Integrating LLMs into Web Applications using Tool Call

Explore the power of Large Language Models (LLMs) in web applications, focusing on how tool calls enable real-world interactions and actions.

Outline

- Software Evolution
- What is tool call
- When to Use Tool Calls
- Best Practices
- Security Considerations

Software Evolution: From 1.0 to AI-Native

Software 1.0: Hand-Coded

Explicitly programmed with human-written rules. Logic is fixed and deterministic.

Software 2.0: Data-Driven

Machine learning models learn patterns from data. Behavior adapts over time.

Software 3.0: AI-Native

LLMs drive core functionality. Systems reason, learn, and use tools to act.

Version	Software 1.0	Software 2.0	Software 3.0 (AI-Native)
Definition	Hand-coded logic and rules	Data-driven models with ML/AI assistance	AI-first systems with reasoning, planning, and memory
Development	Written by developers line-by-line	Trained on data by developers and data scientists	Co-created or self-improving with foundation models
Inputs	Code + Static requirements	Code + Labeled Data	Code + Multimodal data + Natural language prompts
Examples	Word, most legacy software	Recommendation engines	ChatGPT, Copilot, autonomous agents
User Interaction	GUI, predefined workflows	Some personalization or learning	Conversational, adaptive, context-aware
Knowledge Source	Developer logic	Historical data	Language models, real-time data, memory

“In 2025, the best programming language is English”

Tool Call

Toolcall is a way for a LLM to **call external tools or functions** (e.g., APIs) during a conversation, using structured input.

- You define what tools/functions are available.
- The model picks and calls one with arguments.
- Example: The model calls `search_flight("Addis Ababa", "Nairobi")` when asked to book a flight.

AI Agent

AI entities make decisions, take actions. LLMs serve as their 'brain'.
It often uses a language model (LLM) as its "brain".

- Can act independently or with other agents.

Example: A shopping assistant that searches, compares, and buys items for you.

Model Context Protocol

Communication protocol that connects multiple agents, tools, and memory systems to work together in a structured and persistent way..

- Enables complex multi-step, multi-agent workflows.
- Tracks agent context, memory, messages, and tool interactions.

What is a Tool Call?

Actionable Commands

LLMs call functions from user language.

AI & Systems Bridge

Connects AI to APIs, databases, and real-world tools.

Expanded Capabilities

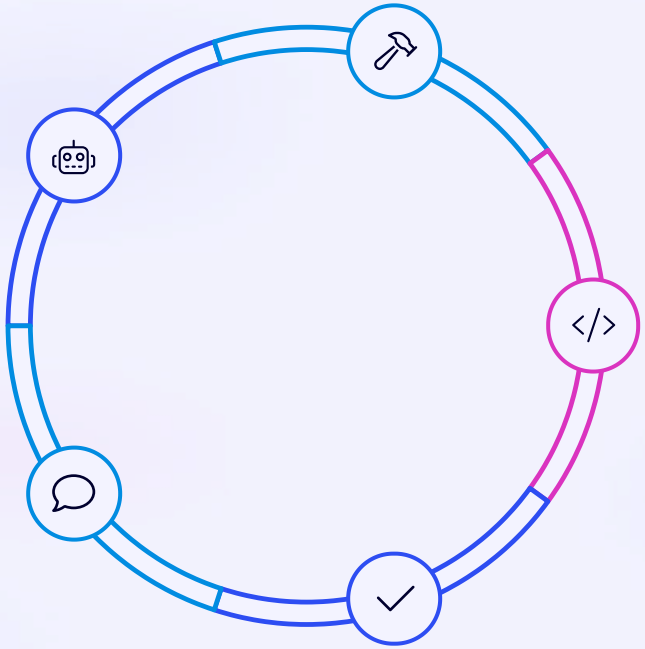
Enables LLMs to perform tasks beyond text generation.

LLM Request

Initiates a query

Tool Call

Triggers function



Function Logic

Executes backend task

LLM Reply

Final response

Result

Data returned

Tool calls allow LLMs to trigger predefined backend functions using structured JSON, enabling dynamic and interactive applications.

When to Use Tool Calls

1. **Real-time data**

   Retrieve current information

2. **Secure actions**

   Perform protected operations

3. **Multi-step workflows**

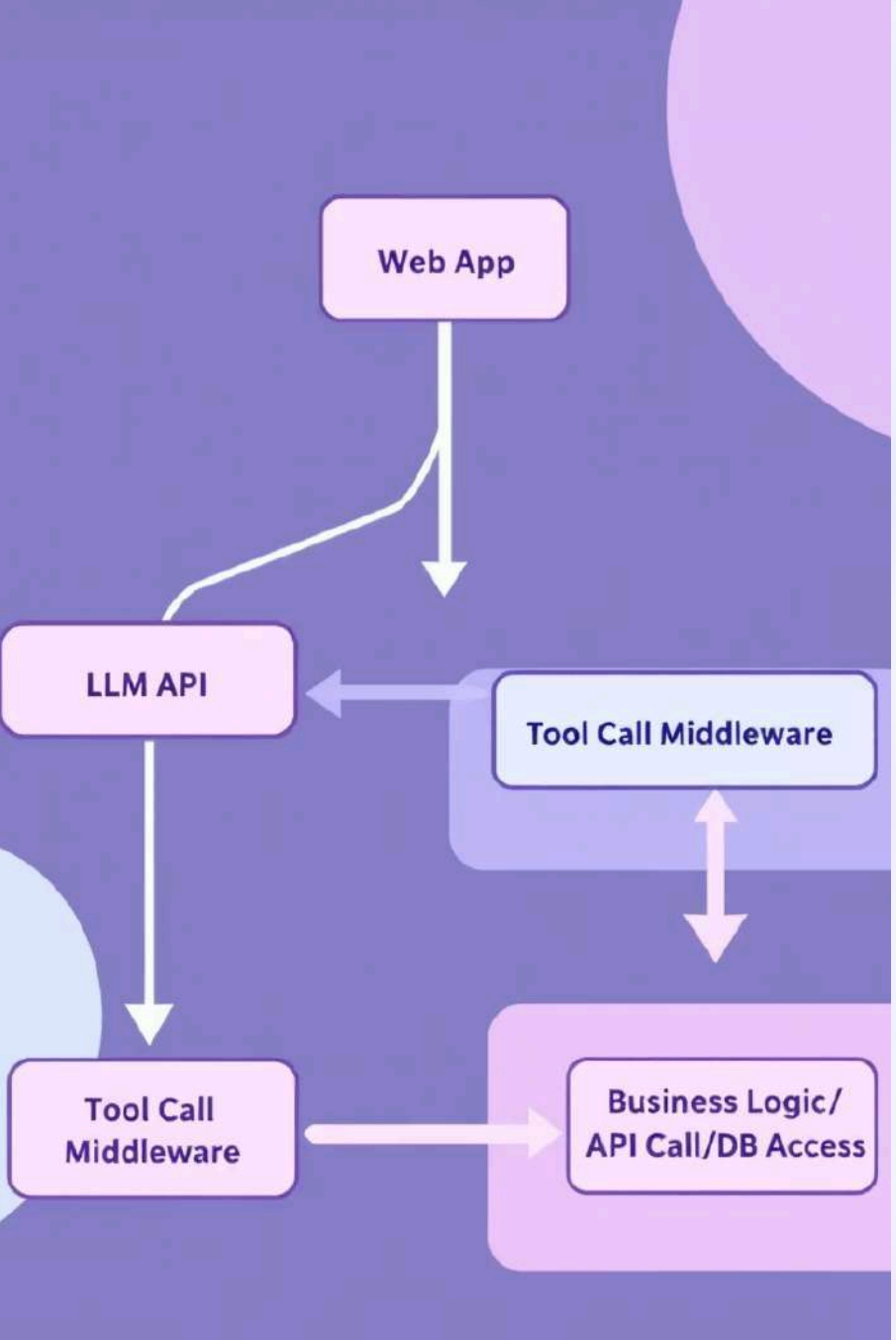
   Automate complex processes

Retrieving Real-Time or External Data

Interacting with Databases and API

trigger real-world actions (book, send, update)

Tool calls are ideal for dynamic data retrieval, secure operations, and complex multi-step logic, enhancing application capabilities.



Architecture Overview

Understand the complete flow from user interaction to LLM response, highlighting the role of tool call middleware in integrating business logic and external APIs.

How to Implement Tool Calls

Define schema

Structure the tool

Register tools

Configure LLM access

Route requests

Process tool calls

Return results

Format model response

```
import { tool } from 'ai';
import { z } from 'zod';

export const browserTool = tool({
  description: 'Browse a web page and return the content',
  parameters: z.object({
    url: z.string().describe('The URL of the page to browse'),
  }),
  execute: async ({ url }) => {
    try {
      const result = await fetch(url);
      const status = result.status;
      if ([2, 3].includes(Math.floor(status / 100))) {
        const text = await result.text();
        return text;
      }
      return `Error browsing page: ${status} ${result.statusText}`;
    } catch (error) {
      console.error(error);
      return `Error browsing page: ${error}`;
    }
  },
});
```

Implementation involves defining tool schemas, registering them with the LLM, and handling requests and responses.

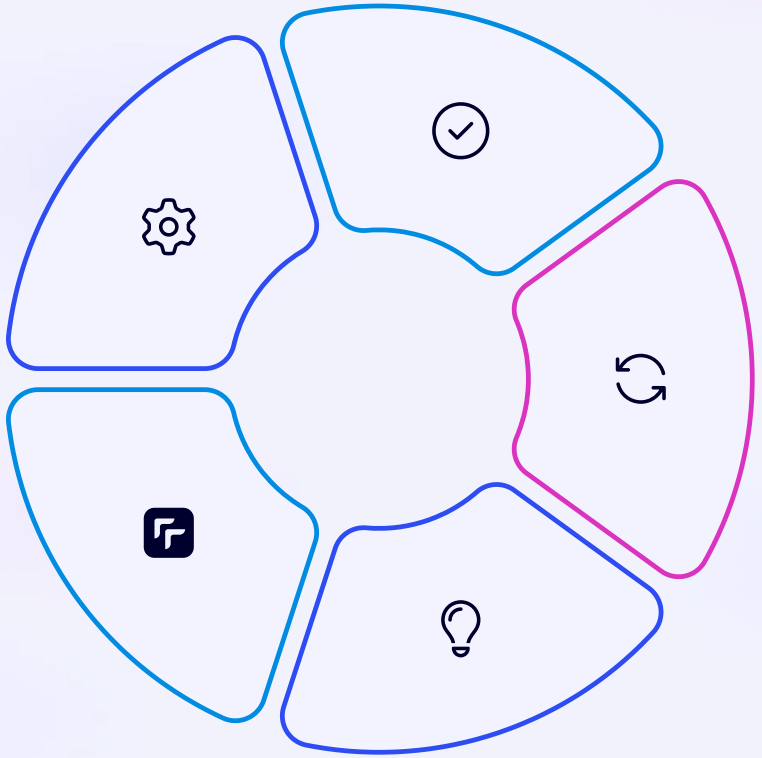
Best Practices

Atomic Tools

Design tools to perform a single, focused function for clarity and efficiency.

Robust Error Handling

Incorporate retry mechanisms and timeouts for resilient and reliable tool execution.



Validate Inputs

Implement strict validation to ensure data integrity and prevent errors.

Predictable Output

Ensure tools consistently return data in a predictable and easily parsable format.

Clear Naming

Use intuitive names and descriptions for tools to enhance LLM understanding and usage.

By adopting these best practices—from precise tool design to robust error handling—you can ensure highly reliable LLM integrations.

Security Considerations

1. **Input sanitization**

   Cleanse all incoming data

2. **Role-based access**

   Restrict tool permissions

3. **Rate limiting**

   Prevent abuse

4. **Secure external APIs**

   Encrypt and authenticate connections

5. **Log tool usage**

   Monitor with redaction

Prioritize security by sanitizing inputs, implementing access controls, and securing external API interactions for safe LLM deployments.

Demo

live demo : https://ai-todo-beta.vercel.app/

Github : **https://github.com/yosefw1221/ai-todo**

Thank You