# MCP Integration & External Services

## 🎯 Tổng quan

Hệ thống MAS-Planning tích hợp với nhiều external services và protocols để cung cấp comprehensive smart home automation solution. Document này mô tả chi tiết các integration patterns, security measures, và best practices.

## 🔌 Model Context Protocol (MCP) Integration

### **1. MCP Architecture Overview**

graph TB  
 subgraph "MAS-Planning System"  
 A[Plan Agent] --> B[Tool Agent]  
 B --> C[MCP Client]  
 end  
   
 subgraph "MCP Server Layer"  
 C --> D[MCP Server]  
 D --> E[Tool Registry]  
 D --> F[Authentication]  
 D --> G[Communication Protocol]  
 end  
   
 subgraph "Smart Home Devices"  
 E --> H[Lights Controller]  
 E --> I[AC Controller]  
 E --> J[Security System]  
 E --> K[Sensors Network]  
 E --> L[Audio System]  
 end  
   
 subgraph "External APIs"  
 F --> M[OXII API]  
 F --> N[Device Vendor APIs]  
 F --> O[Cloud Services]  
 end  
   
 style C fill:#e1f5fe  
 style D fill:#f3e5f5  
 style E fill:#e8f5e8  
 style F fill:#fff3e0

### **2. MCP Client Implementation**

from langchain\_mcp\_adapters.client import MultiServerMCPClient  
  
class MCPIntegration:  
 """MCP Integration Manager"""  
   
 def \_\_init\_\_(self):  
 self.client\_config = {  
 "mcp-server": {  
 "url": env.MCP\_SERVER\_URL,  
 "transport": "sse", # Server-Sent Events  
 }  
 }  
 self.tools\_cache = {}  
 self.connection\_pool = {}  
   
 async def get\_tools(self) -> List[Tool]:  
 """Get available tools từ MCP server"""  
 try:  
 async with asyncio.timeout(5):  
 async with MultiServerMCPClient(self.client\_config) as client:  
 tools = list(client.get\_tools())  
 self.cache\_tools(tools)  
 return tools  
 except asyncio.TimeoutError:  
 logger.error("MCP connection timeout")  
 return self.get\_cached\_tools()  
 except Exception as e:  
 logger.error(f"MCP connection failed: {e}")  
 return []  
   
 async def execute\_tool(self, tool\_name: str, args: Dict, token: str = None) -> Any:  
 """Execute MCP tool với authentication"""  
 try:  
 # Inject authentication token  
 if token:  
 args['token'] = token  
   
 async with MultiServerMCPClient(self.client\_config) as client:  
 tools = list(client.get\_tools())  
   
 # Find matching tool  
 target\_tool = None  
 for tool in tools:  
 if tool.name == tool\_name:  
 target\_tool = tool  
 break  
   
 if not target\_tool:  
 raise ValueError(f"Tool {tool\_name} not found")  
   
 # Execute tool  
 result = await target\_tool.ainvoke(args)  
 return result  
   
 except Exception as e:  
 logger.error(f"Tool execution failed: {e}")  
 raise

### **3. Tool Discovery & Registration**

class ToolRegistry:  
 """Dynamic tool discovery và management"""  
   
 def \_\_init\_\_(self):  
 self.tools = {}  
 self.tool\_metadata = {}  
 self.capabilities = {}  
   
 async def discover\_tools(self) -> Dict[str, Tool]:  
 """Discover và catalog available tools"""  
 discovered\_tools = {}  
   
 async with MultiServerMCPClient(self.client\_config) as client:  
 tools = list(client.get\_tools())  
   
 for tool in tools:  
 # Extract tool metadata  
 metadata = self.extract\_tool\_metadata(tool)  
   
 # Categorize tool capabilities  
 capabilities = self.analyze\_tool\_capabilities(tool)  
   
 # Register tool  
 discovered\_tools[tool.name] = tool  
 self.tool\_metadata[tool.name] = metadata  
 self.capabilities[tool.name] = capabilities  
   
 return discovered\_tools  
   
 def extract\_tool\_metadata(self, tool: Tool) -> Dict:  
 """Extract comprehensive tool metadata"""  
 return {  
 'name': getattr(tool, 'name', 'Unknown'),  
 'description': getattr(tool, 'description', ''),  
 'input\_schema': getattr(tool, 'inputSchema', {}),  
 'output\_schema': getattr(tool, 'outputSchema', {}),  
 'parameters': self.extract\_parameters(tool),  
 'required\_auth': self.check\_auth\_requirements(tool),  
 'device\_types': self.identify\_device\_types(tool),  
 'capabilities': self.extract\_capabilities(tool)  
 }  
   
 def analyze\_tool\_capabilities(self, tool: Tool) -> List[str]:  
 """Analyze tool capabilities for smart routing"""  
 capabilities = []  
   
 name = getattr(tool, 'name', '').lower()  
 description = getattr(tool, 'description', '').lower()  
   
 # Device control capabilities  
 if any(keyword in name + description for keyword in [  
 'switch', 'control', 'turn', 'toggle'  
 ]):  
 capabilities.append('device\_control')  
   
 # Information retrieval  
 if any(keyword in name + description for keyword in [  
 'get', 'list', 'status', 'info'  
 ]):  
 capabilities.append('information\_retrieval')  
   
 # Configuration  
 if any(keyword in name + description for keyword in [  
 'set', 'config', 'setup', 'configure'  
 ]):  
 capabilities.append('configuration')  
   
 return capabilities

### **4. Protocol Communication**

class MCPProtocolManager:  
 """Manage MCP protocol communication"""  
   
 def \_\_init\_\_(self):  
 self.transport\_type = "sse" # Server-Sent Events  
 self.connection\_timeout = 30  
 self.retry\_attempts = 3  
 self.backoff\_factor = 2  
   
 async def establish\_connection(self) -> MultiServerMCPClient:  
 """Establish robust MCP connection"""  
 for attempt in range(self.retry\_attempts):  
 try:  
 client = MultiServerMCPClient(self.client\_config)  
 await client.\_\_aenter\_\_()  
   
 # Verify connection health  
 await self.verify\_connection\_health(client)  
   
 return client  
   
 except Exception as e:  
 wait\_time = self.backoff\_factor \*\* attempt  
 logger.warning(f"Connection attempt {attempt + 1} failed: {e}")  
   
 if attempt < self.retry\_attempts - 1:  
 await asyncio.sleep(wait\_time)  
 else:  
 raise ConnectionError(f"Failed to establish MCP connection after {self.retry\_attempts} attempts")  
   
 async def verify\_connection\_health(self, client: MultiServerMCPClient):  
 """Verify MCP connection health"""  
 try:  
 # Test basic functionality  
 tools = list(client.get\_tools())  
 if not tools:  
 logger.warning("No tools available from MCP server")  
 else:  
 logger.info(f"MCP connection healthy: {len(tools)} tools available")  
 except Exception as e:  
 raise ConnectionError(f"MCP connection health check failed: {e}")

## 🔐 Authentication & Security

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

class AuthenticationManager:  
 """Manage authentication across services"""  
   
 def \_\_init\_\_(self):  
 self.token\_validators = {}  
 self.auth\_cache = TTLCache(maxsize=1000, ttl=300) # 5-minute cache  
 self.security\_logger = logging.getLogger('security')  
   
 def validate\_token(self, token: str, service: str = 'default') -> bool:  
 """Validate authentication token"""  
 if not token:  
 return False  
   
 # Check cache first  
 cache\_key = f"{service}:{hash(token)}"  
 if cache\_key in self.auth\_cache:  
 return self.auth\_cache[cache\_key]  
   
 # Validate token format  
 if not self.validate\_token\_format(token):  
 self.security\_logger.warning(f"Invalid token format for {service}")  
 return False  
   
 # Service-specific validation  
 validator = self.token\_validators.get(service, self.default\_validator)  
 is\_valid = validator(token)  
   
 # Cache result  
 self.auth\_cache[cache\_key] = is\_valid  
   
 return is\_valid  
   
 def inject\_authentication(self, args: Dict, token: str) -> Dict:  
 """Securely inject authentication into tool arguments"""  
 if not token:  
 raise AuthenticationError("Authentication token required")  
   
 # Validate token  
 if not self.validate\_token(token):  
 raise AuthenticationError("Invalid authentication token")  
   
 # Create secure copy  
 authenticated\_args = args.copy()  
 authenticated\_args['token'] = token  
   
 # Log access (without exposing token)  
 self.security\_logger.info(f"Authentication injected: {token[:8]}...")  
   
 return authenticated\_args  
   
 def validate\_token\_format(self, token: str) -> bool:  
 """Validate token format và structure"""  
 if not isinstance(token, str):  
 return False  
   
 if len(token) < 10: # Minimum token length  
 return False  
   
 # Add more validation logic as needed  
 return True

### **2. Secure Communication**

class SecureCommunication:  
 """Secure communication manager"""  
   
 def \_\_init\_\_(self):  
 self.encryption\_enabled = True  
 self.verify\_ssl = True  
 self.timeout = 30  
   
 async def secure\_request(self, url: str, data: Dict, headers: Dict = None) -> Dict:  
 """Make secure HTTP request"""  
 import aiohttp  
 import ssl  
   
 # Configure SSL context  
 ssl\_context = ssl.create\_default\_context()  
 if not self.verify\_ssl:  
 ssl\_context.check\_hostname = False  
 ssl\_context.verify\_mode = ssl.CERT\_NONE  
   
 # Default headers  
 default\_headers = {  
 'Content-Type': 'application/json',  
 'User-Agent': 'MAS-Planning-System/1.0'  
 }  
   
 if headers:  
 default\_headers.update(headers)  
   
 try:  
 async with aiohttp.ClientSession(  
 timeout=aiohttp.ClientTimeout(total=self.timeout),  
 connector=aiohttp.TCPConnector(ssl=ssl\_context)  
 ) as session:  
 async with session.post(url, json=data, headers=default\_headers) as response:  
 response.raise\_for\_status()  
 return await response.json()  
   
 except aiohttp.ClientError as e:  
 logger.error(f"Secure request failed: {e}")  
 raise  
 except Exception as e:  
 logger.error(f"Unexpected error trong secure request: {e}")  
 raise

## 🌐 External API Integration

### **1. OXII API Integration**

class OXIIAPIClient:  
 """OXII Smart Home API Client"""  
   
 def \_\_init\_\_(self):  
 self.base\_url = env.OXII\_ROOT\_API\_URL  
 self.session = None  
 self.rate\_limiter = RateLimiter(requests\_per\_minute=60)  
   
 async def authenticate(self, phone: str, password: str, country: str = "VI") -> str:  
 """Authenticate và get access token"""  
 url = f"{self.base\_url}/api/app/user/signin"  
   
 payload = {  
 "phone": phone,  
 "password": password,  
 "country": country  
 }  
   
 headers = {  
 'Content-Type': 'application/json',  
 'X-Origin': 'smarthiz'  
 }  
   
 try:  
 async with self.rate\_limiter:  
 response = await self.secure\_request(url, payload, headers)  
   
 if response.get('code') == 200:  
 token = response['data']['token']  
 logger.info("OXII authentication successful")  
 return token  
 else:  
 error\_msg = response.get('message', 'Authentication failed')  
 raise AuthenticationError(f"OXII auth failed: {error\_msg}")  
   
 except Exception as e:  
 logger.error(f"OXII authentication error: {e}")  
 raise  
   
 async def get\_device\_list(self, token: str) -> List[Dict]:  
 """Get user's device list"""  
 url = f"{self.base\_url}/api/app/device/list"  
   
 headers = {  
 'Authorization': f'Bearer {token}',  
 'Content-Type': 'application/json'  
 }  
   
 try:  
 async with self.rate\_limiter:  
 response = await self.secure\_request(url, {}, headers)  
 return response.get('data', [])  
   
 except Exception as e:  
 logger.error(f"Failed to get device list: {e}")  
 raise  
   
 async def control\_device(self, token: str, device\_id: str, action: str, parameters: Dict = None) -> Dict:  
 """Control smart home device"""  
 url = f"{self.base\_url}/api/app/device/control"  
   
 payload = {  
 'device\_id': device\_id,  
 'action': action,  
 'parameters': parameters or {}  
 }  
   
 headers = {  
 'Authorization': f'Bearer {token}',  
 'Content-Type': 'application/json'  
 }  
   
 try:  
 async with self.rate\_limiter:  
 response = await self.secure\_request(url, payload, headers)  
 return response  
   
 except Exception as e:  
 logger.error(f"Device control failed: {e}")  
 raise

### **2. Plan Management API**

class PlanAPIClient:  
 """External Plan Management API Client"""  
   
 def \_\_init\_\_(self):  
 self.base\_url = env.PLAN\_API\_BASE\_URL  
 self.api\_key = env.PLAN\_API\_KEY  
 self.default\_headers = {  
 'Authorization': f'Bearer {self.api\_key}',  
 'Content-Type': 'application/json'  
 }  
   
 async def create\_plan(self, plan\_data: Dict) -> Dict:  
 """Create new plan trong external system"""  
 url = f"{self.base\_url}/api/plans"  
   
 # Prepare plan data  
 formatted\_data = {  
 'title': plan\_data.get('plan\_type', 'Smart Home Plan'),  
 'description': plan\_data.get('input', ''),  
 'tasks': plan\_data.get('current\_plan', []),  
 'status': 'created',  
 'created\_at': datetime.utcnow().isoformat(),  
 'metadata': {  
 'source': 'MAS-Planning',  
 'agent\_type': 'PlanAgent',  
 'version': '1.0'  
 }  
 }  
   
 try:  
 response = await self.secure\_request(url, formatted\_data, self.default\_headers)  
 logger.info(f"Plan created successfully: {response.get('id')}")  
 return response  
   
 except Exception as e:  
 logger.error(f"Failed to create plan: {e}")  
 raise  
   
 async def update\_plan\_status(self, plan\_id: str, status: str, details: str = None) -> Dict:  
 """Update plan execution status"""  
 url = f"{self.base\_url}/api/plans/{plan\_id}/status"  
   
 payload = {  
 'status': status,  
 'details': details,  
 'updated\_at': datetime.utcnow().isoformat()  
 }  
   
 try:  
 response = await self.secure\_request(url, payload, self.default\_headers)  
 logger.info(f"Plan status updated: {plan\_id} -> {status}")  
 return response  
   
 except Exception as e:  
 logger.error(f"Failed to update plan status: {e}")  
 raise  
   
 async def update\_task\_status(self, plan\_id: str, task\_id: str, status: str, result: str = None) -> Dict:  
 """Update individual task status"""  
 url = f"{self.base\_url}/api/plans/{plan\_id}/tasks/{task\_id}/status"  
   
 payload = {  
 'status': status,  
 'result': result,  
 'updated\_at': datetime.utcnow().isoformat()  
 }  
   
 try:  
 response = await self.secure\_request(url, payload, self.default\_headers)  
 logger.info(f"Task status updated: {task\_id} -> {status}")  
 return response  
   
 except Exception as e:  
 logger.error(f"Failed to update task status: {e}")  
 raise

### **3. Google Cloud Vertex AI Integration**

class VertexAIIntegration:  
 """Google Cloud Vertex AI Integration Manager"""  
   
 def \_\_init\_\_(self):  
 self.project = env.GOOGLE\_CLOUD\_PROJECT  
 self.location = env.GOOGLE\_CLOUD\_LOCATION  
 self.credentials\_path = env.GOOGLE\_APPLICATION\_CREDENTIALS  
 self.model\_cache = {}  
   
 def initialize\_llm(self, model\_name: str = "gemini-2.5-pro", temperature: float = 0.2, tools: List = None) -> ChatVertexAI:  
 """Initialize Vertex AI LLM với tools"""  
 cache\_key = f"{model\_name}\_{temperature}\_{len(tools) if tools else 0}"  
   
 if cache\_key in self.model\_cache:  
 return self.model\_cache[cache\_key]  
   
 try:  
 # Set up authentication  
 os.environ["GOOGLE\_APPLICATION\_CREDENTIALS"] = self.credentials\_path  
 os.environ["GOOGLE\_CLOUD\_PROJECT"] = self.project  
   
 # Initialize LLM  
 llm\_config = {  
 'model\_name': model\_name,  
 'temperature': temperature,  
 'project': self.project,  
 'location': self.location  
 }  
   
 if tools:  
 llm\_config['model\_kwargs'] = {'tools': tools}  
   
 llm = ChatVertexAI(\*\*llm\_config)  
   
 # Cache initialized LLM  
 self.model\_cache[cache\_key] = llm  
   
 logger.info(f"Vertex AI LLM initialized: {model\_name}")  
 return llm  
   
 except Exception as e:  
 logger.error(f"Failed to initialize Vertex AI LLM: {e}")  
 raise  
   
 def validate\_credentials(self) -> bool:  
 """Validate Google Cloud credentials"""  
 try:  
 if not os.path.exists(self.credentials\_path):  
 logger.error(f"Credentials file not found: {self.credentials\_path}")  
 return False  
   
 # Test basic API access  
 test\_llm = ChatVertexAI(  
 model\_name="gemini-2.5-pro",  
 project=self.project,  
 location=self.location  
 )  
   
 # Simple test call  
 test\_response = test\_llm.invoke("Hello")  
 logger.info("Vertex AI credentials validated successfully")  
 return True  
   
 except Exception as e:  
 logger.error(f"Vertex AI credentials validation failed: {e}")  
 return False

## 📊 Data Storage & Caching

### **1. Redis Integration**

from template.agent.histories import RedisSupportChatHistory  
  
class RedisManager:  
 """Redis integration manager"""  
   
 def \_\_init\_\_(self):  
 self.host = env.REDIS\_HOST  
 self.port = env.REDIS\_PORT  
 self.db = env.REDIS\_DB  
 self.ttl = env.TTL\_SECONDS  
 self.connection\_pool = None  
   
 def get\_connection\_pool(self):  
 """Get Redis connection pool"""  
 if not self.connection\_pool:  
 import redis  
 self.connection\_pool = redis.ConnectionPool(  
 host=self.host,  
 port=self.port,  
 db=self.db,  
 decode\_responses=True,  
 max\_connections=20  
 )  
 return self.connection\_pool  
   
 def get\_chat\_history(self, session\_id: str, conversation\_id: str) -> RedisSupportChatHistory:  
 """Get chat history instance"""  
 return RedisSupportChatHistory(  
 session\_id=session\_id,  
 conversation\_id=conversation\_id,  
 ttl=self.ttl  
 )  
   
 def cache\_plan\_options(self, session\_key: str, plan\_options: Dict, ttl: int = None):  
 """Cache plan options for user selection"""  
 import redis  
 import json  
   
 r = redis.Redis(connection\_pool=self.get\_connection\_pool())  
   
 cache\_data = {  
 'plan\_options': plan\_options,  
 'timestamp': time.time(),  
 'session\_key': session\_key  
 }  
   
 r.setex(  
 f"plan\_options:{session\_key}",  
 ttl or self.ttl,  
 json.dumps(cache\_data)  
 )  
   
 logger.info(f"Plan options cached for session: {session\_key}")  
   
 def get\_cached\_plan\_options(self, session\_key: str) -> Dict:  
 """Retrieve cached plan options"""  
 import redis  
 import json  
   
 r = redis.Redis(connection\_pool=self.get\_connection\_pool())  
   
 cached\_data = r.get(f"plan\_options:{session\_key}")  
 if cached\_data:  
 try:  
 data = json.loads(cached\_data)  
 return data.get('plan\_options', {})  
 except json.JSONDecodeError:  
 logger.error(f"Failed to decode cached plan options: {session\_key}")  
   
 return {}

### **2. Session Management**

class SessionManager:  
 """Comprehensive session management"""  
   
 def \_\_init\_\_(self):  
 self.redis\_manager = RedisManager()  
 self.session\_cache = TTLCache(maxsize=1000, ttl=3600)  
 self.active\_sessions = {}  
   
 def create\_session(self, session\_id: str, conversation\_id: str, user\_data: Dict = None) -> Dict:  
 """Create new session với metadata"""  
 session\_data = {  
 'session\_id': session\_id,  
 'conversation\_id': conversation\_id,  
 'created\_at': datetime.utcnow().isoformat(),  
 'last\_activity': datetime.utcnow().isoformat(),  
 'user\_data': user\_data or {},  
 'plan\_cache': {},  
 'agent\_states': {},  
 'interaction\_count': 0  
 }  
   
 # Store trong Redis  
 session\_key = f"session:{session\_id}:{conversation\_id}"  
 self.redis\_manager.cache\_session\_data(session\_key, session\_data)  
   
 # Cache locally  
 self.session\_cache[session\_key] = session\_data  
 self.active\_sessions[session\_id] = session\_data  
   
 logger.info(f"Session created: {session\_id}")  
 return session\_data  
   
 def update\_session\_activity(self, session\_id: str, conversation\_id: str):  
 """Update session last activity"""  
 session\_key = f"session:{session\_id}:{conversation\_id}"  
   
 if session\_key in self.session\_cache:  
 self.session\_cache[session\_key]['last\_activity'] = datetime.utcnow().isoformat()  
 self.session\_cache[session\_key]['interaction\_count'] += 1  
   
 # Update trong Redis  
 self.redis\_manager.update\_session\_activity(session\_key)  
   
 def get\_session\_data(self, session\_id: str, conversation\_id: str) -> Dict:  
 """Retrieve session data"""  
 session\_key = f"session:{session\_id}:{conversation\_id}"  
   
 # Check local cache first  
 if session\_key in self.session\_cache:  
 return self.session\_cache[session\_key]  
   
 # Fallback to Redis  
 session\_data = self.redis\_manager.get\_session\_data(session\_key)  
 if session\_data:  
 self.session\_cache[session\_key] = session\_data  
 return session\_data  
   
 # Create new session if not found  
 return self.create\_session(session\_id, conversation\_id)

## 🔄 Integration Patterns

### **1. Retry & Circuit Breaker**

import asyncio  
from enum import Enum  
  
class CircuitState(Enum):  
 CLOSED = "closed"  
 OPEN = "open"  
 HALF\_OPEN = "half\_open"  
  
class CircuitBreaker:  
 """Circuit breaker pattern for external service calls"""  
   
 def \_\_init\_\_(self, failure\_threshold: int = 5, timeout: int = 60):  
 self.failure\_threshold = failure\_threshold  
 self.timeout = timeout  
 self.failure\_count = 0  
 self.last\_failure\_time = 0  
 self.state = CircuitState.CLOSED  
   
 async def call(self, func, \*args, \*\*kwargs):  
 """Execute function với circuit breaker protection"""  
 if self.state == CircuitState.OPEN:  
 if time.time() - self.last\_failure\_time > self.timeout:  
 self.state = CircuitState.HALF\_OPEN  
 logger.info("Circuit breaker moving to HALF\_OPEN state")  
 else:  
 raise Exception("Circuit breaker is OPEN")  
   
 try:  
 result = await func(\*args, \*\*kwargs)  
   
 # Success - reset failure count  
 if self.state == CircuitState.HALF\_OPEN:  
 self.state = CircuitState.CLOSED  
 logger.info("Circuit breaker moving to CLOSED state")  
   
 self.failure\_count = 0  
 return result  
   
 except Exception as e:  
 self.failure\_count += 1  
 self.last\_failure\_time = time.time()  
   
 if self.failure\_count >= self.failure\_threshold:  
 self.state = CircuitState.OPEN  
 logger.warning("Circuit breaker moving to OPEN state")  
   
 raise  
  
class RetryManager:  
 """Exponential backoff retry manager"""  
   
 def \_\_init\_\_(self, max\_retries: int = 3, base\_delay: float = 1.0, max\_delay: float = 60.0):  
 self.max\_retries = max\_retries  
 self.base\_delay = base\_delay  
 self.max\_delay = max\_delay  
   
 async def retry\_with\_backoff(self, func, \*args, \*\*kwargs):  
 """Retry function với exponential backoff"""  
 last\_exception = None  
   
 for attempt in range(self.max\_retries + 1):  
 try:  
 return await func(\*args, \*\*kwargs)  
 except Exception as e:  
 last\_exception = e  
   
 if attempt == self.max\_retries:  
 logger.error(f"All retry attempts failed: {e}")  
 raise  
   
 delay = min(self.base\_delay \* (2 \*\* attempt), self.max\_delay)  
 jitter = random.uniform(0, 0.1) \* delay # Add jitter  
 total\_delay = delay + jitter  
   
 logger.warning(f"Attempt {attempt + 1} failed, retrying trong {total\_delay:.2f}s: {e}")  
 await asyncio.sleep(total\_delay)  
   
 raise last\_exception

### **2. Health Check System**

class HealthCheckManager:  
 """Comprehensive health checking for all integrations"""  
   
 def \_\_init\_\_(self):  
 self.health\_checks = {}  
 self.last\_check\_time = {}  
 self.check\_interval = 60 # seconds  
 self.circuit\_breakers = {}  
   
 def register\_health\_check(self, service\_name: str, check\_func, critical: bool = True):  
 """Register health check function"""  
 self.health\_checks[service\_name] = {  
 'check\_func': check\_func,  
 'critical': critical,  
 'last\_status': None,  
 'last\_error': None  
 }  
   
 # Initialize circuit breaker  
 self.circuit\_breakers[service\_name] = CircuitBreaker()  
   
 async def check\_service\_health(self, service\_name: str) -> Dict:  
 """Check health của specific service"""  
 if service\_name not in self.health\_checks:  
 return {'status': 'unknown', 'error': 'Service not registered'}  
   
 check\_config = self.health\_checks[service\_name]  
 circuit\_breaker = self.circuit\_breakers[service\_name]  
   
 try:  
 result = await circuit\_breaker.call(check\_config['check\_func'])  
   
 health\_status = {  
 'service': service\_name,  
 'status': 'healthy',  
 'timestamp': datetime.utcnow().isoformat(),  
 'critical': check\_config['critical'],  
 'details': result  
 }  
   
 check\_config['last\_status'] = 'healthy'  
 check\_config['last\_error'] = None  
   
 return health\_status  
   
 except Exception as e:  
 error\_msg = str(e)  
   
 health\_status = {  
 'service': service\_name,  
 'status': 'unhealthy',  
 'timestamp': datetime.utcnow().isoformat(),  
 'critical': check\_config['critical'],  
 'error': error\_msg  
 }  
   
 check\_config['last\_status'] = 'unhealthy'  
 check\_config['last\_error'] = error\_msg  
   
 return health\_status  
   
 async def check\_all\_services(self) -> Dict:  
 """Check health của all registered services"""  
 results = {}  
 overall\_status = 'healthy'  
   
 for service\_name in self.health\_checks:  
 service\_health = await self.check\_service\_health(service\_name)  
 results[service\_name] = service\_health  
   
 # Update overall status  
 if service\_health['status'] == 'unhealthy' and service\_health['critical']:  
 overall\_status = 'unhealthy'  
   
 return {  
 'overall\_status': overall\_status,  
 'timestamp': datetime.utcnow().isoformat(),  
 'services': results  
 }  
  
# Health check implementations  
async def check\_mcp\_server\_health():  
 """Health check for MCP server"""  
 try:  
 async with MultiServerMCPClient({  
 "mcp-server": {"url": env.MCP\_SERVER\_URL, "transport": "sse"}  
 }) as client:  
 tools = list(client.get\_tools())  
 return {'tools\_count': len(tools), 'status': 'available'}  
 except Exception as e:  
 raise Exception(f"MCP server unavailable: {e}")  
  
async def check\_vertex\_ai\_health():  
 """Health check for Vertex AI"""  
 try:  
 llm = ChatVertexAI(  
 model\_name="gemini-2.5-pro",  
 project=env.GOOGLE\_CLOUD\_PROJECT,  
 location=env.GOOGLE\_CLOUD\_LOCATION  
 )  
 response = await llm.ainvoke("Health check")  
 return {'response\_received': True, 'model': 'gemini-2.5-pro'}  
 except Exception as e:  
 raise Exception(f"Vertex AI unavailable: {e}")  
  
async def check\_redis\_health():  
 """Health check for Redis"""  
 try:  
 import redis  
 r = redis.Redis(host=env.REDIS\_HOST, port=env.REDIS\_PORT, db=env.REDIS\_DB)  
 r.ping()  
 return {'ping\_successful': True, 'host': env.REDIS\_HOST}  
 except Exception as e:  
 raise Exception(f"Redis unavailable: {e}")

*Document này cung cấp comprehensive overview về tất cả external integrations trong MAS-Planning system. Các integration patterns này đảm bảo reliable, secure, và efficient communication với external services while maintaining system resilience và performance.*