OpenClaw Mission Control: Build It Yourself
Step-by-Step Prompts for The Sprint Community

BEFORE YOU START

Prerequisites:
- [ ] Node.js 18+
- [ ] Mac or Linux (Windows via WSL)
- [ ] Code editor (VS Code recommended)
- [ ] Git (optional, but helpful)
- [ ] 30-40 hours of time over 3-4 weeks

Setup (10 minutes):
1. Clone or download the codebase from: [GitHub Link]
2. Run `cd openclaw-control-center`
3. Follow SETUP.md to get the app running locally

Test your setup:
```bash
# Terminal 1 - Backend
cd backend
npm install
npx prisma migrate dev
npm run dev

# Terminal 2 - Frontend
cd frontend
npm install
npm run dev

# Should see:
# Backend: http://localhost:3000
# Frontend: http://localhost:3001
```

PHASE 3: OpenClaw Adapter
Time: 3-4 days | Difficulty: Medium | Core Skill: Child processes & streaming

What You'll Build
A service that:

- Reads your OpenClaw config file
- Discovers available agents
- Executes tasks via child_process
- Streams output to the UI in real-time
- Validates paths for security

Prompt 1: Create the OpenClaw Adapter

Your task:
Create `/backend/src/lib/openclaw.ts` with an `OpenClawAdapter` class.

Requirements:
- Read and parse `~/.openclaw/openclaw.json`
- Extract agent list from config
- Implement `getAgents()` function
- Implement `validatePath()` function (security!)
- Export as singleton

Hint:
- Use Node.js `fs` module to read files
- Use `path.resolve()` for path validation
- Use `os.homedir()` to expand `~` in paths

Verification:
```bash
# In backend directory
npm run type-check  # Should have no errors
```

Code template (fill in the blanks):
```typescript
// /backend/src/lib/openclaw.ts
import fs from 'fs';
import path from 'path';
import os from 'os';

export class OpenClawAdapter {
  private configPath: string;
  private config: any = null;

  constructor(configPath: string = '~/.openclaw/openclaw.json') {
    // TODO: Expand the path (~ → home directory)
    this.configPath = /* YOUR CODE */;
  }
```

```
  private expandPath(filePath: string): string {
    // TODO: Replace ~ with home directory
    if (filePath.startsWith('~')) {
      return path.join(/* YOUR CODE */);
    }
    return filePath;
  }

  async loadConfig(): Promise<any> {
    // TODO: Read and parse the JSON file
    const content = fs.readFileSync(/* YOUR CODE */);
    this.config = JSON.parse(content);
    return this.config;
  }

  async getAgents(): Promise<any[]> {
    if (!this.config) {
      await this.loadConfig();
    }
    // TODO: Return the agents array from config
    return /* YOUR CODE */;
  }

  validatePath(filePath: string, allowlist: string[]): boolean {
    // TODO: Check if filePath is within any allowlist item
    const resolved = path.resolve(filePath);

    for (const allowed of allowlist) {
      const allowedResolved = path.resolve(this.expandPath(allowed));
      // TODO: Check if resolved is within allowedResolved
      if (/* YOUR CODE */) {
        return true;
      }
    }

    return false;
  }
}

export const openclawAdapter = new OpenClawAdapter();
```

Prompt 2: Add Agent Discovery API Endpoint

Your task:
Add a new endpoint to `/backend/src/server.ts` that discovers and stores agents.

Requirements:
- Endpoint: `POST /api/settings/discover-agents`
- Read config using OpenClawAdapter
- Store/update agents in database (Prisma)
- Return list of discovered agents

Hint:
- Use `prisma.agent.upsert()` to create or update agents
- Handle errors gracefully (config file not found, parse errors)

Code template:
```typescript
// Add to /backend/src/server.ts

app.post('/api/settings/discover-agents', async (req, res) => {
  try {
    // TODO: Call openclawAdapter.getAgents()
    const agents = /* YOUR CODE */;

    // TODO: For each agent, upsert it into the database
    for (const agent of agents) {
      await prisma.agent.upsert({
        where: { id: agent.id },
        update: {
          name: agent.name,
          model: agent.model,
          // TODO: Any other fields
        },
        create: {
          id: agent.id,
          name: agent.name || agent.id,
          model: agent.model || 'claude-haiku',
          capabilities: JSON.stringify(agent.capabilities || []),
          // TODO: Any other fields
        }
      });
    }
```

```
    // TODO: Return the updated agents list
    const allAgents = /* YOUR CODE */;
    res.json(allAgents);
  } catch (error) {
    console.error('Agent discovery failed:', error);
    res.status(500).json({ error: 'Failed to discover agents' });
  }
});
```

Test it:
```bash
curl -X POST http://localhost:3000/api/settings/discover-agents

# Should return: [{ id, name, model, capabilities, ... }]
```

Prompt 3: Implement Task Execution

Your task:
Create `/backend/src/services/taskExecutor.ts` that runs a task and streams output.

Requirements:
- Accept task ID, command, and working directory
- Create a TaskRun record in DB
- Spawn child_process with the command
- Capture stdout/stderr
- Broadcast output via WebSocket
- Save final status and logs to DB

Hint:
- Import `spawn` from `child_process`
- Return a Promise that resolves when process exits
- Use WebSocket to broadcast to connected clients

Code template:
```typescript
// /backend/src/services/taskExecutor.ts
import { spawn } from 'child_process';
import { PrismaClient } from '@prisma/client';
import { WebSocketServer } from 'ws';
```

```
export class TaskExecutor {
  constructor(
    private prisma: PrismaClient,
    private wss: WebSocketServer
  ) {}

  async executeTask(
    taskId: string,
    command: string,
    workingDir: string
  ): Promise<string> {
    // TODO: Create TaskRun record
    const taskRun = await this.prisma.taskRun.create({
      data: {
        taskId,
        status: 'Running',
        startTime: new Date(),
        logs: '',
      },
    });

    const streamId = taskRun.id;

    try {
      // TODO: Spawn the process
      const process = spawn(/* YOUR CODE */, {
        cwd: workingDir,
        shell: true,
      });

      let logs = '';

      // TODO: Handle stdout
      process.stdout.on('data', (data) => {
        const chunk = data.toString();
        logs += chunk;

        // TODO: Broadcast to WebSocket subscribers
        /* YOUR CODE */;
      });

      // TODO: Handle stderr
      process.stderr.on('data', (data) => {
        /* YOUR CODE */;
```

```
    });

    // TODO: Wait for process to finish
    const exitCode = await new Promise<number>((resolve) => {
      process.on('exit', (code) => {
        resolve(code || 0);
      });
    });

    // TODO: Update TaskRun with final status
    await this.prisma.taskRun.update({
      where: { id: taskRun.id },
      data: {
        status: exitCode === 0 ? 'Completed' : 'Failed',
        endTime: new Date(),
        exitCode,
        logs,
      },
    });

    return streamId;
  } catch (error) {
    // TODO: Handle errors
    await this.prisma.taskRun.update({
      where: { id: taskRun.id },
      data: {
        status: 'Failed',
        endTime: new Date(),
        logs: String(error),
      },
    });
    throw error;
  }
}

private broadcastTaskStream(streamId: string, event: string, data: string) {
  const message = JSON.stringify({
    type: 'task-stream',
    streamId,
    event,
    data,
    timestamp: new Date().toISOString(),
  });
```

```
    // TODO: Send to all WebSocket clients
    this.wss.clients.forEach((client) => {
      // TODO: Check if client is subscribed to this stream
      if (/* YOUR CODE */ && client.readyState === 1) {
        client.send(message);
      }
    });
  }
}
```

Test it:
```bash
# Create a task first, then:
curl -X POST http://localhost:3000/api/tasks/:id/run

# Should return: { taskRunId: "...", status: "Running" }
```

Prompt 4: Wire Task Execution to API

Your task:
Add `/api/tasks/:id/run` endpoint that calls TaskExecutor.

Requirements:
- Find the task by ID
- Validate it exists
- Call taskExecutor.executeTask()
- Return the stream ID
- Handle errors

Code template:
```typescript
app.post('/api/tasks/:id/run', async (req, res) => {
  try {
    const { id } = req.params;

    // TODO: Find the task
    const task = /* YOUR CODE */;

    if (!task) {
      return res.status(404).json({ error: 'Task not found' });
```

```
    }

    // TODO: Get project details
    const project = /* YOUR CODE */;

    // TODO: Get settings (for allowlist)
    const settings = /* YOUR CODE */;

    // TODO: Create TaskExecutor instance
    const taskExecutor = new TaskExecutor(prisma, wss);

    // TODO: Execute task
    const streamId = await taskExecutor.executeTask(
      task.id,
      'echo "Task executed"', // Stub command for now
      project.projectRootPath
    );

    res.status(202).json({ taskRunId: streamId, status: 'Running' });
  } catch (error) {
    res.status(500).json({ error: String(error) });
  }
});
```

Phase 3 Checkpoint
Verify you can:
- [ ] Discover agents from OpenClaw config
- [ ] Create a task in the UI
- [ ] Click "Run Task" and see it execute
- [ ] Check the database: TaskRun record created
- [ ] See logs streaming via WebSocket (in browser console)

If stuck:
- Check backend logs: `npm run dev` output
- Verify task command works: `echo "test"` in your shell
- Check WebSocket connection: Open browser DevTools → Network → WS

PHASE 4: Scheduler System
Time: 3-4 days | Difficulty: Medium | Core Skill: Cron scheduling

What You'll Build
A scheduler that:
- Loads jobs from database on startup
- Registers jobs with node-cron
- Executes jobs on schedule
- Records execution history
- Updates UI via WebSocket

Prompt 5: Create Scheduler Service

Your task:
Create `/backend/src/services/scheduler.ts` with a `SchedulerService` class.

Requirements:
- Constructor accepts Prisma and WebSocketServer
- `initialize()`: Load all enabled jobs from DB, register them
- `scheduleJob()`: Register a single job with node-cron
- `executeJob()`: Run a job immediately
- `unscheduleJob()`: Stop a scheduled job

Hint:
- Use `import * as cron from 'node-cron'`
- Use `cron.schedule(cronExpression, callback)`
- Use `cron.nextDate()` to calculate next run time

Code template:
```typescript
// /backend/src/services/scheduler.ts
import * as cron from 'node-cron';
import { PrismaClient } from '@prisma/client';
import { WebSocketServer } from 'ws';

export class SchedulerService {
  private scheduledTasks: Map<string, any> = new Map();

  constructor(
    private prisma: PrismaClient,
    private wss: WebSocketServer
  ) {}

  async initialize() {
    // TODO: Load all enabled jobs from DB
    const jobs = /* YOUR CODE */;
```

```
  // TODO: For each job, call scheduleJob()
  for (const job of jobs) {
    /* YOUR CODE */;
  }

  console.log(`Scheduler initialized with ${jobs.length} jobs`);
}

async scheduleJob(job: any) {
  // TODO: Prevent duplicate schedules
  if (this.scheduledTasks.has(job.id)) {
    this.unscheduleJob(job.id);
  }

  try {
    // TODO: Create cron schedule
    const task = cron.schedule(job.cronExpression, async () => {
      await this.executeJob(job.id);
    });

    this.scheduledTasks.set(job.id, task);

    // TODO: Calculate next run time
    const nextRunAt = /* YOUR CODE */;

    // TODO: Update job in DB with nextRunAt
    await this.prisma.schedulerJob.update({
      where: { id: job.id },
      data: { nextRunAt: new Date(nextRunAt.toString()) },
    });
  } catch (error) {
    console.error(`Failed to schedule job ${job.id}:`, error);
  }
}

async unscheduleJob(jobId: string) {
  // TODO: Stop the cron task
  const task = this.scheduledTasks.get(jobId);
  if (task) {
    task.stop();
    this.scheduledTasks.delete(jobId);
  }
}
```

```
async executeJob(jobId: string) {
  // TODO: Find job in DB
  const job = /* YOUR CODE */;

  if (!job) return;

  try {
    // TODO: Create SchedulerJobRun record
    const jobRun = await this.prisma.schedulerJobRun.create({
      data: {
        jobId,
        status: 'Running',
        startTime: new Date(),
      },
    });

    // TODO: If job type is RunTask, execute the task
    if (job.jobType === 'RunTask' && job.targetTaskId) {
      // TODO: Call your TaskExecutor here
      /* YOUR CODE */;
    }

    // TODO: Update job run to completed
    await this.prisma.schedulerJobRun.update({
      where: { id: jobRun.id },
      data: {
        status: 'Completed',
        endTime: new Date(),
        exitCode: 0,
      },
    });

    // TODO: Broadcast update via WebSocket
    this.broadcastSchedulerUpdate(jobId, 'completed');
  } catch (error) {
    console.error(`Job execution failed: ${jobId}`, error);
    // TODO: Handle error
  }
}

private broadcastSchedulerUpdate(jobId: string, status: string) {
  const message = JSON.stringify({
    type: 'scheduler-update',
```

```
      jobId,
      status,
      timestamp: new Date().toISOString(),
    });

    // TODO: Send to all WebSocket clients
    this.wss.clients.forEach((client) => {
      if (client.readyState === 1) {
        client.send(message);
      }
    });
  }
}
```

Prompt 6: Wire Scheduler to Server

Your task:
Update `/backend/src/server.ts` to:
- Instantiate SchedulerService on startup
- Add API endpoints for job management
- Call `initialize()` after server starts

Requirements:
- Add `POST /api/scheduler` (create job)
- Add `GET /api/scheduler` (list jobs)
- Add `POST /api/scheduler/:id/run-now` (execute immediately)
- Add `PUT /api/scheduler/:id/enable` and `/disable`

Code template (endpoints):
```typescript
const schedulerService = new SchedulerService(prisma, wss);

httpServer.listen(PORT, async () => {
  // TODO: Initialize scheduler after server starts
  await schedulerService.initialize();

  console.log(`Server running on port ${PORT}`);
});

app.post('/api/scheduler', async (req, res) => {
  try {
```

```
    const { projectId, name, jobType, cronExpression, timezone, enabled } = req.body;

    // TODO: Validate inputs

    // TODO: Create job in DB
    const job = /* YOUR CODE */;

    // TODO: If enabled, schedule it
    if (enabled) {
      await schedulerService.scheduleJob(job);
    }

    res.status(201).json(job);
  } catch (error) {
    res.status(500).json({ error: String(error) });
  }
});

app.post('/api/scheduler/:id/run-now', async (req, res) => {
  try {
    const { id } = req.params;

    // TODO: Execute the job immediately
    await schedulerService.executeJob(id);

    res.json({ status: 'running' });
  } catch (error) {
    res.status(500).json({ error: String(error) });
  }
});

app.put('/api/scheduler/:id/enable', async (req, res) => {
  try {
    const { id } = req.params;

    // TODO: Find job
    const job = /* YOUR CODE */;

    // TODO: Update enabled = true
    const updated = /* YOUR CODE */;

    // TODO: Schedule it
    await schedulerService.scheduleJob(updated);
```

```
    res.json(updated);
  } catch (error) {
    res.status(500).json({ error: String(error) });
  }
});
```

Phase 4 Checkpoint
Verify you can:
- [ ] Create a scheduler job (POST /api/scheduler)
- [ ] List jobs (GET /api/scheduler)
- [ ] Click "Run Now" and see it execute
- [ ] Enable/disable a job
- [ ] Check database: SchedulerJobRun created with correct status

---

PHASE 5 & 6: Quick Build (Follow These Patterns)

Phase 5: Chat System (2-3 days)
What to build:
- `POST /api/projects/:id/chat` - Send message
- `GET /api/projects/:id/chat` - Get history
- WebSocket handler for streaming responses

Key pattern:
```typescript
// Save user message
await prisma.chatMessage.create({
  data: { chatSessionId, role: 'User', content: message }
});

// Simulate agent response (later: integrate real agent)
const response = `Response to: "${message}"`;

// Save agent message
await prisma.chatMessage.create({
  data: { chatSessionId, role: 'Agent', content: response }
});

// Broadcast
wss.clients.forEach(client => {
```

```
  client.send(JSON.stringify({ type: 'chat-message', content: response }));
});
```

Phase 6: Files System (2-3 days)
What to build:
- `GET /api/projects/:id/files` - List directory
- `POST /api/files/open` - Open folder in OS

Key pattern:
```typescript
import fs from 'fs';
import path from 'path';

function buildFileTree(dirPath, depth = 0) {
  const files = fs.readdirSync(dirPath);
  return files.map(file => ({
    name: file,
    path: path.join(dirPath, file),
    type: fs.statSync(path.join(dirPath, file)).isDirectory() ? 'dir' : 'file',
    children: /* recursively list subdirs */
  }));
}
```

PHASE 7: UI Wiring (Done in Frontend)

Setup State Management

Install Zustand:
```bash
cd frontend
npm install zustand
```

Create store:
```typescript
// /frontend/src/store/appStore.ts
import { create } from 'zustand';

interface AppState {
  projects: any[];
```

```
  tasks: any[];
  selectedProjectId: string | null;

  setProjects: (projects: any[]) => void;
  setTasks: (tasks: any[]) => void;
}

export const useAppStore = create<AppState>((set) => ({
  projects: [],
  tasks: [],
  selectedProjectId: null,

  setProjects: (projects) => set({ projects }),
  setTasks: (tasks) => set({ tasks }),
}));
```

Update Dashboard Component

Replace "Coming Soon" with real data:
```typescript
// /frontend/src/pages/Dashboard.tsx
import { useEffect } from 'react';
import { useAppStore } from '../store/appStore';

export default function Dashboard() {
  const projects = useAppStore(state => state.projects);
  const setProjects = useAppStore(state => state.setProjects);

  useEffect(() => {
    // Fetch projects
    fetch('/api/projects')
      .then(res => res.json())
      .then(data => setProjects(data));
  }, []);

  return (
    <div className="p-8">
      <h1 className="text-4xl font-bold mb-8">Dashboard</h1>

      {/* Real data instead of hardcoded */}
      <div className="grid grid-cols-3 gap-6">
        {projects.map(project => (
          <div key={project.id} className="bg-gray-800 p-6 rounded-lg">
```

```
            <h3 className="text-xl font-semibold">{project.name}</h3>
            <p className="text-gray-400">{project.description}</p>
          </div>
        ))}
      </div>
    </div>
  );
}
```

TROUBLESHOOTING GUIDE

"Port 3000 already in use"
```bash
lsof -ti:3000 | xargs kill -9
```

"Cannot find module 'prisma'"
```bash
npm install
npx prisma generate
```

"Database locked"
```bash
rm backend/prisma/data.db-*
npx prisma migrate dev --skip-generate
```

"WebSocket connection refused"
Check backend is running: `npm run dev` in backend folder

Task execution doesn't stream logs
Check browser DevTools → Network → WS tab for WebSocket messages

---

NEXT STEPS AFTER COMPLETING ALL PHASES

1. **Deploy to a VPS** (DigitalOcean, Render, Fly.io)
2. **Add authentication** (if sharing with team)
3. **Integrate real OpenClaw agents** (sessions_spawn)

4. **Add notifications** (email on task completion)
5. **Build plugins** (custom integrations)

FINAL CHECKLIST

When you're done, you should be able to:
- [ ] Create projects from the UI
- [ ] Create tasks and drag them across a kanban board
- [ ] Click "Run Task" and see logs stream in real-time
- [ ] Create scheduler jobs and see them execute
- [ ] Chat with agents and save conversations
- [ ] Browse project output files
- [ ] All data persists across app restarts
- [ ] No "Coming Soon" text anywhere

If all boxes checked: **You've built Mission Control!**

Questions? Join The Sprint or office hours.

Ready to build?