesión es el canal por el que fluye todo: tus comandos, la ejecución, la salida. Pero también es una dependencia. Cuando cierras la conexión, el servidor interpreta que esa sesión ha terminado. Y con ella, los procesos asociados. **tmux introduce una capa intermedia que elimina esa dependencia.**

SIN TMUX VS CON TMUX

Sin tmux (dependiente)

Al cerrar la terminal → el script muere. Sin recuperación posible.

Con tmux (independiente)

Al cerrar la terminal → tmux sigue. El script sigue. Todo sigue.

COMANDOS TMUX QUE REALMENTE USAMOS

```
# Crear sesión con nombre
tmux new -s trading
tmux new -s watchdog

# Salir sin cerrar (detach) — el gesto clave
Ctrl+B → D

# Volver a entrar (attach)
tmux attach -t trading
tmux a           # a la última sesión

# Ver todas las sesiones activas
tmux ls
# → trading: 1 windows (created Sat Apr 26) [220x50]
# → watchdog: 1 windows (created Fri Apr 25) [220x50]

# Dividir pantalla para ver trader + logs a la vez
Ctrl+B → %    # divide vertical
Ctrl+B → "    # divide horizontal
Ctrl+B → ←→   # moverse entre paneles

# Matar sesión
tmux kill-session -t trading
```

tmux wiki (GitHub)

⌨ tmuxcheatsheet.com

FLUJO DE TRABAJO REAL EN PRODUCCIÓN

SSH al servidor → tmux attach -t trading → compruebas que el trader sigue → Ctrl+B D → cierras la terminal. El sistema sigue corriendo en Hetzner mientras tú haces otra cosa. Días después, tmux attach y vuelves exactamente donde lo dejaste.

CAP. 05

VSCode Conectado al Servidor

Editar código en el servidor como si fuera local — sin fricción

> La terminal es potente, directa y esencial. Pero no está pensada para todo. Escribir código largo, navegar entre archivos, organizar proyectos... todo eso se vuelve lento y poco intuitivo si solo usas comandos. **La solución no es elegir entre comodidad y potencia. Es combinarlas con VSCode Remote SSH.**

Visual Studio Code tiene una extensión que cambia completamente la forma de trabajar: **Remote - SSH**. Esta extensión te permite conectarte a tu servidor igual que lo haces con la terminal, pero desde el editor. Una vez configurada, ves los archivos como si estuvieran en tu ordenador, editas con interfaz gráfica completa, tienes autocompletado, colores, navegación. Pero en realidad, **todo está ocurriendo en tu servidor.** No estás copiando archivos. No estás duplicando nada. Estás trabajando directamente sobre la máquina remota.

CONFIGURACIÓN DE REMOTE SSH

```
# 1. Instalar la extensión en VSCode
#    Ctrl+Shift+X → buscar "Remote - SSH"
#    Publisher: Microsoft · ms-vscode-remote.remote-ssh

# 2. Tu ~/.ssh/config en el ordenador LOCAL
Host mi-servidor
    HostName 65.21.xxx.xxx
    User usuario
    IdentityFile ~/.ssh/id_rsa
    ServerAliveInterval 60

# 3. En VSCode: Ctrl+Shift+P
>Remote-SSH: Connect to Host...
→ selecciona: mi-servidor

# 4. Se abre una nueva ventana VSCode
#    Abajo izquierda: "><< SSH: mi-servidor"
#    File > Open Folder → ~/mi-proyecto/mi-proyecto
```

code.visualstudio.com | Remote SSH extension | Docs Remote SSH

FLUJO DE TRABAJO COMPLETO CON VSCODE

Error típico: copiar archivos

- Escribes código en local
- Lo copias al servidor con scp o FTP
- Lo ejecutas allí

Genera desorden, versiones duplicadas y errores innecesarios. Hay dos "verdades".

Con Remote SSH

- Solo hay una versión: la del servidor
- Guardas → se guarda en Hetzner directamente
- Ejecutas desde terminal integrada

Todo en un solo lugar. Sin sincronización. Sin duplicados.

CAP. 06

Organización del Proyecto y GitHub

La estructura que permite crecer sin caos — empezar bien desde el primer día

Cuando empiezas, todo parece manejable. Un archivo aquí. Otro allá. Un script con nombre improvisado. Todo funciona… hasta que deja de hacerlo. El problema no aparece el primer día. **Aparece cuando tienes 10 scripts, varias versiones y código que no sabes para qué sirve. La organización no es estética. Es funcional.**

ESTRUCTURA DE PROYECTO MINIMALISTA

Ordena tu código en carpetas simples. La estructura que funciona:

```
mi-proyecto/
│
├── src/ # código fuente
│ ├── __init__.py
│ ├── bot.py # lógica principal
│ ├── data.py # descarga y almacenamiento
│ └── alerts.py # notificaciones Telegram
│
├── config/
│ ├── config.yaml # parámetros del bot
│ └── .env # API keys — NUNCA en git
│
├── data/ # base de datos y archivos
│
├── logs/ # archivos de log
│
├── tests/ # tests unitarios
│
├── requirements.txt # dependencias Python
├── .gitignore # qué ignorar en git
└── README.md # documentación básica
```

CONTROL DE VERSIONES CON GIT

> **Git** es el sistema de control de versiones estándar. Te permite guardar "fotos" de tu código en diferentes momentos, volver atrás si algo falla, y trabajar en ramas paralelas sin romper lo que funciona.

CONCEPTOS ESENCIALES

- **Repository (repo)** — La carpeta que Git vigila. Contiene todo el historial.
- **Commit** — Una "foto" del código en un momento específico, con mensaje descriptivo.
- **Branch (rama)** — Una línea de desarrollo paralela. La principal se llama `main` o `master`.
- **Stage** — Área intermedia donde preparas cambios antes de commitear.
- **Push/Pull** — Subir cambios al servidor remoto / bajarlos.

FLUJO DE TRABAJO BÁSICO

```
# Inicializar repo en carpeta existente
git init

# Ver estado de archivos
git status

# Añadir archivos al stage
git add nombre_archivo.py
# o añadir todo:
git add .

# Crear commit con mensaje
git commit -m "feat: añadir módulo de alertas"

# Ver historial de commits
git log --oneline
```

GITHUB: TU CÓDIGO EN LA NUBE

GitHub es el servicio que aloja repositorios Git en internet. Permite:

- **Backup remoto** — Si tu ordenador explota, tu código está a salvo.
- **Colaboración** — Múltiples personas trabajando en el mismo proyecto.
- **Issues** — Seguimiento de bugs y mejoras.
- **GitHub Actions** — Automatización (tests, deployment).
- **README** — Documentación visible para quien visita tu repo.

CONECTAR LOCAL CON REMOTO

```
# Crear repo vacío en GitHub primero
# Luego en tu terminal local:

# 1. Añadir el remoto
git remote add origin https://github.com/tuusuario/mi-proyecto.git

# 2. Subir cambios
git push -u origin main

# Trabajo diario después:
git add .
git commit -m "fix: corregir cálculo de posición"
git push
```

SEGURIDAD: .GITIGNORE

Nunca subas **API keys, contraseñas ni archivos sensibles** a GitHub. Crea un archivo `.gitignore`:

```
# .gitignore
.env
config/secrets.yaml
data/*.csv
logs/*.log
__pycache__/
*.pyc
```

DESHACER ERRORES

```
# Ver cambios antes de commitear
git diff

# Deshacer cambios en un archivo (volver al último commit)
git checkout -- nombre_archivo.py

# Corregir el último commit (si no has hecho push)
git commit --amend -m "mensaje corregido"

# Volver a un commit anterior específico
git checkout abc1234
```

BUENAS PRÁCTICAS

Commits pequeños y frecuentes. Mensajes descriptivos: `"feat: añadir..."`, `"fix: corregir..."`, `"docs: actualizar..."`

 github.com docs.github.com gitignore.io

CAP. 07

Claude Code: Tu Copiloto

La pieza que hace todo el resto posible

No necesitas saber Python. Necesitas saber pedírselo a Claude.

> **Claude Code** es un agente de IA que vive en tu terminal. No es un chat: es un compañero que escribe código, explica errores, refactoriza y conecta sistemas mientras tú piensas en arquitectura.

¿QUÉ HACE EXACTAMENTE?

- **Escribe código** desde descripciones en lenguaje natural
- **Explica** qué hace código que no entiendes
- **Debuggea** errores y propone soluciones
- **Refactoriza** código spaghetti en módulos limpios
- **Conecta** APIs, bases de datos, servicios
- **Genera tests** automáticamente

INSTALACIÓN PASO A PASO

Instalación en el servidor:

```
# Método 1: Script oficial
curl -fsSL https://claude.ai/install.sh | bash

# Método 2: NPM
npm install -g @anthropics/claude-code

# Verificar instalación
claude --version

# Autenticar
claude auth
# Abre navegador para vincular tu cuenta
```

INTEGRACIÓN CON VSCODE

La extensión conecta Claude con tu editor:

```
# 1. En VSCode: Ctrl+Shift+X
# 2. Buscar "Claude Code" (publisher: Anthropic)
# 3. Instalar

# Abrir Claude en un proyecto:
cd ~/mi-proyecto
claude

# Atajo rápido: escribe punto
.
# → Enter abre Claude con contexto del directorio actual
```

CONTEXTO

Claude ve tu código actual, los archivos del proyecto y el historial de la conversación. No necesitas copiar y pegar código: trabaja directamente sobre tus archivos.

PROMPTS QUE FUNCIONAN

Estrategia: ser específico. Qué quieres, qué falla, dónde:

```
# □ Vago
"Arregla esto"

# □ Específico
"Me da error 'Connection reset by peer' cada 3 horas en binance_client.py
línea 45. Usa requests.Session con retries y exponential backoff.
Loguea cada retry con nivel WARNING."

# □ Con contexto
"Este script de 400 líneas es spaghetti. Refactorízalo:
- Extrae la lógica de descarga a una clase DataLoader
- Usa dependency injection para la DB
- Añade type hints
- Genera tests unitarios para cada función pura"
```

PATRONES DE TRABAJO CON CLAUDE

- **Spike** — "Escríbeme un prototipo de..." para explorar
- **Debug** — "Me da este error, investiga y explica por qué"
- **Refactor** — "Este código funciona pero es un desastre, límpialo"
- **Review** — "Revisa este módulo, encuentra bugs potenciales"
- **Connect** — "Conecta este endpoint POST a PostgreSQL"

ALTERNATIVAS VÁLIDAS

- **Cursor** — IDE completo con IA integrada (fork de VSCode)
- **Continue.dev** — open source, múltiples modelos
- **GitHub Copilot** — autocompletado inteligente

- **Cody** de Sourcegraph — entiende tu códigobase

DE AQUÍ EN ADELANTE

Cada capítulo asume Claude instalado. No copiarás código de ejemplos: le pedirás a Claude que lo genere.

CAP. 08

Scripts, Systemd y Telegram

De manual a automático: ejecución 24/7 sin supervisión

Si tienes que estar delante para que funcione, no es un sistema. Es un experimento.

> La automatización tiene tres niveles: **scripts** que hacen el trabajo, **systemd** que los mantiene vivos, y **alertas** que te avisan sin mirar logs.

SCRIPTS CON ESTRUCTURA ROBUSTA

Pide a Claude un template con manejo profesional:

```
# Ejemplo de estructura para scripts de trading
> Crea template Python de bot de trading con:
> - Logging configurable (niveles INFO/WARNING/ERROR)
> - Manejo de KeyboardInterrupt para cierre limpio
> - Carga de variables desde .env con dotenv
> - Bucle try/except que nunca muere, solo loguea errores
> - Función send_alert() para Telegram
```

Resultado demo:

```python
import logging
import os
import time
from dotenv import load_dotenv

# Cargar variables
load_dotenv()
API_KEY = os.getenv('API_KEY')

# Configurar logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s | %(levelname)s | %(message)s',
    handlers=[
        logging.FileHandler('bot.log'),
        logging.StreamHandler()
    ]
)

# Bucle principal
def main():
    logging.info("□ Bot iniciado")
    while True:
        try:
            # Tu lógica aquí
            procesar_velas()
            time.sleep(60)
        except KeyboardInterrupt:
            logging.info("□ Cierre ordenado")
            break
        except Exception as e:
            logging.error(f"□ Error: {e}")
            send_alert(f"Error: {e}")
            time.sleep(10)  # Esperar antes de retry

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

CRON: EL RELOJ SIMPLE

Para tareas periódicas (descargas, reportes):

```
# Editar crontab
crontab -e

# Ejecutar script cada hora
0 * * * * cd ~/mi-proyecto && \
  source venv/bin/activate && \
  python download_data.py >> logs/cron.log 2>&1

# Sintaxis: minuto hora día mes día_semana comando
# @ daily = todos los días a las 00:00
0 9 * * * python reporte_diario.py  # Cada día a las 9:00
```

SYSTEMD: SERVICIOS QUE RESUCITAN

Cuando el script debe estar SIEMPRE corriendo:

```
# Pedir a Claude:
> Crea un servicio systemd para mi bot_trading.py
> Que se reinicie automáticamente, use venv, cargue .env

# Archivo generado: /etc/systemd/system/mi-bot.service
# Contenido típico:
[Unit]
Description=Mi Bot de Trading
After=network.target

[Service]
Type=simple
User=santi
WorkingDirectory=/home/santi/mi-proyecto
Environment=PYTHONPATH=/home/santi/mi-proyecto
ExecStart=/home/santi/mi-proyecto/venv/bin/python bot_trading.py
Restart=always
RestartSec=10

[Install]
WantedBy=multi-user.target

# Comandos para gestionar:
sudo systemctl daemon-reload
sudo systemctl enable mi-bot.service  # Arrancar al boot
sudo systemctl start mi-bot.service   # Iniciar ahora# Ver estado
sudo systemctl stop mi-bot.service    # Detener
sudo journalctl -u mi-bot -f          # Ver logs en vivo
```

TELEGRAM: TU DASHBOARD PORTABLE

Obtén tu token en @BotFather, luego:

```
import asyncio
from telegram import Bot

BOT_TOKEN = os.getenv('TELEGRAM_BOT_TOKEN')
CHAT_ID = os.getenv('TELEGRAM_CHAT_ID')

bot = Bot(token=BOT_TOKEN)

def send_alert(mensaje):
    asyncio.run(bot.send_message(chat_id=CHAT_ID, text=mensaje))

def send_report(datos):
    mensaje = f"""
□ Reporte {datetime.now().strftime('%H:%M')}
Señales: {datos['senal']}
Estado: {datos['estado']}
P/L: {datos['pnl']}%
    """
    send_alert(mensaje)

# Uso
send_alert("⚠ Rate limit alcanzado")
send_alert("□ Bot iniciado correctamente")
```

CRÍTICO: SEGURIDAD

Al crear el bot con @BotFather, ejecuta `/setprivacy` y desactiva el acceso a mensajes grupales. El bot solo debe responder a tu chat_id. Nunca hagas push de tokens.

FLUJO DE AUTOMATIZACIÓN COMPLETO

1. **Desarrollo**: Escribir script con Claude
2. **Test**: Ejecutar manualmente, verificar logs
3. **Systemd**: Crear servicio, enable, start
4. **Monitor**: Telegram confirma inicio
5. **Alertas**: Errores llegan a Telegram
6. **Mantenimiento**: journalctl para debugging

GOLDEN RULE

Un sistema bien automatizado te permite estar una semana sin mirarlo. Si eso te da ansiedad, necesitas mejores alertas, no más pantallas.

CAP. 09

Stack de Datos

De la API de Binance a PostgreSQL/TimescaleDB, pasando por Claude

Sin datos no hay backtesting. Sin backtesting no hay estrategia.

> El stack de datos tiene tres capas: **la fuente** (Binance API), **el almacén** (PostgreSQL + TimescaleDB), y **el motor de análisis** (Pandas o Polars). No escribas cada conector. **Pídeselo a Claude.**

Binance ofrece una API pública gratuita. No necesitas cuenta verificada para descargar velas históricas. Para trading real sí necesitas API keys, pero para obtener datos basta una conexión HTTP. El endpoint `/api/v3/klines` devuelve 1000 velas por llamada con un rate limit de 1200 req/min.

PROMPT: CLIENTE COMPLETO

```
# Prompt para Claude
> Crea un stack de datos completo:
> - Cliente Binance REST que descargue velas históricas
> - Respete rate limits, soporte múltiples símbolos e intervalos
> - Inserte en PostgreSQL con psycopg2 + execute_values
> - Evite duplicados con ON CONFLICT DO NOTHING
> - Esquema SQL con hypertable de TimescaleDB
> - .env, requirements.txt y README.md
> - Logging de progreso y errores
```

QUÉ ESPERAR

Claude generará una clase `DataDownloader`, el esquema SQL, y los archivos de config. Guárdalo en `src/data.py`, revísalo y ejecuta.

POSTGRESQL + TIMESCALEDB

PostgreSQL

Relacional, ACID, maduro. SQL estándar sin sorpresas.

TimescaleDB

Hypertables, compresión nativa, 10-100x más rápido en series temporales.

`recomendado`

TIMEFRAMES

TF	VELAS/DÍA	1 AÑO	USO
`1m`	1,440	~525K	Scalping
`5m`	288	~105K	Intradía
`1h`	24	~8.7K	Swing, ML
`1d`	1	365	Macro

EMPIEZA SIMPLE

No descargues 1m de 500 pares. Empieza con **1h de BTCUSDT y ETHUSDT**. Un año son ~17K velas. Pesa poco y es suficiente para prototipar.

PANDAS VS POLARS

Pandas

Estándar de facto. API madura, integración total con matplotlib y scikit-learn.

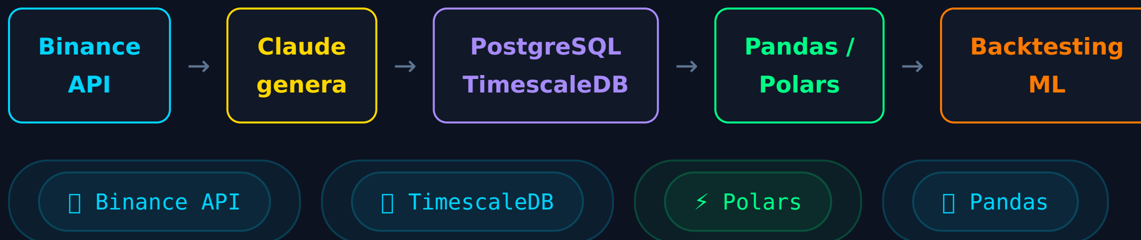
`default · < 1M filas`

Polars

Escrito en Rust. 10-50x más rápido en lecturas grandes. Lazy evaluation.

`performance · > 1M filas`

FLUJO



SIGUIENTE PASO

"Usa los datos de BTCUSDT 1h en PostgreSQL. Haz un backtesting de RSI(14). Métricas: retorno, drawdown, win rate. Guarda en backtest_results."

CAP. 10

Logs y Observabilidad

Saber qué está pasando sin mirar la pantalla

Si no hay logs, no hay sistema. Si no hay alertas, no sabes si falla.

> Un sistema de trading que corre 24/7 necesita **observabilidad**: logs que registren todo, métricas que midan rendimiento, y alertas que avisen cuando algo va mal.

LOGGING EN PYTHON

No uses `print()`. Usa el módulo `logging` con timestamps y niveles:

```
import logging

logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s | %(levelname)s | %(message)s',
    handlers=[
        logging.FileHandler('app.log'),
        logging.StreamHandler()
    ]
)

logging.info("□ Bot iniciado")
logging.warning("⚠ Rate limit")
logging.error("□ Error: %s", e)
```

JOURNALCTL PARA SERVICIOS

Cuando tu script corre como servicio, los logs van a journalctl:

```
# Ver logs en tiempo real
sudo journalctl -u mi-bot -f

# Últimas 100 líneas
sudo journalctl -u mi-bot -n 100

# Desde hoy
sudo journalctl -u mi-bot --since today
```

ROTACIÓN DE LOGS

Los logs crecen infinitamente. Configura rotación:

```
from logging.handlers import RotatingFileHandler

handler = RotatingFileHandler(
    'app.log',
    maxBytes=10*1024*1024,  # 10 MB
    backupCount=5
)
```

BUENA PRÁCTICA

Un archivo por día: `logs/app_2026-04-27.log`. Más fácil de analizar.

TELEGRAM COMO DASHBOARD

Resumen periódico con métricas:

```
send_report("""
 Resumen 4h
• Velas: 240
• Señales: 3 LONG, 1 SHORT
• PnL: +0.45%
• Estado:  Operativo
""")
```

CHECKLIST

- **Logs estructurados** — timestamps, niveles, mensajes claros
- **Rotación** — para no llenar el disco
- **Alertas realtime** — errores críticos por Telegram
- **Reportes periódicos** — métricas de negocio
- **Monitoreo** — CPU, RAM, espacio en disco

MINIMALISMO

No necesitas Grafana ni ELK. Un buen logging + Telegram es suficiente.

CAP. 11

Ahora Estás Preparado

Tienes la máquina. Ahora la estrategia.

> Este manual era sobre **infraestructura**. Las estrategias, los modelos ML, el backtesting — todo eso viene después. Pero sin esta base sólida, nada funciona.

LO QUE TIENES AHORA

- **Claude Code** — Tu copiloto de desarrollo
- **Hetzner VPS** — Servidor 24/7 en Europa
- **PostgreSQL + TimescaleDB** — Millones de velas
- **Pipeline de datos** — Descarga automática
- **systemd** — Servicios persistentes
- **Telegram Bot** — Alertas en tiempo real
- **Logs** — Observabilidad completa
- **Git** — Control de versiones

LO QUE VIENE DESPUÉS

- **Backtesting** — Validar estrategias con Backtrader
- **ML** — Modelos de predicción con XGBoost
- **Optimización** — Hiperparámetros con Optuna
- **Execution** — Trading real en exchanges

EL MINIMALISMO TÉCNICO

Datos → Backtesting → ML si necesario → Execution.
No añadas complejidad hasta que la simpleza te limite.

PRÓXIMO PASO CONCRETO

```
> Crea un backtesting simple con Backtrader
> Usa los datos que ya tienes en PostgreSQL
> Testea RSI(14) con parámetros por defecto
>> Métricas: retorno, drawdown, número de trades
>> Si funciona, iterar. Si no, ajustar.
```

Menos stack. Más ejecución.

La infraestructura no es el objetivo. Es el medio.
El objetivo es ejecutar estrategias que funcionen.

Santiago Fernández — @sanfernandezf

sanfernandezf@gmail.com

LinkedIn

X / Twitter

"Si montas esto y quieres compartir tu progreso, me encantará saber de ti.
El trading algorítmico es difícil. La infraestructura no tiene por qué serlo."