# Tool Agent - Execution Engine

## 🎯 Tổng quan

**Tool Agent** là execution powerhouse của hệ thống MAS-Planning, chịu trách nhiệm thực thi concrete actions thông qua Model Context Protocol (MCP) tools. Đây là “hands” của hệ thống, biến plans và analyses thành real-world device control actions.

## 🏗️ Kiến trúc và Integration

graph TB  
 subgraph "Tool Agent Architecture"  
 A[User Command] --> B[Input Router]  
 B --> C{Command Type}  
   
 C -->|Device Control| D[Execute Tool]  
 C -->|Tool Query| E[List Tools]  
 C -->|Search| F[Search Tools]  
 C -->|Info Request| G[Get Tool Info]  
 C -->|Help| H[Help System]  
   
 D --> I[MCP Client]  
 E --> I  
 F --> I  
 G --> I  
   
 I --> J[MCP Server]  
 J --> K[Smart Home Devices]  
   
 D --> L[Authentication Manager]  
 L --> M[Token Validation]  
   
 N[LLM Tool Calling] --> D  
 O[Response Formatter] --> P[Final Output]  
   
 D --> O  
 E --> O  
 F --> O  
 G --> O  
 H --> O  
 end  
   
 style I fill:#e1f5fe  
 style J fill:#f3e5f5  
 style L fill:#e8f5e8  
 style N fill:#fff3e0

## 🎯 Core Responsibilities

### **1. Device Control Execution**

* **Real-time Device Control**: Immediate response cho device commands
* **Multi-Device Coordination**: Orchestrate multiple device actions
* **State Management**: Track và maintain device states
* **Error Recovery**: Handle device communication failures gracefully

### **2. MCP Tool Integration**

* **Tool Discovery**: Dynamic discovery của available MCP tools
* **Tool Orchestration**: Intelligent tool selection và execution
* **Protocol Management**: Handle MCP communication protocols
* **Service Integration**: Connect với various smart home services

### **3. Authentication & Security**

* **Token-Based Authentication**: Secure authentication với MCP servers
* **Access Control**: Validate permissions cho device operations
* **Security Validation**: Ensure safe device operations
* **Audit Logging**: Comprehensive security event logging

### **4. Response Processing**

* **Result Formatting**: User-friendly response formatting
* **Error Handling**: Graceful error reporting và recovery
* **Status Reporting**: Real-time operation status updates
* **Performance Monitoring**: Track execution performance

## 🧩 Core Components

### **1. Intelligent Router System**

def router(self, state: ToolState):  
 """Route input to appropriate action"""  
 input\_value = state.get('input', '') or ''  
 input\_text = input\_value.lower() if input\_value else ''  
   
 # Priority routing logic  
 if any(keyword in input\_text for keyword in [  
 'device', 'room', 'living room', 'bedroom', 'kitchen',   
 'switch', 'control', 'turn on', 'turn off', 'air conditioner'  
 ]):  
 route = 'execute\_tool'  
 elif any(keyword in input\_text for keyword in [  
 'list tools', 'available tools', 'show tools', 'what tools'  
 ]) and 'device' not in input\_text:  
 route = 'list\_tools'  
 elif any(keyword in input\_text for keyword in [  
 'search', 'find'  
 ]):  
 route = 'search\_tools'  
 elif any(keyword in input\_text for keyword in [  
 'info', 'detail', 'describe'  
 ]):  
 route = 'get\_tool\_info'  
 else:  
 # Default to execution for device commands  
 route = 'execute\_tool'  
   
 return {\*\*state, 'route': route}

**Routing Intelligence:**

* **Context-Aware**: Understanding context từ input patterns
* **Priority-Based**: Device control có highest priority
* **Fallback Logic**: Intelligent defaults cho ambiguous inputs
* **Extensible**: Easy to add new routing rules

### **2. MCP Tool Management**

async def get\_mcp\_tools(self):  
 """Get tools from MCP server với error handling"""  
 try:  
 async with asyncio.timeout(5): # 5 second timeout  
 async with MultiServerMCPClient({  
 "mcp-server": {  
 "url": env.MCP\_SERVER\_URL,  
 "transport": "sse",  
 }  
 }) as client:  
 tools = list(client.get\_tools())  
 return tools  
 except asyncio.TimeoutError:  
 logger.error("❌ MCP connection timeout after 5 seconds")  
 return []  
 except Exception as e:  
 logger.error(f"❌ MCP connection failed: {str(e)}")  
 return []

**Tool Management Features:**

* **Dynamic Discovery**: Real-time tool availability checking
* **Connection Pooling**: Efficient MCP connection management
* **Timeout Handling**: Robust timeout mechanisms
* **Error Recovery**: Graceful handling của connection failures

### **3. Authentication Framework**

def detect\_placeholder\_tokens(self, text: str) -> List[str]:  
 """Detect placeholder tokens in text"""  
 patterns = [  
 r"token\s\*=\s\*['\"]your\_auth\_token['\"]",  
 r"token\s\*=\s\*['\"]your\_token['\"]",   
 r"token\s\*=\s\*['\"]auth\_token['\"]",  
 r"token\s\*=\s\*['\"][^'\"]\*token[^'\"]\*['\"]"  
 ]  
   
 found\_tokens = []  
 for pattern in patterns:  
 matches = re.findall(pattern, text, re.IGNORECASE)  
 found\_tokens.extend(matches)  
   
 return found\_tokens  
  
def replace\_placeholder\_tokens(self, text: str, real\_token: str) -> str:  
 """Replace placeholder tokens with real authentication token"""  
 if not real\_token:  
 return text  
   
 patterns = [  
 (r"token\s\*=\s\*['\"]your\_auth\_token['\"]", f"token='{real\_token}'"),  
 (r"token\s\*=\s\*['\"]your\_token['\"]", f"token='{real\_token}'"),  
 (r"token\s\*=\s\*['\"]auth\_token['\"]", f"token='{real\_token}'"),  
 ]  
   
 result = text  
 for pattern, replacement in patterns:  
 result = re.sub(pattern, replacement, result, flags=re.IGNORECASE)  
   
 return result

**Security Features:**

* **Token Detection**: Intelligent placeholder token identification
* **Secure Replacement**: Safe token substitution
* **No Storage**: Tokens are never persisted
* **Validation**: Token format và validity checking

### **4. Execution Engine**

def execute\_tool(self, state: ToolState):  
 """Execute tool với comprehensive error handling"""  
 query = state.get('input', '')  
 token = state.get('token', '')  
   
 if not self.llm:  
 return {\*\*state, 'output': "❌ LLM not initialized", 'error': 'LLM not initialized'}  
   
 if not self.tools:  
 return {\*\*state, 'output': "❌ No MCP tools available", 'error': 'No MCP tools'}  
   
 try:  
 # Process authentication requirements  
 placeholder\_tokens = self.detect\_placeholder\_tokens(query)  
 processed\_query = self.replace\_placeholder\_tokens(query, token) if token else query  
   
 # Build enhanced system prompt với authentication  
 system\_prompt = TOOL\_PROMPT  
 if token:  
 system\_prompt += f"""  
🔐 AUTHENTICATION TOKEN PROVIDED:  
Your authentication token is: {token}  
  
MANDATORY: Include this exact token in EVERY MCP tool call using the 'token' parameter.  
Example tool calls:  
- get\_device\_list(token="{token}")  
- switch\_device\_control(token="{token}", buttonId=123, action="on")  
  
NEVER make tool calls without the token parameter."""  
   
 # Invoke LLM với tools  
 messages = [SystemMessage(system\_prompt), HumanMessage(processed\_query)]  
 lc\_messages = convert\_messages\_list(messages)  
 response = self.llm.invoke(lc\_messages)  
   
 # Process tool calls  
 if hasattr(response, 'tool\_calls') and response.tool\_calls:  
 return self.process\_tool\_calls(response.tool\_calls, state)  
 else:  
 return {\*\*state, 'output': f"🔧 \*\*Tool Response:\*\*\n\n{response.content}"}  
   
 except Exception as e:  
 return self.handle\_execution\_error(e, query, state)

## 🔄 Execution Workflows

### **1. Device Control Flow**

sequenceDiagram  
 participant U as User  
 participant T as Tool Agent  
 participant L as LLM  
 participant M as MCP Client  
 participant D as Device  
   
 U->>T: "Turn on living room lights"  
 T->>T: Parse Command & Extract Token  
 T->>L: Generate Tool Call with Context  
 L->>T: Tool Call Instructions  
 T->>M: Execute MCP Tool Call  
 M->>D: Device Control Command  
 D->>M: Device Response  
 M->>T: Execution Result  
 T->>T: Format Response  
 T->>U: Success/Failure Report

### **2. Tool Discovery Flow**

sequenceDiagram  
 participant U as User  
 participant T as Tool Agent  
 participant M as MCP Server  
 participant C as Cache  
   
 U->>T: "List available tools"  
 T->>C: Check Tool Cache  
 alt Cache Hit  
 C->>T: Cached Tool List  
 else Cache Miss  
 T->>M: Request Available Tools  
 M->>T: Tool Definitions  
 T->>C: Update Cache  
 end  
 T->>T: Format Tool List  
 T->>U: Available Tools Response

### **3. Error Recovery Flow**

sequenceDiagram  
 participant T as Tool Agent  
 participant M as MCP Client  
 participant L as Logger  
 participant F as Fallback  
   
 T->>M: Execute Tool Call  
 M-->>T: Error Response  
 T->>L: Log Error Details  
 T->>T: Analyze Error Type  
 alt Authentication Error  
 T->>F: Request New Token  
 else Network Error  
 T->>F: Retry with Backoff  
 else Tool Error  
 T->>F: Alternative Tool  
 end  
 F->>T: Recovery Action  
 T->>User: Informative Error Message

## 🧠 Advanced Features

### **1. Intelligent Tool Selection**

async def call\_tool\_async(self):  
 """Call tool với fresh MCP connection"""  
 try:  
 # Get token từ state  
 token = state.get('token', '')  
 if token:  
 tool\_args['token'] = token  
 logger.info(f"🔑 Token injected into tool args")  
   
 # Create fresh MCP client connection  
 async with MultiServerMCPClient({  
 "mcp-server": {  
 "url": env.MCP\_SERVER\_URL,  
 "transport": "sse",  
 }  
 }) as client:  
 # Get fresh tools từ client  
 fresh\_tools = list(client.get\_tools())  
   
 # Find matching tool trong fresh tools  
 fresh\_tool = None  
 for tool in fresh\_tools:  
 if tool.name == tool\_name:  
 fresh\_tool = tool  
 break  
   
 if not fresh\_tool:  
 return f"Tool {tool\_name} not found in fresh MCP connection"  
   
 # Call the fresh tool  
 result = await fresh\_tool.ainvoke(tool\_args)  
 return result  
   
 except Exception as e:  
 error\_msg = f"Tool call failed: {str(e)}"  
 logger.error(f"❌ Tool call error: {error\_msg}")  
 return error\_msg

**Selection Intelligence:**

* **Dynamic Tool Discovery**: Real-time tool availability
* **Fresh Connection Strategy**: Avoid stale connection issues
* **Capability Matching**: Match tools to task requirements
* **Performance Optimization**: Select fastest/most reliable tools

### **2. Response Processing Engine**

def process\_tool\_results(self, result) -> str:  
 """Process và format tool execution results"""  
 if isinstance(result, str):  
 # Check for authentication errors  
 if any(phrase in result.lower() for phrase in [  
 "need a token", "token required", "authenticate",   
 "authorization failed", "invalid token"  
 ]):  
 return self.format\_auth\_error(result)  
 elif result.startswith("Tool call failed:"):  
 return f"\*\*Error\*\*: {result}"  
 else:  
 # Try to parse as JSON for better formatting  
 try:  
 import json  
 parsed = json.loads(result)  
 return self.format\_json\_response(parsed)  
 except (json.JSONDecodeError, TypeError):  
 return result  
 elif isinstance(result, dict):  
 return self.format\_dict\_response(result)  
 else:  
 return str(result)  
  
def format\_device\_list(self, devices: List[Dict]) -> str:  
 """Format device list for user-friendly display"""  
 devices\_info = []  
 for room in devices:  
 room\_name = room.get('room\_name', 'Unknown Room')  
 devices = room.get('devices', [])  
 buttons = room.get('buttons', [])  
   
 devices\_info.append(f"🏠 \*\*{room\_name}\*\*:")  
 for device in devices:  
 status = device.get('device\_status', 'Unknown')  
 devices\_info.append(f" 📱 {device.get('name', 'Unknown Device')} ({status})")  
 for button in buttons:  
 status = button.get('status', 'Unknown')  
 type\_info = button.get('button\_type', 'Unknown Type')  
 devices\_info.append(f" 🔘 {button.get('name', 'Unknown Button')} ({type\_info}) - {status}")  
   
 return "\n".join(devices\_info)

### **3. Error Handling Framework**

def handle\_execution\_error(self, error: Exception, query: str, state: ToolState) -> Dict:  
 """Comprehensive error handling với user-friendly responses"""  
 error\_msg = str(error)  
   
 # Categorize error types  
 if "authentication" in error\_msg.lower():  
 output = self.format\_auth\_error\_response(error\_msg)  
 elif "network" in error\_msg.lower() or "connection" in error\_msg.lower():  
 output = self.format\_network\_error\_response(error\_msg)  
 elif "timeout" in error\_msg.lower():  
 output = self.format\_timeout\_error\_response(error\_msg)  
 else:  
 # Fallback to mock execution for development  
 output = f"""🧪 \*\*Mock Tool Execution\*\* (Error encountered):  
  
Task: {query}  
Status: ⚠️ Simulated execution (Error: {error\_msg})  
Result: Mock response for development purposes  
Note: The actual tool execution failed, but this demonstrates the expected response format."""  
   
 return {\*\*state, 'output': output, 'error': error\_msg}  
  
def format\_auth\_error\_response(self, error\_msg: str) -> str:  
 """Format authentication error messages"""  
 return f"""🔐 \*\*Authentication Required\*\*  
  
The MCP tool requires a valid authentication token. Please ensure you have provided a current OXII API token.  
  
Error Details: {error\_msg}  
  
💡 \*\*How to Fix:\*\*  
1. Obtain a fresh token from the OXII system  
2. Include the token in your request  
3. Ensure the token has not expired  
  
Example: {{"input": "turn on lights", "token": "your\_valid\_token\_here"}}"""

## 🔧 Technical Implementation

### **1. Async Architecture**

async def init\_async(self):  
 """Initialize async components"""  
 self.tools = await self.get\_mcp\_tools()  
 base\_llm = ChatVertexAI(  
 model\_name=self.model,  
 temperature=self.temperature,  
 project=env.GOOGLE\_CLOUD\_PROJECT,  
 location=env.GOOGLE\_CLOUD\_LOCATION  
 )  
   
 # Bind tools to LLM for proper tool calling  
 if self.tools:  
 self.llm = base\_llm.bind\_tools(self.tools)  
 else:  
 self.llm = base\_llm  
   
 if self.verbose:  
 logger.info(f"🔧 Loaded {len(self.tools)} MCP tools")

### **2. LangGraph State Management**

class ToolState(TypedDict):  
 input: str  
 token: str  
 route: str  
 output: str  
 tool\_data: Dict[str, Any]  
 error: str  
  
def create\_graph(self):  
 """Create execution graph for ToolAgent"""  
 workflow = StateGraph(ToolState)  
   
 # Add nodes  
 workflow.add\_node('router', self.router)  
 workflow.add\_node('list\_tools', self.list\_tools)  
 workflow.add\_node('search\_tools', self.search\_tools)  
 workflow.add\_node('get\_tool\_info', self.get\_tool\_info)  
 workflow.add\_node('execute\_tool', self.execute\_tool)  
 workflow.add\_node('help\_with\_tool', self.help\_with\_tool)  
   
 # Set entry point  
 workflow.set\_entry\_point('router')  
   
 # Add conditional edges  
 workflow.add\_conditional\_edges('router', self.controller)  
   
 # All nodes terminate  
 for node in ['list\_tools', 'search\_tools', 'get\_tool\_info', 'execute\_tool', 'help\_with\_tool']:  
 workflow.add\_edge(node, END)  
   
 return workflow.compile(debug=self.verbose)

### **3. Event Loop Management**

def run\_async\_tool(self):  
 """Run async tool trong proper event loop"""  
 try:  
 # Check if we're already in an event loop  
 try:  
 loop = asyncio.get\_running\_loop()  
 logger.info("🔄 Using existing event loop")  
 # Create a new task trong existing loop  
 import concurrent.futures  
 with concurrent.futures.ThreadPoolExecutor() as executor:  
 future = executor.submit(asyncio.run, self.call\_tool\_async())  
 return future.result(timeout=30)  
 except RuntimeError:  
 # No running loop, create a new one  
 logger.info("🆕 Creating new event loop")  
 return asyncio.run(self.call\_tool\_async())  
 except Exception as e:  
 error\_msg = f"Async execution failed: {str(e)}"  
 logger.error(f"❌ Async execution error: {error\_msg}")  
 return error\_msg

## 📊 Performance Features

### **1. Connection Management**

* **Connection Pooling**: Reuse MCP connections efficiently
* **Timeout Management**: Prevent hanging connections
* **Retry Logic**: Intelligent retry strategies
* **Resource Cleanup**: Proper connection disposal

### **2. Caching Strategy**

* **Tool Cache**: Cache available tools to reduce discovery overhead
* **Response Cache**: Cache common tool responses
* **Authentication Cache**: Temporary token validation caching
* **Error Cache**: Cache error patterns for faster recovery

### **3. Resource Optimization**

* **Memory Management**: Efficient state handling
* **CPU Optimization**: Optimize parsing và processing
* **Network Efficiency**: Minimize redundant network calls
* **Async Processing**: Non-blocking execution patterns

## 🔒 Security Implementation

### **1. Token Security**

def secure\_token\_handling(self, token: str, tool\_args: Dict) -> Dict:  
 """Secure token injection into tool arguments"""  
 if not token:  
 logger.warning("⚠️ No token provided for secure operation")  
 return tool\_args  
   
 # Validate token format  
 if not self.validate\_token\_format(token):  
 logger.error("❌ Invalid token format")  
 raise ValueError("Invalid authentication token format")  
   
 # Inject token securely  
 secured\_args = tool\_args.copy()  
 secured\_args['token'] = token  
   
 # Log access (without exposing token)  
 logger.info(f"🔑 Token injected for tool operation: {token[:10]}...")  
   
 return secured\_args

### **2. Input Validation**

* **Command Sanitization**: Clean input commands
* **Injection Prevention**: Prevent command injection attacks
* **Parameter Validation**: Validate tool parameters
* **Access Control**: Verify operation permissions

### **3. Audit & Compliance**

* **Operation Logging**: Log all tool executions
* **Access Tracking**: Track device access patterns
* **Security Events**: Monitor security-related events
* **Compliance Reporting**: Generate compliance reports

## 📈 Monitoring & Analytics

### **1. Performance Metrics**

def track\_execution\_metrics(self, tool\_name: str, duration: float, success: bool):  
 """Track tool execution performance"""  
 metrics = {  
 'tool\_name': tool\_name,  
 'execution\_time': duration,  
 'success': success,  
 'timestamp': time.time(),  
 'session\_id': self.session\_id  
 }  
   
 # Log to monitoring system  
 self.metrics\_logger.info(json.dumps(metrics))  
   
 # Update running statistics  
 self.update\_performance\_stats(tool\_name, duration, success)

### **2. Health Monitoring**

* **Tool Availability**: Monitor MCP tool health
* **Response Times**: Track execution performance
* **Error Rates**: Monitor failure patterns
* **Resource Usage**: Track system resource consumption

### **3. Usage Analytics**

* **Tool Usage Patterns**: Analyze tool usage frequency
* **User Behavior**: Understand user interaction patterns
* **Success Rates**: Track operation success rates
* **Optimization Opportunities**: Identify improvement areas

## 🚀 Advanced Capabilities

### **1. Multi-Device Coordination**

def coordinate\_multi\_device\_operation(self, devices: List[str], action: str) -> Dict:  
 """Coordinate operations across multiple devices"""  
 results = []  
   
 # Execute operations trong parallel where safe  
 for device in devices:  
 if self.is\_safe\_parallel\_operation(device, action):  
 result = self.execute\_parallel(device, action)  
 else:  
 result = self.execute\_sequential(device, action)  
   
 results.append({  
 'device': device,  
 'action': action,  
 'result': result,  
 'timestamp': time.time()  
 })  
   
 return {  
 'operation': 'multi\_device\_coordination',  
 'total\_devices': len(devices),  
 'successful': len([r for r in results if r['result']['success']]),  
 'results': results  
 }

### **2. Predictive Execution**

* **Usage Pattern Learning**: Learn từ user behavior
* **Predictive Caching**: Pre-load likely-needed tools
* **Smart Defaults**: Intelligent parameter defaults
* **Context Awareness**: Adapt to environmental context

### **3. Integration Ecosystem**

* **Plugin Architecture**: Support for custom tools
* **API Extensions**: External service integration
* **Webhook Support**: Event-driven integrations
* **Third-Party Services**: Connect với external platforms

*Tool Agent is the execution powerhouse của MAS-Planning system, transforming high-level plans và analyses into concrete device control actions. Với robust MCP integration, intelligent error handling, và comprehensive security features, nó ensures reliable và secure smart home automation execution.*