Using OpenClaw with Ollama: Building a local data analytics system.

With OpenClaw's channel integrations, the same local system can be extended to interfaces like WhatsApp or Slack, allowing secure access to your processes from familiar environments.

Modern AI processes often rely on cloud APIs. But what if you want a system that runs entirely on your computer, keeps data private, and still supports multi-step agent processes?

With OpenClaw 's channel integrations , the same local system can also be extended to interfaces like WhatsApp or Slack , allowing secure access to your processes from familiar environments.

OpenClaw Ollama User Guide: Building a Local Data Analyst

Step 1: Install OpenClaw

Before building a local data analysis workflow, we need OpenClaw Gateway running on your computer. Think of OpenClaw as the execution layer in this project, receiving requests from the web user interface, loading workspace skills, running local tools (like shell commands and Python scripts), and orchestrating the entire workflow from start to finish.

```
curl -fsSL https://openclaw.ai/install.sh | bash openclaw onboard --install-daem
```

This command installs the OpenClaw CLI, runs the setup wizard to configure the local environment, and sets up the gateway daemon so it can be easily started and stopped. Although we'll be running the gateway in the foreground for this demo, installing the daemon ensures a proper setup and makes troubleshooting easier.

Now, let's confirm everything is working:

```
openclaw doctor openclaw gateway status
```

The OpenClaw doctor reports that OpenClaw has been installed correctly. The OpenClaw gateway status tells you whether the gateway is currently running. At this stage, it might show 'not running', but that's okay. The important thing is that the command works and the installation process is recognized.

If you want detailed instructions on each setting option (channel, authentication, skills, port security), you can refer to the OpenClaw guide for complete step-by-step instructions.

Step 2: Install Ollama

Next, we will set up Ollama, which will act as the local LLM server for this project. OpenClaw will still orchestrate the workflow, but when it needs model information for summarization or inference, it will call the LLMs through Ollama.

Run the following commands:

```
brew install ollama ollama serve ollama pull qwen3:8b
```

The above commands set up the Ollama runtime environment, start the local model server that OpenClaw will communicate with, and download the qwen3:8b model. The example uses qwen3:8b because it provides a good balance between performance and quality for most laptops, but you can choose a different model based on your system resources.

Step 3: Configure OpenClaw

Next, we need to configure OpenClaw to use the local Ollama instance. This ensures that all inference, summarization, and analysis takes place entirely on the machine without calling external APIs.

Create a local configuration folder:

```
mkdir -p .openclaw-local
```

Next, create the file: .openclaw-local/openclaw.json

```
{ "models": { "providers": { "ollama": { "baseUrl": "http://127.0.0.1:11434/v1",
```

The configuration above defines three main components:

1. The baseUrl points to the local API endpoint provided by Ollama, while the api:openai-completions setting enables communication compatible with OpenAI. Registering the model for qwen3:8b specifies its capabilities, including a large 131K context window for handling large datasets and token limits for controlled responses. Because the model runs locally, all cost values ??are set to zero.
2. The agent defaults section controls the model that OpenClaw model agents use for inference. By setting the main model to ollama/qwen3:8b, all agent tasks, such as interpreting prompts, generating summaries, or inferring on data, are automatically routed to the local Ollama model without any external API calls.
3. The configuration tool manages external capabilities such as web search; this feature is disabled to ensure privacy and prevent requests from being sent externally. The fetch tool remains enabled to access limited resources when necessary.

Together, these settings ensure that the entire workflow runs privately on your machine, with OpenClaw handling orchestration and Ollama providing local intelligence.

Step 4: Define workspace skills

In this step, we define workspace skills for OpenClaw to know how to execute our workflow. Instead of relying on a model to plan tool usage, we use command orchestration mode, allowing slash commands to directly trigger local execution commands, making the workflow faster and entirely local.

Create the following SKILL.md file in your workspace:

```
--- name: local-data-analyst description: Local Data Analyst: analyze private loc
```

This skill configuration controls how OpenClaw performs analysis:

1. The header (a structured metadata block at the beginning of the file) defines a user-defined skill called local-data-analyst, which is available as the slash /local-data-analyst command.
2. The slash command is how OpenClaw triggers structured actions from the chat pane or user interface.
3. Setting `command-dispatch:tool` enables command dispatch mode, where OpenClaw directly passes commands to a tool instead of asking the model to decide what to do.
4. With `command-arg-mode: raw`, the entire command sequence is passed intact to the execution engine, ensuring predictable execution.
5. The command to run the `main.py` script locally is this:
    1. Load the dataset and optional context documents.
    2. Use Ollama (qwen3:8b) to reason and summarize.
    3. Create three components, including a chart, a markdown report, and a tool execution trace.

Thus, OpenClaw handles orchestration and execution, while Ollama provides local inference capabilities. In the next step, we will connect this skill to a web interface so that users can upload data and trigger analysis with just one click.

Step 5: Create the web interface

At this point, we have two core components running on the OpenClaw workspace skill and the local model backend (Ollama) providing the inference capabilities. Now, we need a lightweight interface that allows you to upload files and trigger runs without touching the terminal every time.

The web_assistant.py file acts as a simple interface server:

1. Accept file upload,
2. Create a separate executable directory for each execution.
3. Develop an OpenClaw slash command that points to files on the disk.
4. Call the local OpenClaw agent,
5. Waiting for the output,
6. Returns products that are ready for browser preview.

The key design choice here is that the web server never runs the parsing logic itself. It delegates everything to OpenClaw, so the user interface remains simple.

Step 5.1: Building the slash command

The goal here is to create a unique string that OpenClaw can receive as a chat message, for example: /local-data-analyst python3 . --data-file . --output-dir …

```
def build_slash_command( data_path: Path, docs_dir: Path, output_dir: Path, promp
```

The `build_slash_command()` function prepares the exact instructions to be sent to OpenClaw. Instead of performing direct parsing, the web application builds a structured slash command that OpenClaw can route to the appropriate workspace skill. This function performs four main tasks:

1. The list of args defines the CLI call to src/main.py. This is the same command you can run manually from the terminal.
2. The --x-column and --y-column flags are only added when provided. If these values ??are missing, the parsing script can automatically infer the columns from the dataset.
3. The shlex.quote() call encrypts all arguments, including the user's prompt. This is crucial for handling spaces and special characters, and prevents the risk of command injection attacks when passing user input to a shell command.

The function returns a string beginning with: /local-data-analyst. This prefix matches the skill name defined in SKILL.md. When OpenClaw receives this slash command, it immediately forwards the request to the local-data-analyst skill workspace, which then executes the command using the exec tool.

Step 5.2: Execute OpenClaw

Now that we have the slash command, we send it to OpenClaw using the CLI agent runner. This is where the web application transfers execution privileges.

```
slash_message = build_slash_command( data_path=data_path, docs_dir=docs_dir, outp
```

Calling the OpenClaw agent is the point of transition where the web application stops working and instead requires OpenClaw to execute the workflow from start to finish.

1. Using the --local parameter ensures the process runs on your machine and the agent handles requests through a local port and model supported by Ollama, rather than any hosted service.
2. The --session-id stealth-web-{run_id} parameter provides each run with a separate session namespace, preventing state leaks between runs and making it easier to debug a specific execution later.
3. The --message parameter passes the exact slash command string as if the user had typed it into the chat box; OpenClaw receives it, routes it to the /local-data-analyst skill, and executes the underlying command via the exec tool.
4. Finally, --timeout 120 acts as a safety valve to prevent the web UI from permanently hanging if the parsing process stalls, and env=openclaw_env() forces the child process to use the project's local OpenClaw configuration and state directory so that it always targets the intended Ollama setup.

In the next step, we'll load the generated artifacts, including charts, reports, and tool traces, and display a lightweight preview in the browser.

Step 5.3: Server Setup

Finally, web_assistant.py runs a small local HTTP server so you can interact with it through your browser.

```
def main() -> int: host = "127.0.0.1" port = 8765 server = ThreadingHTTPServer(
```

This simplifies the deployment process:

1. ThreadingHTTPServer allows multiple requests without blocking the entire application.
2. All the "actual work" takes place in the Request Handler, receiving uploaded files, creating the run directory, activating OpenClaw, and returning a preview.

Note : The complete source code for web_assistant.py is available in the project's GitHub repository.

Step 6: Build a local analysis tool.

At this stage, OpenClaw can execute the workflow through the workspace skill, and the web user interface can trigger runs via the slash command. The rest is handled by the analysis tool, which receives the files you've uploaded, runs the workflow steps, and generates artifacts.

The main.py file in this repository focuses on two core functions that define the process: loading tabular data and calling Ollama for local inference.

Step 6.1: Load data in table format

This helper supports multiple input formats while maintaining a consistent workflow.

```
def load_tabular_data(data_path: Path, events: List[SkillEvent]) -> pd.DataFrame
```

The `load_tabular_data()` function detects the file type using `data_path.suffix` and redirects it to the correct pandas loader. CSV and TSV files are processed using `read_csv()`, where TSV/tab files simply convert the delimiter to `t`. JSON input is processed first using `read_json()`, and if pandas throws a ValueError, it will convert lines=True for JSONL. Excel support is added via `read_excel()` so users can upload .xlsx files without prior processing.

Finally, the log_event() call records a structured trace entry whose path can then be serialized to tool_trace.json.

Step 6.2: Integrate Ollama

Because this demo doesn't rely on the SDK, it directly calls Ollama's local HTTP API.

```
def ollama_generate(model: str, prompt: str) -> str: url = "http://localhost:114
```

The `ollama_generate()` function sends a JSON payload to the Ollama `/api/generate` endpoint on localhost. This payload specifies the model name (e.g., `qwen3:8b`), the final prompt string, and disables data transmission so the function returns a single complete response.

Using `urllib.request` makes this wrapper lightweight and portable, and the `timeout=45` condition prevents our workflow from hanging indefinitely if the model is slow or the server fails. Finally, the function extracts the model's output from the response field and returns clean text, which is then used to write to the `analysis_report.md` file.

Step 7: Build the web server

This is a small launch script to start a local web user interface, accept uploads, and trigger background OpenClaw runs.

```
set -euo pipefail ROOT="$(cd "$(dirname "$0")" && pwd)" cd "$ROOT/." python3 ./w
```

This script performs three tasks:

1. `set -euo pipefail` causes the script to stop quickly, meaning it stops when it encounters an error, treating unset variables as errors and avoiding silent errors in pipelines.
2. ROOT=. resolves the directory where the script is located, so it works even if you run it from a different location.
3. cd "$ROOT/." navigates to the desired project root directory and then launches web_assistant.py, which contains the user interface and handles the entire pipeline.

After this script runs, your browser's user interface will become the primary gateway to the demo.

Step 8: Start the system

For the final step, we run the system using a two-process setup. The OpenClaw gateway handles all task execution, while the web interface acts as the user layer to send requests and view the generated results.

Terminal 1: Gateway

Before launching the interface, we first start the OpenClaw gateway. This process acts as the system's execution layer, handling agent requests, loading workspace skills, calling local tools, and routing inference calls to the Ollama model.

```
export OPENCLAW_CONFIG_PATH="$PWD/.openclaw-local/openclaw.json" openclaw gatewa
```

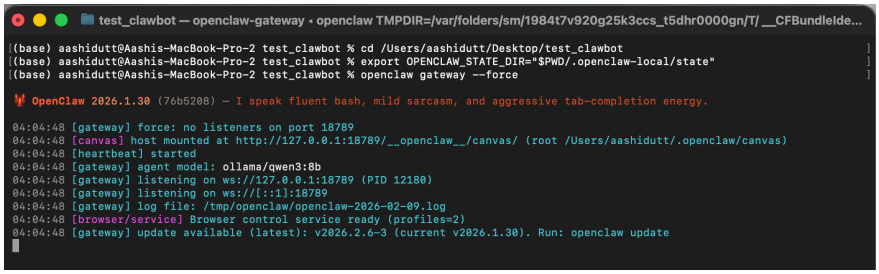
In this terminal, OPENCLAW_CONFIG_PATH points OpenClaw to the project's local configuration, where we've pinned the default model to ollama/qwen3:8b and disabled web search for security. Next, openclaw gateway --force starts the gateway even if OpenClaw assumes something is already running or partially configured.

Once the gateway is set up, it's ready to accept messages from the local agent (including our /local-data-analyst command).

Terminal 2: Web User Interface

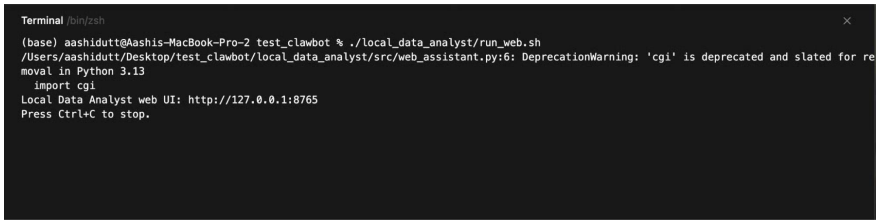
Once the connection is established, we launch the web interface, which collects user input, sends each request to the local OpenClaw agent, and displays the generated graphs, reports, and execution traces.

```
./local_data_analyst/run_web.sh
```



Then open:

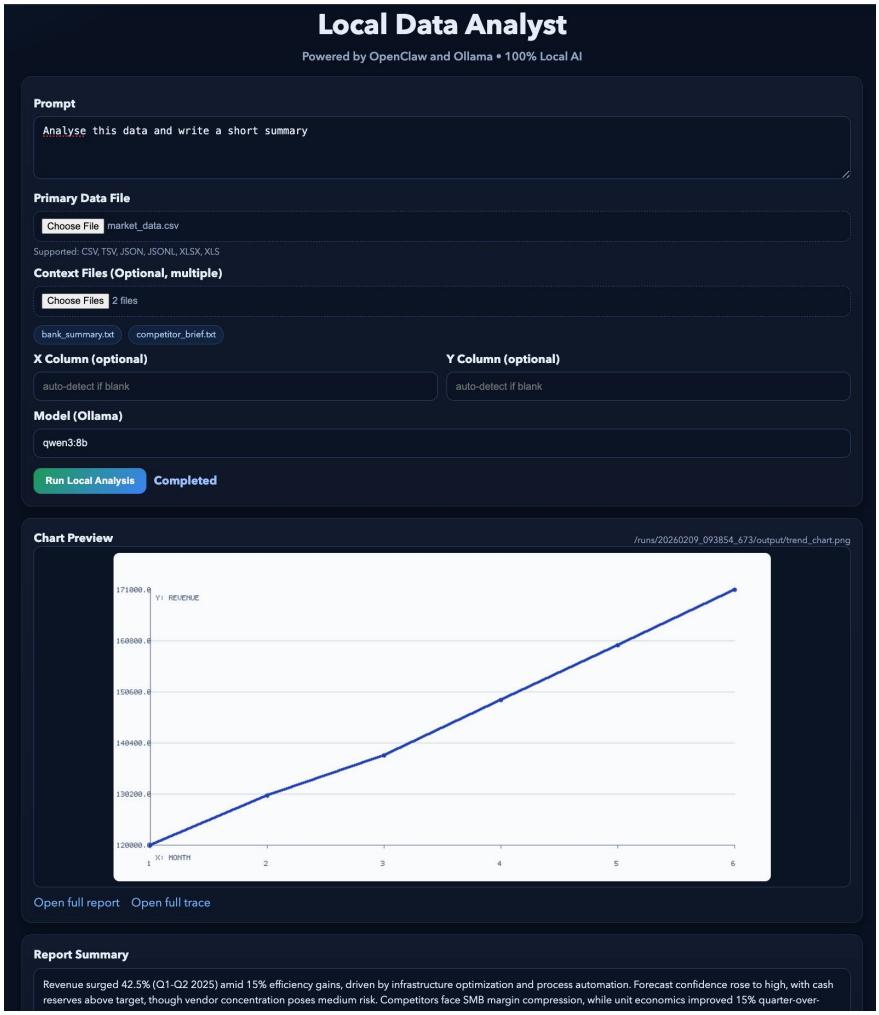
`http://127.0.0.1:8765`



The web server runs on 127.0.0.1, so it's only accessible from your machine. When you click Run Analysis, the user interface will write a runtime directory, build a slash command, call the openclaw agent --local, and then probe the drive for output files to preview:

1. trend_chart.png
2. analysis_report.md
3. tool_trace.json

The final result will look like this. You can test this demo with some example files .



You finished reading the article "**Using OpenClaw with Ollama: Building a local data analytics system.**" edited by the **TipsMake** team. We hope this article has provided you with many useful tech tips and tricks. You can search for similar articles on tips and guides. Thank you for reading and for following us regularly.