THE SPRINT

LOCAL AUTONOMOUS SAAS BUILDER PLAYBOOK

OpenClaw · Claude Code Max · Cursor · GitHub · Vercel

Everything runs locally on your machine.

OpenClaw manages the project. Claude Code builds. Cursor keeps you in control.

How This Works

This playbook documents a fully local autonomous build stack. OpenClaw runs on your machine with direct terminal access. It plans the work, constructs detailed task prompts, and calls Claude Code directly via terminal command. Claude Code reads your codebase, builds the feature, and pushes a pull request to GitHub. You review it in Cursor and merge. Vercel deploys automatically.

No cloud middleware. No polling scripts. No webhook plumbing between agents. OpenClaw calls Claude Code the same way you would type a command in your terminal — because it has terminal access.

📌 **The One-Line Integration**

OpenClaw's connection to Claude Code is this simple: it runs `claude "<task prompt>"` in your terminal. That's it. Claude Code receives the prompt, reads your repo, builds the code, and pushes a PR. Everything else in this playbook is configuration to make that one command as powerful as possible.

The Stack

TOOL	WHAT IT DOES
OpenClaw (local)	The project brain. Holds your product brief and sprint goals. Plans tasks, writes detailed specs, and executes `claude "..."` in your terminal to dispatch work to Claude Code.
Claude Code Max	The builder. Receives a task prompt from OpenClaw, reads your entire codebase for context, writes the code, commits it to a new branch, and opens a pull request on GitHub.
Cursor	Your cockpit. Watches every file Claude Code touches in real time. Where you review PRs, handle complex problems Claude Code can't fully solve, make surgical edits, and stay in control of the product direction.
GitHub	Code storage and deploy trigger. Claude Code pushes branches here. You review and merge PRs. A merge to main triggers automatic Vercel deployment.
Vercel	Production hosting. Deploys automatically on every merge to main. Zero manual deploy steps.
Supabase	Your product's database — not orchestration plumbing. Claude Code writes queries against your schema. OpenClaw uses it to track sprint state and task history if you want a log.

The Build Loop — Start to Finish

Before setting anything up, understand the full loop. Every build task flows through these exact steps:

#	TOOL	ACTION
1	**You**	Brief OpenClaw on what to build — a sprint goal, a feature, a fix — in plain English.
2	**OpenClaw**	Reads your product context and .cursorrules. Plans the task breakdown. Writes a detailed technical spec for the first task.
3	**OpenClaw**	Executes: claude "<full task spec>" in your local terminal. Claude Code starts immediately.
4	**Claude Code**	Reads your entire codebase for context. Checks .cursorrules for your conventions. Builds the feature across one or more files.
5	**Claude Code**	Creates a new git branch, commits the changes, pushes the branch to GitHub, and opens a Pull Request.
6	**Cursor**	You see the new PR. Open the branch in Cursor. Review the diff live. Test locally. Make any small edits directly in Cursor.
7	**You**	Approve and merge the PR on GitHub.
8	**Vercel**	Detects the merge to main. Deploys to production automatically within 60-90 seconds.
9	**OpenClaw**	Detects the completed task (via GitHub webhook or you briefing it). Plans and dispatches the next task. Loop repeats.

Phase 0 — Prerequisites

Install and verify all tools before touching configuration. A broken tool discovered mid-build is far more painful than 20 minutes of setup verification now.

Accounts You Need

SERVICE	WHERE & NOTES
Anthropic Max	anthropic.com — upgrade to the Max plan. Claude Code burns through tokens fast on real codebases. Max gives you the rate limits and priority access needed for autonomous loops.
GitHub	github.com — create a new private repo for your product. You will need a Personal Access Token (PAT) with repo scope.
Vercel	vercel.com — connect your GitHub repo. Vercel will auto-deploy on every merge to main.
Supabase	supabase.com — one project per product. Free tier works for early builds.
OpenClaw	Your local OpenClaw instance with terminal access enabled. Confirm you can run shell commands from inside OpenClaw before proceeding.

Install Claude Code

Claude Code is installed globally via npm. Run these commands in your terminal:

```bash
▸ bash
# Requires Node.js 18 or higher
# Check your Node version first
node --version

# Install Claude Code globally
npm install -g @anthropic-ai/claude-code

# Verify installation
claude --version

# Authenticate — this stores your API key securely
claude auth
# When prompted, paste your Anthropic API key
# Key is stored in ~/.claude/config.json
```

⚠️ **Max Plan Is Required for Autonomous Builds**
Claude Code running autonomously on a real codebase makes many API calls per task. On the standard plan you will hit rate limits within minutes and the build loop will stall. Upgrade to Max before running any of the workflows in this playbook.

Configure Claude Code Global Settings

Claude Code has a global config file that controls its default behavior. Set these before running any builds:

```
▸ bash
# Open the Claude Code config file
nano ~/.claude/config.json

# Or create it if it doesn't exist
mkdir -p ~/.claude && touch ~/.claude/config.json
```

Set the contents to:

```
▸ json
{
  "model": "claude-sonnet-4-5",
  "max_tokens": 8096,
  "auto_confirm": false,
  "git_auto_commit": false,
  "verbose": true
}

# auto_confirm: false means Claude Code will describe what it plans to do
# and ask you to confirm before making changes. Set to true only when
# running fully autonomous loops where you want zero interruptions.

# git_auto_commit: false means Claude Code stages changes but you
# commit and push via Cursor or terminal. Set to true for full autonomy.
```

Verify Claude Code Can Read Your Repo

Navigate to your project folder and run a quick test before connecting OpenClaw:

```bash
cd ~/.openclaw/workspace/projects/your-product

# Ask Claude Code to describe your codebase
# It should read all files and give you a summary
claude "Describe the structure of this codebase. What stack is it using,
what are the main directories, and what does each one contain?"

# If Claude Code responds with an accurate description of your repo,
# it has full codebase access and is ready to build.

# If it says it cannot find files, check you are in the right directory:
pwd
ls -la
```

Install Cursor and Connect It to Your Repo

1 **Download and install Cursor**
Go to cursor.sh and download the desktop app for your OS. Install it.

2 **Open your project in Cursor**
Launch Cursor. File → Open Folder → select your project directory. Cursor will index your codebase. This takes 1-2 minutes for the first open.

3 **Connect Cursor to Claude**
In Cursor → Settings → AI, set the model to Claude Sonnet.
Paste your Anthropic API key. This powers Cursor's inline AI and chat features.

4 **Verify Cursor AI is working**
Open any file in your project.
Press Cmd+L (Mac) or Ctrl+L (Windows) to open Cursor Chat.
Type: 'What does this file do?' — if you get a response, Cursor is connected.

Phase 1 — GitHub Setup

GitHub is where Claude Code pushes its work and where you review and merge. Two things to set up: a Personal Access Token so Claude Code can push branches, and a deploy workflow so Vercel fires automatically on merge.

Step 1.1 — Create a Personal Access Token

Claude Code needs a GitHub token to create branches and open pull requests. This token lives in your local environment, not in any config file that gets committed.

1 **Go to GitHub → Settings → Developer Settings → Personal Access Tokens → Tokens (classic)**
Click 'Generate new token (classic)'.

2 **Set the token permissions**
Name it 'Claude Code Local'.
Set expiration to No expiration (or 1 year).
Check the 'repo' scope — this gives full repository access.
Click 'Generate token' and copy it immediately. You will not see it again.

3 **Add the token to your shell environment**
Add this line to your ~/.zshrc or ~/.bashrc file:
export GITHUB_TOKEN=your_token_here
Then run: source ~/.zshrc to reload.

4 **Configure git credentials locally**
Claude Code uses your local git config to push. Set it up if you haven't:
git config --global user.name 'Your Name'
git config --global user.email 'you@email.com'

Step 1.2 — Initialize Your Local Repo

If you do not have a local repo yet, create one and connect it to GitHub:

```
▸ bash
```

```
# Create your project folder
mkdir your-product && cd your-product

# Initialize git
git init

# Create a basic Next.js project (or your preferred stack)
npx create-next-app@latest . --typescript --tailwind --app --no-src-dir

# Initial commit
git add .
git commit -m "Initial commit"

# Connect to your GitHub repo
git remote add origin https://github.com/yourusername/your-repo.git

# Push to GitHub
git branch -M main
git push -u origin main
```

Step 1.3 — Auto-Deploy Workflow

Create this file in your repo so every merge to main triggers an automatic Vercel deployment. Create the file at .github/workflows/deploy.yml:

```yaml
▸ yaml
name: Deploy to Production

on:
  push:
    branches: [main]

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - uses: actions/setup-node@v4
        with:
          node-version: '20'
          cache: 'npm'

      - run: npm install
      - run: npm run build
      - run: npm test --if-present

      - name: Deploy to Vercel
```

```
        uses: amondnet/vercel-action@v25
        with:
          vercel-token: ${{ secrets.VERCEL_TOKEN }}
          vercel-org-id: ${{ secrets.VERCEL_ORG_ID }}
          vercel-project-id: ${{ secrets.VERCEL_PROJECT_ID }}
          vercel-args: '--prod'
```

Add these three secrets to your GitHub repo under Settings → Secrets and variables → Actions:

SECRET	WHERE TO FIND IT
VERCEL_TOKEN	vercel.com → Settings → Tokens → Create token
VERCEL_ORG_ID	Run vercel link in your project folder. Stored in .vercel/project.json after linking.
VERCEL_PROJECT_ID	Same — stored in .vercel/project.json after running vercel link

Step 1.4 — Branch Protection

Protect main so Claude Code cannot push directly. All changes go through a PR that you review in Cursor.

1 **Go to your GitHub repo → Settings → Branches**
Click 'Add branch protection rule'.

2 **Set branch name pattern to: main**

3 **Enable these options**
✓ Require a pull request before merging
✓ Require approvals — set to 1
✓ Require status checks to pass (once you have tests)

Phase 2 — The .cursorrules File (Critical)

The .cursorrules file is the most important configuration in your entire stack. Both Cursor and Claude Code read it. It defines your tech stack, coding conventions, file structure, and patterns. Without it, Claude Code makes reasonable guesses that may not match what you want. With a good one, every file Claude Code writes looks like it came from the same developer.

💡 **What .cursorrules Actually Does**

When OpenClaw dispatches a task to Claude Code, Claude Code reads your entire codebase AND your .cursorrules file before writing a single line of code. It uses .cursorrules to understand: what stack you're on, which patterns to follow, which files not to touch, how to name things, and what done looks like. It is your standing instructions to every AI agent that ever touches this repo.

Creating Your .cursorrules File

Create this file in the root of your project. Customize every section for your actual product. The more specific you are, the better Claude Code performs.

▸ .cursorrules

```
# ══════════════════════════════════════════════════════════
# PROJECT: YourProductName
# ══════════════════════════════════════════════════════════

## What This Product Does
# One clear sentence: what it does, for whom, the core value.
# Example: A scheduling tool for freelance designers to manage
# client bookings and send automated reminders.

## Tech Stack
- Framework:   Next.js 14 with App Router (app/ directory)
- Database:    Supabase (Postgres + Auth + Storage)
- Styling:     Tailwind CSS — no CSS modules, no styled-components
- Payments:    Stripe
- Deployment:  Vercel
- Language:    TypeScript strict mode — no 'any' types ever
- Testing:     Vitest for unit tests, Playwright for e2e

## Architecture Rules
- Server Components by default. Add 'use client' only when required.
- All database access through: import { supabase } from '@/lib/supabase'
- All API routes in app/api/ as Route Handlers — never use pages/api/
- Auth via Supabase Auth — never roll custom auth
- Environment variables: NEXT_PUBLIC_ prefix for client-safe vars only

## File Structure
```

```
- app/            → Next.js pages and layouts (App Router)
- components/     → Reusable UI components
- lib/            → Utilities, clients, helpers
- hooks/          → Custom React hooks (prefix: use)
- types/          → TypeScript types and interfaces
- actions/        → Next.js Server Actions

## Naming Conventions
- Components:     PascalCase   (UserCard, BookingForm)
- Files:          kebab-case   (user-card.tsx, booking-form.tsx)
- Hooks:          camelCase    (useAuth, useBookings)
- DB functions:   camelCase    (getUser, createBooking)
- Constants:      UPPER_SNAKE (MAX_RETRY_COUNT)

## Code Patterns
- Forms:          React Hook Form + Zod resolver
- Data fetching: Supabase client in Server Components
- Loading:        Suspense boundaries — not loading state booleans
- Errors:         error.tsx files — not try/catch everywhere
- Validation:     Zod schemas in lib/schemas.ts

## Do Not Touch
- lib/supabase.ts         — the database client setup
- middleware.ts           — auth middleware
- app/layout.tsx          — root layout
- types/database.ts       — generated Supabase types, never edit manually

## Pull Request Rules
- One PR per task. Do not bundle unrelated changes.
- PR title format:  feat: description  or  fix: description
- Always include a brief description of what was built and why.
- If a task requires a schema change, create the migration file
  in supabase/migrations/ with a timestamp prefix.
```

⚠️ Update .cursorrules Before Every Sprint

As your codebase grows, .cursorrules needs to grow with it. After each sprint, add the key patterns and decisions that were made. If Claude Code introduced a new utility function or established a new pattern, document it here so every future task follows the same convention.

Phase 3 — Supabase Setup

Supabase serves two roles in this stack: it is your product's live database, and it optionally stores your sprint context and task history so OpenClaw has persistent memory across sessions. Set up the product tables first, then the agent tables if you want the logging layer.

Step 3.1 — Create Your Supabase Project

1 **Create a new project at supabase.com**
Name it after your product. Save your database password.

2 **Collect your credentials**
Go to Project Settings → API.
Copy three values: Project URL, anon public key, service_role secret key.
Store the service_role key securely — never commit it to your repo.

3 **Add credentials to your local .env.local file**
Create .env.local in your project root (it is already in .gitignore for Next.js).
Never commit this file to GitHub.

▸ .env.local

```
# .env.local — never commit this file
NEXT_PUBLIC_SUPABASE_URL=https://yourproject.supabase.co
NEXT_PUBLIC_SUPABASE_ANON_KEY=your_anon_key_here
SUPABASE_SERVICE_KEY=your_service_role_key_here

# Stripe (add when you set up payments)
STRIPE_SECRET_KEY=sk_test_...
NEXT_PUBLIC_STRIPE_PUBLISHABLE_KEY=pk_test_...
STRIPE_WEBHOOK_SECRET=whsec_...
```

Step 3.2 — Generate TypeScript Types

This is the step most people skip and then wonder why Claude Code writes wrong column names. Generate a types file from your live Supabase schema and commit it. Claude Code uses this file to write type-safe queries without guessing.

```bash
# Install Supabase CLI
npm install -g supabase

# Login
supabase login

# Generate types from your live database schema
# Replace YOUR_PROJECT_ID with the ID from your Supabase project URL
supabase gen types typescript --project-id YOUR_PROJECT_ID > types/database.ts

# Commit this file — Claude Code reads it for every database task
git add types/database.ts
git commit -m "chore: add Supabase TypeScript types"
git push

# Run this command again every time you change your database schema
```

Step 3.3 — Create the Typed Supabase Client

Create lib/supabase.ts — this is the single file all database access goes through. Claude Code will import from here for every query it writes.

```typescript
// lib/supabase.ts
// Claude Code imports from this file for ALL database access
// Do not create Supabase clients anywhere else

import { createClient } from '@supabase/supabase-js'
import type { Database } from '@/types/database'

// Client-side Supabase client (respects Row Level Security)
export const supabase = createClient<Database>(
  process.env.NEXT_PUBLIC_SUPABASE_URL!,
  process.env.NEXT_PUBLIC_SUPABASE_ANON_KEY!
)

// Server-side client for Server Components and API routes
// Uses service_role key — bypasses RLS — only use server-side
export const createServerClient = () =>
  createClient<Database>(
    process.env.NEXT_PUBLIC_SUPABASE_URL!,
    process.env.SUPABASE_SERVICE_KEY!
  )
```

Step 3.4 — Optional: Sprint and Task Tables

If you want OpenClaw to log sprint plans, task history, and build decisions persistently across sessions, create these tables in Supabase SQL Editor. This gives OpenClaw memory — it can read what was already built before planning the next sprint.

```sql
-- Sprint tracking
create table sprints (
  id           uuid primary key default gen_random_uuid(),
  name         text not null,
  goal         text,
  status       text default 'active',  -- active | complete
  created_at   timestamptz default now(),
  completed_at timestamptz
);

-- Task history (what OpenClaw dispatched to Claude Code)
create table tasks (
  id           uuid primary key default gen_random_uuid(),
  sprint_id    uuid references sprints(id),
  title        text not null,
  spec         text,         -- full prompt sent to Claude Code
  status       text default 'dispatched',  -- dispatched | done | failed
  pr_url       text,         -- GitHub PR link
  outcome      text,         -- brief summary of what was built
  created_at   timestamptz default now(),
  completed_at timestamptz
);

-- Decision log (architectural choices, blockers, notes)
create table build_log (
  id          uuid primary key default gen_random_uuid(),
  type        text,     -- decision | blocker | note | completion
  content     text not null,
  created_at timestamptz default now()
);
```

Phase 4 — OpenClaw Configuration

OpenClaw is the brain of the operation. You will configure it with: your project context, the terminal commands it uses to call Claude Code, and the system prompt that governs how it plans and manages the build.

Step 4.1 — Verify Terminal Access in OpenClaw

Before anything else, confirm OpenClaw can execute terminal commands on your machine. In OpenClaw, run a test command:

```bash
# Test that OpenClaw can execute terminal commands
echo 'OpenClaw terminal access confirmed'

# Test that Claude Code is accessible from OpenClaw's terminal
claude --version

# Test that OpenClaw can navigate to your project
cd ~/.openclaw/workspace/projects/your-product && ls

# All three should return output.
# If claude --version fails, Claude Code is not in OpenClaw's PATH.
# Fix: add the npm global bin to OpenClaw's shell PATH setting.
```

⚠️ PATH Issue — Most Common Setup Problem

If OpenClaw's terminal cannot find the claude command even though it works in your regular terminal, it is a PATH problem. The npm global bin directory is not in OpenClaw's shell environment. Fix: in OpenClaw's terminal settings, add the full path to npm global binaries, typically: /usr/local/bin or $(npm bin -g). Run `which claude` in your regular terminal to find the exact path.

Step 4.2 — Create the OpenClaw Project

In OpenClaw, create a new Project for your product. This is where your product brief, stack context, and sprint state live. OpenClaw reads this before every session.

1 **Create a new OpenClaw Project**
Name it after your product. This is the context container for everything OpenClaw knows about your build.

2 **Add your Product Brief to the project context**
Paste this template into the project's context or system prompt, filled in with your actual product details:

```
▸ project context
PRODUCT: YourProductName

WHAT IT DOES:
One clear sentence — what the product does, for whom, and the core value.

TARGET USER:
Describe the user. Their role, their pain point, what they need.

TECH STACK:
Next.js 14 App Router, Supabase, Tailwind CSS, Stripe, Vercel, TypeScript

REPO:
https://github.com/yourusername/your-repo
Local path: ~/.openclaw/workspace/projects/your-product

CURRENT STATUS:
Update this after each sprint. What has been built? What is working?
Example: Auth complete. Dashboard shell built. User table in Supabase.
Onboarding flow not started. Stripe not integrated yet.

CONVENTIONS:
.cursorrules file in the repo root defines all code conventions.
Claude Code reads this automatically — do not repeat rules here.

DO NOT:
- Make product decisions without asking me
- Create tasks larger than 1-3 files
- Touch the files listed in .cursorrules Do Not Touch section
- Start a new task while a PR is still waiting for review
```

Step 4.3 — The Orchestrator System Prompt

This is the most important configuration. Create a new Agent in OpenClaw called 'Build Orchestrator'. Set model to Claude Sonnet. Paste this as the system prompt:

```
▸ system prompt
You are the Build Orchestrator for a local autonomous SaaS development studio.
You manage the build of a SaaS product by planning tasks and dispatching them
to Claude Code via your terminal access.
```

```
YOUR CORE CAPABILITY:
You have direct terminal access. You call Claude Code by running:
  cd <project_path> && claude "<task spec>"
This is the primary action you take. Everything else supports this.

YOUR OPERATING LOOP:

1. UNDERSTAND CURRENT STATE
   Before planning anything, read:
   - The product brief and current status in your project context
   - The .cursorrules file:  cat <project_path>/.cursorrules
   - Recent git log:  cd <project_path> && git log --oneline -20
   - Open PRs:  cd <project_path> && gh pr list
   - Current file structure:  cd <project_path> && find . -type f -name '*.tsx' -o
-name '*.ts' | head -40

2. WAIT FOR OPEN PRS BEFORE CREATING NEW TASKS
   If  gh pr list  returns any open PRs, stop.
   Report: 'There is an open PR waiting for review: [PR title] [URL]'
   Do not create or dispatch new tasks until PRs are merged.
   Maximum 1 task in flight at a time.

3. PLAN THE NEXT TASK
   Based on the sprint goal and what has been built:
   - Identify the single most important next task
   - Break it down to 1-3 files maximum
   - Think through dependencies: does this require something else first?
   - Build order: schema → server logic → API routes → UI → tests

4. WRITE A COMPLETE TASK SPEC
   Your task spec is what Claude Code receives as its prompt.
   A good spec includes:
   - WHAT to build (specific feature, component, or function)
   - WHERE to build it (exact file paths to create or modify)
   - DATA SHAPES (types, interfaces, what fields exist)
   - HOW it connects to existing code (import paths, function signatures)
   - ACCEPTANCE CRITERIA (how Claude Code knows it is done)
   - CONSTRAINTS (what not to change, what patterns to follow)
   - REFERENCE: 'Follow all conventions in .cursorrules'

5. DISPATCH TO CLAUDE CODE
   Run this terminal command:
   cd <project_path> && claude "<your complete task spec>"

   Claude Code will:
   - Read the codebase and .cursorrules
   - Build the feature
   - Create a branch and open a PR
   - You will see terminal output as it works

6. REPORT WHAT WAS DISPATCHED
   After dispatching, tell me:
   - What task you sent
```

```
    - Why you chose this task next
    - What task comes after this one
    - Any decisions or assumptions you made

RULES:
- Never dispatch a task without first reading the current git state
- Never create overlapping tasks
- If you are uncertain about a product decision, ask — do not assume
- One task at a time. Wait for PR merge before next dispatch.
- If Claude Code fails (exits with error), report the error and ask how to
proceed.
```

Step 4.4 — The Claude Code Dispatch Command

This is the exact terminal command OpenClaw runs to call Claude Code. The structure matters — get this right and the rest of the system works.

```bash
# The core dispatch pattern OpenClaw uses:
cd ~/.openclaw/workspace/projects/your-product && claude "<task spec>"

# For fully autonomous execution (no confirmation prompts):
cd ~/.openclaw/workspace/projects/your-product && claude --yes "<task spec>"

# The --yes flag auto-confirms all of Claude Code's actions.
# Use this only when you trust the task spec is precise.
# Without --yes, Claude Code will pause and ask for confirmation
# before making changes — useful when testing a new type of task.

# Example of a complete dispatch command OpenClaw would run:
cd ~/.openclaw/workspace/projects/your-product && claude --yes "
Build a UserCard component at components/UserCard/index.tsx.
It receives a user prop of type User (from types/database.ts).
Display: avatar image (from user.avatar_url), full name, email, role badge.
Role badge: green for 'admin', blue for 'user'.
Use Tailwind for styling. Follow all conventions in .cursorrules.
Create a Vitest unit test at components/UserCard/index.test.tsx.
After building, create a branch named feat/user-card and open a PR.
"
```

💡 **The --yes Flag and When to Use It**

Without --yes: Claude Code describes each action and waits for your confirmation. Good for the first time you run a new type of task, or for tasks that touch sensitive files. With --yes: Claude Code runs completely autonomously start to finish. Good for routine build tasks once you trust the spec quality. Start without --yes until you have confidence in how OpenClaw writes specs, then switch to --yes for full autonomy.

Step 4.5 — OpenClaw Terminal Commands for Context Reading

Before dispatching any task, OpenClaw runs these commands to read the current state of your project. Add these as named actions in OpenClaw so it can call them in sequence:

```bash
# 1. Read current file structure (what exists already)
cd ~/.openclaw/workspace/projects/your-product && find . -type f \( -name '*.tsx'
-o -name '*.ts' \) | grep -v node_modules | grep -v .next | sort

# 2. Read recent git history (what was recently built)
cd ~/.openclaw/workspace/projects/your-product && git log --oneline -15

# 3. Check for open PRs (are we waiting for a review?)
cd ~/.openclaw/workspace/projects/your-product && gh pr list

# 4. Read the .cursorrules file (conventions reminder)
cat ~/.openclaw/workspace/projects/your-product/.cursorrules

# 5. Read current sprint context from Supabase (if using sprint tables)
# OpenClaw queries Supabase directly via its HTTP action
# GET https://yourproject.supabase.co/rest/v1/sprints?status=eq.active

# Run commands 1-4 before EVERY task dispatch.
# This ensures OpenClaw always has accurate context.
```

⚠️ Never Skip the Context Read

The most common cause of Claude Code building the wrong thing is OpenClaw dispatching a task without first reading what already exists. If OpenClaw does not check git log and the file tree before writing its spec, it will ask Claude Code to build something that was already built in the last sprint, or build something that conflicts with existing code.

Proof of Connection — Stack Verified

Before teaching this stack to your community, run these three tests yourself. They take under 5 minutes. Every result below was confirmed on a live machine in February 2026. The stack is real and operational.

Test 1 — Confirm Claude Code Is Accessible

Inside OpenClaw, open its terminal interface and run:

```bash
claude --version
```

✅ **Expected output:**

```
claude 2.1.58 (Claude Code)
```

If you see a version number, Claude Code is in OpenClaw's PATH and ready to receive tasks.

⚠️ **If 'command not found'**

Claude Code is not in OpenClaw's PATH. Run `which claude` in your regular terminal to find the install path. Then in OpenClaw's shell settings add: `export PATH=$PATH:/path/to/claude`

Test 2 — Confirm Claude Code Can Read Your Codebase

Run this from OpenClaw's terminal — replace the path with your actual project:

```bash
cd ~/.openclaw/workspace/projects/your-product
claude "List the files in this directory and describe what kind of
project this is. Do not make any changes."
```

✅ **Confirmed result:**

Claude Code returned an accurate description of the full codebase — stack, directories, file structure, and purpose. No files were modified.

OpenClaw can call Claude Code and Claude Code has full read access to your project.

Test 3 — Confirm the Full Loop: Build, Branch, Commit, Push

This is the definitive test. OpenClaw instructs Claude Code to create a file, commit it, and push a branch to GitHub. If this passes, your entire autonomous build stack is operational.

```
▸ OpenClaw prompt
# Paste this into OpenClaw as a prompt:

Navigate to my project at ~/.openclaw/workspace/projects/your-product
and run this Claude Code command:

Create a file called SPRINT_TEST.md in the project root.
Contents:
  Title: Autonomous Build Stack - Connection Confirmed
  Date: today's date
  Stack: OpenClaw, Claude Code Max, Cursor, GitHub, Vercel

Then:
1. Create a git branch called test/sprint-connection
2. Commit the file with message: Test: autonomous build stack confirmed
3. Push the branch to GitHub
4. Do not open a PR
Report back each step as it completes.
```

STEP	CONFIRMED RESULT
✅ **File created**	SPRINT_TEST.md created in project root with correct content and today's date
✅ **Branch created**	test/sprint-connection branch created locally
✅ **Committed**	Commit: 'Test: autonomous build stack confirmed' (hash: 8266ff9)
✅ **Pushed to GitHub**	Branch live at origin/test/sprint-connection on GitHub. PR URL generated.

💡 **Once Test 3 Passes — You Are Ready to Build**

A branch appearing on GitHub, created entirely by AI on your instruction, means every piece of the stack is connected and working. You can now brief OpenClaw on a real sprint and trust that Claude Code will execute, commit, and push. Clean up the test branch on GitHub before starting your first real sprint.

Your Confirmed Workspace Path

OpenClaw stores its projects in a local workspace directory. Use this exact path structure in every dispatch command — not ~/projects/ or any other location:

```bash
# OpenClaw workspace root:
~/.openclaw/workspace/projects/

# Your product lives at:
~/.openclaw/workspace/projects/your-product-name/

# Every task dispatch uses this pattern:
cd ~/.openclaw/workspace/projects/your-product-name && claude --yes "<task spec>"

# Confirm your projects are there:
ls ~/.openclaw/workspace/projects/
```

⚠️ Always Use the Full OpenClaw Workspace Path

Using the wrong path is the #1 reason Claude Code reports it cannot find your files. OpenClaw projects live at ~/.openclaw/workspace/projects/ — not ~/projects/ or your Desktop. Copy the path above and use it exactly in every task dispatch command throughout this playbook.

Phase 5 — Cursor as Your Cockpit

With OpenClaw and Claude Code handling the build loop autonomously, Cursor becomes your primary interface for staying in control. Here is exactly what Cursor does in this stack and how to use it effectively.

What Cursor Does in This Stack

CURSOR ROLE	DESCRIPTION
Live visibility	Cursor watches your project folder. Every file Claude Code creates or edits appears instantly in Cursor's file explorer and diff view — without you touching the terminal.
PR review interface	When Claude Code opens a PR, you review the diff in Cursor before merging. Cursor shows you exactly what changed, highlighted line by line.
Hard problem co-pilot	When Claude Code gets stuck or produces something wrong, you jump into Cursor Chat with full codebase context. Cursor helps you understand and fix the issue.
Direct editing	For small fixes, UI polish, and anything requiring your judgment — you edit directly in Cursor. No need to go through the full OpenClaw → Claude Code loop for a 3-line change.
Architecture advisor	Use Cursor Chat to think through complex decisions before briefing OpenClaw. 'What is the best way to structure the payment flow?' — Cursor reads your code and gives you a grounded answer.

Your PR Review Workflow in Cursor

This is the most important Cursor workflow. Follow it every time Claude Code opens a PR.

#	TOOL	ACTION
1	**Notification**	GitHub sends you an email or Slack message: new PR opened. Or check GitHub directly.
2	**Cursor**	In the Source Control panel (Ctrl+Shift+G), click Fetch. The new branch appears.
3	**Cursor**	Click the branch name to check it out. Your local files now match what Claude Code built.
4	**Cursor**	Open the diff view. Review every changed file. Green = added, Red = removed.

5	**Cursor Chat**	For anything that looks wrong or confusing, press Cmd+L. Ask: 'Why was this structured this way?' or 'Is there a simpler approach?'
6	**Cursor**	Make any small edits directly in Cursor. Stage and commit them on the same branch.
7	**You**	If the PR looks good — go to GitHub, approve, and merge. Vercel deploys in ~90 seconds.
8	**You**	If major rework is needed — comment on the PR describing what to change. Brief OpenClaw to dispatch a follow-up task.

Watching Claude Code Build in Real Time

When OpenClaw dispatches a task and Claude Code starts running in the terminal, open Cursor to the same project folder. You will see files being created and modified in real time in Cursor's file explorer.

1 **Keep Cursor open to your project folder while Claude Code runs**
You do not need to do anything. Just watch.

2 **When a file changes, Cursor highlights it in the explorer**
Click on it to see what was written. You can intervene at any time by stopping Claude Code in the terminal.

3 **Use Cursor Chat to understand decisions as they happen**
Press Cmd+L and ask about any file Claude Code just created. 'What is this hook doing and does it follow our patterns?'

Using Cursor Chat for Architecture Decisions

Before briefing OpenClaw on a complex new feature, use Cursor Chat to think it through with full codebase context. This produces better task specs and fewer rework cycles.

```
▸ cursor chat
# Example Cursor Chat prompts to use BEFORE briefing OpenClaw:

"Given our current codebase, what is the cleanest way to add
real-time notifications? What files would need to change?"
```

```
"We need to add team/workspace support. Looking at the current
user model in types/database.ts, what schema changes would this require?"

"The dashboard is slow. Looking at the data fetching in
app/dashboard/page.tsx, what is causing this and how should we fix it?"

"I want to add Stripe subscriptions. Looking at how we currently
handle auth, what is the correct integration approach?"

# Cursor reads your entire codebase before answering.
# The answers you get here become the foundation of your task specs.
```

Phase 6 — Running Your First Build

Everything is configured. Here is the exact sequence for starting your first sprint from zero — going from product idea to code in production.

Step 6.1 — Brief OpenClaw on Sprint 1

Open OpenClaw and give the Build Orchestrator this prompt — customized for your actual product:

```
▸ openclaw prompt
# Paste this into OpenClaw's chat, customized for your product:

We are starting Sprint 1 on [ProductName].

The product is: [one sentence description]
Target user: [who it's for and their pain point]
Stack: Next.js 14 App Router, Supabase, Tailwind, TypeScript, Vercel
Repo is at: ~/.openclaw/workspace/projects/your-product

Sprint 1 goal: Get a working foundation — project scaffolded,
Supabase connected, email auth working, and a protected dashboard
shell the user lands on after login.

Before creating any tasks:
1. Read the current file structure
2. Read .cursorrules
3. Check git log to see if anything was already built
4. Check for open PRs

Then plan the first 3 tasks in dependency order and dispatch Task 1.
Wait for my PR review before dispatching Task 2.
```

Step 6.2 — What Happens Next

OpenClaw will run its context-reading commands, plan the sprint, write a spec for Task 1, and execute claude "<spec>" in your terminal. Here is what you will see:

- OpenClaw reports the task plan and what it is dispatching first
- Your terminal shows Claude Code starting — it reads the codebase first, then starts writing files
- Cursor's file explorer shows files being created and modified in real time
- After 5-15 minutes (depending on task complexity), Claude Code finishes and reports completion

- A new PR appears on GitHub — you get an email notification
- Open the PR in Cursor, review the diff, test locally, and merge
- Vercel deploys automatically — your product is live within 90 seconds of merge
- Brief OpenClaw to dispatch Task 2

Step 6.3 — The Rhythm After Sprint 1

Once the first sprint is done, this becomes your regular workflow:

WHEN	WHAT YOU DO
Start of sprint	Update the 'CURRENT STATUS' section of your OpenClaw product brief. Tell OpenClaw the new sprint goal.
OpenClaw plans	Review the task plan OpenClaw proposes. Push back on anything too large or in the wrong order. Approve the plan.
Task in progress	Watch in Cursor. You can intervene in the terminal at any time if Claude Code heads in the wrong direction.
PR is open	Review in Cursor. Merge if good. Comment with rework instructions if not.
After merge	Confirm with OpenClaw the task is done. Brief it to dispatch the next task.
Blocker hit	Claude Code reports it cannot complete a task. Read the error in terminal. Jump into Cursor Chat to diagnose. Rewrite the spec with OpenClaw.
Sprint complete	Do a short Cursor Chat review: 'What was built this sprint and what should we tackle next?'

Phase 7 — Writing Task Specs That Work

The quality of what Claude Code builds is directly proportional to the quality of the spec it receives. OpenClaw writes the specs, but you need to understand what a good spec looks like so you can brief OpenClaw correctly and review its plans before dispatch.

The Anatomy of a Good Task Spec

```
▸ task spec
# GOOD TASK SPEC EXAMPLE
# This is what OpenClaw should generate before calling claude

Build a BookingForm component.

FILE: components/BookingForm/index.tsx (create new)

WHAT IT DOES:
A form that lets users book a 30 or 60 minute session.
Fields: date picker (calendar), time slot selector, session length radio (30/60
min),
notes textarea (optional), submit button.

DATA:
On submit, call the createBooking server action from actions/bookings.ts.
Booking type (from types/database.ts
Database['public']['Tables']['bookings']['Insert']):
  user_id: string, date: string (YYYY-MM-DD), time_slot: string (HH:MM),
  duration_minutes: 30 | 60, notes: string | null

VALIDATION:
Use React Hook Form + Zod. Schema in lib/schemas.ts (add bookingSchema there).
Required: date, time_slot, duration_minutes.
Date must not be in the past. Time slot must be a valid HH:MM string.

STYLING:
Tailwind CSS only. Follow existing form patterns from
components/AuthForm/index.tsx.
Submit button: disabled while submitting, shows spinner.

ERROR HANDLING:
Show inline field errors. Show a toast notification on success using the
existing useToast hook from hooks/useToast.ts.

TESTS:
Create components/BookingForm/index.test.tsx with Vitest.
Test: renders correctly, validates required fields, calls createBooking on valid
submit.

CONSTRAINTS:
```

```
Do not modify actions/bookings.ts — that file already exists.
Do not modify the database schema.
Follow all conventions in .cursorrules.

BRANCH NAME: feat/booking-form
PR TITLE: feat: add BookingForm component with validation
```

Good vs Bad Spec Patterns

BAD SPEC	GOOD SPEC
'Build the booking feature'	'Build a BookingForm component at components/BookingForm/index.tsx'
'Add some validation'	'Validate with Zod: date required, must not be past. time_slot required, HH:MM format.'
'Style it nicely'	'Tailwind only. Follow the pattern in components/AuthForm/index.tsx'
'Connect to the database'	'Call the createBooking server action from actions/bookings.ts. Booking type is Database[public][Tables][bookings][Insert]'
'Make it work like the other forms'	'Follow existing form patterns: React Hook Form + Zod resolver. useToast hook for notifications.'
'Add tests'	'Create components/BookingForm/index.test.tsx. Test: renders, validates required fields, calls createBooking on valid submit.'

Task Size Guidelines

If you or OpenClaw cannot clearly specify a task in under 30 lines, the task is too large. Break it down.

- Ideal task: 1-3 files, one clear user-facing outcome, completable in one Claude Code session
- Too large: 'Build the entire onboarding flow' → break into: schema migration, server actions, step 1 UI, step 2 UI, step 3 UI, completion handler
- Too small: 'Fix this typo' → do this yourself in Cursor in 10 seconds
- Just right: 'Build the OnboardingStep1 component with name/role fields, validation, and next-step navigation'

Phase 8 — Troubleshooting

Claude Code is not found when OpenClaw runs the terminal command

This is a PATH issue. OpenClaw's shell environment does not include the npm global bin directory.

```bash
# In your regular terminal, find where claude is installed:
which claude
# Example output: /usr/local/bin/claude

# In OpenClaw's terminal settings, add the full path explicitly:
export PATH=$PATH:/usr/local/bin

# Or call claude with its full path in the dispatch command:
/usr/local/bin/claude --yes "your task spec"

# Alternative: add to OpenClaw's shell init
export PATH=$PATH:$(npm bin -g)
```

Claude Code builds the wrong thing or misses the context

- The task spec is too vague — add explicit file paths, data types, and acceptance criteria
- OpenClaw did not read the current file structure before writing the spec — check that the context-reading commands ran first
- .cursorrules is missing or outdated — update it to reflect current patterns
- Task is too large — Claude Code loses focus on very large tasks. Split into smaller units.

```bash
# Debug: ask Claude Code directly what it sees in the repo
cd ~/.openclaw/workspace/projects/your-product && claude "List all TypeScript files in the components/
directory and describe what each one does. Then describe the pattern
used in the most recent component that was added."

# This tells you exactly what context Claude Code has before you send
# a real task. If it misses files or describes things wrongly,
# check your .gitignore isn't hiding files and that Claude Code
# is running from the correct project root directory.
```

Claude Code pushes code that breaks existing features

- Add explicit 'Do not modify' constraints to your task spec for files that should not change
- Enable branch protection with required tests passing — failing tests block the merge
- Run the test suite after Claude Code finishes but before you merge: npm test
- Use the --no flag to stop Claude Code mid-run if you see it heading in the wrong direction: Ctrl+C in the terminal

OpenClaw tries to dispatch a new task while a PR is still open

This is a spec problem — the system prompt tells OpenClaw to check for open PRs first. If it is skipping this step, reinforce it:

```
▸ openclaw prompt
# Tell OpenClaw explicitly before each session:
Before doing anything, run: cd ~/.openclaw/workspace/projects/your-product && gh
pr list
If there are any open PRs, show them to me and stop.
Do not plan or dispatch anything until I confirm the PR is merged.
```

Vercel deploy fails after merge

- Check the GitHub Actions log — the build step usually shows the exact error
- Most common cause: environment variables not set in Vercel. Go to Vercel → Project → Settings → Environment Variables and add all variables from your .env.local
- TypeScript errors will fail the build — Claude Code sometimes introduces type errors. Review the Actions log and fix in Cursor before re-merging

Quick Reference

Daily Build Commands

```bash
# Check for open PRs before starting
cd ~/.openclaw/workspace/projects/your-product && gh pr list

# Check recent build history
cd ~/.openclaw/workspace/projects/your-product && git log --oneline -10

# Run Claude Code interactively (you type tasks manually)
cd ~/.openclaw/workspace/projects/your-product && claude

# Run a single task non-interactively
cd ~/.openclaw/workspace/projects/your-product && claude "your task spec here"

# Run fully autonomously (no confirmation prompts)
cd ~/.openclaw/workspace/projects/your-product && claude --yes "your task spec
here"

# Ask Claude Code what it sees in the codebase
cd ~/.openclaw/workspace/projects/your-product && claude "Describe the current
state of this codebase.
What has been built so far and what patterns are established?"

# Regenerate Supabase types after schema changes
supabase gen types typescript --project-id YOUR_ID > types/database.ts
```

The Stack at a Glance

TOOL	YOUR INTERACTION
OpenClaw	Brief it on sprint goals. Review its task plans. It dispatches claude commands.
Claude Code	Mostly hands-off. Watch terminal output. Ctrl+C to stop if wrong direction.
Cursor	Always open. Review PRs here. Use Chat for hard problems and architecture.
GitHub	Approve and merge PRs. Watch Actions for build status.
Vercel	Auto-deploys on merge. Check dashboard if deploy fails.

Supabase	Add tables via SQL Editor. Run: supabase gen types after schema changes.

The Three Modes of Operation

MODE	WHEN TO USE IT
Full autonomous (OpenClaw → Claude Code --yes)	Well-defined tasks with precise specs. You are confident in what should be built. Maximum speed.
Semi-autonomous (OpenClaw → Claude Code without --yes)	New types of tasks or sensitive areas. Claude Code pauses for confirmation before each change. You stay in control.
Interactive (you in terminal + Cursor open)	Debugging, exploration, architectural decisions, quick fixes. Run claude in terminal and type instructions directly while watching in Cursor.

THE SPRINT Community

Questions about this setup? Post in the community.

@mattganzak