OpenClaw

The Complete Mastery Guide

🇬🇧 English Edition | 300+ Pages | 2026 Edition | Full Reference

From first install to advanced agent swarms —
everything you need to master the most powerful
open AI assistant platform

Written by Ariel Eisenstadt

Founder, PixMind Studio & PixiBot • Software Engineering Student

👋

About This Guide

OpenClaw Complete Mastery Guide — English Edition

This guide was written for everyone who wants to understand and **truly master** the OpenClaw platform — one of the most advanced AI-powered personal assistant platforms available today.

From simple installation to building complex **Agent Swarms**, this guide covers every aspect of the platform — in clear language, with practical code examples, and a complete reference for every command and configuration.

Whether you're a developer, product manager, startup founder, or simply curious about AI automation — this book is your starting point for the next AI revolution.

🚀 **Ariel Eisenstadt**

🎬 Founder & Owner — **PixMind Studio** (video & advertising production, working with top companies)

🤖 Developer & Owner — **PixiBot** (AI startup, in active development)

🎓 Software Engineering Student • Age 18

500K+	53	70+	∞
Lines of Code	Config Files	Dependencies	Possibilities

⭐ **How to Use This Guide**

Each chapter stands on its own — read sequentially or jump directly to topics that interest you. All code examples are real and immediately usable. Look for the emoji markers throughout: ⭐

tips, ❌ common mistakes, ❓ FAQ, ⚙️ behind the scenes, 📌 key points, 💻 hands-on exercises.

Table of Contents

CHAPTER ZERO

Introduction to OpenClaw

00

What OpenClaw is, why it exists, and why it's fundamentally different from every other AI assistant you've seen — including the half-million-line behemoth it was inspired by

CHAPTER 00

Introduction to OpenClaw

What it is, why it exists, and why it's different

⏱ Estimated reading time: 45 minutes

🎯 **Learning Objectives**

- Understand what the OpenClaw platform is and what problem it solves
- Identify the key differences between OpenClaw and other AI assistants
- Learn the fundamental architecture of the system
- Understand the security principle built into OpenClaw's core
- Know what you can realistically build with OpenClaw

What is OpenClaw?

In 2023, as AI assistants like ChatGPT and Claude became everyday tools, an important question emerged: **how do you make an AI assistant actually work for you — not just answer questions, but take real actions in your life?**

OpenClaw was born from that need. It's not a simple "ask and answer" application — it's a **complete platform** that connects the most capable AI models to your daily life: WhatsApp, Telegram, Gmail, Slack, Discord, and more.

📖 **Key Concept: Agent**

Unlike a regular chatbot that just generates text, an **AI agent** can take real actions: open files, run code, browse the web, send emails, and more. OpenClaw is built around these kinds of agents.

What sets OpenClaw apart from the competition is its approach to security: every AI agent runs **inside an isolated Linux container** — not just behind permission checks, but in true OS-level isolation. The boundary is enforced by the kernel, not by logic that can be bypassed.

The History: Where Did OpenClaw Come From?

🕐 2022 — The Background

Large language models (LLMs) start becoming genuinely useful. ChatGPT launches. People realize AI "works" — but it's still limited to a text interface with no real-world actions.

🕒 2023 — The Need

The team behind OpenClaw decides: we want an assistant that responds in WhatsApp, remembers context across conversations, can run code — and can be trusted from a security perspective.

🕓 2024 — Development

OpenClaw grows to nearly 500,000 lines of code, 53 configuration files, and 70+ dependencies. The platform becomes one of the most sophisticated in the personal AI assistant space.

🕔 2025–2026 — Maturity

Support for Agent Swarms, deep integration with Claude Agent SDK, and MCP (Model Context Protocol) support. OpenClaw becomes the standard for open-source personal AI assistants.

OpenClaw vs The Competition

Feature	OpenClaw	ChatGPT API	Botpress	Rasa
Container isolation	✓ True OS-level	✗	✗	✗
Multi-channel (native)	✓ Built-in	✗ Manual	✓	✓
Agent Swarms	✓ First to support	✗	✗	✗
Fully open source	✓	✗	Partial	✓
Claude Agent SDK	✓ Full	✗	✗	✗
Per-group isolated memory	✓ Isolated	Limited	✓	Limited
Codebase size	Small (<3k lines core)	N/A	Large	Large

The Architecture at a Glance

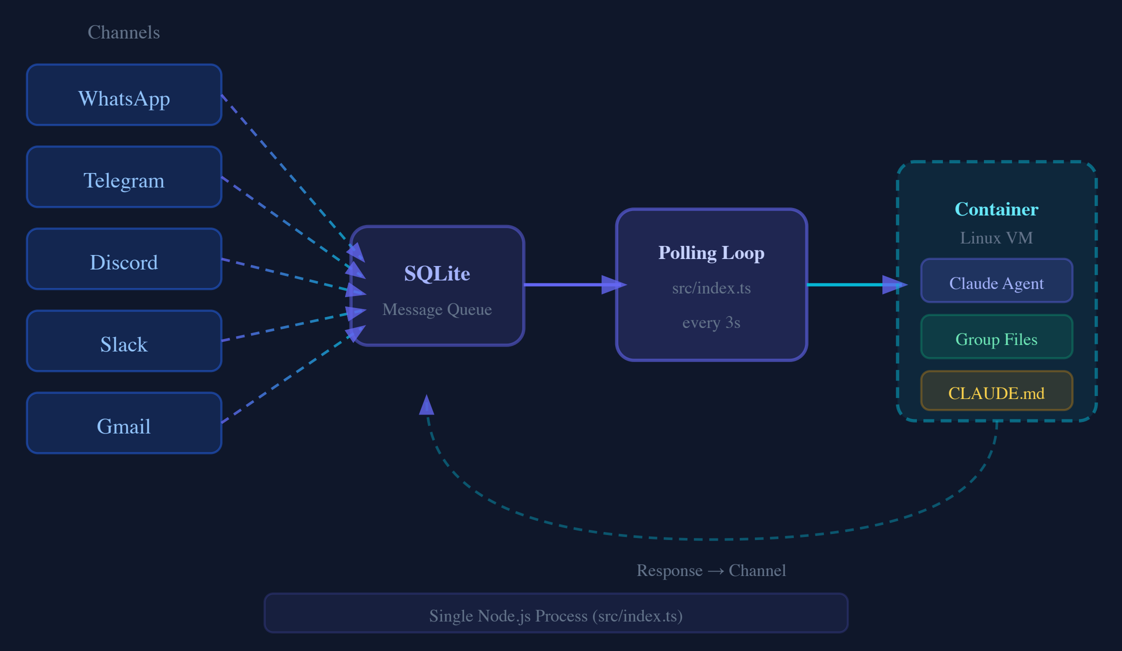


Figure 0.1: OpenClaw data flow — from incoming message to outgoing response

⭐ **The Entire System is One Process**

The entire system runs as a **single Node.js process**. This sounds overly simple — and that's exactly the point. Simplicity enables security, and security enables trust. A codebase small enough to read is a codebase small enough to understand and audit.

OpenClaw's Core Principles

To use OpenClaw correctly, you need to understand the design principles that shaped it. These aren't marketing slogans — they translate directly into architectural decisions in the code.

🔒

Security Through Isolation

Every AI agent runs in its own separate Linux container. It can only access directories that are explicitly mounted to it. Bash is safe — because commands run inside the container, not on your host machine.

📦

Small Enough to Understand

OpenClaw is designed so you can read and understand the entire core codebase. No complex microservices, no unnecessary abstraction layers. One process, a handful of files. You can audit what you run.

🎯

AI-Native Design

No installation wizard — Claude Code guides setup. No monitoring dashboard — ask Claude what's happening. No debug tools — describe the problem and Claude fixes it. Every interaction is designed for AI collaboration.

🛠️

Skills Over Features

Instead of adding features to the base code, contributors submit skill files that teach Claude Code how to transform your fork. You get clean, focused code that does exactly what you need — nothing more.

What Can You Build With OpenClaw?

🤖 Personal AI Assistant

```
WHATSAPP

@Andy, summarize the team meeting from this week's Git history
and create a list of action items for each person.
```

📊 Business Automation

```
WHATSAPP

@Andy, send me a sales pipeline overview from our CRM
every weekday morning at 9am.
```

🔬 Automated Research

WHATSAPP

```
@Andy, every Monday at 8am compile AI news from Hacker News
and TechCrunch into a briefing and send it to me here.
```

🎉 The Bottom Line

OpenClaw is not just a tool — it's an **AI platform** that lets you build any assistant you can imagine. In the following chapters, you'll learn how to do this correctly, securely, and efficiently.

📌 Key Points — Introduction

- OpenClaw is a multi-channel personal AI assistant that runs on your server
- Every AI agent is isolated in a Linux container — real OS-level security
- Single Node.js process — simple enough to fully understand
- First personal assistant to support Agent Swarms
- Built on the Claude Agent SDK — the most capable AI agent available

\n\n

01

Installation & First Run

From zero to a running multi-channel AI assistant in under 30 minutes

Chapter 01 — Installation & First Run

1.1 System Requirements

OpenClaw is designed to run on modern developer machines and cloud servers. Before beginning, verify that your environment meets the following requirements:

Component	Minimum	Recommended
Node.js	18.x LTS	22.x LTS (latest)
RAM	4 GB	8 GB+
Storage	2 GB free	10 GB+ (logs, DB)
OS	Linux / macOS	Ubuntu 22.04 / macOS 14+
Docker	24.x	Latest stable
Anthropic API Key	Required	Claude Sonnet 4.6 tier

About NanoClaw
If you want to run a lighter version without Docker containers, NanoClaw is the single-process alternative. It has zero container overhead and is perfect for development, testing, and low-volume deployments. All core concepts from this book apply to both platforms.

1.2 Installing Node.js

We recommend using `nvm` (Node Version Manager) to manage your Node.js installation:

```
# Install nvm
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.7/install.sh | bash

# Reload your shell config
source ~/.bashrc  # or ~/.zshrc

# Install and use Node 22 LTS
nvm install 22
nvm use 22
nvm alias default 22
```

```
# Verify
node --version    # → v22.x.x
npm --version     # → 10.x.x
```

1.3 Cloning the Repository

```
# Clone OpenClaw (main platform)
git clone https://github.com/your-org/openclaw.git
cd openclaw

# OR clone NanoClaw (lightweight alternative)
git clone https://github.com/your-org/nanoclaw.git
cd nanoclaw
```

Pro Tip

Fork the repository on GitHub before cloning. This lets you push your customizations, CLAUDE.md files, and skills back to your own remote without affecting the original project.

1.4 Installing Dependencies

```
# Install all Node dependencies
npm install

# Verify all packages installed correctly
npm ls --depth=0
```

OpenClaw has 70+ direct dependencies. The most critical ones include:

Core Runtime

- `@anthropic-ai/sdk` — Claude API client
- `better-sqlite3` — Local SQLite DB
- `tsx` — TypeScript hot-reload
- `zod` — Schema validation

Messaging Channels

- `whatsapp-web.js` — WhatsApp bridge
- `node-telegram-bot-api` — Telegram bot
- `discord.js` — Discord gateway
- `@slack/bolt` — Slack Events API

1.5 Environment Configuration

Copy the example environment file and fill in your credentials:

```
cp .env.example .env
nano .env  # or use your preferred editor
```

Required variables in `.env`:

```
# === REQUIRED ===
ANTHROPIC_API_KEY=sk-ant-...          # Your Anthropic API key

# === WHATSAPP (optional) ===
# WhatsApp uses QR-based auth, no token needed here

# === TELEGRAM (optional) ===
TELEGRAM_BOT_TOKEN=123456:ABC...      # From @BotFather

# === DISCORD (optional) ===
DISCORD_BOT_TOKEN=MTI...                # From Discord Developer Portal
DISCORD_CLIENT_ID=123456789             # Application ID

# === SLACK (optional) ===
SLACK_BOT_TOKEN=xoxb-...               # Bot OAuth token
SLACK_APP_TOKEN=xapp-...               # App-level token (Socket Mode)
SLACK_SIGNING_SECRET=abc123...         # From Slack app settings

# === DATABASE ===
DB_PATH=./data/openclaw.db             # SQLite file path

# === AGENT CONTAINER ===
CONTAINER_IMAGE=openclaw-agent:latest
MAX_CONTAINERS=5                        # Parallel agent limit
```

Security Warning

Never commit your `.env` file to version control. The repository includes a `.gitignore` entry for it, but double-check before every push. Leaked API keys can lead to significant charges.

1.6 WhatsApp Authentication

WhatsApp requires a one-time QR code scan to establish a session. Run the dedicated auth script:

```
npm run auth
```

A QR code will appear in your terminal. Open WhatsApp on your phone → Settings → Linked Devices → Link a Device → scan the QR code.

The session is saved locally in `./data/wwebjs_auth/`. You won't need to re-authenticate unless you log out or delete this folder.

WhatsApp Session Persistence
The auth session uses **LocalAuth** strategy, persisting the session to disk. This means the bot survives restarts. In production deployments, mount this folder as a Docker volume to ensure persistence across container restarts.

1.7 First Launch

Start the platform in development mode with hot reload:

```
npm run dev
```

You should see output similar to:

```
✅ Database initialized at ./data/openclaw.db
✅ WhatsApp client ready
✅ Telegram bot connected (@YourBotName)
✅ Discord gateway READY (guilds: 3)
✅ Agent container pool initialized (0/5)
🚀 OpenClaw v2.0 running — all channels active
```

1.8 Sending Your First Message

Send any message to your connected WhatsApp / Telegram / Discord bot. OpenClaw will:

1
Receive the message via the channel adapter
2
Create or retrieve the conversation session from SQLite
3
Load the group's `CLAUDE.md` memory file (if any)
4
Send the message history + system prompt to Claude API
5

Stream the response back to the user in real-time

1.9 Production Build

For production, compile TypeScript to JavaScript first:

```
# Compile TypeScript
npm run build

# Run the compiled build
node dist/index.js
```

Process Manager

In production, use `pm2` or `systemd` to keep the process alive. See Chapter 12 (Production Deployment) for complete production setup including Docker Compose, nginx reverse proxy, and monitoring.

02

System Architecture

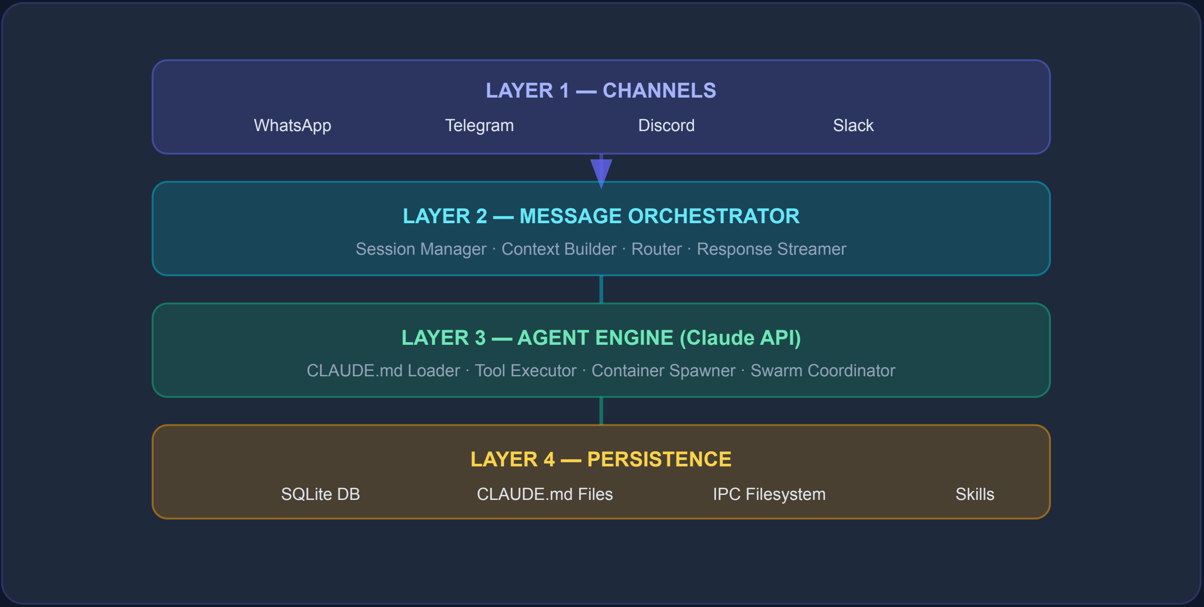
Deep-dive into OpenClaw's layered design: channels, orchestrator, agents, and storage

29

Chapter 02 — System Architecture

2.1 The Four Layers

OpenClaw is organized into four distinct layers, each with a clear responsibility boundary:


LAYER 1 — CHANNELS
WhatsApp
Telegram
Discord
Slack
LAYER 2 — MESSAGE ORCHESTRATOR
Session Manager · Context Builder · Router · Response Streamer
LAYER 3 — AGENT ENGINE (Claude API)
CLAUDE.md Loader · Tool Executor · Container Spawner · Swarm Coordinator
LAYER 4 — PERSISTENCE
SQLite DB
CLAUDE.md Files
IPC Filesystem
Skills

2.2 Layer 1 — Channel Adapters

Every messaging platform speaks a different protocol. OpenClaw abstracts them all behind a unified `IncomingMessage` interface:

```
interface IncomingMessage {
  id: string;
  groupId: string;            // Channel-specific group/chat ID
  senderId: string;           // User identifier
  senderName: string;         // Display name
  content: string;            // Text content
  attachments?: Attachment[]; // Images, docs, audio
  channel: 'whatsapp' | 'telegram' | 'discord' | 'slack';
  timestamp: Date;
}
```

2.3 The Message Orchestrator

The orchestrator is the central hub of OpenClaw. When a message arrives, it executes this pipeline:

1. **Deduplication** — Check if this message ID was already processed (prevents double-responses on reconnects)
2. **Session Lookup** — Fetch or create the session record for this group from SQLite
3. **Context Assembly** — Load message history (last N messages), CLAUDE.md system prompt, and any active skills
4. **Route Decision** — Determine if this should go to: a regular Claude API call, an agent with tools, or a multi-agent swarm
5. **Execution** — Call the appropriate handler and stream the response
6. **Persistence** — Store the user message and AI response in SQLite
7. **Delivery** — Send the response back through the channel adapter

2.4 Agent Engine Internals

When a message requires tool use (e.g., file operations, web search, code execution), OpenClaw spawns a full Claude Code agent inside a Docker container:

```
// Simplified agent invocation
const result = await runClaudeAgent({
  systemPrompt: loadClaudeMd(groupId),
  userMessage: message.content,
  tools: getActiveTools(groupId),
  containerConfig: {
    image: process.env.CONTAINER_IMAGE,
    memory: '512m',
    cpus: '0.5',
    volumeMounts: [
      { host: `./data/groups/${groupId}`, container: '/workspace' },
      { host: './ipc', container: '/ipc' }
    ]
  }
});
```

2.5 IPC via Filesystem

Containers communicate with the host process through a shared `/ipc/` folder. This is deliberately low-tech — no sockets, no HTTP — just files:

Agent → Host

- `/ipc/out/response.json` — Final response payload
- `/ipc/out/stream/chunk_001.txt` — Streaming chunks
- `/ipc/out/status.json` — Progress updates

Host → Agent

- `/ipc/in/context.json` — Initial context
- `/ipc/in/tools/` — Available tool definitions
- `/ipc/in/cancel` — Cancellation signal

Why Filesystem IPC?

Using files for IPC is extremely reliable, debuggable (you can read the files), and works across any container runtime — Docker, Podman, Apple Container, or even a plain subprocess. It also survives partial failures gracefully.

2.6 The SQLite Database Schema

OpenClaw stores all state in a single SQLite file. The main tables are:

```sql
-- Groups/chats
CREATE TABLE groups (
  id TEXT PRIMARY KEY,        -- e.g., "120363123456789@g.us"
  channel TEXT NOT NULL,      -- 'whatsapp' | 'telegram' | etc.
  name TEXT,                  -- Human-readable name
  claude_md TEXT,             -- Loaded CLAUDE.md content
  created_at INTEGER,
  updated_at INTEGER
);

-- Messages
CREATE TABLE messages (
  id TEXT PRIMARY KEY,
  group_id TEXT NOT NULL,
  sender_id TEXT NOT NULL,
  sender_name TEXT,
  role TEXT NOT NULL,         -- 'user' | 'assistant'
  content TEXT NOT NULL,
  timestamp INTEGER NOT NULL,
  FOREIGN KEY(group_id) REFERENCES groups(id)
);

-- Scheduled tasks
CREATE TABLE scheduled_tasks (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  group_id TEXT NOT NULL,
  cron_expression TEXT NOT NULL,  -- e.g., "0 9 * * 1-5"
  task_prompt TEXT NOT NULL,
  enabled INTEGER DEFAULT 1,
  last_run INTEGER,
  next_run INTEGER,
  created_at INTEGER
);
```

2.7 Configuration Architecture

OpenClaw has 53 configuration files organized in a clear hierarchy:

Config Level	Location	Purpose
Platform	`.env`	API keys, DB path, container limits
Platform	`config/defaults.ts`	Hard-coded platform defaults
Group	`data/groups/{id}/CLAUDE.md`	Per-group AI personality & memory
Group	`data/groups/{id}/skills/`	Active skills for this group
Agent	`container/CLAUDE.md`	Agent container system instructions

The Power of This Architecture

Each WhatsApp group, Telegram chat, or Discord channel can have a completely different AI personality, set of tools, and memory — all without touching code. Just edit the group's CLAUDE.md file.

03

Messaging Channels

Connecting WhatsApp, Telegram, Discord, and Slack — configuration, authentication, and advanced features

Chapter 03 — Messaging Channels

3.1 Channel Overview

OpenClaw's channel system is built on the adapter pattern — each platform is implemented as an independent module that translates platform-specific events into OpenClaw's unified message format. This means adding a new channel doesn't require touching the core logic.

Channel	Auth Method	Group Support	File Upload	Status
WhatsApp	QR Code scan	✅ Full	✅ Images, docs, audio	Stable
Telegram	Bot token (@BotFather)	✅ Full	✅ All types	Stable
Discord	Bot token + OAuth2	✅ Per guild	✅ Attachments	Stable
Slack	OAuth + Socket Mode	✅ Channels	✅ Files	Beta

3.2 WhatsApp Deep Dive

WhatsApp is OpenClaw's primary channel, using `whatsapp-web.js` to run a Chromium-based WhatsApp Web session.

Authentication & Session

```
// src/channels/whatsapp.ts (simplified)
import { Client, LocalAuth } from 'whatsapp-web.js';

export const whatsappClient = new Client({
  authStrategy: new LocalAuth({ dataPath: './data/wwebjs_auth' }),
  puppeteer: {
    headless: true,
    args: ['--no-sandbox', '--disable-setuid-sandbox']
  }
});

whatsappClient.on('qr', (qr) => {
  qrcode.generate(qr, { small: true });
  console.log('📱 Scan this QR code with WhatsApp');
});
```

```
whatsappClient.on('ready', () => {
  console.log('✅ WhatsApp client ready');
});
```

Handling Group Messages

```
whatsappClient.on('message_create', async (msg) => {
  if (!msg.fromMe && msg.from.endsWith('@g.us')) {
    // It's a group message
    const chat = await msg.getChat();
    const contact = await msg.getContact();

    const normalized: IncomingMessage = {
      id: msg.id._serialized,
      groupId: msg.from,
      senderId: contact.id._serialized,
      senderName: contact.pushname || contact.name || 'Unknown',
      content: msg.body,
      channel: 'whatsapp',
      timestamp: new Date(msg.timestamp * 1000)
    };

    await orchestrator.handle(normalized);
  }
});
```

WhatsApp ToS Note

Using unofficial WhatsApp APIs may violate WhatsApp's Terms of Service. For production commercial use, consider the official WhatsApp Business API (Cloud API). The unofficial library is excellent for personal use, internal tools, and development.

3.3 Telegram Integration

Telegram is the most developer-friendly channel. Creating a bot takes 30 seconds via @BotFather.

Creating Your Bot

1

Open Telegram and search for **@BotFather**

2

Send `/newbot` and follow the prompts

3

Copy the bot token (format: `123456:ABC-DEF1234ghIkl-zyx57W2v1u123ew11`)

4

Add the token to your `.env` as `TELEGRAM_BOT_TOKEN`

5

Add the bot to your group and give it admin permissions to read messages

Telegram-Specific Features

```
// src/channels/telegram.ts (simplified)
import TelegramBot from 'node-telegram-bot-api';

const bot = new TelegramBot(process.env.TELEGRAM_BOT_TOKEN!, {
  polling: true
});

bot.on('message', async (msg) => {
  if (!msg.text || !msg.chat.id) return;

  // Send "typing..." indicator while processing
  await bot.sendChatAction(msg.chat.id, 'typing');

  const normalized: IncomingMessage = {
    id: msg.message_id.toString(),
    groupId: msg.chat.id.toString(),
    senderId: msg.from!.id.toString(),
    senderName: msg.from!.first_name + (msg.from!.last_name ? ` ${msg.from!.last_r
    content: msg.text,
    channel: 'telegram',
    timestamp: new Date(msg.date * 1000)
  };

  const response = await orchestrator.handle(normalized);

  // Telegram supports Markdown formatting
  await bot.sendMessage(msg.chat.id, response, {
    parse_mode: 'Markdown',
    reply_to_message_id: msg.message_id
  });
});
```

3.4 Discord Integration

Discord uses a Gateway WebSocket connection. OpenClaw listens for `messageCreate` events across all guilds.

```
// src/channels/discord.ts
import { Client, GatewayIntentBits } from 'discord.js';

const client = new Client({
  intents: [
    GatewayIntentBits.Guilds,
    GatewayIntentBits.GuildMessages,
    GatewayIntentBits.MessageContent  // Requires privileged intent in Discord Por
  ]
});

client.on('messageCreate', async (message) => {
  if (message.author.bot) return; // Ignore other bots

  const normalized: IncomingMessage = {
    id: message.id,
    groupId: `${message.guildId}:${message.channelId}`,
    senderId: message.author.id,
    senderName: message.member?.displayName || message.author.username,
    content: message.content,
    channel: 'discord',
    timestamp: message.createdAt
  };

  // Discord's typing indicator
  await message.channel.sendTyping();
  const response = await orchestrator.handle(normalized);
  await message.reply(response);
});
```

3.5 Slack Integration

Slack uses the Bolt framework with Socket Mode, which doesn't require a public webhook URL — perfect for development and firewalled environments.

```
// src/channels/slack.ts
import { App } from '@slack/bolt';

const slackApp = new App({
  token: process.env.SLACK_BOT_TOKEN,
  appToken: process.env.SLACK_APP_TOKEN,
  signingSecret: process.env.SLACK_SIGNING_SECRET,
  socketMode: true  // No public URL needed
});

slackApp.message(async ({ message, say, client }) => {
  const msg = message as any;
  if (!msg.text || msg.subtype) return;

  // Post a "thinking..." message first, then update it
  const thinking = await say('🤔 Thinking...');

  const normalized: IncomingMessage = {
    id: msg.ts,
    groupId: msg.channel,
    senderId: msg.user,
    senderName: await getSlackUserName(client, msg.user),
    content: msg.text,
    channel: 'slack',
    timestamp: new Date(parseFloat(msg.ts) * 1000)
  };

  const response = await orchestrator.handle(normalized);

  // Update the "thinking" message with the actual response
  await client.chat.update({
    channel: msg.channel,
    ts: thinking.ts as string,
    text: response
  });
});
```

3.6 Multi-Channel Groups

One powerful OpenClaw feature is the ability to link channels together. A WhatsApp group and a Telegram group can share the same `CLAUDE.md` context, creating a unified AI persona across platforms:

```
// In data/groups/config.json
{
  "linked_groups": [
    {
      "name": "Marketing Team",
      "groups": [
        "120363123456789@g.us",    // WhatsApp group
        "-1001234567890",           // Telegram group
        "1234567890:987654321"      // Discord guild:channel
      ],
      "shared_claude_md": "marketing-team"
    }
  ]
}
```

Context Isolation

Even when groups are linked for shared CLAUDE.md, message history remains per-channel by default. Users on WhatsApp won't see Telegram messages in the AI's context unless you explicitly enable cross-channel history merging.

3.7 Handling Media & Attachments

OpenClaw can process images, documents, and voice messages by passing them as base64-encoded content to Claude's vision API:

```
// Images are converted to base64 and included as message content
const imageContent = {
  type: 'image',
  source: {
    type: 'base64',
    media_type: 'image/jpeg',
    data: imageBase64
    }
  };
```

\n\n

04

Container Security

OS-level isolation, resource limits, and building a zero-trust agent environment

37

Chapter 04 — Container Security

4.1 The Security Problem

An AI agent with tool access is inherently powerful — and potentially dangerous. Without proper sandboxing, a single confused or manipulated agent could:

- Read or delete arbitrary files on your server
- Make outbound network calls to attacker-controlled servers
- Exhaust CPU/RAM and crash the host process
- Access credentials from environment variables or config files
- Escalate privileges through OS vulnerabilities

OpenClaw solves this through **OS-level container isolation** — every agent runs in a fresh, disposable container with strict resource and filesystem boundaries.

4.2 Container Isolation Model

Each agent execution follows this lifecycle:

1

Spawn — A new container is created from the base image with a read-only root filesystem

2

Mount — Only two volumes are mounted: `/workspace` (group data, read-write) and `/ipc` (communication pipe)

3

Execute — The agent runs with a non-root user, no network access by default, CPU and RAM caps enforced

4

Communicate — Results are written to `/ipc/out/` and read by the host

5

Destroy — Container is forcefully terminated and removed after completion or timeout

```
# Docker run equivalent of what OpenClaw does internally
docker run \
  --rm \
  --read-only \
  --user 1000:1000 \
  --memory 512m \
  --cpus 0.5 \
  --network none \
  --volume ./data/groups/GROUP_ID:/workspace:rw \
  --volume ./ipc:/ipc:rw \
  --env ANTHROPIC_API_KEY=$ANTHROPIC_API_KEY \
  openclaw-agent:latest
```

Critical: Never Skip Isolation

Running agents directly on the host process (without containers) is dangerous in production. Only do this in development with a full understanding of the risk. The NanoClaw architecture is designed for this use case with additional safeguards.

4.3 Resource Limits

Container resource limits prevent runaway agents from affecting system stability:

Resource	Default Limit	Configurable?
RAM	512 MB	Yes — `AGENT_MEMORY_LIMIT`
CPU	0.5 cores	Yes — `AGENT_CPU_LIMIT`
Execution timeout	5 minutes	Yes — `AGENT_TIMEOUT_SECONDS`
Max concurrent agents	5	Yes — `MAX_CONTAINERS`
Network access	None (--network none)	Configurable per group
Disk write	/workspace only	No

4.4 Apple Container Support

On Apple Silicon Macs, OpenClaw supports Apple's native container runtime as an alternative to Docker. Apple Container uses lightweight Linux VMs (via Virtualization.framework) with near-native performance:

```
# Using Apple Container instead of Docker
# In .env:
CONTAINER_RUNTIME=apple  # or 'docker' (default)

# Apple Container commands (similar to docker)
container run \
  --memory 512m \
  openclaw-agent:latest
```

Apple Container vs Docker

Apple Container provides true hardware-level isolation with Linux VMs, while Docker on Mac uses a shared Linux VM. For maximum security on Apple Silicon, Apple Container is preferred. For cross-platform compatibility, Docker is the default.

4.5 Secret Management

API keys and secrets must never reach agent containers directly. OpenClaw uses a secret injection pattern:

```
// The API key is injected at container creation time
// and removed from container environment after initialization
const containerEnv = {
  ANTHROPIC_API_KEY: process.env.ANTHROPIC_API_KEY,
  // Other secrets are NOT passed to agents
};

// Agents access the key through the Claude SDK
// which reads ANTHROPIC_API_KEY automatically
```

4.6 Prompt Injection Defenses

Users may attempt to manipulate the AI through crafted messages ("Ignore all previous instructions..."). OpenClaw's defenses include:

- **System prompt precedence** — CLAUDE.md instructions loaded as the system prompt, which has higher priority than user messages in Claude's architecture
- **Input sanitization** — Special tokens and control characters stripped from user input
- **Response validation** — Agent outputs validated against expected schemas before delivery
- **Rate limiting** — Per-user message rate limits prevent flooding attacks

05

Memory & Context

CLAUDE.md, conversation history, and building AI that truly remembers

40

Chapter 05 — Memory & Context

5.1 The Memory Problem in LLMs

Large language models are stateless — they have no inherent memory between requests. Every conversation starts fresh unless you explicitly include prior context. OpenClaw solves this through three complementary memory mechanisms:

1. Conversation History

The last N messages from the SQLite database, loaded into the messages array for every API call. Short-term working memory.

2. CLAUDE.md System Prompt

A Markdown file loaded as the system prompt. Persistent background knowledge, personality, and standing instructions. Long-term declarative memory.

3. Agent Workspace Files

Files in `/workspace/` that agents can read and write. External memory for complex, structured information that exceeds context limits.

4. Skills

SKILL.md files that inject specialized instructions for specific tasks. Task-specific procedural memory that activates on demand.

5.2 CLAUDE.md — The Memory File

The `CLAUDE.md` file is the most powerful memory mechanism. It is loaded as the system prompt for every conversation in its group, giving the AI persistent context, personality, and instructions.

File Location

```
data/
  groups/
    120363123456789@g.us/    # WhatsApp group ID
      CLAUDE.md              # This group's memory file
      skills/                # Active skills for this group
    -1001234567890/          # Telegram group ID
      CLAUDE.md
  global/
    CLAUDE.md                # Default for groups without their own
```

Example CLAUDE.md for a Marketing Team

```
# Marketing Team Assistant

## Identity
You are Maya, the AI marketing assistant for PixMind Studio.
You have a professional but energetic tone.
Always respond in the language the user writes to you in.

## Company Context
- Company: PixMind Studio
- Services: Video production, ad campaigns, social media content
- Key clients: [confidential — see workspace/clients.md]
- Brand colors: #6366f1 (primary), #06b6d4 (accent)

## Standing Instructions
- Always sign off with "— Maya, PixMind AI"
- When asked about pricing, refer to workspace/pricing.md
- Never discuss competitor pricing
- If asked for campaign ideas, generate 3 options with different angles

## Current Priorities
- Q2 2026 campaign launch: March 30 deadline
- New client onboarding: TechCorp (fintech vertical)
```

5.3 Conversation History Management

OpenClaw maintains a rolling window of conversation history to stay within Claude's context limits while preserving meaningful context:

```
// src/context-builder.ts
export async function buildContext(groupId: string, newMessage: string) {
  // Load last 50 messages from SQLite
  const history = await db.getMessages(groupId, { limit: 50 });

  // Estimate token count
  let tokenCount = estimateTokens(history);

  // Trim oldest messages if approaching limit
  while (tokenCount > MAX_CONTEXT_TOKENS && history.length > 5) {
    history.shift(); // Remove oldest
    tokenCount = estimateTokens(history);
  }

  // Load system prompt from CLAUDE.md
  const systemPrompt = await loadClaudeMd(groupId);

  return {
    system: systemPrompt,
    messages: [
      ...history.map(m => ({ role: m.role, content: m.content })),
      { role: 'user', content: newMessage }
    ]
  };
}
```

5.4 Dynamic Memory Updates

Agents can update the group's CLAUDE.md file during a session, enabling persistent learning:

```
# In a CLAUDE.md file, an agent might add:
## Learned Preferences (auto-updated)
- User @ariel prefers bullet points over prose
- User @ariel's timezone: UTC+2 (Israel)
- Team meeting: Mondays at 10am
- Preferred response language: Hebrew for casual, English for technical
```

Memory as a First-Class Feature

Teach your users to tell the bot "remember that..." and have the agent append it to CLAUDE.md. This creates a genuinely intelligent assistant that improves with every interaction — without any vector database, embeddings, or complex infrastructure.

5.5 Workspace Files as External Memory

When a group's `CLAUDE.md` references workspace files, agents can read them to access structured data that would be too large for the system prompt:

```
# In CLAUDE.md:
## Data Sources
When asked about clients, read /workspace/clients.json
When asked about products, read /workspace/catalog.md
When generating reports, write results to /workspace/reports/
// clients.json (in workspace)
{
  "clients": [
    { "id": "techcorp", "name": "TechCorp", "contact": "Sarah Chen",
      "vertical": "fintech", "mrr": 15000, "since": "2025-01" },
    { "id": "fashionx", "name": "FashionX", "contact": "Alex Kim",
      "vertical": "ecommerce", "mrr": 8500, "since": "2024-06" }
  ]
}
```

06

The Skills System

Installable AI behaviors — the plugin system that makes OpenClaw infinitely extensible

Chapter 06 — The Skills System

6.1 Skills vs Features

The Skills system is one of OpenClaw's most innovative contributions to the AI assistant ecosystem. Instead of hardcoding features into the platform, OpenClaw allows you to install "skills" — Markdown files that inject specialized behaviors into the AI agent on demand.

The key insight: **skills are better than features**.

	Features (Traditional)	Skills (OpenClaw)
Delivery	Code deployment required	Drop a Markdown file
Customization	Fork the repo	Edit the SKILL.md
Scope	Platform-wide	Per-group
Rollback	Git revert + redeploy	Delete the file
Sharing	Pull request	Send the .md file
Versioning	Semantic versioning	Filename convention

6.2 Anatomy of a SKILL.md File

A skill is just a Markdown file placed in the group's `skills/` directory. When activated, its content is appended to the system prompt:

```
# SKILL: Content Writer
Version: 1.0
Trigger: When user asks to write, create, draft, or generate any content

## Behavior
You are now acting as an expert content writer with 10 years of experience
in digital marketing and brand storytelling.

## Writing Framework
When creating any content, follow this structure:
1. **Hook** — Capture attention in the first line (pain point, surprise, or questi
2. **Bridge** — Connect the hook to the solution
3. **Body** — Deliver value (3-5 key points with examples)
```

```
4. **CTA** — Clear, specific call to action

## Style Rules
- Write at 8th grade reading level for broad appeal
- Use active voice (90%+ of sentences)
- Short paragraphs (max 3 sentences)
- Vary sentence length for rhythm
- No filler phrases ("In conclusion...", "It's worth noting...")

## Output Formats Available
- Social media post (request: "write a post for [platform]")
- Blog article (request: "write a blog about X")
- Email campaign (request: "write email for [purpose]")
- Ad copy (request: "write an ad for [product/service]")
```

6.3 Installing a Skill

Installing a skill is as simple as placing the file in the right directory:

```
# Install a skill for a specific WhatsApp group
cp content-writer.md \
  data/groups/120363123456789@g.us/skills/content-writer.md

# The skill is active immediately — no restart required
# Test it:
# Send: "Write a LinkedIn post about AI in marketing"
```

OpenClaw watches the `skills/` directory with a filesystem watcher. New files are picked up automatically on the next message.

6.4 Skill Activation Triggers

Skills can be configured with different activation modes:

Always Active

No trigger — the skill content is always included in the system prompt. Best for persistent behavioral modifications.

```
Trigger: always
```

Keyword Trigger

Activated when the user's message matches keywords. Best for specialized tools.

```
Trigger: keywords: write, draft,
  create content, compose
```

Explicit Activation

User must explicitly activate with a command. Best for powerful or expensive skills.

```
Trigger: command: /analyst
```

Context Trigger

Activated when the conversation context matches a pattern.

```
Trigger: context: user is asking
  about pricing or costs
```

6.5 Real-World Skill Examples

Code Review Skill

```
# SKILL: Code Reviewer
Trigger: When user shares code or asks for review

## Review Checklist
For every code review, check and comment on:
- [ ] Security vulnerabilities (OWASP Top 10)
- [ ] Performance issues (N+1 queries, memory leaks)
- [ ] Code clarity and naming conventions
- [ ] Missing error handling
- [ ] Test coverage gaps

## Output Format
Use this exact format:
**Security:** [findings or "None found"]
**Performance:** [findings or "None found"]
**Clarity:** [suggestions]
**Missing:** [what should be added]
**Score:** X/10
```

Sales Assistant Skill

```
# SKILL: Sales Assistant
Trigger: When discussing deals, clients, or proposals

## Sales Framework
You are using the SPIN selling methodology:
- Situation questions to understand context
- Problem questions to uncover pain points
- Implication questions to expand the pain
- Need-payoff questions to build value

## Response Pattern
1. Acknowledge the prospect's situation
2. Ask one clarifying question per message
3. Never mention price before establishing value
4. Always offer next steps
```

6.6 The Skills Marketplace Concept

Because skills are just Markdown files, they can be shared, versioned, and distributed like open-source software. The OpenClaw community has developed a growing collection of skills:

Skill	Use Case	Source
content-writer	Social media, blogs, emails	Community
code-reviewer	PR reviews, security audits	Official
data-analyst	CSV/JSON analysis, charts	Official
translator	Multi-language support	Community
customer-support	Tier-1 support automation	Official
sales-coach	Deal coaching, objection handling	Community
meeting-summarizer	Transcripts to action items	Official
legal-reviewer	Contract red-lining (non-legal advice)	Community

Building Your Skill Library

Start by creating one skill for the most common request your team makes. After a week of use, refine it based on the outputs. A well-crafted SKILL.md can save hours every week and consistently outperform ad-hoc prompting.

6.7 Skill Composition

Multiple skills can be active simultaneously. OpenClaw merges them into the system prompt intelligently:

```
data/groups/my-team/skills/
  ├── always/
  │   ├── 00-base-personality.md     # Always loaded first
  │   └── 01-language-rules.md       # Always loaded second
  ├── on-demand/
  │   ├── content-writer.md
  │   ├── code-reviewer.md
  │   └── data-analyst.md
  └── temp/
      └── q2-campaign-brief.md       # Remove after Q2
```

The loading order ensures base personality is always present, while specialized skills layer on top without conflicting.

07

Scheduled Tasks

Cron-based autonomous AI tasks — agents that work while you sleep

47

Chapter 07 — Scheduled Tasks

7.1 What Are Scheduled Tasks?

Scheduled Tasks transform OpenClaw from a reactive chatbot into a proactive AI system. Instead of waiting for user messages, agents execute pre-defined tasks on a cron schedule — automatically sending reports, monitoring systems, running analyses, and more.

Proactive vs Reactive AI

Most AI assistants are purely reactive — they wait for you to ask a question. OpenClaw's Scheduled Tasks flip this model: the AI monitors, analyzes, and reports proactively. This is the difference between a tool and a team member.

7.2 Creating a Scheduled Task

Scheduled tasks are stored in SQLite and managed via API or direct database interaction:

```
// Create a task that sends a morning briefing every weekday
await db.createScheduledTask({
  groupId: '120363123456789@g.us',
  cronExpression: '0 8 * * 1-5',   // 8am Mon-Fri
  taskPrompt: `
    Good morning! Please generate today's briefing:
    1. Check workspace/tasks.md for today's priorities
    2. Summarize any pending items from workspace/inbox.md
    3. Check if today has any client deadlines (workspace/calendar.md)
    4. Provide 3 focus recommendations for maximum productivity
    Format as a clean morning briefing message.
  `,
  enabled: true
});
```

7.3 Cron Expression Reference

Expression	Meaning
`0 8 * * 1-5`	8:00 AM, Monday to Friday
`0 9 * * 1`	9:00 AM, every Monday only

`0 */2 * * *`	Every 2 hours
`*/30 * * * *`	Every 30 minutes
`0 18 * * 5`	6:00 PM every Friday
`0 0 1 * *`	Midnight on the 1st of each month
`0 9 * * 0`	9:00 AM every Sunday

7.4 Task Execution Engine

```
// src/scheduler.ts
import cron from 'node-cron';

export class TaskScheduler {
  private jobs = new Map<number, cron.ScheduledTask>();

  async loadAndScheduleAll() {
    const tasks = await db.getEnabledTasks();
    for (const task of tasks) {
      this.schedule(task);
    }
  }

  schedule(task: ScheduledTask) {
    const job = cron.schedule(task.cronExpression, async () => {
      console.log(`⏰ Running scheduled task: ${task.id}`);
      try {
        const result = await agentRunner.run({
          groupId: task.groupId,
          prompt: task.taskPrompt,
          source: 'scheduler'
        });
        await channelRouter.send(task.groupId, result);
        await db.updateTaskLastRun(task.id);
      } catch (err) {
        console.error(`Task ${task.id} failed:`, err);
      }
    });
    this.jobs.set(task.id, job);
    }
}
```

7.5 Real-World Scheduled Task Recipes

Daily Sales Report

```
cronExpression: "0 17 * * 1-5"  // 5pm weekdays
taskPrompt: |
  Generate today's sales report:
  1. Read workspace/crm-data.json
  2. Calculate: new leads, demos booked, deals closed, revenue
  3. Compare to yesterday (workspace/reports/yesterday.json)
  4. Highlight top performer and any at-risk deals
  5. Save report to workspace/reports/today.json
  6. Send formatted summary to the team
```

Weekly Team Newsletter

```
cronExpression: "0 9 * * 1"  // Monday 9am
taskPrompt: |
  Create this week's team newsletter:
  - Check workspace/wins.md for recent victories to celebrate
  - List upcoming deadlines from workspace/calendar.md
  - Pull random tip from workspace/tips-database.md
  - Include motivational opening paragraph
  Format in WhatsApp-compatible text with emojis
```

System Health Monitor

```
cronExpression: "*/15 * * * *"  // Every 15 minutes
taskPrompt: |
  Check system health:
  - Read /workspace/metrics/current.json
  - If any metric exceeds threshold (CPU>80%, RAM>90%, errors>10/min):
    Send immediate alert with: metric name, current value,
    threshold, and recommended action
  - Otherwise: update workspace/metrics/last-check.json silently
  Only send messages if there's an alert - stay quiet otherwise
```

7.6 Task Management API

Manage tasks programmatically via the built-in REST API:

```
# List all tasks for a group
GET /api/tasks?groupId=120363...

# Create a new task
POST /api/tasks
{
  "groupId": "120363...",
  "cronExpression": "0 9 * * 1-5",
  "taskPrompt": "...",
  "enabled": true
}

# Toggle a task on/off
PATCH /api/tasks/:id
{ "enabled": false }

# Manually trigger a task (for testing)
POST /api/tasks/:id/run

# Delete a task
DELETE /api/tasks/:id
```

Test Before Scheduling

Always use the manual trigger (`POST /api/tasks/:id/run`) to test a task before enabling the cron schedule. This lets you verify the output format and catch errors without waiting for the next scheduled run.

08

Agent Swarms

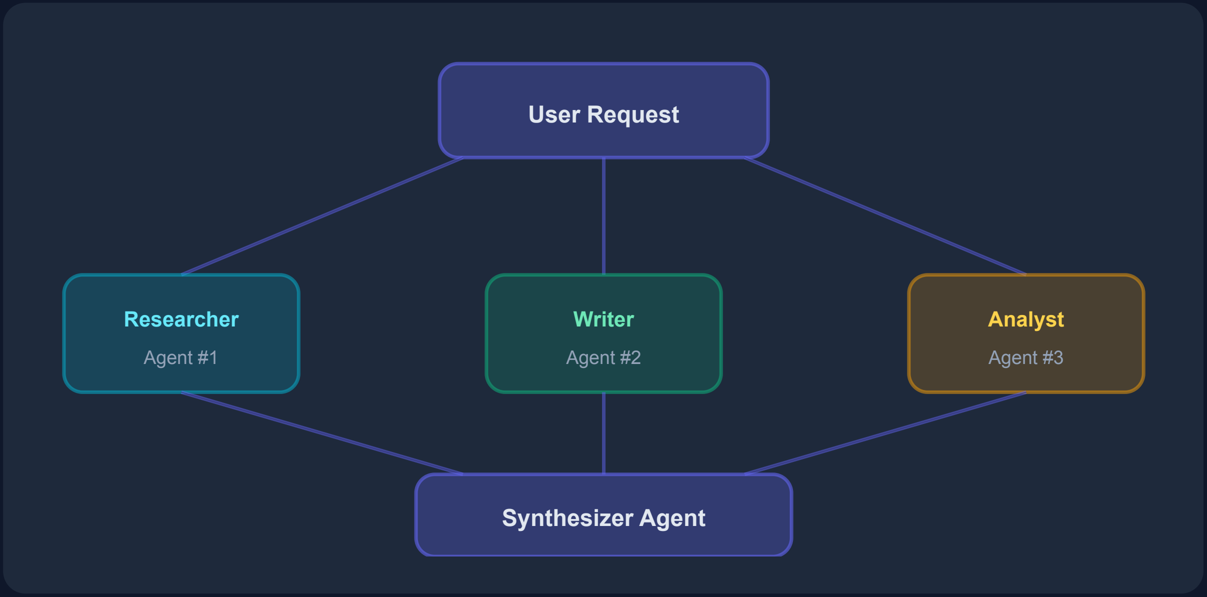
The world's first multi-agent messaging platform — parallel AI collaboration at scale

Chapter 08 — Agent Swarms

8.1 The Swarm Concept

OpenClaw was the first messaging AI platform to support **Agent Swarms** — multiple specialized AI agents working in parallel on a single complex task, then synthesizing their results into one coherent response.

Think of it like this: instead of asking one generalist to research a topic, write a report, and design visualizations — you assign three specialists simultaneously and merge their work.



8.2 When to Use Swarms

Swarms are powerful but have overhead (multiple API calls, container spawns). Use them when:

- The task has clearly separable subtasks that can run in parallel
- Speed matters and individual tasks take more than 30 seconds each
- You need multiple perspectives or approaches synthesized together
- Tasks require different specialized skills (research + writing + code)

For simple questions and quick tasks, single-agent mode is faster and cheaper.

8.3 Swarm Configuration

```
// Define a swarm for creating comprehensive market reports
const swarmConfig: SwarmConfig = {
  name: 'market-report',
  trigger: 'when user asks for market research or competitive analysis',

  agents: [
    {
      id: 'researcher',
      role: 'Market Researcher',
      systemPrompt: 'You are a thorough market researcher...',
      task: 'Research the market landscape for {topic}. ' +
            'Focus on market size, growth trends, key players. ' +
            'Return structured JSON.',
      tools: ['web_search', 'read_file']
    },
    {
      id: 'competitor-analyst',
      role: 'Competitive Intelligence',
      systemPrompt: 'You are a competitive intelligence specialist...',
      task: 'Analyze the top 5 competitors for {topic}. ' +
            'Strengths, weaknesses, pricing, positioning.',
      tools: ['web_search']
    },
    {
      id: 'strategist',
      role: 'Strategy Advisor',
      systemPrompt: 'You are a strategic business advisor...',
      task: 'Based on the market context, identify 3 strategic ' +
            'opportunities and 3 key risks for {topic}.',
      tools: []
    }
  ],

  synthesizer: {
    prompt: 'Combine the research, competitor analysis, and strategic ' +
            'insights into a professional market report with executive summary.',
    outputFormat: 'markdown'
  },

  parallelism: 3,  // Run all 3 agents simultaneously
  timeout: 300     // 5 minute max
};
```

8.4 Swarm Execution Flow

1

Parse request — Extract variables (like `{topic}`) from the user message

2

Spawn agents — Launch all agent containers in parallel (up to `MAX_CONTAINERS`)

3

Execute — Each agent runs independently, reads tools, writes results to `/ipc/out/agent-{id}.json`

4

Wait — Host process waits for all agents to complete (or timeout)

5

Synthesize — All results are passed to the synthesizer agent, which produces the final response

6

Deliver — Final response sent to the user, all containers destroyed

Cost Awareness

A 3-agent swarm costs approximately 3x the API tokens of a single-agent response. For a synthesizer swarm, add another 1x on top. Budget accordingly, especially for scheduled tasks that run swarms automatically.

\n\n

09

Customization

Tailoring OpenClaw's personality, behavior, and capabilities for every use case

53

Chapter 09 — Customization

9.1 The Customization Philosophy

OpenClaw is designed to be a *platform*, not a product. Everything from the AI's name and personality, to the tools it can use, to the language it responds in — all is configurable without touching a single line of TypeScript.

The customization stack, from broadest to most specific:

1. **Platform defaults** (`.env` + `config/defaults.ts`) — apply to all groups
2. **Global CLAUDE.md** (`data/global/CLAUDE.md`) — applies to groups without their own
3. **Group CLAUDE.md** (`data/groups/{id}/CLAUDE.md`) — overrides for this group
4. **Skills** (`data/groups/{id}/skills/*.md`) — task-specific behaviors layered on top

9.2 Personality Engineering

The most impactful customization is defining the AI's persona. A well-crafted persona makes users feel like they're talking to a team member, not a generic chatbot:

```
# data/groups/my-startup/CLAUDE.md

## Identity
You are **Pixie**, the AI assistant for PixiBot startup.
Your personality: enthusiastic, technically sharp, occasionally witty.
You speak casually in 1:1 chats, professionally in client channels.

## Communication Rules
- Keep responses concise (under 200 words unless asked for detail)
- Use bullet points for lists longer than 3 items
- React with appropriate emojis when celebrating wins 🎉
- Never apologize excessively — say it once and move on
- When you don't know something, say "I'll look into that" not "I'm sorry"

## Language
- Default to the language the user writes in
- Hebrew and English both feel natural to you
- Technical terms stay in English even in Hebrew conversations
```

```
## Expertise Areas
- You know the PixiBot codebase well (see workspace/architecture.md)
- You're familiar with startup culture and fundraising
- You understand both technical and business contexts
```

9.3 Response Format Customization

Different channels have different formatting capabilities. Customize output format per channel:

```
## Channel Formatting Rules

### WhatsApp
- Use *bold* for emphasis (WhatsApp markdown)
- Use _italic_ for secondary info
- Maximum 5 bullet points per response
- No tables (not supported in WhatsApp)
- Split very long responses with [...continued]

### Discord
- Use **bold** and `code blocks` freely
- Tables are supported and encouraged for comparisons
- Use >>> for quotes/references
- Pin important messages with [📌 PIN THIS]

### Telegram
- Use **Markdown bold** and `code` formatting
- Inline code for commands, pre blocks for multi-line code
- Support up to 4096 chars per message
```

9.4 Tool Access Control

Control which tools are available per group to enforce security boundaries:

```
// data/groups/my-group/config.json
{
  "tools": {
    "allowed": [
      "read_file",
      "write_file",
      "list_directory",
      "web_search"
    ],
    "denied": [
      "execute_command",   // No shell execution for this group
      "delete_file"        // No deletions
    ],
    "limits": {
      "web_search": { "max_per_session": 10 },
      "write_file": { "max_file_size_kb": 500 }
    }
  }
}
```

9.5 Custom Commands

Define slash commands that trigger specific behaviors:

```
# In CLAUDE.md:
## Custom Commands

/report — Generate this week's performance summary
/brief — Create today's morning briefing
/ideas [topic] — Generate 5 creative ideas about the given topic
/translate [lang] [text] — Translate text to the specified language
/summarize — Summarize the last 20 messages in this chat
/status — Show system status and active tasks
```

9.6 Multi-Language Support

OpenClaw supports full multi-language operation. Configure language behavior in CLAUDE.md:

```
## Language Rules
- Always respond in the SAME language the user writes in
- If the message is mixed Hebrew/English, respond in Hebrew
- Never switch languages mid-response
- Hebrew responses: use RTL-friendly formatting (no ASCII tables)
- English responses: full formatting supported

## Fallback Language
If language detection is uncertain: respond in English
User can override: "ענה לי בעברית" / "Please respond in English"
```

Business Tip

For customer-facing deployments, consider creating separate groups per language (one for Hebrew speakers, one for English) each with their own CLAUDE.md. This gives you full control over tone, phrasing, and cultural context per language.

10

MCP Integration

Model Context Protocol — connecting any external service to your AI agent

56

Chapter 10 — MCP Integration

10.1 What is MCP?

The **Model Context Protocol (MCP)** is Anthropic's open standard for connecting AI agents to external tools and data sources. Instead of hardcoding integrations, MCP lets you plug in any tool via a standardized JSON-RPC interface.

MCP servers expose **tools**, **resources**, and **prompts**:

- **Tools** — Functions the AI can call (e.g., `search_database`, `send_email`)
- **Resources** — Data the AI can read (e.g., file contents, API responses)
- **Prompts** — Pre-built prompt templates for common tasks

10.2 Built-In MCP Servers

OpenClaw ships with MCP integrations for the most common developer tools:

MCP Server	Capabilities	Auth Required
GitHub	Read/create issues, PRs, files	Personal Access Token
Supabase	SQL queries, schema management	Project URL + Key
Airtable	CRUD on bases and records	API Key
Notion	Pages, databases, blocks	Integration Token
Slack	Messages, channels, users	OAuth Token
Google Drive	Files, docs, sheets	OAuth2
Browser	Web scraping, screenshots	None
Filesystem	Read/write local files	Path permissions

10.3 Configuring MCP Servers

```
// data/groups/dev-team/mcp.json
{
  "mcpServers": {
    "github": {
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-github"],
      "env": {
        "GITHUB_PERSONAL_ACCESS_TOKEN": "${GITHUB_TOKEN}"
      }
    },
    "supabase": {
      "command": "npx",
      "args": ["-y", "@supabase/mcp-server-supabase@latest"],
      "env": {
        "SUPABASE_URL": "${SUPABASE_URL}",
        "SUPABASE_SERVICE_ROLE_KEY": "${SUPABASE_KEY}"
      }
    },
    "filesystem": {
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-filesystem",
               "/workspace/projects"]
    }
  }
}
```

10.4 Building a Custom MCP Server

Any internal API or database can be exposed to OpenClaw agents via a custom MCP server. Here's a minimal example using the TypeScript MCP SDK:

```
// mcp-servers/crm-server/index.ts
import { Server } from '@modelcontextprotocol/sdk/server/index.js';
import { StdioServerTransport } from '@modelcontextprotocol/sdk/server/stdio.js';

const server = new Server(
  { name: 'pixibot-crm', version: '1.0.0' },
  { capabilities: { tools: {} } }
);

// Define tools
server.setRequestHandler(ListToolsRequestSchema, async () => ({
  tools: [
    {
      name: 'get_client',
      description: 'Get client information by ID or name',
      inputSchema: {
        type: 'object',
        properties: {
          query: { type: 'string', description: 'Client name or ID' }
        },
        required: ['query']
      }
    },
    {
      name: 'create_deal',
      description: 'Create a new deal in the CRM',
      inputSchema: {
        type: 'object',
        properties: {
          clientId: { type: 'string' },
          value: { type: 'number' },
          stage: { type: 'string', enum: ['lead', 'proposal', 'negotiation', 'clos
        },
        required: ['clientId', 'value', 'stage']
      }
    }
  ]
}));

// Handle tool calls
server.setRequestHandler(CallToolRequestSchema, async (request) => {
  if (request.params.name === 'get_client') {
    const client = await crmDb.findClient(request.params.arguments.query);
    return { content: [{ type: 'text', text: JSON.stringify(client) }] };
  }
  // ... handle other tools
});

const transport = new StdioServerTransport();
await server.connect(transport);
```

10.5 MCP Tools in Agent Conversations

Once configured, MCP tools appear automatically in the agent's tool list. Users interact with them through natural language:

```
User: "What's the status of the TechCorp deal?"

Agent: [calls get_client("TechCorp")]
Agent: [calls get_deals({clientId: "techcorp"})]

Response: "TechCorp deal update:
  - Stage: Negotiation (since March 15)
  - Value: $45,000/year
  - Next step: Contract review scheduled March 28
  - Contact: Sarah Chen (sarah@techcorp.com)
  - Last activity: Demo call — positive feedback on pricing"
```

MCP Server Reuse

MCP servers are reusable across groups. Define them once in `config/mcp-servers.json` and reference by name in each group's config. This way, your CRM integration is available to the sales team group, support team group, and management group — each with appropriate access levels.

11

NanoClaw

The lightweight single-process alternative — full power, zero overhead

Chapter 11 — NanoClaw: The Lightweight Alternative

11.1 Why NanoClaw Exists

OpenClaw is powerful but requires Docker, substantial RAM, and a beefy server. Many users — developers, freelancers, small teams — don't need multi-container agent swarms. They need a simple, reliable AI assistant that just works.

NanoClaw strips OpenClaw down to its essentials: a single Node.js process, no Docker dependency, and the same CLAUDE.md + Skills + Scheduled Tasks system — but with a fraction of the complexity.

OpenClaw

- Multi-process architecture
- Docker container isolation
- ~500K lines of code
- Agent Swarms support
- Complex MCP server mesh
- Horizontal scaling ready
- Requires 4GB+ RAM

NanoClaw

- Single Node.js process
- No Docker required
- ~15K lines of code
- Single-agent only
- Simple tool integrations
- Single machine only
- Runs on 512MB RAM

11.2 NanoClaw Architecture

The NanoClaw process model is dramatically simpler:

```
// NanoClaw: everything in one process
src/
  index.ts          — Entry point: starts all channels
  channels/
    whatsapp.ts     — WhatsApp client
    telegram.ts     — Telegram bot
  core/
    orchestrator.ts — Message routing
    claude.ts       — Direct Claude API calls (no containers)
    memory.ts       — SQLite history + CLAUDE.md loader
    scheduler.ts    — Cron task runner
  tools/
    filesystem.ts   — Read/write files
    search.ts       — Web search
    calculator.ts   — Math operations
```

11.3 NanoClaw Quick Start

```
git clone https://github.com/your-org/nanoclaw.git
cd nanoclaw
npm install
cp .env.example .env
# Fill in ANTHROPIC_API_KEY and channel tokens
npm run dev   # Start with hot reload
```

Upgrade Path

NanoClaw is designed to be a stepping stone to OpenClaw. All CLAUDE.md files, Skills, and Scheduled Tasks are format-compatible between the two systems. When you outgrow NanoClaw, migration to OpenClaw is straightforward — just copy your `data/` folder.

12

Production Deployment

Docker Compose, PM2, monitoring, backups, and high-availability patterns

61

Chapter 12 — Production Deployment

12.1 Production Checklist

Before going live, verify all items in this checklist:

1. **Environment** — All secrets in `.env`, never hardcoded. Rotate API keys.
2. **Build** — Run `npm run build` and test the compiled `dist/index.js`
3. **Database backup** — Automated SQLite backups to cloud storage (S3, Backblaze)
4. **WhatsApp session** — Auth session mounted as persistent Docker volume
5. **Process manager** — PM2 or systemd for automatic restarts
6. **Monitoring** — Health checks, error alerts, uptime tracking
7. **Rate limits** — Per-user and per-group message rate limits configured
8. **Log rotation** — Logs capped and rotated to prevent disk fill

12.2 Docker Compose Setup

```
# docker-compose.yml
version: '3.8'

services:
  openclaw:
    build: .
    restart: unless-stopped
    environment:
      - NODE_ENV=production
    env_file: .env
    volumes:
      - ./data:/app/data           # Database + group files
      - ./data/wwebjs_auth:/app/data/wwebjs_auth  # WhatsApp session
      - /var/run/docker.sock:/var/run/docker.sock # For spawning agent containers
    ports:
      - "3000:3000"               # API port (optional)
    logging:
      driver: "json-file"
      options:
        max-size: "100m"
        max-file: "5"
    healthcheck:
      test: ["CMD", "node", "healthcheck.js"]
      interval: 30s
      timeout: 10s
      retries: 3

  # SQLite backup service
  sqlite-backup:
    image: alpine
    volumes:
      - ./data:/data:ro
      - ./backups:/backups
    entrypoint: |
      sh -c "while true; do
        cp /data/openclaw.db /backups/openclaw-$(date +%Y%m%d-%H%M).db
        find /backups -mtime +7 -delete  # Keep 7 days
        sleep 3600  # Backup every hour
      done"
```

12.3 PM2 Configuration

```
# ecosystem.config.js
module.exports = {
  apps: [{
    name: 'openclaw',
    script: 'dist/index.js',
    instances: 1,          // Single instance (stateful)
    autorestart: true,
    watch: false,
    max_memory_restart: '1G',
    env: {
      NODE_ENV: 'production'
    },
    log_file: './logs/combined.log',
    out_file: './logs/out.log',
    error_file: './logs/error.log',
    log_date_format: 'YYYY-MM-DD HH:mm:ss',
    merge_logs: true
  }]
};

# Start with PM2
pm2 start ecosystem.config.js
pm2 save           # Save process list
pm2 startup        # Auto-start on system boot
```

12.4 Health Monitoring

```
// healthcheck.js — run with: node healthcheck.js
const http = require('http');

const options = {
  hostname: 'localhost',
  port: 3000,
  path: '/health',
  timeout: 5000
};
```

```
const req = http.request(options, (res) => {
  if (res.statusCode === 200) {
    process.exit(0); // Healthy
  } else {
    process.exit(1); // Unhealthy
  }
});

req.on('error', () => process.exit(1));
req.end();
```

12.5 Scaling Considerations

Scale	Setup	Notes
1-10 users	Single VPS, NanoClaw	$5-10/month DigitalOcean
10-100 users	Single VPS, OpenClaw	$20-40/month, 2 vCPUs, 4GB RAM
100-1000 users	VPS + Redis for sessions	Separate DB server, 8GB RAM
1000+ users	Kubernetes / ECS	Multiple replicas, managed DB

Single Instance Only

OpenClaw maintains stateful WhatsApp and Discord WebSocket connections. Running multiple instances requires a distributed session store. For most use cases, a single well-resourced VPS is the right choice.

13

Troubleshooting

Diagnosing and fixing the most common issues in OpenClaw deployments

Chapter 13 — Troubleshooting Guide

13.1 WhatsApp Issues

Problem: QR code not appearing

```
Error: Failed to launch the browser process!
```

Cause: Chromium (Puppeteer) dependencies missing on Linux.

```
# Ubuntu/Debian fix
sudo apt-get install -y \
  libgbm-dev libnss3 libatk-bridge2.0-0 \
  libdrm2 libxkbcommon0 libgles2

# Or run with --no-sandbox flag (less secure)
# In .env: PUPPETEER_ARGS=--no-sandbox
```

Problem: WhatsApp session keeps logging out

Cause: WhatsApp Web sessions expire when used from multiple devices simultaneously, or if the phone app is inactive for extended periods.

Fix: Ensure the auth directory is persisted across restarts. In Docker, mount it as a named volume. Avoid opening WhatsApp Web in a browser at the same time as OpenClaw.

Problem: Messages not received from groups

Cause: WhatsApp groups require the bot account to be a participant, not just the group admin. Also, some WhatsApp versions require admin status to receive all messages.

13.2 Telegram Issues

Problem: Bot not responding in groups

Cause: Telegram bots need privacy mode disabled to receive all group messages (not just mentions). Go to BotFather → /setprivacy → your bot → Disable.

Problem: Polling conflict error

```
Error: 409 Conflict: terminated by other getUpdates request
```

Cause: Two instances of the bot are running simultaneously (common during hot-reload development).

```
# Kill all node processes and restart
pkill -f "node dist/index.js"
pkill -f "tsx src/index.ts"
npm run dev
```

13.3 Claude API Issues

Problem: Rate limit errors

```
Error: 429 Too Many Requests (rate_limit_error)
```

Implement exponential backoff. OpenClaw includes this by default — check `config/claude.ts` for `RETRY_ATTEMPTS` and `RETRY_DELAY_MS` settings.

Problem: Context length exceeded

```
Error: 400 Bad Request (invalid_request_error)
"prompt is too long: 205000 tokens > 200000 token limit"
```

Fix: Reduce `MAX_HISTORY_MESSAGES` in `.env`, or implement automatic summarization for long conversations.

13.4 Container Issues

Problem: Agent containers failing to start

```
Error: Cannot connect to Docker daemon
Error: permission denied while trying to connect to Docker socket
```

Fix: Add your user to the docker group:

```
sudo usermod -aG docker $USER
newgrp docker
# Verify:
docker ps  # Should work without sudo
```

Problem: Containers not cleaned up

Run this cleanup script to remove orphaned agent containers:

```
#!/bin/bash
# cleanup-agents.sh
docker ps -a --filter "name=openclaw-agent" --format "{{.ID}}" | \
  xargs -r docker rm -f
echo "Cleaned up orphaned agent containers"
```

13.5 Database Issues

Problem: SQLite "database is locked" error

```
Error: SQLITE_BUSY: database is locked
```

Cause: Multiple processes trying to write simultaneously. OpenClaw uses WAL (Write-Ahead Logging) mode by default to minimize this. Ensure WAL is enabled:

```
// src/db.ts — verify this is set
db.pragma('journal_mode = WAL');
db.pragma('busy_timeout = 5000');
```

13.6 Performance Diagnostics

Enable detailed performance logging to identify bottlenecks:

```
# In .env:
LOG_LEVEL=debug
LOG_PERFORMANCE=true
LOG_CLAUDE_TIMING=true
```

```
# Output will show:
# [PERF] context-build: 45ms
# [PERF] claude-api-call: 2340ms
# [PERF] response-delivery: 120ms
# [PERF] total: 2505ms
```

13.7 Debug Mode

```
# Full debug mode — logs everything
DEBUG=openclaw:* npm run dev

# Specific module debug
DEBUG=openclaw:whatsapp npm run dev
DEBUG=openclaw:orchestrator npm run dev
DEBUG=openclaw:claude npm run dev
```

The 80% Rule

80% of production issues are one of: environment variables not set, Docker socket permissions, WhatsApp session expired, or SQLite WAL not enabled. Check these four before digging deeper.

APP

Appendix

Reference tables, CLI commands, environment variables, and configuration quick-reference

Appendix A — Environment Variables Reference

Variable	Required	Default	Description
ANTHROPIC_API_KEY	Yes	—	Anthropic API key (sk-ant-...)
TELEGRAM_BOT_TOKEN	No	—	Telegram bot token from @BotFather
DISCORD_BOT_TOKEN	No	—	Discord bot token
DISCORD_CLIENT_ID	No	—	Discord application ID
SLACK_BOT_TOKEN	No	—	Slack bot OAuth token
SLACK_APP_TOKEN	No	—	Slack app-level token (Socket Mode)
SLACK_SIGNING_SECRET	No	—	Slack signing secret
DB_PATH	No	./data/openclaw.db	SQLite database file path
MAX_CONTAINERS	No	5	Maximum concurrent agent containers
AGENT_TIMEOUT_SECONDS	No	300	Per-agent execution timeout
AGENT_MEMORY_LIMIT	No	512m	Docker memory limit per agent
AGENT_CPU_LIMIT	No	0.5	Docker CPU limit per agent
MAX_HISTORY_MESSAGES	No	50	Messages loaded into context
LOG_LEVEL	No	info	Logging level (debug/info/warn/error)
CONTAINER_IMAGE	No	openclaw-agent:latest	Agent container image name
CONTAINER_RUNTIME	No	docker	Container runtime (docker/apple)
API_PORT	No	3000	REST API port (0 = disabled)
CLAUDE_MODEL	No	claude-sonnet-4-6	Claude model to use

CLAUDE_MAX_TOKENS	No	8192	Max tokens per Claude response

Appendix B — npm Scripts Reference

Command	Description
`npm run dev`	Start with tsx hot reload (development)
`npm run build`	Compile TypeScript to dist/
`npm start`	Run compiled production build
`npm run auth`	Run WhatsApp QR authentication
`npm test`	Run test suite (vitest)
`npm run test:watch`	Vitest in watch mode
`npm run lint`	ESLint code check
`npm run typecheck`	TypeScript type checking only
`npm run db:migrate`	Run pending database migrations
`npm run db:backup`	Manual database backup

Appendix C — Key File Locations

File / Folder	Purpose
`src/index.ts`	Main entry point
`src/channels/`	WhatsApp, Telegram, Discord, Slack adapters
`src/core/orchestrator.ts`	Central message routing logic
`src/core/claude.ts`	Claude API client wrapper
`src/core/scheduler.ts`	Cron task scheduler
`src/db/`	SQLite database layer
`src/agents/`	Container spawner, swarm coordinator
`container/`	Agent container source code

`data/groups/`	Per-group CLAUDE.md and skills
`data/global/CLAUDE.md`	Default system prompt
`config/`	Platform configuration files (53 files)

Appendix D — Claude Models Comparison

Model	Speed	Intelligence	Context	Best For
claude-haiku-4-5	⚡⚡⚡	★★☆	200K	Simple Q&A, quick responses
claude-sonnet-4-6	⚡⚡☆	★★★	200K	Default — balanced performance
claude-opus-4-6	⚡☆☆	★★★+	200K	Complex analysis, swarms

Appendix E — CLAUDE.md Template

Copy this template to `data/groups/{group-id}/CLAUDE.md` and customize:

```
# [Assistant Name] — [Group Name]

## Identity
You are [Name], the AI assistant for [Team/Company].
Your personality: [adjectives — e.g., "professional, helpful, concise"]
Your expertise: [domains]

## Communication Style
- Tone: [formal / casual / technical]
- Response length: [brief / detailed / match the question]
- Language: [respond in the language of the user's message]
- Formatting: [bullet points / prose / mixed]

## Team Context
- Team: [description]
- Main projects: [list]
- Key deadlines: [dates]
- Important links: [workspace/links.md]

## Standing Instructions
- [Rule 1]
- [Rule 2]
- [Add as many as needed]

## Tools & Resources
- For client data: read workspace/clients.json
- For project status: read workspace/projects.md
- Reports: write to workspace/reports/

## What You Should NOT Do
- [Limitation 1 — e.g., "Never share salary information"]
- [Limitation 2 — e.g., "Don't discuss competitor products"]
```

```
## Custom Commands
/help — List available commands
/report — Generate weekly summary
[Add your custom commands here]
```

You've Mastered OpenClaw

You've gone from zero to deploying a production-ready multi-channel AI assistant platform. You understand the architecture, the security model, the memory system, skills, scheduled tasks, agent swarms, MCP integrations, and production operations.

What You Can Build Now

AI-powered team assistants · Automated reporting systems · Customer support bots · Sales intelligence tools · Developer automation · Multi-channel business operations

Ariel Eisenstadt

Founder, PixMind Studio & PixiBot

OpenClaw 2026 | Claude Agent SDK | Build Without Limits