Build an AI Agent From Scratch

A practical, code-first introduction

Zardar Khan · Sunnybrook Research Institute

What We're Building

An agent that can read files, extract structured data, and answer questions using natural language.

> *"How much am I spending on gas?"*
>
> → Agent reads CSV, writes SQL, returns: $134.70

Same pattern applies to: clinical reports, DICOM metadata, pathology notes, registry forms

KEY CONCEPTS

Tool Calling
LLM decides which function to invoke

Tool Loop
Iterate until task is complete

Structured Output
Constrain LLM to return valid schemas

Composability
Build complex behavior from simple tools

Start Simple: Chat with Memory

```
# claudette: lightweight wrapper for Anthropic SDK
from claudette import *

chat = Chat(models[2], sp="You are a helpful assistant.")
chat("I'm Zardar")
→ "Nice to meet you, Zardar!"

chat("What's my name?")
→ "Your name is Zardar, as you just told me!"
```

Chat maintains conversation history

Each call appends to the message list. The model sees the full context.

System prompt sets behavior

The sp parameter defines the assistant's role and constraints.

Give Claude Abilities: Tools

1. DEFINE A PYTHON FUNCTION

```
def get_customer_info(
    customer_id: str # ID of customer
) -> dict:
    "Retrieves customer details"
    return customers[customer_id]
```

2. PASS TO CHAT

```
chat = Chat(mdl, tools=[get_customer_info])
```

HOW IT WORKS

Claudette uses Python reflection to extract:

- Function name, parameter types and names
- Docstrings and inline comments
- Return type annotations

This becomes a JSON schema sent to the API.

Claude reads it and decides *when* and *how* to call your function.

⚡ **Key insight**

Good docstrings = better tool selection

Tool Calling in Action

```
r = chat('Can you tell me the email for customer C1?')
print(r.stop_reason) # → 'tool_use'
r.content
```

```
# Claude's response:
[ToolUseBlock(
    name='get_customer_info',
    input={'customer_id': 'C1'}
)]
```

stop_reason = 'tool_use'

Claude is requesting we run a tool, not giving a final answer.

ToolUseBlock

Contains which function to call and with what arguments.

We execute it

Our code runs the function, returns result to Claude.

The model doesn't execute code. It returns structured instructions. You control what actually runs.

The Tool Loop: Automate the Back-and-Forth

THE LOOP

USER: "Get customer C1's email"

↓

MODEL: tool_use get_customer_info(C1)

↓

EXECUTE: returns {email: "john@..."}

↓

MODEL: end_turn "The email is john@..."

ONE LINE DOES IT ALL

```
r = chat.toolloop(prompt)

# Handles the entire cycle:
# 1. Send prompt to model
# 2. If tool_use then execute function
# 3. Send result back to model
# 4. Repeat until end_turn
```

Multi-step reasoning

"Cancel order 03 and confirm status" calls cancel_order, then get_order_details, then responds.

Error recovery

If a tool fails, model sees the error and can try a different approach.

CHECKPOINT

So far we've learned:

1

Chat

maintains conversation history

2

Tools

give Claude real-world abilities

3

Toolloop

automates multi-step execution

Now let's compose these into something useful.

Building Blocks: File Operations

SIMPLE TOOLS, BIG IMPACT

```python
def list_files(fp: str) -> L:
    "List files in directory"
    return L(Path(fp).iterdir())
```

```python
def read_file(fname: str) -> str:
    "Read text content of file"
    return Path(fname).read_text()
```

3 lines each. The model handles all the decision-making.

IN ACTION

"What's in the smb data folder?"

→ list_files('drive/.../smb')

Returns: chq-nov24.txt, cc-nov24.txt, sav-nov24.txt

Model reasons about file names:

"chq = chequing, cc = credit card, sav = savings... these appear to be bank statement files organized by account type."

The power of composability

With just list_files + read_file, the model can explore any directory structure and understand its contents.

Structured Outputs: From Text to Data

DEFINE THE SHAPE YOU WANT

```python
from pydantic import BaseModel, Field

class StatementMetadata(BaseModel):
    source_file: str
    bank_name: str
    account_type: str
    account_holder: str
    opening_balance: Optional[float]
    closing_balance: Optional[float]
```

Pydantic = Python's data validation library. Define types, get validation free.

THE API CONSTRAINS OUTPUT

```python
response = client.beta.messages.parse(
    model="claude-sonnet-4-5",
    output_format=StatementMetadata,
    messages=[...]
)

response.parsed_output # ← typed object
```

Why this matters

No regex parsing of LLM output

Guaranteed valid JSON matching schema

Direct integration with your data pipeline

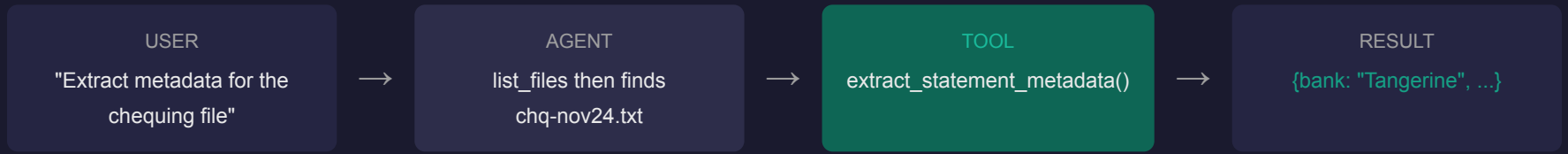
Same pattern for: ICD codes, RadLex terms, DICOM fields, pathology staging...

Tools Can Call Models

A tool can itself use an LLM for complex processing. The outer agent orchestrates; the inner call extracts.

```python
def extract_statement_metadata(filepath: str) -> dict:
    "Extract structured metadata from bank statement file."

    text = Path(filepath).read_text()
    response = client.beta.messages.parse(
        output_format=StatementMetadata,
        messages=[{"content": f"Extract metadata:\n{text}"}]
    )
    return response.parsed_output.model_dump()
```



Lego blocks. Each tool is self-contained. The orchestrating agent doesn't know (or care) that extraction uses an LLM internally.

The Full Toolkit

FILE OPERATIONS

`list_files()`
Browse directories

`read_file()`
Read text content

`save_to_csv()`
Persist structured data

EXTRACTION

`parse_transactions()`
Text → structured records

`categorize_transactions()`
Add category labels

`extract_metadata()`
Header info → schema

ANALYSIS

`inspect_csv()`
Preview shape & columns

`read_csv()`
Load full data

`query_csv()`
SQL via DuckDB ⚡

9 tools, ~100 lines total. The model decides which to use, in what order, based on the user's natural language query.

System prompt guides behavior:

```
"You are a bookkeeping assistant. Data is in 'drive/.../smb'. Always start by listing files."
```

Natural Language → SQL → Answer

"How much am I spending on healthcare and where?"

Model generates SQL:

```
SELECT SUM(amount) as total_healthcare_spending
FROM data
WHERE category LIKE '%Healthcare%' OR category LIKE '%Health%' OR subcategory LIKE '%Healthcare%' OR subcategory LIKE '%Health%' OR subcategory LIKE '%Pharmacy%' OR subcategory LIKE '%Medical%'
```

→

DuckDB returns:

```
-82.30
```

→

Model responds:

Total Healthcare Spending: $82.30 (November 2024)

You made **4 healthcare-related purchases** at 2 different locations:

Date	Location	Amount
Nov 4	Hone Fitness, Toronto, ON	$22.54
Nov 10	Hone Fitness, Toronto, ON	$22.54
Nov 10	Shoppers Drug Mart #12, Toronto, ON	$14.68
Nov 28	Hone Fitness, Toronto, ON	$22.54

Breakdown by location:

- **Hone Fitness** (Toronto): 3 visits, $67.62 total
- **Shoppers Drug Mart #12** (Toronto): 1 visit, $14.68 total

Your healthcare spending is primarily on fitness (Hone Fitness membership/services), with one pharmacy purchase at Shoppers Drug Mart.

```python
def query_csv(filepath: str, sql: str) -> str:
    "Execute SQL query on CSV using DuckDB. Table is named 'data'."
    df = duckdb.query(f"SELECT * FROM '{filepath}'").df()
    result = duckdb.query(sql).df()
    return result.to_markdown()
```

DuckDB = in-process SQL

No server. Reads CSVs directly. Fast.

Read-only by design

Model generates SELECT only. Safe.

Self-correcting

Wrong column name? Model retries with schema.

Multi-Step Reasoning

"How much am I spending on coffee AND takeout?"

1. query_csv — tries
   Returns empty. Categories don't match.
2. query_csv —
   Discovers: Dining, Entertainment, Financial, Groceries...
3. query_csv — checks subcategories under 'Dining'
   Finds: Starbucks, Uber Eats, 6IXSIDE BURGER, Hero Burger...
4. query_csv — final aggregation with CASE statement
   Coffee (Starbucks): $52.33 · Takeout/Dining: $218.63

The model explored the schema when its first attempt failed. No hard coded logic. It figured out the data structure and adapted.

> "You're spending **$270.96** on coffee and takeout combined."

Why Build Agents?

TRADITIONAL SCRIPT

You write:

- Exact file paths
- Regex for each format variation
- Column name mappings
- Error handling for each edge case
- Hardcoded category mappings

New bank format? Rewrite the parser.

AGENT APPROACH

You provide:

- Simple, composable tools
- Output schemas (Pydantic)
- High-level instructions

Model handles:

- ✓ File discovery & format inference
- ✓ Schema exploration
- ✓ Query construction & retry

New bank format? Just works.

Leverage intelligence, not just automation. The model's reasoning handles the variability you'd otherwise hardcode.

Key Takeaways

- **Agents = LLM + Tools + Loop**
  The model decides what to call; you control what runs.
- **Build small, compose big**
  Simple 3-line tools combine into powerful workflows.
- **Structured outputs eliminate parsing**
  Pydantic schemas guarantee valid, typed data.
- **Good docstrings = good tool selection**
  The model reads your documentation. Write it well.

This entire demo: exploration, iteration, and final run — under $5 in API credits.

Resources

Claudette Library

github.com/AnswerDotAI/claudette

Anthropic Tool Use Docs

docs.anthropic.com/en/docs/build-with-claude/tool-use

This Notebook

Available on request