

Tool Calling Uitgelegd

Hoe Claude Sonnet 4 Tools Aanroept

Guest Search System

Communicatie tussen Model, Prompts & Tools

Overzicht: Wat is Tool Calling?

Tool Calling = LLM kan externe functies aanroepen tijdens conversatie

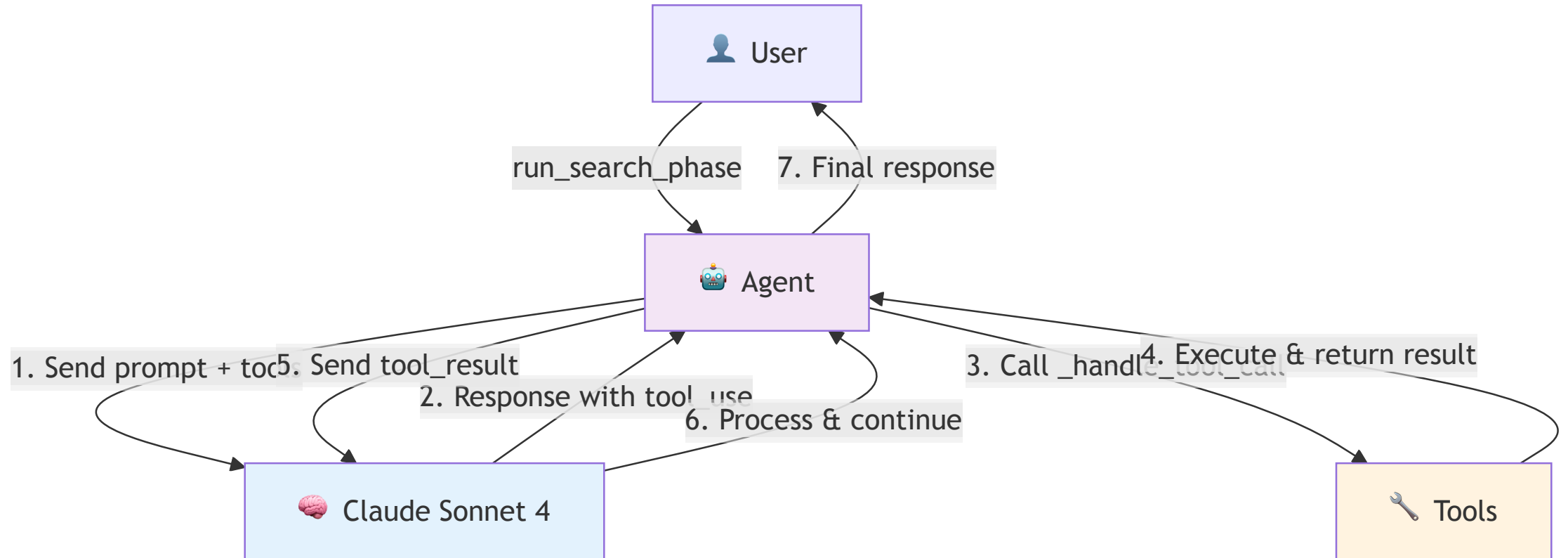
In ons systeem:

- Claude Sonnet 4 = Het brein
- Tools = De handen (web_search, fetch_page_content, etc.)
- `_handle_tool_call()` = De vertaler tussen brein en handen

Waarom belangrijk:

- LLM kan geen direct web zoeken
- LLM kan geen files lezen/schrijven
- Tools geven LLM deze mogelijkheden

De Complete Flow



Stap 1: Agent Stuert Prompt + Tools

Code:

```
response = self.client.messages.create(  
    model="claude-sonnet-4-20250514",  
    max_tokens=8000,  
    tools=self.tools, # ← Tool definities  
    messages=conversation # ← Conversatie geschiedenis  
)
```

Wat gebeurt er:

1. Agent roept Claude API aan
2. Stuurt prompt: "Zoek AI experts in Nederland"
3. Stuurt tool definities mee (JSON schema)

Tool Definitions

JSON Schema:

```
{
  "name": "web_search",
  "description": "Search web",
  "input_schema": {
    "type": "object",
    "properties": {
      "query": {
        "type": "string",
        "description": "Query"
      }
    },
    "required": ["query"]
  }
}
```

Claude leert:

Stap 2: Claude Beslist Tool Te Gebruiken

Claude's response:

```
{
  "content": [
    {
      "type": "text",
      "text": "Ik ga zoeken naar AI experts in Nederland..."
    },
    {
      "type": "tool_use",
      "id": "toolu_01ABC123",
      "name": "web_search",
      "input": {
        "query": "AI experts Nederland 2025"
      }
    }
  ]
}
```

Stap 3: Agent Detecteert Tool Call

Code in `run_search_phase()` :

```
for block in response.content:
    if block.type == "text":
        # Gewone tekst van Claude
        assistant_message["content"].append(block)

    elif block.type == "tool_use":
        # 🎯 Claude wil een tool gebruiken!
        result = self._handle_tool_call(
            block.name,      # "web_search"
            block.input,     # {"query": "..."}
            silent=True
        )
```

Wat gebeurt:

1. Loop door alle content blocks

Stap 4: `_handle_tool_call()` Voert Tool Uit

De vertaler functie:

```
def _handle_tool_call(self, tool_name, tool_input, silent=False):  
    """Verwerk tool calls van de agent"""  
  
    if tool_name == "web_search":  
        query = tool_input["query"]  
  
        # Gebruik SmartSearchTool  
        search_result = self.smart_search.search(  
            query,  
            num_results=10  
        )  
  
        # Format resultaten voor Claude  
        return {  
            "results": [  
                {  
                    "title": r.get("title", ""),  
                    "snippet": r.get("snippet", ""),  
                    "url": r.get("link", "")  
                }  
                for r in search_result["results"]  
            ],  
            "tool_name": tool_name, "tool_input": tool_input, "tool_output": search_result["results"]  
        }
```


_handle_tool_call: Alle Tools

1. web_search

- Roept SmartSearchTool aan
- Multi-provider fallback
- Returns: lijst met results

2. fetch_page_content

- Haalt HTML op met requests
- Parse met BeautifulSoup
- Returns: schone tekst (max 4000 chars)

3. check_previous_guests

- Checkt previous_guests.json

Stap 5: Tool Result Terug Naar Claude

Code:

```
# Voeg tool_use toe aan conversatie
assistant_message["content"].append(block)
conversation.append(assistant_message)

# Voeg tool_result toe
conversation.append({
    "role": "user",
    "content": [{
        "type": "tool_result",
        "tool_use_id": block.id,
        "content": json.dumps(result)
    }]
})
```

Conversatie wordt:

Assistant: Ik ga zoeken [tool_use: web_search]

Stap 6: Claude Verwerkt Result

Claude ontvangt:

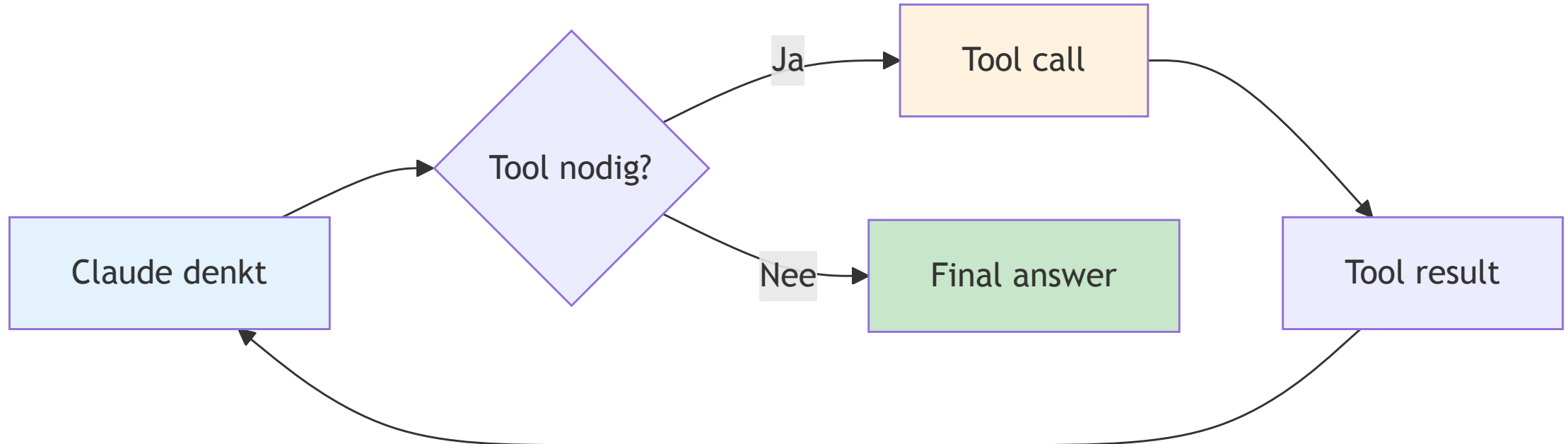
```
{  
  "type": "tool_result",  
  "tool_use_id": "toolu_01ABC123",  
  "content": "{\\"results\\": [{\\"title\\": \\"...\\", ...}]}"  
}
```

Claude kan nu:

1. Resultaten analyseren
2. Beslissen: meer tools nodig?
3. Of: antwoord geven aan user

Dit heet "agentic loop"

De Agentic Loop



Voorbeeld sequence:

1. Claude: "Ik zoek eerst op Google" → tool_use
2. Result: 10 URLs gevonden
3. Claude: "Ik fetch de eerste URL" → tool_use

Real Example: Een Complete Cyclus

1. User prompt:

"Zoek Nederlandse AI experts"

2. Claude's eerste response:

```
{  
  "type": "tool_use",  
  "name": "web_search",  
  "input": {"query": "Nederlandse AI experts 2025"}  
}
```

3. Tool result:

```
{  
  "results": [  
    {"title": "Dr. Sarah AI Expert", "url": "https://..."}  
  ]  
}
```

Real Example (vervolg)

5. Tool result:

```
{  
  "content": "Dr. Sarah Veldman is AI researcher at TN0..."  
}
```

6. Claude's derde response:

```
{  
  "type": "tool_use",  
  "name": "check_previous_guests",  
  "input": {"name": "Dr. Sarah Veldman"}  
}
```

7. Tool result:

```
{  
  "already_recommended": false
```

Conversatie Geschiedenis

Waarom important?

- Claude moet context bewaren
- Volgende tool call gebruikt eerdere results
- Staat in `conversation` list

Structuur:

```
conversation = [  
    {"role": "user", "content": "Zoek AI experts"},  
    {"role": "assistant", "content": [tool_use_1]},  
    {"role": "user", "content": [tool_result_1]},  
    {"role": "assistant", "content": [tool_use_2]},  
    {"role": "user", "content": [tool_result_2]},  
    # etc...  
]
```

Silent Mode vs Verbose

In onze code:

```
result = self._handle_tool_call(  
    block.name,  
    block.input,  
    silent=True # ← Geen print statements  
)
```

Waarom silent?

- Zoek fase: veel tool calls (10-50x)
- Terminal zou vol staan met logs
- Progress bar is genoeg feedback

Alternatief: verbose mode voor debugging

Error Handling in Tools

Voorbeeld: fetch_page_content

```
try:
    response = requests.get(url, timeout=10)
    if response.status_code == 200:
        # Parse en return content
        return {"content": text, "status": "success"}
    else:
        return {"error": f"HTTP {response.status_code}",
                "status": "error"}
except Exception as e:
    return {"error": str(e), "status": "error"}
```

Claude krijgt error info:

- Kan beslissen: skip deze URL
- Of: probeer andere tool

Waarom Deze Architectuur?

Voordelen:

Voor LLM:

- Blijft stateless
- Geen side effects
- Deterministisch

Voor Code:

- Tools zijn testbaar
- Easy te debuggen
- Modulair design

Voor Ons:

Tool Chain Example

Scenario: Guest zoeken met LinkedIn

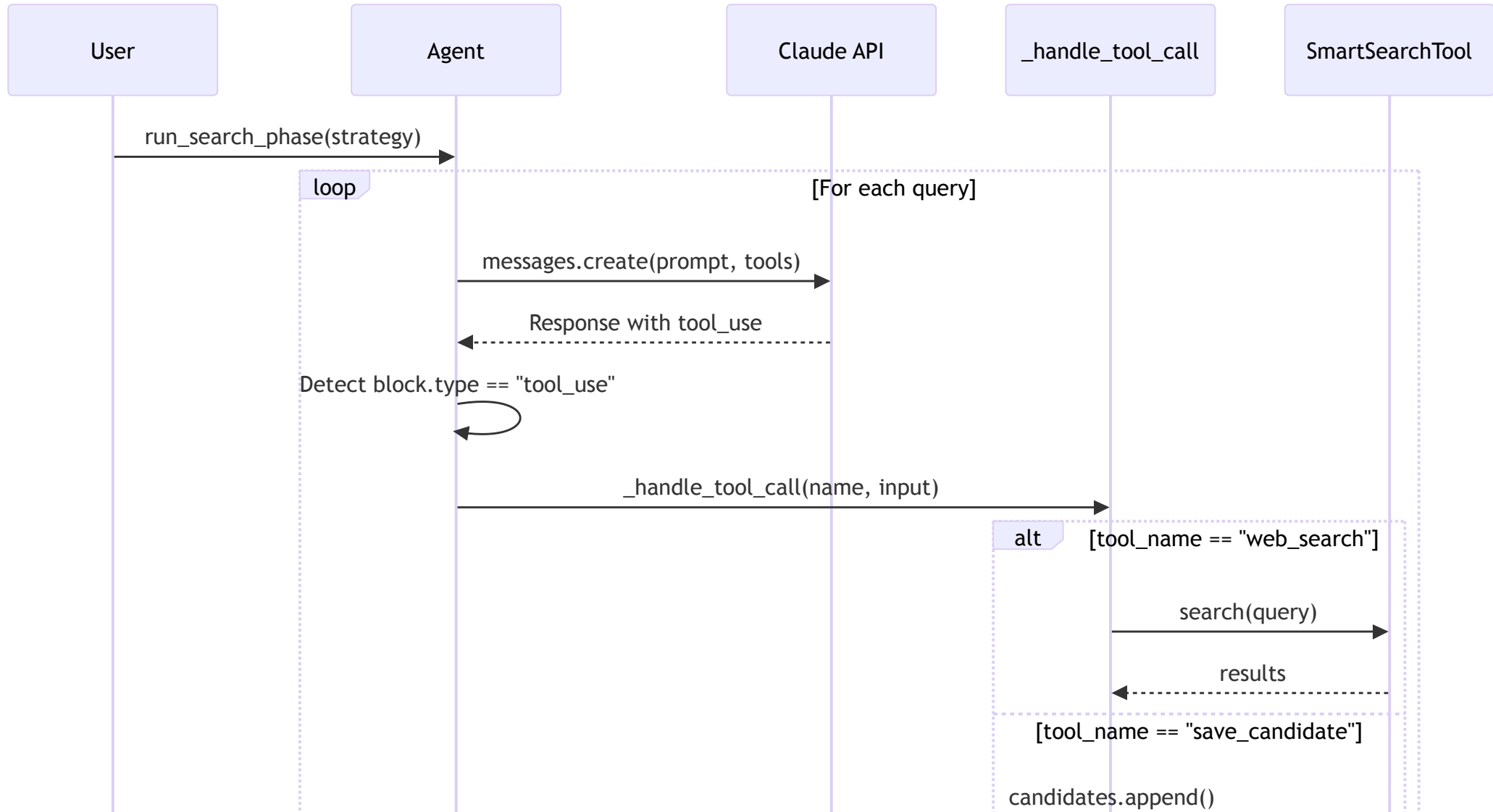


Elke tool:

- Voegt info toe
- Valideert data
- Bereidt voor voor volgende stap

Claude orchestreert de chain

Code Flow Diagram



Performance Optimizations

In ons systeem:

1. Silent mode tijdens loops

```
result = self._handle_tool_call(..., silent=True)
```

2. Caching in SmartSearchTool

- 1-day cache voor search results
- Scheelt API calls & kosten

3. Content truncation

```
if len(text) > 4000:  
    text = text[:4000] + "\n\n[...truncated...]"
```

4. Progress tracking

Testing _handle_tool_call

Unit test voorbeeld:

```
def test_web_search_tool():  
    agent = GuestFinderAgent()  
  
    result = agent._handle_tool_call(  
        tool_name="web_search",  
        tool_input={"query": "test query"}  
    )  
  
    assert "results" in result  
    assert isinstance(result["results"], list)  
    assert "provider" in result
```

Elke tool heeft:

- Unit tests (gemockt)
- Integration tests (real API)

Extending: Nieuwe Tool Toevoegen

Stap 1: Definieer tool







```
{
  "name": "find_email",
  "description": "Find email address for a person",
  "input_schema": {
    "type": "object",
    "properties": {
      "name": {"type": "string"},
      "company": {"type": "string"}
    },
    "required": ["name", "company"]
  }
}
```

Stap 2: Implementeer in _handle_tool_call




```
elif tool_name == "find_email":
    name = tool_input["name"]
```

Best Practices

Do's:

-  Return consistent JSON format
-  Handle errors gracefully
-  Validate input parameters
-  Log voor debugging (optioneel)
-  Timeout voor externe calls
-  Cache waar mogelijk

Don'ts:

-  Raise exceptions (return errors)
-  Side effects zonder return
-  Blocking calls zonder timeout

Common Pitfalls

1. Vergeten tool_result terug te sturen

```
# ❌ Wrong
assistant_message["content"].append(block)
conversation.append(assistant_message)
# Claude wacht op result!

# ✅ Correct
conversation.append({
    "role": "user",
    "content": [{"type": "tool_result", ...}]
})
```

2. JSON serialization errors

```
# ✅ Always use json.dumps()
"content": json.dumps(result)
```

Debugging Tips

1. Print conversatie geschiedenis

```
for msg in conversation:  
    print(f"{msg['role']}: {msg['content']}")
```

2. Log tool calls

```
def _handle_tool_call(self, tool_name, tool_input, silent=False):  
    if not silent:  
        print(f"🔧 Tool: {tool_name}")  
        print(f"📥 Input: {tool_input}")  
  
    result = # ... tool logic  
  
    if not silent:  
        print(f"📤 Output: {result}")  
  
    return result
```

Advanced: Thinking Budget

Claude Sonnet 4 heeft "thinking":

```
response = self.client.messages.create(  
    model=Config.MODEL,  
    thinking={  
        "type": "enabled",  
        "budget_tokens": 2000 # ← Extended reasoning  
    },  
    messages=[...]  
)
```

Wat doet dit:

- Claude krijgt "scratchpad" voor redeneren
- Niet zichtbaar in output
- Betere tool decisions

Gebuilt in planning fase

Comparison: Met vs Zonder Tools

Zonder Tools:

User: Zoek AI experts

Claude: Ik kan niet zoeken.
Ik heb geen toegang tot
internet. Ik kan alleen
algemene info geven over
AI experts.

Met Tools:

User: Zoek AI experts

Claude: [calls web_search]
Claude: [calls fetch_page]
Claude: [calls save_candidate]

Claude: Ik heb 5 experts

Real World Metrics

Ons Guest Search systeem:

- **Tools per search:** ~15-30 calls
- **Search phase:** ~50 tool calls totaal
- **Success rate:** ~95% (5% errors)
- **Average latency:** ~2s per tool
- **Tokens per tool result:** ~500-1000

Cost implication:

- Without tools: ~\$0.10 (prompt only)
- With tools: ~\$0.40 (prompt + results)
- **Value:** Automated guest finding! 🎯

Summary: Key Takeaways

1. Tool Calling = LLM + External Functions

2. Flow:

User → Agent → Claude → tool_use → _handle_tool_call → tool_result → Claude

3. _handle_tool_call:

- Central dispatcher
- Maps tool names to implementations
- Returns standardized JSON

4. Conversatie geschiedenis cruciaal:

- Context voor volgende calls
- Claude ziet alle tool results

Resources

Documentation:

- Anthropic Tool Use docs: <https://docs.anthropic.com/en/docs/tool-use>
- Code: `src/guest_search/agent.py`
- Tests: `tests/test_agent.py`

In dit project:

- Tool definitities: `src/guest_search/tools.py`
- SmartSearchTool: `src/utls/smart_search_tool.py`
- Prompts: `src/guest_search/prompts.py`

Experimenteren:

- Anthropic Workbench

Vragen?

Belangrijkste concepten:

1. Claude beslist wanneer tools te gebruiken
2. `_handle_tool_call()` voert tools uit
3. Tool results gaan terug naar Claude
4. Agentic loop: Claude kan meerdere tools chainen

Volgende stap: Probeer zelf een tool toe te voegen!

Demo Time!

Live demonstratie:

1. Run agent met verbose logging
2. Zie tool calls in real-time
3. Inspect conversatie geschiedenis
4. Debug een tool failure

```
python main.py met debug mode
```

Thank You! 🎉

Happy Tool Calling! 🛠️🤖