# Plan Agent - Strategic Planning Engine

## 🎯 Tổng quan

**Plan Agent** là strategic planner của hệ thống MAS-Planning, chịu trách nhiệm tạo ra các kế hoạch smart home automation thông minh và thực thi chúng theo cách có tổ chức. Đây là brain của hệ thống khi nói đến việc planning và orchestration.

## 🏗️ Kiến trúc và Workflow

graph TB  
 subgraph "Plan Agent Architecture"  
 A[User Request] --> B[Router]  
 B --> C{Plan Type}  
   
 C -->|New Planning| D[Priority Plan Generator]  
 C -->|Plan Selection| E[Execute Selected Plan]  
   
 D --> F[Security Plan]  
 D --> G[Convenience Plan]   
 D --> H[Energy Plan]  
   
 F --> I[Plan Options Cache]  
 G --> I  
 H --> I  
   
 E --> J[Meta Agent Integration]  
 E --> K[Tool Agent Integration]  
 E --> L[API Integration]  
   
 J --> M[Task Analysis]  
 K --> N[Device Control]  
 L --> O[Status Tracking]  
   
 M --> P[Execution Results]  
 N --> P  
 O --> P  
 end  
   
 style B fill:#e1f5fe  
 style D fill:#f3e5f5  
 style E fill:#e8f5e8  
 style I fill:#fff3e0

## 🎯 Core Responsibilities

### **1. Strategic Planning**

* **Multi-Priority Planning**: Tạo 3 loại plan (Security, Convenience, Energy)
* **Context-Aware Generation**: Xem xét available MCP tools và device capabilities
* **User-Centric Design**: Tailored plans based on user requirements
* **Scalable Architecture**: Support cho complex multi-room scenarios

### **2. Plan Execution Orchestration**

* **Multi-Agent Coordination**: Integrate với Meta Agent và Tool Agent
* **Workflow Management**: Control execution flow và dependencies
* **Error Handling**: Graceful recovery từ execution failures
* **Progress Tracking**: Real-time monitoring của plan execution

### **3. API Integration**

* **Plan Persistence**: Upload plans to external API systems
* **Status Synchronization**: Real-time status updates
* **Task Management**: Individual task tracking và reporting
* **Completion Reporting**: Comprehensive execution summaries

## 🧩 Core Components

### **1. Router System**

def router(self, state: PlanState):  
 """Intelligent routing based on input analysis"""  
 selected\_plan\_id = state.get('selected\_plan\_id')  
 if selected\_plan\_id:  
 return {\*\*state, 'plan\_type': 'execute'}  
   
 # Check if message indicates plan selection  
 input\_msg = state.get('input', '').strip().lower()  
 if input\_msg in ['plan 1', 'plan 2', 'plan 3', '1', '2', '3']:  
 # Extract plan number và route to execution  
 plan\_id = self.extract\_plan\_number(input\_msg)  
 return {\*\*state, 'plan\_type': 'execute', 'selected\_plan\_id': plan\_id}  
   
 # Default to priority planning  
 return {\*\*state, 'plan\_type': 'priority'}

**Routing Logic:**

* **New Planning**: Khi user request planning mới
* **Plan Selection**: Khi user chọn từ existing options
* **Plan Execution**: Khi có selected\_plan\_id
* **Context Detection**: Smart routing based on conversation context

### **2. Priority Plan Generator**

def priority\_plan(self, state: PlanState):  
 """Generate 3 priority-based plans"""  
 # Get available MCP tools  
 available\_tools = []  
 if self.tools:  
 available\_tools = [f"- {tool.name}: {tool.description}" for tool in self.tools]  
   
 tools\_info = "\n".join(available\_tools)  
   
 # Use structured prompt với MCP tools information  
 system\_prompt = PLAN\_PROMPTS.replace("<<Tools\_info>>", tools\_info)  
   
 messages = convert\_messages\_list([  
 SystemMessage(system\_prompt),  
 HumanMessage(f"User request: {state.get('input')}")  
 ])  
   
 # LLM generation  
 llm\_response = self.llm.invoke(messages)  
 plan\_data = extract\_priority\_plans(llm\_response.content)  
   
 return {  
 \*\*state,   
 'plan\_options': {  
 'security\_plan': plan\_data.get('Security\_Plan', []),  
 'convenience\_plan': plan\_data.get('Convenience\_Plan', []),  
 'energy\_plan': plan\_data.get('Energy\_Plan', [])  
 },   
 'needs\_user\_selection': True  
 }

**Plan Categories:**

* **Security Plan**: Focus trên home security và protection
* **Convenience Plan**: Emphasize user comfort và ease of use
* **Energy Plan**: Optimize cho energy efficiency và sustainability

### **3. Plan Execution Engine**

def execute\_selected\_plan(self, state: PlanState):  
 """Execute selected plan với full workflow"""  
 selected\_plan\_id = state.get('selected\_plan\_id')  
 plan\_options = state.get('plan\_options', {})  
   
 # Select appropriate plan  
 if selected\_plan\_id == 1:  
 selected\_plan = plan\_options['security\_plan']  
 plan\_type = 'Security Priority Plan'  
 elif selected\_plan\_id == 2:  
 selected\_plan = plan\_options['convenience\_plan']  
 plan\_type = 'Convenience Priority Plan'  
 elif selected\_plan\_id == 3:  
 selected\_plan = plan\_options['energy\_plan']  
 plan\_type = 'Energy Efficiency Priority Plan'  
   
 # Initialize sub-agents  
 self.init\_sub\_agents()  
   
 # Execute plan through MetaAgent + ToolAgent workflow  
 execution\_results = []  
 for i, task in enumerate(selected\_plan, 1):  
 # MetaAgent analysis  
 meta\_result = self.meta\_agent.invoke({  
 "input": task,  
 "context": f"Task {i} of {len(selected\_plan)} from {plan\_type}",  
 "previous\_results": execution\_results[-3:]  
 })  
   
 # ToolAgent execution  
 tool\_result = self.tool\_agent.invoke({  
 "input": task,   
 "token": state.get('token', '')  
 })  
   
 # Collect results  
 execution\_results.append({  
 "task\_number": i,  
 "task": task,  
 "meta\_analysis": meta\_result.get('output', ''),  
 "tool\_execution": tool\_result.get('output', ''),  
 "status": "completed" if tool\_result.get('tool\_agent\_result') else "failed"  
 })

**Execution Features:**

* **Sequential Processing**: Tasks được execute theo order
* **Meta Analysis**: Mỗi task được analyze bởi Meta Agent trước
* **Tool Integration**: Actual execution through Tool Agent
* **Progress Tracking**: Real-time status updates
* **Error Recovery**: Graceful handling của failed tasks

### **4. State Management System**

class PlanState(TypedDict):  
 input: str  
 plan\_status: str  
 route: str  
 plan: List[str]  
 plan\_options: Dict[str, List[str]]  
 needs\_user\_selection: bool  
 selected\_plan\_id: Optional[int]  
 token: str  
 output: str  
 execution\_results: List[Dict[str, Any]]

**State Components:**

* **Input Tracking**: User requests và commands
* **Plan Storage**: Generated plans và options
* **Selection State**: User choice tracking
* **Execution State**: Progress và results
* **Authentication**: Token management cho MCP tools

## 🔄 Workflow Patterns

### **1. New Plan Creation Flow**

sequenceDiagram  
 participant U as User  
 participant P as Plan Agent  
 participant L as LLM  
 participant M as MCP Server  
   
 U->>P: "Create smart home plan"  
 P->>M: Get Available Tools  
 M->>P: Tool List  
 P->>L: Generate Plans with Tool Context  
 L->>P: 3 Priority Plans  
 P->>P: Cache Plan Options  
 P->>U: Present 3 Plan Choices

### **2. Plan Selection & Execution Flow**

sequenceDiagram  
 participant U as User  
 participant P as Plan Agent  
 participant MA as Meta Agent  
 participant TA as Tool Agent  
 participant API as External API  
   
 U->>P: "Select plan 2"  
 P->>P: Retrieve Cached Plan  
 P->>API: Upload Plan  
   
 loop For Each Task  
 P->>MA: Analyze Task  
 MA->>P: Analysis Result  
 P->>TA: Execute Task  
 TA->>P: Execution Result  
 P->>API: Update Task Status  
 end  
   
 P->>API: Mark Plan Complete  
 P->>U: Final Report

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

sequenceDiagram  
 participant P as Plan Agent  
 participant TA as Tool Agent  
 participant API as External API  
 participant Log as Logging  
   
 P->>TA: Execute Task  
 TA-->>P: Error Response  
 P->>Log: Log Error Details  
 P->>API: Update Task as Failed  
 P->>P: Continue Next Task  
 P->>API: Update Plan Status

## 🧠 Intelligent Features

### **1. Context-Aware Planning**

def priority\_plan(self, state: PlanState):  
 """Context-aware plan generation"""  
 # Incorporate available MCP tools  
 available\_tools = self.get\_available\_tools()  
   
 # Consider user environment  
 user\_context = self.extract\_user\_context(state.get('input'))  
   
 # Generate plans with context  
 system\_prompt = PLAN\_PROMPTS.format(  
 tools\_info=tools\_info,  
 user\_context=user\_context  
 )

**Context Factors:**

* **Available Tools**: Current MCP tool capabilities
* **Device Status**: Real-time device states
* **User Preferences**: Learned behaviors và settings
* **Environmental Factors**: Time, season, occupancy

### **2. Adaptive Task Sequencing**

def optimize\_task\_sequence(self, tasks: List[str]) -> List[str]:  
 """Optimize task execution order"""  
 # Dependency analysis  
 dependencies = self.analyze\_task\_dependencies(tasks)  
   
 # Priority scoring  
 priorities = self.calculate\_task\_priorities(tasks)  
   
 # Resource optimization  
 optimized\_sequence = self.optimize\_for\_resources(tasks, dependencies, priorities)  
   
 return optimized\_sequence

**Optimization Criteria:**

* **Dependency Resolution**: Prerequisites được execute trước
* **Resource Efficiency**: Minimize conflicts và maximize parallelism
* **User Impact**: Prioritize high-impact tasks
* **Error Resilience**: Robust sequencing cho error recovery

### **3. Dynamic Plan Adjustment**

def adjust\_plan\_execution(self, execution\_state: Dict) -> Dict:  
 """Dynamically adjust plan based on execution results"""  
 completed\_tasks = execution\_state.get('completed', [])  
 failed\_tasks = execution\_state.get('failed', [])  
 remaining\_tasks = execution\_state.get('pending', [])  
   
 # Re-evaluate remaining tasks  
 adjusted\_tasks = self.re\_evaluate\_tasks(remaining\_tasks, failed\_tasks)  
   
 # Alternative task suggestions  
 alternatives = self.suggest\_alternatives(failed\_tasks)  
   
 return {  
 'adjusted\_tasks': adjusted\_tasks,  
 'alternatives': alternatives,  
 'recommendations': self.generate\_recommendations(execution\_state)  
 }

## 🔧 Technical Implementation

### **1. LangGraph StateGraph Integration**

def create\_graph(self):  
 """Create execution graph for Plan Agent"""  
 graph = StateGraph(PlanState)  
   
 # Add nodes  
 graph.add\_node('route', self.router)  
 graph.add\_node('priority', self.priority\_plan)  
 graph.add\_node('execute\_selected', self.execute\_selected\_plan)  
 graph.add\_node('wait\_selection', lambda state: {  
 \*\*state,   
 'output': 'waiting\_for\_selection'  
 })  
   
 # Define edges và conditional routing  
 graph.add\_edge(START, 'route')  
 graph.add\_conditional\_edges('route', self.route\_controller)  
 graph.add\_conditional\_edges('priority', lambda state:   
 'wait\_selection' if state.get('needs\_user\_selection') else 'execute')  
   
 return graph.compile(debug=False)

### **2. Async MCP Integration**

async def init\_async(self):  
 """Initialize async components"""  
 self.tools = await self.get\_mcp\_tools()  
 self.llm = ChatVertexAI(  
 model\_name=self.model,  
 temperature=self.temperature,  
 model\_kwargs={"tools": self.tools},  
 project=env.GOOGLE\_CLOUD\_PROJECT,  
 location=env.GOOGLE\_CLOUD\_LOCATION  
 )  
  
async def get\_mcp\_tools(self):  
 """Get tools from MCP server"""  
 async with MultiServerMCPClient({  
 "mcp-server": {  
 "url": env.MCP\_SERVER\_URL,  
 "transport": "sse",  
 }  
 }) as client:  
 tools = list(client.get\_tools())  
 return tools

### **3. Sub-Agent Integration**

def init\_sub\_agents(self):  
 """Initialize MetaAgent và ToolAgent when needed"""  
 if self.meta\_agent is None:  
 self.meta\_agent = MetaAgent(  
 model=self.model,  
 temperature=self.temperature,  
 verbose=self.verbose  
 )  
   
 if self.tool\_agent is None:  
 self.tool\_agent = ToolAgent(  
 model=self.model,  
 temperature=self.temperature,  
 verbose=self.verbose  
 )  
   
 # Async initialization với proper event loop handling  
 self.initialize\_tool\_agent\_async()

## 📊 Performance Features

### **1. Caching Strategy**

* **Plan Options Cache**: In-memory storage cho quick selections
* **Tool Cache**: MCP tools caching để avoid repeated fetches
* **LLM Response Cache**: Cache cho similar planning requests
* **Session State**: Persistent state across user interactions

### **2. Resource Optimization**

* **Lazy Loading**: Sub-agents chỉ load khi cần
* **Memory Management**: Efficient state cleanup
* **Connection Pooling**: Reuse MCP connections
* **Parallel Processing**: Concurrent task analysis where possible

### **3. Scalability Design**

* **Stateless Core**: Scalable across multiple instances
* **Event-Driven**: Reactive architecture
* **Microservice Ready**: Easy containerization và deployment
* **Load Balancing**: Distribute requests efficiently

## 🔒 Security Features

### **1. Token-Based Authentication**

def execute\_selected\_plan(self, state: PlanState):  
 """Execute với secure token handling"""  
 token = state.get('token', '')  
   
 # Pass token to Tool Agent securely  
 if token:  
 tool\_result = self.tool\_agent.invoke({  
 "input": task,   
 "token": token  
 })  
 else:  
 # Handle missing authentication gracefully  
 tool\_result = self.handle\_missing\_auth(task)

### **2. Input Validation**

* **Plan Validation**: Ensure generated plans are safe
* **Task Sanitization**: Clean task descriptions
* **Device Safety**: Prevent harmful device commands
* **Access Control**: Role-based plan execution permissions

### **3. Error Security**

* **Sanitized Logs**: No sensitive data trong error messages
* **Secure Fallbacks**: Safe defaults when operations fail
* **Audit Trail**: Comprehensive execution logging
* **Privacy Protection**: User data anonymization

## 📈 Monitoring và Analytics

### **1. Execution Metrics**

def execute\_selected\_plan(self, state: PlanState):  
 """Execution với comprehensive metrics"""  
 start\_time = time.time()  
   
 # Track execution progress  
 for i, task in enumerate(selected\_plan, 1):  
 task\_start = time.time()  
   
 # Execute task  
 result = self.execute\_task(task)  
   
 task\_duration = time.time() - task\_start  
   
 # Log metrics  
 self.log\_task\_metrics(task, task\_duration, result)  
   
 total\_duration = time.time() - start\_time  
 self.log\_plan\_metrics(selected\_plan, total\_duration, execution\_results)

### **2. Success Rate Tracking**

* **Plan Success Rates**: Track completion rates by plan type
* **Task Success Patterns**: Identify commonly failing tasks
* **Tool Performance**: Monitor MCP tool reliability
* **User Satisfaction**: Implicit feedback từ user behavior

### **3. Performance Analytics**

* **Response Times**: End-to-end planning performance
* **Resource Usage**: Memory và CPU utilization
* **Cache Hit Rates**: Effectiveness của caching strategies
* **Scalability Metrics**: Performance under load

## 🚀 Advanced Capabilities

### **1. Learning và Adaptation**

* **Pattern Recognition**: Learn từ successful plan patterns
* **User Preference Learning**: Adapt to individual user styles
* **Optimization Learning**: Improve task sequencing over time
* **Failure Analysis**: Learn từ execution failures

### **2. Multi-Modal Planning**

* **Voice Integration**: Voice-activated plan creation
* **Visual Planning**: Image-based room analysis
* **Sensor Integration**: Environmental data-driven planning
* **Predictive Planning**: Anticipate user needs

### **3. Extensibility Framework**

* **Custom Plan Types**: Support cho specialized planning modes
* **Plugin Architecture**: Easy integration của new capabilities
* **Template System**: Reusable plan templates
* **API Extensions**: Custom external integrations

*Plan Agent là strategic heart của MAS-Planning system, biến user intentions thành actionable smart home automation plans. Với sophisticated planning algorithms và robust execution framework, nó đảm bảo users có experience seamless và effective.*