LLM building blocks

Using Tool Use and Structured Output

What are Tools?

CW
JV

- **Functions that AI models can call** to perform specific tasks
- **Extend model capabilities** beyond text generation
- **Bridge between AI and the real world** (APIs, databases, calculations)
- **Enable more reliable and accurate responses**

Examples of Tools

- Calculator functions (add, multiply, square root)
- Web search and data retrieval
- Database queries and updates
- File system operations
- API calls to external services

Why Use Tools?

CW
JV

Without Tools

- **Hallucination risk**: AI might make up facts
- **Limited capabilities**: Only what's in training data
- **Unreliable calculations**: Math errors are common
- **No real-time data**: Can't access current information

With Tools

- **Factual accuracy**: Tools provide real data
- **Extended capabilities**: Access to any API or service
- **Reliable calculations**: Delegate math to calculators
- **Real-time information**: Live data access

The Key Insight

Tools let AI models **acknowledge what they don't know** and **use the right tool for the job**.

LangChain Tool Creation

The `@tool` Decorator

```python
from langchain_core.tools import tool

@tool
def add(a: float, b: float) -> float:
    """Add two numbers together."""
    return a + b

@tool
def search_web(query: str) -> str:
    """Search the web for information."""
    # Implementation here
    return search_results
```

Key Requirements

- **Function name** becomes tool name
- **Docstring** describes what it does
- **Type hints** define parameters
- **Return type** specifies output

Inputs can be structured

Advanced Tool Definition

```python
from pydantic import BaseModel, Field

class SearchInput(BaseModel):
    """Input for web search tool"""
    query: str = Field(description="Search query")
    max_results: int = Field(
        default=5,
        description="Maximum number of results"
    )

@tool("web_search", args_schema=SearchInput)
def search_web(query: str, max_results: int = 5) -> str:
    """Search the web for current information."""
    # Implementation here
    return search_results
```

Calculator Agent Example

Defining Calculator Tools

```python
@tool
def add(a: float, b: float) -> float:
    """Add two numbers together."""
    return a + b

@tool
def multiply(a: float, b: float) -> float:
    """Multiply two numbers together."""
    return a * b

@tool
def square_root(number: float) -> float:
    """Calculate the square root of a number."""
    if number < 0:
        raise ValueError("Cannot calculate square root of negative r
    return math.sqrt(number)

CALCULATOR_TOOLS = [add, multiply, square_root]
```

Binding Tools to Model

```python
from langchain_google_genai import ChatGoogleGenerativeAI

def create_calculator_agent():
    # Initialize the LLM
    llm = ChatGoogleGenerativeAI(
        model="gemini-2.5-flash-preview-04-17",
        temperature=0
    )

    # Bind tools to the model
    llm_with_tools = llm.bind_tools(CALCULATOR_TOOLS)

    return llm_with_tools
```

Tool Execution Flow

CW JV

1. Send Prompt with Tools

```
agent = create_calculator_agent()

prompt = "What is the square root of 144?"

response = agent.invoke([
    HumanMessage(content=prompt)
])
```

2. Model Decides to Use Tools

```
# Response contains tool calls
if response.tool_calls:
    for tool_call in response.tool_calls:
        print(f"Tool: {tool_call['name']}")
        print(f"Args: {tool_call['args']}")
        # {'name': 'square_root', 'args': {'number': 144}}
```

3. Execute Tools & Continue

```
for tool_call in response.tool_calls:
    tool_name = tool_call["name"]
    tool_args = tool_call["args"]

    # Find and execute the tool
    result = tool_function.invoke(tool_args)

    # Message to send back to AI
    tool_message = ToolMessage(
        content=str(result),
        tool_call_id=tool_call["id"]
    )

final_response = agent.invoke([
    HumanMessage(content=prompt),
    response,
    tool_message
])
```

Structured Output Basics

- **Predictable data formats** instead of free-form text
- **Type safety** and validation
- **Easier integration** with other systems
- **Reduced parsing errors**

Common Approaches

- **JSON Schema** - Define expected structure
- **Pydantic Models** - Python classes with validation
- **TypedDict** - Lightweight type definitions
- **Tool Schemas** - Tools that return structured data

Pydantic for Structured Output

CW
JV

Define Your Schema

```python
from pydantic import BaseModel, Field
from typing import List

class JobExtraction(BaseModel):
    """Extracted job posting information"""
    title: str = Field(description="Job title")
    company: str = Field(description="Company name")
    salary_range: str = Field(
        description="Salary range if mentioned"
    )
    requirements: List[str] = Field(
        description="List of job requirements"
    )
    location: str = Field(description="Job location")
    remote_ok: bool = Field(
        description="Whether remote work is allowed"
    )
```

Use with LangChain

```python
from langchain_google_genai import ChatGoogleGenerativeAI

llm = ChatGoogleGenerativeAI(
    model="gemini-2.5-flash-preview-04-17"
)

# Create structured output model
structured_llm = llm.with_structured_output(JobExtraction)

# Now responses are guaranteed to match schema
job_data = structured_llm.invoke(
    "Extract job info from this posting: ..."
)

# job_data is a JobExtraction object
print(job_data.title)  # Guaranteed to exist
print(job_data.salary_range)  # Type-safe access
```

Tools vs Structured Output

Use `bind_tools()` When:

- **Model should choose** whether to use tools
- **Multiple tools available** for different tasks
- **Agent-like behavior** where model decides actions
- **Interactive workflows** with multiple steps

Use `with_structured_output()` When:

- **Always need structured data** from the model
- **Information extraction** tasks
- **Consistent data format** required
- **Integration with APIs** or databases

Real-World Examples

Calculator Agent

Problem: Reliable math calculations **Solution**: Mathematical operation tools

```python
@tool
def compound_interest(
    principal: float,
    rate: float,
    time: float
) -> float:
    """Calculate compound interest."""
    return principal * (1 + rate) ** time

# Usage: "Calculate compound interest on $1000
# at 5% for 10 years"
```

Job Extractor

Problem: Parse job postings consistently **Solution**: Structured output schema

```python
class JobPosting(BaseModel):
    title: str
    company: str
    requirements: List[str]
    salary_min: Optional[int]
    salary_max: Optional[int]
    remote_allowed: bool

# Always get clean, structured job data
# for database insertion
```

Best Practices

- **Tool design**: Keep tools simple and focused on one task
- **Clear documentation**: Docstrings are crucial for AI understanding
- **Type hints**: Always use proper type annotations
- **Error handling**: Tools should handle edge cases gracefully
- **Validation**: Use Pydantic for robust data validation
- **Testing**: Test tools independently before AI integration

Common Pitfalls

- **Too complex tools**: AI struggles with multi-purpose functions
- **Poor descriptions**: Unclear docstrings lead to misuse
- **Missing validation**: Unhandled errors break workflows

Key Takeaways

- **Tools extend AI capabilities** beyond text generation
- `@tool` **decorator** makes function definition simple
- `bind_tools()` gives models access to tools
- **Structured output** ensures predictable data formats
- **Good documentation** is essential for tool success
- **Test tools independently** before AI integration
- **Choose the right approach** for your use case

Remember

Tools and structured output are **fundamental building blocks** for reliable AI applications. Mastering these patterns increases what you can build with AI.

Build Something Amazing!