Lecture 2: Universal Tool Calling Demo
AI Agent Development with Multi-Backend LLM Integration

(Jack) Yansong Li

October 7, 2025

1 Introduction and Course Overview

1.1 Learning Objectives

By the end of this lecture, students will be able to:

1. Understand the architecture of universal AI agents with automatic backend selection
2. Implement OpenAI-compatible tool calling mechanisms with proper error handling
3. Design and develop streaming architectures for real-time AI applications
4. Build cross-platform compatible AI systems with intelligent fallback strategies
5. Create extensible tool registries with dynamic registration capabilities
6. Apply modern software engineering patterns to AI agent development
7. Deploy and test AI agents across different hardware configurations

1.2 Project Overview

The Universal Tool Calling Demo is a sophisticated educational project that demonstrates advanced LLM tool calling functionality across multiple platforms. This project serves as a comprehensive example of modern AI agent development, showcasing:

- **Automatic Backend Selection**: Intelligent detection of system capabilities and optimal backend routing
- **OpenAI-Compatible Tool Calling**: Standard function calling format with extensible tool registry
- **Real-time Streaming**: Multi-phase streaming with visual feedback for different content types
- **Cross-Platform Compatibility**: Universal deployment across Windows, macOS, and Linux
- **Modern Development Practices**: Comprehensive testing, configuration management, and deployment strategies

1.3 Technology Stack

See Table 1.

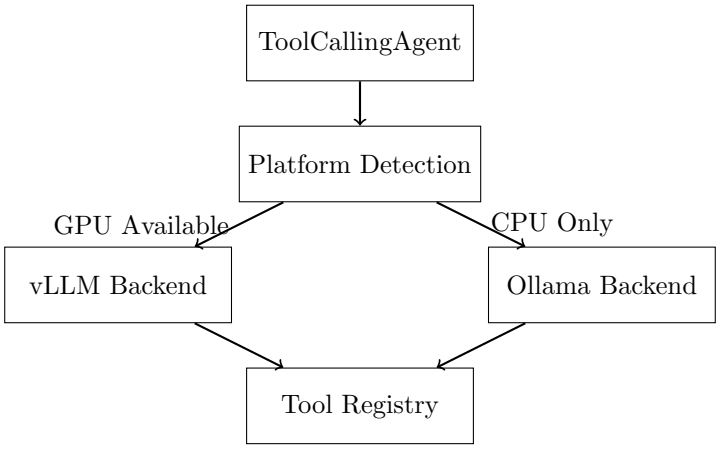
Component	Technology	Purpose
LLM Inference	vLLM (¿=0.6.0)	High-performance GPU inference
Local Inference	Ollama (¿=0.1.0)	CPU-based local inference
API Framework	FastAPI + Uvicorn	vLLM server management
Package Management	UV	Fast, reliable dependency management
Model	Qwen3-0.6B	Tool-optimized language model
Testing	pytest + coverage	Comprehensive test suite
Configuration	python-dotenv	Environment-based configuration

Table 1: Technology Stack Overview

2 Architecture and Design Patterns

2.1 High-Level Architecture

The system employs a **Universal Agent Pattern** with intelligent backend selection:



2.2 Core Design Patterns

2.2.1 Strategy Pattern: Backend Selection

The system intelligently selects the optimal backend based on platform capabilities:

```python
class ToolCallingAgent:
    def _detect_best_backend(self) -> str:
        """Intelligent backend selection based on system capabilities."""
        system = platform.system()

        # macOS - Always use Ollama for optimal compatibility
        if system == "Darwin":
            return "ollama"

        # Windows/Linux - Check for CUDA availability
        if torch.cuda.is_available():
            gpu_memory = torch.cuda.get_device_properties(0).total_memory
            if gpu_memory >= 4 * 1024**3:  # 4GB minimum
                return "vllm"

        # Fallback to Ollama for CPU systems
        return "ollama"
```

Listing 1: Backend Selection Strategy

2.2.2 Factory Pattern: Agent Creation

Automatic agent initialization based on detected capabilities:

```python
def _initialize_backend(self):
    """Factory method for backend initialization."""
    if self.backend_type == "vllm":
        self._init_vllm()
    elif self.backend_type == "ollama":
        self._init_ollama()
    else:
        raise ValueError(f"Unsupported backend: {self.backend_type}")
```

Listing 2: Factory Pattern Implementation

2.2.3 Observer Pattern: Streaming Architecture

Real-time streaming with multiple content types:

```python
def chat_stream(self, message: str, use_tools: bool = True):
    """Streaming chat with multiple chunk types."""
    for chunk in self._process_stream(message):
        chunk_type = chunk.get("type")
        content = chunk.get("content", "")

        if chunk_type == "thinking":
            yield {"type": "thinking", "content": content}  # Gray text
        elif chunk_type == "tool_call":
            yield {"type": "tool_call", "content": content}  # Blue notification
        elif chunk_type == "tool_result":
            yield {"type": "tool_result", "content": content}  # Green result
        elif chunk_type == "content":
            yield {"type": "content", "content": content}  # Final response
```

Listing 3: Streaming Observer Pattern

2.2.4 Registry Pattern: Tool Management

Extensible tool system with dynamic registration:

```python
class ToolRegistry:
    def __init__(self):
        self.tools = {}
        self._register_builtin_tools()

    def register_tool(self, name: str, function: callable,
                      description: str, parameters: Dict):
        """Register a new tool with OpenAI-compatible schema."""
        self.tools[name] = {
            "function": function,
            "description": description,
            "parameters": parameters
        }

    def execute_tool(self, name: str, arguments: Dict) -> str:
        """Execute a registered tool with error handling."""
        if name not in self.tools:
            return json.dumps({"error": f"Tool {name} not found"})

        try:
            tool_func = self.tools[name]["function"]
            result = tool_func(**arguments)
            return json.dumps({"success": True, "result": result})
        except Exception as e:
            return json.dumps({
                "success": False,
```

```
                "error": str(e),
                "traceback": traceback.format_exc()
            })
```

Listing 4: Tool Registry Pattern

3 Tool Calling Mechanisms

3.1 OpenAI-Compatible Function Calling

The system implements the standard OpenAI function calling format:

```
{
  "tool_calls": [{
    "id": "call_123",
    "type": "function",
    "function": {
      "name": "get_weather",
      "arguments": {
        "location": "Tokyo, Japan",
        "unit": "celsius"
      }
    }
  }]
}
```

Listing 5: Function Calling Schema

3.2 ReAct Loop Implementation

The ReAct (Reasoning + Acting) pattern enables multi-step problem solving:

```
def _process_with_tools(self, message: str, max_iterations: int = 10):
    """Multi-step reasoning with tool execution."""
    iteration = 0

    while iteration < max_iterations:
        # Get model response
        response = self._get_model_response(message)

        # Parse tool calls from response
        tool_calls = self._parse_tool_calls(response)

        if not tool_calls:
            # No tools needed, return final response
            return response

        # Execute tools and collect results
        tool_results = []
        for tool_call in tool_calls:
            result = self.tool_registry.execute_tool(
                tool_call["name"],
                tool_call["arguments"]
            )
            tool_results.append({
                "tool_call_id": tool_call["id"],
                "result": result
            })

        # Add results to conversation history
        self.conversation_history.extend(tool_results)

        iteration += 1

```

```
    return "Maximum iterations reached"
```

Listing 6: ReAct Loop Implementation

3.3 Built-in Tools Analysis

3.3.1 Weather Tool

Real-time weather data retrieval with location parsing:

```
def get_current_temperature(location: str, unit: str = "celsius") -> str:
    """Get current temperature for a location using Open-Meteo API."""
    # Parse location to coordinates
    lat, lon = parse_location(location)

    # API call to Open-Meteo
    url = f"https://api.open-meteo.com/v1/forecast"
    params = {
        "latitude": lat,
        "longitude": lon,
        "current_weather": True,
        "temperature_unit": unit
    }

    response = requests.get(url, params=params)
    data = response.json()

    temperature = data["current_weather"]["temperature"]
    return f"Current temperature in {location}: {temperature}{unit}"
```

Listing 7: Weather Tool Implementation

3.3.2 Code Interpreter

Full Python environment with error handling:

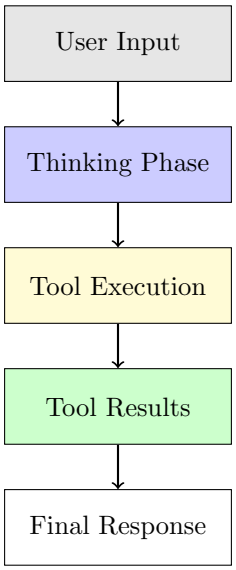
```
def code_interpreter(code: str) -> str:
    """Execute Python code safely with output capture."""
    try:
        # Capture stdout and stderr
        old_stdout = sys.stdout
        old_stderr = sys.stderr
        sys.stdout = captured_output = StringIO()
        sys.stderr = captured_error = StringIO()

        # Execute the code
        exec(code, {"__builtins__": __builtins__})

        # Restore streams
        sys.stdout = old_stdout
        sys.stderr = old_stderr

        output = captured_output.getvalue()
        error = captured_error.getvalue()

        if error:
            return f"Output:\\n{output}\\nError:\\n{error}"
        return output if output else "Code executed successfully"

    except Exception as e:
        return json.dumps({
            "success": False,
            "error": str(e),
            "traceback": traceback.format_exc()
        })
```

Listing 8: Code Interpreter Tool

4 Streaming Architecture

4.1 Multi-Type Streaming System

The streaming architecture provides real-time feedback with different content types:



4.2 Streaming Implementation

```
CHUNK_TYPES = {
    "thinking": "Internal reasoning process",
    "tool_call": "Tool execution initiation",
    "tool_result": "Tool execution completion",
    "content": "Final response content",
    "error": "Error notifications"
}

def chat_stream(self, message: str, use_tools: bool = True):
    """Streaming chat with visual feedback."""
    for chunk in self._process_stream(message):
        chunk_type = chunk.get("type")
        content = chunk.get("content", "")

        if chunk_type == "thinking":
            # Gray text for internal reasoning
            print(f"\\033[90m{content}\\033[0m", end="", flush=True)
            yield {"type": "thinking", "content": content}

        elif chunk_type == "tool_call":
            # Blue notification for tool execution
            print(f"Tool: {content['name']}")
            yield {"type": "tool_call", "content": content}

        elif chunk_type == "tool_result":
            # Green result display
            print(f"Result: {content}")
            yield {"type": "tool_result", "content": content}

```

```
        elif chunk_type == "content":
            # Standard text for final response
            print(content, end="", flush=True)
            yield {"type": "content", "content": content}
```

Listing 9: Streaming Chunk Processing

4.3 Performance Optimization

The streaming system includes several optimization strategies:

- **Chunk Batching**: Groups related content to reduce overhead
- **Buffer Management**: Intelligent buffering for smooth streaming
- **Backpressure Handling**: Prevents overwhelming the client with too many chunks
- **Timeout Protection**: Ensures streaming doesn't hang indefinitely

5 Cross-Platform Compatibility

5.1 Platform Detection Strategy

Intelligent platform detection with capability assessment:

```
def check_system_compatibility():
    """Comprehensive platform and capability detection."""
    system = platform.system()

    # macOS Detection
    if system == "Darwin":
        # Check for Apple Silicon vs Intel
        if platform.machine() == "arm64":
            return "ollama", "Apple Silicon Mac (M1/M2)"
        else:
            return "ollama", "Intel Mac"

    # Windows/Linux CUDA Detection
    if torch.cuda.is_available():
        gpu_name = torch.cuda.get_device_name(0)
        gpu_memory = torch.cuda.get_device_properties(0).total_memory

        if gpu_memory >= 4 * 1024**3:  # 4GB minimum
            return "vllm", f"NVIDIA GPU: {gpu_name} ({gpu_memory//1024**3}GB)"

    # CPU-based fallback
    cpu_count = multiprocessing.cpu_count()
    memory = psutil.virtual_memory().total

    return "ollama", f"CPU system ({cpu_count} cores, {memory//1024**3}GB RAM)"
```

Listing 10: Platform Detection Implementation

5.2 Backend Abstraction

Unified interface across different backends:

```
from abc import ABC, abstractmethod

class BaseAgent(ABC):
    """Abstract base class for all backend implementations."""

    @abstractmethod
```

```python
    def chat(self, message: str, use_tools: bool = True,
             stream: bool = False) -> Union[str, Generator]:
        """Process a chat message with optional tool usage."""
        pass

    @abstractmethod
    def reset_conversation(self) -> None:
        """Reset conversation history."""
        pass

    @abstractmethod
    def get_system_info(self) -> Dict[str, Any]:
        """Get system and model information."""
        pass

class VLLMAgent(BaseAgent):
    """vLLM-specific implementation with GPU acceleration."""

    def __init__(self, model_name: str, host: str, port: int):
        self.model_name = model_name
        self.server = VLLMServer(host, port)
        self.client = OpenAI(base_url=f"http://{host}:{port}/v1")

    def chat(self, message: str, use_tools: bool = True,
             stream: bool = False):
        # vLLM-specific implementation
        pass

class OllamaAgent(BaseAgent):
    """Ollama implementation for local CPU inference."""

    def __init__(self, model_name: str):
        self.model_name = model_name
        self.client = ollama.Client()

    def chat(self, message: str, use_tools: bool = True,
             stream: bool = False):
        # Ollama-specific implementation
        pass
```

Listing 11: Backend Abstraction Interface

5.3 Platform-Specific Setup

5.3.1 macOS Setup

```bash
# Install via Homebrew
brew install ollama

# Start Ollama service
ollama serve  # Run in separate terminal

# Download default model
ollama pull qwen3:0.6b
```

Listing 12: macOS Installation

5.3.2 Windows Setup

```bash
# Download from official website
# https://ollama.com/download/windows

# Install and start service
# Ollama will run as system service
```

```
6
7 # Download model via command line
8 ollama pull qwen3:0.6b
```

Listing 13: Windows Installation

5.3.3 Linux Setup

```
1 # Install Ollama
2 curl -fsSL https://ollama.com/install.sh | sh
3
4 # Start service
5 sudo systemctl start ollama
6 sudo systemctl enable ollama
7
8 # Download model
9 ollama pull qwen3:0.6b
10
11 # For vLLM (requires CUDA)
12 # Install CUDA toolkit from NVIDIA
13 # vLLM will be automatically detected and used
```

Listing 14: Linux Installation

6 Testing and Development

6.1 Comprehensive Testing Framework

The project includes extensive testing with multiple categories:

```
1 tests/
2 |-- test_main.py              # Main agent functionality
3 |-- test_streaming.py         # Streaming implementation
4 |-- test_code_interpreter_full.py  # Tool execution testing
5 `-- conftest.py              # Test fixtures and mocks
```

Listing 15: Test Structure

6.2 Unit Testing

```
1 def test_backend_detection():
2     """Test automatic backend selection logic."""
3     with patch('platform.system') as mock_system:
4         with patch('torch.cuda.is_available') as mock_cuda:
5             # Test macOS detection
6             mock_system.return_value = "Darwin"
7             backend, description = check_system_compatibility()
8             assert backend == "ollama"
9             assert "Mac" in description
10
11            # Test Windows with CUDA
12            mock_system.return_value = "Windows"
13            mock_cuda.return_value = True
14            backend, description = check_system_compatibility()
15            assert backend == "vllm"
16            assert "NVIDIA GPU" in description
17
18            # Test Linux without CUDA
19            mock_system.return_value = "Linux"
20            mock_cuda.return_value = False
21            backend, description = check_system_compatibility()
```

```
            assert backend == "ollama"
```

Listing 16: Backend Detection Testing

6.3 Integration Testing

```
def test_tool_execution_workflow():
    """Test complete tool execution workflow."""
    agent = ToolCallingAgent(backend="ollama")

    # Test weather tool execution
    response = agent.chat("What's the weather in Tokyo?")

    # Verify response contains weather information
    assert "temperature" in response.lower() or "weather" in response.lower()
    assert "Tokyo" in response

    # Test code interpreter
    response = agent.chat("Calculate 2 + 2")
    assert "4" in response
```

Listing 17: Tool Execution Integration Test

6.4 Mock-Based Testing

```
def test_error_handling():
    """Test error propagation and formatting."""
    registry = ToolRegistry()

    # Test tool not found
    result = registry.execute_tool("nonexistent_tool", {})
    result_dict = json.loads(result)
    assert result_dict["success"] == False
    assert "not found" in result_dict["error"]

    # Test execution error
    result = registry.execute_tool("code_interpreter",
                                   {"code": "1/0"})
    result_dict = json.loads(result)
    assert result_dict["success"] == False
    assert "ZeroDivisionError" in result_dict["error"]
    assert "traceback" in result_dict
```

Listing 18: Mock Testing Strategy

6.5 Modern Development Workflow

```
# Install dependencies with UV
uv sync

# Run all tests
uv run pytest

# Run specific test file with verbose output
uv run pytest tests/test_main.py -v

# Run tests with coverage reporting
uv run pytest tests/ --cov=src/local_llm_serving

# Run single test by name pattern
uv run pytest -k "test_tool_execution"

```

```
# Format code with black
uv run black src/ tests/

# Sort imports with isort
uv run isort src/ tests/
```

Listing 19: Development Commands

7 Configuration Management

7.1 Environment-Based Configuration

```
# Load environment variables
from dotenv import load_dotenv
load_dotenv()

# Configuration with defaults
class Config:
    MODEL_NAME = os.getenv("MODEL_NAME", "Qwen/Qwen3-0.6B")
    VLLM_HOST = os.getenv("VLLM_HOST", "localhost")
    VLLM_PORT = int(os.getenv("VLLM_PORT", "8000"))
    LOG_LEVEL = os.getenv("LOG_LEVEL", "INFO")
    MAX_ITERATIONS = int(os.getenv("MAX_ITERATIONS", "10"))
    TOOL_TIMEOUT = int(os.getenv("TOOL_TIMEOUT", "30"))

    # Backend-specific settings
    OLLAMA_HOST = os.getenv("OLLAMA_HOST", "localhost")
    OLLAMA_PORT = int(os.getenv("OLLAMA_PORT", "11434"))
```

Listing 20: Configuration Management

7.2 Logging Configuration

```
import logging
import sys

# Configure logging
logging.basicConfig(
    level=getattr(logging, Config.LOG_LEVEL),
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.StreamHandler(sys.stdout),
        logging.FileHandler('logs/agent.log')
    ]
)

# Module-specific loggers
logger = logging.getLogger(__name__)
```

Listing 21: Logging Setup

8 Practical Exercises

8.1 Exercise 1: Custom Tool Development

Objective: Implement a custom tool that integrates with an external API.

Requirements:

- Create a tool that fetches current cryptocurrency prices

- Implement proper error handling and rate limiting
- Follow OpenAI function calling format
- Add comprehensive unit tests

Solution Template:

```python
def get_crypto_price(symbol: str, currency: str = "USD") -> str:
    """Get current cryptocurrency price."""
    # Implementation here
    pass

# Register the tool
registry.register_tool(
    name="get_crypto_price",
    function=get_crypto_price,
    description="Get current price of a cryptocurrency",
    parameters={
        "type": "object",
        "properties": {
            "symbol": {"type": "string", "description": "Crypto symbol (BTC, ETH, etc.)"},
            "currency": {"type": "string", "description": "Currency code (USD, EUR, etc.)"}
        },
        "required": ["symbol"]
    }
)
```

Listing 22: Custom Tool Implementation

8.2 Exercise 2: Backend Implementation

Objective: Implement a new backend for a different LLM provider.

Requirements:

- Create a backend for Hugging Face Transformers
- Implement the BaseAgent interface
- Support both streaming and non-streaming modes
- Add tool calling capabilities
- Write integration tests

Implementation Guidelines:

```python
class HuggingFaceAgent(BaseAgent):
    """Hugging Face Transformers backend implementation."""

    def __init__(self, model_name: str, device: str = "auto"):
        self.model_name = model_name
        self.device = self._get_device(device)
        self.tokenizer = AutoTokenizer.from_pretrained(model_name)
        self.model = AutoModelForCausalLM.from_pretrained(model_name)
        self.model.to(self.device)

    def chat(self, message: str, use_tools: bool = True,
             stream: bool = False):
        # Implementation here
        pass
```

Listing 23: New Backend Template

8.3 Exercise 3: Cross-Platform Deployment

Objective: Deploy the application on different platforms with automated setup.
Requirements:

- Create Docker containers for different platforms
- Implement automated setup scripts
- Add health checks and monitoring
- Document deployment procedures
- Test on multiple cloud providers

8.4 Exercise 4: Performance Optimization

Objective: Optimize the streaming performance and reduce latency.
Requirements:

- Implement chunk batching for reduced overhead
- Add buffer management for smooth streaming
- Optimize tool execution parallelism
- Add caching for frequently used tools
- Measure and document performance improvements

9 Assessment Criteria

9.1 Learning Outcomes Assessment

Students will be evaluated based on their ability to:

A. **Architecture Understanding** (25%)

- Explain the Universal Agent Pattern and its benefits
- Describe the backend selection strategy and rationale
- Identify and explain the design patterns used

B. **Implementation Skills** (35%)

- Implement custom tools following best practices
- Create backend implementations with proper interfaces
- Handle errors gracefully with appropriate feedback

C. **Testing Proficiency** (20%)

- Write comprehensive unit tests with good coverage
- Implement integration tests for complex workflows
- Use mocking strategies effectively

D. **Cross-Platform Development** (20%)

- Deploy applications on different platforms
- Handle platform-specific requirements
- Document deployment procedures clearly

9.2 Practical Project Assessment

Final Project: Students must implement a complete AI agent application with:

- Custom domain-specific tools (minimum 3 tools)
- Multi-backend support (minimum 2 backends)
- Comprehensive test suite (minimum 80% coverage)
- Cross-platform deployment documentation
- Performance optimization and benchmarking
- User documentation and API reference

10 Conclusion

This lecture has covered the comprehensive architecture and implementation of the Universal Tool Calling Demo, demonstrating advanced concepts in:

- Modern AI agent development with multi-backend support
- OpenAI-compatible tool calling with extensible architecture
- Real-time streaming with visual feedback systems
- Cross-platform compatibility and intelligent fallback strategies
- Production-ready testing and deployment practices
- Software engineering patterns in AI application development

The project serves as an excellent foundation for understanding how to build sophisticated AI applications that work reliably across diverse computing environments while maintaining high performance and user experience standards.

10.1 Further Reading

1. OpenAI Function Calling Documentation: `https://platform.openai.com/docs/guides/function-calling`
2. ReAct Paper: "ReAct: Synergizing Reasoning and Acting in Language Models"
3. vLLM Documentation: `https://docs.vllm.ai/`
4. Ollama Documentation: `https://ollama.com/`
5. Python Packaging with pyproject.toml: `https://packaging.python.org/en/latest/guides/writing-pyproject-to`